THE 12TH INTERNATIONAL SOFTWARE

Quality Week 1999

San Jose, California
24 - 28 May, 1999

Facing the Future

Conference
Tutorials
Expo
Sponsors

Sponsored by

Software Research, Inc.

Conferences

Keynote Presentations
Birds-Of-A-Feather
Technology Track
Vendor Technical Track
Applications Track
Extra Presentations
Tools & Solutions Track
Exhibitors Informantion
Management Track
Authors Index
Quick-Start
Tutorial Sessions
Sponsors

Expo

Exhibitors Information

Vendor Technical Track

Wednesday, 26 May, 1999
Thursday, 27 May, 1999

Keynote Presentations

Plenary Session

Conference Welcome	Dr. Edward Miller, *Software Research* Conference Overview
Plenary Keynote #1 (DAY 1)	Dr. Martin Pol, *IQUIP Informatica* Facing Test maturity
Plenary Keynote #2 (DAY 1)	Mr. Jeff Schuster, *Rational* E-Commerce Quality and YOU!
Plenary Keynote #3 (DAY 2)	Mr. Cam Kaner, *Independent Consultant* Facing the Future: The Law
Plenary Keynote #4 (DAY 2)	Mr. Roger Sherman, *Consultant* Commercial Product Testing
Plenary Keynote #5 (DAY 3)	Dr. Jakob Nielsen, *Nielsen Norman Group* Usability Aspects of Quality
Plenary Keynote #6 (DAY 3)	Mr. Brian Marick, *Reliable Software Technologies* Trapped by Models
Plenary Keynote #7 (DAY 3)	Dr. Boris Beizer, *Independent Consultant* The Mavin

Technology Track

Conference Day #1 (2T1 - 4T2)

Wednesday, 26 May, 1999

Conference Day #2 (6T1 - 8T2)

Thursday, 27 May, 1999

Conference Day #3 (9T1 - 9T2)

Friday, 28 May, 1999

Application Track

Conference Day #1 (2A1 - 4A2)

Wednesday, 26 May, 1999

Conference Day #2 (6A1 - 8A2)

Thursday, 27 May, 1999

Conference Day #3 (9A1 - 9A2)

Friday, 28 May, 1999

Tools & Solutions Track

Conference Day #1 (2S1 - 4S2)

Wednesday, 26 May, 1999

Conference Day #2 (6S1 - 8S2)

Thursday, 27 May, 1999

Conference Day #3 (9S1 - 9S2)

Friday, 28 May, 1999

Management Track

Conference Day #1 (2M1 - 4M2)

Wednesday, 26 May, 1999

Conference Day #2 (6M1 - 8M2)

Thursday, 27 May, 1999

Conference Day #3 (9M1 - 9M2)

Friday, 28 May, 1999

QuickStart Track

Conference Day #1 (2Q - 4Q)

Wednesday, 26 May, 1999

Conference Day #2 (6Q - 8Q)

Thursday, 27 May, 1999

Conference Day #3 (9Q)

Friday, 28 May, 1999

Birds-of-a-Feather

Conference Day #1 (2B - 4S)

Wednesday, 26 May, 1999

Conference Day #2 (6B1 - 8B2)

Thursday, 27 May, 1999

Conference Day #3 (9B1 - 9B2)

Friday, 28 May, 1999

12th International Software Quality Week

Vendor Technical Track

Wednesday, 26 May, 1999

VT1	8:30	CSVerilog	Alain Kerbrat
VT2	9:15	McCabe & Associates	Mike Smith
VT3	10:30	Q-Labs	Ara Kouchdjian
VT4	11:15	Cyrano	Shari Turney
VT5	12:00	Performance Research	Eugenio Cervetto
VT6	12:45	Compuware	Steve Goodhall
VT7	1:30	Soffront	Manu Das
VT8	2:15	Information Balance	Peter Szirmak
VT9	3:30	Intrinsa	Steven Prothero
VT10	4:15	Interim Technology	Fred Scheck

Thursday, 27 May, 1999

VT11	8:30	Sunpower Computing	Dave Priest
VT12	9:15	Rational Software	TBA
VT13	10:30	ErgoLight	Avi Harel
VT14	11:15	KeyLabs	Eric Bowden
VT15	12:00	Software Emancipation Tech.	Bruce Boes
VT16	12:45	Computer Associates	John Okanishi
VT17	1:30	Testmasters	George Van der Veen
VT18	2:15	Data Dimensions	Gerry Ocampo
VT19	3:30	Software Research, Inc.	Edward Miller
VT20	4:15	International Institute for Software Testing	Magdy Hanna

Technology Track

Conference Day #1 (Wednesday, 26 May, 1999)

2T1	Dr. Selim Aissi & Ms. Wendi Hummel, *Applied Dynamics International* Automating Syntax Testing: The Case of a Real-Time Simulation Tool
2T2	Mr. Kenneth Nagin & Dr. Alan Hartman, *IBM* TCBeans Software Test Tool Kit
3T1	Mr. Brian Miller, *Teradyne* Automated Test Generation for Computer Telephony Systems
3T2	Dr. Bettina Buth, Prof. Jan Peleska & Dr. Hui Shi, *Bremen University* Combining Methods for the Analysis of a Fault-Tolerant System
4T1	Mr. Ira Baxter, Mr. Andrew Yahin, Mr Srinivas Neunuri & Mr. Leonardo Moura, *Semantic Designs* Lowering Maintenance Costs by Code Clone Removal
4T2	Mr. Christopher Agruss, *Autodesk, Inc.* Automating Software Installation Testing

Technology Track

Conference Day #2 (Thursday, 27 May, 1999)

6T1	**Ms. Fan Yang, Mr. Trung Nguyen & Mr. Anant Adiga,** ***Sequent Computer Systems, Inc.*** **A Web-Based System Testing Repository Model**
6T2	**Ms. Frances Medina & Mr. Andrew Van Kraanen,** ***AT&T Operational Technology Center*** **Test Automation of a GUI WEB Based Application: An Experience Developing Reusable Automated Testing**
7T1	**Mr. Patrick Copeland,** ***Microsoft*** **Approaches in Testing Componentization in the Windows CE Operating System**
7T2	**Mr. Sergio Cherskov,** ***Microsoft*** **Testing Windows CE 3.0 Real-Time Kernel**
8T1	**Mr. Steven Toeppe & Mr. Scott Ranville,** ***Ford Motor Company*** **An Automated Inspection Tool for a Graphical Specification and Programming Language**
8T2	**Mr. John Kent,** ***CISS Ltd.*** **Advanced Automated Testing Architectures**

Technology Track

Conference Day #3 (Friday, 28 May, 1999)

9T1	**Mr. Bor-Yuan Tsai, Mr. Simon Stobart, Mr. Norman Parrington & Mr. Ian Mitchell,** ***University of Sunderland*** **A State-Based Testing Approach Providing Data Flow Coverage in Object-Oriented Class Testing**
9T2	**Ms. Martina Marre, Ms. Monica Bobrowski & Mr. Daniel Yankelevich,** ***Universidad de Buenos Aires*** **A Software Engineering View of Data Quality**

Applications Track

Conference Day #1 (Wednesday, 26 May, 1999)

2A1	Mr. Graham Thompson, *InCert Software* Minimizing Testing While Maximizing Failure Detection
2A2	Dr. Frank Ackerman & Ms. Cherie McKinney, *Institute for Zero Defect Software / Adobe Systems* Measuring Fault Density in the Real World
3A1	Mr. Ron Silacci, *Lucent Technologies* A Tester's Top 10 List
3A2	Ms. Lisa Boden & Mr. Jon Hagar, *Lockheed Martin* How to Build a 20 Year Successful Independent Verification and Validation (IV&V) Program for the Next Millennium
4A1	Mr. Steven Rabin, *Interworld Corporation* eCommerce Performance Benchmarking -- Methodology and Criteria
4A2	Mr. Federico Pacquing, Jr., *TechWave Inc.* Usability Testing in E-Commerce Applications

Applications Track

Conference Day #2 (Thursday, 27 May, 1999)

6A1	**Dr. Mei-Hwa Chen & Mr. Ming-Hung H. Kao,** ***SUNY Albany*** **Investigating Test Effectiveness on Object-Oriented Software -- A Case Study**
6A2	**Mr. Glen Xia,** ***Deloitte Consulting*** **An Industrial Case Study of Quantitative Management for Object-Oriented Software Testing**
7A1	**Mr. Leon Slota,** ***Neoglyphics Media Corporation*** **Developing Load and Performance Requirements for Web Sites**
7A2	**Mr. Sam Guckenheimer,** ***Rational Software Corp.*** **Effective Testing for Java-Based Web Software**
8A1	**Mr. Lorenzo Lattanzi & Mr. Mario Musmeci,** ***Alenia Aerospazio*** **Safety Critical S/W Development for a Satellite Based Navigation System**
8A2	**Mr. Matias Vierimaa & Ms Minna Makarainen,** ***VTT Electronics.*** **Improving DSP Software Engineering Processes from the Testing Viewpoint**

Applications Track

Conference Day #3 (Friday, 28 May, 1999)

9A1	**Mr. Jon Hagar,** ***Lockheed Martin*** **Industrial Experiences in Establishing Laboratories and Software Models for Testing Software**
9A2	**Ms. Johanna Rothman,** ***Rothman consulting Group*** **Using Quality to Drive Product Development Processes**

Tools & Solutions Track

Conference Day #1 (Wednesday, 26 May, 1999)

2S1	Mr. Douglas Hoffman, *Software Quality Methods, LLC* Test Automation Architectures: Planning for Test Automation
2S2	Mr. Alan Ark & Ms. Sarah Ackroyd, *Thomson Financial Services* Euro: An Automated Solution to Currency Conversion
3S1	Mr. Ron Silacci, *Lucent Technologies* A Tester's Top 10 List
3S2	Mr. Rob Oshana, *Raytheon Systems Company* An Automated Testing Environment to Support Operational Profiles of Software Intensive Systems
4S1	Dr. Heesun Park, *SAS Institute Inc.* Optimum Level of Test Automation for Multi-Platform Client/Server Software
4S2	Dr. Huey-Der Joseph Chu, *National Defense Management College* Automating Client/Server Testing in the Real World

Tools & Solutions Track

Conference Day #2 (Thursday, 27 May, 1999)

6S1	**Mr. Hanania T. Salzer, *RTS Software Ltd.*** **ATRs (Atomic Requirements) Used Throughout Development Lifecycle**
6S2	**Mr. Larry Apfelbaum & Mr. Steve Meyer, *Teradyne / At&T*** **Use Cases Are Not Requirements**
7S1	**Mr. Jim Williams, *CableData, Inc.*** **Testing for Y2K Compliance: A Case Study**
7S2	**Mr. Gunther Chrobok-Diening, Mr. Andreas Ulrich & Mr. Peter Zimmerer, *Siemens AG*** **Test Architectures for Testing Distributed Systems**
8S1	**Dr. Siegfried Voessner, Mr. Michael O'Sullivan & Mr. Joachim Wegener, *Stanford / Daimler-Chrysler*** **Genetic Algorithm with Cluster Analysis for Software Testing**
8S2	**Mr. Yuri Chernak, *Valley Forge Consulting, Inc.*** **In-Process Validation and Improvement of Test-Case Effectiveness**

Tools & Solutions Track

Conference Day #3 (Friday, 28 May, 1999)

9S1	**Mr. Alain Kerbrat & Mr. Iulian Ober, *Verilog*** **Automated Test Generation from SDL/UML Specifications**
9S2	**Dr. Ingrid B. Ottevanger, *IQUIP*** **Test Factory (TSite): A Next Step in Structured Testing**

Management Track

Conference Day #1 (Wednesday, 26 May, 1999)

2M1	Mr. Roger Records, Mr. Jay Ahlbeck, Ms. Linda Lin Mr. Jon Scharer & Ms. Margaret Stocking, *Boeing Commercial Airplanes* Deploying SQA in Very Small Projects
2M2	Mr. Mike Ross, *Quantitative Software Management* Size Does Matter: Continuous Size Estimating and Tracking
3M1	Mr. John N. Romanak, *Bellcore* Life As A CMM Level 5 Test Organization
3M2	Dr. Ilene Burnstein, Dr. Ariya Homyen, Dr. Taratip Suwannasart, Mr. Robert Grom & Mr. Gary Saxena, *Illinois Institute of Technology* Using the Testing Maturity Model (TMM) to Assess and Improve Your Software Testing Process
4M1	Mr. Nathan Baddoo & Dr. Tracy Hall, *South Bank Univeristy* The Impact of Software Practitioner on Software Quality
4M2	Mr. Michael Deck, *Cleanroom Software Engineering* Process Diversity: How I Stopped Worrying and Learned to Love Chaos

Management Track

Conference Day #2 (Thursday, 27 May, 1999)

6M1	**Mr. Herb Krasner,** ***Krasner Consulting*** **Using the Cost of Quality Approach for Software**
6M2	**Mr. Tom Gilb,** ***Result Planning Limited*** **Risk Management Technology: A Toolkit for Identifying, Documenting, Analyzing and Coping with Project Risk**
7M1	**Mr. Philip Lones,** ***Lucent Technologies*** **Revolutionary? a Development Method That Works**
7M2	**Mr. Nick Borelli,** ***Microsoft Corporation*** **Seizing Control of the Development Lifecycle**
8M1	**Mr. James Bindas,** ***Intel Corporation*** **Tactical Improvement Projects: Real-Life Lessons in Leading Change**
8M2	**Mr. Scott Young,** ***Perot System Corporation*** **Them and Us: Communication Between Development and Test**

Management Track

Conference Day #3 (Friday, 28 May, 1999)

9M1	**Mr. Tom Wissink,** ***Lockheed Martin Mission Systems*** **Test Engineering -- A "Value Add" Career Path**
9M2	**Mr. Keith Stobie,** ***BEA Systems, Inc.*** **Creating a Testing Culture**

QuickStart Track

Conference Day #1 (Wednesday, 26 May, 1999)

2Q	**Mr. Rainer Pirker & Mr. Andreas Rudolf,** ***IBM*** **Millennium is Getting Closer -- The Quickstart to Y2K Testing**
3Q	**Mr. Tom Gilb,** ***Result Planning Limited*** **Evolutionary Project Management ('Evo')**
4Q	**Mr. Bill Deibler,** ***Software Quality Consulting (SSQC)*** **Making the CMM Work: Streamlining the CMM for Small Projects and Organizations**

QuickStart Track

Conference Day #2 (Thursday, 27 May, 1999)

6Q	**Mr. Douglas Hoffman & Dr. Cem Kaner,** ***Software Quality Methods, LLC / Independent Consultant*** **Thoughts on Oracles and Software Test Automation**
7Q	**Mr. Ted Hammer,** ***NASA GSFC SATC*** **Continuous Risk Management at NASA**
8Q	**Dr. Cem Kaner,** ***Independent Consultant*** **Recruiting Software Testers**

QuickStart Track

Conference Day #3 (Friday, 28 May, 1999)

9Q	**Ms. Elfriede Dustin & Mr. John Paul,** ***CSC / Freddie Mac*** **Moving From Conventional Testing to Object-Oriented Testing**

Birds-of-a-Feather

Conference Day #1 (Wednesday, 26 May, 1999)

2B	**Ms. Elisabeth Hendrickson,** ***Aveo, Inc.*** **Mass Market Software Testing**
3B1	**Ms. Peggy Fouts,** ***Compuware Corporation*** **Medical and Safety Critical Application Testing**
3B2	**Mr. Mark Wiley,** ***nCUBE*** **OS and Embedded System Testing Techniques**
4B1	**Mr. Larry Apfelbaum,** ***Teradyne*** **Telecommunications Software**
4B2	**Mr. Jon Hagar,** ***Lockheed Martin*** **Testing for Military and Government Software**

Birds-of-a-Feather

Conference Day #2 (Thursday, 27 May, 1999)

6B	**Ms. Johanna Rothman,** ***Rothman Consulting Group*** **Life as a New Test Manager**
7B1	**Dr. Cem Kaner,** ***Independent Consultant*** **Status Report on US Software Quality Laws**
7B2	**Mr. Richard Denney,** ***Landmark Graphics*** **Blue Collar Formal Methods**
8B1	**Mr. Rodney Wilson,** ***KLA-Tencor*** **Care and Feeding of a Testing Career**
8B2	**Mr. Mark Johnson,** ***OrCAD*** **Productivity in Small Integrated Teams**

Birds-of-a-Feather

Conference Day #3 (Friday, 28 May, 1999)

9B1	**Mr. Mark D. Anderson,** ***Discerning Software Corp.*** **Client/Server Load Testing**
9B2	**Ms. Carla Oexmann,** ***ATI Research*** **Running a Nightly Test**

Extra Presentations

Application Track	Mr. Rex Black, *Rex Black Consulting Services* Charting the Progress of System Development Using Defect Data
Tool & Solutions Track	Mr. J. P. Schroeder, *Teradyne* Model-based Testscript Generation for a C++ Class

Exhibitors Information

AutoTester View ad

AZOR, Inc. View ad

Capital One View ad

Computer Associates International, Inc. View ad

Compuware

Cisco Systems View ad

CS VERILOG View ad

CYRANO View ad

Data Dimensions View ad

ErgoLight View ad

Hall Kinion View ad

Information Balance View ad

Interim Technology View ad

Intrinsa Corporation View ad

KeyLabs, Inc. View ad

McCabe & Associates View ad

MicroCrafts, Inc. View ad

Microsoft Corporation View ad

Performance Research View ad

Quantitative Software Management

Q-Labs, Inc. View ad

Rational Software View ad

Reasoning, Inc. View ad

Soffront Software View ad

Software Development Technologies View ad

Software Emancipation Technology View ad

Software Quality Engineering View ad

Software Research, Inc. View ad

Sunpower Computing View ad

Technology Search Int'l

Teradyne Software & Systems Test View ad

Testmasters View ad

The International Institute for Software Testing View ad

Authors Index

Frank Acherman 2A2
Sara Ackroyd 2S2
Anant Adiga 6T1
Christopher Agruss 4T2
Selim Aissi 2T1
Mark Anderson 9B1
Larry Apfelbaum 4B1
Larry Apfelbaum 6S2
Alan Ark 2S2
Nathan Baddoo 4M1
Ira Baxter 4T1
Boris Beizer F1-F2
Boris Beizer 10P3
William Bently C2
James Bindas 8M1
Robert Binder H1-H2
Karen Bishop-Stone C1
Rex Black ST
Monica Bobrowski 9T2
Lisa Boden 3A2
Bruce Boes VT15
Nick Borelli 7M2
Eric Bowden VT14
Ilene Burnstein 3M2
Bettina Buth 3T2
Hans Buwalda A1-A2
David Carman 3S2
Eugenio Cervetto VT5
Mei-Hwa Chen 6A1
Yuri Chernak 8S2
Sergio Cherskov 7T2
G. Chrobok-Diening7S2
Huey Joseph Chu 4S2
Patrick Copeland 7T1
Manu Das VT7
Michael Deck K1-K2
Michael Deck 4M2
Bill Deibler 4Q
Richard Denney 7B2
Thomas Drake B2
Sally Drew J2
Elfriede Dustin 9Q
Peggy Fouts 3B1
Tom Gilb G2
Tom Gilb 3Q
Tom Gilb 6M2
Steve Goodhall VT6
Robert Grom 3M2
S. Guchenheimer 7A2
Jon Hagar
Jon Hagar 4B2
Jon Hager 9A1
Tracy Hall 4M1
Ted Hammer 7Q
Magdy Hanna G1
Magdy Hanna VT20
Avi Harel VT13
Alan Hartman 2T2
E. Hendrickson 2B
Douglas Hoffman 6Q
Ariya Homyen 3M2
Wend Hummel 2T1
Mark Johnson 8B2
Cam Kaner 5P2
Cam Kaner 7B1
Cem Kaner 6Q
Cem Kaner 8Q
Ming-Hung Kao 6A1
John Kent 8T2
Alain Kerbrat 9S1,vt1
Atte Kinnula 8A2
Edward Kit A1-A2
Ara Kouchakdjian VT3
Herb Krasner 6M1
Lorenzo Lattanzi 8A1
Phillip Lones 7M1
M. Makarainen 8A2
Brian Marick 10P2
Martina Marre 9T2
John McGregor B1
Frances Medina 6T2
Steve Meyer 6S2
Brian Miller 3T1
Edward Miller 1P1
Edward Miller 5P1
Edward Miller 10P4
Edward Miller VT19
Ian Mitchell 9T1
Leonardo Moura 4T1
John Musa D2
Mario Musmeci 8A1
Kenneth Nagin 2T2
Srinivas Neunuri 4T1
Trung Nguyen 6T1
Jakob Nielsen 10P1
Gerry Ocampo VT18
Carla Oexmann 9B2
John Okanishi VT16
Rob Oshana 3S1
Ingrid Ottevanger 9S2
Federico Pacquin 4A2
Heesun Park 4S1
John Paul 9Q
Rainer Pirker 2Q
Norman Parrinton 9T1
Jan Peleska 3T2
Martin Pol 1P2
Dave Priest VT11
Shel Prince VT12
Steven Prothero VT9
Steven Rabin 4A1
Jeff Raksha 9Q
Scott Ranville 8T1
Roger Records 2M1
John Romanak 3M1
Linda RosenbergE1-E2
Mike Ross 2M2
Johanna Rothman 6B
Johanna Rothman 9A2
Andreas Rudolph 2Q
Hanania Salzer 6S1
Gary Saxena 3M2
Fred Scheck VT10
N. Schneidewind D1
J.P. Schroeder ST
Jeff Schuster 1P3
Roger Sherman 5P3
Hui Shi 3T2
Ron Silacci 3A1
Leon Slota 7A1
Mike Smith VT2
Keith Stobie 9M2
T. Suwanasart 3M2
Peter Szirmak VT8
Graham Thompson 2A1
Steven Toeppe 8T1
Bor-Yuan Tsai 9T1
Shari Turney VT4
Andreas Ulrich 7S2
G. van der Veen VT17
Leonard Verhoef J1
Matias Vierimaa 8A2
Siegfried Voessner 8S1
Mark Wiley 3B2
Jom Williams 7S1
Rodney Wilson 8B1
Tom Wissink 9M1
Glen Xia 6A2
Andrew Yahin 4T1
Fan Yang 6T1
Daniel Yankelevich 9T2
Scott Young 8M2
Peter Zimmerer 7S2

Tutorial Sessions

TUTORIAL DAY #1

A 1	Mr. Edward Kit & Mr. Hans Buwalda *DT / CMG*	A 2	Mr. Edward Kit & Mr. Hans Buwalda *SDT / CMG*
B 1	Dr. John McGregor *Clemson University*	B 2	Mr. Thomas Drake *Coastal Research & Technology, Inc.*
C 1	Ms. Karen Bishop-Stone *Testware Associates, Inc.*	C 2	Mr. William Bently *mu_Research*
D 1	Dr. Norman Schneidewind *Naval Postgraduate School*	D 2	Dr. John D. MUsa *Consultant*
E 1	Dr. Linda H. Rosenberg *SATC, NASA*	E 2	Dr. Linda H. Rosenberg *SATC, NASA*

TUTORIAL DAY #2

A 1	Dr. Boris Beizer *Independent Consultants*	A 2	Dr. Boris Beizer *Independent Consultants*
B 1	Dr. Magdy Hanna *Int'l Institute for Software Testing*	B 2	Mr. Tom Gilb *Result Planning Limited*
C 1	Mr. Robert Binder *RBSC Corporation*	C 2	Mr. Robert Binder *RBSC Corporation*
D 1	Mr. Leonard Verhoef *Human Efficiency*	D 2	Ms. Sally Drew *Tescom UK SGT*
E 1	Mr. Michael Deck *Cleanroom Software Engineering Inc.*	E 2	Mr. Michael Deck *Cleanroom Software Engineering Inc.*

The Tutorial Notes and prior Conference Publications are available from Software Research Institute

Click here for an order form

For more information about the Quality Week Conferences series, please visit our web site at: http://www.soft.com/QualWeek/

Sponsors

Software Research, Inc.

Co-Sponsors:

Gold Sponsors:

Rational Software

Software SETT Corp.

Sun Microsystems

Silver Sponsors:

Capital One

Interim Technology

Testmasters

Facing the Future means Facing Test Maturity

by

Martin Pol

IQUIP Informatica B.V., The Netherlands

Agenda

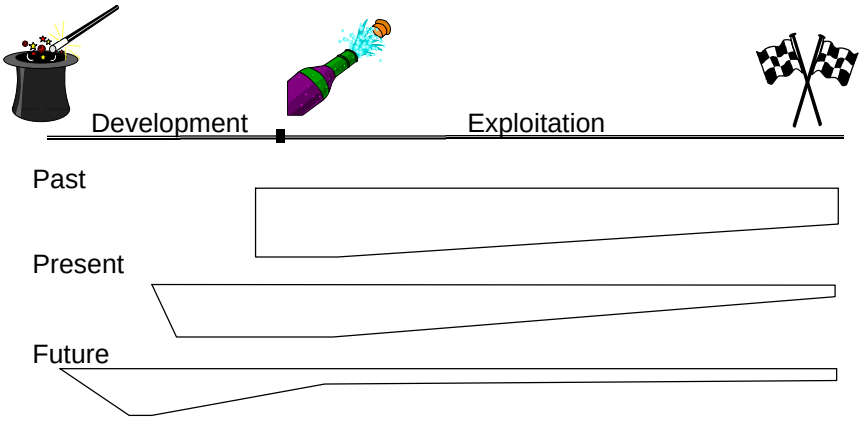
- Facing the Future
- Future Challenges
- Facing Test Maturity

Why keep on Testing ?

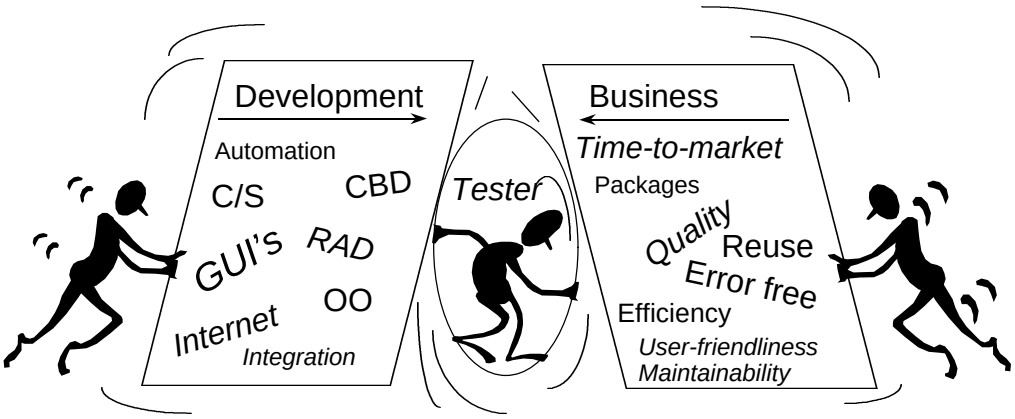
- Maturity of the industry
  - Error free software?
  - Prevention not enough
  - Technogoly push
- Business risks
  - Importance software quality
  - Integration business processes
  - Time-to-market
  - Competition

No Risk, No Test

Testing in the Past, Present and Future



Testing under Pressure



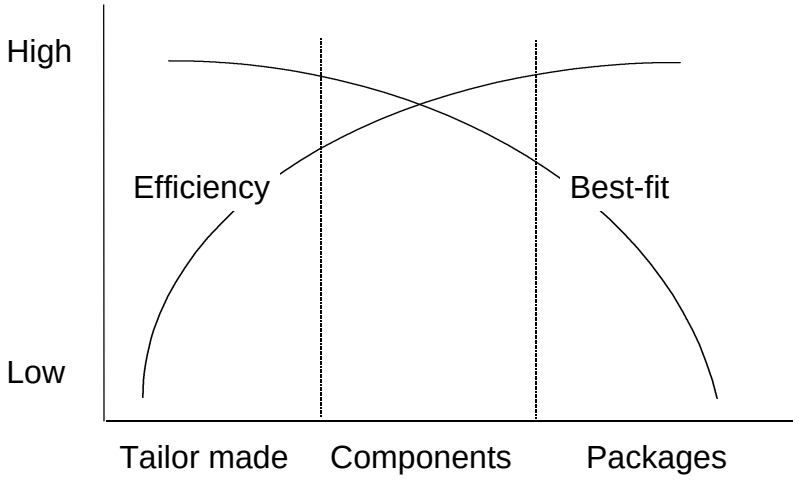
Agenda

- Facing the Future
- Future Challenges
- Facing Test Maturity

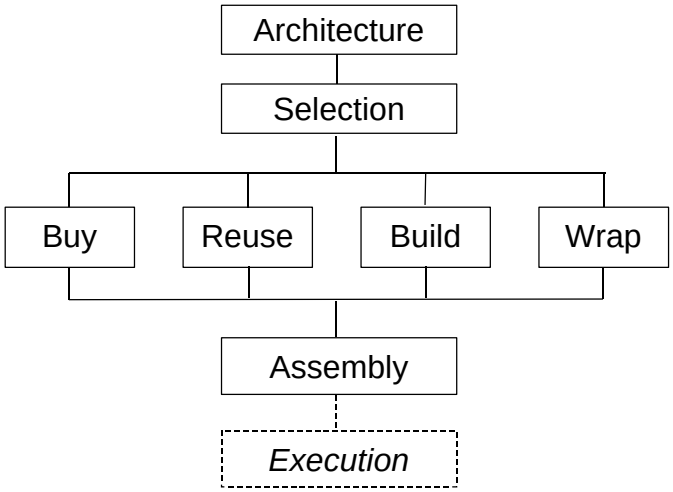
Future Challenges

- Component Based Development
- Growing Complexity
- Suitable Test Methods and Tools
- Adequate Test Process Maturity

Component Based Development What's happening?



The CBD Process



Growing Complexity

back-office
front-office
customer
supplier
employee
government

7 days a week
24 hrs
world wide

hardware
software
network
color
graphics
art
sound
video
TV

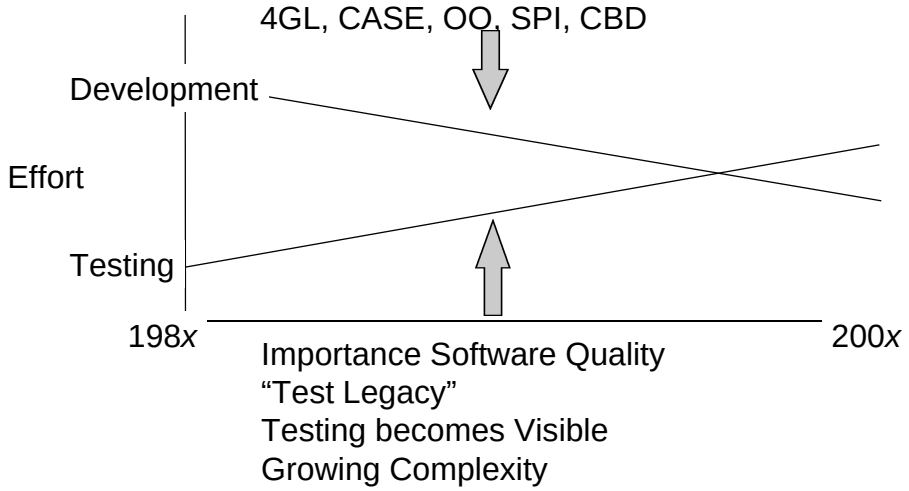
e-commerce
virtual organizations
knowledge mngmnt
data mining
computer & telephone

object orientation
client / server
GUI's
component assembly
internet

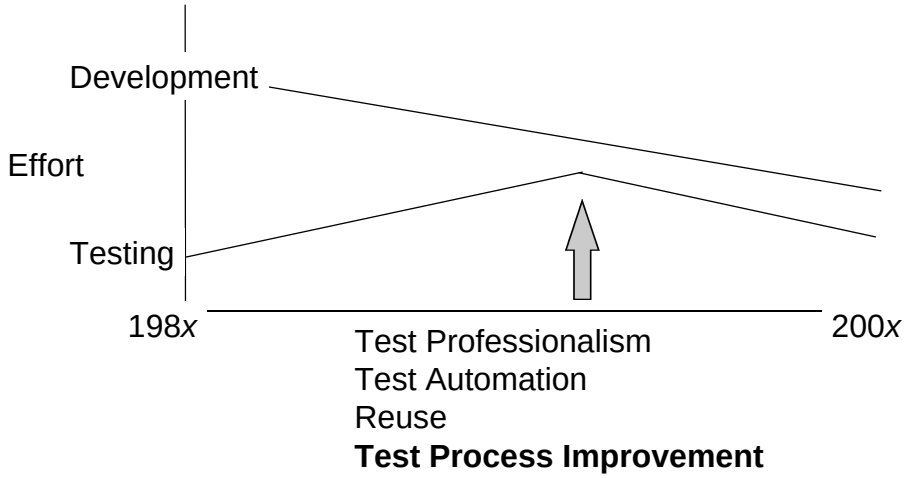
Suitable Test Methods and Tools

- Research & Development
  - Vendors & service suppliers
  - Universities, EC, trendwatchers
  - In company
  - SIGISTs
- Training, conferences & publications
  - Certified staff
- Tester's professionalism
  - Skills and expertise for "whatever" test

Unwanted Situation



Desired Situation



Agenda

- Facing the Future
- Future Challenges
- Facing Test Maturity

What is Test Process Improvement ?

"Continuous improvement of the quality and the efficiency of the testing process, related to the output of the total software process"

- Quality — Insight, Coverage, Control, Timeliness
- Costs — Cheaper
- Lapse time — Faster

General Improvement Life-cycle

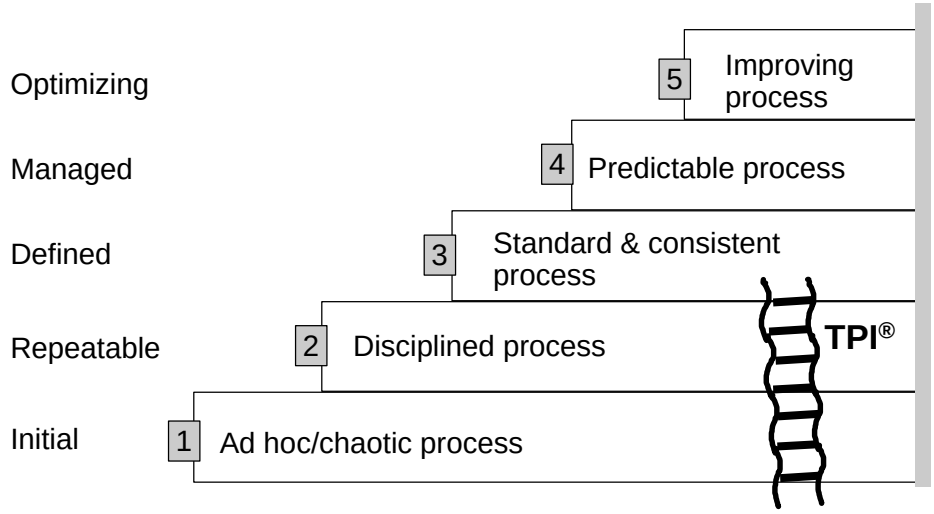
- Goal & scope
- Current situation
- Desired situation
- Implementing changes

Required: reference model

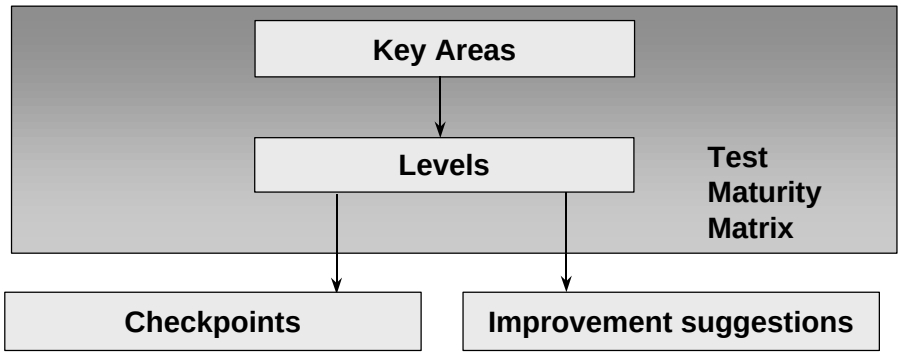
Model Requirements

- Controlled improvement steps
- Practical
- As objective as possible
- Options and priorities
- Highly detailed
- Fast assessment
- Independent

TPI® and CMM



The TPI®-model



Key Areas

- Testing strategy
- Life-cycle model
- Moment of involvement
- Estimating & planning
- Design techniques
- Static test techniques
- Metrics
- Test tools
- Testing environment
- Office environment
- Commitment & motivation
- Testing functions & training
- Scope of methodology
- Communication
- Reporting
- Defect management
- Testware management
- Test process management
- Evaluation
- White-box testing

Test Maturity Matrix

	Key Area / Schale	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	Test strategy		A					B				C		D	
2	Life cycle model		A			B									
3	Moment of involvement			A				B				C		D	
4	Estimating & planning				A							B			
5	Test design techniques		A		B										
6	Static test techniques					A		B							
7	Metrics						A			B			C		D
8	Test tools					A			B			C			
9	Testing environment				A				B						C
10	Office environment				A										
11	Commitment & motivation		A				B						C		
12	Testfunctions & training				A			B			C				
13	Scope of methodology					A						B			C
14	Communication			A		B							C		
15	Reporting		A			B		C					D		
16	Defect management		A				B		C						
17	Testware management			A			B				C				D
18	Test process management		A		B								C		
19	Evaluation							A			B				
20	White-box testing					A		B		C					

Test Maturity Matrix Structure

- Each Key Area has from 1 up to 4 Levels
- Each Level includes prior Level
- Based on priorities and dependencies
- To enable correlation: 13 Scales of maturity

Current Situation - example

	Key Area / schale	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	Test strategy		A					B				C		D	
2	Life-cycle model		A			B									
3	Moment of involvement			A				B				C		D	
4	Estimating & planning				A							B			
5	Test design techniques		A		B										
6	Static test techniques					A		B							
7	Metrics						A			B			C		D
8	Test tools					A			B			C			
9	Testing environment				A				B						C
10	Office environment				A										
11	Commitment & motivation		A				B						C		
12	Testfunctions & training				A			B			C				
13	Scope of methodology					A						B			C
14	Communication			A		B							C		
15	Reporting		A			B		C					D		
16	Defect management		A				B		C						
17	Testware management			A			B				C				D
18	Test process management		A		B								C		
19	Evaluation							A			B				
20	White-box testing					A		B		C					

Desired Situation - example

	Key Area / schale	0	1	2	3	4	5	6	7	8	9	10	11	12	13
1	Test strategy		A					B				C		D	
2	Life cycle model		A			B									
3	Moment of involvement			A				B				C		D	
4	Estimating & planning				A							B			
5	Test design techniques		A		B										
6	Static test techniques					A		B							
7	Metrics						A			B			C		D
8	Test tools					A			B			C			
9	Test environment				A				B						C
10	Office environment				A										
11	Commitment & motivation		A				B						C		
12	Test functions & training				A			B			C				
13	Scope of methodology					A						B			C
14	Communication			A		B							C		
15	Reporting		A			B		C					D		
16	Defect management		A				B		C						
17	Testware management			A			B				C				D
18	Test process management		A		B								C		
19	Evaluation							A			B				
20	White-box testing					A		B		C					

Levels: Test Strategy

Levels:

A) Testing strategy for single test

B) Combined testing strategy for black-box tests

C) Combined strategy for black-box tests plus white-box tests or evaluation

D) Combined strategy for all testing and evaluation activities

Checkpoints: Test Strategy

Level A: Strategy for single test

Checkpoints:

- Conscious risk analysis
- Differentiation in scope and depth of the tests (subsystems, quality attributes), depending on the risks
- > 1 design technique is used, suitable for the desired testing depth
- Re-tests also have a (simple) strategy definition
- Strategy is defined, executed and controlled

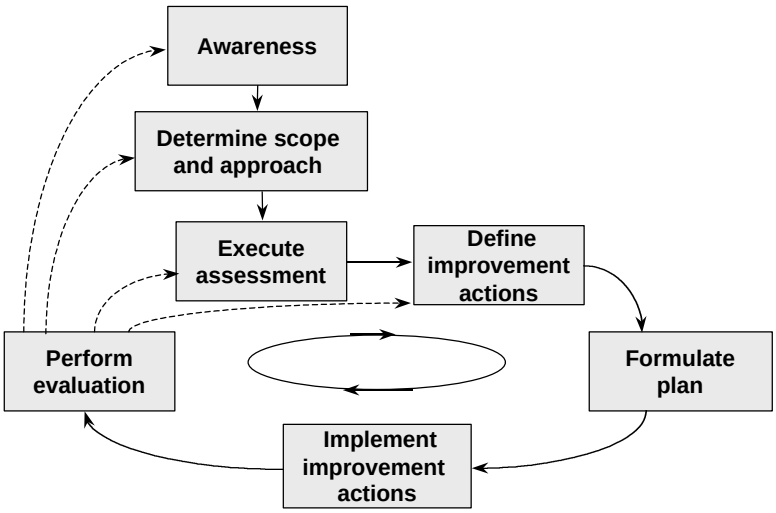
Improvement Suggestions: Test Strategy

Level A: Strategy for single test

Improvement Suggestions (from 0 --> A):

- If only 1 technique is available, make simple variations, which give either more or less test depth
- Define a re-test approach which makes a balance between total or partial testing
- Divide the system under test in testable subsystems: give priorities
- Divide the system under test in testable quality attributes: give priorities
- To shorten the critical path testing-time: incremental testing

Process of Change



Don’ts

- Strictly top-down or strictly bottom-up
- Education only
- One-sided improvement
- Unsuitable pilots
- Black-box tests only
- Underestimation of resistance & culture
- (Too) high expectations

Who ! How ? Can you explain…? Do you really think … Where …

Questions ?

What ??

Information:

Books: **TPI (Addison Wesley)**

Website: **www.iquip.nl/tpi**

Email: **tpi@iquip.nl**

FACING THE FUTURE MEANS FACING TEST MATURITY

Martin Pol
IQUIP Informatica B.V.
Wildenborch 3, 1110 AG Diemen, The Netherlands,
phone: +31 20 660 66 00, fax: +31 20 695 32 98
email: polmarti@iquip.nl,

TPI^{Ò}email: tpi@iquip.nl
TPI^{Ò}-website: www.iquip.nl/tpi (in Dutch, English, German and Spanish)
TPI^{Ò}-book authors: Tim Koomen and Martin Pol
(books available in Dutch, English and German (August 1999)

1. Background

Apart from continuous evolution in the technical infrastructure (e.g. Client/Server, GUI's and the Internet) and in the development infrastructure (e.g. OO, RAD), the challenges for testing lie in the integration of business processes and related innovations such as electronic commerce and computer telephony integration. We will connect customers, suppliers, employees, public services, etc., 24 hours per day, 7 days a week, world-wide. Complexity of the IT-solutions and testing will grow dramatically. The industry will focus on component integration: components of tailor-made software, packages, odd data files and incompatible technical infrastructure have to play together. Reusability and adaptability will be major quality requirements.

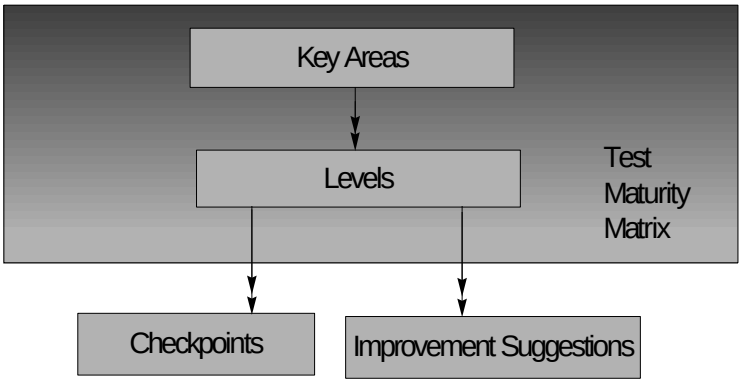
The future requires new test solutions and more mature test processes. Many organisations still struggle with the implementation and consolidation of structured testing. Frequently, day-to-day's pressure causes reinvention of the testing wheel over and over again. Structured, continuous test improvement programmes are required to establish a stable foundation for the challenges to come.

The TPI®-model, which is based on current state-of-the-art test process improvement practices, gives practical guidelines for assessing the maturity level of testing in an organisation and for step by step improvement of the process. The purpose of such improvement could be reaching CMM level 3.

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

2. Description of the TPI® model

The model is visualised as follows:



2.1 Key Areas

In each test process certain areas need specific attention in order to achieve a well defined process. These **Key Areas** are therefore the basis for improving and structuring the test process. The TPI® model has 20 key areas.

The scope of test process improvement usually comprises black-box tests like system and acceptance tests. Most key areas are adjusted to this. However, to improve more "mature" test processes, attention must also be given to verification activities and white-box tests like unit and integration tests. Separate key areas are included in order to give due attention to these processes as well.

A full list of key areas is given below, followed by an explanation.

Test strategy	Test tools	Reporting
Life cycle model	Test environment	Defect management
Moment of involvement	Office environment	Testware management
Estimating & planning	Commitment and motivation	Test process management
Test design techniques	Testing functions and training	Evaluation
Static test techniques	Scope of methodology	White-box testing
Metrics	Communication	

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

Key Area	Description
Test strategy	Test strategy should be aimed at finding the most important defects as early and as cheap as possible. In the test strategy it is determined what (quality)risks are covered by testing. By involving more tests and more detective measures, a better balanced strategy is possible. Unintentional overlaps or gaps between different tests can be prevented by co-ordinating testers and test activities, and by determining thoroughness.
Life cycle model	Within the test process a number of phases are discerned: planning, preparation, design, execution and completion. In each phase several activities are performed. For each activity aspects like goal, input, process, deliverables, dependencies, techniques and tools, facilities and documentation are recorded. The importance of a life cycle model lies in better control of the test process, since the activities can be planned and controlled consistently.
Moment of involvement	Although the actual test execution usually starts after the realisation of the software, the test process should start a lot earlier. Earlier involvement of testing in the system development life cycle helps detecting defects as early and/or as easy as possible, and even helps preventing defects. Better co-ordination between tests is possible and the critical path time of testing can be greatly reduced.
Estimating & planning	Estimating and planning are required in order to define which activities are performed at what moment and how many resources will be needed. Estimating and planning is the basis for reserving capacity and for co-ordinating test activities and project activities.
Test design techniques	A test design technique is defined as 'a standardised approach for deriving test cases from documentation'. Usage of these techniques increases insight in the quality and coverage of tests and leads to higher re-usability of tests. Based on a test strategy, different test design techniques are used in order to produce test coverage of the intended parts of the software to the extent which was agreed upon.
Static test techniques	Not everything can and needs to be tested dynamically, i.e. by running the programmes. The phenomenon of checking products without running the actual programmes or evaluating specified quality measures, is called static testing. Checklists and similar devices are very useful here.
Metrics	Metrics are quantified observations (measurements). For the test process, measuring the progress and the quality of the software under test is very important and so are metrics in these areas. Metrics are used in order to be able to manage the test process, in order to support advice to be given, and also in order to compare different systems or processes. It helps answering questions such as 'Why is it that system A has far less failures in production than system B has, why is it that test process A can be performed faster and more thoroughly than process B can?' In the improvement of the test process, metrics are specifically important for judging the results of certain improvement actions. This is done by measuring before and after the improvement takes place.
Test tools	Automation of the test process can be done in a variety of ways. As a rule, automation serves one of the following goals: - less resource consumption; - less time consumption; - better test coverage; - more flexibility; - more or faster insight in the status of the test process; - better motivation of test staff.
Testing environment	Test execution takes place in a so called test environment. This environment consists of the following components: • hardware; • software; • communication facilities; • facilities for creation and use of data sets; • procedures. The environment should be arranged so that optimal testing is possible. The environment greatly influences the quality, duration and costs of the test process. Important aspects of the environment are responsibilities, control, timely and sufficient availability, flexibility and representativeness of the actual production situation.

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

Office environment	Test personnel need offices, desks, chairs, PC's, word processing facilities, printers, telephones, etc. Good and timely arrangement of the office environment positively influences motivation of testers, and communication and efficiency of (the execution of) test tasks.
Commitment & motivation	Commitment and motivation of the people involved in testing are conditions for a mature test process. People involved are not only members of the test team, but also, amongst others, project managers and senior management. The test process is supplied with sufficient time, money and resources (both quantitatively and qualitatively) to perform a good test. Co-operation and communication with the others in the project results in an efficient process and in earlier involvement.
Testing functions & training	The test team requires a certain composition. A mixture is needed of different disciplines, functions, knowledge and skills. For example, apart from specific test expertise, also knowledge of the system under test is necessary, knowledge of the organisation and general knowledge of automation. Certain social skills are also very important. In order to get this mixture, education and training is needed.
Scope of methodology	For each test process a certain methodology or approach is used, consisting of activities, procedures, standards, techniques, etc. If these methodologies differ for each test process in the organisation, or if the methodology used is too generic, a lot of things have to be reinvented over and over again. The aim for an organisation is to use a methodology that is sufficiently generic to be widely applicable, but that has enough detail at the same time to be able to prevent undesired reinvention for each new test process.
Communication	In a test process communication takes place in a number of ways, both between testers as a group, and between testers and other members of the project, such as the developer, the end-user, the project manager. Topics of communication are test strategy, progress and quality of the software under test.
Reporting	Testing does not only deal with the detection of defects, but also with giving advice on (the lack of) quality of software. Reporting should be aimed at giving well funded advice to the project and customer on (the quality of) software and even on the software development process.
Defect management	Although defect management is in fact the project's responsibility, testers are strongly involved here. A good administration should be able to control the life cycle of a defect and to create several (statistical) reports. These reports are used to give well funded quality advice.
Testware management	Test products should be maintainable and reusable, and should therefore be managed. Apart from test products, also the products of prior phases, such as design and realisation, have to be managed well (although not by testers!). The test process can be seriously disrupted by delivery of wrong programme versions, etc. The demand of good management of these products, increases testability (and quality) of software.
Test process management	In order to manage each process and each activity, the four steps of the so called Deming circle are essential: plan, do, check, act. A well managed test process is of the utmost importance to perform the best possible test in the often very turbulent test arena.
Evaluation	Evaluation in this context means testing deliverables such as the functional design. In comparison to testing, the advantage of evaluation is the opportunity of early detecting defects. This causes the costs of repair to be considerably lower. Also, evaluation is relatively simple to establish since no programmes have to be run, no environment needs to be composed, etc.
White-box testing	A white-box test is defined as a test of the internal properties of the object, using knowledge of internal functioning. These tests are performed by developers. Well known white-box tests are the unit test and the integration test. Just like evaluation these tests take place earlier in the system development life cycle than black-box tests. Also, white-box tests are relatively cheap because less communication is required and because analysis is easier (the person detecting the defect is often the same person as the one who is to do the repairing. Besides, smaller objects are tested).

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

2.2 Levels

The way key areas are organised within a test process determines the 'maturity' of the process. It is obvious that not each key area will be addressed equally thoroughly: each test process has its strengths and weaknesses.

In order to enable insight in the state of the key areas, the model supplies them with **Levels** (from A to B to C). On the average, there are three levels for each key area.

Each higher level (C being higher than B, B being higher than A) is better than its prior level in terms of time (faster), money (cheaper) and/or quality (better). By using levels we can unambiguously assess the current situation of the test process. It also increases the ability to advice targets for stepwise improvement.

Each level consists of certain requirements for the key area. The requirements (= checkpoints) of a certain level also comprise the requirements of lower levels: a test process at level B fulfils the requirements of both level A and B. If a test process does not satisfy the requirements for level A, it is considered to be at the lowest and, consequently, undefined level for that particular key area.

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

Below a description is given of the different levels of the key areas.

Levels Key Area	A	B	C	D
Test strategy	Test strategy for single test	Combined test strategy for black-box tests	Combined strategy for black-box tests plus white-box tests or evaluation	Combined strategy for all test and evaluation activities
Life cycle model	Planning, Design, Execution	Planning, Preparation, Design, Execution, Completion		
Moment of involvement	Completion of specification	Start of specification	Start of requirements definition	Project initiation
Estimating and planning	Fundamental estimating & planning	Statistically founded estimating & planning		
Design techniques	Informal techniques	Formal techniques		
Static test techniques	Intake test basis	Checklists		
Metrics	Project statistics (product)	Project statistics (process)	System statistics	Organisation statistics
Test tools	Planning & control tools	Test execution & analysis tools	Integrated test automation	
Test environment	Managed and controlled environment	Testing in most suitable environment	Environment on call	
Office environment	Adequate & timely office environment			
Commitment and motivation	Assignment of budget & time	Testing integrated in project organisation	Test engineering	
Test functions and training	Test manager and testers	Support (methodical, technical, functional), control	Internal Quality Assurance	
Scope of methodology	Project specific	Organisation, generic	Organisation, optimising (R&D)	
Communication	Internal communication	Project communication (defects, change control)	Communication in organisation	
Reporting	Defects	Progress (status of tests and products), activities (costs + time, milestones), defects with priorities	Risks & advice, including statistics	SPI advice
Defect management	Internal defect management	Extended defect management, flexible reporting facilities	Project defect management	
Testware management	Internal management & control of test deliverables	External management & control of test basis and test object	Reusable testware	Traceability: from requirements to test cases
Test process management	Plan, do	Plan, do, check, react	Check, react in organisation	
Evaluation	Evaluation techniques	Evaluation strategy		

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

White-box testing	Life-cycle: Planning, Design, Execution	White-box design techniques	White-box test strategy	

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

2.3 Checkpoints

In order to determine levels, the TPI®-model is supported by an objective measurement instrument. The requirements for each level are defined in the form of **Checkpoints**: questions that need to be answered positively in order to classify for that level. Based on the checkpoints a test process can be assessed, and for each key area the proper level can be established. As each next level of a key area is considered an improvement, this means that the checkpoints are cumulative: in order to classify for level B the test process needs to answer positively to the checkpoints both of level B and of level A.

2.4 Test Maturity Matrix

After determining the levels for each key area, attention should be directed as to which improvement steps to take. This is because not all key areas and levels are equally important. For example, a good test strategy (level A of key area Test Strategy) is more important than a description of the test methodology used (level A of key area Scope of Methodology). In addition to these priorities there are dependencies between the levels of different key areas. Before statistics can be gathered for defects found (level A of key area Metrics), the test process has to classify for level B of key area Defect management. Such dependencies can be found between many levels and key areas.

Therefore, all levels and key areas are related to each other in a **Test Maturity Matrix**. This has been done as a good way to express the internal priorities and dependencies between levels and key areas. The vertical axis of the matrix indicates key areas, the horizontal axis shows scales of maturity. In the matrix each level is related to a certain scale of test maturity. This results in 13 scales of test maturity. The open cells between different levels have no meaning in themselves, but indicate that achieving a higher maturity for a key area is related to the maturity of other key areas. There is no gradation between levels: as long as a test process is not entirely classified at level B, it remains at level A.

Scale **Key Area**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**
Test strategy		A					B				C		D	
Life cycle model		A			B									
Moment of involvement			A				B				C		D	
Estimating and planning				A							B			
Test design techniques		A		B										
Static test techniques					A		B							
Metrics						A			B			C		D
Test tools					A			B			C			
Test environment				A				B						C
Office environment				A										
Commitment and motivation		A				B						C		
Test functions and training				A			B			C				
Scope of methodology					A						B			C
Communication			A		B							C		
Reporting		A			B		C					D		
Defect management		A				B		C						

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

Scale / Key Area	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Testware management			A			B				C				D
Test process management		A		B								C		
Evaluation							A			B				
White-box testing					A		B		C					

The main purpose of the matrix is to show the strong and weak sides of the current test process and to support prioritising actions for improvement. A filled in matrix offers all participants a clear view of the current situation of the test process. Furthermore, the matrix helps in defining and selecting proposals for improvement.

The matrix works from left to right, so low mature key areas are improved first. As a consequence of the dependencies between levels and key areas, practice has taught us that real 'outlyers' (i.e., key areas with high scales of maturity, whereas surrounding key areas have medium or low scales) give little return on investment. For example, what is the use of a very advanced defect administration, if it is not used for analysis and reporting? Without violating the model, deviation is permitted, but sound reasons should exist for it.

In the example below, the test process does not classify for the lowest level of the key area test strategy(level < A), the organisation is working conform a life cycle model (level A) and the testers are involved at the moment when the specifications are completed (level A).

Scale / Key Area	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Test strategy		A					B				C		D	
Life cycle model		A			B									
Moment of involvement			A				B				C		D	
etc.														

Based on this instance of the matrix, improvements can be discussed. In this example, a choice is made for a combined test strategy for black-box tests (=> level B) and for a full life cycle model (=> level B). Earlier involvement is at this moment not considered to be of relevance. The required situation is represented in the following matrix.

Scale / Key Area	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Test strategy		A					B				C		D	
Life cycle model		A			B									
Moment of involvement			A				B				C		D	
etc.														

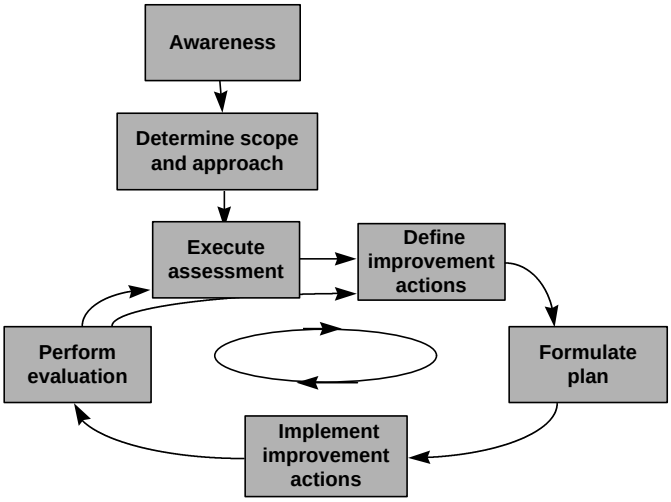
2.5 Improvement Suggestions

Improvement actions can be defined in terms of desired higher levels. For reaching a higher level the checkpoints render much assistance. Beside these, the model has other means of support for test process improvement: the **Improvement Suggestions**, which are different kinds of hints and ideas that help to achieve a certain level of test maturity. Unlike the use of checkpoints, the use of improvement suggestions is not obligatory. Each level is supplied with several improvement suggestions.

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

3. Application of the TPI® model

The process of test improvement is similar to any other improvement process. The figure below shows the various activities of an improvement process. These activities are discussed, with special attention for the places where the TPI® model can be used.



Awareness
The first activity of a test improvement process is to create awareness for the necessity to improve the process. Generally speaking, a number of problems concerning testing is the reason for improving the test process. There is a need to solve these problems and an improvement of the test process is regarded as the solution. This awareness also implies that the parties mutually agree on the outlines and give their commitment to the change process. Commitment should not only be acquired at the beginning of the change process, but be retained throughout the project. This requires a continuous effort.

Determine scope and approach
We determine what the improvement targets are and what the scope is. Should testing be faster, cheaper or better? Which test processes are subjects for improvement, how much time is available for the improvement and how much effort is it allowed to cost?

Execute assessment
In the assessment activity, an evaluation is given of the current situation. The use of the TPI® model is an important part of the assessment, because it offers a frame of reference to list the strong and weak points of the test process. Based on interviews and documentation, the levels per key area of the TPI® model are examined by using checkpoints, and it is determined which checkpoints were met, which were not met, or only partially. The Test Maturity Matrix is used here to give the complete status overview of the test process. This will show the strengths and weaknesses of the test process in the form of levels assigned key areas and their relative position in the matrix.

Define improvement actions
The improvement actions are determined based on the improvement targets and the result of the assessment. These actions are determined in such a way that gradual and step by step improvement is possible.

The TPI® model helps to set up these improvement actions. The levels of the key areas and the Test Maturity Matrix give several possibilities to define gradual improvement steps. Depending on the targets, the scope, the available time and the assessment results, it can be decided to carry out improvements for one or more key areas. For each selected key area it can

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

be decided to go to the next level or, in special cases, even to a higher level. Besides this, the TPI® model offers a large number of improvement suggestions which help to achieve higher levels.

Formulate plan
A detailed plan is drawn up to implement (a part of) the short term improvement actions. In this plan the aims are recorded and it is indicated which improvements have to be implemented at what time to realise these aims. The plan deals with activities concerning the content of the test process improvement as well as general activities needed to steer the change process in the right direction.

Implement improvement actions
The plan is executed. Because during this activity the consequences of the change process have the largest impact, much attention should be spent on communication. Opposition, which no doubt is present, must be brought to the surface and be discussed openly.
It has to be measured to what extent actions have been executed and have been successful. A means for this is the so-called "self assessment", in which the TPI® model is applied in order to quickly determine the progress. Here, the persons involved inspect their own test processes using the TPI® model.
Another vital part of this phase is consolidation. It should be prevented that the implemented improvement actions have a once-only character.

Perform evaluation
To what extent did the implemented actions yield the intended result? In this phase the aim is to see to what extent the actions were implemented successfully as well as to evaluate to what extent the initial targets were met. A decision about the continuation of the change process is made based on these observations.

4. Conclusions and remarks

Current developments proceed at a very high speed. The productivity of developers is rising continuously and the customers demand ever higher quality. Even if your current test process is fairly satisfactory, your process will need to improve in the future. The TPI®-model can help you with this.

The TPI®-model is an objective means to gain quick insight in the current situation of the test process. The model greatly offers help for improvement in the form of key areas, levels and improvement suggestions. It supports the definition of small and controlled improvement steps, based on priorities.

The reader might get the impression that use of the TPI®-model automatically leads to good analysis of the current and required situation. This is not true. The model should be seen as a tool for structuring the improvement of the test process and as a very good means of communication. Apart from the tool, improvement of test processes demands a high degree of knowledge and expertise of people involved, at least in the areas of testing, organisation and change management.

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

5. References

Beizer, B. (1990), *Software Testing Techniques*, International Thomson Computer Press, ISBN 1-850-32880-3

Bender, R. (1996), *SEI/CMM Proposed Software Evaluation and Test KPA*, STAR '96

Boehm, B.W. (1979), *"Software Engineering Economics"*, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632

Boeters, A., Noorman, B., (1997), *Kwaliteit op maat*, Kluwer Bedrijfsinformatie, ISBN 90-267-2579-5

Burns, T., Stalker, G.M. (1995), *The Management of Innovation*, Oxford University Press, ISBN 0-19-828878-6

Deming, W. Edwards (1992), *Out of the crisis*, University of Cambridge, ISBN 0-521-30553-5

Emam, K. El (editor), Drouin, J. (editor), (1998), *Spice: The Theory and Practice of Software Process Improvement and Capability Determination*, IEEE Computer Society, ISBN 0-81867-798-8

Ericson, T., Subotic, A., Ursing, S. (1996), *Towards a Test Improvement Model*, EuroSTAR '96

Gelperin, D. (1996), *A Testability Maturity Model*, STAR '96

Grady, Robert B., Caswell, Deborah L. (1987), *Software Metrics: Establising a Company-Wide Program*, Prentice-Hall, ISBN 0-13-821844-7

Graham D., Herzlich, P., Morelli, C. (1996*), Computer Aided Software Testing, The CAST-report*, Cambridge Market Intelligence Limited, ISBN 1-897977-74-3

Hall, Terence J. (1995), *The Quality Systems Manual : The Definitive Guide to the Iso 9000 Family and Tickit*, John Wiley & Sons, ISBN 0-471-95588-4

Hetzel, W. (1993), *Making Software Measurement Work*, Wiley-QED, ISBN 0-471-56568-7

Horch, John W. (1996), *Practical Guide to Software Quality Management*, Artech House Publishers, ISBN 0-89006-865-8

Humphrey, Watts S. (1989), *Managing the Software Process*, Addison-Wesley, ISBN 0-201-18095-2

Jarvis, A., Crandell, V. (1997), *Inroads to Software Quality*, Prentice Hall, ISBN 0-13-238403-5

Juran, J.M. (1988), *Juran's Quality Control Handbook*, McGraw-Hill, ISBN 0-070-33176-6

Kaner, C., Falk, J., Nguyen, H. Q. (1993), *Testing Computer Software (second edition)*, International Thomson Computer Press, ISBN 1-85032-847-1

Kit, Edward (1995), *Software testing in the real world*, Addison-Wesley, ISBN 0-201-87756-2

Kuvaja, P., Simila, J., Krzanik, L., Bicego, A., Saukkonen, S., Koch, G. (1994), *Software process assessment and improvement: the Bootstrap approach*, Blackwell

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

McFeeley, Bob (1996), *IDEAL^{sm}: a user's guide for Software Process Improvement*, Software Engineering Institute

Myers, G.J. (1979), *The Art of Software Testing*, Wiley-Interscience, New York NY10158, ISBN 0-471-04328-1

Perry, William E., Rice, Randall W. (1997), *Surviving the Challenges of Software Testing*, Dorset House Publishing, ISBN 0-932633-38-2

Pol, M., Teunissen, R., Veenendaal, E. van (1995), *Testen volgens TMap®*, Tutein Nolthenius, 's-Hertogenbosch, ISBN 90-72194-33-0

Pol, M., Teunissen, R., Veenendaal, E. van (1996), *Gestructureerd testen: een introductie tot TMap®*, Tutein Nolthenius, 's-Hertogenbosch, ISBN 90-72194-45-4

Pol, M., Veenendaal, E. van (1998*), Structured Testing of Information Systems: an Introduction to TMap®*, Kluwer, Deventer, ISBN.90-267-2910-3

Pulford, K., Kuntzmann-Combelles, A., Shirlaw S. (1995), *A quantitative approach to Software Management, the ami Handbook*, Addison-Wesley, ISBN 0-201-87746-5

Software Engineering Institute, Carnegie Mellon University (1995), *The Capability Maturity Model*, Addison-Wesley, ISBN 0-201-54664-7

Trienekens, J., Veenendaal, E. van (1997*), Software Quality from a Business Perspective*, Kluwer Bedrijfsinformatie, ISBN 90-267-2631-7

Books on TPI®:

Dutch:

Koomen, T. and M. Pol (1998), Test Process Improvement®, Leidraad voor stapsgewijs beter testen, published by Kluwer BedrijfsInformatie,The Hague

English:

Koomen, T. and M. Pol (1999), Test Process Improvement®: a Practical Step-by-Step Guide to Structured Testing, Published by Addison Wesley Longman, London

German:

Koomen, T. and M. Pol (1999), Test Process Improvement®, Anleitung fü ein stufenweise optimiertes Testen, planned to be published in August 1999

Internet:

at 'www.iquip.nl/tpi' several TPI®-products can be viewed and downloaded. Also questions can be asked and remarks can be made. Products are available in Dutch, English and German.

Copyright © 1998 IQUIP Informatica B.V., Diemen, The Netherlands

E-Commerce Quality and You

presented by

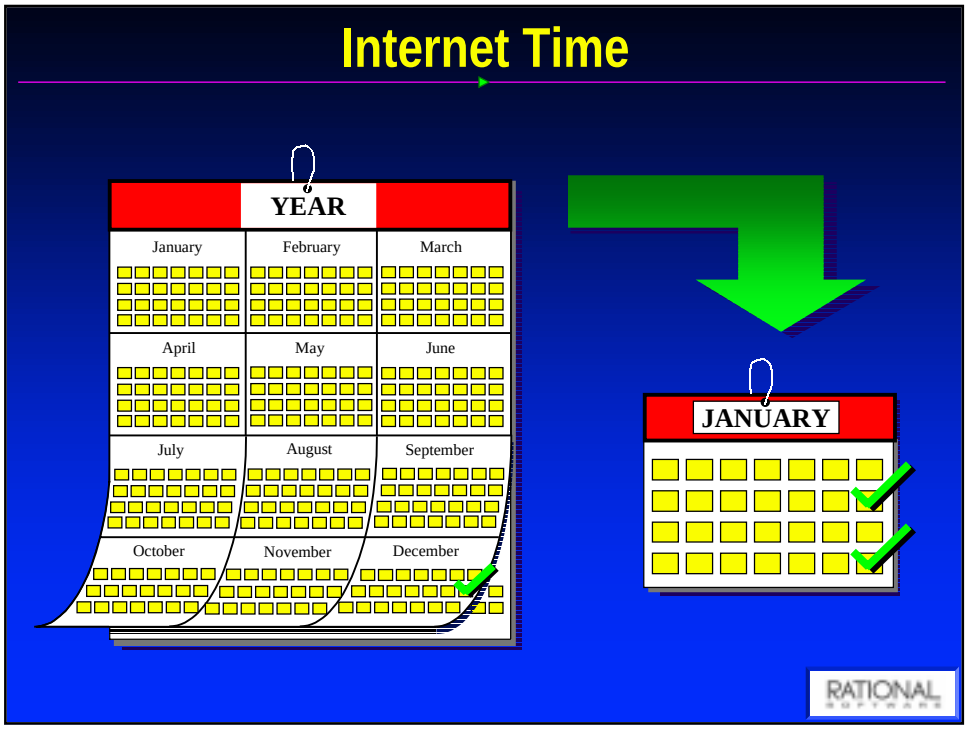
Jeff Schuster

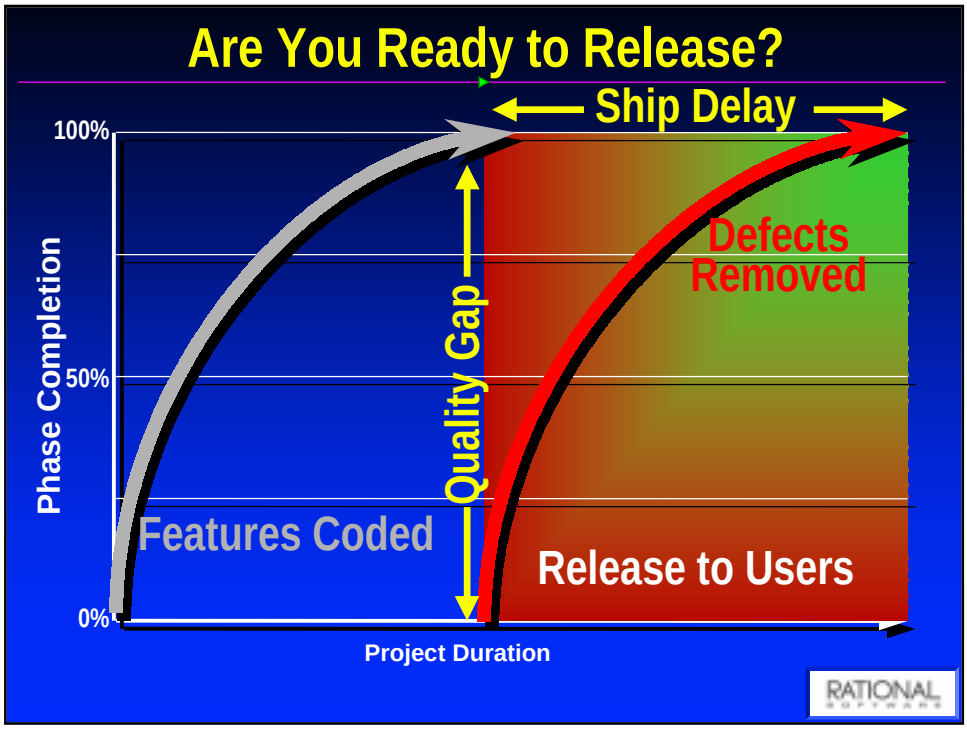
Director, Performance Test Product Management

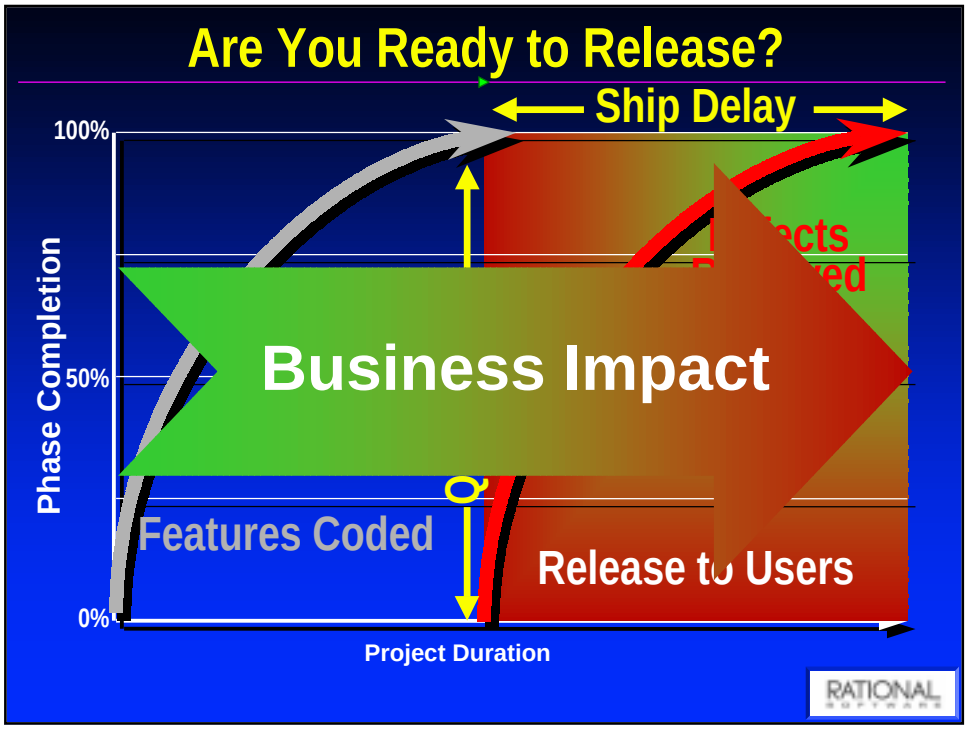
Rational Software Corporation

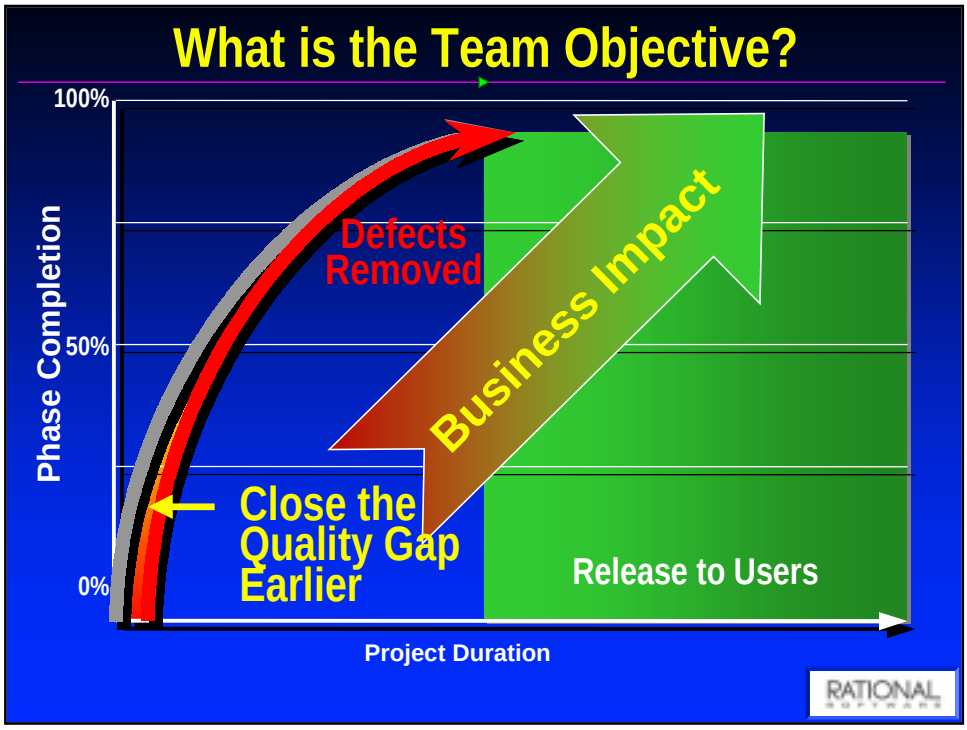
RATIONAL SOFTWARE

Do You Know Where Your Icebergs Are?


Internet Time
YEAR
January
February
March
April
May
June
July
August
September
October
November
December
JANUARY
RATIONAL


Are You Ready to Release?
Ship Delay
100%
50%
0%
Phase Completion
Quality Gap
Defects Removed
Features Coded
Release to Users
Project Duration
RATIONAL


Are You Ready to Release?
Ship Delay
100%
50%
0%
Phase Completion
Business Impact
Features Coded
Release to Users
Project Duration
RATIONAL


What is the Team Objective?
100%
50%
0%
Phase Completion
Defects Removed
Business Impact
Close the Quality Gap Earlier
Release to Users
Project Duration
RATIONAL

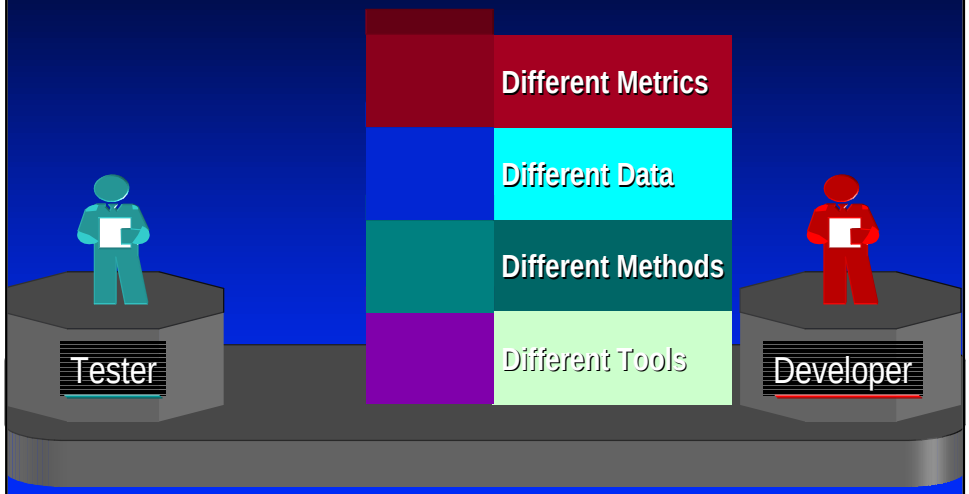
What's Your Most Common Defect "Resolution"?

1. Cannot reproduce
2. Not a bug
3. Deferred to next release

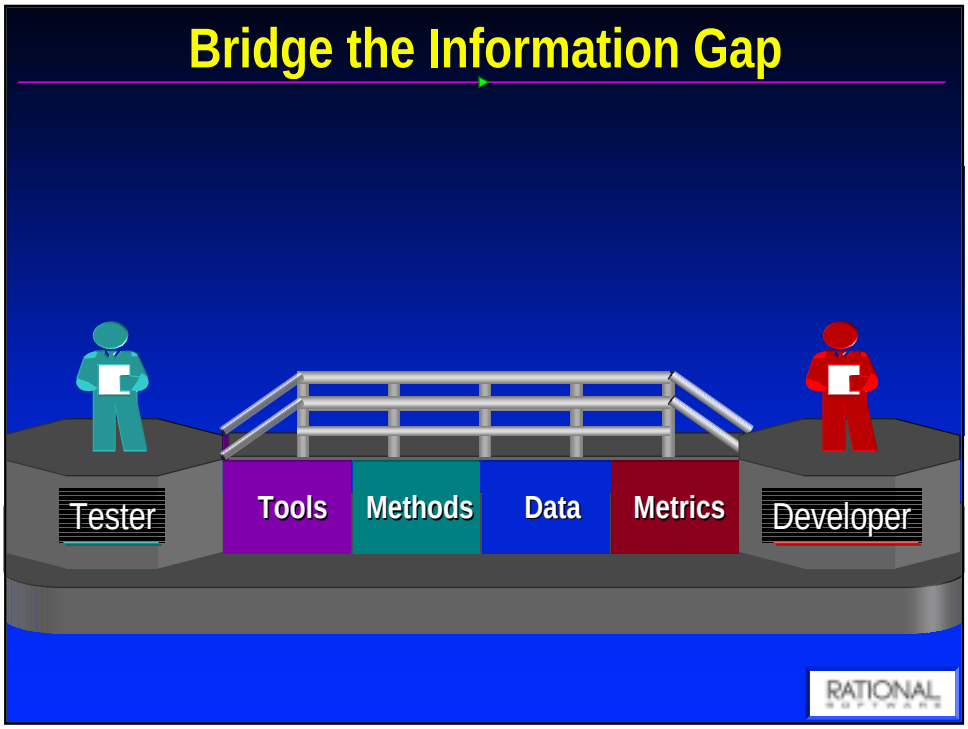
100. FIXED

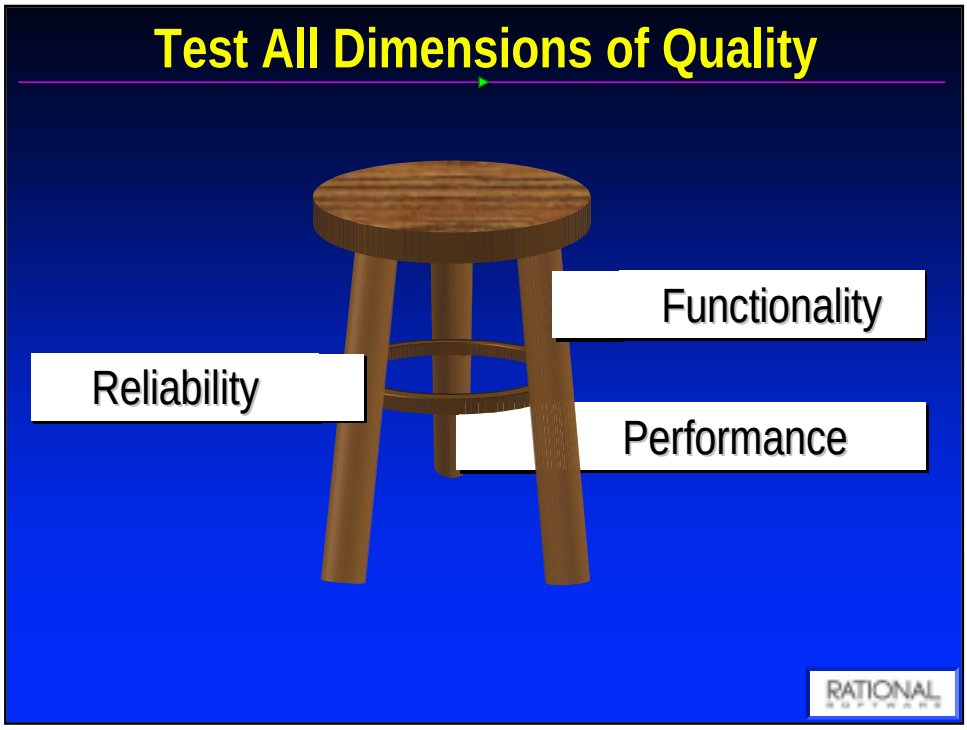
RATIONAL SOFTWARE

How Testers and Developers Work

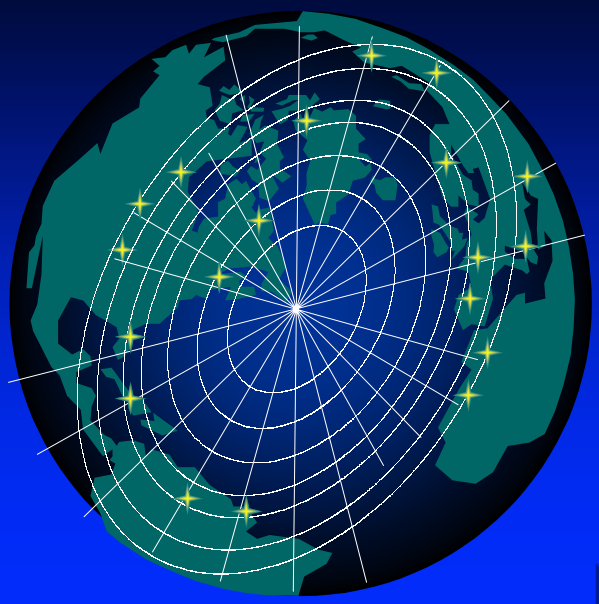


RATIONAL SOFTWARE


Bridge the Information Gap
Tester
Tools
Methods
Data
Metrics
Developer
RATIONAL


Test All Dimensions of Quality
Functionality
Reliability
Performance
RATIONAL

The Web



RATIONAL SOFTWARE

What Do These Companies Have In Common?

E*TRADE

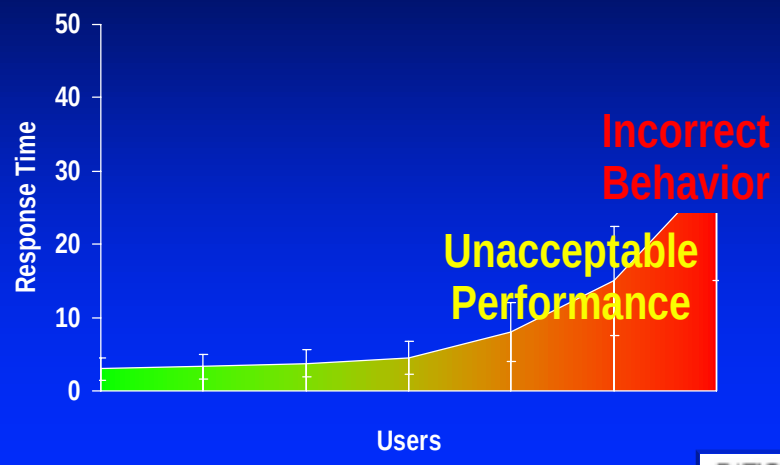
VICTORIA'S SECRET

ESPN.com

RATIONAL SOFTWARE

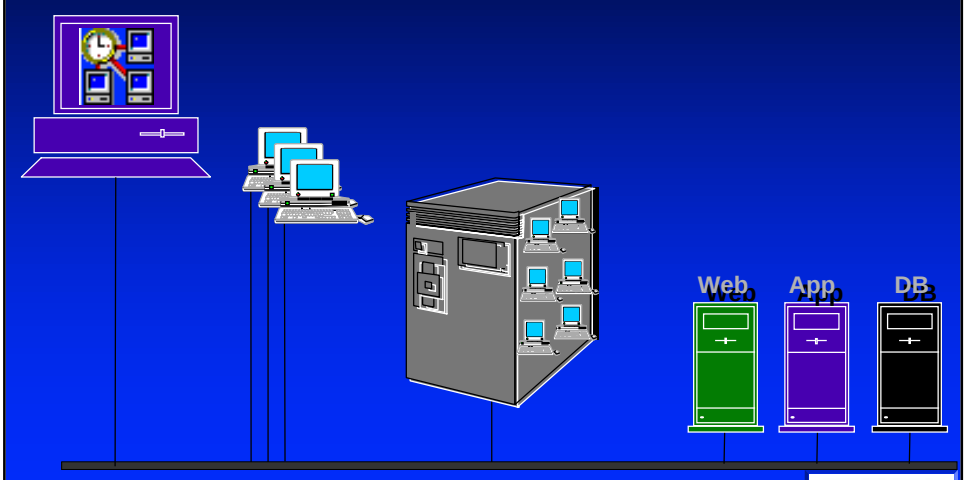
Locating the Bottleneck

- Predict when the system break in a production environment?



Load Testing Tools

- Measures system behavior under realistic load

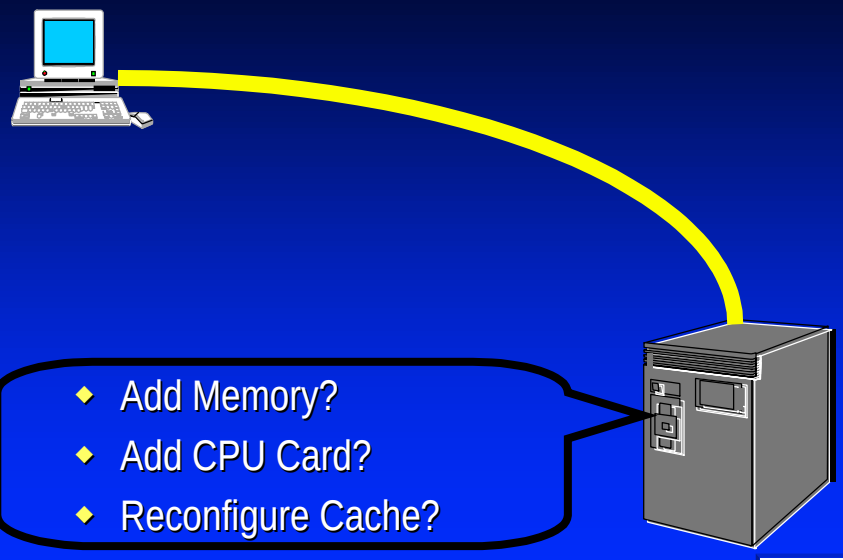


Lets Review a Simple Web Environment

- Customer presses button
- Browser submits request(s)
- Order gets processed

RATIONAL SOFTWARE

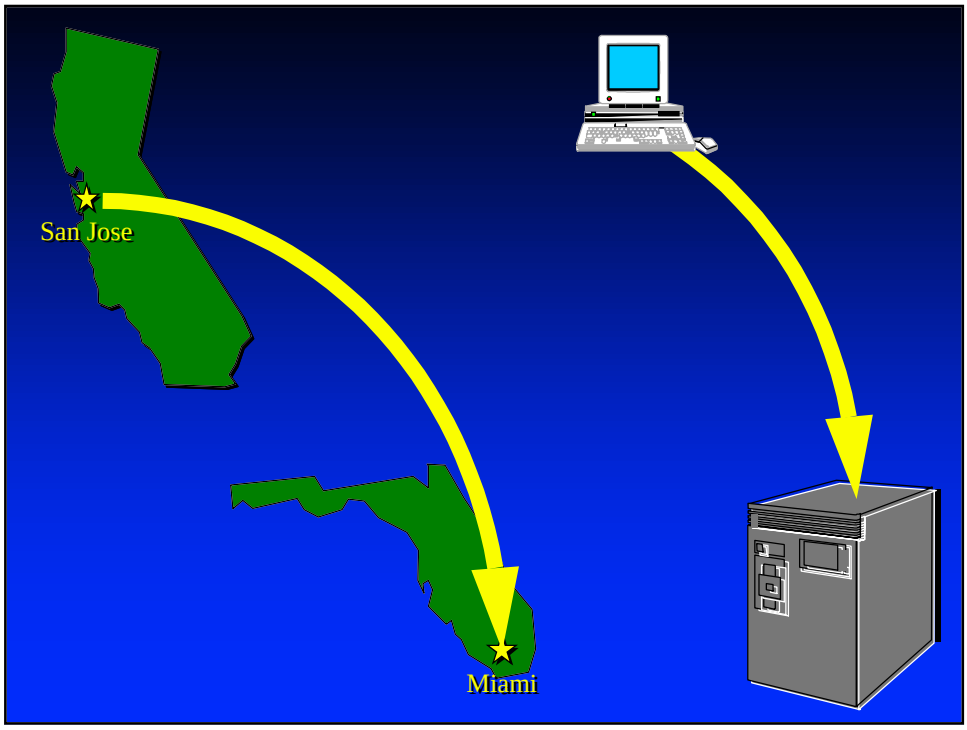
What if performance is too slow?



- Add Memory?
- Add CPU Card?
- Reconfigure Cache?

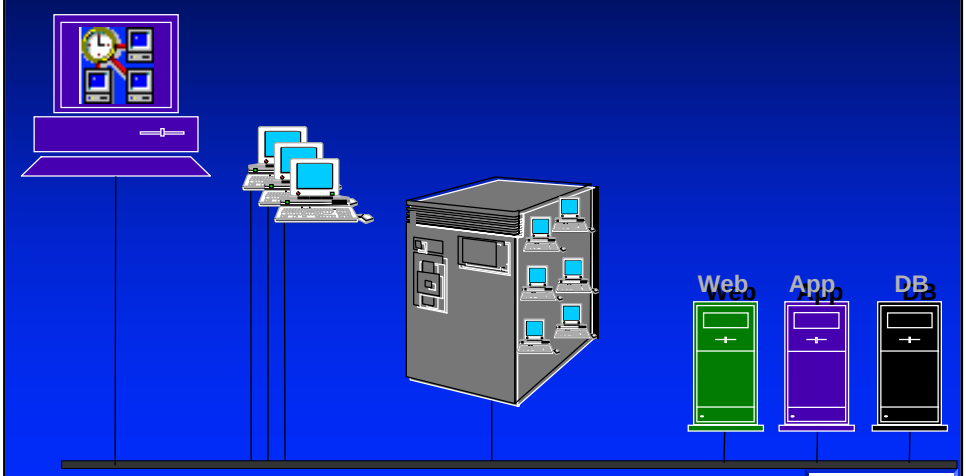
RATIONAL SOFTWARE

What if the Letter does not Arrive?
San Jose
Miami
Buy a Faster Delivery Truck?
Buy a new Watch?
Buy a Ferrari?
RATIONAL


San Jose
Miami

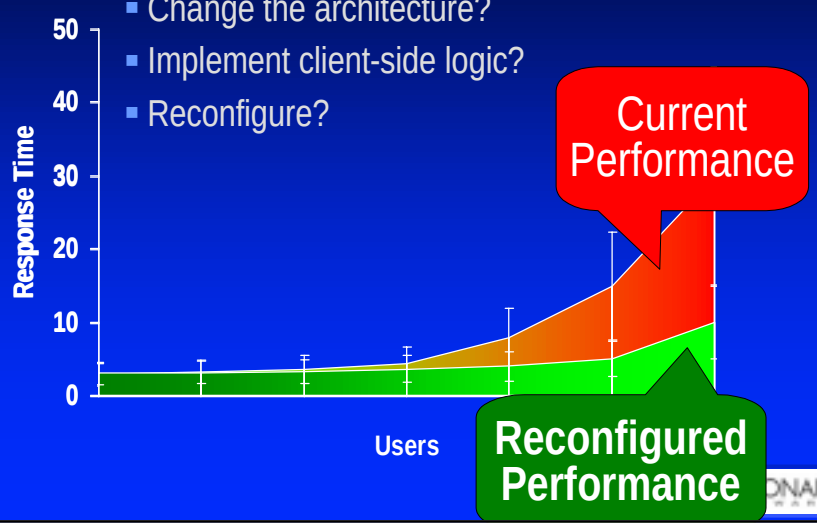
Load Testing Tools

- Measures system behavior under realistic load



Tuning the System

- Fix the system before release?
  - Change the architecture?
  - Implement client-side logic?
  - Reconfigure?



The Future: Law of Software Quality

Cem Kaner
Quality Week
May, 1999

Copyright (c) Cem Kaner, 1999.

New Statutes Are New Products

- During development, a new statute has the same problems as any other new product:
  - Different stakeholders have different, often conflicting interests
  - Stakeholders might be favored, disfavored, or simply silent and unnoticed (for now)
- Core requirements for any commercial law:
  - The purpose of commercial law is to facilitate commerce.
  - Parties to a transaction must be able to understand and trust the transaction.
  - Uniformity facilitates commerce.
  - Stability facilitates commerce.

Copyright (c) Cem Kaner, 1999.

Key Legislative Proposals and Cases

Uniform Commercial Code: Article 2B

- Rewrites all laws governing software contracts. Virtually eliminates accountability of vendors to customers, especially in COTS products. (Protects computer vendors too.)

Software Lemon Laws

- Nothing official yet, but I've seen drafts. UCC 2B pulls the pendulum one way. These pull it the other way.

Wild swings between these agendas will drive the industry into chaos.

Copyright (c) Cem Kaner, 1999.

Key Legislative Proposals and Cases

Y2K Liability Shield Laws

- On January 1, 2000, the skies will open and rain lawyers. We will be visited with a plague of lawsuits, taking over the entire judicial system for years to come. This is what happened in the Y1K crisis, and the result was the Dark Ages. To avoid this, we allegedly need Y2K shield laws (or at least Y2K Umbrellas.)

Important Recent Cases

- *Timberline Software:* Known defects didn't trigger liability.
- *Gateway 2000*: Claims of massive consumer fraud can be buried in arbitration
- *Boyd:* The shrink-wrap reasoning (of 2B) applies to mobile homes.
- *Creative Labs:* Claims of compatibility may be fully enforceable against the vendor.

Copyright (c) Cem Kaner, 1999.

Bad Software: Mis-set Customer Expectations

The Canadian government (Industry Canada, Competition Bureau) recently completed a study of the claims made on the packaging of consumer software. Here is the bottom line:

> "Over 2000 claims were evaluated during this survey. Overall, some 163 or 8.1% of all claims evaluated were potentially false or misleading. While this may appear to be a small percentage based on the number of claims evaluated, these incorrect claims represent 65% of all the software titles tested."

For detail, http://strategis.ic.gc.ca/FBP and search for "software".

50% of software publishers don't give their manuals to the test group for testing. (Savings: about 15 minutes labor per page.) (Risk: In most states, the manual creates express warranties. The product must conform to all of the manual's "statements of fact". The manual provides evidence of breach of contract.)

Copyright (c) Cem Kaner, 1999.

Bad Software: Defects

- ***In software, we routinely ship products with many known defects.***
- Complete testing is impossible. ***No one ships bug-free software.*** The most responsible publishers pick the bugs they ship more carefully and deal with complaining customers more responsibly.
- Immense pressure to ship products quickly: The 4th competitor to market probably gets less than a 3% share.
- High cost of entry for new publishers' products. Over-investment in reliability or too-high a risk of liability will kill startups.
- Watts Humphrey and colleagues report products with nearly zero coding errors discovered in the field. The state of the art is advancing.

Copyright (c) Cem Kaner, 1999.

Bad Software: Pressure on Support

- Increasingly complex hardware/software configurations drive up support calls and costs.
- Skyrocketing support costs: staff ratios rose from 1:12 to 1:7
- In 1996, 200 million calls to tech support. Software customers spent over 3 billion minutes on hold. This is tip of the iceberg because most American customers don't complain.
- Cross-industry study of call hold times: Complaining software customers left on hold for longer than any other industry studied, even longer than airlines and government offices.
- Software companies have started charging for support. $3 per minute or $35-95 per call (or incident) are common. Some companies charge even in the event of known bugs. Quality / cost pressures from support cost go away.
- Customer satisfaction with software technical support declined for ten straight years.

Copyright (c) Cem Kaner, 1999.

Bad Software: Genuinely Bad Support

Companies routinely deny their defects during calls for support (even known defects):

- we've never heard of that
- it must be your video card
- it must be you
- it's a feature

Doing research for Bad Software, David Pels and I dug up bug reports on BugNet and called publishers to complain about those bugs. We always got denials. After getting past denial, we still often heard these other excuses.

- For more data, see *Bad Software* or www.badsoftware.com/stats.htm
- For examples of lawsuits deriving from bad support, see my paper, "Liability for Bad Software and Support" at www.badsoftware.com.

Copyright (c) Cem Kaner, 1999.

Irrational Myths About Customer Dissatisfaction

- Myth: no one uses documentation.
  - *Dataquest -- 85% of people in trouble solve their own problem*
  - *Kaner's data (financial application) -- 88% of callers said they checked the docs first and could identify the weakness in the doc that led them to give up and call for help.*
- Myth: investments in support don't improve sales.
  - *Jeff Tarter, SoftLetter, on MS's $500 million investment in support: "Despite lots of wishful thinking to the contrary, spending money to upgrade a company's service reputation remains a lousy investment."*
- Myth: most calls for help reflect customer ignorance or customer fault.
  - *Kaner / Pels data (desktop publishing application): 50% of calls could have been prevented with cheap fixes.*

Copyright (c) Cem Kaner, 1999.

The Law Today: Uniform Commercial Code

- Uniform Commercial Code (UCC) is the law in 50 states.
- Article 2 governs contracts for sale of goods in USA in 49 states.
  - Sale of packaged software is a sale of goods.
  - Sale of custom software is a sale of services, not directly covered by the UCC.
- UCC is supplemented by laws governing fraud, deceptive trade practices, unfair competition, public safety, and consumer protection.

Copyright (c) Cem Kaner, 1999.

The Law Today: Copyright Act

- Copyright Act is federal law. Supercedes state laws that try to govern copying and distribution of original works.
- Copyright Act provides a balance of rights to creators / publishers and buyers.
  - First sale doctrine
    - Buyer of a copy may lend, resell, destroy, or mark up her copy. The seller's rights to that particular copy are exhausted when the sale takes place.
  - Fair use rights: limited copying allowed for
    - reviews, parody
    - classroom use
    - reverse engineering

Copyright (c) Cem Kaner, 1999.

The Law Today: Copyright

First sale and some fair use rights depend on ownership of a copy. Article 2B defines a sale of a copy of a program as a license, but read the book of Ray Nimmer, Article 2B's senior author.

> Ownership of a copy should be determined based on the actual character, rather than the label, of the transaction by which the user obtained possession. Merely labeling a transaction as a lease or license does not control. If a transaction involves a single payment giving the buyer an unlimited period in which it has a right to possession, the transaction is a sale. In this situation, the buyer owns the copy regardless of the label the parties use for the contract. Course of dealing and trade usage may be relevant, since they establish the expectations and intent of the parties. The pertinent issue is whether, as in a lease, the user may be required to return the copy to the vendor after the expiration of a particular period. If not, the transaction conveyed not only possession, but also transferred ownership of the copy.

Ray Nimmer, *The Law Of Computer Technology* § 1.18[1] P. 1-103 (1992).

Copyright (c) Cem Kaner, 1999.

UCC Article 2B: Background

- UCC is jointly maintained and updated by the National Conference of Commissioners on Uniform State Laws (NCCUSL) and by the American Law Institute (ALI).
- NCCUSL is a legal drafting organization funded by the 50 US states that writes all "Uniform" laws. It has about a 50% success rate in passage of bills introduced into state legislatures. (If the same bill is introduced into 50 legislatures, on average, 25 would pass it.)
- The American Law Institute is a non-profit body of 3000 senior lawyers (judges, professors, senior partners in BIG law firms).
- UCC and ALI appoint a joint Drafting Committee to prepare recommended revisions.

Copyright (c) Cem Kaner, 1999.

UCC Article 2B: Background

- Will govern all contracts for the development, sale, licensing, maintenance and support of software and almost all contracts involving "digital" information (will gradually encompass most books, movies, etc.).
- Current draft is over 200 pages
- 12 years in the works so far.
- Current draft significantly criticized at ALI, which passed motions calling for fundamental revision. ALI will not approve 2B in 1999.
- Despite the fact that ALI will not approve 2B and a UCC change cannot be made without ALI's approval, I believe that there is a good chance that legislators will receive 2B in 1999.

Copyright (c) Cem Kaner, 1999.

Summary of Objections to 2B: Mass-Market

- No accountability for known defects.
- Makes warranty disclaimers too easy
- Eliminates applicability of key consumer protection laws
- Limits express warranties
- Limits your uses of the software
- Limits transfer (such as resale or gift of used software)
- Lets the publisher choose state/country's law and forum
- No duty to protect your privacy
- Unreasonable electronic commerce rules
  - handling of message "receipt", online fraud, and online error
- No pre-sale or time-of-sale disclosure of the contract terms

Copyright (c) Cem Kaner, 1999.

Braucher / Linzer Resolution (ALI)

At its Annual Meeting, in May, 1998, the American Law Institute passed the following resolution (available at www.ali.org):

- "The current draft of proposed UCC Article 2B has not reached an acceptable balance in its provisions concerning assent to standard form records and should be returned to the Drafting Committee for fundamental revision of the several related sections governing assent."

The authors of the ALI resolution (Braucher and Linzer) wrote in their supporting memo:

- "The Draft reflects a persistent bias in favor of those who draft standard forms, most commonly licensors. It would validate practices that involve post-purchase presentation of terms in both business and consumer transactions (using "shrink-wrap" and "clickwrap"), undermining the development of competition in contingent terms, such as warranties and remedies. It would also allow imposition of terms outside the range of reasonable expectations and permit routine contractual restrictions on uses of information traditionally protected by federal intellectual property law. A fundamental change of approach is needed."

Copyright (c) Cem Kaner, 1999.

Summary of Objections to 2B: Larger Business

- Eliminates the perfect tender rule (cancellation for obvious defects that are found right away)
- Makes it harder to cancel a contract for “material breach”
- Lets publisher say you can never cancel the contract
- Transfer rules inflate costs of mergers and acquisitions
- Makes it risky to try an evaluation copy (publisher immunized from liability for defects that you could have found during evaluation)
- Allows self-help (shutdown of your software without a court order)

Copyright (c) Cem Kaner, 1999.

Objections to 2B: Independent Developers / Authors

- Lets publishers ban reverse engineering
- Gives big customers stricter warranties from small vendors
- Subjects consultants to mixed or uncertain law
- Reduces publishers’ duty to actively market a work
- Makes it easier to refuse payment for competently written, contracted-for articles
- Makes it easier to refuse payment for ideas submitted under contract
- Lets publishers breach their writers’ transfer restrictions

Copyright (c) Cem Kaner, 1999.

Ban On Reverse Engineering

What's the problem? Well, here's why my colleagues and I have reverse engineered.

- Personal education.
- Understand and work around (or fix) limitations and defects in tools that I was using..
- Understand and work around (or fix) defects in third-party products.
- Make my product compatible with (able to work with) another product.
- Make my product compatible with (able to share data with) another product.
- To learn the principles that guided a competitor's design.
- Determine whether another company had stolen and reused some of my company's source code.
- Determine whether a product is capable of living up to its advertised claims.

Copyright (c) Cem Kaner, 1999.

Perlman Amendment (NCCUSL)

"If a court as a matter of law finds the contract or any term of the contract to have been unconscionable or contrary to public policies relating to innovation, competition, and free expression at the time it was made, the court may refuse to enforce the contract or it may enforce the remainder of the contract without the impermissible term as to avoid any unconscionable or otherwise impermissible result."

Corresponding 2B change

"If a contract term violates a fundamental public policy, the court may refuse to enforce [it]. . . to the extent that the interest in enforcement is clearly outweighed by a public policy against enforcement of that term."

Copyright (c) Cem Kaner, 1999.

Growing Opposition to Article 2B

Here are some of the organizations that have recently asked that 2B be tabled or cancelled or that have raised fundamental criticisms of Article 2B:

- fifty intellectual property law professors (www.2BGuide.com/docs/1198ml.html)
- American Association of Law Libraries (www.arl.org/info/letters/libltr.html and www.arl.org/info/letters/Wright_ALI_letter.html)
- American Library Association (www.arl.org/info/letters/libltr.html and www.arl.org/info/letters/Wright_ALI_letter.html)
- American Society of Media Photographers (www.nwu.org/pic/uccasmp.htm)
- Association for Computing Machinery (www.acm.org/usacm/copyright/usacm-ucc2b-1098.html)
- Association of Research Libraries (www.arl.org/info/letters/libltr.html and www.arl.org/info/letters/Wright_ALI_letter.html)
- Consumer Federation of America (www.cptech.org/ucc/sign-on.html)
- Consumer Project on Technology (Ralph Nader) (www.cptech.org/ucc/sign-on.html)
- Consumers Union (www.2BGuide.com/docs/cu1098.html)
- Independent Computer Consultants Association (unpublished)
- Institute for Electrical & Electronics Engineers (IEEE) submitted specific criticisms of 2B (www.ieee.org/usab/FORUM/POLICY/98feb23.html) which have not been resolved in the ways requested. The IEEE suggested in its most recent letter (www.ieee.org/usab/FORUM/POLICY/98oct09.html) that if these issues were not satisfactorily resolved, it too would recommend tabling.
- Magazine Publishers of America (www.2BGuide.com/docs/v9-98.pdf)
- Motion Picture Association of America (www.2BGuide.com/docs/v9-98.pdf and www.2BGuide.com/docs/mpaa1198.html)
- National Association of Broadcasters (www.2BGuide.com/docs/v9-98.pdf)
- National Cable Television Association (www.2BGuide.com/docs/v9-98.pdf)
- National Consumer League (www.cptech.org/ucc/sign-on.html)
- National Music Publishers Association (unpublished)
- National Writers Union (www.nwu.org/pic/ucc1009a.htm)
- Newspaper Association of America (www.2BGuide.com/docs/v9-98.pdf)
- Recording Industry Association of America (www.2BGuide.com/docs/v9-98.pdf and www.2BGuide.com/docs/riaa1098.html)
- Sacramento Area Quality Association (unpublished)
- Society for Information Management (www.2BGuide.com/docs/simltr1098.html)
- software-test-discuss (this is the Net's largest e-mail discussion forum on software quality control)
- Special Libraries Association (www.arl.org/info/letters/libltr.html and www.arl.org/info/letters/Wright_ALI_letter.html)
- United States Public Interest Research Group (www.cptech.org/ucc/sign-on.html).

Copyright (c) Cem Kaner, 1999.

Software Lemon Laws

By overreaching, the industry is costing itself goodwill and credibility. We are risking a backlash:

- 2B has been expensive
- More press coverage of 2B than any other part of UCC
- Microsoft trial

Y2K provides one of several opportunities for dissatisfied customers (and consumer advocates) to demand greater accountability of software vendors.

- Not all demands will take into account the difficulties of developing software in this industry.

I have seen several proposals. None of these are (yet) draft legislation. I am REPORTING these, NOT ENDORSING them.

Copyright (c) Cem Kaner, 1999.

Software Lemon Laws

Cluster 1: Licensing model

- *Theme: software should be developed by professionals and craftspeople who are licensed by the state.*
- Analogy is to the building of a house. Individuals should be held liable (as are architects and plumbers). Vendors can be held liable for defects caused by their employees / contractors.
- Predicted as fallout from Y2K by Dick Bender.
- Concerns:
  - Licensed professionals are subject to malpractice suits.
  - Malpractice involves failure to exercise the skill, knowledge and diligence of a reasonably prudent, skilled professional.
  - Do we KNOW what the reasonable standards of the industry are? Could this create a malpractice lottery?.

Copyright (c) Cem Kaner, 1999.

Software Lemon Laws

Cluster 2: Consumer protection model

- *Theme: contract terms must be disclosed and some unfair terms are forbidden.*
- Special rules for known defects.
- Special incentives to disclose defects.
- Strict application of deceptive trade practices statutes.
- All vendor claims become warranties.
- Free support for actual defects (known or unknown).
- Defects give rise to entitlement to refund or free upgrades.
- Limited reimbursement for consequences of defects.

Copyright (c) Cem Kaner, 1999.

Software Lemon Laws

Cluster 3: Punitive model

- *Theme: It's their fault. They should pay for the consequences.*
  - On proof that publisher / reseller knew or should have known that a given defect existed, or on proof that a defect was caused by a development practice was unreasonably unsound---> Publisher / vendor accountable to customer for refund plus some portion of other losses.
  - On proof of a false or deceptive claim (product falsely claimed to provide benefits, meet a standard, be compatible with something else, have other characteristics or attributes) then easy-to-bring class action for reimbursement or partial refund or slightly harder class action for triple damages. Lawsuits always OK in local courts.

Copyright (c) Cem Kaner, 1999.

Other Y2K Fallout

- Extensive study of software liability by legal profession.
- Many cases fail because they involve only a breach of contract (and the contract comes with no warranties, etc.)
- Y2K and other recent suits are teaching lawyers:
  - New ways to demand assurances, statements that can later be used as evidence of fraud or deceptive practices
  - New ways to apply deceptive practice laws to contractual statements, packaging, etc.
  - New ways to prove that vendors' software-related claims were fraudulent.
- Very few current cases along these lines are getting anywhere, but people are in training. ***Where there is serious injustice, some way, eventually, people will figure out how to fight it.***

Copyright (c) Cem Kaner, 1999.

Web Sites

Article 2B

- www.law.upenn.edu/bll/ulc/ulc.htm

Kaner:

- www.badsoftware.com

Kunze:

- www.2bguide.com

NCCUSL

- www.nccusl.org

Copyright (c) Cem Kaner, 1999.

What Can You Do?

- Get organizations that you belong to to write letters to NCCUSL / ALI asking for termination of 2B. These letters from the industry are impressive.
- Attend NCCUSL meeting in late July, in Denver. Details, see www.nccusl.org.
- Write your state's NCCUSL members (e-mail me for addresses. kaner@kaner.com)
- Write your local legislators and protest 2B
- Write the heads of ALI and NCCUSL
- Write op-eds, or get press involved.
- Help me plan for an opposition campaign. I need advice, help.

Copyright (c) Cem Kaner, 1999.

Facing the Future in Testing Commerical Software:

Improving the Dev/Test Cycle

Roger Sherman

1

Definitions

(commercial software)

- Bug (1): any controversial issue that arises from the use of a product
- Bug (2): any issue that might generate a product support call

- A Bug Report is a proposed unit of labor formalized in the bug database which must be addressed (resolved) before product release

2

Definitions
(commercial software)

- Subclasses of bugs:
  - Defect: any unintended behavior of the product
  - Fault: a defect in the code (may or may not be observable through program execution)
  - Failure: a defect observed during program execution
- Testability: the likelihood that a fault will become a failure through program execution
  - *Product testability will be a key theme*

3

Definitions
(commercial software)

- Testing effectiveness
  - Testing goal: all bugs are known before release
  - Effectiveness = were all bugs known?
- Testing efficiency
  - How much labor did it take?
  - Number of formalized bugs is the biggest variable

4

Test strategy issues

- You can't test everything
- You can attempt to test anything anytime, but…
  - Test too early and you generate excessive bug reports
  - Test too late and you slip RTM or ship bugs you can't fix in time
- Automated testing (running the app) is relatively easy -- automated verification (finding the bugs) is really hard

5

Where we can go from here

- Recognize that code (in its various forms) is a database of information
- Build tools that can serve and analyze data from these models of the code
- Build standard automation and verification tools that can run on the code or models of the code

6

Method

- Gather the Test Managers of the largest projects at Microsoft
  - What factors “overwhelm” testing?
  - What are the key obstacles to testing efficiency?
  - Where are the biggest potential “wins”?
- Look for tools and research opportunities which address these problems

7

The Microsoft Test Manager’s Top Five Countdown

Biggest opportunities for increasing the efficiency of the dev/test cycle

8

V. Improve automation of component and integration testing

- Essence of the problem:
  - Developers now snap together extremely large programs from many pre-developed components
  - This creates integration bugs
  - We do not have an equally efficient test strategy/tools for component integration
- Blocking issue: no standard automation interface in components for automated testing and verification

9

V. Improve automation of component and integration testing

- Possible solutions:
  - Monkeys combined with standard automation interfaces

10

IV. Executing the right tests at the right time

- Smart testing is a “when-driven” activity
  - A story…
  - A moral...
  - Data based tools are needed -- gut feel doesn’t work

11

IV. Executing the right tests at the right time

- Same problem, bigger scale: when have we really hit code complete?
  - Test Manager: do I test now, or wait 4 weeks until they are *really* done?
  - Test Manager: do I test now, or use the next 4 weeks to substantially increase my test automation database?
  - Test Manager: what parts are ready for test, which ones are not?

12

IV. Executing the right tests at the right time

- We use crude code metrics for these decisions now
  - Churn data tells us which areas are incomplete, high risk (at project end)
  - KLOC growth trend indicates whether development is writing features or killing bugs

13

IV. Executing the right tests at the right time

- Possible solutions:
  - Orthogonal Defect Classification (ODC) from IBM
  - Better code churn analysis
  - Any data which categorizes how code is being changed by product area

14

III. Improve automation tools used for platform & configuration testing

- The proliferation of platform/configs now required is stressing the capacity of our test organizations:
  - Operating systems (including supported earlier versions)
  - Languages
  - Different SKU's
  - QFE's
  - Service Packs
  - Prior version interoperability

15

III. Improve automation tools used for platform & configuration testing

- What are the blocking problems?
  - Record/playback tools are not necessarily cross-platform
  - Record/playback tools may not even work on the *same* platform!
  - The verification problem
    - Automatic verification may be invalid on a different platform/config
    - Different platforms may require different tests

16

III. Improve automation tools used for platform & configuration testing

- Possible solutions?
  - Better visually based verification tools
  - Standardized automation interfaces built into code or OS
  - Automation built on models of the code
  - Analysis of code vulnerability

17

II. Increase the rate Development can diagnose and fix bugs found by Testing

- Improving fix rates could provide a more stable code base, which increases the efficiency of both dev and test
- Testing efficiency is also function of the number of active bugs
- Most groups “carry” one month or more worth of active bugs

18

II. Increase the rate Development can diagnose and fix bugs found by Testing

- Possible solutions:
  - Better debugger and tracing tools
  - Automatic code instrumentation
  - Stronger types

19

I. Reduce the quantity of bugs released to Testing

- Labor cost of a bug is 10 times higher if Testing finds it, than if Dev finds and fixes before check-in
- Key goal: improve the visibility of bugs in the code prior to check-in
- Optimizing the find-fix cycle before check-in is a huge win in terms of total project labor costs
- The developer's desktop is where test automation has the highest payback

20

I. Reduce the quantity of bugs released to Testing

- What we do today:
  - Check-in/Sniff tests (high leverage)
  - Build Verification Tests (almost as high)
  - Ask Developers to walk their code through the debugger before check-in (tedious, and not done very often)

21

I. Reduce the quantity of bugs released to Testing

- Possible solutions:
  - Better source code analysis tools
  - Better automated unit testing tools for developers

22

The Agenda for improving the Dev/Test cycle in commercial software

1. Reduce the quantity of bugs released to Testing.
2. Increase the rate at which Development can diagnose and fix bugs found by Testing.
3. Improve automation used for platform and configuration testing.
4. Execute the right tests at the right time.
5. Improve automation of component and integration testing.

23

Facing the Future: Usability Aspects of Quality

Jakob Nielsen, Ph.D.
Nielsen Norman Group
www.useit.com and www.NNgroup.com

Do you go to more than one search directory on a regular basis?

81%	Yes
19%	No

How do you decide which portal or search directory to use?

54%	Easy navigation
21%	Fast downloads
13%	Brand name content
6%	It was an icon on my desktop
4%	A friend recommended it
1%	Saw an ad on TV

Survey of 1,032 MSNBC readers, January 1999
http://www.msnbc.com/modules/surveys/portal.asp#survey

Factors Driving Repeat Visitors To Their Favorite Websites

75%	High Quality Content
66%	Ease of Use
58%	Quick to Download
54%	Updated Frequently
14%	Coupons and Incentives
13%	Favorite Brands
12%	Cutting Edge Technology
12%	Games
11%	Purchasing Capabilities
10%	Customizable Content
10%	Chat and BBS
6%	Other

Sources Used for Web Addresses

57%	Search Engine
38%	E-mail Messages
35%	Other Websites
28%	Word of Mouth
25%	Magazine Ads
14%	TV Commercials
11%	Periodical Articles
11%	Vendor Catalogs
9%	Newspaper Ads
7%	Banner Ads
2%	Radio Ads

Forrester survey of 8,600 Web households, January 1999
http://www.webreference.com/new/990125.html#survey

Additional Reading

My recommended list of websites about usability: http://www.useit.com/hotlist
My recommended list of books about usability: http://www.useit.com/books
Textbook: Jakob Nielsen: *Usability Engineering*, Academic Press, 1994. ISBN 0-12-518406-9

Reports about the usability of e-commerce and employment sites: http://www.goodreports.com
Unfortunately these reports are rather expensive (but enter the discount code "useit" on the online order form to save $100), so I can only recommend them for people who are in fact going to build an e-commerce site or a recruiting/HR subsite. But such readers will easily save at least a week's work in collecting usability information about specific design elements that work and don't work when doing commerce on the Web, so the report is worth it for people who are actively working on e-commerce projects. Both reports are a series of usability reports on "top" sites. Despite the sites being very famous, their usability problems are legion!

New Models for Test Development

Brian Marick
Reliable Software Technologies
marick@rstcorp.com
www.rstcorp.com/marick/

Copyright © 1999 by Reliable Software Technologies. All rights reserved.
Some images copyright www.arttoday.com

My Goals

- **Convince you of the harmful effects of piggyback models**
- **Describe a model useful to <u>testers</u>**
- **Requirements justification**
  - **see paper for explicit requirements**

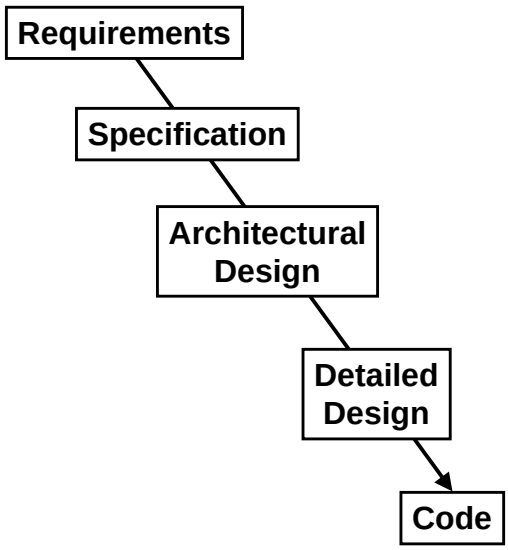
3

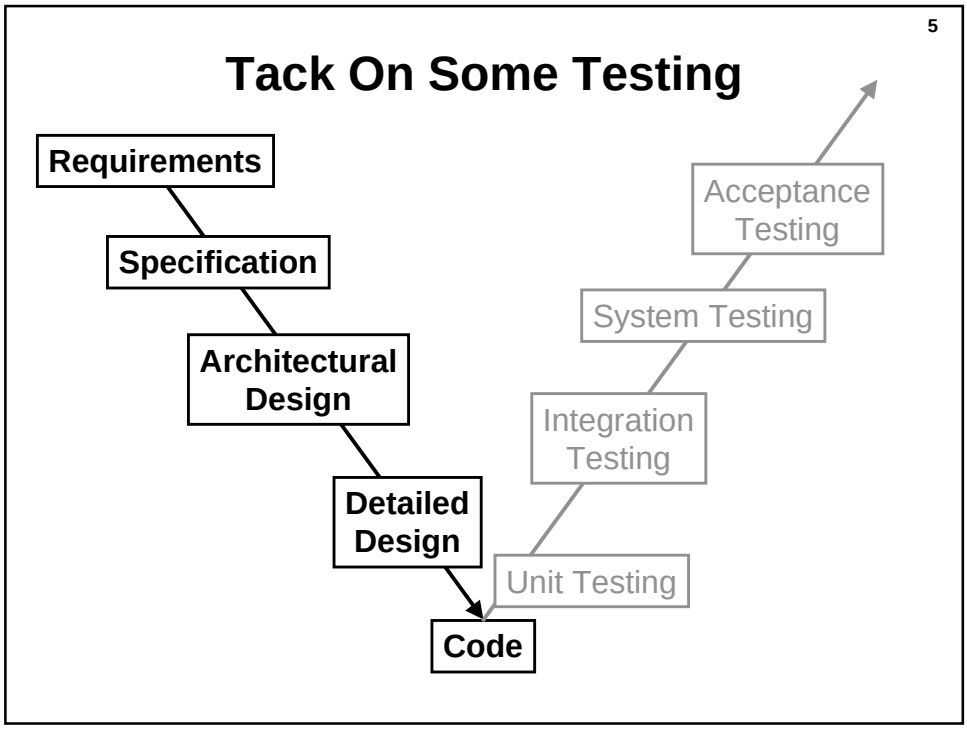
Step 1: Destroy Old Models (Build Requirements for a New One)

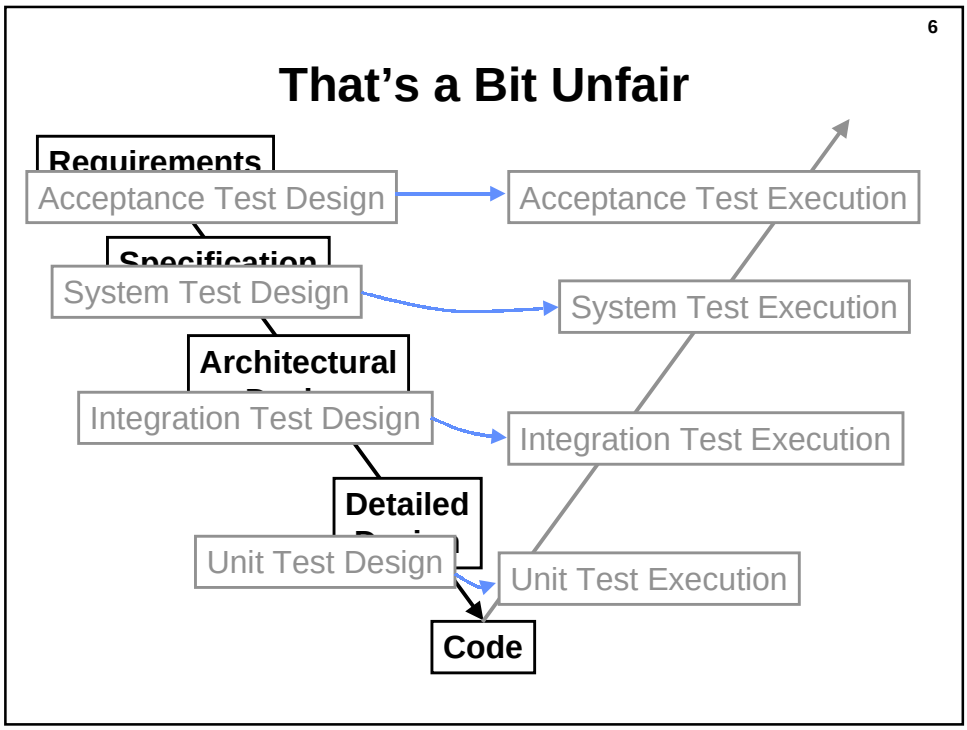
- There is no such thing as unit testing
- There is no such thing as integration testing

4

A Typical Development Model

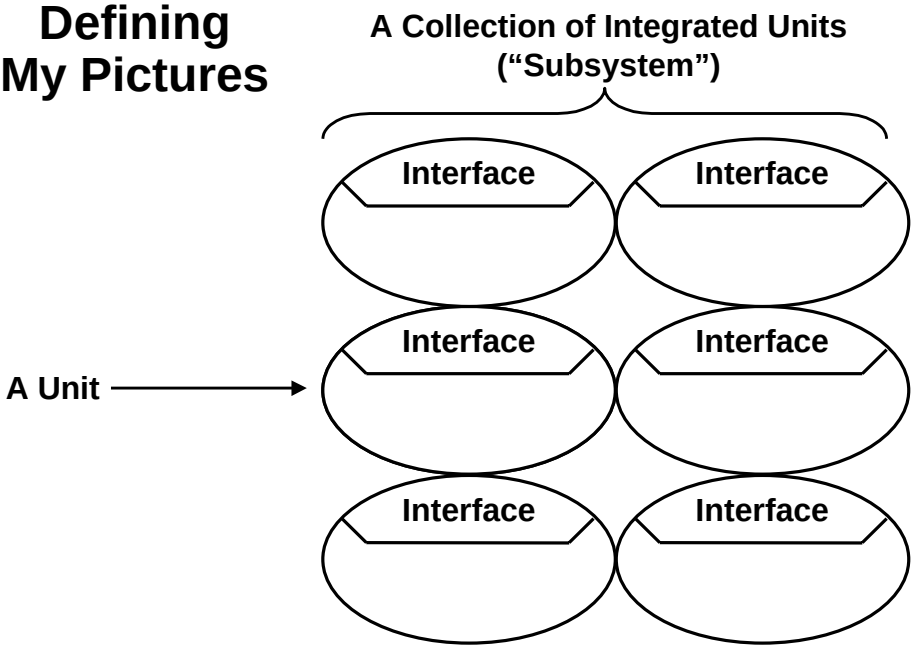



5
Tack On Some Testing
Requirements
Specification
Architectural Design
Detailed Design
Code
Unit Testing
Integration Testing
System Testing
Acceptance Testing


6
That's a Bit Unfair
Requirements
Acceptance Test Design
Acceptance Test Execution
Specification
System Test Design
System Test Execution
Architectural
Integration Test Design
Integration Test Execution
Detailed
Unit Test Design
Unit Test Execution
Code

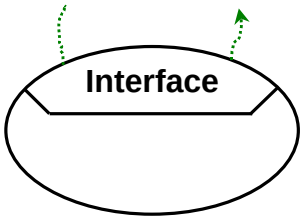
7

Defining My Pictures



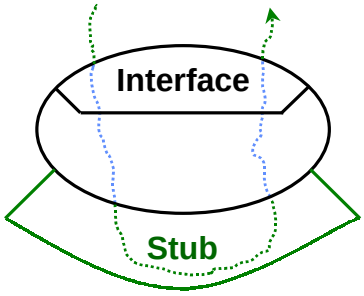
8

Unit Test Execution



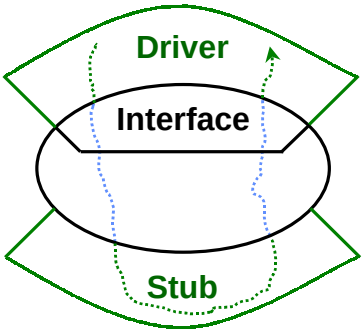
9

Unit Test Execution



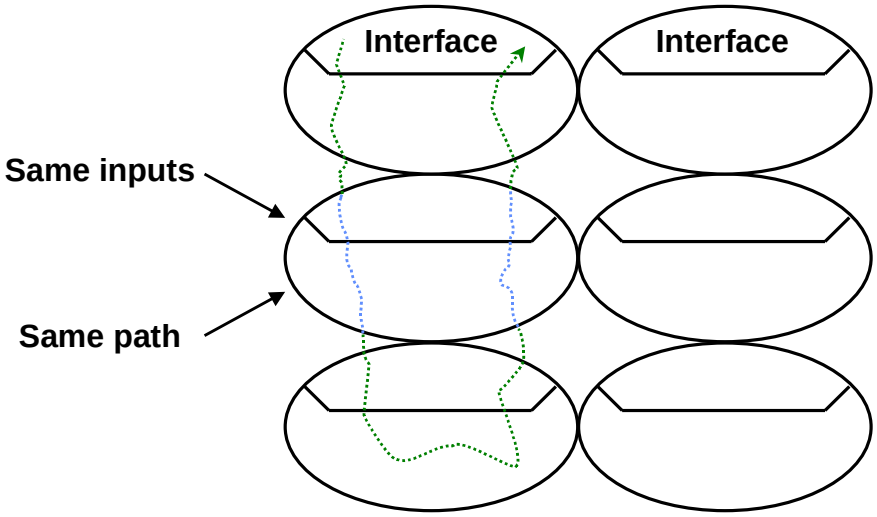
10

Unit Test Execution



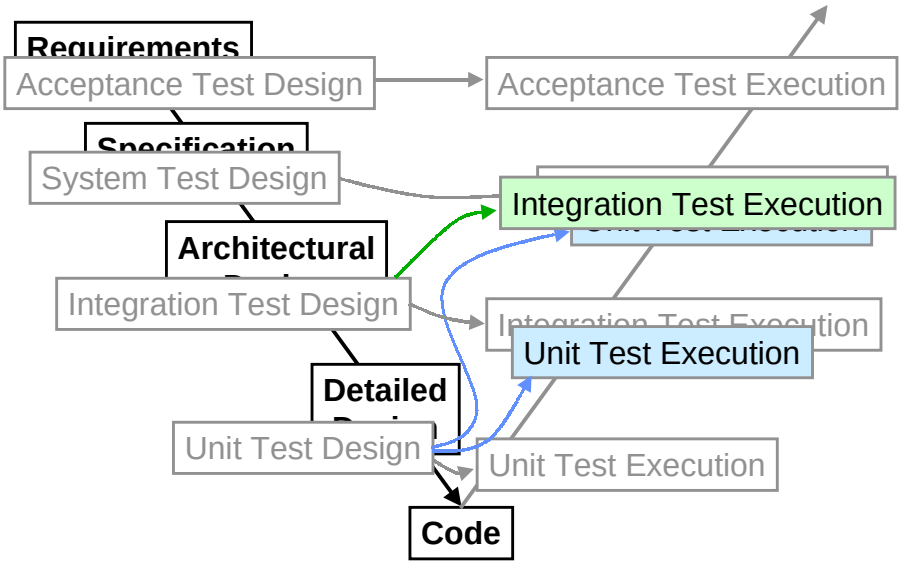
11

Unit Test Execution?



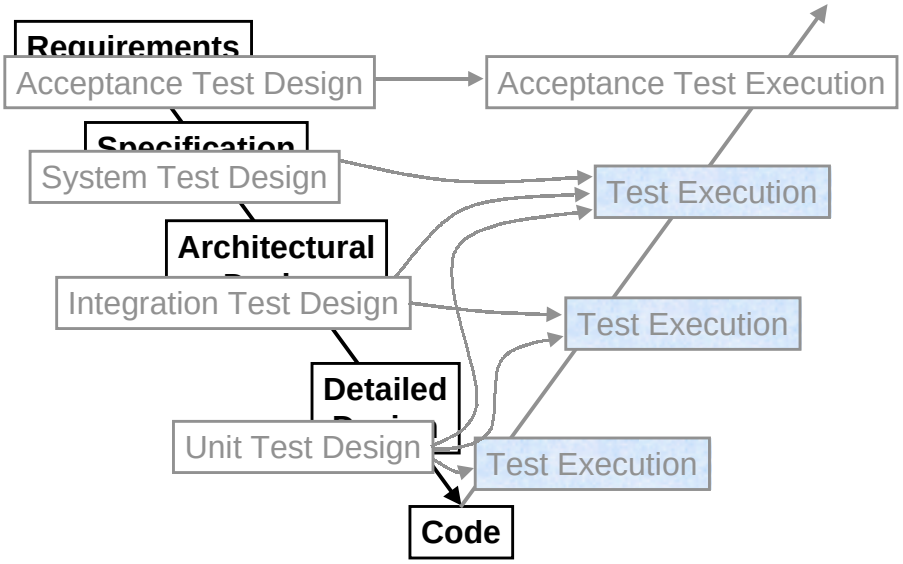
12

More Test Execution



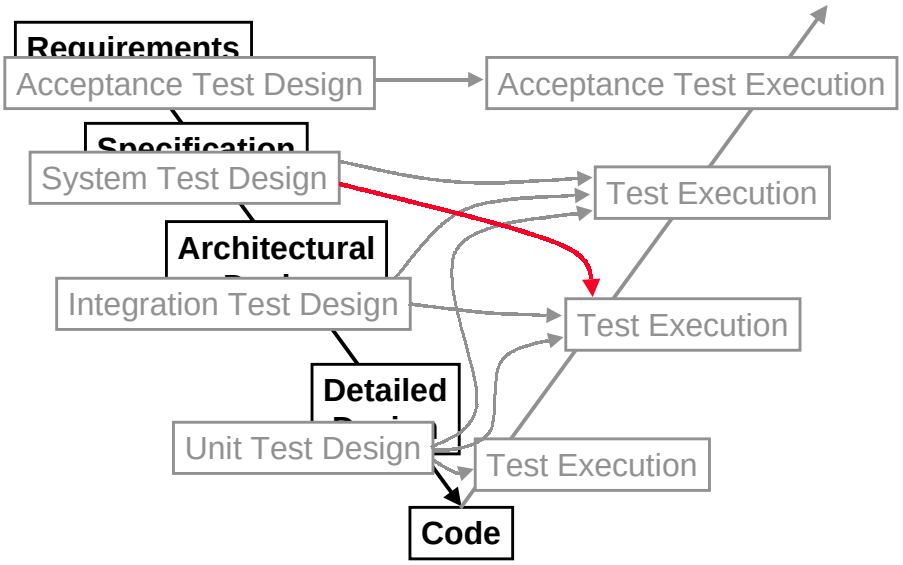
13

Eliminate Artificial Distinctions



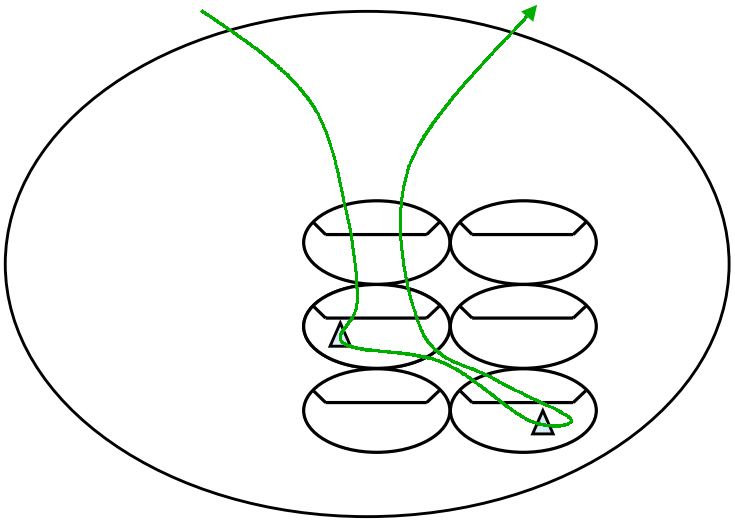
14

Downward Arrows



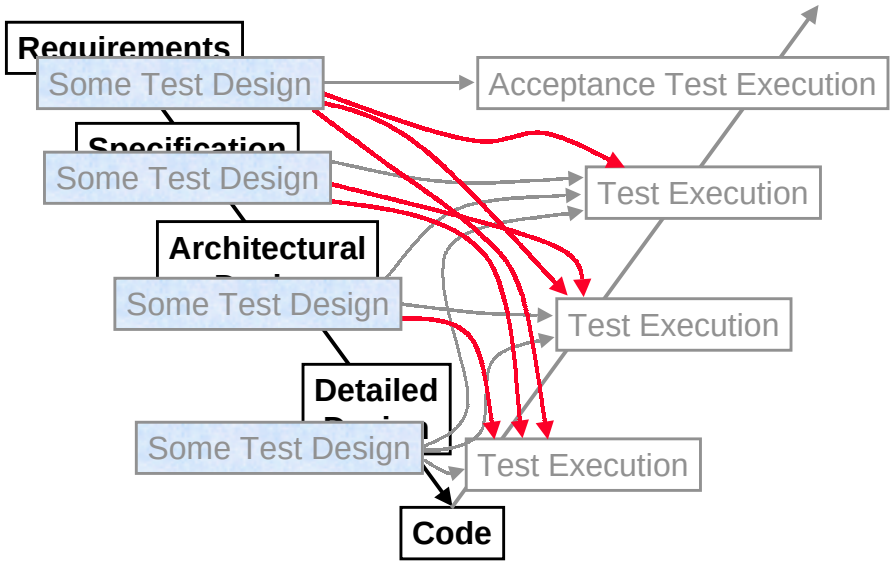
15

How Might This Happen?



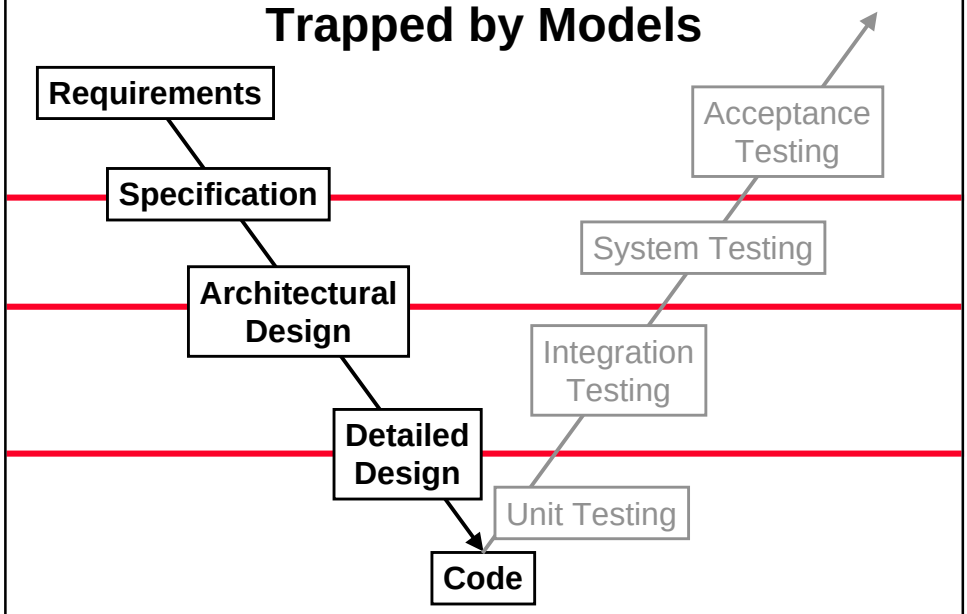
16

Downward Arrows



17

Step 1 Summary: Trapped by Models



18

The Consequences

- Information does not flow to where it can be most efficiently used
- Decisions are locked in prematurely
  - "we signed off on that already"
- Money is wasted

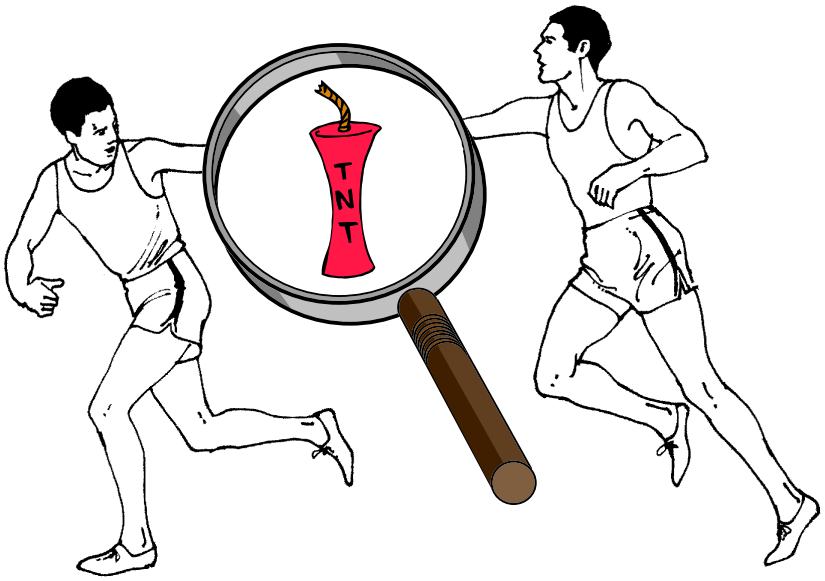
19

Step 2: Sketch a Model (Hint at New Requirements)

- Project documents are a source of information
  - but do not <u>drive</u> testing
- The project schedule drives the testing process
- See paper for requirement descriptions

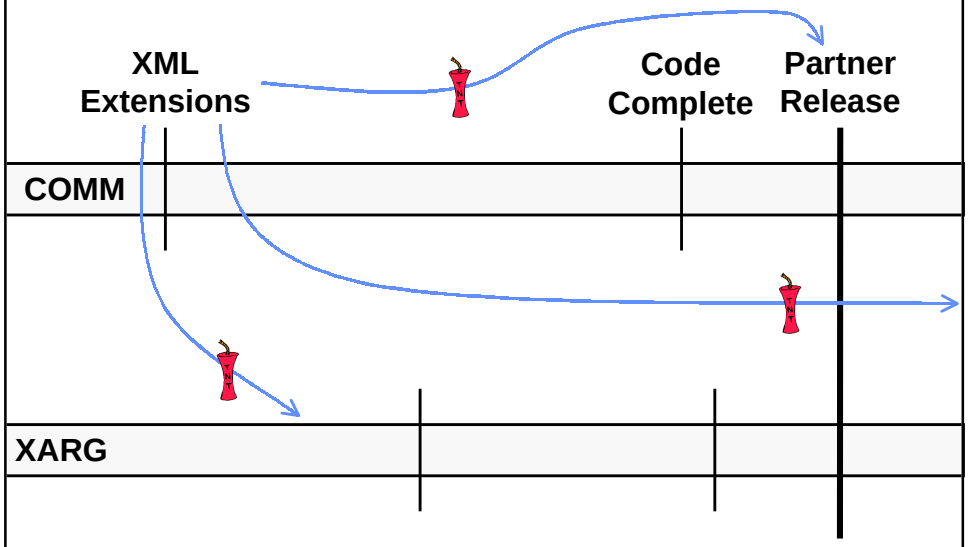
20

What Is Our Job?



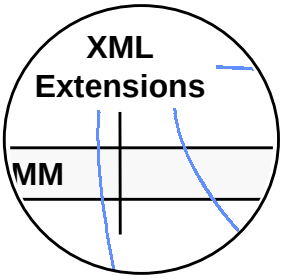
21

What Tells Us About Handoffs?

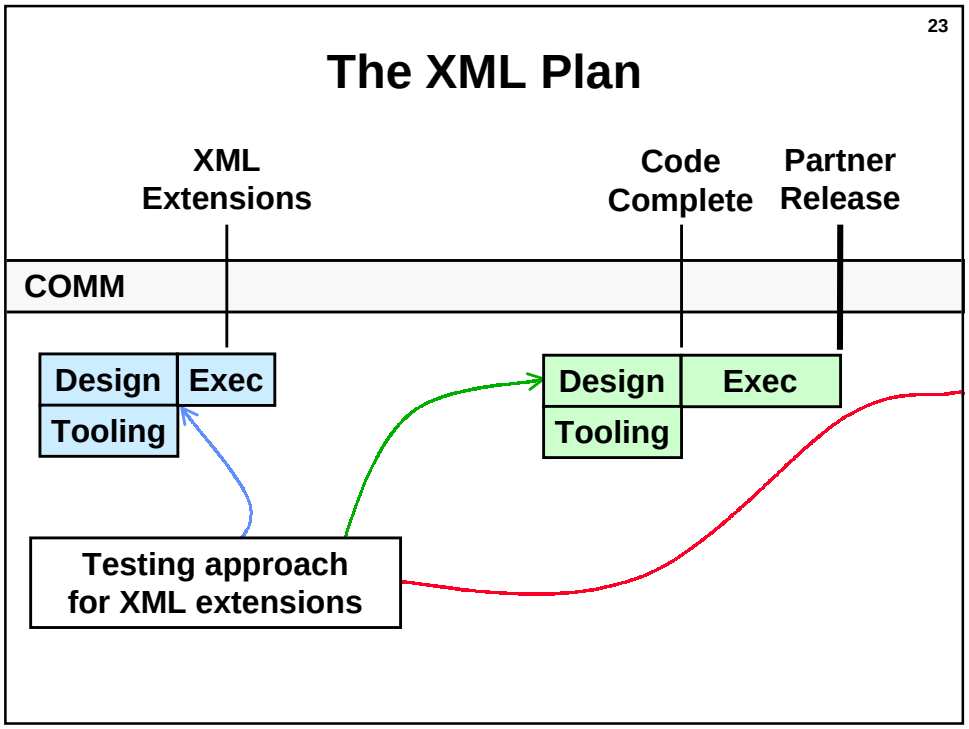


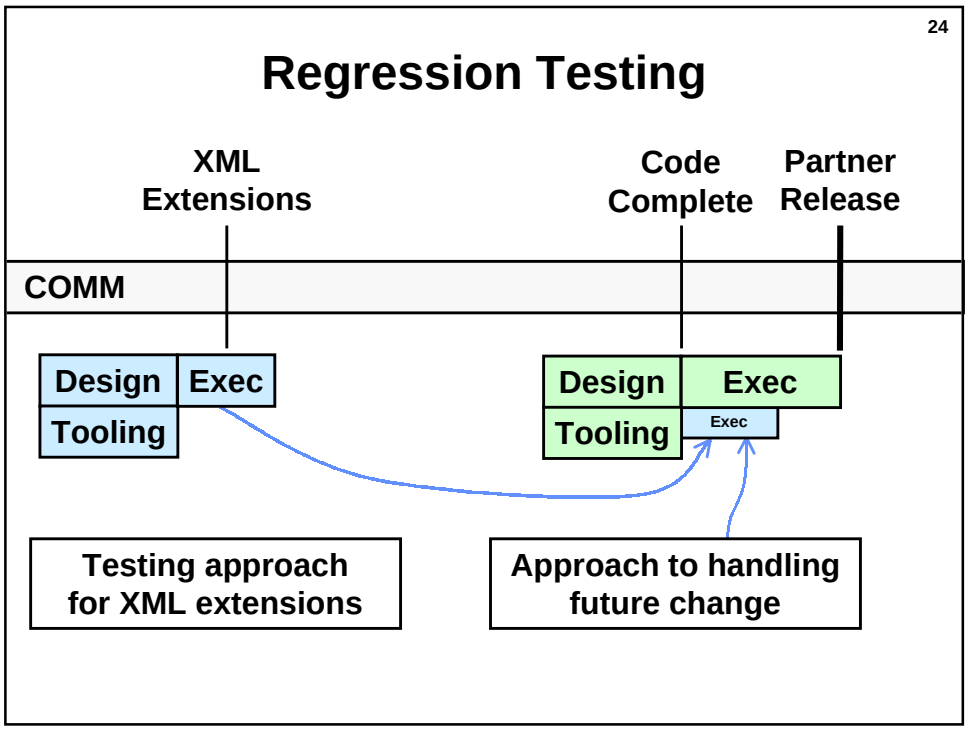
22

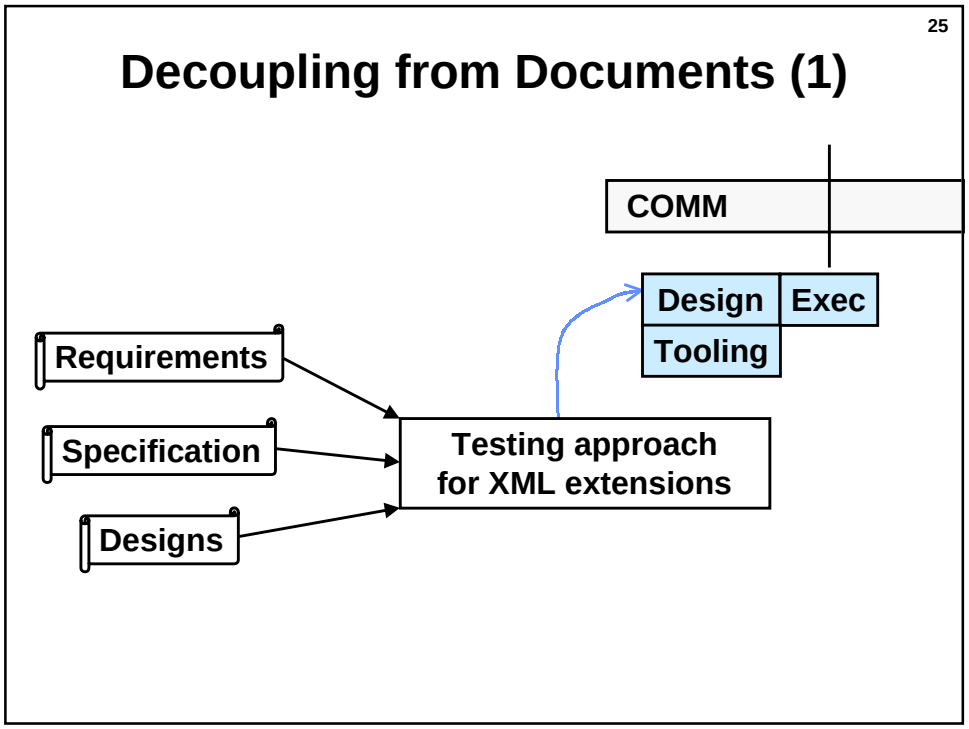
Building a Test Plan (Working Backwards)

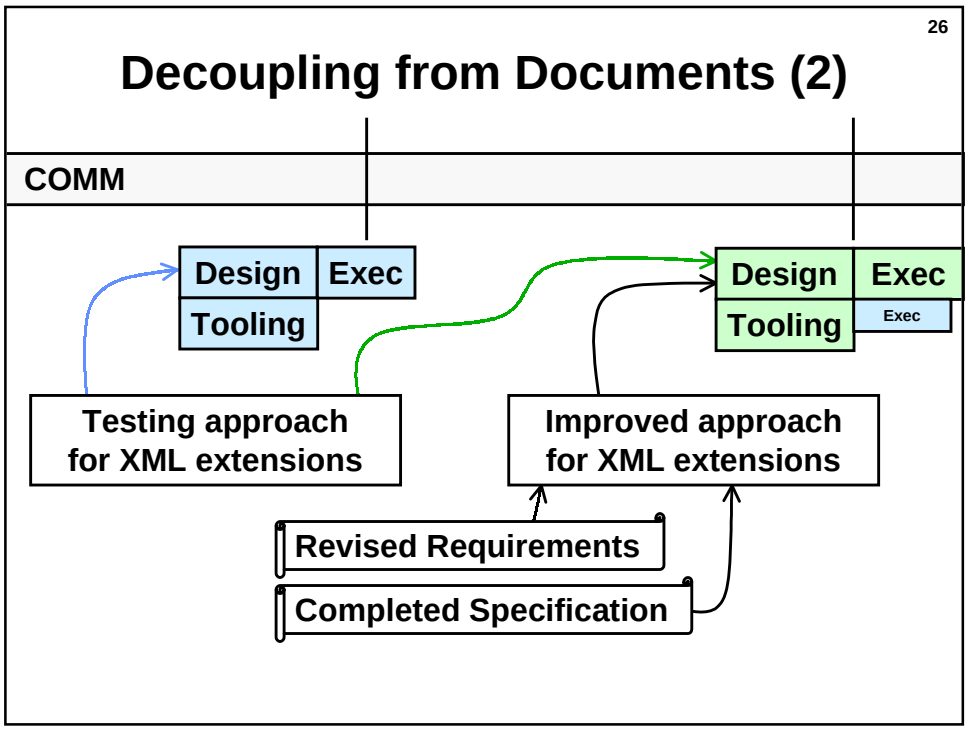


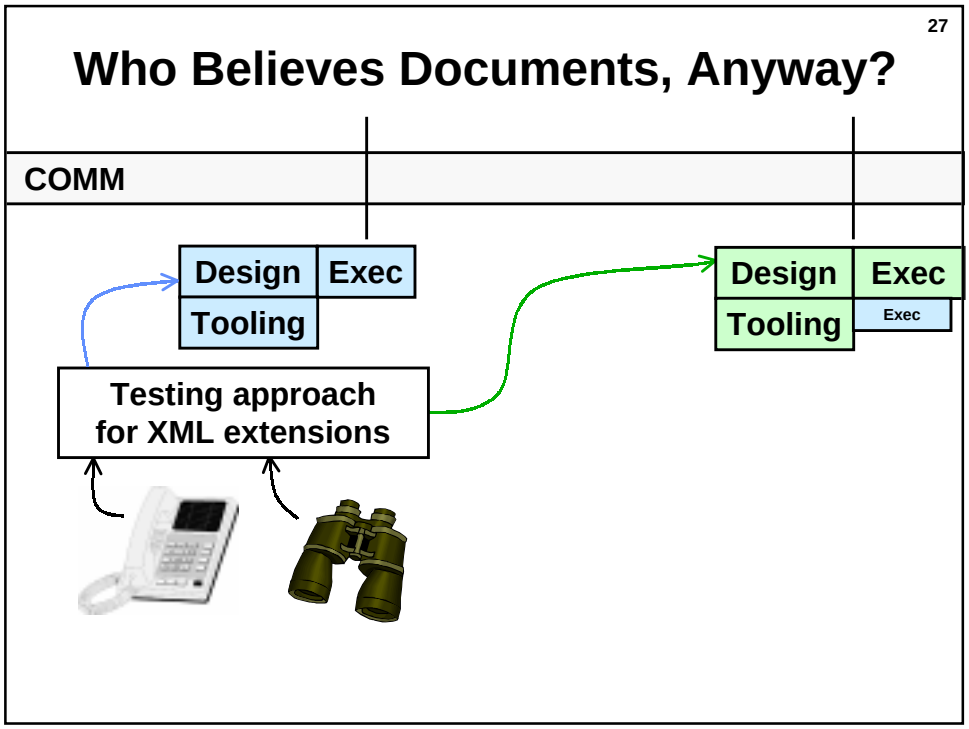
- What has changed?
- Who might be damaged?
- What tests are required?
  - when are they best executed?
  - what resources (tools, people) are required?
- (Integrate with other handoff plans)

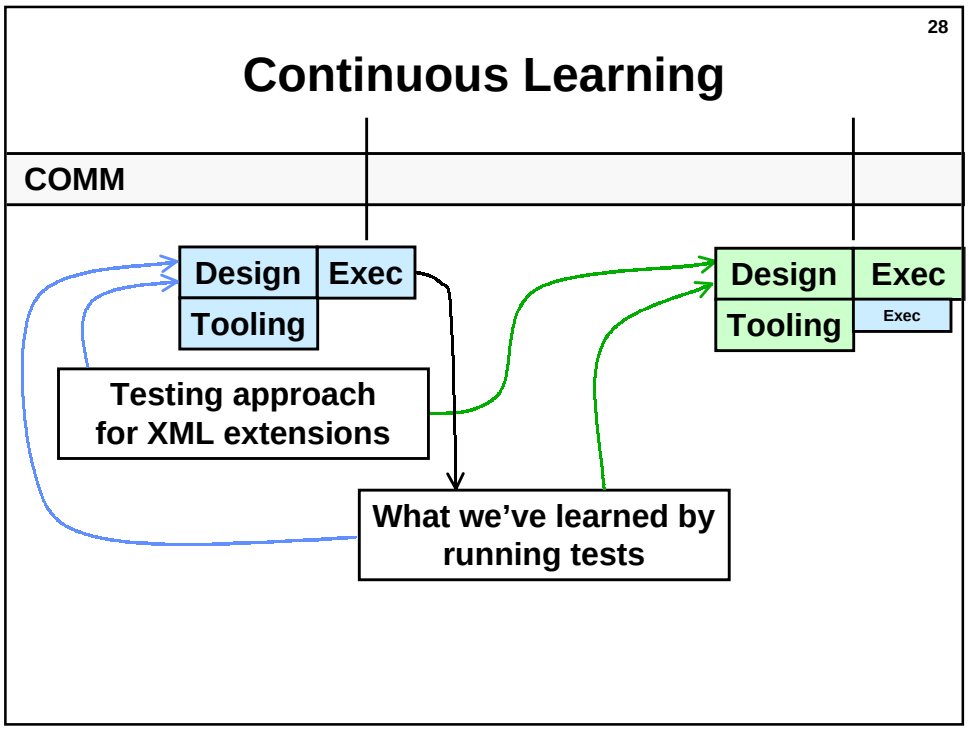

23
The XML Plan
XML Extensions
Code Complete
Partner Release
COMM
Design
Exec
Tooling
Design
Exec
Tooling
Testing approach for XML extensions


24
Regression Testing
XML Extensions
Code Complete
Partner Release
COMM
Design
Exec
Tooling
Design
Exec
Tooling
Exec
Testing approach for XML extensions
Approach to handling future change


25
Decoupling from Documents (1)
COMM
Design
Exec
Tooling
Requirements
Specification
Designs
Testing approach
for XML extensions


26
Decoupling from Documents (2)
COMM
Design
Exec
Tooling
Design
Exec
Tooling
Exec
Testing approach
for XML extensions
Improved approach
for XML extensions
Revised Requirements
Completed Specification


27
Who Believes Documents, Anyway?
COMM
Design
Exec
Tooling
Design
Exec
Tooling
Exec
Testing approach
for XML extensions


28
Continuous Learning
COMM
Design
Exec
Tooling
Design
Exec
Tooling
Exec
Testing approach
for XML extensions
What we've learned by
running tests

Facing the Future: Requirements for Models

- Test planning driven by handoffs
  - change and risk
- Test design that draws from all available sources
  - with reduced dependence on order
- Test execution informed by cost tradeoffs
- Regression testing driven by predicted change

EXPLICIT

New Models for Test Development

Brian Marick
Reliable Software Technologies
marick@rstcorp.com
http://www.rstcorp.com/marick/

A software testing model summarizes how you should think about test development. It tells you how to plan the testing effort, what purpose tests serve, when they're created, and what sources of information you use to create them. A good model guides your thinking; a bad one warps it.

I claim that most software testing models are bad ones.

They're bad because they're mere embellishments on software *development* models. People think hard and argue at length about how to do software development. They invent a model. Then they add testing as an afterthought.

A testing model has to be driven by development – after all, we're testing their work. But when the testing model is an afterthought, it's driven in the wrong way. It connects to development activities in the places that are easiest to describe, not those that give testing the most leverage. It builds upon what developers *ought* to do, not upon what they always do even when they're not following the rules. It lumps together activities that have no essential connection and separates others that belong together. It suffers from all the flaws of hasty thinking.

The result is ineffective and (especially) inefficient testing. Ineffective testing misses bugs. Inefficient testing wastes money.

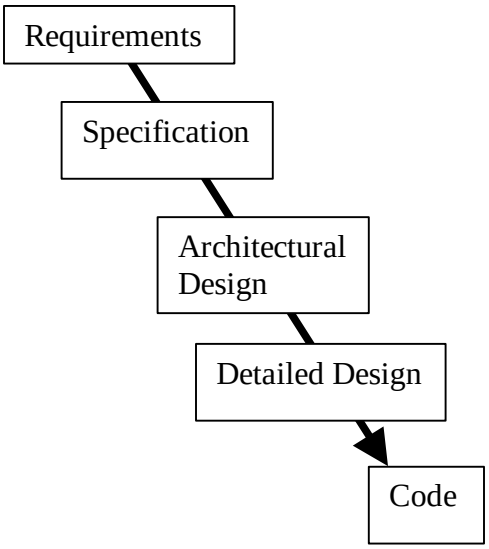
In this paper, I'll do two things. First, I'll attempt to demolish a bad model, the quite popular "V model". In the process, I hope to banish the phrases "unit testing" and "integration testing" from our vocabularies. Second, I'll describe a model I think is better. But my primary purpose is not to claim I have the right model – it's too early for that – but to describe important requirements for test models that will replace mine. Those requirements are summarized at the end of the paper.

What's wrong with the V model?

I will use the V model as my example of a bad model. I use it because it's the most familiar.

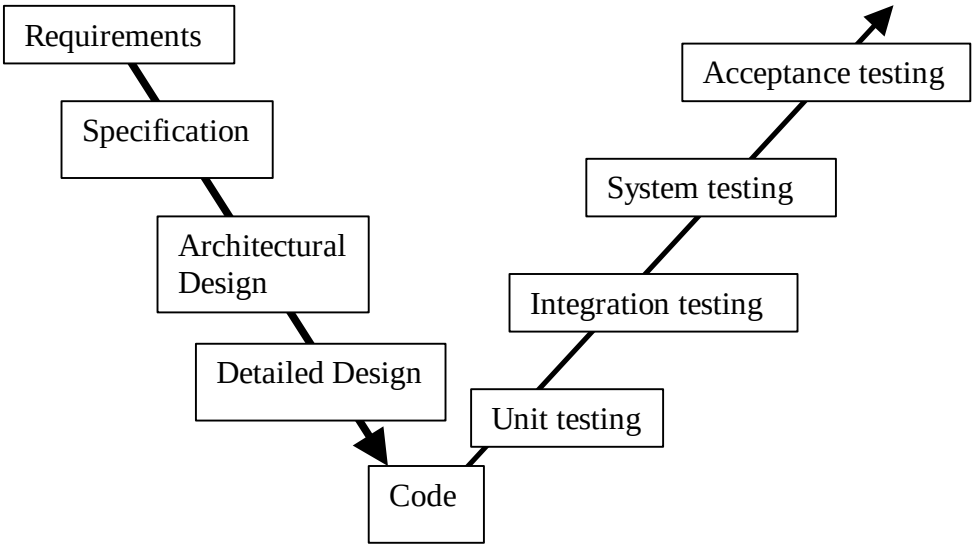
Copyright © 1999 Reliable Software Technologies. All Rights Reserved.

A typical version of the V model begins by describing software development as following the stages shown here:



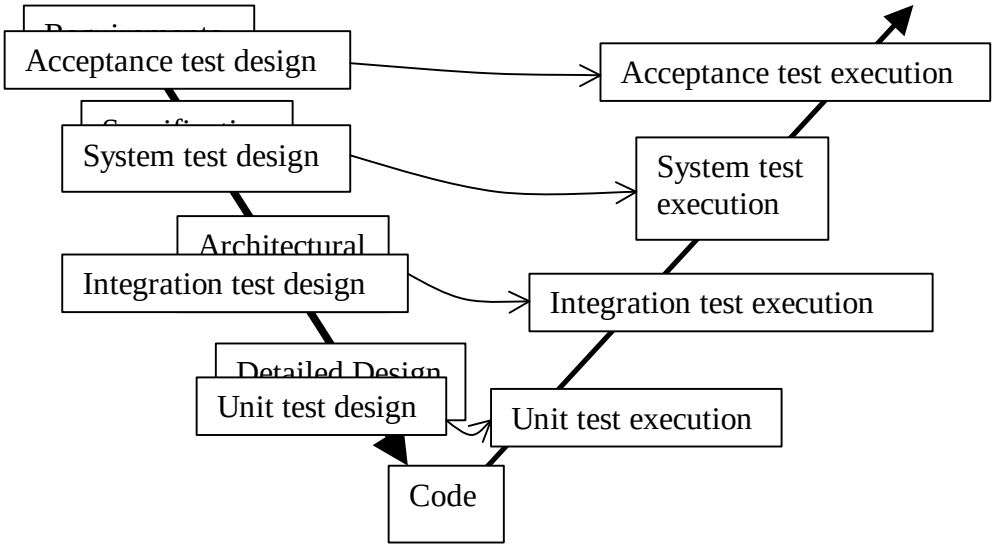
That's the age-old waterfall model. As a development model, it has a lot of problems. Those don't concern us here – although it is indicative of the state of testing models that a development model so widely disparaged is the basis for our most common testing model. My criticisms also apply to testing models that are embellishments on better development models, such as the spiral model [Boehm88].

Testing activities are added to the model as follows:



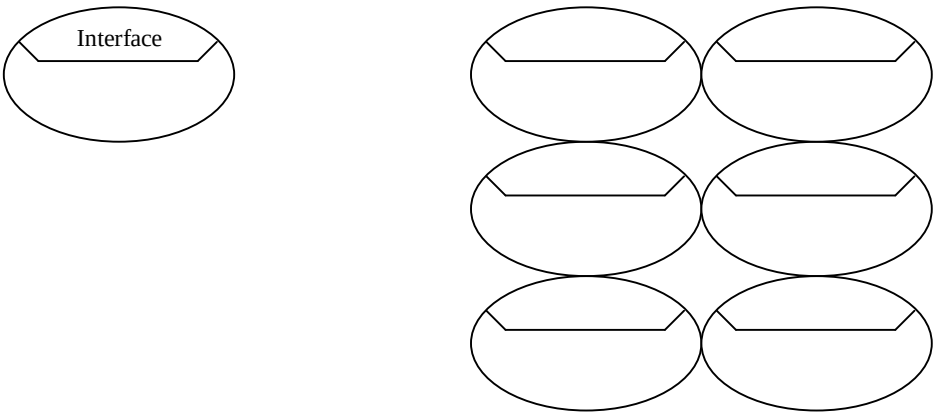
Unit testing checks whether code meets the detailed design. Integration testing checks whether previously tested components fit together. System testing checks if the fully integrated product meets the specification. And acceptance testing checks whether the product meets the final user requirements.

To be fair, users of the V model will often separate test design from test implementation. The test design is done when the appropriate development document is ready. That looks like this:



This model, with its appealing symmetry, has led many people astray. I'll concentrate on the confusion caused at the unit and integration levels.

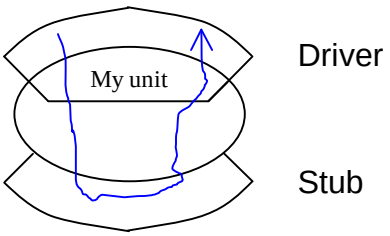
Here, I show a picture of a unit and of an aggregation of units, which I will call a *subsystem*.



There's always some dispute over how big a unit should be (a function? a class? a collection of related classes?) but that doesn't affect my argument. For my purposes, a unit is the smallest chunk of code that the developers can stand to talk about as an independent entity.

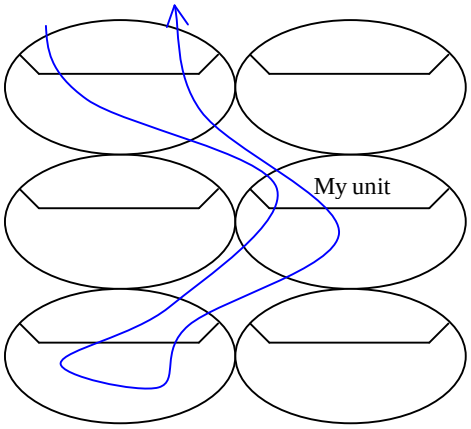
The V model says that someone should first test each unit. When all the subsystem's units are tested, they should be aggregated and the subsystem tested to see if it works as a whole.

So how do we test the unit? We look at its interface as specified in the detailed design, or at the code, or at both, pick inputs that satisfy some test design criteria, feed those inputs to the interface, then check the results for correctness. Because the unit usually can't be executed in isolation, we have to surround it with stubs and drivers, as shown at the right. The arrow represents the execution trace of a test.



That's what most people mean when they say "unit testing".

I think that approach is sometimes a bad idea. The same inputs can often be delivered to the unit through the subsystem, which thus acts as both stub and driver. That looks like the picture to the right.

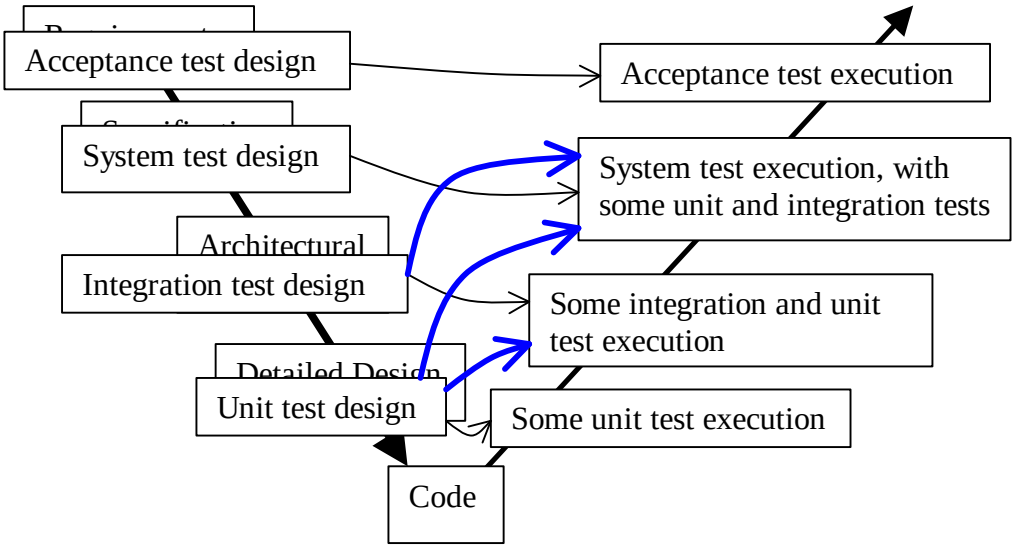


The decision about which approach to take is a matter of weighing tradeoffs. How much would the stubs cost? How likely are they to be maintained? How likely are failures to be masked by the subsystem? How difficult would debugging through the subsystem be? If tests aren't run until integration, some bugs will be found later. How does the estimated cost of that compare to the cost of stubs and drivers? And so forth.

The V model precludes these questions. They don't make sense. Unit tests get executed when units are done. Integration tests get executed when subsystems are integrated. End of story. It used to be surprising and disheartening to me how often people simply wouldn't think about the tradeoffs – they were trapped by their model.

Therefore, **a useful model must allow testers to consider the possible savings of deferring test execution**.

A test designed to find bugs in a particular unit might be best run with the unit in isolation, surrounded by unit-specific stubs and drivers. Or it might be tested as part as the subsystem – along with tests designed to find integration problems. Or, since a subsystem will itself need stubs and drivers to emulate connections to other subsystems, it might *sometimes* make sense to defer both unit and integration tests until the whole system is at least partly integrated. At that point, the tester is executing unit, integration, and system tests through the product's external interface. Again, the purpose is to minimize total lifecycle cost, balancing cost of testing against cost of delayed bug discovery. The distinction between "unit", "integration", and "system" tests begins to break down. In effect, you have this picture:[1]

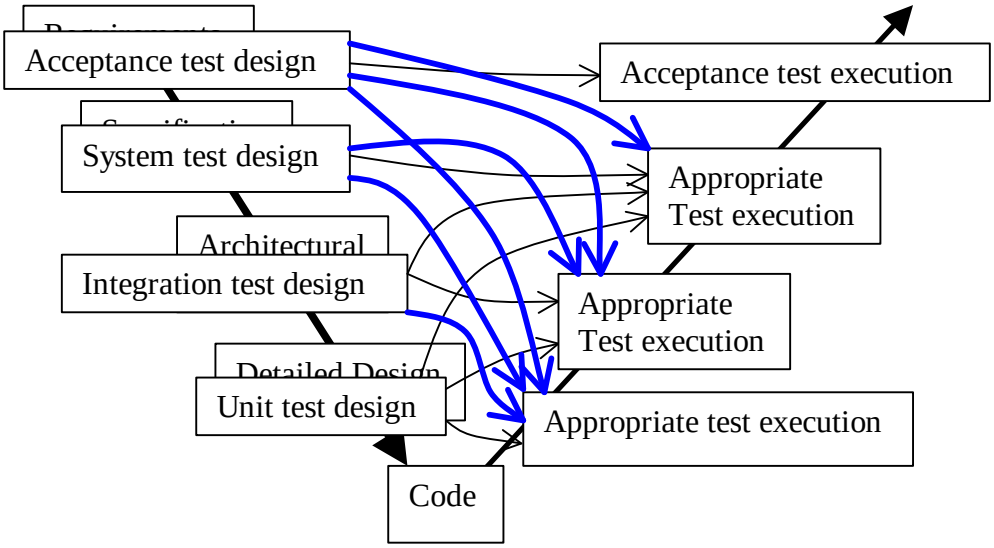


It would be better to label each box on the right "execution of appropriate tests" and be done with it.

What of the left side? Consider system test design, an activity driven by the specification. Suppose that you knew that two units in a particular subsystem, working in conjunction, implemented a particular statement in the specification. Why not test that specification statement just after the subsystem was integrated, at the same time as tests derived from the design? If the statement's implementation depends on nothing outside the subsystem, why wait until the whole system is available? Wouldn't finding the bugs earlier be cheaper?

In the previous diagram, we had arrows pointing upward (effectively, later in time). It can also make sense to have arrows pointing downward (earlier in time):

[1] I'm leaving acceptance test execution alone. It's usually done by a customer. Since it's an activity of a different group, I exclude it from this discussion.



In that case, the boxes on the left might be better labeled "Whatever test design can be done with the information available at this point".

Therefore, **when test design is derived from a description of a component of the system, the model must allow such tests to be executed before the component is fully assembled.**

I have to admit my picture is awfully ugly – all those arrows going every which way. I have two comments about that:

1. We are not in the business of producing beauty. We're in the business of finding as many serious bugs as possible as cheaply as possible.
2. The ugliness is, in part, a consequence of assuming that the order in which developers produce system description documents, and the relationships among those documents, is the mighty oak tree about which the slender vine of testing twines. If we adopt a different organizing principle, things get a bit more pleasing. But they're still complicated, because we're working in a complicated field.

The V model fails because it divides system development into phases with firm boundaries between them. It discourages people from carrying testing information across those boundaries. Some tests are executed earlier than makes economic sense. Others are executed later than makes sense.

Moreover, it discourages you from combining information from different levels of system description. For example, organizations sometimes develop a fixation on "signing off" on test designs. The specification leads to the system test design. That's reviewed and signed off. From that point on, it's done. It's not revised unless the specification is. If information relevant to those tests is uncovered later – if, for example, the architectural design reveals that some tests are redundant – well, that's too bad. Or, if the detailed design reveals an internal boundary that could easily be incorporated into existing system tests, that's tough: separate unit tests need to be written.

Therefore, **the model must allow individual tests to be designed using information combined from various sources.**

And further, **the model must allow tests to be redesigned as new sources of information appear.**

A different model

Let's step back for a second. What is our job?

There are times when some person or group of people hands some code to other people and says, "Hope you like it." That happens when the whole project puts bits on a CD and gives them to customers. It also happens within a project:

- One development team says to other teams, "We've finished the XML enhancements to the COMM library. The source is now in the master repository; the executable library is now in the build environment. The XARG team should now be unblocked – go for it!"
- One programmer checks in a bug fix and sends out email saying, "I fixed the bug in allocAttList. Sorry about that." The other programmers who earlier stumbled over that code can now proceed.

In all cases, we have people handing **code** to **other people**, possibly causing them **damage**. Testers intervene in this process. Before the handoff, testers execute the code, find bugs (the damage), and ask the question, "Do you really want to hand this off?" In response, the handoff may be deferred until bugs are fixed.

This act is fundamental to testing, regardless of the other things you may do. If you don't execute the code to uncover possible damage, you're not a tester.

Our test models should be built around the essential fact of our lives: code handoffs.

Therefore, **a test model should force a testing reaction to every code handoff in the project.**

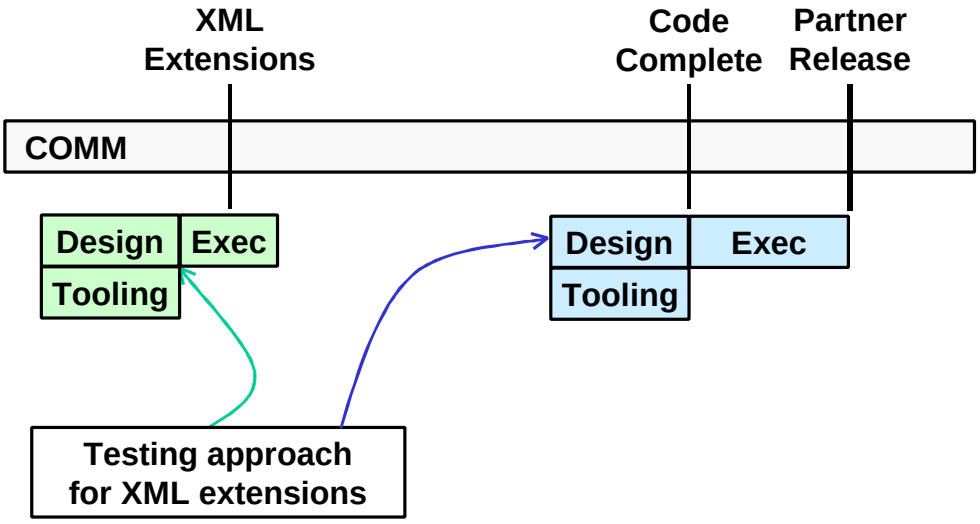
I'll use the XML-enhanced COMM library as an example. That's a handoff from one team to the rest of the project. Who could be damaged?

- It might immediately damage the XARG team, who will be using those XML enhancements in their code.
- It might later damage the marketing people, who will be giving a demonstration of the "partner release" version of the product at a trade show. XML support is an important part of their sales pitch.
- Still later, it might damage a partner who adopts our product.

We immediately have some interesting test planning questions. The simple thing to do would be to test the XML enhancements completely at the time of handoff. (By "completely," I mean "design as many tests for them as you ever will.") But maybe some XML features aren't required by the XARG team, so it makes sense to test them through the integrated partner release system. That means moving some of the XML-inspired testing to a later handoff. Or we might move it later for less satisfying reasons, such as

that other testing tasks must take precedence in the near term. The XARG team will have to resign itself to stumbling over XML bugs for a while.

Our testing plan might be represented by a testing schedule that annotates the development timeline:



Based on our understanding of the change, we've committed to doing some testing around the time the XML extensions are handed off. Test design and test support work precedes test execution. Other XML testing has been deferred to the project-wide "Code Complete" milestone, which is when all of the subsystems are integrated together and the whole product is stabilized to create the version for the trade show.

For clarity, two things aren't shown at the Code Complete milestone:

- There's a great deal of other test execution (and design and tooling) that happens there. It was deferred from other handoffs of subsystems other than COMM. Moreover, there are tests for the milestone's specific dangers. For example, there might be a set of tests that runs through the marketer's demo script, including all deviations she might inadvertently make. The goal is that she will not stand in front of an audience of one thousand and be the first person to ever try a particular sequence of inputs.
- Some of the first handoff XML tests will be re-executed at the Code Complete milestone.

My point is that test planning is made up of hard decisions about staffing, machine resources, allocation of time to test design, the amount of test support code to produce, which tests should be automated, and so forth. All those hard decisions should be driven by information about what's new in individual handoffs. If there were only one handoff, you'd work your way backwards from it:

1. Analyze risks. Who could be damaged by this change, and in what ways?
2. Decide on a test approach that addresses the specific risks.

3. Estimate the test design and implementation cost and schedule.
4. At the appropriate point in the project schedule, put the plan into action:
   A. Begin designing tests…
   B. … while also designing and creating any test support code…
   C. … and quite likely executing some tests before all have been designed.

Since there's more than one handoff, there's a potentially complex interplay between the planning driven by each handoff. Staffing has to be balanced. Test support code and tools have to be shared among tasks. You must consider to what extent tests designed for earlier handoffs will need to be re-executed on later ones.[2] And so forth.

That sounds complicated. It sounds like there's too much to keep track of, too much that could be overlooked. It may seem as if I'm asking you to perform the thinking process required by an IEEE 829 [IEEE98] Test Plan for each handoff, then merge all those thoughts into one test plan that spreads handoff-specific testing tasks across the project.

Yes and no. Thinking takes time. Too much planning can lead to diminishing returns. At some point, you need to stop planning and start acting. For example, you cannot write – or even think of – an elaborate test plan for each bug fix, even though a bug fix is a handoff.

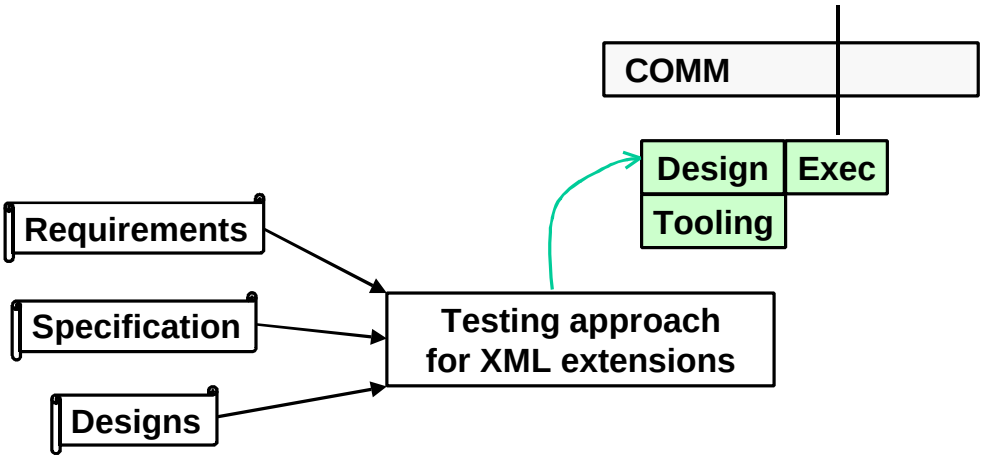
But bug fixes are a fact of life. The overall project test plan should address them. It should insist that you have a default procedure for all bug fixes. That procedure should include a build verification ("smoke test") suite that's run against each proposed fix. Effort should be devoted to thinking through what makes a good suite. Too often, build verification suites are slapped together haphazardly.

To be realistic, a model must allow some rote behavior ("do the following things for any handoff of type *X*") while encouraging just enough examination of what's special about particular handoffs. The smaller the risk of the handoff, the more rote the behavior.

A model that is explicit about the fundamental realities of testing simply must work better than a model that ignores them, that abstracts away all the real complexity of your job. As another example, there's the matter of project documentation.

I haven't mentioned requirements, specification, or design documents yet. Instead, the discussion has been driven by a list of changes in a handoff. What's the role of those documents? They're used like this:

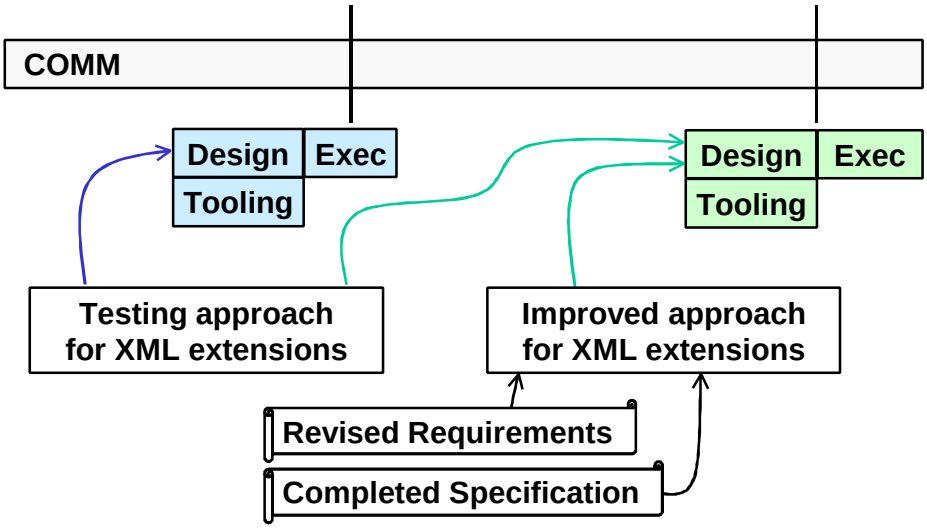
[2] Tests should be re-executed on a later handoff if the changes made as a part of that handoff are likely to cause bugs that the re-executed tests would find. Predicting which tests have value when re-executed is an interesting and tough problem. It's nowhere near as simple as "repeat them all" – the economics of that strategy aren't good. See [Marick98].



Documents guide how you react to a handoff's changes. If you had a *good* requirements document, it would be a description, perhaps indirect, of the problem the product solves. That would help you analyze risk. A *good* specification would describe the system's behavior. That would help you convert your test approach into specific tests. A *good* architectural design would help you understand the ramifications of a change: what other parts of the system might be affected? What tests need to be re-executed?

I don't see good documents very often. A requirements document is likely to be a marketing feature checklist. A specification is user documentation, delivered after the code is done. Designs don't exist.

That's OK. By concentrating on the list of changes in a handoff, I've decoupled the testing process from the software design process. If the XML additions to COMM are poorly described, I'll design the best tests I can with what I know. If, later in the project, the XML-relevant user documentation becomes available, I'll add more testing effort to a later handoff. If the marketing requirements change – as they so often do – I'll add or remove testing effort later. All that looks like this:



Therefore, **the test effort may be degraded by poor or late project documentation, but it should not be blocked entirely.**

A savvy tester doesn't trust the documentation anyway. After all, the whole point of testing is that people make mistakes. Well, didn't people write those documents?

Since "official" documents are weak, **the test model must explicitly encourage the use of sources of information other than project documentation during test design.**

Testers should talk to developers, users, marketers, technical writers, and anyone else who can give clues about better tests. Testers should attempt to immerse themselves in the culture that builds up around any technology. For example, I would expect testers working with XML to stay current with the World Wide Web Consortium's XML links (http://www.w3.org/XML/) and other XML sites and mailing lists, even including quirky ones like Dave Winer's DaveNet / Scripting News (http://www.scripting.com/). These should not be "side channels," unacknowledged sources of information. They should be sources that are planned and scheduled.

The executing tests are themselves a useful source of information. Good testers read bug reports carefully because they teach about weaknesses in the system. In particular, they often give a feel for the implications of architectural decisions that the official architectural design cannot. The act of executing tests should yield at least a trickle of new test ideas; it's a poor model that doesn't take that into account.

Therefore, **the test model must include feedback loops so that test design takes into account what's learned by running tests.**

The real complexity in our jobs is that all planning is done under conditions of uncertainty and ignorance. The code isn't the only thing that changes. Schedules slip. New milestones are added for new features. Features are cut from the release. During development, everyone – marketers, developers, and testers – comes to understand better what the product is really *for*. Given all that, how can the first version of the test plan possibly be right?

Therefore, **the model must require the test planner to take explicit, accountable action in response to dropped handoffs, new handoffs, and changes to the contents of handoffs.**

Summary

The V model is fatally flawed, as is any model that:

1. ignores the fact that software is developed in a series of handoffs, where each handoff changes the behavior of the previous handoff,
2. relies on the existence, accuracy, completeness, and timeliness of development documentation,
3. asserts a test is designed from a single document, without being modified by later or earlier documents, or
4. asserts that tests derived from a single document are all executed together.

I have sketched – but not elaborated – a replacement model. It organizes the testing effort around code handoffs or milestones. It takes explicit account of the economics of testing: that the goal of test design is to discover inputs that will find bugs, and that the goal of test implementation is to deliver those inputs in *any* way that minimizes lifecycle costs. The model assumes imperfect and changing information about the product. Testing a product is a learning process.

In the past, I haven't thought much about models. I ostensibly used the V model. I built my plans according to it, but seemed to spend a lot of my time wrestling with issues that the model didn't address. For other issues, the model got in my way, so I worked around it.

I hope that thinking explicitly about requirements will be as useful for developing a testing model as it is when developing a product. I hope that I can elaborate on the model presented in this paper to the point that it provides as much explicit guidance as the V model *seems* to.

Acknowledgements

James Bach first made me realize that test plans have to take handoffs into account. Noel Nyman and Johanna Rothman made helpful comments on a draft. Kamesh Pemmaraju and Carol Stollmeyer do not let me get away with hand-waving explanations. They want to know how ideas will work, in detail, in real life, on real projects. I sense they have only begun to force me to think things through. I look forward to it.

References

[Boehm88]
Barry W. Boehm, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, May, 1988.

[IEEE98]
"IEEE Standard for Software Test Documentation," IEEE Std 829-1998, The Institute of Electrical and Electronics Engineers, 1998.

[Marick98]
Brian Marick, "When Should a Test be Automated?" Proceedings of International Quality Week, May, 1998.
ftp://ftp.rstcorp.com/pub/papers/automate.pdf

A checklist of model requirements

These requirements are grouped according to testing activity. They were presented in a different order in the text.

A test model should:

1. force a testing reaction to every code handoff in the project.
2. require the test planner to take explicit, accountable action in response to dropped handoffs, new handoffs, and changes to the contents of handoffs.
3. explicitly encourage the use of sources of information other than project documentation during test design.
4. allow the test effort to be degraded by poor or late project documentation, but prevent it from being blocked entirely.
5. allow individual tests to be designed using information combined from various sources.
6. allow tests to be redesigned as new sources of information appear.
7. include feedback loops so that test design takes into account what's learned by running tests.
8. allow testers to consider the possible savings of deferring test execution.
9. allow tests of a component to be executed before the component is fully assembled.

The Mavin

Some lessons on predicting the end of testing.
A poem with apologies to Edgar Allan Poe
Copyright 1998, Boris Beizer

Once upon a midnight dreary as I tested, lone and weary
Copious quaint and curious cases previously untested for —
While I nodded, nearly napping, suddenly there came a tapping,
As of someone gently rapping, rapping at my office door—
"It's a manager," I muttered, "harping at my office door—
Only this and nothing more."

I distinctly still remember it was in the bleak December:
As each case's twisted member left its tracks about the store.
Eagerly I wished the daylight; and to vindicate my foresight
That this software would prove upright. And for testing? Nothing more.
For the project's culmination meant for testing, no encore.
Meant for testing, no encore.

Then the scurrilous screen saver like a silent semiquaver
Pulsed a message to consider "**YOU ARE WRONG: IT'S EVERMORE!**"
So that now, to still the beating of my heart I stood repeating
"It's a manager entreating entrance at my office door—
Some foul manager commanding closure at my office door —
This is it and nothing more."

Presently resolve grew stronger; hesitating then no longer,
"Sir" I said, "Or Madam, truly your forgiveness I implore;
But the fact is I was napping, and so gently you came rapping,
And so faintly you came tapping, tapping at my office door,
That I scarce was sure I heard you" — here I opened wide the door—
Vacant corridor and no more.

Deep along that hallway peering, long I stood there, wondering, fearing,
Doubting, dreaming dreams no mortal ever dared to dream before;
But the silence was unbroken, and the stillness gave no token,
And the only word there spoken was my whispered words, "Yet more?"
This I whispered, and an echo rumbled back the words, "Yet more!"
Merely this and nothing more.

Back into the chamber turning, and my soul within me churning;
And just then I heard a beeping louder than the tap before.
"Surely" said I, "surely that is something at a window's orifice;
Let me see then, what thereat is, and this mystery explore—
Let my heart be still a moment while this mystery explore —
Must be email and no more!"

Open then, I flung that window, maximized and then, oh, oh—
From therein erupts a Mavin, black-cloaked he, and furthermore
Through my desktop he came stomping; "What", I shuddered, "was this thing?"
Glowing, growing, and expanding, it materialized with a roar.
What's this digital Djinni wanting? I knew not what it forebore.
I've got troubles now galore!

His black suit, adroitly drafted, by expensive tailors crafted.
This consultant was no blockhead, with such clothes superior.
If I win him o'er with flattery, and beguile with triviality,
He'll return to virtuality, and then trouble me no more.
"Tell me Mavin what your name is? Are you from a distant shore?"
Quoth the Mavin, "Evermore."

Much I marveled this iconic spirit wrought by dark arithmetic,
The result of surplus double-click and too full a monitor.
For we cannot help agreeing that no living human being
Ever yet was stressed with seeing spirit Mavins unasked-for.
Vibrant, virtual Mavin planting feet so firm upon the floor,
With a name like "Evermore?"

I'm so lucky, I smiled smugly, consultations given freely —
He will answer my big question: "Tell me Mavin, I implore—
How much longer must this testing, of this software that I'm wresting?
When's the ending? Am I besting? I'm dead tired and footsore;
I've no rest nor compensation. Look how much I've done before.
Tell me, Mavin, what's the score?"

But that mavin, standing stolidly, mocked my agony, he spoke only
That one word, that word absurd, as if that word was all his lore.
Nothing further had he uttered, nor my query had he suffered:
Just one lone word, uncluttered, for his rapt inquisitor.
So I stridently demanded, "Tell me Mavin, what's the score?"
Quoth the Mavin: "Evermore!"

Startled by the stillness broken by reply so curtly spoken,
"Doubtless," said I, "what he mutters, is his only stock in store.
Just another transient hireling, cryptic words is all he's blustering,
That subverts our accomplishing what we now are sweating o'er.
On the morrow he will leave me, like the tide upon the shore.
"Thin advice; inferior."

But this Mavin was intriguing, and despite his measly offering,
Merely started, not yet parted, was this just progenitor?
Deeper wisdom still unshown? From this icon imp unknown
Might we harvest thoughts new-mown? Yet for me, he *will* mentor.
Surely there must be addenda; further insights lay in store
Than that scanty "Evermore."

Once again, I begged abjectly, "How much testing? Spare my agony,
One-word statements are no help now; this release must out the door.
Will a thousand cases stress it? Will a million just requite it?
Would a billion overpress it? Tell me *now* or quit the floor!
We must stop upon the morrow, or displease our creditor."
Yet again, just "Evermore."

So I sat in silence, staring at this somber sprite whose glaring
Gave no comfort or compassion, nor my quietus restore.
He had forfeited all amity. And I damned his dubious pedigree
Bought by artifices binary. "All my pleadings he'll ignore.
He's no help or ministration, this cantankerous counselor,
With his chant of 'Evermore.'"

"Prophet!" said I, "thing of evil! — Prophet still if sham or devil!
Whether Satan sent, or whether wrought by errant semaphore;
Tell this soul with anguish wearied; ease the pain that I have suffered;
The whole place with tests is littered — Are there bugs yet to explore?
How much better must this stuff be? When's the ending of this chore?"
Quoth the Mavin, "Evermore."

"Let that word denote our parting, foolish fiend!" I shrieked upstarting—
"Back to virtual space return ye; quit my screen for ever more!"
But the monster just laughed darkly. So uncaring of the injury,
I dispatched him to eternity— pulled the plug and dumped the core.
"Virtual demons cannot plague me — just pull the plug" is what I swore.
Pull the plug and peace restore.

Oh wretched fate that e'er I tried it. The result was just the opposite.
There's no system balm or tonic to restore my world of yore.
On my screen he squats sardonic; in my registry demonic;
And every hour on the hour's tick, blinks a message I can't ignore;
The pale blue screen of death is blinking, blinking, blinking "**EVERMORE**!"
Blue screens blinking "**EVERMORE!**"

So when angry bosses fret thee, rant and rave around the scenery,
To your science counters violence, and premature release implore;
When the pressure's overwhelming, 'cause the end you're not foretelling,
When he screams that "Were not selling!", tell him what you knew before:
Think upon my fractious Mavin, smile and tell him "Evermore!"
Smile and tell him "Evermore."

Copyright, 1998, Boris Beizer
Reproduction of this poem or any part thereof without the written permission of the author is forbidden.

ADI APPLIED DYNAMICS INTERNATIONAL

Automating Syntax Testing: The Case of a Real-Time Simulation Tool

Dr. Selim Aissi
Wendi Hummel
Systems Quality Assurance

Applied Dynamics International
Ann Arbor, Michigan

ADI APPLIED DYNAMICS INTERNATIONAL

Goal

Introduction
Application Under Test
Syntax Testing
Applying Syntax Testing to the Application Under Test
Conclusion

To automate the fault-detection process in a Hardware-In-the-Loop (HIL), Real-Time Simulation (RTS) tool by using a Record/Replay testing tool along with established testing methodologies.

ADI APPLIED DYNAMICS INTERNATIONAL

Application Under Test

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Description

SIMsystem executing Plant Model on an actual Controller

SIMsystem may be used to:

- model a complex dynamic system to test an embedded controller, or,
- model the controller, or,
- model both.

ADI APPLIED DYNAMICS INTERNATIONAL

Application Under Test

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Description

- SIMsystem Software:
  - Advantage IDE:
    - Supports the migration of non-real-time development to real-time HIL execution
    - Accepts models created using the popular graphical programming packages, such as Simulink, SystemBuild, ACSL, and EASY5
  - Interactive Run-Time Software:
    - Provides graphical monitoring and control capabilities
- SIMsystem Hardware:
  - Compute engines (currently, PowerPC 604 microprocessor)
  - I/O devices
  - Parallel Intelligent Resource (PIR)
    - Operates the I/O in parallel with model execution without cluttering VMEbus traffic

ADI APPLIED DYNAMICS INTERNATIONAL

Application Under Test

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Description

- The ADvantage Integrated Development Environment (IDE) is a graphical interface layered on top of ADI's SIMsystem and AD RTS products.
- ADvantage IDE is a software tool that allows users to configure real-time simulations.
- ADvantage automatically configures the system and builds the files necessary for real-time simulations.

ADI APPLIED DYNAMICS INTERNATIONAL

Application Under Test

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Features

- Eliminates the need for programming the processor communication, system functions, and I/O device programming
- Process-oriented approach
- Building-block methodology
- Simple I/O connectivity
- Project-level documentation
- Able to reuse models

ADI APPLIED DYNAMICS INTERNATIONAL

Application Under Test

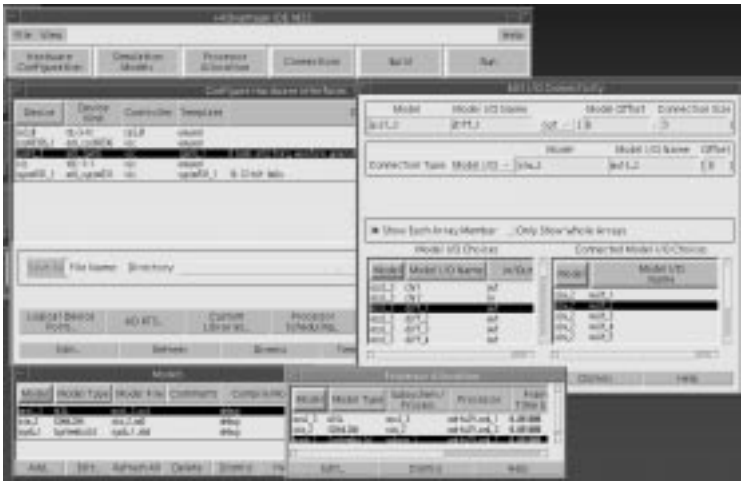
Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion



ADI APPLIED DYNAMICS INTERNATIONAL

Application Under Test

Some (Safety-Critical) Customers

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

- United Defense (The Crusader Project)
- Cummins Engine Company (Diesel Engine Controller)
- Boeing North American (Space Systems)
- Hughes Space and Communications (Satellite Control Systems)
- Rolls-Royce Military Aero Engines, Ltd. (Jet Engine Simulator)
- GEC Marconi Radar and Defense Systems Ltd. (Missile Systems Simulation)

ADI APPLIED DYNAMICS INTERNATIONAL

Application Under Test

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Testing Strategy

ADvantage IDE is used to configure and build the necessary files to run a real-time simulation.

- Uncouple semantic checks from syntactic checks
- Automate the testing process as much as possible
- Continuously maintain the test scripts
- Test the ADvantage IDE features separately
- Implement established testing methodologies, such as *Syntax Testing*

ADI APPLIED DYNAMICS INTERNATIONAL

Syntax Testing

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Analysis

- Syntax testing uses a model of the formally defined syntax of the inputs to a component.
- The Syntax is represented as a set of rules, each of which defines the possible ways of generation of a symbol in terms of sequences of other symbols.

ADI APPLIED DYNAMICS INTERNATIONAL

Syntax Testing

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Design

- Test cases with valid and invalid syntax are designed from the syntax definitions of the inputs to the component.
- Valid syntax test cases are designed to execute options that are derived from rules. Although additional rules may be applied where appropriate, the following must be included:
  - When a selection is made, an option is derived for each alternative by replacing the selection with that alternative.
  - When an iteration is made, at least two options are derived: one with the minimum number of iterated symbols, and the other with more than the minimum number of repetitions.

ADI APPLIED DYNAMICS INTERNATIONAL

Syntax Testing

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Design - Continued

- Test cases may exercise any number of options. For each test case the following must be identified:
  - The input(s) to the component (specific syntax)
  - The option(s) exercised
  - The expected outcome of the test case (new window, specific output, etc.)
- *Invalid syntax* test cases are designed as follows:
  - Create a checklist of (generic) mutations that can be applied to rules or parts of rules to generate a part of the invalid input,
  - Then, apply this checklist to the syntax in order to identify specific mutations of the invalid input, each of which employs at least one (generic) mutation
  - Finally, test cases are designed to execute those specific mutations

ADI APPLIED DYNAMICS INTERNATIONAL

Syntax Testing

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Design - Continued

- For each test case the following must be identified:
  - The input(s) to the component
  - The generic mutations used
  - The syntax element(s) to which the mutations are applied
  - The expected outcome of the test case (error message, new window, tool "crash", etc.)

ADI APPLIED DYNAMICS INTERNATIONAL

Syntax Testing

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Coverage

- There are no formal test coverage measures defined for syntax testing.
- Any measures of coverage would be based on the rules used for generating valid syntax inputs and the checklist for generating test cases with invalid syntax.
- The rules nand the checklist are NOT definitive: therefore any syntax test coverage measure based on a particular set will be specific to that set of rules and checklist.

ADI APPLIED DYNAMICS INTERNATIONAL

Syntax Testing

Introduction
Application Under Test
Syntax Testing
Applying Syntax Testing to the Application Under Test
Conclusion

Assumptions

Every testing methodology/technique is based upon some assumptions.

ADI APPLIED DYNAMICS INTERNATIONAL

Syntax Testing

Introduction
Application Under Test
Syntax Testing
Applying Syntax Testing to the Application Under Test
Conclusion

Assumptions - Continued

- **Basic Assumptions for Mutation Analysis:**
  - *The Competent Programmer Hypothesis:* Competent programmers tend to write programs that are "close" to being correct. In other words, a program written by a competent programmer may be incorrect, but it will differ from a correct version only by a few faults.*
  - *The Coupling Effect:* A test data set that detects all simple faults in a program is so sensitive that it will also detect more complex faults. In other words, complex faults are coupled to simple faults. The coupling effect cannot be proved, but it has been demonstrated experimentally and supported probabilistically.**

* A.T. Acree, *et al.* Mutation Analysis. Technical Report GIT-ICS-79/08, September 79.
** R.A. DeMillo, *et al.* Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4), April 78.

ADI APPLIED DYNAMICS INTERNATIONAL

Syntax Testing

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Assumptions - Continued

- **Bug Assumptions**:
  - Bad (Incomplete or Inconsistent) Syntax
  - Lexical (Alphabet) Problems
  - False Acceptance or Rejections of Syntax
  - Syntax/Semantics Mismatches

ADI APPLIED DYNAMICS INTERNATIONAL

Applying Syntax Testing to the Application Under Test

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Testing the "**Logical Device Parameter**":

- Value represents the value of the selected application parameter.

- The Value type is dependent on the application parameter selected.
- Valid type: float

- Wrong values may cause invalid results at simulation run-time.

ADI APPLIED DYNAMICS INTERNATIONAL

Applying Syntax Testing to the Application Under Test

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Questions

- Does the ADvantage IDE:
  - Accept all valid inputs for Value?
  - Reject all invalid inputs for Value?
  - Provide the user with useful messages when input values are invalid?
  - Have the necessary error-handling implemented to not crash the real-time simulation environment when invalid inputs are entered?

ADI APPLIED DYNAMICS INTERNATIONAL

Applying Syntax Testing to the Application Under Test

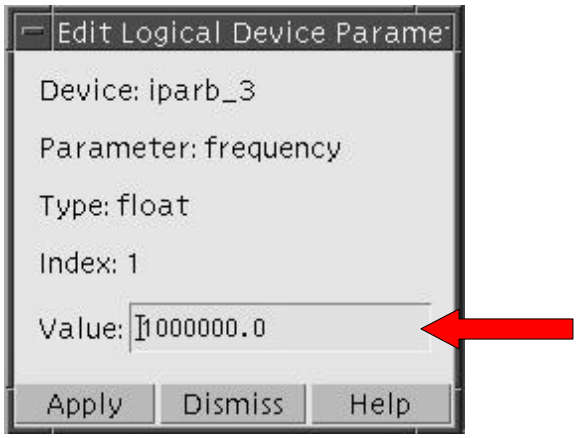
Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion



ADI APPLIED DYNAMICS INTERNATIONAL

Applying Syntax Testing to the Application Under Test

Introduction

Application Under Test

Syntax Testing

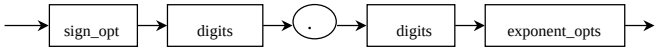
Applying Syntax Testing to the Application Under Test

Conclusion

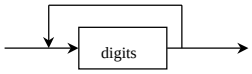
Example Railroad Diagram

VALUE:

real_constant:



digits:



ADI APPLIED DYNAMICS INTERNATIONAL

Applying Syntax Testing to the Application Under Test

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Mutation Table: *Test Cases with Valid Inputs*

Test Case	Input Value
1	3e2
2	+2e+5
3	-6e-7
4	6e-2
5	1234567890e-3
6	0e0
7	1e1
8	2e2
9	3e3
10	4e4
11	5e5
12	6e6
13	7e7
14	8e8
15	9e9

ADI APPLIED DYNAMICS INTERNATIONAL

Applying Syntax Testing to the Application Under Test

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Mutation Table: *Test Cases with Invalid Inputs*

Test Case	Input Value
1	xe0
2	0x0
3	0ex
4	x0e0
5	+xe0
6	ee0
7	+e0
8	000
9	0+0
10	0ee
11	0e+
12	e0e0
13	+ee0
14	++e0
15	e0

ADI APPLIED DYNAMICS INTERNATIONAL

Applying Syntax Testing to the Application Under Test

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Mutation Table: *Test Cases with Invalid Inputs* - *Continued*

Test Case	Input Value
16	00
17	0e
18	y0e0
19	0ye0
20	0ey0
21	0e0y
22	y+0e0
23	+y0e0
24	+0yeo

ADI APPLIED DYNAMICS INTERNATIONAL

Applying Syntax Testing to the Application Under Test

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Sample TSL Script

```
while (getline val < "float_valid.txt")
{
        set_window ("Edit Logical Device Parameters", 5);
        edit_set ("Value:_0", val);
        button_press ("Apply");
        edit_get_block ("Value:_1", 0, 0, 0, 15, info);
        edit_get_block ("Value:_1", 1, 4, 1, 20, apply);
        if (info == ">>>Information:")
                if (apply == "Changes applied.")
                        tl_step ("PASS", 0, val);
                else
                        tl_step ("FAIL", 1, val);
        else
                tl_step ("FAIL", 1, val);
}
close ("float_valid.txt");
```


ADI APPLIED DYNAMICS INTERNATIONAL

Applying Syntax Testing to the Application Under Test

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

Sample TSL Script - Continued

```
while (getline val < "float_invalid.txt")
{
        set_window ("Edit Logical Device Parameters", 5);
        edit_set ("Value:_0", val);
        button_press ("Apply");
        edit_get_block ("Value:_1", 0, 0, 0, 15, info);
        edit_get_block ("Value:_1", 1, 4, 1, 20, apply);
        if (info == ">>>Information:")
                if (apply == "Changes applied.")
                        tl_step ("FAIL", 1, val);
                else
                        tl_step ("PASS", 0, val);
        else
                tl_step ("PASS", 0, val);
}
close ("float_invalid.txt");
```

ADI APPLIED DYNAMICS INTERNATIONAL

Concluding Remarks

Introduction

Application Under Test

Syntax Testing

Applying Syntax Testing to the Application Under Test

Conclusion

- Use of a Record/Replay tool to implement a formal testing methodology (versus "Monkey Testing")
- Bug rate of the Syntax Testing technique as implemented was 5.5 "defect-categories" per hour of testing
- A reusable test suite
- An automated process for syntax regression testing

ADI APPLIED DYNAMICS INTERNATIONAL

Thank you.

Dr. Selim Aissi
Wendi Hummel

Systems Quality Assurance
Applied Dynamics International
Ann Arbor, Michigan

a

TCBeans Test Case Tool Kit

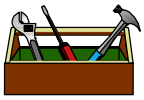
&

GOTCHA Model Based Automatic Software Testing

IBM Haifa Research Laboratory
www.haifa.il.ibm.com
Kenneth Nagin (nagin@il.ibm.com)
Alan Hartman (alan_hartman@vnet.ibm.com)

March 1999

© Copyright IBM Haifa Research Lab 1999. All rights reserved.



Agenda

- Objectives & History
- TCBeans Test Case Tool Kit
- GOTCHA Model Based Automatic Testing
- Future Work
- Case Studies:
  - Hardware test
  - Software test

© Copyright IBM Haifa Research Lab 1999. All rights reserved.

Objectives

- Function Test
  - APIs and Protocols
  - Java, C, C++ applications
- Promote Reuse
- Learn from Hardware Test Experience
- Support testing methodologies
  - Rule Based
  - Modeling
- Increase coverage
- Automate non-creative tasks

© Copyright IBM Haifa Research Lab 1999. All rights reserved.

History

- Hardware
  - CDTG Methodology in HRL/IBM - 1994
  - Murϕ - public domain model checker 1992-1996
  - GOTCHA combines Murϕ technology with CDTG methodology - 1998
- Software
  - Java Distributed API Tester - 1997
    - Java->C->PCI device
  - TCBeans/GOTCHA - 1998

© Copyright IBM Haifa Research Lab 1999. All rights reserved.

Component Based Testing Architecture
TCBeans Tool Kit

- Test Case:
  - JavaBean
  - Runnable Serializable Java Class
- Test Case Assembly
- Runtime Customization
- Simplified Test Execution
- Simplified Test Administration

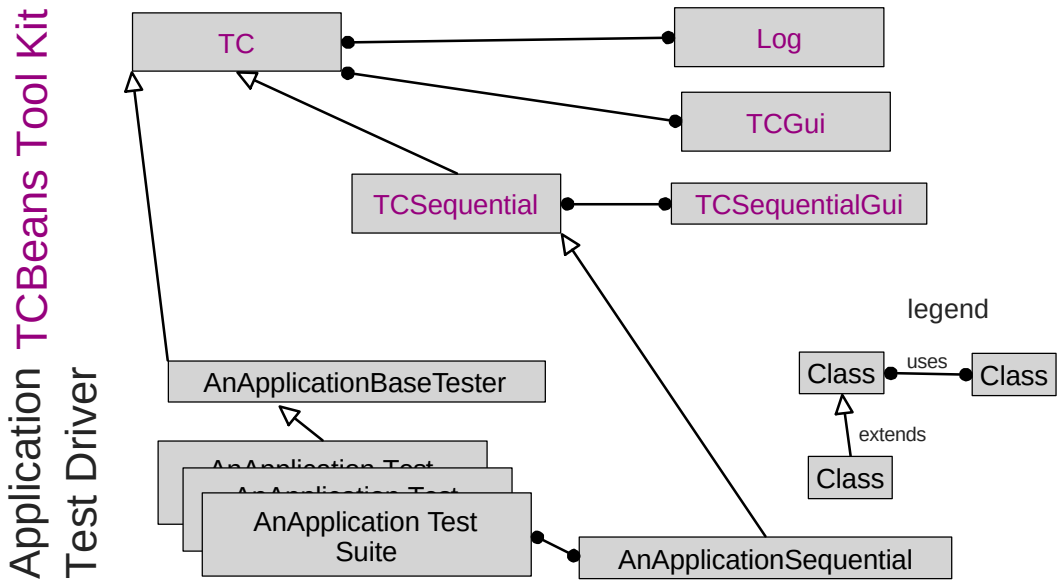
© Copyright IBM Haifa Research Lab 1999. All rights reserved.

Component Based Testing Architecture
TCBeans Test Case

- Reuse via class inheritance
  - runtime interaction
  - time
  - repetition
- Formalized Expected Results Processing
  - invariant
  - post condition
  - assertion
- Versioning

© Copyright IBM Haifa Research Lab 1999. All rights reserved.

TCBeans Software Test Tool Kit



© Copyright IBM Haifa Research Lab 1999. All rights reserved.

TCBeans Software Test Toolkit

A Use Case Scenario

- Tester creates AnApplicationTester which extends TC
- Tester adds properties shared by all tests
- Tester overrides TC's invariant()
- Tester creates a suite of test cases extending AnApplicationTester, overriding run() and postCondition()
- Tester adds test names to TCSequential's TCNames property
- Tester alters shared properties and specialized properties
- Tester saves current version
- Tester runs TCSequential and examines results
- Lots of bugs found Tester celebrates

© Copyright IBM Haifa Research Lab 1999. All rights reserved.

Model Based Automatic Testing

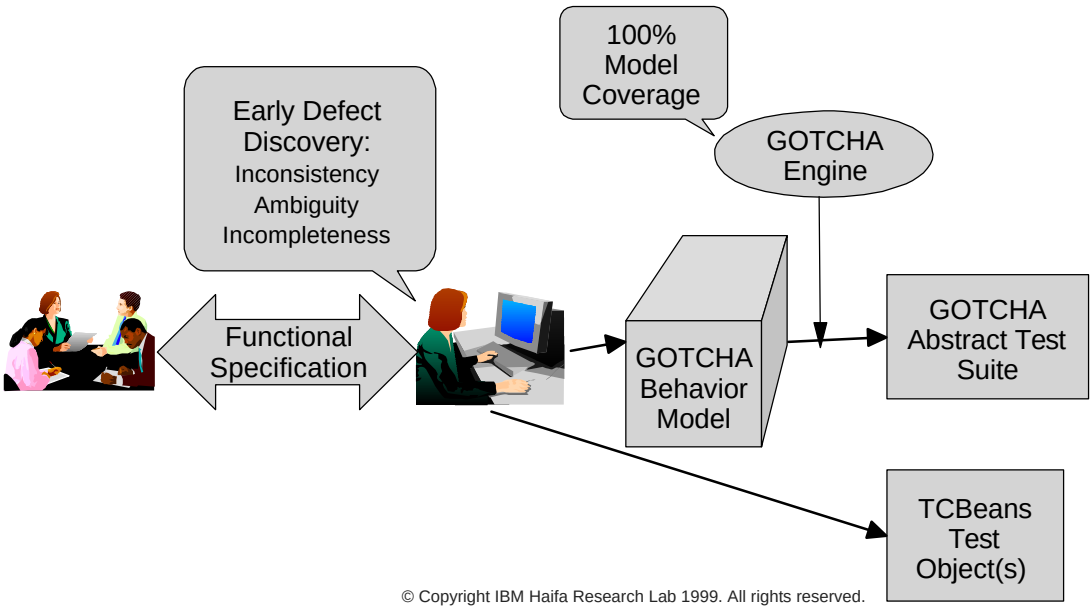
GOTCHA Testing Methodology

- Input: functional specification
- Produce model with GOTCHA compiler
- Generate abstract test case suite
  - Sequences of rules and states (assertions and post conditions)
- Run tests
- Benefits:
  - 100% coverage of model
  - Early Defect Prevention
    - Inconsistencies
    - Ambiguity
    - Incompleteness

© Copyright IBM Haifa Research Lab 1999. All rights reserved.

TCBeans/GOTCHA Software Testing Methodology

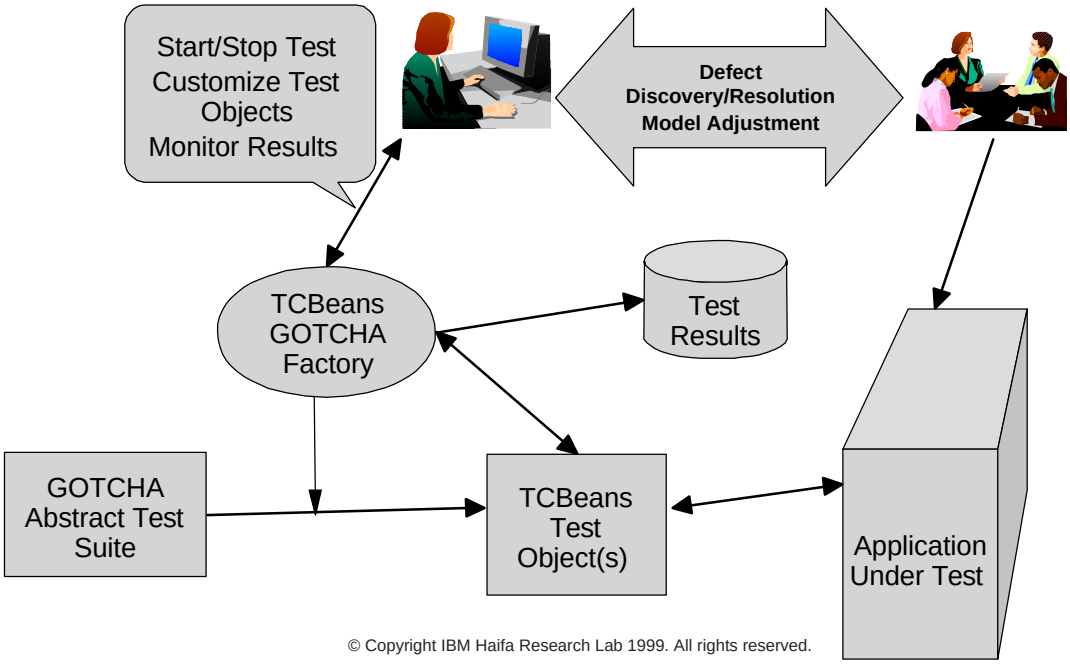
Test Design



© Copyright IBM Haifa Research Lab 1999. All rights reserved.

TCBeans/GOTCHA Software Testing Methodology

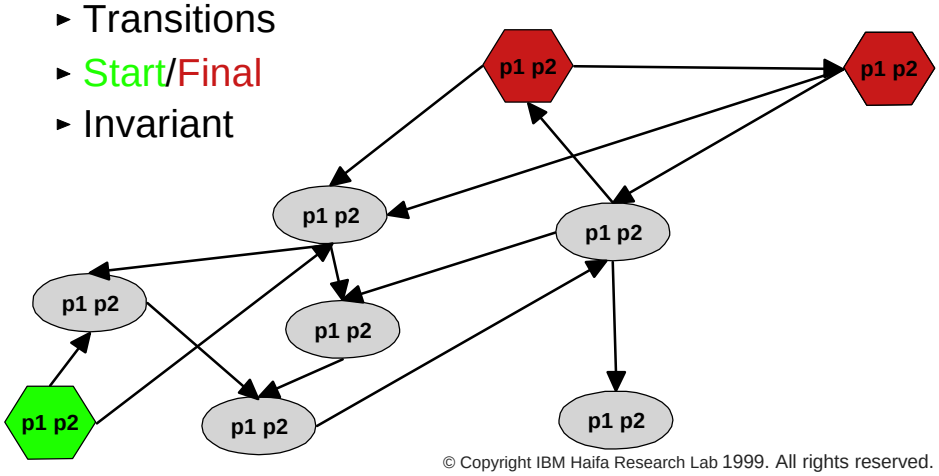
Test Execution



© Copyright IBM Haifa Research Lab 1999. All rights reserved.

The GOTCHA Definition Language

- Used to define a finite state machine
  - States
  - Transitions
  - Start/Final
  - Invariant



© Copyright IBM Haifa Research Lab 1999. All rights reserved.

GOTCHA Suite Generation

- Input -
  - GOTCHA definition language model
  - A coverage model - projected state/transition
  - A definition of start and final states for tests
- Output
  - A test suite guaranteed to cover all coverable tasks
  - A list of reachable but uncoverable tasks
- Test
  - An execution path from a start state to a final state
- Byproduct
  - Violated invariance encountered

© Copyright IBM Haifa Research Lab 1999. All rights reserved.

Model Based Automatic Testing

TCBeans GOTCHA Test Case Factory

- JavaBeans Test Case Component
- Input: GOTCHA abstract test suite (text)
- Output: concrete test case execution + results
  - no additional source files
  - no explicit compile and link phase
  - 100% coverage of model
- Target and Expected object (JavaBeans)
  - rules = target method invocation
  - states = expected property setting
- Process
  - parse test suite
  - execute
  - compare Target and Expected properties
- Interactive customization with BDK

© Copyright IBM Haifa Research Lab 1999. All rights reserved.

Model Based Automatic Testing

A Use Case Scenario

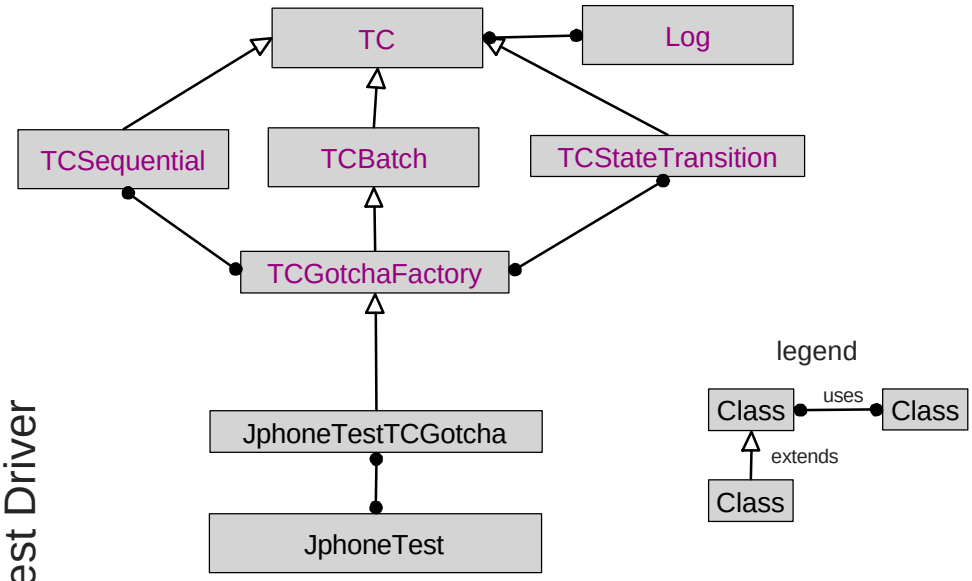
- Tester reviews functional specification
- Tester codes a behavior model in GOTCHA
- Tester uncovers defects in specification
- Tester runs GOTCHA to create abstract test suite
- Tester codes a Java class (JavaBeans)
  - wrapper methods to application methods
  - wrapper properties to application state variables
  - static properties for interactive customization
- Tester runs TCBeans GOTCHA Test Case Factory
- Tester customizes Target properties
- Tester runs test, uncovers lots of bugs, celebrates
- Analyses structural coverage and adjust GOTCHA model

© Copyright IBM Haifa Research Lab 1999. All rights reserved.

TCBeans Software Test Tool Kit

TCBeans Tool Kit

Application Test Driver



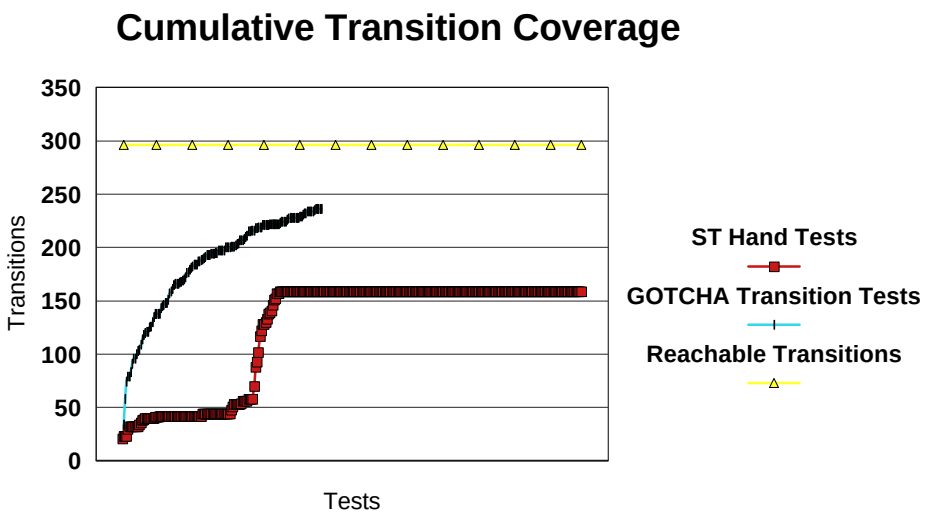
© Copyright IBM Haifa Research Lab 1999. All rights reserved.

Future

- Distributed testing support
- Concurrent testing support
- Better imbedded BDK support
- Interactive Method Invocation
- Strengthen GOTCHA ties and support
  - Software test mission
  - Object Oriented programming, e.g. Java like
- Data Domain modeling/testing
- Formalize test results reports
  - UML browser for results and GOTCHA test suite
- Coverage analysis
- Additional ease of use features

© Copyright IBM Haifa Research Lab 1999. All rights reserved.

GOTCHA Hardware Test Study



Case Study

- HOD V4.0 Macro (on going)
  - Defects:
    - 2 x Severity 1
    - 9 x Severity 2 (3 from previous release)
    - 6 x Severity 3
    - Many due to Specification Review
- JavaPhone to JavaPhone model (constrained)
  - Defects:
    - 9 x Severity 2
    - 10 x Severity 3

© Copyright IBM Haifa Research Lab 1999. All rights reserved.

IBM Research Laboratory in Haifa
Matam, Haifa 31905, Israel
Tel 972-4-8296211
Fax 972-4-855070
www.haifa.il.ibm.com

TCBeans/GOTCHA Software Test Tool Kit
Quality Week 1999

© Copyright IBM Haifa Research Lab 1999. All rights reserved.

Kenneth Nagin (nagin@il.ibm.com)
Alan Hartman (alan_hartman@vnet.ibm.com)
March 1999

What is the Tool Kit:

The TCBeans Software Test Tool Kit is a framework designed to assist testers in developing, executing and organizing function tests directed against Application Program Interfaces (APIs) and software protocols written in Java, C or C++ . Test time is saved because the tool kit supports reuse of many common testing functions. Test quality and complexity is enhanced because the tool kit supports a systematic approach to test suite generation. The systematic approach, in turn, yields higher function and code coverage, uncovering more defects early on in the software development cycle. Testing is more interesting since non-creative tasks associated with testing are automated, shielding against the high attrition rate associated with testing personnel.

Why invest in another test tool:

Tools for testing a particular application are often developed on an ad hoc basic. Each project develops its own tools and framework to solve its particular testing needs. Usually a tool begins as a simple interpreter that can execute a limited set of APIs. The test case commands are submitted interactively or by way of a batch text file. With time, the test tool becomes more complicated as it tries to solve more and more problems, e.g. concurrent and parallel testing. However the primary focus of the project is not the tester so resources are scarce and often inadequate to satisfy all the testing requirements. This ad hoc approach to test tool development is expensive. It is preferable to develop a framework that could be used by many test organizations to amortize the cost and create a more powerful, general-purpose tool set.

Test tools do exist in the market. However these tools are either extremely expensive or not intended for testing APIs and protocols. Automatic Test Suite generators like Teradyne's TestMaster[1] start at $50000 + $20000 for instruction. Less expensive record and replay test

[1] TestMaster see http:/www.teradyne.com/sst/

tools like Mercury Interactive's WinRunner[2] are designed to assist in testing Graphic User Interfaces, not APIs or protocols.

There are two reasons to support an API, even when an application's current market requirement does not include a programming interface. The first is directly related to the testing of the application, and the second relates to its future market requirements. Most present day applications contain a lot of function, supported by a large code base. The functions interact with themselves and the outside world. The manual approach to testing simply can not exercise the complex behavior inherent in the function interactions or cover many code paths beyond those of the main function paths. Furthermore, a good API extends the utility and life span of an application, since it allows the application to be used as a component in other applications. In both cases, a programming solution is required, meaning that the application must support an API.

How does it work:

We present a novel approach to software testing by treating a test case as a software component that can be reused, customized and assembled off-line during code development, or dynamically at run time. Specifically, we treat a test case as a JavaBeans[3] component which can be customized using a Bean Development Kit (BDK)[4] or during actual testing by loading the test cases within the tool kit's run time testing framework which supports the functions expressed in a BDK. Reusability is gained primarily through the mechanism of class inheritance. Basic functionality shared by most test cases is supported in a super class JavaBeans test case which testers extend and specialize for their own special test environment.

We automate software testing by exploiting the characteristics of JavaBeans to bridge the gap between an abstract test suite and concrete test execution. Typically an abstract test suite is a text file describing a set of test cases. The test cases describe the procedure to exercise a software application being tested. The test case procedure may simply be a list of execution statements or in some cases the execution statements are accompanied by clauses to determine the correctness of the execution. The abstract test suite may be generated manually, or automatically using an automatic test suite generator, e.g. GOTCHA [5]. In either case, the abstract test suite must be translated into object code in order to actually execute the test. Typically the text is parsed into source code and recompiled for execution. Alternatively, specialized parsers are designed for the specific software application being tested. The parser examines the test suite and executes the test against the software application. However, we exploit the characteristics of JavaBeans to create a general-purpose parser that directly executes the test case procedures and determines the correctness of the execution.

What is GOTCHA?

GOTCHA is the tool kit's engine for generating an abstract test suite for a finite state machine (FSM) driven by a coverage model. The finite state machine is described in a high level

---

[2] WinRunner see http:/www.merc-int.com/products/winrunguide.html

[3] JavaBeans see http://www.javasoft.com/beans/index.html

[4] BDK see http://www.javasoft.com/beans/software/bdk_download.html. Most of the leading Java Interactive Development Environments (IDEs) support their own BDK. Some leading commercial Java IDEs are IBM's Visual Age for Java, Symantec's Visual Cafe, Borland's Jbuilder, Microsoft's Visual J++, etc.

[5] GOTCHA Generation of Test Cases for Hardware Architectures, see /afs/haifa.ibm.com/u11/hartman/public/gotcha_release1.2KN.htm, http://www.leda.fr/genevieve.htm

language for modeling concrete systems. Such systems may be hardware architectures or components, software systems, communication protocols, or other complex systems and processes. A test case is a sequence of stimuli for the model. The test suites generated by GOTCHA guarantee 100% coverage of the coverage model specified by the user. The GOTCHA prototype was developed for hardware architecture models but we have exploited it for modeling and testing software systems. The coverage models currently supported by GOTCHA are projected FSM state and transition coverage.

The GOTCHA tool uses a variant of the hardware verification methodology introduced by R. C. Go, C. Han Yang, M. A. Horowitz and D. L. Dill[6] . This methodology contains four basic phases: (1) describing the processor implementation control as a Finite State Machine, (2) deriving an abstract test suite guaranteeing coverage of the FSM model using methods from formal verification, (3) automatic translation of the covering tours to test vectors, (4) translating the test vector to executable tests.

The use of GOTCHA in the context of automated software testing is slightly different than its hardware counterpart. In phase 1, the behavior of a software application is described by the software functional specification and other user requirement documentation. In the hardware context, the processor implementation is described in a hardware definition language like VHDL or Verilog. The software tester translates the specification into a FSM model by coding in the GOTCHA definition language, a Pascal like derivation of the Murϕ[7] Description Language. In the hardware context, the VHDL is translated automatically into the Murϕ Description Language with an automated extraction of the control signals. The software model builder constrains the size of the test suite by specifying which state variables must be covered. The size of the FSM being modeled is controlled by the level of abstraction used in the manual modeling process. In phases 2 and 3, the GOTCHA tool generates the abstract test suite insuring that all desired states and/or state transitions are covered. The test generation algorithms used by GOTCHA are different from those used by Dill's abstract suite generator and provide for a larger range of user defined constraints on the test suite. The final phase which is only briefly discussed in the original hardware verification methodology is the subject of the TCBeans GOTCHA Automatic Software Test Factory (see below).

GOTCHA modeling can begin as soon as a functional specification is prepared. Our experience shows that the analysis required to generate a model uncovers defects in the specification that escape the usual review process. Specifically, specification defects related to ambiguity, inconsistency and incompleteness are discovered due to the systematic analysis required when building the GOTCHA model. We also note that several GOTCHA models may be built at varying levels of abstraction and specialization. A more detailed model implies a larger FSM to be explored, stretching the limits of the underlying formal verification technology. Like wise, several specialized GOTCHA models can be designed each intended to explore specific aspects of the application being tested. In the hardware context, only a single model is built automatically extracting the control FSM from the hardware description. This can easily lead to state explosion problems. The ease with which more detailed or more specialized GOTCHA models are generated and deployed is exploited to make the test more responsive. In the initial

---

[6] "Architecture Validation for Processors" in ACM Isca 1995 by R. C. Ho, C. Han Yang, M.A. Horowitz, D. L. Dill

[7] "The Murϕ Verification System", David L. Dill:. CAV 1996: 390-393

stages of test, the software being tested is unstable, i.e. more serious defects are discovered. These defects impede testing since they prevent the exercise of other code and function near the defective areas. Specialized models and GOTCHA test constraints can be exploited to avoid the defect path and allow testing to continue. As the test progresses, defects are fixed and more code paths may be explored. Testers can respond by deploying more detailed models, and increasing code and function coverage as the application matures.

The GOTCHA test suite may contain a multitude of tests that cover the application's complex behavior. The tester creates a JavaBean test object which maps the GOTCHA test suite to the application being tested. The tool kit supports an automatic test function called TCBeans GOTCHA Automatic Software Test Factory (see below) which executes the GOTCHA abstract test suite against the JavaBean test object. Once the application is ready for test, the whole test suite can be run automatically or the tester can select particular tests to run.

A frequently used metric for the quality of a test suite is to gauge how much of the code and/or paths through the code are covered by the test suite. Hardware studies[8] have shown that FSM state and transition coverage achieves high levels of branch and statement coverage of hardware definition languages. We still need to verify whether similar results can be achieved in software testing. However, we have already seen that the test suites generated exercise paths through the application under test that would typically be ignored by a less systematic approach.

Why Java?

Test cases, test objects, and the tool kit are coded in the Java programming language. Ideally the same test code can run on all target machines. The Java programming language is extremely portable since it runs on all systems that support a Java Virtual Machine and most major computer vendors support Java. When the application being tested is written in C or C++ code, Java provides a standardized procedure for accessing the C or C++ code called native implementation. This means that the native portion of the JavaBeans test objects may have to be ported between different platforms.

What is a TCBeans Test Case?

A TCBeans Test Case is a JavaBeans component that extends the Runnable class and implements Serializable. We exploit the JavaBean properties to simplify reuse, customization, run time interaction, test execution, test administration, and test assembly:

- Reuse: Test case reuse is gained primarily through the mechanism of class inheritance. Basic functionality shared by most test cases is supported in a super class which testers extend and specialize for their own special test environment. The super class, TCBeans test case, supports the following functionality, which may be reused by its subclasses. (note: The test case's associated GUI (see below) supports run time interactive with the supported functions)
    - It can start itself running as a thread.
    - It can stop itself from running upon command or based on some wait time property.
    - It can iterate the number of runs per start.
    - It can notify to the outside when it completes and the results of its execution. The tool kit supports central logging to write the results.

[8] "A Study in Coverage Driven Test Generation", M. Benjamin, D. Geist, A. Hartman, G. Mas, R. Smeets, Y. Wolfsthal, September 1998, submitted for publication.

- Three methods are supported that formalize the test process and promote reusability. They are intended to be overridden by a subject subclass:
  - *run():* The run() method is invoked as a thread. The user's subclass inserts test logic here, e.g. it invokes methods belonging to the object under test or changes the state of the object under test in some way.
  - *postCondition():* The postCondition() method is invoked when the run() method completes. The user's subclass inserts test verification logic here, e.g. it determines whether the state of the object under test is as expected. The postCondition() determines the success or failure of the test performed by the run() method.
  - *invariant():* The invariant() method is invoked prior to invoking the run() method and after the run() method completes. The user's subclass inserts test verification logic here that should always be true for the object under test regardless of any particular run() method testing logic. Invariance promotes reuse within a particular test suite.
- It can load and instantiate versions of itself, other test cases or resources that the test cases requires. The test cases may be in their raw class form, serialized form, or wrapped in visual components created within other BDK environments. JavaBeans test cases created within other BDK environments are usually housed within some other visual JavaBean component, e.g. Frame, and are not directly accessible so a Wrapper function is required to point to the subject JavaBean. The java.beans package is exploited to achieve interactive loading and instantiation of a JavaBeans Test Case.

- It can save and restore versions of itself, other test cases or resources to and from the file system. This is referred to as Java Serialization.

• Test case customization: TCBeans test case properties, i.e. variables, may be modified prior to testing or during testing. Prior to testing, the properties may be customized interactively using a commercial Java BDK or simply by extending the test case class and altering the properties. During actual testing, the properties may be modified programmatically since properties are expressed as public members (public setter and getter methods). Or they may be modified interactively with a run time testing framework, which supports the functions expressed in a BDK. Since a test case is serializable a particular version can be saved and reloaded from a host file system.

- Run time interaction: TCBeans test cases are non-visual JavaBeans components however testers need to interact with the test cases during testing. Thus, a test case has a Graphic User Interface (GUI) with which it is associated. The GUI must support customization of a test case's properties and the ability to activate its methods. Reuse is supported by the following innovations:
  - JavaBeans Test Case properties and test case resources may be customized using the BDK's property sheet supported by the tool kit. This means that a test case writer need not create special GUI components in order to customize itself and associated resources. Rather test cases express their public variables using JavaBeans property design patterns and they automatically receive interactive customization support from the tool kit.

  Note: BDKs are commonly used to customize JavaBeans prior to execution. The customized JavaBeans are then shipped and executed in its "frozen" version. In

addition, BDKs are primarily used to support visual components, e.g. windows, buttons, etc.. However, we extend this common practice since the tool kit is a testing run time environment whose purpose is to customize non-visual components, i.e. JavaBeans Test Cases.

- Methods belonging to a TCBeans test case and associated resources may be deduced and invoked using functions supported by the tool kit. This means that a tester need not create special GUI components in order to invoke its methods, rather it declares the methods as public and the tool kit supports a mechanism with which the tester interacts with the methods and their parameters during testing. The Java reflection feature enables the framework to support the method interaction.

The TCBeans test case super class supports a public method that invokes the GUI associated with the test case. The method deduces the name of the GUI class by requiring that its associate GUI follow the design pattern that the GUI's name is the TCBeans test case name appended with the string "Gui". The method invokes the GUI class that follows the design pattern. If none exist it invokes its super class based on the same criteria. It continues searching for a match until one is found. Of course the framework includes such a GUI for the TCBeans test case super class.

- Test Execution: TCBeans test case execution is simplified since JavaBeans are simply Java classes, and these classes can by dynamically loaded from the file system using a Java class loader, a common Java function.

- Test Administration: Test administration is simplified since TCBeans test cases are individual Java files. The test cases can be managed organized and archived like any standard file. Also JavaDoc[9] provides a simplified and standardized documentation facility that can be used to easily document the test cases which compromise a test suite.

- TCBeans Test Case Assembly: Test cases are often assembled into groups. The test cases may directly associated with each, e.g. a step in a series of processes. Or they may be independent members of a test suite. In either case it required to assemble the test cases together. Two subclasses of the JavaBeans Test Case are introduced to facilitate test case assembly:
  - Sequential: The Sequential class supports running a vector of TCBeans test cases sequentially, i.e. one completes before the next starts.
  - Concurrent: The Concurrent class supports running a vector of TCBeans test cases concurrently, i.e. they all start together.

Both classes provide mechanisms to manipulate the subject vector, e.g. add or remove test cases.

What is the TCBeans GOTCHA Automatic Software Test Factory?

The TCBeans Automatic Software Test Factory is a component in the tool kit which extends the functionality of the TCBeans test case. It bridges the gap between the GOTCHA abstract test suite and the JavaBeans test objects created by the tester. GOTCHA generates an abstract test suite that is a text file describing a set of test cases. The test cases describe the assertions (input/stimuli), called rules, that exercise the application being tested followed by post conditions which express the state of the abstract software model after a series of assertions.

TCBeans test objects are Java classes that publicize their variables as properties. These properties are mapped to the abstract test suite post condition state variables. The TCBeans test object methods are

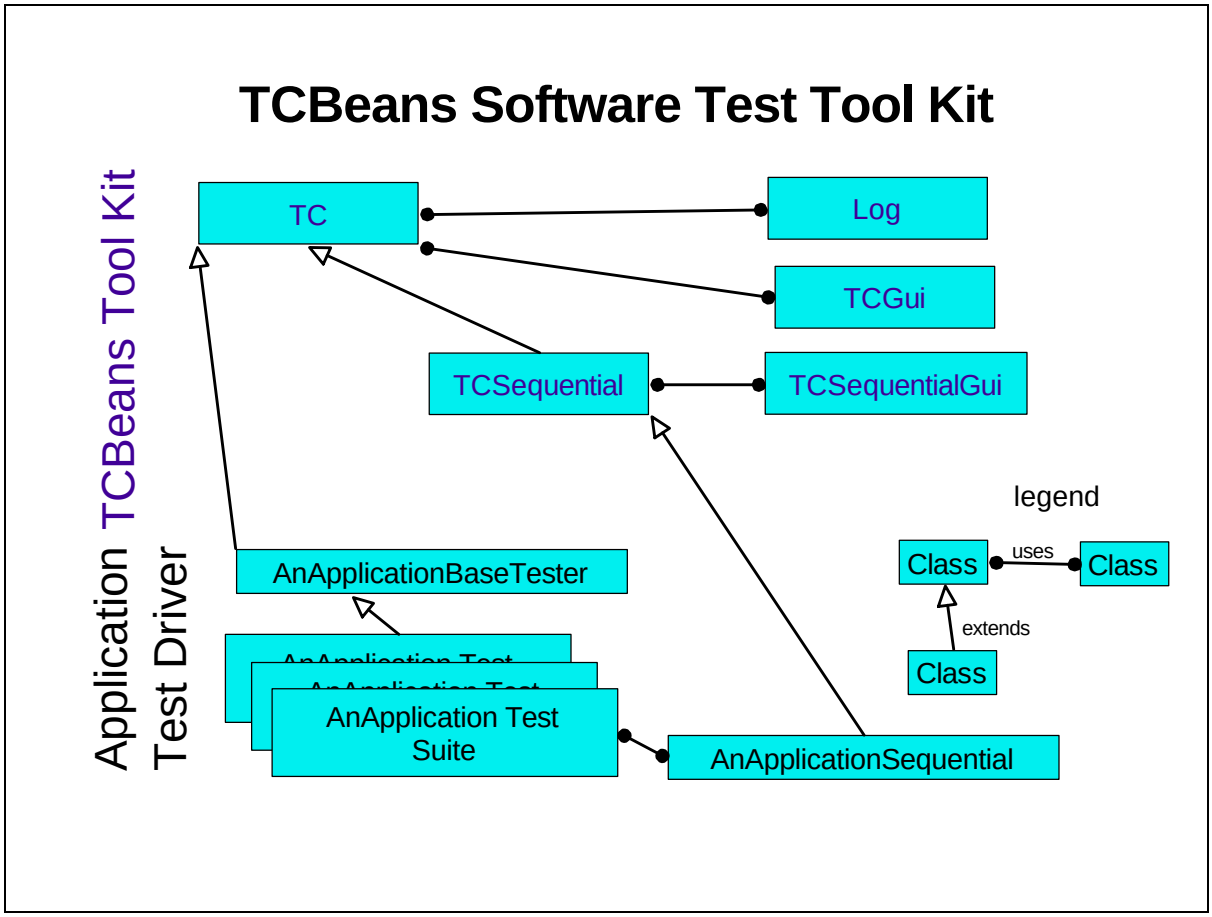
[9] JavaDoc see http://www.javasoft.com/products/jdk/javadoc/index.html

mapped to the abstract test suite assertions. The methods exercise the application being tested and the properties express the application's state.

The factory parses the abstract test suite by translating the assertion statements and post conditions to JavaBeans test objects' methods and property settings, i.e. assertion statement -> method and post condition -> property setting. The objects' methods are invoked in the order prescribed by the abstract test suite. The objects' property settings are compared against the abstract test suite's post conditions. If the post conditions and property settings are equivalent the test succeeded otherwise it fails. It is worth noting that there is no explicit compile and link phase between parsing the abstract test suite and executing the test. This means that there are no additional class files to administer.

Examples:
The examples below illustrate how the tool kit is exploited to build a test suite. TC is the super class of all test cases in the tool kit. It provides common function shared by most test cases, e.g. interactive starting and stopping of itself, centralized reporting to a log, etc. Another class frequently used is TCSequential. TCSequential extends TC and supports a property called TCNames which may be customized, online or offline, to reference the class names of test cases in a test suite. TCSequential runs the test cases listed in TCNames sequentially and accumulates their results.

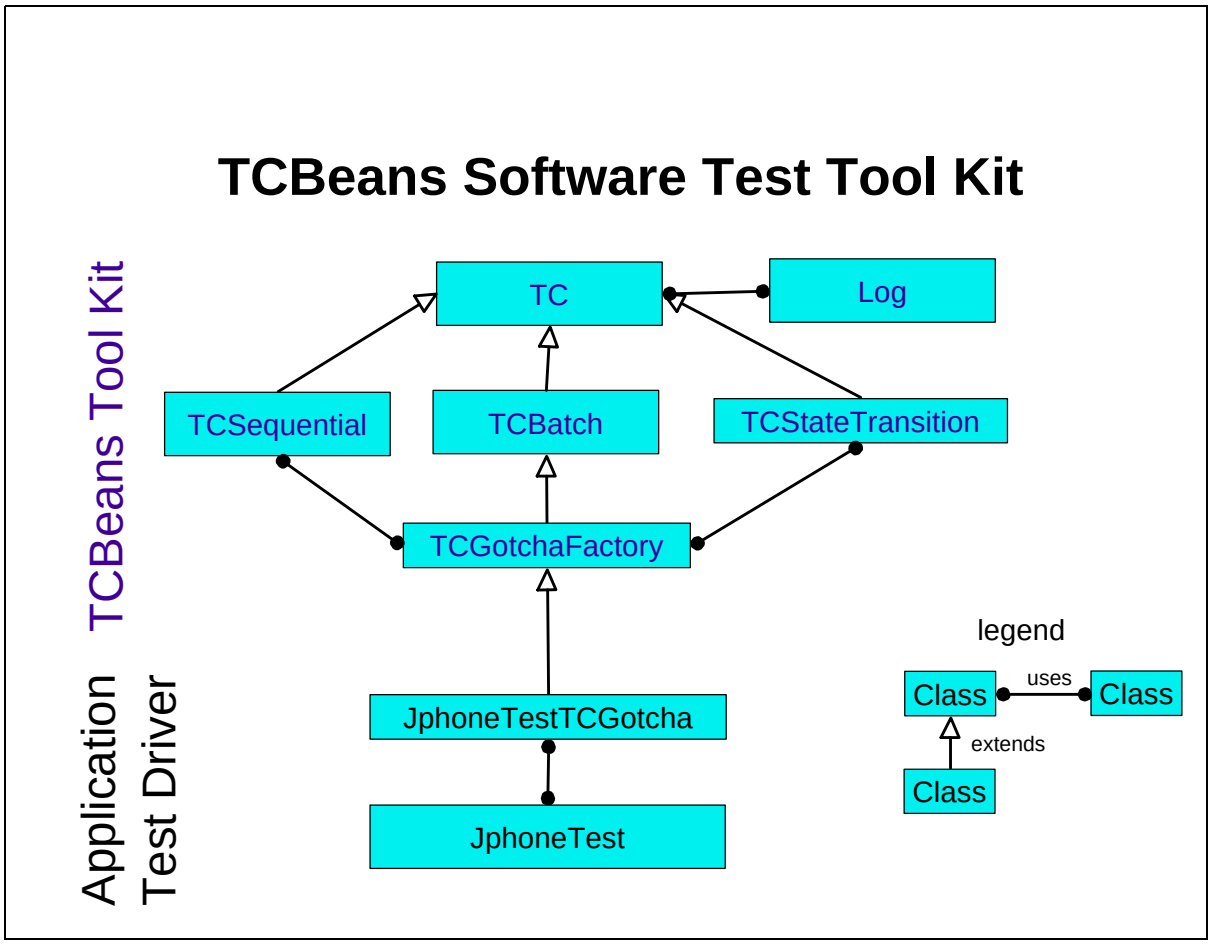
AnApplication Test Suite



1) Manual Test Suite Development with TCBeans Tool Set:

In the figure *AnApplication Test Suite* we illustrate how a tester reuses the TCBeans Software Test Tool Kit to manually create a test suite. We describe the test driver for an example application, AnApplication. The AnApplication test suite contains a set of methods that invoke AnApplication functions. AnApplicationBaseTester extends TC. AnApplicationBaseTester contains functions and properties common to all of the AnApplication test suite. Most importantly it overrides the TC's *invarient()* method with application verification logic common to all of the test suite. The rest of the test suite extends AnApplicationBaseTester. Each test overrides TC's *run()* and *postCondition()* to reflect its particular test logic. The test suite is assembled into the AnApplicationSequential which extends TCSequential by customizing TCNames with names of the AnApplication Test Suite.

TCGotchaFactory Usage



2) Automatic Software Test with TCBeans GOTCHA Factory:

In the figure *TCGotchaFactory Usage* we illustrate how a tester reuses the TCBeans Software Test Tool Kit to create an automatic software test driver for a GOTCHA test suite. We describe the actual test driver for the IBM's JavaPhone[10]. The JavaPhone test driver consists of only two classes, JavaPhoneTest and JavaPhoneTestGotcha. The rest of the classes are provided by the tool kit, i.e. reused. The important point here is that the tester writes only code that is specific to the application being tested. The reporting mechanism, the graphic user interface, etc. are all provided by the tool kit.

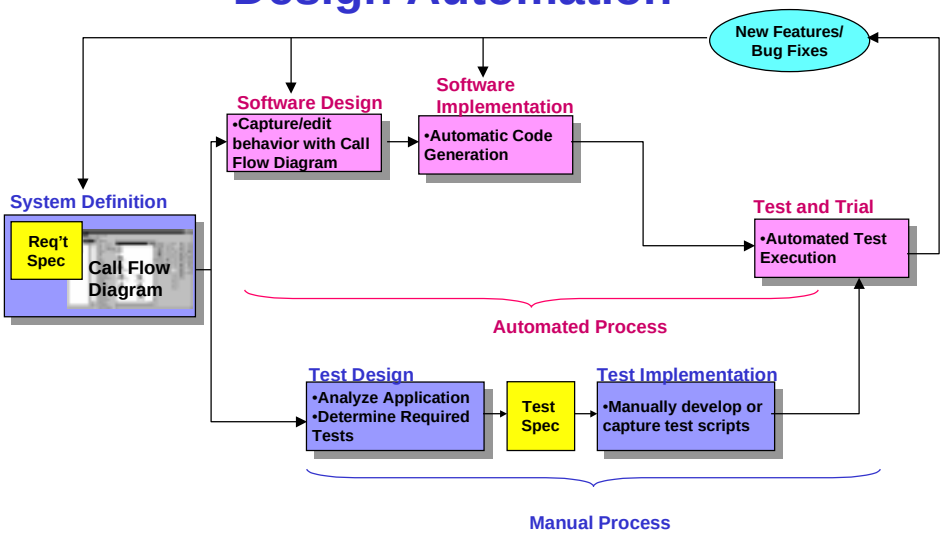
[10]JavaPhone see http://www.haifa.il.ibm.com/javbro.htm

The class JavaPhoneTest contains a set of methods that invoke JavaPhone functions and a set of properties that reflect the JavaPhone current state. JavaPhoneTestGotcha extends TCGotchaFactory. It is only a few lines of code which instruct TCGotchaFactory to focus on JavaPhoneTest as its test object. TCGotchaFactory understands the special format of the GOTCHA abstract test suite and executes it against its test object, e.g. JavaPhoneTest. TCGotchaFactory extends TCBatch and references TCSequential and TCStateTransition. TCBatch parses any text file and executes it methods against a specified test case. TCSequential runs a set of test cases in sequence and accumulates their results. TCStateTransition executes methods, sets properties and compares objects. TC is the super class of all the above TC* classes.

Conclusion:

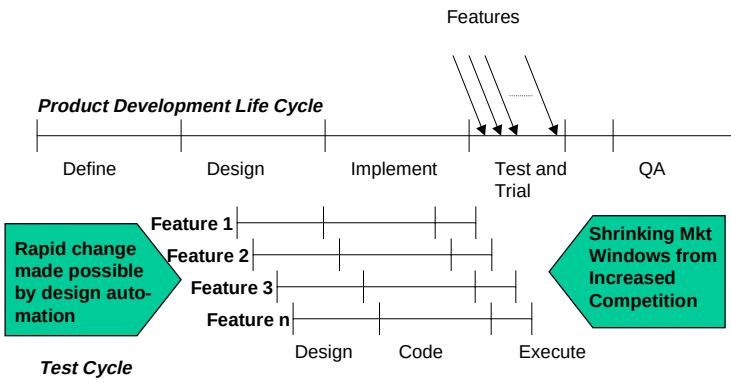
The TCBeans Software Test Tool Kit exploits JavaBeans and GOTCHA to assist testers in developing, executing and organizing function tests directed against Application Program Interfaces (APIs) and software protocols written in Java, C or C++ . The tool kit's component architecture expressed as JavaBeans components save test development time by supporting reuse of many common testing functions. GOTCHA FSM modeling building and simulation yield high quality test suites, that cover the complex behavior of the application being tested. In addition, GOTCHA automates non-creative testing tasks, and thus makes testing more interesting.

Test Process Automation Lags Design Automation



TERADYNE Software & Systems Test

Test Development Process Squeezed by Rapid Change and Shrinking Market Window

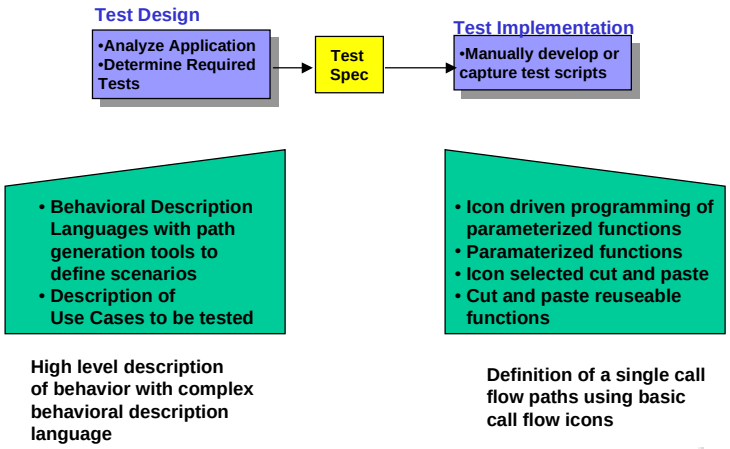


Result:

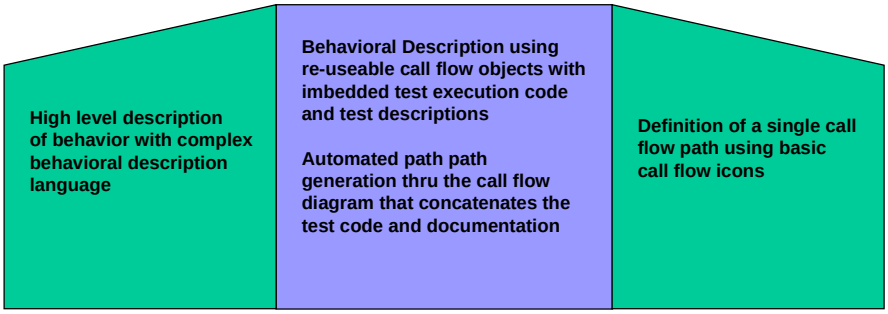
1. *Difficult to keep pace with new features/changes, making test the bottleneck*
2. *Little time for testing feature interaction until late in process, extending the time required for QA and field trials*

TERADYNE Software & Systems Test

A Gap Exists Between Automation Approaches For Test Design and Test Implementation



Closing the Automation Gap



Implementing an Automation Solution

- Define a set re-useable objects
- Define approach for defining the connection of the objects
- Define methodology/algorithm for generating optimal test paths
- Define test output formats

TERADYNE Software & Systems Test

Requirements for Re-useable Objects

- Re-useable across applications
- User can configure without low level coding
- Understandable by non-programmers
- Should be easy to map to icons used by App Builder Tools

TERADYNE Software & Systems Test

Re-useable Call Flow Objects Must Describe Application Behavior at User Interface

Call Flow Object	*Description*
1. Prompt-Response	play a greeting and receive a response
2. Play Voice	play an audio file
3. Place Call	make a phone call
4. Record Voice	save voice data in a file
5. Go On-Hook	disconnect the call
6. Go Off-Hook	prepare to accept incoming calls
7. Send Tones	send MF or DTMF tones
8. Receive Tones	receive MF or DTMF tones
9. Wait For Hangup	wait for call to be disconnected
10. Wait For Call	wait for an incoming call
11. Wait For Energy	wait for energy on line
12. Wait for Silence	wait for silence on line
13. Recognize Speech	compare speech data with a vocab
14. Reject Call	reject incoming call on the line
15. Transfer Call	transfer a call to another number
16. Error	default error handler
17. Stop Channel	stop communications on the channel
18. Send Fax	send a fax
19. Receive Fax	receive a fax

TERADYNE Software & Systems Test

Specialized Test Call Flow Objects May Be Required

- **Dialing ahead of prompts**
- **Measurement of response time**
- **Capture and processing of performance data**
- **Importing database information**
- **Verification of screen pops**

TERADYNE Software & Systems Test

Contents of a Call Flow Object

- Code for executing action on target test execution system or systems
  - Must be parameterized to allow programming of different digits, voice prompts, telephony parameters, phone lines, etc.
  - Includes error handling routines
- Input parameter definition
- Test description for documentation purposes

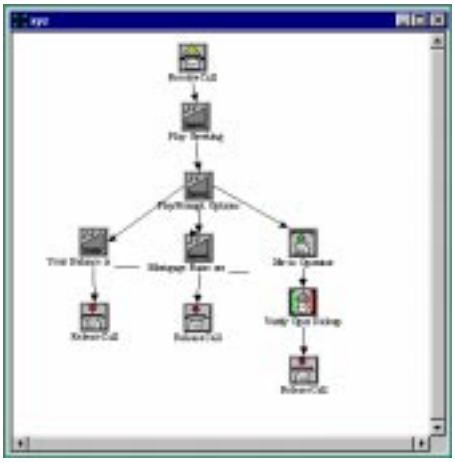
TERADYNE Software & Systems Test

Specifying the Connection of Call Flow Objects

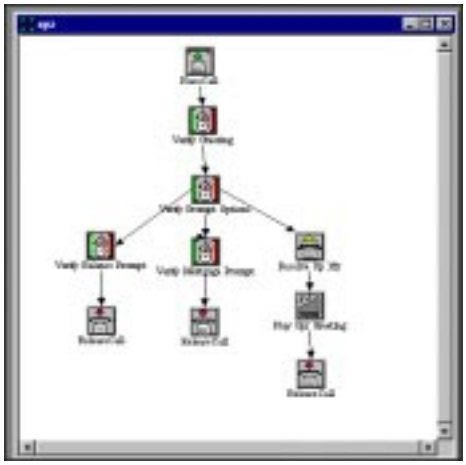
- Best case, integrate with App Builder tool directly.
- At minimum, non programmer must be able to easily create a new description from a design call flow diagram
- Must be able to specify conditional branching depending on various data values
- Support of hierarchical call flows enables greater re-use and more readable descriptions

TERADYNE Software & Systems Test

An Application View Is Required for Easy Mapping to Design Call Flow Diagram



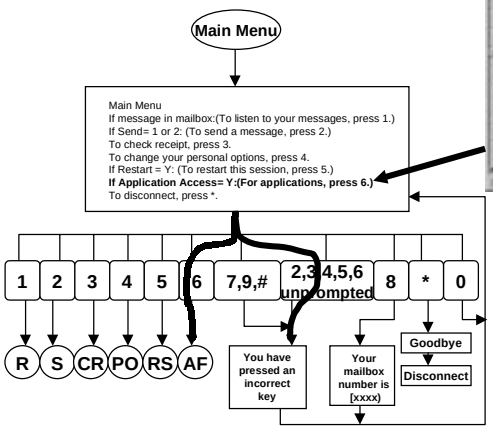
Application View



Test View

TERADYNE Software & Systems Test

Call Flow Descriptions Need to Handle Data Dependencies

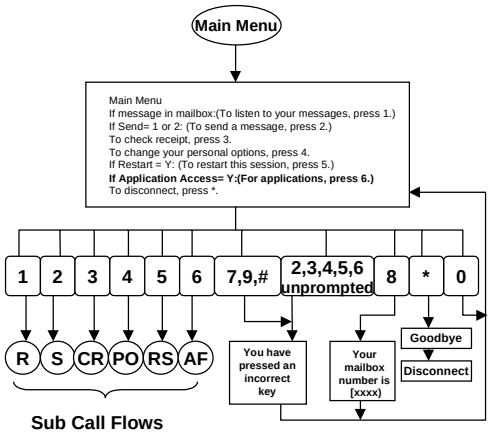


Path thru call flow diagram depends on data values

TERADYNE Software & Systems Test

Hierarchy is Required for Modeling Complex Call Flows

- Allows problem to be decomposed to modular components
- Work can be distributed and done in parallel
- Facilitates re-use of sub call flows across platforms
- Changes to sub-call flows used multiple times are made in one place



Test Generation Requirements

- Touch all parts(data and call flow objects) at least once
- Test for response to negative behavior
- Verify that illegal conditions can't occur

Test Generation Methods

- Manually Determine data and call flow paths required to achieve test requirements
  - Construct each path separately with call flow objects and automatically generate test code for path
- Use path generator tool to find paths thru the call flow diagram and automatically create tests

TERADYNE Software & Systems Test

Path Generation Tools Can Provide Significant Advantages

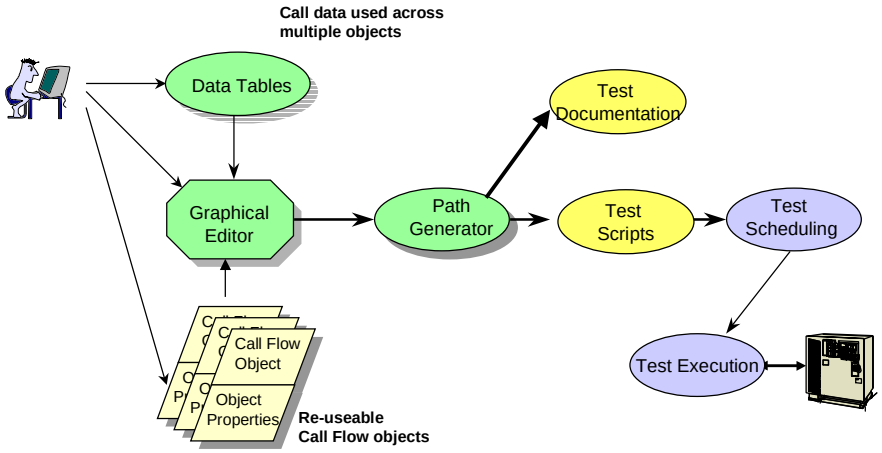
- Advantages
  - Automatically create paths to cover all objects
  - Rapidly respond to changes in call flow
  - Lower skill level required to design tests
  - Can be directed to generate large test sets for problem parts of the call flow
- Key Issues:
  - Are machine generated paths valid scenarios?
    - Do they reflect actual operational scenarios?
    - Do they adequately test error conditions?
  - Once tests are created, should users be able to manually edit the low level code, or forced to only make changes to re-useable objects?

TERADYNE Software & Systems Test

Test Documentation Requirements

- **Test scripts that are generated should be self documenting**
  - **eliminates error prone separate step of creating documentation**
- **Need graphical representation of each test path**
  - **Overall sequencing of call flow objects**
  - **Relative sequencing of call flow actions occurring on each test channel**

Integrated Solution for Automated Telephony Test Generation



Automated Test Generation for Computer Telephony Systems

Brian C. Miller
Teradyne Software and System Test
44 Simon Street, Nashua, NH 03060
miller.brian@teradyne.com

Introduction

Significant progress has been made in automating the front-end application design process and back-end test execution for today's Computer Telephony applications. Sophisticated application builder tools coupled with configurable Interactive Voice Response (IVR) platforms allow end users, integrators, VARs or dealers to build sophisticated IVR applications. Robust automated test execution systems can automatically emulate protocols and voice to greatly increase the thoroughness and repeatability of functional test, load testing, regression testing and in-service monitoring[1]. Despite the automation in design and execution, the process for designing and implementing tests is still mostly manual and largely unchanged from what was done a decade ago. *What's missing is an automated test generation process that integrates the upfront call flow descriptions of the App Builders with the back-end automated test execution environments*. This paper will examine the issues with the current process and discuss the requirements for an automated approach.

Current Design and Test Process

Design Process

Figure 1 illustrates a typical development process for a Computer Telephony system. Today, most Computer Telephony systems are defined in some form of Call Flow diagram. This is done with either a graphical drawing tool, or the graphical editor of an application builder toolset. Once the definition is complete, the application builder can be used to graphically design the system and then automatically generate code. The system can then be exhaustively tested from both a functional and a stress perspective using a variety of available automated test execution environments[2].

Test Process

While this design process is highly automated, the test process is still mostly a manual process. Tests are developed by a laborious process of first analyzing a call flow diagram of the system design to determine what tests are required. The tests are then manually implemented, reviewed and debugged in low level scripting languages, requiring skilled programmers with both programming and telephony test experience. This process

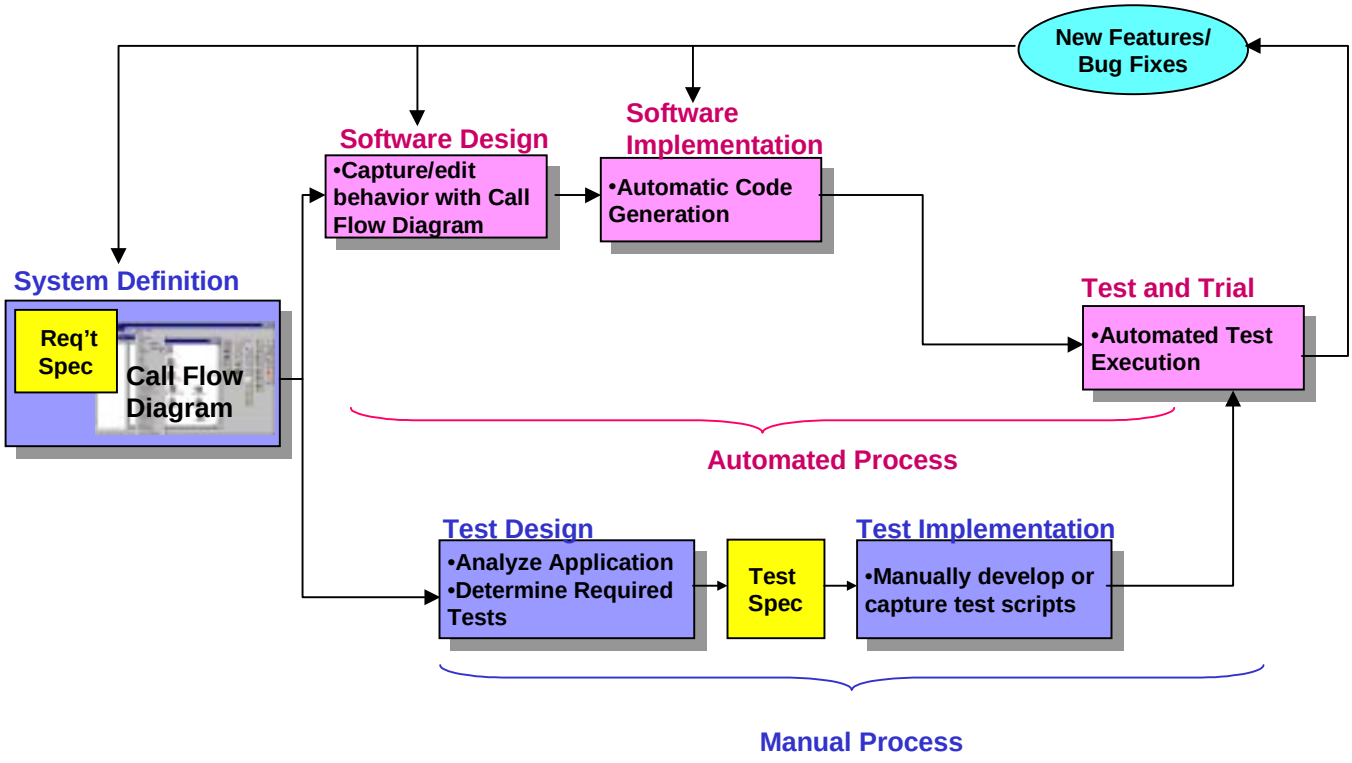


Figure 1: Test Process Automation Lags Design Process Automation

is repeated every time the application is updated or changed. At the end of the process there is no explicit measure of the test thoroughness. This can result in long field test cycles or worse, problems found by customers.

Test is becoming the bottleneck

Features are released incrementally during the design process (Figure 2). The tests for each feature require a separate design and coding process. The time available for this process is being squeezed from two directions:

- Application builder tools make it possible for new features to be added or changed quickly, making it difficult for the manual driven test process to keep up and creating a bottleneck.
- Market pressure is reducing the time for creating tests and increasing the need for new features. These features are implemented rapidly using automated application builder tools.

Also, because of the time required to develop feature tests, little testing of feature interaction is done until late in the process. Problems found at this phase can be the result of a inaccurate requirement, poor design or an implementation error (Figure 1). Changes in the requirements require changes to the existing tests, further squeezing the process.

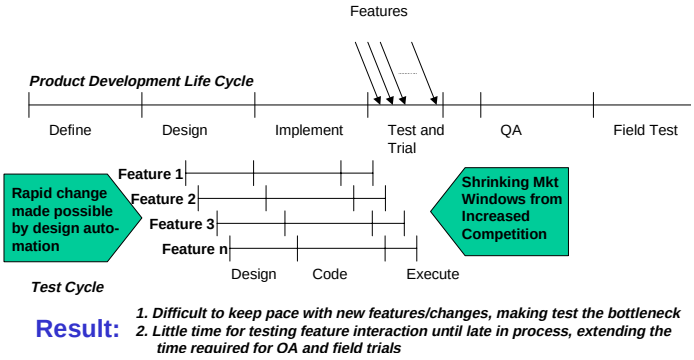


Figure 2: Rapid change and shrinking market windows are squeezing the Test Development Process

Available Test Automation

Today there is a gap between automation tools used for test *design* and those used for test *implementation*. There are test *design* tools that model an application's behavior and then automatically find test paths through the model[3]. Test *implementation* tools automate the creation of test scripts for a single call flow path, once it is defined. There is a gap in that the tools defining the test paths don't easily implement the code for those tests. In most cases, the link is a Test Specification defining the test paths that need to be implemented (Figure 3).

The test *design* tools are sophisticated general-purpose behavioral modeling tools that can either find paths through the model, or validate specific use cases[4]. The languages for these tools are complex and require skilled programmers to implement.

There are several different approaches being used for automating the test *implementation*. All are based on having a set of re-useable test functions that are linked together to implement the actions of a single call path.

At the simplest level, the functions are re-used by cutting and

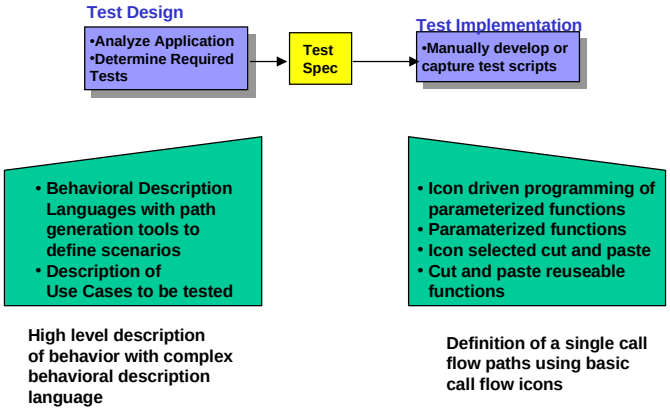


Figure 3: A gap exists between the current automation tools used for test design and test implementation

pasting them into a new program using either an editor or clicking on an icon that automatically pastes the code. The script code is then hand edited to tweak it for new applications.

A more sophisticated approach involves having parameterized functions that can be customized by modifying parameter values. This reduces the need for modifying low level code and allows for a more general-purpose library of functions.

Implementing an Integrated Approach for Test Design and Implementation

The gap between test design and test implementation tools can be filled by developing a set of re-useable call flow objects that can be used by the test design tools to model the behavior of the application and automatically find the test paths. By concatenating test execution code embedded in each re-useable object, the path generators in these test design tools can automatically implement the tests for each path found.

To implement this kind of integrated approach, careful consideration should be given in defining the following:

- A set of re-useable objects
- A means of connecting the re-useable objects to represent the call flow
- A methodology or algorithm for generating optimal test paths
- An output format suitable for debug and documentation

The remainder of this paper will discuss the requirements for each of the above implementation steps.

Defining Reuseable Objects

What objects are required?

The first step in implementing an automated test generation solution is defining a set of re-useable objects that can be used to describe the behavior of a Computer Telephony application from a user interface perspective. The goal is to have a set of objects that a non-programmer can use to describe the application behavior. A starting point is found in the building blocks used by the design team's application builder tool. Unfortunately, depending on the tool, there could be 100's of building blocks. Many of these are operations internal to the design that an end user would not see (i.e., connect to database, string operations, file operations, etc.). Figure 4 illustrates a list of objects that would be needed to model at typical CT application at the interface level.

Call Flow Object	***Description***
1. Prompt-Response	play a greeting and receive a response
2. Play Voice	play an audio file
3. Place Call	make a phone call
4. Record Voice	save voice data in a file
5. Go On-Hook	disconnect the call
6. Go Off-Hook	prepare to accept incoming calls
7. Send Tones	send MF or DTMF tones
8. Receive Tones	receive MF or DTMF tones
9. Wait For Hangup	wait for call to be disconnected
10. Wait For Call	wait for an incoming call
11. Wait For Energy	wait for energy on line
12. Wait for Silence	wait for silence on line
13. Recognize Speech	compare speech data with a vocab
14. Reject Call	reject incoming call on the line
15. Transfer Call	transfer a call to another number
16. Error	default error handler
17. Stop Channel	stop communications on the channel
18. Send Fax	send a fax
19. Receive Fax	receive a fax

Figure 4: Re-useable Call Flow objects must describe the behavior of a Computer Telephony application at the user interface

In defining the library of re-useable objects, you should consider whether the objects are defined from an application perspective or a test perspective. The application perspective is the same view a designer uses with an application builder tool. The test perspective is a mirror image of the application view, looking into the application from the tester. Test engineers use this perspective to think about what the tester is doing instead of how the application is reacting. As an example, a "Place Call" object in an application perspective (which causes the application to dial a number) is the equivalent of a "Receive Call" object in the test perspective, which causes the tester to receive an incoming call. The advantage of the applications perspective is that enables a direct mapping to the design call flow diagram and facilitates easier interaction with design engineers. The disadvantage is that it is an inside out view of the tests that are run, adding complexity to the test problem.

In addition to defining a set of call flow objects that can represent the application behavior, it may be necessary to develop additional specialized test objects. As an example, you may want to verify that the application responds properly when an end user dials ahead of the prompts. This could require some form of "listen and respond" object that sends digits a specified time after the prompt begins. It may also be necessary to develop objects that can capture and log various response data.

Many applications are set up to have data dependent call flows. For example, an investment firm may want to have different customer call flows based account type or balance. An insurance company may want to route DNIS numbers from a disaster area directly to an agent. Testing these applications requires an object that enables data to be specified or imported.

It may also be necessary to define objects that represent screen pops. Some applications have parallel processes that route a call to an agent while at the same time looking up an account balance in a database and popping it on the screen of a designated agent.

Implementing the Re-useable Objects

Once the objects are defined, the first step in implementing them is to develop the code for executing the object action on the target test execution system. To make the object as general purpose as possible, this code should be parameterized to allow for broad re-use. Some of the parameters include digits to be dialed/received, voice prompts, telephony parameters and phone lines used. The code for the objects should also include default error handling routines.

The objects also require a means of inputting the parameter values. Ideally, this should be done graphically with text boxes, buttons, or pull downs to enable use by non-programmers. Finally, there should be a means of providing

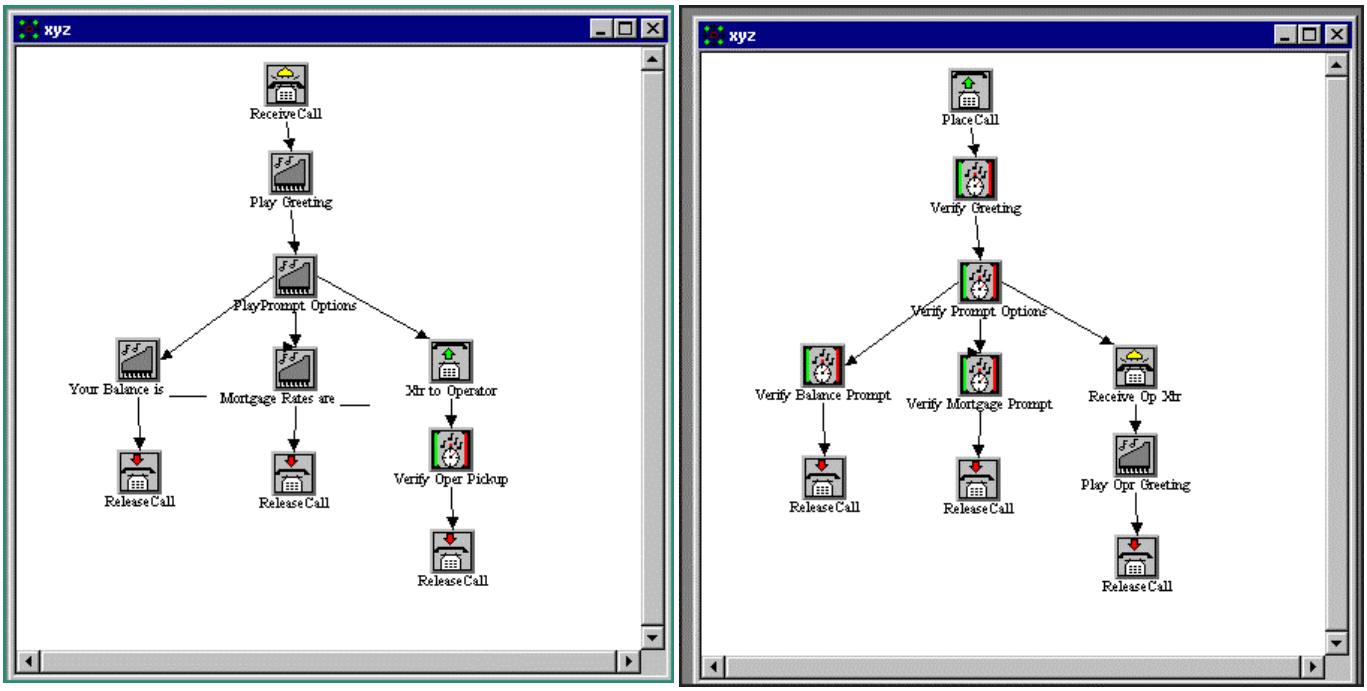
specific test comments for documentation purposes.

Creating the Call Flow Diagram

Once the call flow objects are defined and implemented, there needs to be a method of connecting them together to model the application behavior in the form of a call flow diagram. The ideal solution would be to import a call flow diagram directly from the chosen application builder tool. Unfortunately, there are no graphical interface standards for call flow diagrams to facilitate this. Also, since there is not a dominant supplier in this area, there are no third party solutions for integrating a call flow diagram of any application builder tool with any test design tool. Lacking the above, the best approach is to make it possible for a non-programmer to create a new call flow diagram that can easily be mapped to the design call flow diagram. To do this, it must be possible to enter the call flow diagram from the same application perspective used by the application builder tools (Figure 5).

In order to support data dependent call flows, the call flow diagram must be able to represent conditional branching dependent on data values. Figure 6 shows a call flow diagram of a voicemail system where the main menu prompt and the call flow path will vary depending on the data values defined in the table. For example, a caller with "application access" turned off would not hear the "press 6 for applications" prompt. If the 6 digit were then pressed, the caller would be routed to an error message.

Finally, being able to specify hierarchical call flows enables a modular approach to test. Figure 6 illustrates a series of sub-call flow diagrams that get branched to from the main menu. Drawing all of these on a single flat call flow diagram would be hard to organize and even more difficult to read. Using hierarchy, the application can be decomposed to modular elements that can then be worked on in parallel. Additionally, having re-useable sub call flows allows the work to be leveraged across multiple platforms. If the sub call flow changes, the change can be implemented across all instances of the sub call flow with a single edit.



Application View **Test View**

Figure 5: An application view is required for easy mapping to the design call flow diagram.

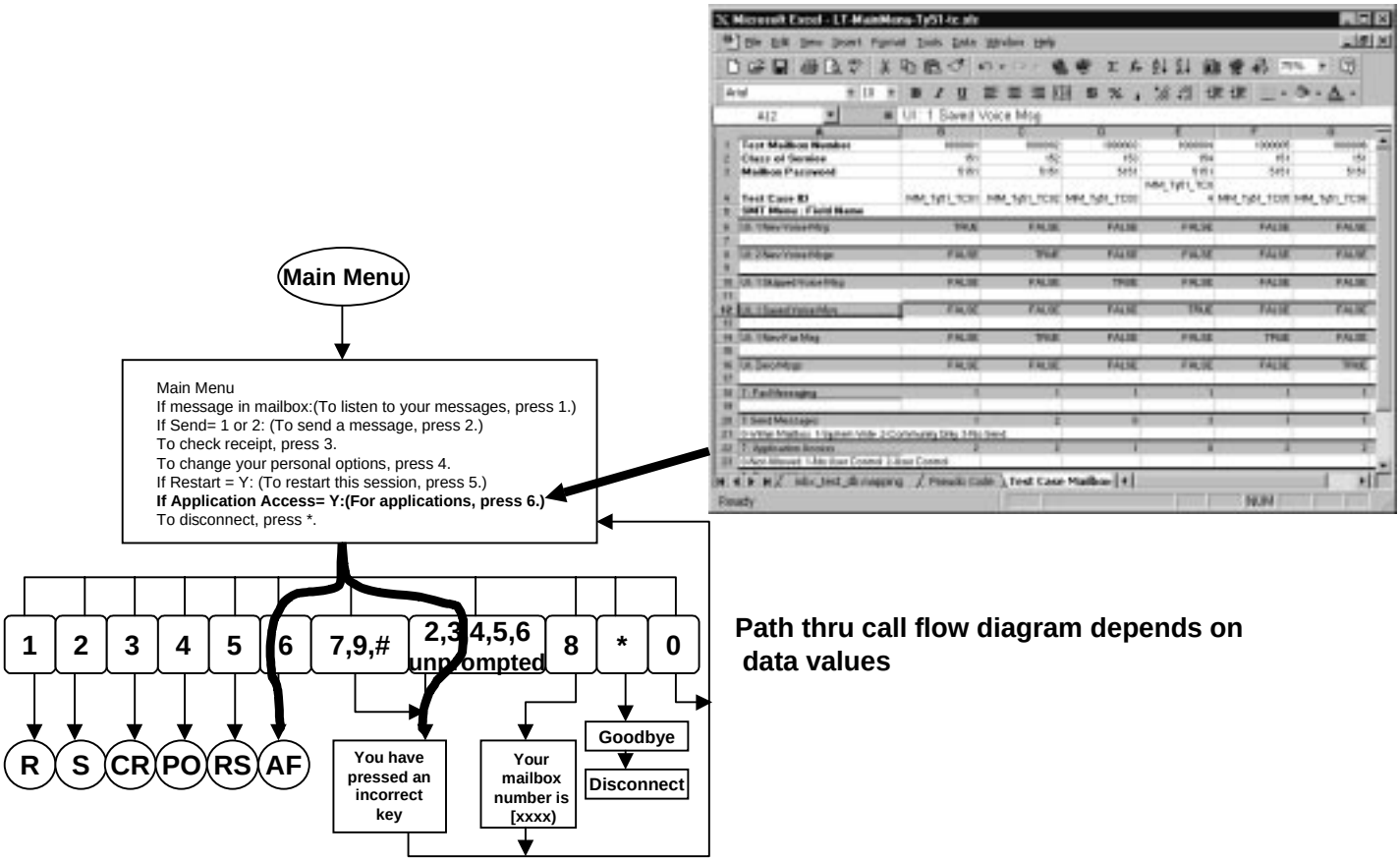


Figure 6: Call Flow descriptions must support data dependencies and hierarchy

Test Generation Requirements

After the call flow diagram is entered, the next step is to define an optimal set of test paths through the diagram. At a minimum, these test paths should accomplish the following:

- Touch all elements of the call flow diagram(data and call flow objects) at least once.
- Test for response to negative behavior.
- Verify that illegal conditions can't occur.

Achieving the above can be done either manually or automatically. In a manual approach, each path is separately constructed with the call flow objects and the code from each object is assembled to create a test script. Alternatively, using the path generation capability of a test design tool can automatically create the test paths. This approach offers several advantages:

- Automatically create test paths that cover all objects. This is particularly important on large, complex call flows. Manually analyzing dozens of pages of call flows to determine the optimal set of test paths is tedious, time consuming and error prone.
- Rapid response to changes in the call flow diagram. Instead of editing low-level test scripts, tests are modified or added by editing the call flow diagram and using the path generator to rapidly create new tests. This has the greatest impact on new applications with volatile requirements, or in competitive situations where new features are frequently added.
- Lower skill levels required to design tests. By reusing debugged call flow objects, non-technical people can enter diagrams for test engineers.

The key issue in the using an automated path generator is to have the test paths be relevant applications scenarios. Having 100's or 1000's of tests on a part of the call flow happening 1% of the time may add nothing to quality and just decrease throughput. In some of the path generation tools, the relative likelihood of each path can be specified so that the distribution of paths created is consistent with the operational behavior of the application.

A philosophical issue to consider is whether or not to let test programmers debug test scripts by hand editing the script code after it is generated instead of making the necessary changes to the code embedded in the template. The problem with tweaking the generated code instead of the code in the call flow objects is that the test program becomes disconnected from the original call flow diagram. If new features are added to the call flow diagram, the tests that are automatically

generated will have the same problems that were previously fixed. The best approach is to edit the generated code until the script works, back annotate the changes into the call flow objects, and then re-generate the test. This way, the necessary changes are permanently captured in the call flow object.

Test Documentation

Ideally, the documentation for the test scripts should automatically be generated as a separate output whenever new test scripts are generated. This can be accomplished by having the generation tool concatenate test descriptions annotated on each call flow object as it creates each path through the call flow diagram. This eliminates having an error prone separate step of documenting the scripts after they are generated. It also assures that the documentation is always in sync with the test scripts

While debugging scripts it is also helpful to have a graphical display that shows the sequence of events on each channel for each test script. This sequence-based view simplifies review of the test scripts with other test or development engineers.

Summary

Figure 7 summarizes the components of an automated solution discussed in this paper. The first step is to create a library of re-useable call flow objects that define the actions at the application's user interface. These objects contain the code for executing the action on the target test execution environment. The objects are connected using a graphical editor to link the call flow objects together in the form of a call flow diagram. As the objects are placed, they are edited with comments describing the test action. Separate tables for data effecting the path through the call flow diagram must then be developed. Once the call flow diagram is completed, the test generator of an automated test design tool can be used to find paths through the call flow diagram. As each path is found, the test execution code and test comments are concatenated together to automatically create test scripts and documentation.

Implementing this approach can have a significant impact on the overall product development process by:

- Reducing the time and skill level required to develop comprehensive test scripts
- Allowing tests for feature changes or additions to be developed in minutes
- Increasing the quality level by automatically generating comprehensive end to end tests across the entire call flow diagram.

These benefits must be weighed against the cost developing or purchasing such a system.

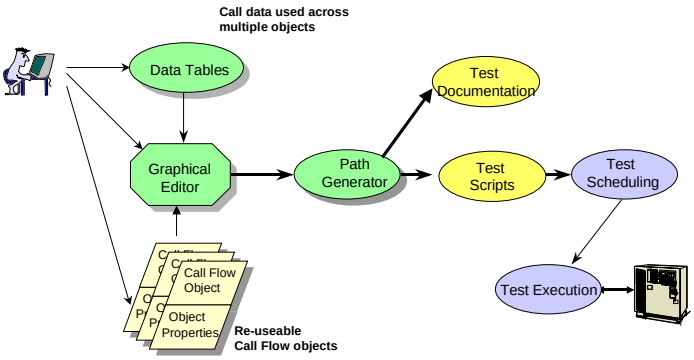


Figure 7: Example of an integrated solution for automated Computer Telephony test generation

References

[1] Knowland, D., Life-Cycle Testing For Call Center Quality, CTI Management Magazine, December 1998

[2] Gladstone, S., Testing Computer Telephony Systems and Networks, Flatiron Publishing, New York, 1994, ISBN 0-936648-57-0

[3] Beizer, B., Black Box Testing, ,John Wiley & Sons, New York, 1995. ISBN 0-471-12094-4

[4] Clarke, J., "Automated Test Generation from a Behavioral Model", Proceedings of the Software Quality Week 1998 Conference, 1998

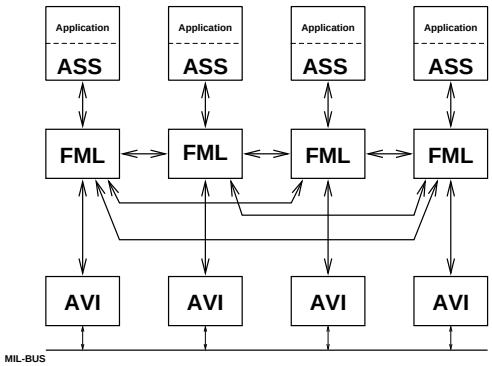
Combining Methods for the Analysis of a Fault-Tolerant System

Bettina Buth, Jan Peleska, Hui Shi
BISS – Bremen University

Overview

- Background
- Approach: Abstraction – Algebraic Transformation – Model Checking – Generic Theories – Liveness Induction – Analysis of Cycles
- Future Plans

International Space Station – FTC Structure:



Basis:
- occam code/pseudocode
- diagrams of data flow via channels
- informal description

Goals

Verification Goal	Verification Method
deadlock freedom	CSP: generic theories - abstraction - model checking - algebraic laws - compositional theory
livelock freedom	as for deadlock analysis + dependency analysis
absence of bottlenecks	stochastic petri nets
correct implementation of Byzantine Protocol and failure detection	abstraction - model checking - compositional proof theory
correct implementation of application services	verification using Hoare Logic
"most non-deterministic admissible software/hardware integration is correct"	Hardware-In-The-Loop-Testing

Deadlock Analysis :

"Whenever the system reaches a stable state where all communications are blocked, all processes reading from interface channels are ready for input."

Livelock Analysis :

"The system will never engage in an unbounded sequence of internal communications without visible communications on the interface."

Assumption :

Environment always accepts outputs and sometimes yields inputs.

Combining Methods

- **Basic Techniques:**
  - **Abstraction of** occam **code**
  - **Algebraic Transformations**
  - **Generic Theories**
  - **Model Checking using FDR2**

- **Composition of Results:**
  - **Compositional Proof Theory**
  - **Liveness Induction**
  - **Analysis of Cycles, e.g. using Fault Tree Methods**

Abstract Interpretation of occam Code:

- **sequential code not influencing communication behaviour is deleted,**
- **channel protocols are reduced to minimal set of values,**
- occam `IF condition THEN P ELSE Q` **may be replaced by** CSP **if-construct or internal choice,**
- **abstract interpretation is compositional:**
  If $\mathcal{A}(P)$, $\mathcal{A}(Q)$ **are valid abstract interpretations of** P **and** Q **and consistent with respect to interface** I**, then** $\mathcal{A}(P) \underset{I}{\|} \mathcal{A}(Q)$ **is a valid abstract interpretation of** $P \underset{I}{\|} Q$.

Model Checking

Tool Support: FDR
(Failures Divergence Refinement)

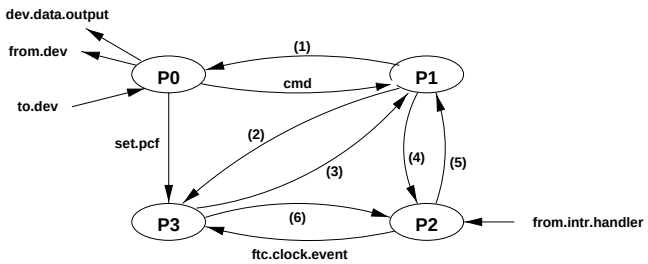
- **Prerequisite: Finite State Space**
- **Problem: State Space finite, but too large**
- **Solution: reduce problem by**
  - **algebraic transformation**
  - **generic theories and abstraction/refinement**
  - **other methods like analysis of cycles**

Generic Theories

Idea : use knowledge about special processes for
- **further abstraction**
- **combination of results**

process types
- `CHAOS(X) = ⊓ x:X @ x → CHAOS(X) ⊓ STOP`
- `RUN(X) = [] x:X @ x → RUN(X)`
- `YIELD(X) = ⊓ x:X @ x → YIELD(X)`
- **pipes and generalized pipes**
- **buffers**
- **multiplexer/concentrator**
- **cycles with capacity**

Analyzing Cycles: Dependency Analysis



- investigate relevance of cycles in $\mathcal{G}$
  SYS is livelock free if any cycle is either
  - not possible in the CSP abstraction; i.e if one edge does not depend on its predecessor in the cycle,
  - possible in the CSP process, but additional information in the occam code allows to exclude this situation.

Experiences: deadlock vs livelock

- abstraction with more details e.g. timer events
- failure-divergence model produces larger state space
  $\Rightarrow$ further decomposition
- new techniques for combining results
  - liveness induction
  - dependency analysis

Future Work:

- Tools supporting abstraction
  (for occam or other languages)
- support for cycle detection
- support for load analysis
- more generic theories

Combining Methods for the Analysis of a Fault-Tolerant System

Bettina Buth* Jan Peleska† Hui Shi‡

1 Introduction

The acceptance of Formal Methods in industries essentially depends on their scalability, i.e. their applicability in large scale realistic industrial projects. An important aspect is the availability of suitable tools, but from our experience this is but one aspect. The diverse nature of system components and the techniques used in the different steps of the development process require the use of a combination of methods for the development as well as for the analysis of these components. In this paper we report experiences in using a combination of methods for the analysis of a large software system, namely the fault-tolerant data management system for the International Space Station (ISS). It was necessary to use these methods in order to break down the respective tasks into manageable sub-tasks.

The project started in 1995 when JP Software-Consulting (now Verified Systems International) in collaboration with the Bremen Institute for Safe System (BISS) were contracted by Daimler-Benz Aerospace (DASA) to perform an analysis of a fault-tolerant data management system for the ISS. Up to now various aspects of the system correctness have been investigated:

- freedom of deadlock (see for example Buth et al. [2]),
- freedom of livelock ([3]),
- correct implementation of voting algorithms used for majority-based decision making,
- correct implementation of the Byzantine Agreement Protocol used to reach agreement between correctly operating system components ([18]),
- performance properties depending on clock rates([19]),
- Hardware-In-The-Loop tests for the overall system.

*BISS, Bremen Institute for Safe Systems, Email bb@informatik.uni-bremen.de
†Verified Systems International, Bremen, Email jp@informatik.uni-bremen.de
‡University of Bremen, Email shi@informatik.uni-bremen.de

Different approaches were employed for these analysis tasks. These comprise abstraction of the occam code to CSP processes and reasoning about these abstractions, Hoare-style program verification for sequential code, and the use of stochastic Petri nets for the throughput analysis.

Whereas the results presented in Buth et.al. [2, 3] relate to deadlock resp. livelock analysis from a more theoretical point of view, this article focuses on the description how we overcame the essential problems of size and complexity of a realistic large-scale system by employing a combination of methods for the analysis. Due to the confidential nature of the material unfortunately it is not possible in this framework to make available details of the original code or our specifications (these are available only in the internal reports [4, 5]).

2 Technical Background: The Fault Tolerant Computer

The software to be analyzed is part of a fault tolerant computer to be used in the International Space Station (ISS) to control space station assembly, reboost operations for flight control, and data management for experiments carried out in the space station. In the following we will describe the system architecture and the goal of the analysis as well as the starting point for the verification.

2.1 FTC Architecture

The overall architecture consists of up to four communicating lanes, each providing services for the applications. Each of these lanes is structured into an application services layer (ASS), a fault management layer (FML), and the avionics interface (AVI). The ASS resides on the application layer board and contains table driven services for the application software and the operating system. The AVI is in charge of the MIL Bus protocol handling according to predefined timing slot allocations. These are defined in an input/output table. The function of FML is twofold: First, it provides the interface between ASS and AVI of one lane, transferring messages from AVI to ASS and vice versa. Second, it performs the data transfer between lanes thus allowing communication between the fault management layers of all lanes. This communication is the basis for error detection, error correction, lane isolation (in the case of an unrecoverable error), and lane reintegration. In each lane, the application layer plus ASS runs on a customized Matra board using a SPARC CPU. Both FML and AVI reside on separate transputer boards. The lanes communicate only at FML level using the transputer links. Each FML uses up to three links for communication with the other lanes, and one link (link 0) for communication with AVI. Data transfer with ASS is performed using a VME interface. See Figure 1 for the architecture of a full four-lane system.

Error detection is essentially based on a two round Byzantine distribution schema [12] where data are communicated between FMLs and voted using various specialized voters. The aim is to ensure that (1) all ASS instances of

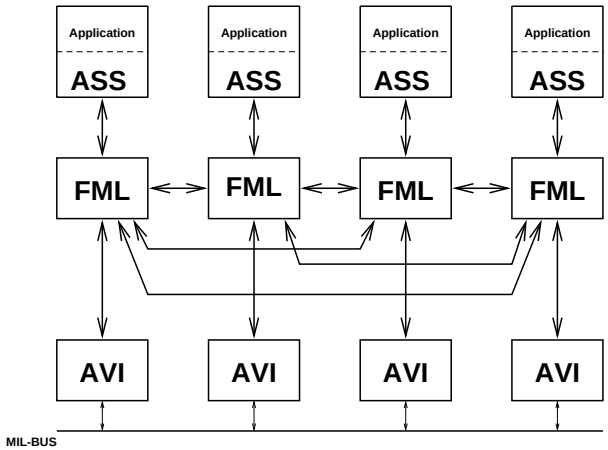


Figure 1: FTC Architecture

non-faulty lanes get identical messages from FML, (2) all AVI instances of non-faulty lanes get identical messages from FML, (3) for data calculated by all lanes (congruent source messages) all non-faulty lanes get the correct(ed) message, (4) for data calculated by one lane (single source messages) all non-faulty lanes get the correct(ed) message if the originator is not faulty. The implemented design allows detection of one Byzantine or deterministic fault in a four-lane system and recognition of a deterministic fault in a three-lane system.

2.2 Goals of Communication Behavior Analysis

Both FML and AVI software are implemented in occam [11], consisting of systems of processes running concurrently and communicating via internal channels and partly using shared memory to exchange data without the additional communication overhead. The software is structured hierarchically: larger processes, each consisting of several subprocesses, implement different parts of the respective functionality and communicate the computation results to the other main processes. In addition to the usual aspects of functional correctness and timing, a concurrent architecture of software adds the new problem of potential blocking. It must be guaranteed that the software does provide its services to the adjacent components of the overall system. Blocking can be due to two causes: either the system does not work at all because components mutually wait for results, or it does only work internally without communicating with its environment. The first situation is called a *deadlock*, the second a *livelock*. Our essential task for the analysis of the communication behavior was to show that neither deadlock nor livelock situations can occur in the implementation of AVI and FML. The first step of the analysis was to find out what actually had to be checked with this respect.

For the deadlock analysis we have to check whether any of the communicat-

ing system of processes that forms the software continuously blocks the communication with the environment. Precisely speaking, the following verification goal had to be investigated:

> In an environment that always accepts outputs from the system but may or may not refuse to provide inputs, the following assertion holds: Whenever the system reaches a stable state where all internal communications are blocked, the system will always accept new input from the environment.

In high level concurrent programming languages, such as occam it is conventional for communication channels between two processes to be concealed from the environment. This can potentially cause a form of divergence known as *livelock*.

The objective of our livelock analysis is to investigate occurrence of internal divergence only. A CSP (or occam) system X is called *livelock free* with respect to interface channels $c_1, c_2, \ldots, c_n$, if the system will never engage in an unbounded sequence of (internal) communications without interleaved (visible) communications on the interface $c_1, c_2, \ldots, c_n$. This property is as important as the *absence of deadlocks* since it ensures the interaction of the system with its environment.

2.3 The Basis for the Analysis

The material provided as basis for the analysis consisted of

- a printed version of pseudo code for AVI,
- electronical files of the final implementation of FML and AVI (in the latter case for re-verification),
- diagrams depicting the overall architecture and communication behavior of individual processes,
- general information about the system requirements and technical details of the system,
- verbal communication with the system engineers.

3 Verification Approach

The general idea for the analysis of communication properties of occam programs as proposed here is to exploit the fact that occam channel communication can easily be modeled in CSP. For CSP a wide range of theories and suitable tools like theorem provers and model checkers are available. The basic idea for both deadlock and livelock analysis performed in this project is to use model checking and the tool FDR [9]. After manually abstracting the occam programs to CSP

processes the systems still is too large for a direct approach using FDR. Thus it is necessary to decompose the task and use other techniques for combining the results to obtain an overall result for the full system. Some of these techniques are tool supported, others currently are only performed unassisted.

For real-time applications, periodic internal events such as time-scheduled interrupts cannot cause divergence as long as their period is long enough to allow for other activities in between. These events may cause cycles of internal communications which are intended and not to be considered as livelock, because their occurrence is guarded by time intervals where other communication events are possible. To reflect this properly in an untimed CSP model, we regard these channels as part of the interface for the purpose of livelock analysis.

The mathematical proof theory of CSP allows to verify properties of CSP specifications by means of logic reasoning. The CSP language, its mathematical foundations and its possible applications have been thoroughly investigated since the late sixties (see [10, 9, 8]). For the verification goal described above we apply various verification techniques. They can roughly be divided into those which are applied to basic components, namely

- *Model checking* is used for the mechanized verification of small-sized CSP components,
- *Abstract Interpretation* is applied to "lift" occam process components to CSP components reflecting the essential aspects of the process communication behavior while abstracting from details irrelevant for the verification goal,
- *Generic theories* increase the efficiency of analysis: process instances of a generic class inherit the class properties, which means that it is only necessary to show that concrete processes are instances of a suitable generic process,
- *Algebraic reasoning* is applied to transform CSP process specifications into equivalent ones better suited for the model checker.

and those which are used for combining the results about the basic components:

- The *compositional proof theory* of CSP is applied in connection with refinement properties to derive global properties of the complete system from the local properties established for the isolated components,
- A special form of *cycle analysis* is used on the data flow diagrams to show that cycles in the communication can not induce an infinite backlog as cause of a livelock,
- A *dependency analysis* similar to fault tree methods is applied in order to reduce the number of cycles to be investigated for livelock analysis,

- *Liveness induction* provides a suitable means of extending the system step-by-step and making use of local livelock freedom results of the component processes.

3.1 Model Checking and Refinement

The specification language CSP (Communicating Sequential Processes) is associated with a formal method allowing to verify properties of parallel systems. (See Hoare [10] and the recent book by Roscoe [17] for more details). CSP processes proceed by engaging in communications. Processes may be composed by operators which require synchronization on some communications. This, rather than assignments to shared state variables, is the fundamental means of interaction between agents. The theory of CSP has classically been based on mathematical models remote from the language itself. These models have been based on observable behaviors of processes such as *traces*, *failures* and *divergences*. The semantics of a CSP process can be given in three models:

- In the *trace model* a process is represented by the set of finite sequences of communications it can perform,
- In the *failures model* a process is represented by its set of traces as above and also by its refusals – a set of communications it can refuse after a sequence of communications,
- The *failures-divergence model* extends the failures model with the divergences of a process – the traces during or after which the process can perform an infinite sequence of consecutive internal actions or otherwise show chaotic behavior.

FDR (*Failures-Divergence Refinement*) is a model-checking tool for state machines, with foundations in the theory of concurrency based upon CSP. Its main method for establishing whether a property P holds for a CSP process system SYS is to investigate refinement properties: SYS is compared to a specification process $SPEC$ for which P is known to be valid. The comparison is performed in one of the semantic models of CSP which is known to preserve property P under refinement. The main ideas behind FDR are presented in [15].

Every CSP specification consisting of finite-state processes with finite-value channels can be translated into a finite *transition graph* representation. This graph contains all the semantic information of the original CSP specification. As a consequence, every property of the specification – as, for example, livelock freedom or more specific refinement properties– can be verified by exhaustive analysis of the transition graph. Moreover, such an analysis can be mechanized. The FDR tool provides this mechanization and has been used for all model checking results about the abstract interpretations described in this document.

The notion of refinement is a particularly useful concept in many forms of engineering activity. If we can establish a relation between components of a system which captures the fact that one satisfies at least the same conditions

as another, then we may replace a worse component by a better one without degrading the properties of the system. Refinement relations can be defined for systems described in CSP in several ways, depending on the semantic model of the language which is used.

3.2 Abstract Interpretation

For the analysis of deadlock or livelock freedom it is unnecessary to inspect every detail of the occam code, since only a subset of the statements of this code influences the communication behavior. It is therefore possible to generate a CSP specification which represents an abstract version of the original occam process P and shows only that amount of detail which is relevant for its communication behavior. In general, we call such a CSP specification $\mathcal{A}(P)$ a *valid abstract interpretation for a property* p of the corresponding occam process P, if

> *Whenever $\mathcal{A}(P)$ has property p this implies that P has property p as well.*

If a valid abstract interpretation $\mathcal{A}(P)$ is available, we only need to analyze $\mathcal{A}(P)$ instead of P. In our case this means: If $\mathcal{A}(P)$ is free of deadlock resp. livelock, the same must hold for P.

The basic approach to construct valid abstract interpretations for deadlock and livelock freedom uses four techniques in the translation from occam to CSP:

1. Every sequential algorithm whose results do not influence communication behavior is deleted.

2. Each occam channel protocol is reduced to the set of values influencing the communication behavior in a distinctive way.

3. Every occam IF-construct `IF condition THEN P ELSE Q` may be replaced by the equivalent if-construct in CSP or by the internal choice operator of CSP yielding $P \sqcap Q$.

4. If valid abstractions $\mathcal{A}(P)$, $\mathcal{A}(Q)$ for two processes P and Q are available and these interpretations use the same abstractions on their communication interface I, then $\mathcal{A}(P) \underset{I}{\|} \mathcal{A}(Q)$ is a valid abstract interpretation of P and Q operating in parallel. Using this technique, larger abstractions can be build from existing ones.

Figure 2 provides an overview about the relationship between occam and CSP constructs.

3.3 Generic Theories

Generic theories in CSP provide theorems about generic classes of processes. Such process classes are parameterized over number, names, and data types

occam-**construct**	**CSP-construct**
IF_THEN_ELSE	$\sqcap$ or if_then_else
PAR (PRIPAR)	$\|\|\|$ or $\|\|$
ALT (PRIALT)	$[]$
WHILE	$P = \cdots \rightarrow P$ or $P =$ if b then $\cdots$ else SKIP
c?x	c or c?x
c!a	c or c!a
SEQ	; or $\rightarrow$

Figure 2: Abstraction Table

of channels or process parameters acting as "software switches" controlling the process behavior. Process instances of a generic class are created by providing specific values for each generic parameter. A generic theory provides properties valid for each process instance of the associated class; in general, the property is instantiated analogously to the corresponding instance process.

Thus it is useful to identify general classes of processes and investigate their properties independently from a specific analysis goal. During abstraction and combination of results (see below) the use of instances of generic theories allows us to reduce the verification task in various ways. One aspect is to use these instances directly as abstractions if they lead to smaller state spaces for model checking. Another is the combination of results from smaller components in connection with general theorems about the combination of various types of generic processes. In both cases it is necessary to make sure that the concrete process is a refinement of the instance of the generic theory.

Some classes of generic processes identified during the analysis of AVI and FML are the following:

- `CHAOS(X) = ⊓ x:X @ x → CHAOS(X) ⊓ STOP`
  Instances of `CHAOS(X)` may offer or refuse any event from the event set `X`.

- `RUN(X) = [] x:X @ x → RUN(X)`
  Instances of `RUN(X)` always accept any event from set `X`.

- `YIELD(X) = ⊓ x:X @ x → YIELD(X)`
  For a set of events `X` `YIELD(X)` will always provide at least one element of `X`. Note that this does not exclude that one or more events are never offered.

- pipes
  consist of several processes which transfer input from their predecessor to their successor in a serial way.

- buffers
  store data up to a certain capacity and allow to retrieve these data in *first in-first out* order.

- multiplexer/concentrator
  accept inputs and provide outputs on several channels with the constraint that there always is a maximum number of outputs after which the multiplexer can not refuse a new input.

- generalized pipes
  process information in a series of processes each of which accepts additional inputs.

- cycles with capacity
  guarantee that an output can only be delayed by a maximum number of internal cycles depending on the number of inputs.

3.4 Algebraic Reasoning

CSP possesses a rich proof theory which provides algebraic laws for the transformation of CSP processes into equivalent ones. Specifically, deadlock and livelock freedom are preserved under application of these laws. Algebraic manipulations are useful to simplify process terms and reduce the state spaces of processes before starting the model checking process. This is often useful for abstractions of occam processes where several different cases in alternatives are collapsed into one CSP case.

3.5 Compositional Proof Theory

Real-world systems such as AVI and FML usually consist of a number of processes communicating with each other and the environment. Each of these may again comprise several subprocesses. Since in general it is not feasible to deal with the full system as a whole, one of the crucial points is how to compose the analysis results achieved for the individual subprocesses and processes. Here, CSP provides useful support with its compositional proof theory, which was used for the analysis of FML and AVI in various ways.

The strongest way of exploiting compositionality is by using theorems which allow to reduce the overall obligation to obligations just involving components of the system. A simple example is the following:

> *Let $SYS = P \,|||\, Q$ be a system of two processes P and Q which run concurrently without any synchronization. Then SYS is free of deadlocks whenever P and Q are free of deadlock individually.*

Another way of exploiting compositionality is to refer to an essential property of refinement in connection with operators for the composition of processes (preservation of refinement):

> *If $P_i \sqsubseteq_{FD} Q_i$ for $i : 0 \,..\, n$ and ω is an n-ary operator, then $\omega(P_0, \ldots, P_n) \sqsubseteq_{FD} \omega(Q_0, \ldots, Q_n)$ holds.*

This allows to proceed as follows for livelock analysis: If livelock freedom of $\omega(Q_0, \ldots, Q_n)$ can not be established because of the problem size, find simpler processes $P_0, \ldots, P_n$ such that $P_i \sqsubseteq Q_i$ for each $i \in \{1, \ldots, n\}$ and $\omega(P_0, \ldots, P_n)$ is livelock free. If this property can be established, compositionality combined with refinement imply the livelock freedom of $\omega(Q_0, \ldots, Q_n)$ as well. Note that this approach is also possible for deadlock analysis and other properties.

In connection with generic theories, compositionality can also be used to reduce the verification task. The idea is to establish theorems about the composition of instances of generic theories and use these whenever a system is built accordingly. In that case the remaining task for the concrete system is to show that the component processes are instances, i.e. in CSP refinements of the generic processes.

3.6 Cycle Analysis

In this section we will present some results to prove that the FML of each isolated lane fulfills the deadlock freedom verification goal stated in Section 2.2. The main result is stated in the following theorem:

Theorem 1 *In every context which never blocks outputs on transputer links, outputs to the* ASS *layer and outputs to the* AVI *layer, the following property holds:*

Whenever the FML *system reaches a stable state where no communications are possible, all processes reading from vital channels are ready for input on these channels.*

Here, *vital channels* means all input channels where data from the environment should always be processed.

The proof of this theorem is based on the observation that FML consists of two types of processes which differ in their behavior regarding channel input:

- **Unconditional Input:** The process will always accept a new input on the channel, possibly after a bounded number of non-blocking internal communications. The acceptance of a new input does *not* depend on successful outputs to other channels prior to the next input. Unconditional input is provided whenever the corresponding process abstraction behaves as RUN on the input channel.

- **Bounded Conditional Input:** *Provided that the environment will not block process outputs*, the process will always accept a new input on the channel, possibly after a bounded number of non-blocking internal and external communications. This type of input behavior is guaranteed whenever the corresponding abstraction behaves as $MUXOUT(max, IN, OUT)$ on its input channels IN and output channels OUT.

Additionally it is possible to reduce the overall system of processes by application of two lemmata (see Buth et.al. [2] for a more detailed description

of these lemmata and their application) which do not contribute to a potential deadlock behavior of the whole system. After this reduction of the proof obligation the proof proceeds by analyzing the remaining processes individually and their communication behavior with special emphasis on communication cycles.

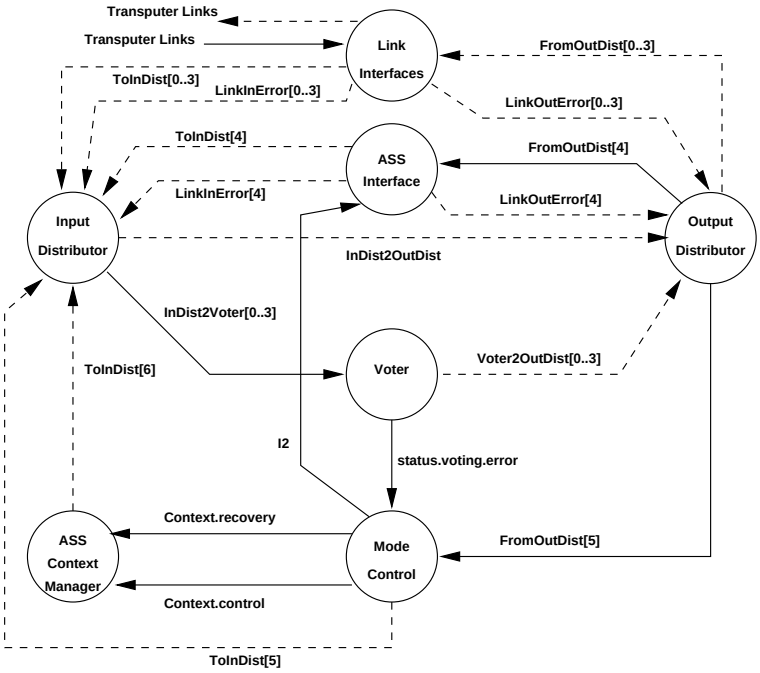


Figure 3: Data flow graph for deadlock analysis in Theorem 1

Proof sketch of Theorem 1.
The two preceding lemmas can be applied to conclude that the remaining channels and processes that are still to be analyzed with respect to deadlock freedom are the ones depicted in Figure 3, where unconditional input is depicted by a dashed arrow and bounded conditional input is marked by a solid arrow, labeled by the channel name.

Analysis of the data flow diagram and the process behaviors regarding channel input shows that every cycle in the diagram contains at least one channel where the corresponding reader process provides unconditional input. Furthermore, every process provides at least bounded conditional input. As a consequence, a blocking situation on a cycle can never occur, as long as a process is still willing to output. Therefore a stable state without active communications implies that all the processes shown in Figure 3 are waiting for new inputs. These input channels include the set on vital channels. This completes the proof sketch of Theorem 1 (detailed proofs can be found in [4]).
□

3.7 Liveness Induction

Another result which we applied during the FML analysis was developed by Roscoe (detailed in [7]). It allows to derive livelock freedom of a collection of *triple-disjoint* communicating processes:

Definition 1 (Triple-Disjoint Process System) *A system of processes is called triple-disjoint, if no communication requires the participation of more than two processes.*

Theorem 2 *Suppose $V = \langle P_1, \ldots, P_n \rangle$ is a triple-disjoint system of non-divergent processes such that for every P_i in V*

$$P_i \backslash (\bigcup_{j<i} (\alpha(P_i) \cap \alpha(P_j)))$$

is divergence-free. Then

$$(P_1 \parallel \ldots \parallel P_n) \backslash H \quad \textit{where } H = (\bigcup_{j \neq i} (\alpha(P_i) \cap \alpha(P_j)))$$

is also divergence-free.
□

In the above theorem $\alpha(P_i)$ $(1 \leq i \leq n)$ denotes the alphabet of P_i. Informally speaking, this theorem states that if no process P_i in the system can perform an infinite sequence of communications with P_j (for all $j \leq i$), then the system is livelock free with respect to its incoming and outgoing channels.

Theorem 2 is useful in many cases, but finding an order on the processes of a parallel system may be very ineffective. An FML lane consists of 8 processes which have a complicated communication behavior. Consequently, it is very difficult to arrange them in a proper order for applying the theorem. One could try a large number of combinations (in our application at worst 8!); there are 7 proof obligations for each combination.

For this reason we have developed a new technique, *liveness induction*, which can be used to verify livelock freedom at the channel level. To verify that process P is livelock free with respect to channels of interface I, the induction is based on a set $\mathcal{L}$ of channels, inductively defined by

1. The interface I is contained in $\mathcal{L}$.
2. If $c_1, \ldots, c_n$ are channels in $\mathcal{L}$ and there exist a channel d which is not an interface channel and

   the number of communications via channel d is bounded by a function depending on the number of communications via channels $c_1, \ldots, d_n$,

   then d is contained in $\mathcal{L}$, too.
3. No other channel is in $\mathcal{L}$.

Liveness induction is now applied according to the rules of the following theorem (see Peleska et.al. [14]):

Theorem 3 *Let $P_1, \ldots, P_n$ form a triple-disjoint system of concurrent processes, such that each P_i is livelock free with respect to its local interface $I_i =_{df} \alpha(P_i) - L_i$, where L_i is the set of its internal channels. Then $(P_1 \parallel \ldots \parallel P_n)$ is non-divergent with respect to global interface $I \subseteq \bigcup_{i=1}^{n} I_i$, if $\mathcal{L} =_{df} \bigcup_{i=1}^{n} I_i$ can be inductively constructed according to the rules given above.*
□

This theorem can be proved by induction on the number of processes involved based on the compositionality of bounding functions. If communications on channel c_2 is bounded by communications on c_1 with bounding function f_1, and communications on c_3 is bounded by communications on c_2 with bounding function f_2, then communications on c_3 is bounded by communications on c_1 with $f_1 \circ f_2$ as a bounding function.

Liveness induction was used for the proof of FML in the following way: After showing the livelock freedom of each top-level process of FML we treat all the processes except the process *Link Interface* as a single process, say $REST$.Then $SYS = \langle REST, \mathcal{A}(\textit{Link Interface})\rangle$ is a triple-disjoint system. The following assertion can be proved using FDR:

$$\mathcal{A}(\textit{Link Interface}) \backslash (\alpha(\mathcal{A}(\textit{Link Interface})) \cap \alpha(REST)) \text{ is free of livelock}$$

If we can show that $REST$ is livelock free, then the whole system is livelock free, too, using Theorem 2.

The next verification target is to show the livelock freedom of $REST$, which is reached by liveness induction using Theorem 3. At the beginning, $\mathcal{L}$ contains all channels in the interface of $REST$, such as $ToInDist[0..6]$. Then, all channels bounded by the channels in $\mathcal{L}$ are contained in $\mathcal{L}$. In this way, we show that all channels involved in the communications of $REST$ are contained in $\mathcal{L}$.

Note that liveness induction is a forward analysis for livelock freedom, and contrasts with the approach described in the next section.

3.8 Dependency Analysis

Whereas the software structure of FML allowed to use the repeated abstraction by using generic processes multiplexer/concentrator and livelock induction, the complex communication behavior of AVI required a different approach. It turned out that there are three obstacles:

- even some of the main processes are too complex to be verified using model-checking directly;
- it is not possible to decompose the task into subtasks based on the compositional theory since the communication behavior turns out to be too complex;
- for the same reason no further abstraction is possible.

This combination of problems prevents the applicability of liveness induction as presented in the previous section. As an alternative a special form of dependency analysis was used for investigating cycles in the communication graph of processes which could not be checked using FDR directly due to the size of the state space. This analysis is a backwards analysis that determines all possible chains of communication events that may lead to a specific communication. The analysis technique is derived from tree analysis techniques as the *Fault Tree Analysis* (for the use of Fault Tree Analysis in software applications see for example Lyu [13]).

Consider a set of processes $P_1, \ldots, P_n$ which have been proved to be free of livelock individually. Let $SYS = P_1 \parallel \ldots \parallel P_n$ be a net of these processes with internal synchronization communications $\mathcal{L}$. The *communication graph* of SYS displays the channels and the direction of communication between the processes $P_i, i \in \{1, \ldots, n\}$. If this graph for SYS contains cycles it is not necessarily the case that these cycles can actually occur. If this is the case for all cycles SYS is livelock free. For the analysis of AVI there are two reasons that a cycle can not actually occur:

- the CSP processes involved do not produce traces that correspond to the cycle; this is for example the case if one of the communications does not depend on its predecessor in the cycle,
- the cycle is possible in the CSP process, but additional information derived from the occam code allow to exclude the occurrence of this situation.

For the analysis of AVI most of the cycles could be invalidated due to the first cause. The approach is illustrated in the following using a small example from AVI.

The structure of module BC (broadcast unit) of AVI is presented in the communication graph of Figure 4. The individual processes `Pi`, $i \in \{0, \ldots, 3\}$ can easily be checked for livelock freedom using FDR, i.e. it is guaranteed that no non-terminating internal communications can occur within the `Pi`. Each of these processes is itself a system of several subprocesses, which are denoted $P_{i1}, \ldots, P_{in_i}$. In order to ensure that the whole BC module is free of livelock it suffices to prove that none of the cycles in the graph can actually occur[1].

Regard for example the cycle involving `P0` and `P1`. A livelock situation can only occur if output `cmd` directly depends on the events `c_stat`, `real.reset.01`, or `real.reset.02` and vice versa. But this is not the case as code inspection of the CSP abstraction shows: the output of `cmd` only depends on the external input `to_dev`. In order to prove this it is necessary to trace the preceding communications leading to the output through the internal subprocesses of `P0`.

This type of analysis can be performed systematically for all cycles. In general it is only necessary to find one edge in a cycle that does not depend on its predecessors. This breaks a cycle and makes it uncritical with respect to livelock. The analysis itself is documented using techniques gleaned from

[1] This does not guarantee a fair treatment of the inputs; such a property has to be proved using different means.

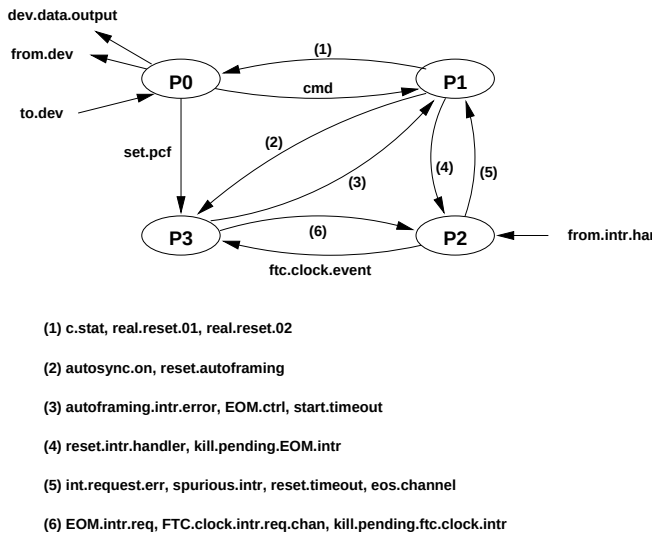


Figure 4: BC communication graph

fault-tree analysis. For each relevant communication it is traced by which other communications, resp. sequence of communications it can be triggered. The following notations are used:

For channels $ch_1, \ldots ch_n$ and process P

$$ch_1 \xleftarrow{\boxed{\text{P}}} ch_2$$

denotes that in P an output $ch1$ is produced if beforehand an input on channel $ch2$ did occur.

$$\begin{array}{l} ch_1 \\ \quad \xleftarrow{\boxed{\text{P}}} ch_2 \\ \quad \ldots \\ \quad \xleftarrow{\boxed{\text{P}}} ch_n \end{array}$$

means that an output on channel ch_1 is only possible after any of the inputs $ch_i, i \in \{2, \ldots, n\}$.

As an example the dependencies of event `kill_pending_EOM_intr` from `P1` to `P2` are presented here:

$$\begin{array}{l} \text{kill\_pending\_EOM\_intr} \\ \quad \xleftarrow{\boxed{\text{P12}}} \text{start\_timeout.1} \end{array}$$

```
    [P31]
    <-- ftc_clock_event
        [P22a]
        <-- from_intr_handler [OK]
[P12]
<-- start_timeout.2
    [P33]
    <-- (P33loop after enabled) start_subframe_timer
        [P31]
        <-- ftc_clock_event
            [P22a]
            <-- from_intr_handler [OK]
```

Note that it is necessary to trace the communications through several subprocesses not only of process `P1` and `P2` but also of process `P3`. Furthermore, it is essential that the dependency analysis is complete, i.e. that every sequence of communication leading to the event under consideration are investigated. This means that in the example above `start_timeout.1` and `start_timeout.2` are the inputs to process `P1` which can cause an output of `kill_pending_EOM_intr`.

In the case of `BC` and other units of the AVI this kind of analysis was used to ensure that most of the cycles in the communication graphs do not occur and to identify critical cycle situations as well. For AVI all the latter situations could be excluded due to further reasoning based on the original code.

4 Conclusion

4.1 Summary

The main obstacle for the analysis of deadlock or livelock freedom or other properties of realistic industrial systems is their size and complexity. Without suitable approaches for splitting the task into manageable subtasks the attempt of using formal methods and related tools in this area is futile. For our project it proved to be necessary to use a combination of techniques in order to decompose the goal appropriately. These techniques comprise

- abstraction of occam code to CSP,
- use of the compositional proof theory of CSP,
- abstraction to process instances of generic theories,
- model checking (for sufficiently small subprocesses),
- liveness induction
- dependency analysis

Figure 5 gives an overview of the methods used for the communication behavior analysis as well as the other verification and test goals for the FTC system.

Verification Goal	Verification Method
deadlock freedom	CSP: generic theories - abstraction - model checking - algebraic laws - compositional theory- cycle analysis - liveness induction
livelock freedom	as for deadlock analysis plus dependency analysis
absence of bottlenecks	stochastic petri nets
correct implementation of Byzantine Protocol and failure detection	abstraction - model checking - compositional proof theory
correct implementation of application services	verification using Hoare Logic
"most non-deterministic admissible software/hardware integration is correct"	Hardware-In-The-Loop-Testing

Figure 5: Goals of FTC Analysis

4.2 Some Experiences

Starting point for the livelock analysis for both AVI and FML were the abstractions that had been used in the deadlock analysis (see Buth et.al. [4]). Due to the hierarchical architecture of both components the freedom of livelock for each of the main processes was tried to be established first. During this phase some adjustments in the abstractions were necessary to eliminate non-determinism and formalize the new proof obligations. One example is the explicit introduction of timer processes in order to avoid divergence through timer dependent events. These did not pose any problems for the deadlock analysis.

During this first step it became soon obvious that for the subprocesses of both FML and AVI it was not feasible to use model-checking directly, even in cases where it was possible for the deadlock analysis. On the one hand this is due to the changes in the abstraction which enlarge the state space, on the other hand the problem arises since the states themselves are larger. The reason for this is that livelock analysis uses the *failure-divergence* model of CSP, while the deadlock analysis could be performed within the *failures* model. The internal representation of the states has to contain the additional information about the divergence sets and thus is larger. In some cases it was thus not possible to derive even the results for the main processes of FML and AVI. Additionally it was necessary to find a suitable approach of combining the results from the livelock analysis of the main processes to obtain a result for the two units as a whole.

In general two different approaches are possible in this situation:

- further abstraction and exploitation of the theorems for preserving the results under refinement,
- further decomposition of components, separate analysis for each basic unit, and derivation of properties for the combined units.

While the first proved to be a suitable way of dealing with the main processes of FML the complex communication behavior of AVI made it necessary to pursue the second approach. In both cases it was necessary to employ suitable means for combining the results for the overall unit.

4.3 Future Work

While there is tool support for model checking, namely the system FDR, the other techniques had to be applied manually. Future work will comprise the search and if necessary development of tools supporting these tasks.

One useful tool would be one that interactively guides the abstraction process, not only for occam programs but for other compatible languages as well. In order to gain more confidence in the abstraction process, it will be necessary to theoretically justify the preservance of specification properties by the abstraction steps. The idea is to classify abstractions with respect to livelock, deadlock, and general safety properties. This will allow to reach a basis for the other methods in a systematic and even tool supported way.

Another aspect of abstraction is the use of generic theories. Here it will be necessary to look for more generic patterns and their specific properties. Some such patterns like buffer, systolic arrays, and multiplexer could be found in the software of our project, others will be found during other case studies. Again it will be useful to classify these theories with respect to their specific properties in order to develop heuristics for their application in the context of abstractions. Furthermore, it will be necessary to prove the properties for the generic theories in a formal way. We would like to employ tools like HOL, Isabelle, or PVS for this task. Some work in this direction can for example be found in Buth et al. [1].

Additional abstraction methods for abstracting from irrelevant details can be used for certain verification obligations that are not covered by refinement relations. During the verification of the Byzantine Agreement Protocol implementation we have developed several such abstraction methods. For example, *data abstraction*, which abstracts a process from concrete data information of communications, is proved to be useful for protocol verifications, where some of the properties of message distributions can be verified without the data information of messages. For different applications different abstraction methods have been or will be developed, which together with compositional theories and generic theories will form a framework for our future activities in this area.

Tool support is also feasible for the dependency analysis. A rough idea is to use an approach with three phases:

- in the first phase the graph is build from the abstraction, i.e. the CSP processes;
- in the second phase all cycles are marked;
- in the third phase the cycles are investigated with respect to the dependencies of their constituent edges.

This kind of backwards cycle analysis could be complemented by a forward analysis for determining the bounds of the communication load of the cycles. This could help or replace the livelock induction process. Tools for these tasks might be found with groups working with graph transformation systems or in the area of graph theory in general.

References

[1] Buth, B., Cardell-Oliver, R., Peleska, J.: Combining tools for the verification of fault-tolerant systems. In Berghammer, R., Buth, B., Peleska, J. (eds.), *Tools for Software Development and Verification*, volume 1 of *BISS Monographs*. Shaker-Verlag, 1998.

[2] Buth, B., Kouvaras, M., Peleska, J., Shi, H.: Deadlock analysis for a fault-tolerant system. In Johnson, M. (ed.), *Algebraic Methodology and Software Technology. Proceedings of the 6th International Conference, AMAST'97*, number 1349 in LNCS, pages 60–75. Springer, December 1997.

[3] Buth, B., Peleska, J., Shi, H.: Combining Methods for the livelock analysis of a fault-tolerant system. In Haeberer, A.M. (ed.), em Algebraic Methodology and Software Technology. Proceedings of the 7th International Conference, AMAST'98, number 1548 in LNCS, pages 124–139. Springer, January 1999.

[4] Buth, B., Peleska, J.: *Daimler-Benz Aerospace – Project DMS-R, FTC Development – Verification of Avionics Interface AVI.* Technical Report, JP Software-Consulting, (1996).

[5] Buth, B., Peleska, J., Shi, H.: *Daimler-Benz Aerospace – Project DMS-R, FTC Development – Fault Management Layer (FML): Verification of Deadlock Freedom.* Technical Report, JP Software-Consulting, (1996).

[6] Daimler-Benz Aerospace: *DMS-R FTC Detailed Design Document Volume 3 (FML Software)*

[7] Dathi, N.: *Deadlock and Deaklock-Freedom,* Oxford University, D. Phil Thesis,1990.

[8] Davies, J.: *Specification and Proof in Real-Time CSP.* Cambridge University Press (1993).

[9] Formal Systemes: *Failures Divergence Refinement FDR2* Preliminary Manual. Formal Systems (Europe) Lts (1995).

[10] Hoare, C.A.R.: *Communicating Sequential Processes.* Prentice-Hall International (1985).

[11] inmos ltd. occam *2 Reference Manual. Series in Computer Science*, Prentice Hall International, 1988.

[12] Lamport, L., Shostak, R., Pease, M.: *The Byzantine Generals Problem*, In: ACM Transactions on Programming Languages and Systems, Vol.4, Nr. 3, (1982)

[13] Lyu, M. R. (ed.): *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, Computing McGraw-Hill

[14] Peleska, J., Shi, H.: *Daimler-Benz Aerospace – Project DMS-R, FTC Development – Fault Management Layer (FML): Verification of Livelock Freedom.* Technical Report, JP Software-Consulting, (1998).

[15] Roscoe, A. W.: *Model-Checking CSP.* In: A Classical Mind, Eassys in Honour of C.A.R. Hoare. Prentice-Hall Internationaal (1994).

[16] Roscoe, A. W.: *CSP and determinism in security modelling.* In: IEEE Symposium of Security and Privacy, (1995).

[17] Roscoe, A. W.: *The Theory and Practice of Concurrency* Prentice-Hall International (1997).

[18] Shi, H., Peleska, J.: *Daimler-Benz Aerospace – Project DMS-R, FTC Development – Fault Management Layer (FML): Verification of Byzantine Agreement Protocol Implementation.* Technical Report, JP Software-Consulting, (1998).

[19] Twele,L., Schlingloff, H., Szczerbicka,H.: *Performability Analysis of an Avionics-Interface.* In: Proc. IEEE Conf. on Systems, Man and Cybernetics; San Diego, N.J., pp. 499-504, (Oct. 1998)

Lowering Maintenance Costs by Code Clone Removal

Ira Baxter, A. Yahin, L. Moura, M. Sant'Anna

May 26, 1999

12th INTERNATIONAL SOFTWARE QUALITY WEEK (QW'99)

© 1999, Semantic Designs, Inc.

Code Clones in Software Systems

- **Fact: ~10% of a large application system is *cloned* code!**
  - **Exact copies: caused by inlined code and lazy programmers**
    - **[Lague97] 6-7% *function* copies 15M SLOC across 6 releases, 3 years**
  - **Near misses: "copy-n-hack" mode (good and bad reuse)**

```
s=s-n;
p->queue->length= // adjust queue size
      p->queue->length+k;
…
s=s-n;
p->queue->length= // adjust queue size
      p->queue->length+k;
```

Sample Exact Clones

```
s=s-n;
p->queue->length= // adjust queue size
      p->queue->length+k;
…
qp=p->queue; // get queue pointer
s=s-n;
qp->length= // adjust queue size
      qp->length+k;
```

Sample Near-Miss Clones

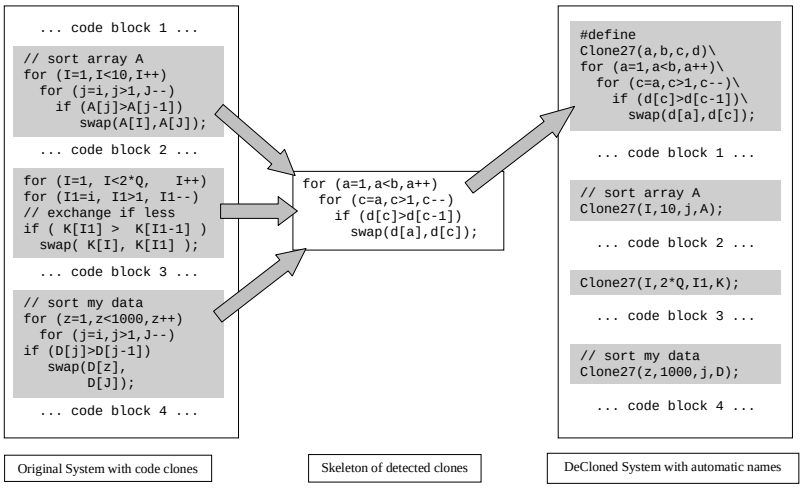
- ***Managing such clones could provide significant savings***

© 1999, Semantic Designs, Inc.

Managing Clones…

- **Problems with clones**
  - **A fixed bug in one code block often missed in its clone**
    - **Organization pays TWICE to fix: dollars, time**
    - **Different fixes --> TWO code blocks that are even less clone-like**
    - **Cost in customer perception *is worse than twice!***
  - **A clone is often a hint at a missed domain abstraction**
    - **Capture as procedure can clarify code and reduce errors**
    - **Can be used to enhance design and train new programmers**
- **Significant savings possible by managing clones**
    - **Locate exact clones for repair: [Lague97] ≥2000 chances over 6 releases**
    - **Eliminate clones**
    - ***Annual cost savings ~~ amount of removed code***

Goal: Remove Clones from a System



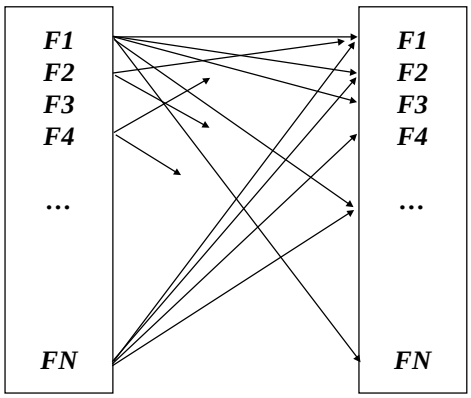
Basic Steps to Manage Clones

- Comparing every "fragment" to every other "fragment"
  - Problem: Approximating semantic idea of "same" in practice
  - Problem: Defining "fragment": line, block, procedure, ...
  - Problem: Ignoring accidental properties of fragment (text properties: spacing, comments, ...)
  - Problem: Composing matching fragments into larger fragments
  - Problem: Overlapping clone fragments
  - Problem: Detecting Near Misses
  - Problem: Very expensive!
- Discovering the commonality between clones
- Name that clone!
- Replacing clone instances with equivalent abbreviation

© 1999, Semantic Designs, Inc.

Basic Clone Detection is Expensive!

N^2 Line Comparisons for 10^6 SLOC is 10^{12} steps!

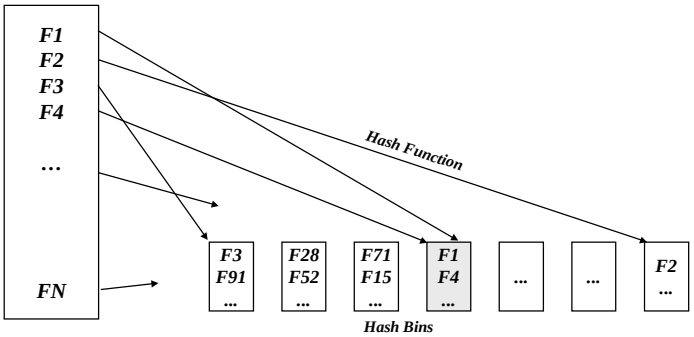


Comparing every fragment to every other fragment of a system

© 1999, Semantic Designs, Inc.

Standard Trick: Use Hashing to Control Costs

- **Compute Hash Code for each value to compare**
- **Put values into bins according to hash codes**
- **Compare only values in the same bins** ***(F1, F4)***
- *N*(N/(# buckets)) Comparisons with 10^6 SLOC, 10^5 buckets = 10^7 steps*



Clone Detection in Text Files: Problems

- **Confused by blanks, comments, line breaks, strings (relatively easily fixed)**
- **Can't detect clones that don't start/end on line boundaries ("C" language)**
- **Can't detect clones involving commutative operators (a+b vs. b+a)**
- **Confused by identical identifiers distinguished by language (local variables)**
- **Extremely difficult to detect "near misses"**
- **No support for managing clones after detection**

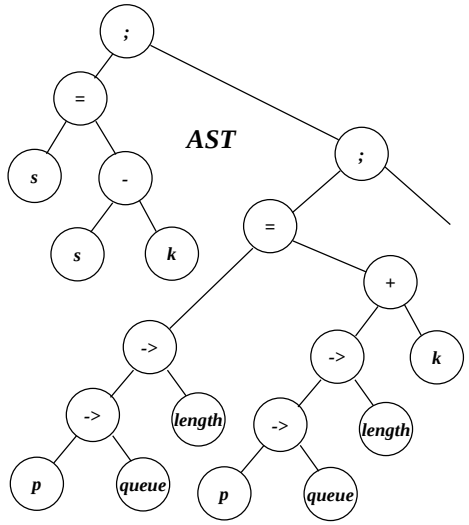
```
…                                          HashCode  CloneSize
        s=s-k;                               6254       ~
p->queue->length= // adjust queue size      12834       ~
       p->queue->length+k;                   83291       ~
…
{ s=s-n;                                     72824       ~
  // Inline for speed:                       60225       ~
  p->queue->length=n+p->queue->length;       40029       ~
};
```

Idea: Use Compiler Abstract Syntax Trees (AST) to avoid Lexical problems

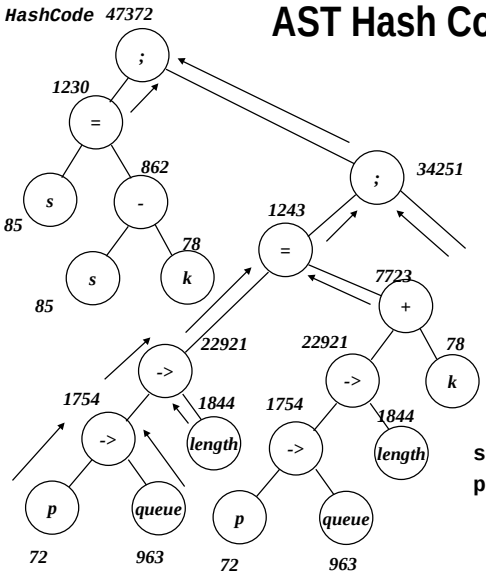
Source Text

```
s=s-k;
p->queue->length=
        // adjust queue size
         p->queue->length+k;
```

Compiler-style "parse"



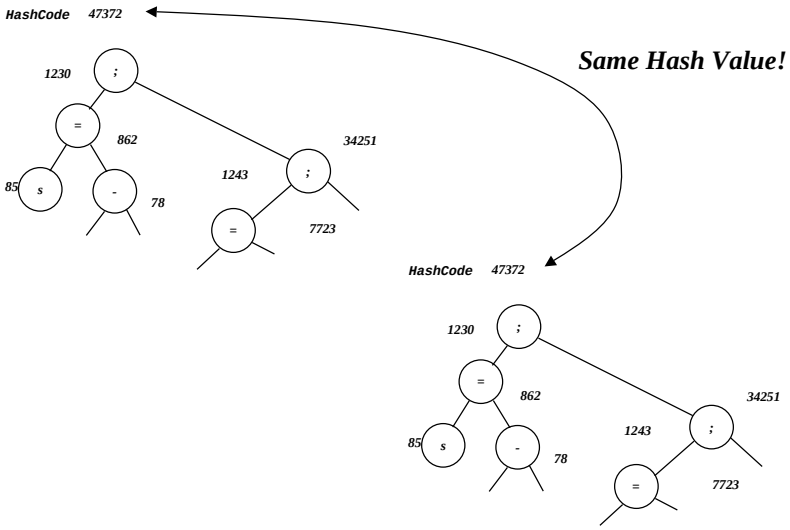
AST Hash Codes



Compute Hash for Tree by combining Hash values of subtrees. Compute bottom up for speed.

```
s=s-k;
p->queue->length=
        // adjust queue size
         p->queue->length+k;
```

Exact Clone Detection using AST Hash Codes



Exact Clone Detection on ASTs/DAGs

1. Parse Source files into Abstract SyntaxTrees (DAGs)
2. For each subtree, hash into a bucket
3. For each subtree i:
    - If node i marked as subclone, mark i's children as subclones
    - If node i marked as clone, done with i
    - For each subtree j≠i in same bucket as i
        - if EqualTrees(i,j) // can stop on clone children having same clone master
            - Mark i as (master) clone of i; mark j as clone of i
            - Mark children(i), children(j) as subclones
4. For each subtree i:
    - if node(i) marked as clone and node(i) not marked as subclone
    - Print "Node ",i," is a clone", Node(i)

Improvements over Source Line method:

- Language parser automatically ignores blanks, comments, linebreaks
- Language name resolution (DAG) not confused by identical identifiers
- DAG version can unify near-misses written using different names
- Can be extended to handle near-misses as parameters
- Can be extended to handle simple commutative operations
- Can be extended to account to type information

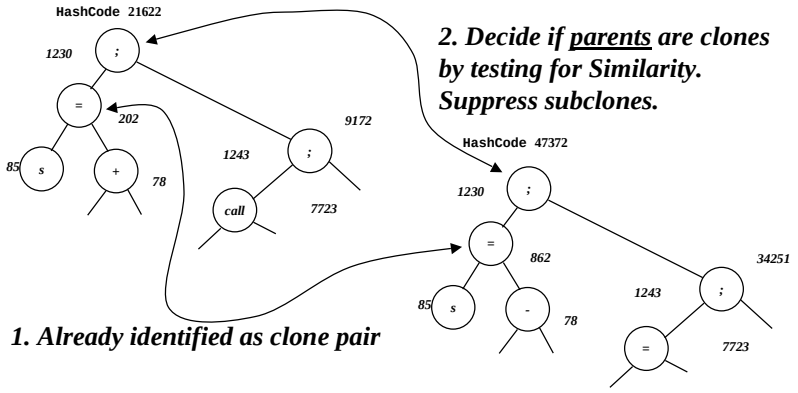
Basic Near-Miss Clone Detection Algorithm

- Comparing for *exact* matches doesn't work!
- Define Similarity = 2 S / (2S + L + R)
  - S = number of shared nodes
  - L = number of non-shared nodes held in left subtree
  - R = number of non-shared nodes held in right subtree
- Define hashing function to *ignore* small subtrees (e.g., identifiers, expressions, ...)

```
Clones=∅
For each subtree i:
        If mass(i)>=MassThreshold
        Then hash i to bucket
For each subtree i and j in the same bucket
        If CompareTree(i,j) > SimilarityThreshold
        Then { For each subtree s of I
                        If IsMember(Clones,s)
                        Then RemoveClonePair(Clones,s)
                For each subtree s of j
                        If IsMember(Clones,s)
                        Then RemoveClonePair(Clones,s)
                AddClonePair(Clones,i,j)
             }
```


ASTs with different hash codes?

- Can be clones, but won't be detected by previous scheme
- Solution: Check *parents* of clones pairs for similarity



Sample COBOL Clones

```
Similarity = .99178082191781
from 35179 to 35204 file = example.cbl

  9700-OUTPUT-REPORT-TOTALS.
      MOVE CURRENT-COLLEGE-ID
          TO REPORT-CODE1  REPORT-CODE2  REPORT-CODE3.
      SET EDIT-ERROR-LITERAL-INDEX TO 1.
      SET DISTRICT-COUNT-ROW-INDEX TO 1.
      PERFORM 9710-OUTPUT-TOTALS1
          UNTIL EDIT-ERROR-LITERAL-INDEX > 30.
      PERFORM 9720-OUTPUT-TOTALS2.
      PERFORM 9730-OUTPUT-TOTALS3.

  9710-OUTPUT-TOTALS1.
      MOVE REPORT-SUM1 TO TOTALS-ID1.
      MOVE ELEMENT-NUMBER (EDIT-ERROR-LITERAL-INDEX)
              TO DED-NUMBER.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  1)
              TO EXCEPT-COUNT.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  2)
              TO UNKNOWN-COUNT.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  3)
              TO REASON-COUNT.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  4)
              TO GRP3-COUNT.
      WRITE REPORT-TOTALS-RECORD1.
      SET EDIT-ERROR-LITERAL-INDEX UP BY 1.
      SET DISTRICT-COUNT-ROW-INDEX UP BY 1.
```

#1

#2

Report by College-ID

```
#2  ==>  30
    ==>  17
---------
#1
==>     CURRENT-COLLEGE-ID
==>     HOLD-DISTRICT-ID
```

```
-----------------------------------
from 15368 to 15393 file = example.cbl

  9700-OUTPUT-REPORT-TOTALS.
      MOVE HOLD-DISTRICT-ID
          TO REPORT-CODE1  REPORT-CODE2  REPORT-CODE3.
      SET EDIT-ERROR-LITERAL-INDEX TO 1.
      SET DISTRICT-COUNT-ROW-INDEX TO 1.
      PERFORM 9710-OUTPUT-TOTALS1
          UNTIL EDIT-ERROR-LITERAL-INDEX > 17.
      PERFORM 9720-OUTPUT-TOTALS2.
      PERFORM 9730-OUTPUT-TOTALS3.

  9710-OUTPUT-TOTALS1.
      MOVE REPORT-SUM1 TO TOTALS-ID1.
      MOVE ELEMENT-NUMBER (EDIT-ERROR-LITERAL-INDEX)
              TO DED-NUMBER.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  1)
              TO EXCEPT-COUNT.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  2)
              TO UNKNOWN-COUNT.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  3)
              TO REASON-COUNT.
      MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX  4)
              TO GRP3-COUNT.
      WRITE REPORT-TOTALS-RECORD1.
      SET EDIT-ERROR-LITERAL-INDEX UP BY 1.
      SET DISTRICT-COUNT-ROW-INDEX UP BY 1.
```

Report by District-ID

The generated COPYLIB

```
9700-OUTPUT-REPORT-TOTALS .
    MOVE PARAMETER-1
        TO REPORT-CODE1  REPORT-CODE2  REPORT-CODE3 .
    SET EDIT-ERROR-LITERAL-INDEX TO 1 .
    SET DISTRICT-COUNT-ROW-INDEX TO 1 .
    PERFORM 9710-OUTPUT-TOTALS1
        UNTIL EDIT-ERROR-LITERAL-INDEX > PARAMETER-2    .
    PERFORM 9720-OUTPUT-TOTALS2 .
    PERFORM 9730-OUTPUT-TOTALS3 .
9710-OUTPUT-TOTALS1 .
    MOVE REPORT-SUM1 TO TOTALS-ID1 .
    MOVE ELEMENT-NUMBER ( EDIT-ERROR-LITERAL-INDEX )
            TO DED-NUMBER .
    MOVE DISTRICT-COUNT ( DISTRICT-COUNT-ROW-INDEX 1 )
            TO EXCEPT-COUNT .
    MOVE DISTRICT-COUNT ( DISTRICT-COUNT-ROW-INDEX  2 )
            TO UNKNOWN-COUNT .
    MOVE DISTRICT-COUNT ( DISTRICT-COUNT-ROW-INDEX 3 )
            TO REASON-COUNT .
    MOVE DISTRICT-COUNT ( DISTRICT-COUNT-ROW-INDEX 4 )
            TO GRP3-COUNT .
    WRITE REPORT-TOTALS-RECORD1 .
    SET EDIT-ERROR-LITERAL-INDEX UP BY 1 .
    SET DISTRICT-COUNT-ROW-INDEX UP BY 1 .
```

Source file change

```
    MOVE DISTRICT-INT-CNT (11)
         TO INT-CNT-OUT-B.
    WRITE PRINT-RECORD-2 FROM INTEGRITY-ERROR-B
          AFTER ADVANCING 2 LINES.
    MOVE DISTRICT-INT-CNT (12)
         TO INT-CNT-OUT-C.
    WRITE PRINT-RECORD-2 FROM INTEGRITY-ERROR-C
          AFTER ADVANCING 2 LINES.
```

```
    MOVE DISTRICT-INT-CNT (11)
         TO INT-CNT-OUT-B.
    WRITE PRINT-RECORD-2 FROM INTEGRITY-ERROR-B
          AFTER ADVANCING 2 LINES.
    MOVE DISTRICT-INT-CNT (12)
         TO INT-CNT-OUT-C.
    WRITE PRINT-RECORD-2 FROM INTEGRITY-ERROR-C
          AFTER ADVANCING 2 LINES.
```

```
9700-OUTPUT-REPORT-TOTALS.
    MOVE HOLD-DISTRICT-ID
        TO REPORT-CODE1, REPORT-CODE2, REPORT-CODE3.
    SET EDIT-ERROR-LITERAL-INDEX TO 1.
    SET DISTRICT-COUNT-ROW-INDEX TO 1.
    PERFORM 9710-OUTPUT-TOTALS1
        UNTIL EDIT-ERROR-LITERAL-INDEX > 17.
    PERFORM 9720-OUTPUT-TOTALS2.
    PERFORM 9730-OUTPUT-TOTALS3.


9710-OUTPUT-TOTALS1.
    MOVE REPORT-SUM1 TO TOTALS-ID1.
    MOVE ELEMENT-NUMBER (EDIT-ERROR-LITERAL-INDEX)
          TO DED-NUMBER.
    MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX, 1)
          TO EXCEPT-COUNT.
    MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX, 2)
          TO UNKNOWN-COUNT.
    MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX, 3)
          TO REASON-COUNT.
    MOVE DISTRICT-COUNT (DISTRICT-COUNT-ROW-INDEX, 4)
          TO GRP3-COUNT.
    WRITE REPORT-TOTALS-RECORD1.
    SET EDIT-ERROR-LITERAL-INDEX UP BY 1.
    SET DISTRICT-COUNT-ROW-INDEX UP BY 1.
```

```
COPY CDR_clone10
     REPLACING PARAMETER-1
                   BY == HOLD-DISTRICT-ID ==
               PARAMETER-2
                   BY == 17 ==
```

```
9720-OUTPUT-TOTALS2.
    MOVE REPORT-SUM2 TO TOTALS-ID2.
    MOVE DISTRICT-INT-CNT (1)
        TO INTEGRITY-ERROR-COUNT.
    MOVE '01' TO INTEGRITY-ERROR-CODE.
    WRITE REPORT-TOTALS-RECORD2.
    MOVE REPORT-SUM2 TO TOTALS-ID2.
    MOVE DISTRICT-INT-CNT (2)
        TO INTEGRITY-ERROR-COUNT.
    MOVE '02' TO INTEGRITY-ERROR-CODE.
```

```
9720-OUTPUT-TOTALS2.
    MOVE REPORT-SUM2 TO TOTALS-ID2.
    MOVE DISTRICT-INT-CNT (1)
        TO INTEGRITY-ERROR-COUNT.
    MOVE '01' TO INTEGRITY-ERROR-CODE.
    WRITE REPORT-TOTALS-RECORD2.
    MOVE REPORT-SUM2 TO TOTALS-ID2.
    MOVE DISTRICT-INT-CNT (2)
        TO INTEGRITY-ERROR-COUNT.
    MOVE '02' TO INTEGRITY-ERROR-CODE.
```


Detector Engineering

- **Parsing**
  - **YACC/Lex disappointing: grammar ambiguities --> hacking**
  - **Use Tomita (GLR) parsing style**
  - **Handle ambiguities in parsing by choosing canonical form**
  - **Capture lexical information: spacing, comments, formats**
  - **Preprocessor Directives**
    - **Programmers program in *C+preprocessor*, not C!**
    - **Add "Structured Preprocessor" directives to grammar**
      - **accepts those that fit well into structure of language**
      - **expand/fix those few directives that don't fit**
- **AST scale: 1M SLOC in 2 Gb RAM**
  - **Implement in PARLANSE, parallel programming language**
    - **8 way SMP Windows N/T --> Linear speedup in detection time**

Clone Detection/Removal Statistics

Application	*Language*	*SLOC*	*Removable SLOC (est.)*	*Removed SLOC*
Process Control	C	427921	54346 12.7%	N.A. (pre removal capability for C 6/98)
PARLANSE Compiler	C	42413	4133 9.7%	2568 6.0%
CCC Registration	COBOL85	77631	15500 20.0%	15734 20.3%
SLC Insurance	PROGRESS	356413	58783 16.5%	N.A. (not yet performed 2/99)
SAPV Insurance	PROGRESS	132656	16251 12.3%	N.A. (not yet performed 2/99)
SAPV+SLC	PROGRESS	489069	105644 21.6%	N.A. (not yet performed 2/99)

Conclusions

- **Can detect code clones on 1M SLOC scale using ASTs**
  - **Algorithms: Basic, Sequence detection, Generalization**
  - **Scale requires hashing, tuples, parallelization**
  - **1Gb RAM, 40 minutes CPU**
  - **Languages 2/99: C/C++, COBOL/2, Fortran95, Java, PROGRESS, Pascal**
  - **Confirm other work: *5-20% code redundancy***
- **Clone removal is practical on-scale!**
  - **Requires special parsers to allow regeneration of lexical information**
  - **Language-specific function abstraction is hard: dataflow analyses**
  - **Interactive mode to choose abstraction method and name**
  - **Tool for C,C++ exists 2/99, COBOL/2 by 5/99**
    - **Other languages OK: need syntax, abstraction proposers**
  - **Demo copy downloadable from *www.semdesigns.com***
- ***Clone Removal can lead to significant maintenance savings***

Lowering Maintenance Costs by Code Clone Removal

Ira D. Baxter[1] **Andrew Yahin[1]**
Leonardo Moura[2] **Marcelo Sant'Anna[2]**

[1] Semantic Designs
12636 Research Blvd. Suite C214
78759-220 Austin Texas
512 250 1018
{idbaxter, ayahin}@semdesigns.com
http://www.semdesigns.com

[2] Departamento de Informática
Pontifícia Universidade Católica
Do Rio de Janeiro
R. Marquês de São Vicente, 225
22453-900 Rio de Janeiro, Brazil
+55 21 529 9396
{moura, draco}@les.inf.puc-rio.br

Keywords
Software maintenance, clone detection, software evaluation, Design Maintenance System

Abstract

A commonly used method for enhancing an application program is to copy ("clone") an existing fragment of the code and customize the copy. Existing research suggests that a considerable fraction (5-20%) of the source code of large-scale computer programs is redundant cloned code. Such clones increase the cost of maintaining code, by raising the bulk of the code that maintainers must keep up to date. This paper shows how a tool, CloneDR™, can automatically detect and replace fragments and their clones by their shared abstraction in the form of a procedure or a macro, preserving the program function. *Removal of all code clones by their abstraction promises decreased software maintenance costs of possibly the same magnitude as the reduction.* Additional benefits accrue from having code that is easier to explain because some of the code fragments have been identified directly as useful abstractions, and the early discovery of relatively common errors that occur in the code cloning process.

The method uses a language grammar as a parameter to achieve language-independent detection of clones despite spacing, comment and name changes. The paper sketches how the CloneDR harnesses these ideas, and provides statistics on redundancy levels in large, real C and COBOL programs of significant scale and shows actual code size reductions for these cases. It outlines some of the additional obstacles to actual clone removal (such as scale, disabled abstractions, preserving comments, etc.). Finally, the paper shows how manual inspection of the detected clones can sometimes lead to the discover y of latent software bugs.

1 Introduction

Data from previous work [Lague97, Baker95] suggests that a considerable fraction (5-10%) of the source of large computer programs is duplicated code. Programmers routinely perform ad hoc code reuse by brute-force copying code fragments that implement actions similar to their current need, and performing a cursory (often empty!) customization of the copied code to the new context.

The act of copying indicates the programmer's intent to reuse the implementation of some abstraction. The act of pasting is breaking the software engineering principle of encapsulation. What the programmer should do is to declare and invoke the abstraction, but this is often not done, even though the language already offers suitable abstraction tools, such as macros or procedures. Often, the programmer doesn't quite know what the abstraction is, and so he clones and edits. Regardless of the reason, cloning is commonplace and unlikely to disappear via fiat.

The part of the cloned code that is unchanged is redundant. Detection and replacement of such redundant code by subroutine calls, in-lined procedure calls, macros, or other equivalent shorthand that effect the same result, promises decreased software maintenance costs corresponding to the reduction in code size. *Much of present software engineering is focused on finding small-percentage process gains. Mechanical clone detection and removal offers the potential for a relatively easy, annually recurring 10% savings or better.*

We define an *idiom* as a program fragment that implements a recognizable concept (data structure or computation). A *clone* is a program fragment that identical to another fragment. A *near miss clone* is a fragment, which is nearly identical to another. Clones usually occur when an idiom is copied and optionally edited, producing exact or near-miss clones.

Previous clone detection work was limited to detection of either exact textual matches, or near misses only on complete function bodies. This paper presents practical methods, using abstract syntax trees (ASTs), for detecting exact and near miss clones for arbitrary fragments of program source code. The method uses a variation of the well-known compiler method for detecting common sub-expressions [Aho85], which determines exact tree matches essentially by hashing. A number of adjustments are needed to detect clones in the face of commutative

operands, near misses, and statement sequences. .It also presents methods for constructing abstractions and removing the redundancy from the source using conventional transformational methods.

A tool, CloneDR, using these detection techniques has been applied to large programs in a number of real programming languages, and exhibits similar detected levels of duplication found by previous work. Unlike previous work, this tool is also capable of removing the code clones.

We additionally suggest that clone detection could also be useful in producing more structured code, as well as discovering domain concepts and their idiomatic implementations.

Section 2 discusses the causes of clones. Section 3 discusses why we chose AST-based clone detection. Section 4 describes the basic AST clone detection algorithm. Section 5 builds on the basic method to detect clone sequences. Section 6 discusses detection of near-miss clones generalized from previously discovered clones. Section 7 considers how abstractions for clones can be constructed. Section 8 discusses the problems of engineering a clone detector/remover for scale. Section 9 reports the results of applying the clone detector to running software systems, and analyzes the results. Section 10 discusses the relation between clones and domain concepts. Section 11 reports possible future work. Section 12 describes related work.

2 Why Do Clones Occur?

Software clones appear for a variety of reasons:

- Code reuse by copying pre-existing idioms
- Coding styles
- Instantiations of definitional computations
- Failure to identify/use abstract data types
- Performance enhancement
- Accidents

State of the art software design has structured design processes, and formal reuse methods. Legacy code (and, alas, far too much of new code) is constructed by less-structured means. In particular, a considerable amount of code is or was produced by ad hoc reuse of existing code. A programmer intent on implementing new functionality find some code idiom that performs a computation nearly identical to the one desired, copies the idiom wholesale and then modifies it in place. The availability of screen editors with “copy” and “paste” functions hasten the ubiquity of this event.

In large systems, this method may even become a standard way to produce variant modules. When building device drivers for operating systems, much of the code is boilerplate, and only the part of the driver dealing with the device hardware needs to change. In such a context, it is commonplace for a device driver author to copy entirely an existing, well-known, trusted driver and simply modify it. While this is actually good reuse practice, it exacerbates the maintenance problem of fixing a bug found in the “trusted” driver by replicating its code (and reusing its bugs) over many new drivers.

Sometimes a “style” for coding a regularly needed code fragment will arise, such as error reporting or user interface displays. The fragment will purposely be copied to maintain the style. To the extent that the fragment consists only of parameters this is good practice. Often, however, the fragment unnecessarily contains considerably more knowledge of some program data structure, etc. A required adjustment to the style must then be copied to every instance.

It is also the case that many repeated computations (payroll tax, queue insertion, data structure access) are simple to the point of being definitional. As a consequence, even when copying is not used, a programmer may use a mental macro to write essentially the same code each time a definitional operation needs to be carried out. If the mental operation is frequent, he may even develop a regular style for coding it. Mental macros produce near-miss clones: the code is almost the same ignoring irrelevant order and variable names.

Some clones are in fact complete duplicates of functions intended for use on another data structure instance of the same type; we have found many systems with poor copies of insertion sort on different arrays scattered around the code. Such clones are an indication that the data type operation should have been supported by reusing a library function rather than pasting a copy.

Some clones exist for justifiable performance reasons. Systems with tight time constraints are often hand-optimized by replicating frequent computations, especially when a compiler does not offer in-lining of arbitrary expressions or computations.

Lastly, there are occasional code fragments that are just accidentally identical (false positive clones), but in fact are not clones. When investigated fully, such apparent clones just are not intended to carry out the same computation. Fortunately, as size goes up, the number of accidents of this type drops off dramatically.

Ignoring accidental clones, the presence of clones in code unnecessarily increases the mass of the code. This forces programmers to inspect more code than necessary, and consequently increases the cost of software maintenance. One could replace such clones by invocations of clone abstractions once the clones can be found, with potentially great savings.

3 Clone Detection using Abstract Syntax Trees

The basic problem in clone detection is the discovery of code fragments that compute the "same" result. To do this, we must first fragment the program in parts we are willing to compare, and then determine if fragment pairs are equivalent. Since determining even if single fragment halts is generally impossible, we cannot in general determine that two arbitrary program fragments halt under the same circumstance, and thus it is impossible in theory to decide that they compute identical results. Since false negatives are acceptable (as engineers we have no choice), deep semantic analysis conservatively bounded by time limits could be used for equivalence detection. However, considerable infrastructure may be required—in the form of semantic definitions, theorem provers, etc. In practice, we are willing to give up detecting complete semantic equivalence because many clones come about due to copy-and-paste editing processes, which can be detected more easily.

Simpler definitions of code equivalence may suffice if too many false positives are not produced. This suggests clone detection by more syntactic methods. One can go as far as comparing source lines. Source line equality assumes that the cloning process introduced no changes in identifiers, comments, spacing, or other non-semantic changes, and thus limits clone detection to exact matches. Consequently, it fails to detect near-miss clones. Closer to full semantics but still a practical possibility would be to compare program representations in which control and data flows are explicit.

Semantic Designs is building transformational tools (DMS) to help modify large software systems [Baxter97]. Such tools typically parse source programs into ASTs as a first step before transformation. Because DMS produces ASTs for arbitrary grammars easily, we chose to investigate comparing syntax trees. This had the immediate advantage of directly avoiding confusing but uninteresting changes at the lexical level.

As a first step in the clone detection process, the source code is parsed and an AST is produced for it. After that, three main algorithms are applied to find clones. The Basic Clone detection algorithm detects sub-tree clones; this is purely syntactic. The Sequence Clone Detection algorithm, is concerned with the detection of variable-size sequences of sub-tree clones, and is used essentially to detect statement and declaration sequence clones; sequence detection is a weak but practical concession to program semantic equivalence. The Clone Generalization algorithm looks for more complex near-miss clones by attempting to generalize combinations of already-detected clones.

Clones resulting from all three algorithms are candidates for removal by abstraction, and can be prettyprinted. This clone set is passed to a language specific abstractor algorithm whose job is to propose the appropriate language abstraction, and actually remove the clones.

4 Finding Sub-tree Clones

In principle, finding sub-tree clones is easy: compare every subtree to every other sub-tree for equality. In practice, several problems arise: near-miss clone detection, sub-clones, and scale. Near misses we handle by comparing tress for similarity rather than exact equality. The sub-clone problem is that we wish to recognize maximally large clones, so clone subtrees of detected clones need to be eliminated as reportable clones.

To determine if two trees are clones, rather than comparing trees for exact equality, we compare instead for similarity, using a few parameters. The similarity threshold parameter allows the user to specify how similar two sub-trees should be. The similarity between two sub-trees is computed by the following formula:

Similarity = 2 x S / (2 x S + L + R)

where:

S = number of shared nodes from root
L = number of different nodes in sub-tree 1
R = number of different nodes in sub-tree 2

A mass threshold parameter specifies the minimum sub-tree mass (number of nodes) value to be considered, so that small fragments of code (e.g., subexpressions) are rejected as potential clones.

The scale problem is much harder. For an AST of N nodes, comparing all pairs of trees is O(N^3), and, empirically, a large software system of M lines of code has N=4*M AST nodes (if we consider comparing sequences of trees, the process is O(N^4)!). For $N=10^6$ SLOC, this is on the order of 10^{24} units of work, which is prohibitively large.

We tackle this problem by hashing trees to equivalence classes, and compare class members as possible clones. The approach is based on the tree matching technique for building DAGs for expressions in compiler construction [Aho86], which allows the straightforward detection of exact sub-tree clones. If we hash sub-trees to B buckets, then only those trees in the same bucket need be compared, cutting the number of comparisons by a factor of B. We choose a B of approximately the same order as N; in practice, B=10% N means little additional space at great savings in terms of computation. We have found that the cost of comparing individual trees averages much closer to a constant, as most trees that differ do so at the subtree root, rather than O(N), and so hashing allows this computation to occur in practice in time O(N).

This approach works well for finding exact clones. When locating near-miss clones, hashing on complete subtrees fails precisely because a good hashing function

includes all elements of the tree, and thus sorts trees with minor differences into different buckets. We solved this problem by choosing an artificially bad hash function. This function must characterized in such a way that the main properties one wants to find on near-miss clones are preserved. As we described in Section 2, near miss clones are usually created by copy and paste procedures followed by small modifications. These modifications usually generate small changes to the shape of the tree associated with the copied piece of code. Therefore, this kind of near-miss clone often has only some small sub-trees which differ. Based on this observation, a hash function that ignores small sub-trees is a good choice. We have successfully used a hash function that ignores only the identifier names (leaves in the tree). Thus our hashing function puts trees which are similar modulo identifiers into the same hash bins for comparison.

We combine these ideas to give the Basic Clone detection algorithm in Figure 1. This algorithm is straightforward. In Step 2, the hash codes for each sub-tree are computed to place them in the respective hash bucket. This step ignores small subtrees, thus implementing the mass threshold in a way that further reduces the number of comparisons required considerably, as the vast majority of trees are small. After that, every pair of sub-trees located in the same hash bucket is compared; if the similarity between them is above the specified threshold, the pair is added to the clone list, and all respective sub-clones are removed.

```
1. Clones=∅
2. For each subtree i:
      If mass(i)>=MassThreshold
      Then hash i to bucket
3. For each subtree i and j in the same bucket
      If CompareTree(i,j) > SimilarityThreshold
         Then { For each subtree s of i
                  If IsMember(Clones,s)
                  Then RemoveClonePair(Clones,s)
                For each subtree s of j
                  If IsMember(Clones,s)
                  Then RemoveClonePair(Clones,s)
                AddClonePair(Clones,i,j)
              }
```

Figure 1 - Basic Subtree Clone Detection Algorithm

By enumerating the trees from leaves upward, the mass threshold test can be implemented very cheaply. The parents of all subtrees with sufficient mass also have sufficient mass.

An enhancementm enables the detection of similar trees containing commutative operators such as add ("+"). This will detect re-ordered operands in "mental macros"; in reused-code, it is rare for a programmer to swap operands while editing. Implementing this requires merely that such tree node types be identified as commutative, that the hashing function produces identical values on commutative trees, and that the similarity function tries all child orderings on commutative subtrees.

5. Finding Clone Sequences

The preceding section shows how to detect clones as trees, and is purely syntax driven. In practice, we are interested in code clones that have some semantic notion of sequencing involved, such as sequences of declarations or statements. In this section, we show how to detect statement sequence clones in ASTs using the Basic Algorithm as a foundation.

Such sequences show up in ASTs not as arbitrary trees, but rather as right- or left-leaning trees with some kind of identical sequencing operator as root. Sequences of sub-trees appear in an AST as a consequence of grammar rules which accept sequences of zero or more syntactic constructs. These sequence rules are typically expressed by the use of left or right recursive rules. When a parser generator produces parsers that automatically generate AST, it is common, as in our case, that the trees have a left-leaning shape. Consider Figure 2, which shows a pair of short sequences of statements along with their corresponding trees. Note that the left-leaning tree reverses the order of the statements because of the order in which the parse reductions are done as determined by the controlling grammar rule. In this example, nodes labeled with a **";"** are sequence nodes for statements belonging to a compound statement. Because a generic clone detector has no idea which tree node types constitute sequence nodes, these nodes must be explicitly identified to the clone detector.

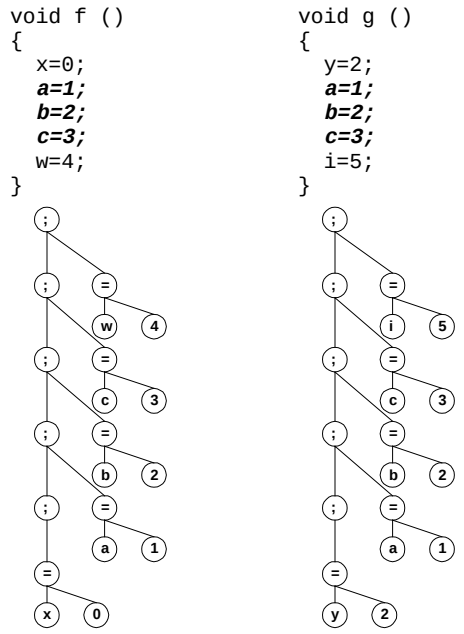


Figure 2 – Example of clone sequence

Such sequences of sub-trees are not strictly trees, and consequently require a special treatment. In Figure 2, the Basic Algorithm finds three clones corresponding to the

assignment statements for variables a, b and c. But, it is unable to detect the clone sequence, because it is not a single sub-tree, but rather a sequence of sub-trees. The sequence detection algorithm copes with this problem by comparing each pair of sub-trees containing sequence nodes, looking for maximum length sequences that

```
void f ()
{
  x=0;
  if (d>1)
  {
    y=1;
    z=2;
  }
  else
  {
    h=2;
    z=1;
    y=3;
  }
}
```

The program has three sequences.

List Structure:

1. {x=0; if(d>1) … }

   hashcodes = 675, 3004

2. {y=1; z= 2;}

   hashcodes = 1020,755

3. {h=2; z=1; y=3;}

   hashcodes = 786, 756, 704

Figure 3 – List structure example

encompasses previously detected clones. Short sequences (especially those of length one) are not interesting sequence clones. A minimum-sequence length threshold parameter controls the minimum acceptable size of a sequence.

To find sequence clones, we build a list structure where each list is associated with a sequence in the program, and stores the hash codes of each sub-tree element of the associated sequence. Figure 3 shows an example of the list structure that is built. This list structure allows us to compute the hash code of any particular subsequence very quickly.

```
1. Build the list structures describing sequences
2. For k = MinimumSequenceLengthThreshold
       to MaximumSequenceLength
3. Place all subsequences of length k
   into buckets according to subsequence hash
4. For each subsequence i and j in same bucket
   If CompareSequences (i,j,k) >
             SimilarityThreshold
   Then { RemoveSequenceSubclonesOf(clones,i,j,k)
          AddSequenceClonePair(Clones,i,j,k)
       }
```

Figure 4 – Sequence detection algorithm

Figure 4 gives the Sequence Clone Detection algorithm. This algorithm compares each pair of sub-trees containing sequence nodes looking for the maximum length of possible sequencing that encompasses a clone. Whereas the Basic algorithm finds *three* clones in Figure 2, the sequence detection algorithm finds the sequence of length 3 comprising the assignments for variables a, b and c as a *single* clone. Following the requirement that larger clones subsume smaller ones, detecting this sequence immediately invalidates the clone status of the atomic statements found as clones by the Basic Algorithm.

A weakness of this scheme is that it does not does not have a method for detecting near miss sequence clones in which the sequences are of unequal length. This causes the algorithm to miss clones in which statements have been inserted or deleted. We don't know how common this situation is.

6. Generalizing clones

After finding exact and near-miss clones, we use another method (Figure 5) to detect more complex near-miss clones. The method consists of visiting the parents of the already-detected clones to check if the parents are near miss clones too. We also delete subsumed clones.

A significant advantage of this method is that any near-miss clones must be assembled from some set of exact sub-clones, and therefore no near-miss clones will be missed.

```
1.  ClonesToGeneralize=Clones
2.  While ClonesToGeneralize≠∅
3.   Remove clone(i,j) from ClonesToGeneralize
4.   If CompareClones(ParentOf(i), ParentOf(j))
        > SimilarityThreshold
     Then {
        RemoveClonePair(Clones,i,j)
        AddClonePair(Clones,
                     ParentOf(i), ParentOf(j))
        AddClonePair(ClonesToGeneralize,
                     ParentOf(i),ParentOf(j))
           }
5.  End While
```

Figure 5 – Detecting more complex clones

A similar generalization algorithm is needed to test the parents of detected sequence clones. The details have been omitted for space. The final detected clone set is the union of generalization algorithm for both subtree clones and sequence clones.

Statistically, most clones occur only in pairs, and a pairwise clone detector is adequate. In practice, we sometimes find a skewed distribution, in which some clones occur many times. Even if the final detected set has few multiply-occurring clones, the intermediate detection process often finds many small clones that are identical. This can have serious scale space consequences, as K identical blocks of code can produce O(K^2) clone pairs. The CloneDR tool actually keeps sets of similar clones rather than just pairs to avoid this problem.

7 Constructing Abstractions

After all clones are detected, with a suitable prettyprinter, one can display the detected clones and a language-independent skeletal macro from the abstraction tree that is shared by both subtrees of each pair of clones. The skeletal macro is simply the subtree the clones have in common, with parameters where the abstraction differs from the clone instance. Figure 6 shows an example of a near miss sequence clone detected by the CloneDR tool in

```
--------------------- CLONE ---------------------
 Similarity = 0.929411764705882
 From 13407 To 13424

Db_err error;
Bool fail;
if ( db_get_int ( DB_CF_VAC_GROUP_REGEN_IV ,
                  & error ) )
{
  vac_regen_group_ob ( );
  return ( 0 );
}
db_put_ts ( DB_VAC_OI_P5_CRYO_REGEN_IN_PROG_TSV ,
            TRUE , & error );
db_put_int ( DB_VAC_CRYO_P5_STATE_IV , VAC_REGEN ,
              & error );
CALL ( fac_n2 ( ) );
alm_activate ( ALM_VAC_P5_REGENING ,
               ALM_DONT_BLOCK , ALM_DONT_ACK ,
               60 , MDP_NULL , MDP_NULL );
return ( 1 )

-----------------------------------------------
 From 13208 To 13225

Db_err error;
Bool fail;
if ( db_get_int ( DB_CF_VAC_GROUP_REGEN_IV,
                  & error ) )
{
  vac_regen_group_ob ( );
  return ( 0 );
}
db_put_ts ( DB_VAC_OI_P4_CRYO_REGEN_IN_PROG_TSV ,
            TRUE , & error );
db_put_int ( DB_VAC_CRYO_P4_STATE_IV , VAC_REGEN ,
             & error );
CALL ( fac_n2 ( ) );
alm_activate ( ALM_VAC_P4_REGENING ,
               ALM_DONT_BLOCK , ALM_DONT_ACK ,
               60 , MDP_NULL , MDP_NULL );
return ( 1 )
```

Figure 6 – Sequence clones found (from Application)

an industrial control application. Figure 7 shows the skeletal macro generated by the clone detector for the clones in Figure 6, and the argument values that need to be bound to the abstraction parameters.

For languages such as C, trivial syntax modifications can almost always turn the skeletal macro into legal C preprocessor directives (Figure 8a). The detector applied to "C" programs will find clones in any subtree, both code and data; macro abstractions can be proposed in both places. The constructed macro can be given a (gensym'd) name, placed into a "ABSTRACTIONS.H" file along with other clone macro abstractions, and the clone site code replaced by a macro invocation with suitable parameters. The original source file must be modified to INCLUDE the "ABSTRACTIONS.H" file.

Complications sometimes arise in real languages when forming abstractions. For C, macros can abstract almost every clone, but a macro argument that would contain a comma is expressly forbidden by the language. While this

```
--------------------MACRO-------------------
Db_err error;
Bool fail;
if ( db_get_int ( DB_CF_VAC_GROUP_REGEN_IV,
                  & error ) )
{
  vac_regen_group_ob ( );
  return ( 0 );
}
db_put_ts ( #0 , TRUE , & error );
db_put_int ( #1 , VAC_REGEN , & error );
CALL ( fac_n2 ( ) );
alm_activate ( #2 , ALM_DONT_BLOCK ,
               ALM_DONT_ACK , 60 , MDP_NULL ,
               MDP_NULL );
return ( 1 );

--------------- MACRO BINDINGS --------------
#2 = ALM_VAC_P5_REGENING ,
     ALM_VAC_P4_REGENING
#0 = DB_VAC_OI_P5_CRYO_REGEN_IN_PROG_TSV ,
     DB_VAC_OI_P4_CRYO_REGEN_IN_PROG_TSV
#1 = DB_VAC_CRYO_P5_STATE_IV ,
     DB_VAC_CRYO_P4_STATE_IV

-----------------------------------------------
```

Figure 7 – A Skeletal Macro Abstraction

seems unusual, it is possible to detect near-miss clones in C function argument lists, which would can in turn cause the proposal of a C macro is whose argument would be itself a C argument list, which would contain the forbidden comma. In this case, the abstractor must reject a C macro as a viable abstraction. If an abstraction cannot be formed, then a detected clone simply cannot be removed. This type of problem sometimes prevents the CloneDR from achieving the removal of all the clones it can detect.

A language may offer several forms of abstraction. For C, this includes:

- Macros
- TYPEDEFs (only for exact clones)
- INCLUDE files (only for exact clones)
- Function calls

Each language abstraction type will have rules about where it can be legally used, and software engineers will have heuristics about whether it should be used. Macros may be used to abstract clones in the middle of a STRUCT definition, but a TYPEDEF would be better. For each language, the CloneDR tool implements abstraction-forming critics which can deny the formation of a particular type of abstraction.

Macros are useful as abstractions, but sometimes function abstractions are better, because of a desire to save space in the compiled program, an organizational desire to avoid macros, or the need to debug in clearly defined contexts. Forming function abstractions is considerably more complex than forming macro abstractions, because function bodies usually have different lexical scopes than the clone body expects. For each small data item outside

```
#macro CDRM297(CDR0,CDR1,CDR2)\
Db_err error;\
Bool fail;\
if ( db_get_int ( DB_CF_VAC_GROUP_REGEN_IV,\
                  & error ) )\
{\
  vac_regen_group_ob ( );\
  return ( 0 );\
}\
db_put_ts ( CDR0 , TRUE , & error );\
db_put_int ( CDR1 , VAC_REGEN , & error );\
CALL ( fac_n2 ( ) );\
alm_activate ( CDR2 , ALM_DONT_BLOCK ,\
               ALM_DONT_ACK , 60 , MDP_NULL ,\
               MDP_NULL );\
return ( 1 )


                        . . .

Former Line 13208
CDRM297(DB_VAC_OI_P4_CRYO_REGEN_IN_PROG_TSV,
        DB_VAC_CRYO_P4_STATE_IV
        ALM_VAC_P4_REGENING)

                        . . .

Former Line 13407
CDRM297(DB_VAC_OI_P5_CRYO_REGEN_IN_PROG_TSV,
        DB_VAC_CRYO_P5_STATE_IV,
        ALM_VAC_P5_REGENING)
```

Figure 8a – C Macro Abstraction and Invocations

```
int CDRFN316(int CDR0, CYRO_STATE* CDR1,int CDR2)
{  Db_err error;
   Bool fail;
   if ( db_get_int ( DB_CF_VAC_GROUP_REGEN_IV,
                  & error ) )
   {
     vac_regen_group_ob ( );
     return ( 0 );
   }
   db_put_ts ( CDR0 , TRUE , & error );
   db_put_int ( CDR1 , VAC_REGEN , & error );
   CALL ( fac_n2 ( ) );
   alm_activate ( CDR2 , ALM_DONT_BLOCK ,
               ALM_DONT_ACK , 60 , MDP_NULL ,
               MDP_NULL );
   return ( 1 )
}

                        . . .

Former Line 13208
 return
CDRFN316(DB_VAC_OI_P4_CRYO_REGEN_IN_PROG_TSV,
         &DB_VAC_CRYO_P4_STATE_IV
         ALM_VAC_P4_REGENING)

                        . . .

Former Line 13407
CDRFN316(DB_VAC_OI_P5_CRYO_REGEN_IN_PROG_TSV,
         &DB_VAC_CRYO_P5_STATE_IV,
         ALM_VAC_P5_REGENING)
```

Figure 8b – C Function Abstraction and Invocations

the scope of the clone body, which is only read in the clone body, the value of that data item is computed at the abstraction invocation site and passed to a value parameter to the abstraction. For each data item outside the scope of the clone body which is large or updated, a reference to that value can be formed at the abstraction invocation site and passed as an additional "lexical-access" parameter to the function. Returns inside the clone scope must be handled carefully. Control transfers outside the scope of the clone prevent formation of the function abstraction. Implementing such lexical-access parameters requires control and data flow analysis procedures which are beyond the scope of this paper. The detected clone of Figure 7, removed as a function abstraction, would appear as in Figure 8b.

The mechanism to produce function abstractions is about the same order of complexity as the clone detector. We are actively implementing this facility now.

8 Clone Remover Engineering

To build a practical clone detector and remover one must address other several issues:

- Parsing and building the AST
- Preprocessor directives
- PrettyPrinting with Fidelity
- Interactive Decloning
- Industrial scale source codes

Parsing the program suite of interest requires a parser for the language *dialect* of interest. While this is nominally an easy task, in practice one must acquire a tested grammar for the dialect of the language at hand. Often for legacy codes, the dialect is unique and the developing organization will need to build their own parser. Worse, legacy systems often have a number of languages and a parser is needed for each. Standard tools such as Lex and Yacc are rather a disappointment for this purpose, as they deal poorly with lexical hiccups and language ambiguities. In addition, the parser must accurately capture lexical information such as spacing, comments, and number formats, to allow later regeneration of the modified program text in nearly its original form.

In our Design Maintenance System™ (DMS™) tool [Baxter97] we use a variation of a Tomita-style parser [Tomita91], that can parse ambiguous grammars (such as C++) with impunity, cutting the time to develop a usable parser significantly. (We can presently parse/clone detect in C/C++, COBOL/2, Java, Fortran77/90/95 and PROGRESS). .In particular, we use the parsing algorithm of [Wagner97], which also conveniently builds the parse tree. However, when using a Tomita-style parser, special nodes called Symbol nodes, representing ambiguous parses, must be taken into account in the clone detection algorithm. We ignore ambiguous sub-trees by choosing a canonical son based on grammar rule number. Lexical formatting information is captured by DMS.

Preprocessor directives always complicate processing source files. For C, we use a combination of means:

- Added preprocessor directives to the language syntax, effectively extending the language to include well-structured directives;
- Built a custom preprocessor to pass well-structured directives to the parser, and expand the rest.
- Read INCLUDE files to get macro and type definitions. When detecting clones in the presence of INCLUDE files, the included text must be treated as part of the include file, not part of the including file.

Removing clones requires the original program be modified; modulo complex analyses such as dataflow, this is relatively easy given an AST. But the modified program must be regenerated, preserving spacing, comments and lexical formatting where given by the original program, and prettyprinting newly generated code. In order to reproduce the program, each program fragment must also be able to identify the file from which it originally came, so that a corresponding set of files are accurately reproduced. For DMS, we use a TeX-like layout language to describe how to do prettyprinting, and integrate all these information sources.

One defect of purely automated clone removal is that the abstractions get poor names (Figures 8a and 8b), as the tool has no clue as to the abstraction purpose. A related problem has to do with how to abstract a clone if multiple abstractions are possible, or indeed even whether to abstract every possible abstraction. Our CloneDR tool will carry out batch decloning, under the assumption that maintainers will rename the clones during later maintenance, but we also decided that an interactive mode is necessary. In such a mode, the CloneDR presents each detected abstraction to an engineer, and allows him to choose the dispositioning method (or reject it), and name the abstraction and its parameters. (This is a little easier than it might seem, as the abstractions usually represent some useful application concept.) We implement abstraction critics that eliminate formation and interactive proposal of a particular type of abstraction type if it would be illegal or inappropriate.

Program scale is a problem for any clone detection scheme. The CloneDR can presently handle about 1 MSLOC code in 1GB RAM. We believe one can perform clone detection on some 10 million lines in 2Gb with careful design and one tree node per processor cache line. A disk-based implementation could handle larger size, but batch performance is likely to be much worse because one would replace RAM-based hash bucket access and tree walking times with disk I/O times. A reasonable alternative might be to batch declone once, and then incrementally compute the hash codes and do clone detection for changed modules.

As the computation demand is severe, we have implemented CloneDR in PARLANSE, Semantic Design's parallel programming language for symmetric multiprocessors. We achieve a linear speedup on an 8-way 200 Mhz Pentium Pro. With this speedup, it takes some 40 minutes to parse and detect the clones in 350KSLOC. Batch clone removal after detection takes negligable time.

9 Clone Detection Applied

In this section, we discuss various statistics reported by the clone detector when applied to real, working programs of scale. First, we discuss an industrial process control application coded in C. Then we discuss comparative numbers across several programming languages.

The clone detector was applied to the source code for a industrial process-control system having approximately 400 KSLOC of C code. The system was created 7 years ago by reusing and modifying a then 3-year-old code of a system with similar functionality. 15 programmers presently maintain it. The programmers who did the port were not those who developed the original system. The software is partitioned into subsystems according to function.

Figure 9 shows the percentage of cloned code in 19 different subsystems, computed as the ratio of redundant SLOC and subsystem SLOC. Three subsystems have approximately 28% cloned code. Subsystem 15 was created by cloning device driver code capable of handling one I/O port to allow handling of a second port. Done under schedule pressure, it was already known to be highly redundant. We observe that schedule pressure continuously prevented this redundancy from ever being fixed, and the clone tool could do this automatically. Subsystem 16 is code that services a number of devices and was created similarly to subsystem 15, as confirmed by the developers. The average clone percentage is 12.7% for all subsystems.

If software maintenance costs were distributed evenly across source, this suggests 12% savings in maintenance costs. In practice, some systems are more troublesome than others. Subsystem 2 was difficult to maintain and accounts for a large percentage of defects written against the system. A reduction in code here could have disproportionate savings in costs.

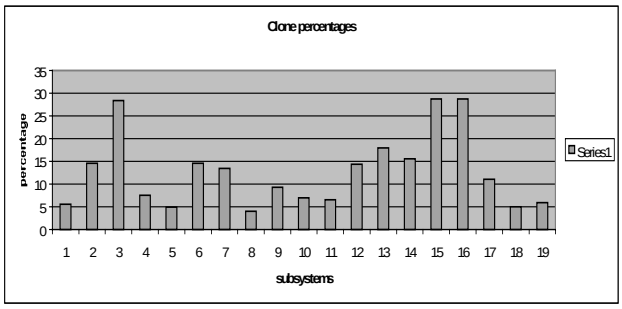


Figure 9 – Clone percentages

An interesting question is how often code is copied, and how much is copied. Figure 10 shows the number of times that clones of various sizes (SLOC) were found. The three largest subsystems are represented in this graph. Clone sizes larger than 25 are rare and are therefore not shown. However, within subsystem 15 an unexpected clone of size 497 was found; it seems common to find a few very large clones. We conclude that most clones are relatively small in size, on the order of 10 SLOC. This isn't very surprising, given that a clone is often the implementation of some application abstraction.

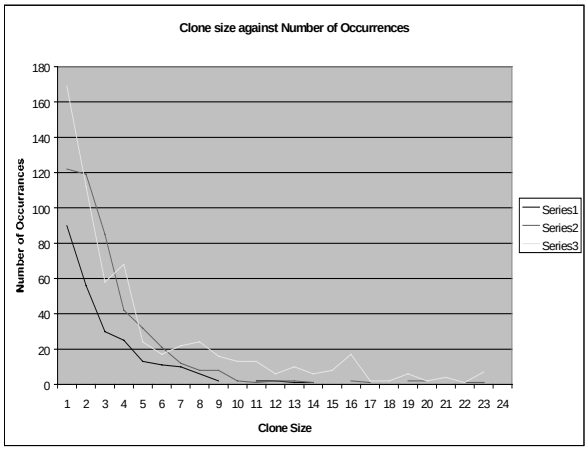


Figure 10 – Clone size against number of occurrences

One hopes that as a system matures, it becomes better organized. Figure 11 shows the changes in percentage of clones for four fairly volatile subsystems over four releases with a periodicity of approximately 6 months. Three of the four subsystems show a steady improvement in clone density, but one shows a sharp increase. All could clearly use a clone removal pass.

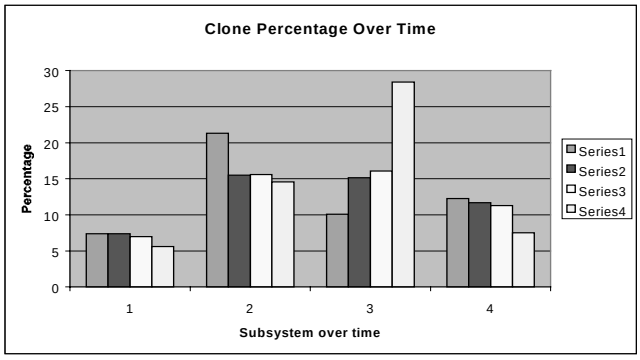


Figure 11 – Clone percentage over time

We had hypothesized that larger files might have larger clone percentages, because of opportunity and complexity. Figure 12 compares clone percentage with subsystem size in KSLOC. Although the 3 largest files have the largest number of clones, no pattern is recognizable. We conclude that clone ratios are relatively independent of (sub)system size: "Clones happen".

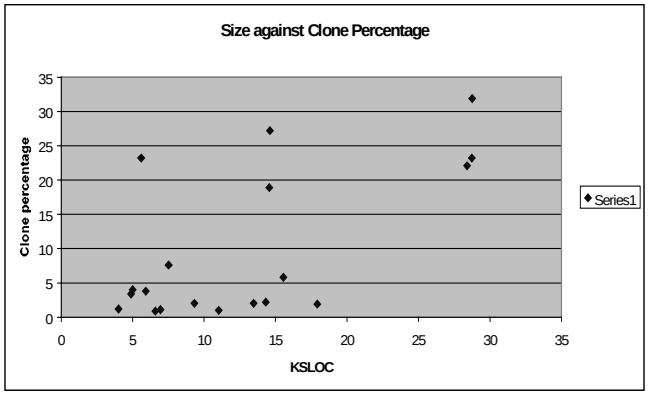


Figure 12 – Size against clone percentage

We draw the following conclusions from the data:

1. Higher percentages of cloned code were found in newer subsystems, which might be expected to be freer of clones because they are closer to a clean, original design. However, it appears that cloning is used to get the code to work, the code is released as soon as it is working, and then improved as it stabilizes. [Lague97] also found this.

2. We examined the clones in the code for subsystems having higher clone percentages. Many clones were separate functions performing identical operations on different data types or devices having left versus right symmetry. This appears to have been a stylistic choice, perhaps made in the interests of apparent modularity or clarity (although the cost of maintaining all those copies argues against actual modularity). It is interesting to notice that subsystems having a greater degree of this stylistic copying are also the subsystems which change most infrequently, largely low-level utility programs (data access, data initialization, etc). This supports finding 1) in that utility code is often stable, therefore rarely modified and therefore not 'cleaned up'.

3. There is no correlation between subsystem size and clone percentage. One might expect to have more clones in larger subsystem because of the conceptual complexity of grasping the entire subsystem. It appears that deadline pressures, similarities in functionality, and programmer style have more effect on clone density than code size.

We have seen various clone detection statistics for a single large application program in a single language. We now consider detection and removal statistics for programs in several languages. Figure 12 shows several applications, the PARLANSE Compiler source, a statewide community college course registration program, and two related insurance applications from the same vendor, SLC and

SAPV, written in very different programming languages.

Application	Lang-uage	SLOC	Est. RSLOC	Actual RSLOC
P0 Compiler	C	42413	4133 9.7%	2568 6.0%
CCC Registration	COB-OL85	77631	15500 20.0%	15734 20.3%
SLC	PROG-RESS	356413	58783 16.5%	na
SAPV	PROG-RESS	132656	16251 12.3%	na
SAPV+SLC	PROG-RESS	489069	105644 21.6%	na

Figure 12 – Detection/Removal across Languages

The table shows program sizes, estimated removable source lines of code based purely on clone detection, and actual removed source lines of code after abstraction by macro for C, and by COPY LIB for COBOL (we do not have clone abstraction running for PROGRESS just yet). The removed source line count varies from the estimated because the estimator uses the skeleton macro, and the actual abstraction usually requires some extra source code overhead for macro declaration and vertical whitespace gaps.

All the programs show a significant volume of removable clones. We think this makes a clear economic case for clone removal. The COBOL application has a high rate of redundancy because the data declarations for several modules have a clearly-copied large table of school districts, which would clearly be better as a COPYLIB. The insurance applications individually seem to have typical redundancy numbers by themselves, but are quite high when combined. In fact, SAPV was derived from SLC, and although they are separate products, they have considerable redundancy. The vendor organization owning these products would do well to remove clones across both systems and combine them under a single configuration manager.

We do not have numbers for Actual Removal for abstraction by function yet. We expect them to be a little lower than the macro abstraction numbers, due to more restrictions on applicability.

10 Clones as Domain Concepts

We believe that a clone detector tool can be of great value for a domain analyst capturing domain concepts from the source code of similar systems. A key issue in domain analysis is the need to understand commonalities in concepts and implementations across families of software systems. As defined earlier, idioms are program fragments that implement specific concepts in a given application domain. It is reasonable to believe that some of the clones found in a system are realizations of these concepts, precisely because they are in repeated demand for carrying out the application. Such understanding can be used as an interesting starting point for the abstraction of domain concepts from source code [DeBaud97].

A domain-analysis process step might include running a clone analysis for each of several similar systems, followed by interviews with domain specialists. During these interviews, the cloned code is shown and the specialists are asked to provide possible abstractions motivating such implementations. The results would be domain concepts and their abstract implementations as generalized from the clone instances. The tool presented in this paper represents a step forward in the development of auxiliary tools for the hard task of finding commonalties in similar systems.

```
device.mode = MOVE_ABS;
device_xf.position = (int) db_get_int(
               DB_XFER_ARM_LOAD_IV, &dbstat);
move_swap (& device_xf);
data = (char *) & device_xf;
data_size = sizeof(Msg_device_move);
msg_send((Process_id) DI_DEVICE_XCR,
               data ,(Uint16) data_size,
               MSG_NO_FLAGS ,
               MSG_XCR_DEVICE_XFER_MOVE);
printf("Xfer Arm rotated to load \n");
```

Figure 13 - A domain idiom found by clone detection

An example of a clone representing a domain idiom is shown in Figure 13. These seven lines were repeated eight times within one source file. Here the control system is commanding an arm, which transfers product through the system, to move to the load position, a basic command for that device.

The fact that clones are often implementations of domain concepts can, remarkably, used to find bugs in programs being decloned. We have observed that when a parameterized abstraction is being formed, what is often of interest are the parameters themselves. Given an abstraction, one can often spot bad argument values directly; such failings are much harder to see in the clone code. More importantly, a poorly modified clone usually has abstractions with spurious extra parameters, or parameters with obviously incorrect values. We found a number of bugs in rarely-used error reporting logic in our PARLANSE compiler when we investigated abstraction invocations with unusual arguments.

11 Future Work

The CloneDR tool presently implements clone abstraction by macro. The most obvious next step is to automate the removal of detected clones using function abstractions, and we are completing that for COBOL/2. An interactive user interface is being implemented. We expect to implement macro abstraction for PROGRESS, and declone the SAPV and SLC systems using the interactive process.

The present model for Clone removal is a batch process, but this is a fairly expensive proposition, not so much for the computers but rather for the engineer who interactively must choose sensible names for several thousand clones in a 1MSLOC system. A friendlier method might integrate the clone remover into the software check-in process to eliminate clones as they arise.

There are a number of improvements one could make to the clone detector itself, the use of DAGs and/or dataflow graphs instead of trees and operations.

The algorithms we presented here will work more effectively on DAGs, with no change, than on trees. For DMS, we plan to convert identifiers in tree leaves into cross-links to the syntactic construct defining that identifier, forming DAGs from the parse trees. This has the advantage of preventing false-positive exact clones using accidentally the same textual name, but which actually refer to different identifiers according to the language scoping rules (Near miss clones that are parameterized by these non-identical names will still get found). Type information could be used to prevent detection of accidental clones, as many such clones use different types in their computation. In practice, we presently see few false positives, so the incremental value of adding this is unclear. For DMS, it will essentially be free, so it won't hurt us to try it.

What one would often like is the identification of two blocks of code that were cloned, and then patched differently. If the patch is the replacement of one language construct by another, our detector will find it. If the patch is the insertion of new code in the middle of a clone, forming a "split clone", our present detector will not be able to find the entire clone, but can identify the preceding and following fragments as clones. It would be convenient if the detector identified such split clones, as they are often an indication that a clone was defective, and only one instance was fixed.

A program representation in which control and data flows are explicit would allow a more semantic-oriented clone detector that would be insensitive to variable names and statement ordering. In particular, this would detect, as monoliths, clones that have been split by irrelevant insertions. It would also not be fooled by reordered data declarations or shuffled statements that achieve the same effect. For DMS, we will construct such representations, and so should be able to build more sophisticated detectors in the future.

12 Related Work

Neural nets have been proposed for clone detection [Barson95]. This requires training, performs poorly in the face of complex lexical differences that occur in real languages and programs, and does not scale to millions of lines of code. We think this path is a definite dead end.

A fast string-based method, DUP, for detecting exact clones and simple near-miss clones, was considered by [Baker95], who reported 13-20% clones, for a large application (1M SLOC). When producing clones, programmers often change white spacing (blanks, tabs, newlines) and comments, which will disable recognition based purely on strings. A simple lexical processor can overcome this particular problem. DUP actually compares strings of lexemes rather than strings of characters to combat this problem.

Their lexeme-based algorithm seems easy to implement and operates nicely on scale. On a positive note, it apears that DUP should be able to detect sequence clones., but has some shortcomings. First, it fails to detect clones which differ by other than a trivial substitution of one lexeme for an identifier. Second, based on the description, we believe that such a detector will also falsely detect nonsensical clones, such as the lexeme sequence frequently found in C programs at the end of a function: "**`}; int SomeName(`**" Furthermore, DUP's matching process cannot detect exact clones with commutative operators. An easy way to cure these problems is to take the language syntax into account, as we do. Our detector finds clones, which differ in complex language constructs, such as expressions or statements. In any case, more sophisticated clone detection, using name resolution, cannot be accomplished without parsing and applying deeper knowledge of the language scoping rules. [Johnson94] applies similar methods.

The method used by [Lague97] apparently parses the program text, and then computes a hash code for function bodies by forming a vector of predefined software metrics (e.g. SLOC, Halstead, etc). While this can be used for recognizing function clones, it appears to make the hashing function unnecessarily complicated without necessarily providing a good hash function. Implementing and running the clone detection process is not likely to be nearly as fast as a straightforward implementation like ours, which is important for the scale in which clone detection is interesting. It also has the defect that it appears to allow many false positives, which would require an engineers's attention to eliminate; we think this is bad for the engineer and defeats much of the purpose of an automated tool. (The CloneDR seems to produce very few false positives). We remark that comparing just function bodies, however,

cuts the volume of comparisons required, so perhaps this is not a problem if one wants to limit clone detection to just function bodies. Furthermore, it is limited to language constructs on which all the metrics apply, hinting at the reason it operates only on function bodies. Such a scheme probably would be ineffective for detecting data declaration clones (which could be removed in "C" by using TYPEDEFs). If a wide variety of metrics were used, it would be likely parsing is necessary to support one of them, and our scheme could be used directly instead. If parsing is not required, then lexical differences may damage clone detection ability as already suggested. Finally, one has to augment simple hashing schemes to reasonably detect near misses, or sequence clones. Our method will detect cloned expressions, statement sequences, data declarations, etc., anything which has a syntactic structure in a language.

Neither Baker nor Lague have actually achieved the promise of clone detection: the removal of the clones; we have done this via macros and are actively pursuing function abstraction. The CloneDR is the only clone detection program that uses parallelism to achieve its goal, and may be the only compiler-like tool anywhere to achieve good speedups on commodity hardware. Finally, the CloneDR applies easily to multiple languages, by virtue of being parameterized by the language grammar, and using the robust parsers for industrial languages encoded for DMS, such as C, C++, COBOL and FORTRAN.

Conclusions

A practical method for detecting near-miss and sequence clones on scale has been presented. The approach is based on variations of methods for compiler common-subexpression elimination using hashing. The method is straightforward to implement, using standard parsing technology, detects clones in arbitrary language constructs, and computes macros that allow removal of the clones without affecting the operation of the program.

We have applied the method to a number of real application programs in different programming languages, all of moderate to large scale, and confirmed previous estimates of clone density of 7-15%, suggesting there is a "manual" software engineering process "redundancy" constant. We offer evidence that organizations that have multiple related products will see even higher redundancy rates, and can therefore obtain correspondingly a larger benefit by clone removal. Automated methods can detect and remove such clones, lowering the value of this constant, at concomitant savings in software engineering or maintenance costs. Clone detection tools also have good potential for aiding domain analysis.

A commercial tool for carrying out Clone Detection and Removal, the CloneDR, is being prepared for market.

References

[Aho86] Alfred Aho, Ravi Sethi and Jeffrey Ullman, **Compilers, Principles, Techniques and Tools**, Addison-Wesley 1986.

[Baker95] Brenda Baker, On Finding Duplication and Near-Duplication in Large Software Systems, Working Conference on Reverse Engineering 1995, IEEE.

[Barson95] P. Barson, N. Davey, S. Field, R. Frank, D.S.W. Tansley, Dynamic Competitive Learning Applied to the Clone Detection Problem, Proceedings of International Workshop on Applications of Neural Networks to Telecommunications 2, 0-8058-2084-1, Lawrence Erlbaum, Mahwah, NJ 1995

[Baxter97] Ira Baxter and Christopher Pidgeon, Software Change through Design Maintenance, International Conference on Software Maintenance, 1997, IEEE.

[DeBaud97] Jean-Marc DeBaud, DARE: Domain-Aumented ReEngineering, Working Conference on Reverse Engineering, 1997, IEEE.

[Kontoginnis96] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, Pattern Matching for Clone and Concept Detection, Journal of Automated Software Engineering 3, 77-108, 1996, Kluwer Academic Publishers, Norwell, Massachusetts

[Johnson94] J.H. Johnson, Substring Matching for Clone Detection and Change tracking, Proceedings of the International Conference on Software Maintenance 1994, IEEE.

[Johnson96] H. Johnson, Navigating the Textual Redundancy Web in Legacy Source, Proceedings of CASCON '96, Toronto, Ontario, November 1996.

[Lague97] B. Lague, D. Proulx, E. Merlo, J. Mayrand, J. Hudepohl, Assessing the Benefits of Incorporating Function Clone Detection in a Development Process, International Conference on Software Maintenance 1997, IEEE.

[Tomita91] **Generalized LR Parsing**, Masaru Tomita ed., 1991, Kluwer Academic Publishers, Norwell, Massachusetts.

[Wagner97] Tim Wagner and Susan Graham, Incremental Analysis of Real Programming Languages, Proceedings 1997 SIGPLAN Conference on Programming Language Design and Implementation, June 1997, ACM

Automated Software Installation Testing

Christopher Agruss
Alexei Tcherkassov
Autodesk Incorporated

Quality Week, May 1999

The infinitesimal defect

”Little bugs have littler bugs,
upon their backs to bite 'em,
and the littler bugs have littler bugs,
and so ad infinitum.”

Verse from "There ain't no bugs on me”
-traditional children's song

Introduction

- ✔ Installers make a strong first impression
- ✔ Manual installer testing can be tedious
- ✔ Scope of this talk
  - Which aspects of installation testing lend themselves best to automated methods?
  - Appendixes for general installer testing checklists, tools, and other references

Overview

- ✔ Meta-automation and the installer
- ✔ Establishing machine base states
- ✔ Designing the automated system
- ✔ Selecting the appropriate tools
- ✔ Verifying the results
- ✔ Testing the uninstaller

Installer testing as meta-automation

- ✔ Installers have built-in automated tests
  - Can you think of any examples?
- ✔ One purpose of your test code is to verify the assumptions of the installer's own automated script

Establishing machine base states

- ✔ Return the machine's disk to a base state
  - Reformat hard disk
  - Install operating system
  - Create base image
- ✔ How often do you need to restore the full base state?
- ✔ See paper for Disk Imaging tool list in Appendix B

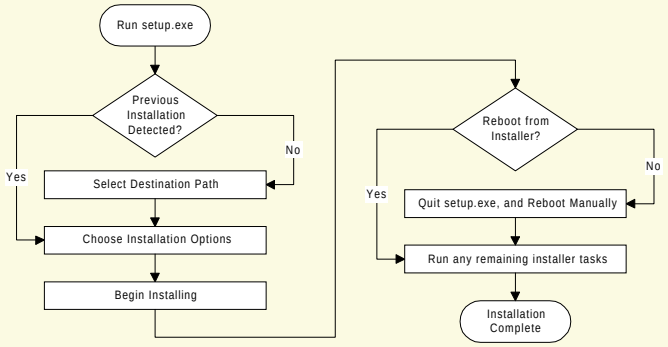
Select a tool to drive the installer

- ✔ Define the scope of your testing
  - responsible only for the end results?
  - need to test the user interface?
- ✔ Does your installer have a silent option?
- ✔ Two options to consider
  - use the installer's built-in scripting capability
  - use an industrial strength UI test driver

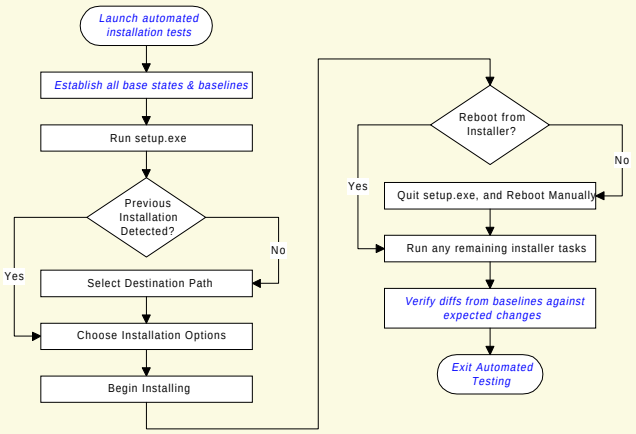
Designing automated tests via flow diagrams

- ✔ Begin with a diagram for the installer
- ✔ Superimpose an automation wrapper
- ✔ Add any intervention tests needed
- ✔ Include as much detail as possible
- ✔ Review the design, then begin implementation

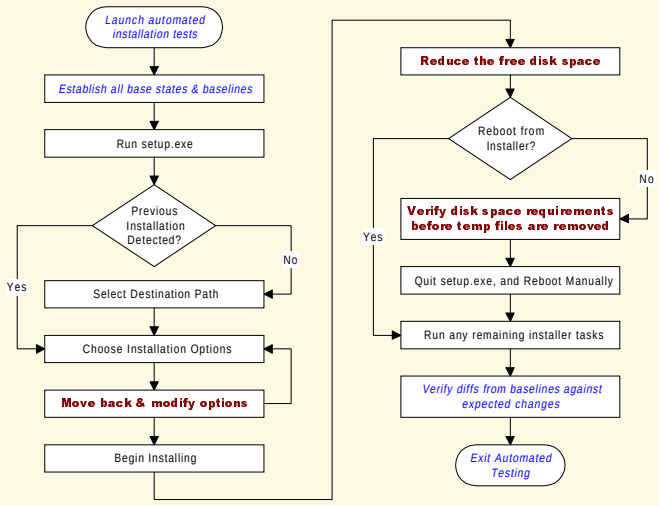
Sample installer flow diagram



Installer diagram with automated test wrapper



Installer diagram with automation wrapper and intervention tests



The master/slave configuration

- ✔ Different types of distributed testing
- ✔ Pros
  - makes full use of available machines
  - facilitates the reboot process
- ✔ Cons
  - heavy dependency on the master machine
  - if the master gets stuck, the slaves cease

Verifying the results

- ✔ System diffs, before and after installing
  - verify changes to the system registry
  - verify file changes against disk spec
  - use the disk spec as your Oracle
- ✔ If you're including UI testing
  - backward navigation
  - canceling installations
- ✔ Does the program start up afterwards?

Computing disk space requirements

- ✔ Sources of error
  - file systems, updated files, hard coded values
- ✔ Computation procedure
  - record purported disk space requirements
  - capture baseline free disk space
  - for each flavor of installation, note disk space at the end
  - quit installer, restart, and record free disk space
  - subtract baselines from each measure

Testing the uninstaller

- ✔ Uninstaller as a big Undo operation
- ✔ Automation tips
- ✔ What files do not get uninstalled?
- ✔ Should you reboot?
- ✔ Shared components, and the refcount
- ✔ Generic testing libraries, for automatically verifying uninstallations

Conclusions and future directions

- ✔ Automated installation testing is feasible, especially with a good design
- ✔ Our implementation is only the tip of the iceberg
- ✔ Future directions
  - network installations
  - installations over the web
  - installer compatibility testing

References

- "Install Risk Catalog", personal communication from James Bach, 1999
- "Specific Installer Issues", personal communication from Noel Nyman, 1999
- "Installer Testing", by Pete Schneider:
  - http://www.data-dimensions.com/newsletters/testnet/docs/installertest.htm

Acknowledgements

- Special thanks to the people who reviewed this paper, and provided ideas:
  - Noel Nyman
  - Bret Pettichord
  - Johanna Rothman
- Also, thanks to people at Autodesk who supported this project
  - Michael Lapinskas
  - Gary McEwan

Automated Software Installation Testing

Christopher Agruss & Alexei Tcherkassov
Autodesk Inc.

International Quality Week, 1999

"Little bugs have littler bugs,
upon their backs to bite 'em,
and the littler bugs have littler bugs,
and so ad infinitum."
[verse from "There ain't no bugs on me," trad. children's song]

Abstract

Testing software installation is a relatively constrained and repetitive task that lends itself well to automated methods. One of the prime advantages in doing this is that the installer tests can be run in parallel against an array of different machine configurations, minimizing the repetitive tedium of manual testing. A well-crafted automated system can verify many aspects of an installation, including: the accuracy of the purported disk space requirements, confirming that the expected files are written to disk, and checking that the program can be launched successfully afterwards. With a modular design, many of these same routines can be used to verify uninstallation of the software as well. After describing our experience with creating such a system, we'll also discuss some readily available freeware, shareware, and commercial tools that we found helpful for automating installation testing.

Introduction

A user's first encounter with a new software program is typically with its installer. Consequently, your installer needs to be of the highest possible quality, in order to be confident about the customer's initial experience with your software. One way to verify the quality of your installer is to design a smart balance of manual and automated tests that can be run across a range of different machine configurations. In theory, this improves the likelihood that important defects will be found and fixed before the installer arrives in your customer's hands.

One of the authors was managing a release testing effort some years ago, and toward the end of the last day of testing he asked someone to run one more installation, to make sure that it worked on a particular machine configuration. The person's response was, "Alright, I'll run this last one, but if you ask me to run any more installation tests, I quit!"

One bothersome attribute of installer testing is that it can take an awfully long time to run a single testcase, especially for larger software packages. Multiply this by the number of machine configurations that you need to cover, and the time required for testing becomes daunting. This is one of the primary reasons that automated installer testing is so appealing.

The goal of this paper is not to provide a comprehensive approach to installation testing, but rather to identify which aspects of it are best suited to automated methods. Still, it would be silly to build the automation list without having a clear idea of the full extent of the testing required. Some friends (James Bach and Noel Nyman) generously shared with us their installer testing notes, which I've reproduced in Appendixes C & D. Also, for an introduction to installation testing in general, please see Pete Schneider's online article, listed in the References section at the end of this paper.

Our paper is based largely on recent experience with automating installation tests for Autodesk's programs, and describes some of the thought process we went through in designing our tests. As such, the examples we'll use are from doing installations on Windows platforms. The topics we'll cover include: establishing machine base states; designing the automated system via flow diagrams; selecting the appropriate tools; implementing distributed tests; and verifying the results. Briefly, we'll also cover some issues with automated testing of the complementary operation, uninstallation, but this will receive relatively little attention here. Automated testing of uninstallation is closely tied with the concepts in this paper, but there are enough issues unique to uninstallation to warrant a separate paper on the subject.

Installer testing as meta-automation

In a manner of speaking, a software installer is already an automated testing routine in its own right. Most installers follow a script, which determines how it writes files to disk, configures your system settings, and makes any other changes necessary so that you can run the software being installed. On top of this, many installers do some type of checking to verify that the machine being used is compatible with the new software, that it has enough free disk space, and the like.

In this light, automated installer testing takes the form of meta-automation. For example, the installer checks whether you have sufficient disk space to install successfully, but your automated routines (meta-automation) might test whether the installer is checking for the correct amount of disk space. If the installer is checking whether you have at least 1,000,000 bytes of free space, but you really need 1,000,001 bytes, an automated test could flag this as a defect.

When designing this meta-automation, its vital that you understand fully the assumptions of the installer's own script. For instance, the installer's disk space checking logic might intentionally be assuming that the file format for the disk being used is FAT; if so, then by design it will overestimate the amount of free space needed for an NTFS or FAT32 disk. Knowing this, you could then design your installer tests to only flag underestimation errors, and to ignore overestimation errors, within reasonable limits.

Another aspect of installer testing as meta-automation is that your suite of tests can span a series of related tasks, much broader than simply running the installer. For example, your test suite might include setting a machine base state, running an installation, rebooting, launching the program, and then running the uninstaller. Along the way, it will probably be verifying a variety of critical items too. Before diving into further examples, let's first take a look at some ideas for setting up machines in your installation testing lab.

Establishing machine base states

Establishing a baseline machine state is a fundamental first step for any installation testing. Disk imaging programs such as Ghost and IC3 (please see Appendix B for specific tool references) are widely used now to return a machine's disk to a base state. You'll probably need an array of machine base state images, depending on how many configurations you need to test. One way to approach this is to have one image for every combination of operating system and disk file format that you plan to test.

If you've never done this before, you could start by reformatting a machine's disk drive to the most commonly used file format, and follow that by freshly installing the most commonly used version of the operating system. Create the image for that base state, and then move on to create images for the other file formats and operating system versions that you need to test.

Ideally, you should assemble some dedicated lab machines for installation testing, so that they can be set to the base state when needed, without interrupting other activities. Imaging and restoring an array of machines is commonly automated, and will reap you time savings, regardless of whether you'll be doing manual or automated installer testing.

At some point, you'll need to answer the question of whether to restore your baseline images in between each installation testcase, or whether it is valid to do it less often? If your uninstall program is reasonably free of defects, then one alternative is to only reset to the full base state when receiving a new build. If this is the case, you should be able to rely on the uninstallation program to restore your base state; if not, then you have probably found an important defect in your uninstaller.

To answer that question more fully, you could make this your first testcase: does running the installer followed by running the uninstaller truly return the machine to the base state? If that test is successful, we would suggest an expanded follow-up test: does running the installer, followed by launching the program to run some tests, followed by running the uninstaller also return your machine to the base state? The answers to these questions will help you determine how often you need to restore the machine base state from the image file.

Selecting a testing tool to drive the installer

The next decision you're faced with is how to drive your automated installer tests. One option is to use the installer's own scripting capability, if it has one built in. For example, InstallShield provides a command line switch for recording a given installation, allowing subsequent playback in "silent" mode. For details on this method, please refer to Appendix A. One advantage of using the silent install is that it doesn't require purchasing a test tool to drive your installer's UI; in so doing it will also save you the expense of writing test code to drive that UI. Another advantage is that the tests may run faster than if you drive them through a UI testing tool.

On the other hand, there are many advantages to using an industrial strength UI testing tool, such as SilkTest (www.segue.com) or VisualTest (www.rational.com). To begin with, a good tool will provide a recovery system in case you wind up in an error condition. Moreover, these tools already have extensive testing libraries, helping you to verify the results in an automated manner.

Of course, if you want your automated installation tests to include some cases that cover your installer's user interface, then you'll need to use a UI tool. For example, if you wish to automate the testing of navigating backwards in the Installer, or of canceling at various stages, then you'll need a UI tool.

Designing the automated tests via flow diagrams

To design the flow of the tests, we recommend beginning with a simple flow diagram of your installer's behavior. The behavior of most installers is constrained enough to allow them to be diagrammed without difficulty. Please consider the simplified example of a flow diagram for a generic installer, shown below in Figure 1.

This initial flow diagram then becomes the template on top of which you can superimpose the flow of your automated tests. In doing this, be sure to indicate on the diagram the points where you will verify the test results. For example, Figure 2 expands on the previous diagram, by placing the installer within an automated test wrapper (shown in blue italics). Notice that in Figure 2, all the test-related items occur before the installer is launched, and after it has completed; if this is acceptable for your testing purposes, then it might suffice to use the installer's own scripting capability to drive it during your automated routines.

The final diagram, Figure 3, is based on the previous ones, but has some testing-related items (shown in bold magenta) added in the midst of the installation procedure. To drive your installer through these types of intervention tests, you'll need a UI testing tool. In this example, one of the checkpoints we've included is for disk space calculations at the end of the installation process, *before* quitting the installer, in order to get a direct reading about how much space was required for the temporary files used.

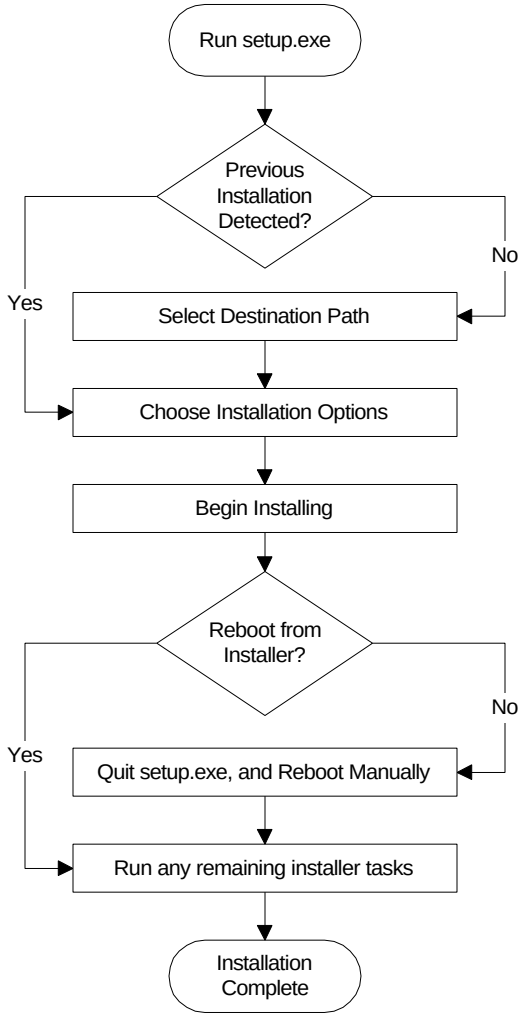


Figure 1: Flow diagram for the Installer alone

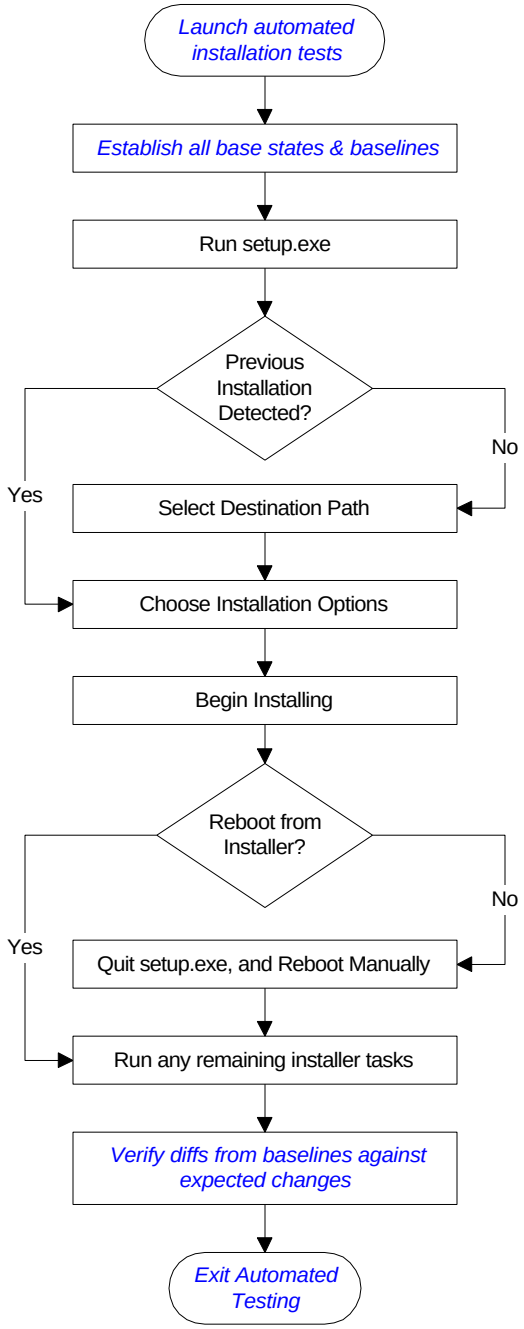


Figure 2: Flow diagram for the Installer, with an automated test wrapper, indicated by blue italics

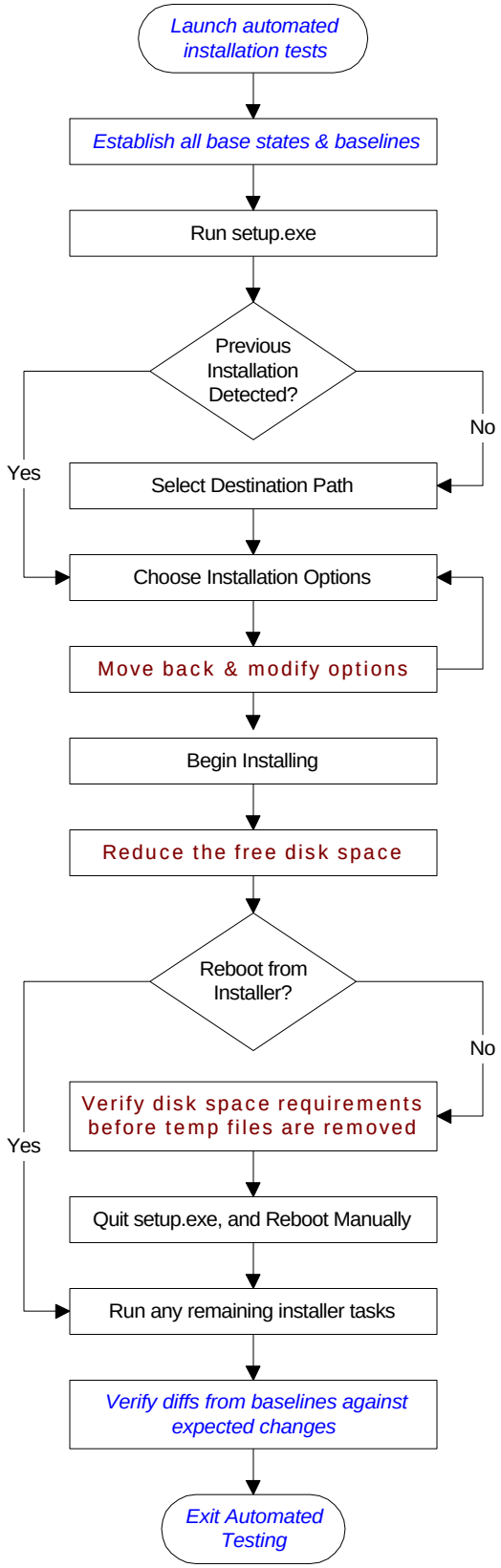


Figure 3: Flow diagram for the Installer, with test wrapper, plus "intervention" tests in bold Magenta

As mentioned earlier, these examples of flow diagrams above are vastly oversimplified. When expanding them into more realistic versions, here are some things to consider adding. If this is not the first release of the software being installed, be sure to include different logical paths, depending on whether a previous version of the program is installed. Similarly, be sure to include paths for re-installing the same version on top of itself, and for installing a full version on top of a compact version.

Before actually translating your flow diagram into test routines, check that your design is flexible. For example, the same routine you use to verify that the correct files are written to disk during an installation can also be used to verify that those same files are removed after running the uninstaller. The same goes for changes to the registry. A little planning like this can save you major rewrites later on.

When your diagrams are complete, they should make the task of automating the process more straightforward. If the diagrams contain sufficient detail, we're often able to translate them directly into automated testing code.

Some advantages of a Master/Slave configuration

We recommend using a master/slave configuration, in which one master machine drives the tests on multiple slave machines. This is one example of a distributed testing environment, though there are many other ways to distribute tests. Some test driving tools include master/slave functionality via an agent running on the slave machine.

This method affords a number of benefits: it allows you to trigger installations simultaneously on as many machines as you have available; it consolidates all the result logging easily into a single file; and, it makes rebooting easier. With respect to the ease of rebooting, the master/slave approach allows for simplified flow control after the reboot has completed. To accomplish the same thing using a single machine, it becomes a challenge picking up the flow of events after a reboot; it's possible to do, but is more complicated. In contrast, with a master machine controlling the slave, the master simply needs to wait until the slave finishes rebooting, and then continues with the next routine in the test suite.

For some tips on how to restart a machine using the 4test automation environment, please see Mr. Cluey's Kludge Page: www.ultranet.com/%7Edanil/Cluey/shutdown.html

Tangentially, one aspect of automating the rebooting process to consider is how you will log the machine back onto the network after a reboot. One easy solution for this on Windows machines is to use Microsoft's free program TweakUI, which makes it easy to manipulate the appropriate registry settings, so that your machine can log itself onto the network automatically. Another alternative is to use a UI test driver to watch for the login dialog, and then to feed it a username and password. One drawback to the latter method is that the password becomes available to anyone who views your test code.

Verifying the results of an installation

Clearly, one of the fundamental things you'll want to verify is that the installer writes the expected files to disk. The number of files written typically grows with the size and complexity of the program being installed. Further, the specific files required often change during the project's life cycle, creating the potential for lots of maintenance work on your verification routines.

One way to minimize this problem is to create data driven tests, which read the list of files to be verified from an external table. This has two advantages. Firstly, it's often easier to maintain a table, using a spreadsheet program, than to edit hard coded lists of files inside a test routine. Secondly, you might be able to get the developers of your installer program to maintain the list for you. Installer programs for major software companies are commonly written based on something called the disk specification (or disk spec), which lists all the files that are supposed to be installed. A more sophisticated approach would be to imbue your verification routines with enough intelligence to parse the disk spec, and to determine on that basis which files to verify on disk.

You could begin by simply verifying that the expected files exist in the correct location after installing, and that they don't exist after uninstalling. Following on that, you may want to develop routines to verify deeper levels of complexity, such as: is the file the correct version; can a newer version be installed over an older one; is an older version prevented from replacing a newer one; and so on.

If you're testing on Windows, you'll probably need to verify changes to the system registry after installing; this can be automated without much difficulty. Many commercial testing tools allow you to test for specific expected changes to the registry. There are also some free tools (described in Appendix D) that will report these changes.

Another test you should consider automating is checking how well the installer behaves if you cancel an installation in the middle. You could design your automated routines to include one cancel operation at every step of the way. The expected behavior is usually that everything the installer has done up to that point is completely undone.

Computing disk space requirements

Closely associated with the disk spec is the related issue of how much disk space is required for a given installation. Fortunately, this is one aspect of installation testing that can be automated handily. Some test tools provide a library function that will report the current free disk space, which makes these computations especially easy.

Ordinarily the installer will check that there is sufficient free disk space, and then will either prevent the installation if there isn't enough space, or else will warn you first that you don't have enough. The amount of space required is typically the sum of the permanent disk space needed, plus short-term storage space for temporary files that are used during the installation. The total disk space needed is often hard coded into the installer by the developer, and even if it is accurate to start with, it is subject to error as soon as the disk spec changes. Another source of inaccuracy is the file system in use. For example, what's sufficient space for NTFS or FAT32 may not be enough for a disk using FAT.

When automating tests for disk space requirements, we use the test tool's ability to capture the free disk space at strategic points in time. Here's the basic scheme we've implemented:

- ✓ Record the amount of the space the installer claims that it needs for the various types of installation
- ✓ Capture the baseline free space before launching the installer
- ✓ Run each type of installation (beginning with the Typical default option), recording the free disk space after installing, but while the installer is still running
- ✓ Quit the installer (don't choose a restart option), and record disk space after that, which will reflect temp files that were deleted on quitting
- ✓ Restart, and allow any final setup to occur during the boot process, then record the final amount of free disk space
- ✓ Subtract the baseline from each of the subsequent measures, to see how much space was actually required at each step of the way.

The method above will give you direct information about how accurate your installer's disk space requirements are. As a follow-up to the procedure above, we sometimes use the tool named Stress (described in Appendix D) to create low disk space conditions. It can create files of any size you choose, to consume disk space, thus helping to test your installer's error handling when it finds itself running low on available space.

Testing the Uninstaller

A fundamental uninstallation issue is that some popular installers (such as InstallShield) will only uninstall the last operation run, even if the last operation was a small addition to a large existing installation. In this sense, the uninstaller resembles an undo function. Knowing this, its relatively

straightforward to automate tests for the uninstaller. One approach is to simply follow each installation test with its equal and opposite uninstallation.

However, before uninstalling a given installation, you'll probably want to launch the installed program at least once, to confirm that it was basically installed successfully. One side effect of this is that launching a program will often introduce some configuration files that will not be removed by a subsequent uninstallation. For these cases, you could write a garbage collection routine to delete the user files prior to the next automated installation.

As with many installations, it's usually safest to reboot after running the uninstaller. Some uninstallers prompt with a suggested reboot at the end, and others don't. Our experience is that uninstalling DLLs, without an immediate reboot, may lead to system instability. As testers, we're obliged to check the behavior with and without a reboot, but for the purposes of automated testing, we recommend a routine reboot after uninstalling.

Conclusions and Future Directions

To summarize, we've found it worthwhile to automate some aspects of installation testing. By using the techniques described above, we've been able to reduce the amount of manual drudgery normally involved with testing installers. We've also been able to get more rigorous results from these tests, by paying special attention to the automated verification routines.

As we worked through some of the issues presented in this paper, we realized that we'd only scratched the surface, with respect to the many other dimensions of installer testing that would be useful to automate.

Some future directions include:

- Network installations
- Installations over the web
- Generic installer test drivers
- Installation and uninstallation of 3 rd party software

To give one example of the latter topic, 3rd party software, consider the impact of installing shared DLLs on Windows platforms. As you may have noticed if you've run many uninstallations for different programs, you will sometimes receive an alert to confirm that you want to uninstall a shared DLL.

In a nutshell, a counter (refcount) is incremented in the system registry when you install a program that installs a shared DLL. If you later uninstall that program, then the counter is decremented, but the DLL won't be uninstalled if the counter indicates that some other program that uses the DLL is still installed. However, if the program you're uninstalling is the last one using the DLL, then you'll get the alert message asking whether you really want to remove the DLL.

This is a more complex aspect of installation testing, because it requires you to install two or more programs that share a component (or else to artificially manipulate the refcount in the registry), and then to observe the behavior during uninstallation. We have not tried to automate the testing of this area, but we don't think it would be terribly difficult. We also believe it would be worthwhile, given the apparent growth in the use of shared components.

In closing, we wish you success with your installer testing, and hope that you find something analogous to Newton's third law to be true for your products: for every installation, there is an equal and opposite uninstallation.

Acknowledgements

The authors wish to thank all those who contributed information for this paper, especially Noel Nyman, James Bach, and Johanna Rothman. Additional thanks to all the other good folks from the Los Altos Workshop on Software Testing (LAWST), including Brian Lawrence, Cem Kaner, Bret Pettichord, Melora Svoboda, and Brian Marick, for the stimulating conversations from which some of these ideas emerged.

References

"Install Risk Catalog", personal communication from James Bach, 1999
"Specific Installer Issues", personal communication from Noel Nyman, 1999
"When Should a Test Be Automated?", by Brian Marick, Quality Week, May 1998
"Installer Testing", by Pete Schneider:
http://www.data-dimensions.com/newsletters/testnet/docs/installertest.htm

Appendixes

Contents

Appendix A: Instructions for using InstallShield's silent install
Appendix B: Some tools that help with installer testing
Appendix C: Installer risk checklist, by James Bach
Appendix D: Specific installer issues, by Noel Nyman

Appendix A: Instructions for using InstallShield's silent install

From InstallShield's (www.installshield.com) documentation:

InstallShield Silent - InstallShield Silent and silent installations are supported only on 32-bit platforms.

InstallShield Silent allows automated electronic software distribution, also known as silent installation. With InstallShield Silent, there is no need for a user to monitor the installation and provide input via dialog boxes. An InstallShield Silent installation runs on its own, without human intervention. You must launch InstallShield Silent with the SETUP.EXE -s command line parameter. To comply with Windows 95 logo requirements, a silent installation must create a response file in which the default installation options are selected.

You can run your setup with the SETUP.EXE -r parameter to select installation options and automatically record the InstallShield Silent Response File, or you can create your own from scratch. To view a real world example of a response file, refer to the SETUP.ISS file located on InstallShield3 distribution disk one. In addition, a description of the response file format is provided in the Create the Response File topic.

Steps to Create a Silent Installation

Follow the steps below to create a silent installation :

1) Create the Setup Program
2) Create the Response File
3) Play Back the Silent Installation

Appendix B: Some tools that help with installer testing

Tools that track changes during installations

Because so much of an installer's work is writing files to disk, it's invaluable to have a tool that will take snapshots of your disk before and after an install. More to the point, this tool should also be capable of reporting the differences before and after the install. There are a number of such tools that we've come across, but the good news is that many of them are free. The best of these will also report changes to the system registry.

InCtrl 3 lets you track system changes made by Windows installation programs. One of the authors used this tool last year, and likes it. It tracks file additions and deletions, changes made to ini files and changes to the Registry. InCtrl 3 handles long filenames. This utility is easy to use, but reporting could be more customizable. A free demo is available for download from http://www.zdnet.com/swlib/

Analyzer, from Veritest, comes recommended from several sources. Veritest is the independent test lab for Windows Logo compliance. Using this tool may also help testers/vendors/customers who have no interest in the Logo, but do need to test an installer. Download for free from:
http://www.veritest.com/mslogos/nt98/analyzer.htm

Disk imaging tools

InfoWorld compared the top commercial disk-imaging packages toward the end of 1998, and you can read their review, together with the ratings, on the following page:
http://www.infoworld.com/cgi-bin/displayTC.pl?/981116comp.htm

Their top two picks among the competition were:
ImageCast IC3 - http://www.innovativesoftware.com/imagecast/home.htm
FlashClone - http://www.suredata.com

Other miscellaneous tools that help with installer testing

The Stress utility (STRESS.EXE) lets you allocate system resources to see how your application behaves under low-resource conditions. You can allocate resources such as global memory, User and GDI heap memory, file handles, and disk space. This handy tool is produced by Microsoft and can be found on every Visual C++ disc.

TweakUI is a Windows control panel, which has a component that allows you to automatically log a machine onto the network after a reboot. TweakUI is a component of Power Toys from Microsoft Press, available for free from:
http://mspress.microsoft.com/reslink/nt40/toolbox/tools/tweakui.htm

SilkTest is the UI test driver we use. It has a range of built-in libraries we use for installation testing, including capturing the amount of free disk space, checking specific registry settings, and reporting on whether specific files are located where you expect them to be. For more information, please see Segue's site:
http://www.segue.com

For additional testing tools of all sorts, please see Brian Marick's Corner:
http://www.stlabs.com/marick/faqs/tools.htm

Appendix C: Install Risk Catalog

Courtesy of James Bach

- Capability
  - Installer lacks modern, expected features
    - no uninstall
    - no custom install
    - no partial install ("add")
    - no upgrade install
  - Wrong files installed
    - temporary files not cleaned up
    - old files not cleaned up after upgrade
    - unneeded file installed
    - needed file not installed
    - correct file installed in the wrong place
  - Wrong INI settings/registry settings
  - Wrong autoexec/config settings
  - Files clobbered
    - older file replaces newer file
    - user data file clobbered during upgrade
- Reliability
  - Intermittent failure
  - Fault tolerance/recoverability
    - Can't back up
    - Can't abort
    - No clean up after abort
    - Mishandled read error
    - Mishandled disk full error
- Compatibility
  - Installer does not function on certain platforms
  - Other apps clobbered
  - HW not properly configured
  - HW clobbered for other apps
  - HW not set for installed app
  - Screen saver disrupts install
  - No detection of incompatible apps
    - apps currently executing
    - apps currently installed
- Efficiency
  - Excessive temporary storage required by install process
- Usability
  - Installer silently replaces or modifies critical files or parameters
  - Install process is too slow
  - Install process requires constant user monitoring
  - Install process is confusing
  - UI is unorthodox
  - UI is easily misused
  - Messages and instructions are confusing
  - Mistakes during install process readily cause loss of effort

Appendix D: Specific installer issues

Courtesy of Noel Nyman

Notes on issues I'd test for, based on bugs I've seen

There are several Windows app installer programs around. InstallShield (http://www.installshield.com/) and Wise (http://www.glbs.com/) may be the best known, but there are others, too. Those vendors have already worked out a lot of the bugs and issues that you'll have to deal with if you roll-your-own installer. They also present a familiar interface to the user, if you use one of the popular programs. Consider the licensing fees well spent.

To use commercial installers, you create a script (source code) that supplies parameters to drive the installer. Then it "compiles" the final .exe and support items you need, and probably creates the archive(s) as well. You can make even an excellent installer look bad by creating funky scripts. Treat the installer scripts with the same tester skepticism that you'd use for any other source your developers create. If you find that developers have bypassed main features of the installer or replaced some with their own "better" items, probe *very* deeply about the benefits and possible problems. If a feature is so popular or necessary that this vendor spent its own development/test time to create it, is it really something that should be quickly discarded or ignored?

Use the most current version of the installer. If that vendor has done their job well, they've included items that will work seamlessly with the current and *next* version(s) of Windows. Take advantage of their efforts. The older version you may still have won't be as smart.

A good installer fails elegantly in environments where it isn't intended to be run. For 32-bit Windows that usually means that the installer starts out life as a 16-bit app, just so it can use its own message to tell the user that they aren't going to successfully install on Windows 3.x. There are several implications of that. The most significant is that the installer's 16-bit .exe will expand/create a 32-bit version to do the "real" work, then transfer control to that 32-bit installer.

The 32-bit app stuff is usually created/copied/expanded into a temporary folder. Most systems have a folder named TEMP that Windows knows about and can point to. Good installers ask Windows what this folder is named, where it is, and use that info to store and run their 32-bit incarnations. Several tester issues come to mind:

- ✓ Does Windows really have a vector to a temp folder? It usually does, but clever or ignorant-but-experimental users can manage to remove that information from the OS.
- ✓ Is the name the "expected" TEMP or TMP or something else?
- ✓ Long file names may fool some older versions of even commercial installation programs.
- ✓ Is there a folder named TEMP that's *NOT* the "temp" folder? If so, the installer may use it, regardless of what Windows says.
- ✓ Is the temp folder known to Windows missing? Perhaps the user has removed it as part of a search-and-destroy mission to recover more hard drive space. The installer should handle this gracefully, even if that means failing with an *appropriate* message.
- ✓ Is the temp folder on a drive other than C:? That's legal and cool. Windows will tell you both drive and folder name, but the installer may use only the folder name and ignore the drive designation.
- ✓ Does the user have adequate permissions to the temp folder? If they don't, there's not much you can do about it, but failing elegantly and documenting the requirements in a ReadMe would be polite.
- ✓ Even more polite, don't put the ReadMe in the archive that the installer expands. A ReadMe about installer issues is useless if the user has to install the product to read the file.

If you need to attach a debugger to the installation process, you'll need to do it after the 16-bit part passes control to the 32-bit part. I've seen "bug free" installers fail when the user's 16-bit process has been

munged by something that may have happened weeks before. Usually the installer just "fails to start." The 16-bit VDM (Virtual DOS Machine) is damaged and nothing will run in it, but the user doesn't realize that since most apps these days are 32-bit and don't run in the VDM. A simple solution is to reboot. In NT you can also use TaskMan to kill all the ntvdm processes. It might be a good idea to replicate this problem during testing to see if your installer aborts after it leaves bits of itself in the system.

Make sure the installer cleans up after itself. All it's files and folders in the temp folder should be gone when the installer closes. If they aren't, assume it's your bug first. One irate vendor reported an NT4 bug because their installer could not delete its temporary folder on NT4. The same installer worked as expected on Win95. Turned out the installer was not closing its handle to the folder after deleting all the files. NT is quite righteous about not letting a process delete resources when other processes have handles open to them. You could say the "bug" was in Win95 for deleting the folder while the handle was open, although that level of security was beyond its design parameters.

More general items about installers once they get going:

- ✓ Test all the "Back" options that let users back up and change options. It's easy to create installer scripts that mess up the backtrace paths or ignore the users' changes.
- ✓ Many apps need to register DLLs with Windows and the use RegSvr32.exe to do that. *Most* Windows installations have this .exe in the system path. But a few will not. Your installer should either be prepared to use its own copy of the .exe or tell the user *explicitly* why it's failing and what they need to do to fix the problem.
- ✓ Test on "virgin" machines. There are many reasons, but one sometimes overlooked is that you've probably been testing early versions of the app before a real installer was available. To do that you manually put things on the system or hacked the registry. Your uninstaller may not remove all that stuff. With it already in place, you won't find the bugs when your installer fails to install the same stuff. Programs like Ghost and Disk Image can archive images of "virgin" systems and "install" them more quickly than actually reinstalling Windows and other apps every time. You can also use them to recreate several OSs, versions and file systems on the same machine.
- ✓ Antivirus programs can interfere with your installer in ways that may not be obvious or detectable by the installer. So, it may report success when your app was not actually or completely installed. There are probably too many AV programs to be able to detect them all, but warnings to users are good.
- ✓ Use the Windows file versioning system on your own app's files and to determine if anything you intend to replace is really older than your current versions. Using time-date stamps, files sizes, or even CRCs is unreliable.
- ✓ Refcount what you should, don't refcount what you shouldn't.
- ✓ Ask if it's OK to change existing associations to your app's file extensions. Don't just assume the user wants your app to take over all .xyz files. Check even unrelated extensions to be sure their associations are still correct after your app is installed. Those tricky developers may have accidentally added an extension your app doesn't use to the associations list in the install script.
- ✓ If you have special support for other apps, perhaps Microsoft Word and WordPerfect, don't install that support or natter at the user about not having Word if they don't have it installed. Make sure you can accurately identify the presence/absence of those apps.
- ✓ Make sure you get the right number of program groups on all OSs, they're in the expected place(s) and the right things are in them.
- ✓ Make sure the user can't deselect incompatible options in your installer. If you have optional parts X, Y and Z, but part Y needs part X, the user should not be able to uncheck X and check Y.
- ✓ Try installing on a system that has an earlier version of the development environment your developers are using now. If your app is compiled with RealCoolC++ V7.3, install RealCoolC++ V6.x on your test machine, then install your app. The installer may find older libraries, misinterpret the versions, and not replace them with newer items, breaking some of your app's functions.
- ✓ Try installing with various permissions, unsuccessful logins, etc. Remember that in some environments users may have limited "personal" disk space and your installer program may not have enough room to expand itself *and* install your app. Users who have their own folders or drives may be

able to install your app but not fonts or other system resources because they're restricted from those parts of the machine.

- ✓ Not all machines have floppy drives. Not all machines have hard drives. Not all "hard" drives are fixed. Some disk and CD-ROM drives do not have letters assigned to them. Some drive letters are assigned to network resources. Different users on a given machine may have their "personal" space on any combination of those drives. Your installer should do the right thing...succeed or fail with an appropriate message...in all cases.
- ✓ Consider supporting the Zero Administration Workstation (ZAW) program so enterprises who are using it have an incentive to buy your product over the competition. You may need a separate installer to work in the ZAW environment.
- ✓ If your suite is made up of several stand alone apps A, B and C, make sure the user can install any combination (perhaps A and C) then install other parts (B) successfully later.
- ✓ Try reinstalling over an existing installation of your app.

Have an uninstaller and test it:

- ✓ Rerun your "win diff" procedure after uninstall and make sure you remove everything. There are some design decisions here. Should you remove all your private registry items or keep it to retain preferences in case the user reinstalls? Should you unregister your CLSIDs? Probably, but what if they're used by other apps from you or companion vendors?
- ✓ Did you add plug-ins or other items to other apps (Word, browser)? How do you remove them? Should you? If you do add plug-ins, try updating that app (Word, browser) to a (newer) version that doesn't have your plug-in, then uninstall. Do you trash that app trying to remove the no-longer-existing plug in?

End of Appendixes for Automated Installation Testing

A Web-based System Testing Repository Model

Fan Yang , Trung Nguyen & Anant Adiga
Sequent Computer Systems, Inc.

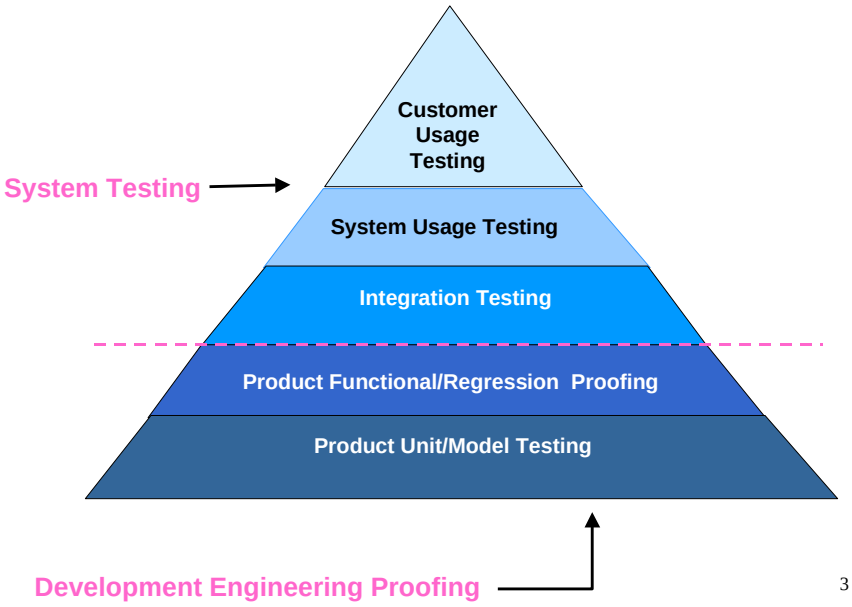
1

Overview

- Systems Test Model
- Activities and Challenges
- Requirement Analysis
- Environment and Tools
- Phases and Modules
- Process Flow
- Implementation
- Summary

2

System Testing Pyramid



3

Systems Test Model

- System Testing consists of
  - Integration Testing: configuration, compatibility and installation/de-installation
  - System Usage Testing: volume/load, stress, availability, recovery, stability and maximum/minimum
  - Customer Usage Testing: simulation of customer operation profiles

4

Activities and Challenges

- System Testing activities include
  - Test Plan Generation
  - Resource Allocation
  - Test Case Execution
- Challenges
  - Knowledge transfer
  - Automation of test plan generation
  - Test execution
  - Efficiency of resource allocation

5

Requirement Analysis

- Automation: system test plan generation, test case manipulation, requirement tracing and test case execution
- Query: for various information and for different views
- Lab resource management: equipment flow and inventory, hardware configurations
- Web-based application

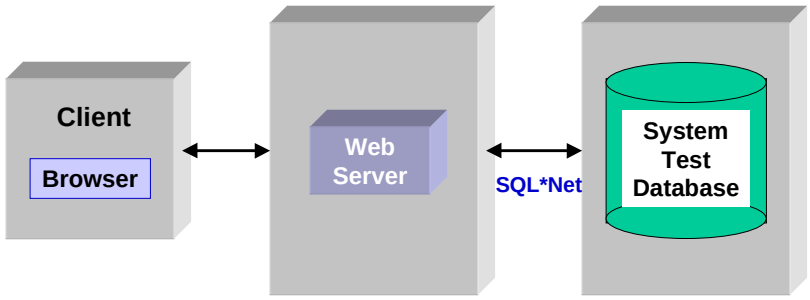
6

Environment and Tools

- Existing Environment: disparate data sets from multiple applications
- Limitations of tools in the market:
  - Portability
  - Customization levels
  - Coverage of Systems Test requirements
  - Costs
- Development environment and tools: web-based three-tier client/server model

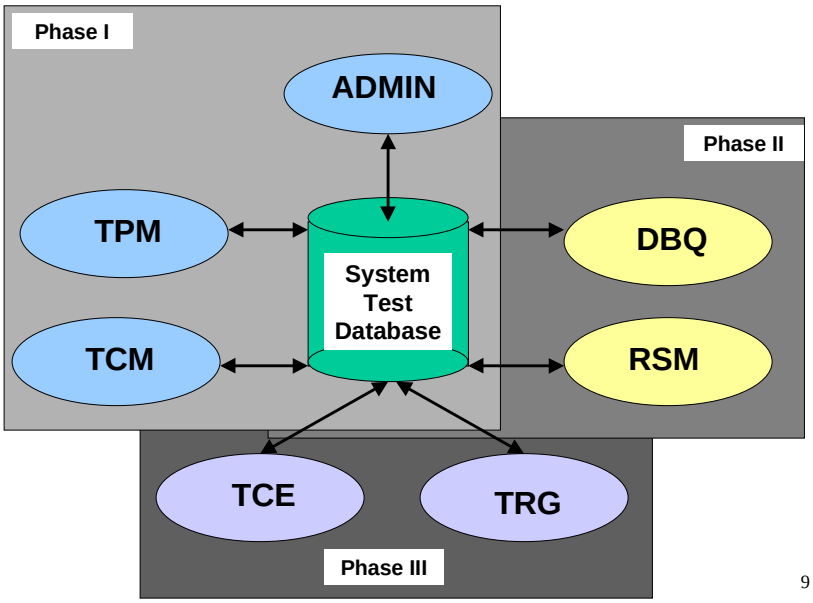
7

Three-Tier Client/Server Model



8

Project Phases and Modules



9

Phases and Modules

- Phase I:
  - Test Plan Manager (TPM): ***a module that generates system test plan automatically.***
  - Test Case Manager (TCM): ***a module that generates generic test cases.***
  - Administration (ADMIN): ***a module that provides administration tools for System.***

10

Phases and Modules (Con.)

- Phase II:
  - Resource Manager (RSM): ***a module that control resource flow and allocation***
  - Database Query (DBQ): ***a module that provides varies query mechanism***
- Phase III:
  - Test Case Execution (TCE): ***a module that integrates test cases, test hardness and lab.***
  - Test Report Generator (TRG): ***a module that generates test report prototype after each release.***

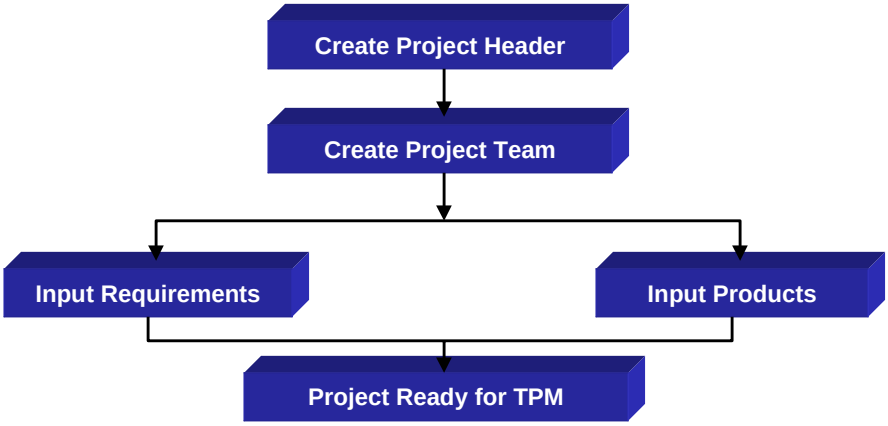
11

Process Flow

- Project creation process - in ADMIN
- Test plan generation Process - in TPM
- Generic test case manipulation process - in TCM
- Resource allocation process - in RSM
- Query interface - in DBQ
- Test case execution model - in TCE
- Test report generator - in TRG

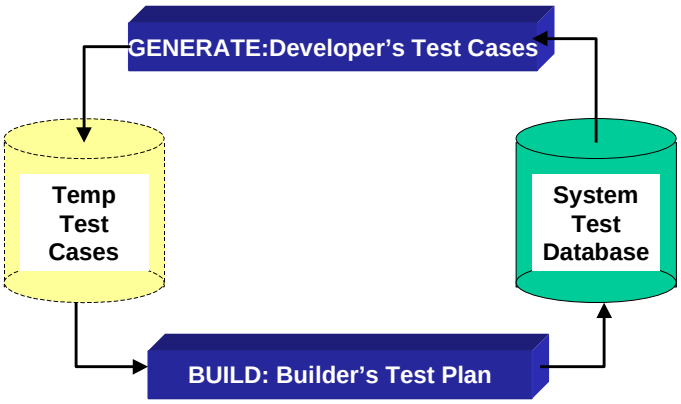
12

Project Creation Process



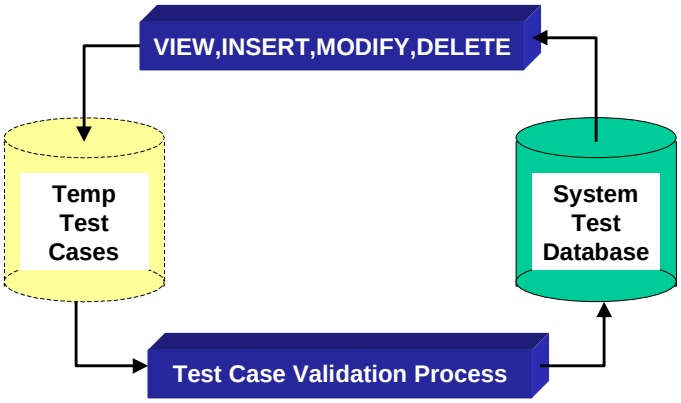
13

Test Plan Generation Process



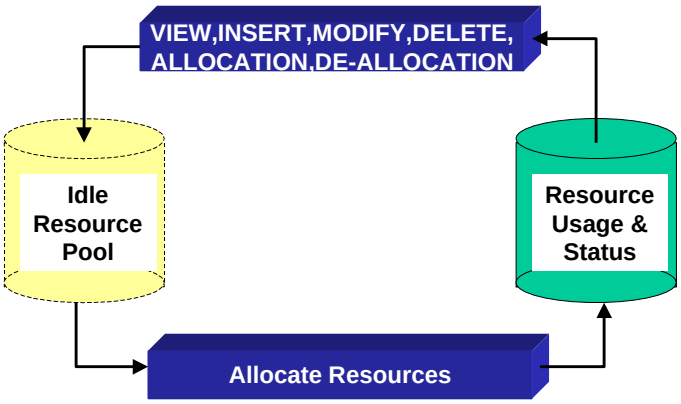
14

Generic Test Case Manipulate Process

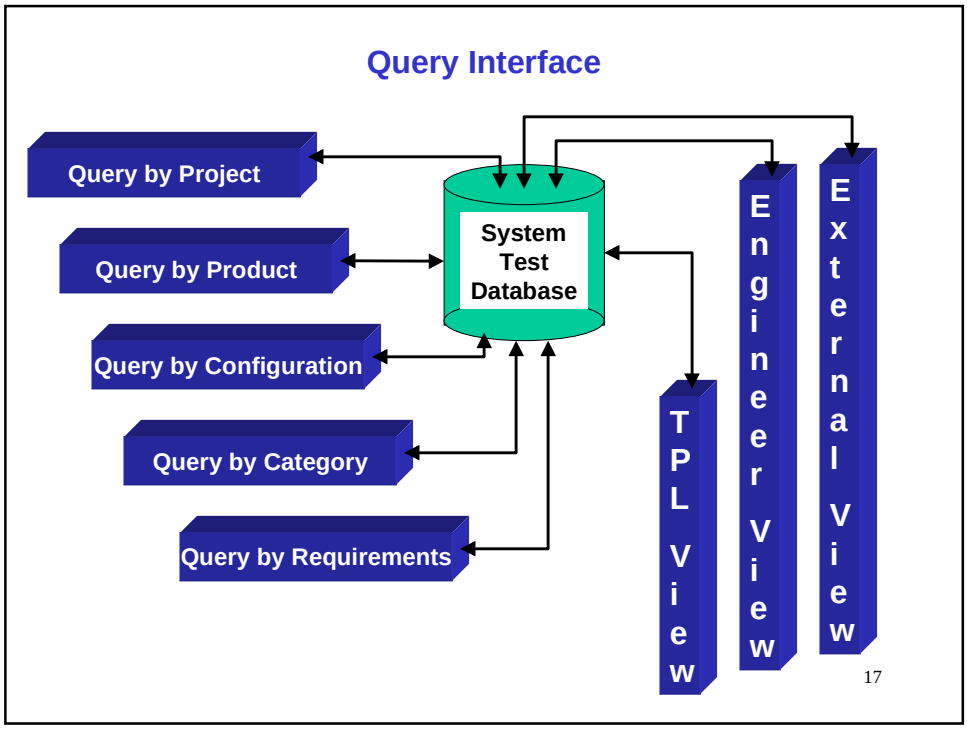


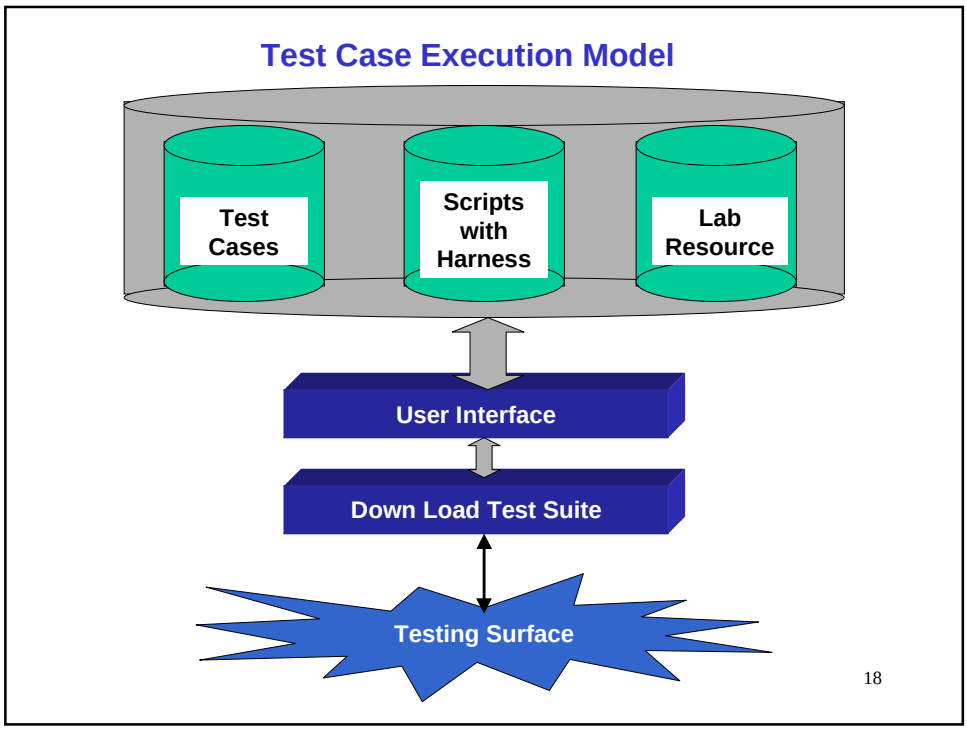
15

Resource Allocation Process

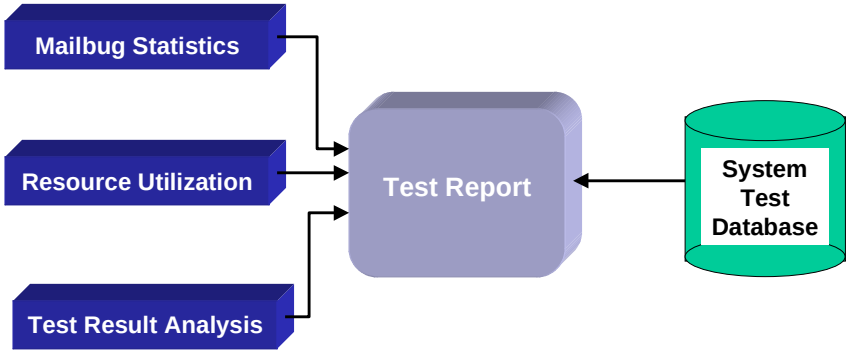


16


Query Interface
Query by Project
Query by Product
Query by Configuration
Query by Category
Query by Requirements
System Test Database
TPL View
Engineer View
External View
17


Test Case Execution Model
Test Cases
Scripts with Harness
Lab Resource
User Interface
Down Load Test Suite
Testing Surface
18

Test Report Generator



19

Implementation

- Process: plan, design, implement and test
- Fully implemented Phase I with module: TPM, TCM and ADMIN
- Design method: database, web interface, top-to-down functional module and detailed functions
- Testing phase: Alpha for functional test and Beta in a real release

20

Summary

- Phase I completed successfully and in production.
- Enhancement requirements were generated for Phase II and Phase III
- Primary objective has been achieved with:
  - Test Plan generation cycle time reduction
  - Limited resources
  - Limited time frame

21

Definitions

- **Test Case Developer**: **responsible for developing test case that fulfills testing requirements.**
- **Test Plan Builder**: **duty to build the final system test plan based on individual test plans.**
- **Test Case Validate Process**: **a process that changes a test case status from 'NEW' to 'VALID'**
- **Idle Resource Pool**: **contains current available resources.**
- **System Components**: **hardware resources used to build a specific system configuration.**

22

Definitions (Con.)

- **Query for information**: **query System Testing Database (STDB) for information on project, product, configuration, category and requirement.**
- **Query for views**: **query System Testing Database (STDB) through different roles - Technical Project Leader (TPL), Internal Engineer and External Engineer.**
- **Components of Test Case automation database**: **Test Cases, Test Scripts with Test Harness and Lab Resource.**

23

Definitions (Con.)

- **User Interface**: **web-based test case driven environment.**
- **Testing Surfaces**: **a testing environment that consists of hardware and software.**
- **Mailbug Statistics**: **metrics for mailbug types, severity, histograms.**
- **Resource Utilization**: **metrics of resource flow, allocation and reference.**
- **Test Result Analysis**: **metrics of test status for pass, fail, delete, etc.**

24

A Web-based System Testing Repository Model

Fan Yang, Trung Nguyen & Anant Adiga
Sequent Computer Systems, Inc.
15450 SW Koll Parkway
MS=DES 2-798
Beaverton, OR 97007
Fax: (503) 578-3228

Abstract

This paper describes the design and implementation of a web-based-automated System Test repository model.

Traditionally, the test automation process is focused on the automating of test scenarios under test harnesses. However, looking at the entire testing process, the automation of test scenarios itself is only one element of a testing process. There are many other important elements. Elements that can significantly impact this process, and yet are often overlooked are the generation of test plans, test case generation, resource management and test report creation.

This paper describes the details of this web-based System Testing model through the design, implementation, and testing for the first phase that consists of a Test Plan Manager, Test Case Manger, and Administration modules. It also describes the future elements for the next phases, consisting of the Resource Manager, Database Query, Test Case Execution, and Test Report Generator modules.

Also included in this paper are the details of a feasibility study and benefit analysis for this model. It describes the strategy employed to overcome problems of complexity in the requirements, budget limitations, and the challenge of reducing testing cycles in order to meet the Time-To-Market (TTM) pressures.

1 Introduction

This paper will cover the analysis of the system test work tasks, and the design of the system test repository model that is currently being developed at Sequent.

1.1 Systems Test at Sequent

The Systems Test group at Sequent is part of Sequent's World Wide Engineering and is responsible for all the system level testing for Sequent. System level testing consists of Integration testing, System Usage testing, and Customer Usage testing with the test level of involvement increasing as shown in Figure 1.

- Integration testing consists of installation/de-installation testing, configuration testing and compatibility testing.
- System usage testing consists of volume/load testing, stress testing, system migration testing, maximum/minimum system configuration testing, performance testing, recovery testing, SW/HW fault insertion testing and interoperability testing.
- Customer Usage testing focuses on simulating customer operation profiles.

To effectively conduct the Integration tests, it requires a thorough understanding of the usage of the products under test. For System Usage tests, it requires a broad knowledge of all components of the system (HW, FW and SW). Lastly, for Customer Usage tests, it requires an understanding of some specific customer's environment, which includes system configurations and application usage patterns.

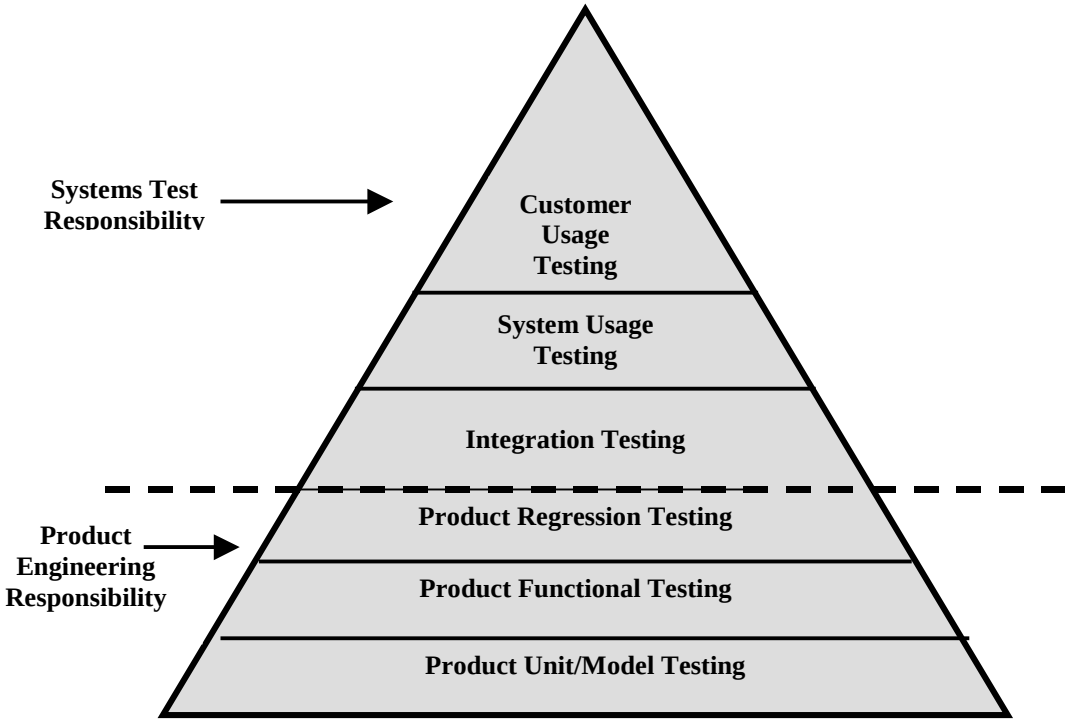


Figure 1. Product Engineering and Systems Test Responsibility Pyramid

From our analysis, tasks performed in Systems Test are divided into unique processes. These processes are test plan generation that includes test plan generation, resource assignment and allocation (utilizing system configurations and system test engineers), and test case execution.

2 Analysis Phase

In the analysis phase, we wanted to know what are the requirements for the model. Once they were established, we did a search to find if any current tools exist that would meet the requirements for the model.

2.1 System Test Requirements Analysis

We started the project by gathering the requirements in a brain storming session among managers, project leaders, and senior engineers in Systems Test. There were several main goals that were deemed mandatory for the product model:

- It should be able to capture the knowledge and expertise that a system test engineer has developed over the years in a form that can be transferred effectively to a new system test engineer. The test plan and test case generation depends largely on the expertise and skill sets of on individual test engineer that often take a long time to develop.
- It should be able to automate production of test plan and test case generation to reduce the time to produce the documents required for testing.
- It should be able to quickly select test cases for a specific product or system for regression testing.
- The model should increase testing efficiency and reduce redundancy at the same time. This goal was set in an effort to reduce the testing back-end schedule by creating a smaller number of better test cases to execute. This is important in situations where there are SW development delays and the ship date is fixed. The burden is always on systems testing to compress its work tasks. This can be expensive and frustrating to most test engineers.
- The model should be able to automate test setup and test case execution.
- The model should automate resource assignment and allocation to efficiently handle Systems Test capital spending.

2.2 Existing Tool Analysis

A common thread running through the requirement analysis was a need for an integrated data source. We needed to determine if there was an existing product that was based on a common data source that would meet our requirements.

What was found was that information generated, like test cases and test plans, were driven by formats dictated by the application used to generate it. This generated disparate data sets from multiple applications. The problems to acquire and access a central data set for generating accurate and useful information were apparent.

- These *disparate* data sets did not lend themselves easily to any kind of postmortem or decision making for any future System Test Plan.
- End users would not be aware of the availability of historical data or its existence. The test engineers would not have visibility of previous documents; for example, test cases or test scripts.

- The data generated by these multiple applications were not thoroughly documented; the real content and meaning of those data were seldom understood

This also hampered the efficiency in which System Testing could:

- Conduct Root cause Analysis
- Define test harnesses
- Streamline Hardware testing process with Software test process
- Maintain an integrated data resource

2.3 Limitation of Applications/Tools in the Market

During our search for a third party product that would meet our requirements, we studied several products of which one product closely matched our requirements. However, there were some major constraints that prevented us from using it.

- Limited portability to the Sequent platform. Other hardware would add to initial project cost plus the costs of its maintenance, administration and any future upgrades.
- Costly porting both in terms of time and budget to the Sequent platform.
- Customization levels that were too restrictive to meet our needs. This was basically because of the complicated nature of testing configurations we use in Systems Test.
- Not Web enabled. Being web-based was not first considered a requirement, but was later seen as an advantage to being part of the store model.
- Built around a proprietary SQL database.
- Not targeted on entire System Testing integrated automation as our requirements but focused on single point of a specific test harness for test case execution.

3 Development

3.1 Development Environment and Tools Selection

We chose a three-tier model as our project development environment as shown in Figure 2. In this model, we configured the database server on an NT machine. The database was created on a Dynix/ptx system. The end users access the application from the web-browser. This model provides a flexible application and database maintenance ability and good data throughput.

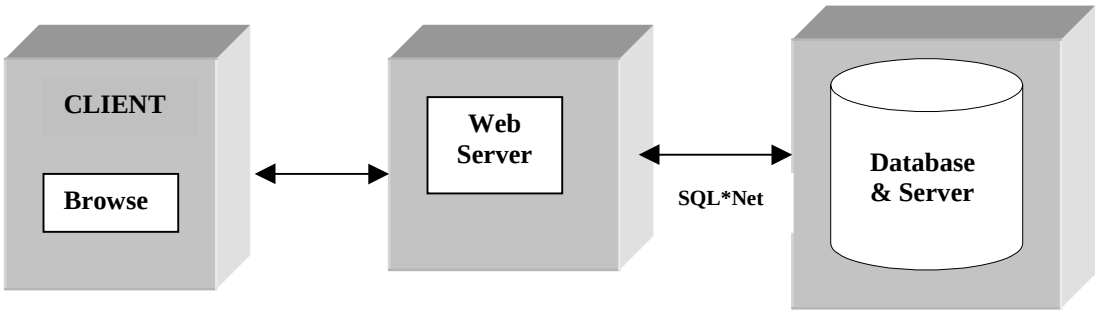


Figure 2. Three-Tier Client/Server Model

We selected the Oracle database server and its set of development tools for development and deployment of this project. Key factors that contributed to this decision were the availability of in-house systems and human resources, the expertise levels of in-house engineers, time frame and budget restrictions. Other factors that were also considered were:

- Oracle Products are strongly supported on Sequent computer systems; for customers and in-house IS usage.
- The product met the requirements for data backup/restore and data import/export tools.
- The web-based features of the Oracle Products lend strong support to the Repository Model.

3.2 Partitioning the Module Design and Project Phases

A significant number of requirements were collected and organized under several major functional and operational categories. The next step was to convert these components from the model into software modules that to be developed in three phases. Table 1 shows the Software Modules defined for this project based on the requirement analysis. The phase notation covering the project phase scope for modules is shown in Figure 3.

Requirement	Software Module
Automation	
Automating system test plan generation	Test Plan Manager (TPM) – Phase I
Managing and manipulating test cases	Test Case Manager (TCM) – Phase I
Tracing from test cases to release requirements	Administrator (ADMIN) – Phase I
Supporting GUI interface (web-base)	
General query and resource management	
Querying database with different roles	Resource Manager (RSM) – Phase II
Querying system configurations	Database Query (DBQ) – Phase II
Automating lab resource management	
Accessing hardware configurations	
Test execution automation	
Driving the test case execution	Test Case Execution (TCE) – Phase III
Generating test report	Test Report Generator(TRG) – Phase III
Keeping history of test case execution and mailbug records	

Constructing the uniformity of test harnesses	
Controlling elements using configuration control mechanism	

Table 1. Requirement and Software Modules

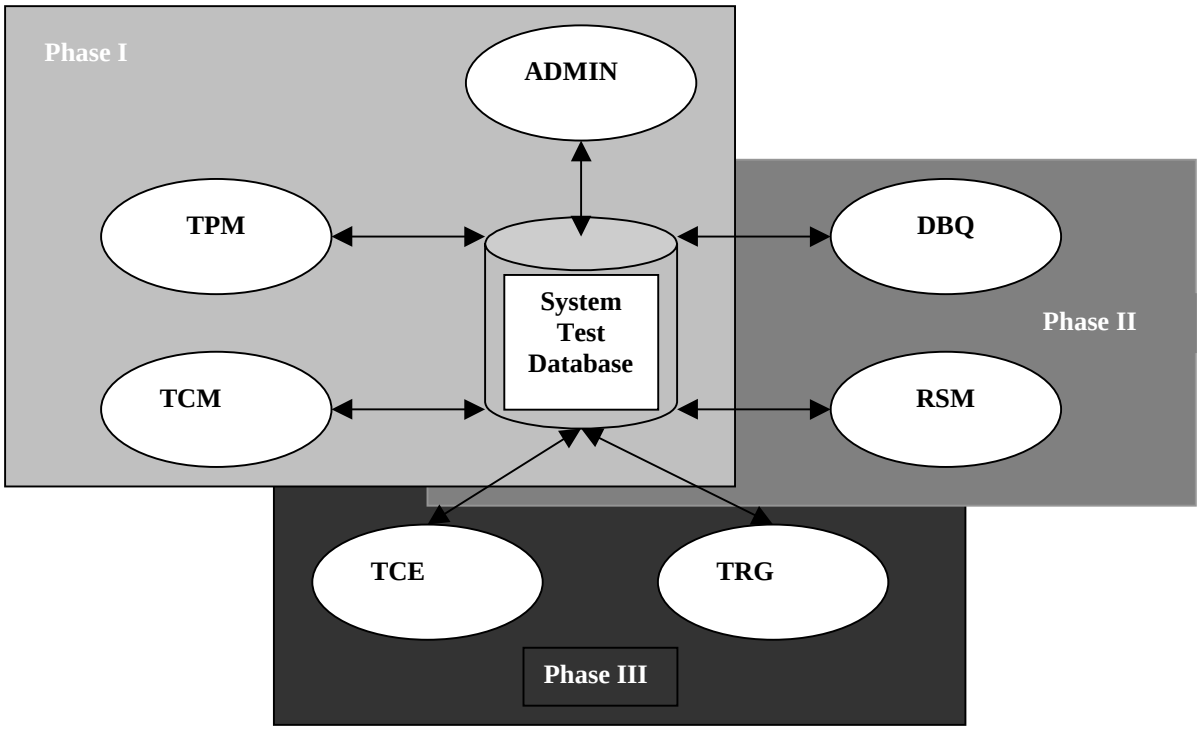


Figure 3. Project Phases and Modules

3.3 Module Design Detail

The next section will cover each of the module design details.

3.3.1 Test Plan Manager (TPM)

This module implements the functionality of generating a system test plan. Traditionally, we manually create a system test plan by referring to the previous plans and extract proper test scenarios for the new test plan. There are several major disadvantages of using this method:

- Easy to make a mistake.
- Takes significant amounts of time to read through previous plans.
- Difficult to keep consistent test case names from release to release.
- Hard to keep track of the coverage of products and configurations.
- Hard to map requirements to test cases.

At a minimum we require the Test Plan Manager (TPM) to address and resolve the above drawbacks. In this module, we define two roles for test plan generation: ***Test Case Developer*** and ***Test Plan Builder***.

Test Case Developer (Developer) – is responsible to generate the test cases that fulfill testing system requirements by utilizing existing test cases or creating new test cases.
Test Plan Builder (Builder) – is responsible to build the final system test plan based on the ***Test Case Developer***'s test cases from requirements.

The process to generate the system test plan for a specific software release bundle is:
Step 1. ***Developers*** create the test cases based on system requirements from the generic database tables.
Step 2. The test cases are stored in temporary tables in the database.
Step 3. ***Builder*** builds the final test plan from the temporary tables in the database.
Step 4. Repeat from Step 1 to Step 3 as necessary. The Figure 4 shows the test plan generation process.

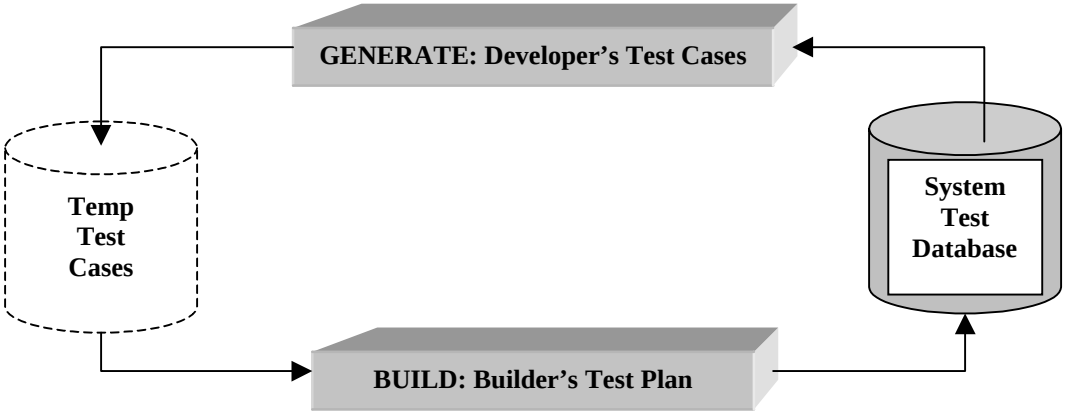


Figure 4. Test Plan Generation Process

3.3.2 Test Case Manager (TCM)

How are the System Test Cases re-used from release to release? How are the data shared among the projects? How are the new test case data captured? In the traditional way, we re-use the test case data only if the test scenarios are still valid in the current release. Most of them have to be re-written and different names are typically used in each release. To overcome the drawback of the traditional method and answer the above questions, we define a set of seed test cases in System Testing as the ***Generic Test Cases***.

Generic Test Case – a test case that contains only generic and core test scenarios and can be re-used from release to release.
The Test Case Manager (TCM) is designed to generate the ***Generic Test Cases***. It provides a flexible environment for engineers in Systems Test department as well as external engineering groups to view the data in the System Test Database. The new

test case scenarios and ideas can be captured into the database independently of any release or project.

The test case developers can VIEW, INSERT, MODIFY and DELETE the test cases in generic database tables. Any newly inserted test cases need to be validated through the ***Validate Process*** before being marked as 'VALID' test cases in generic database tables. The ***Validate Process*** is defined in section 3.3.3.2. The process that manipulates generic test cases is shown in Figure 5.

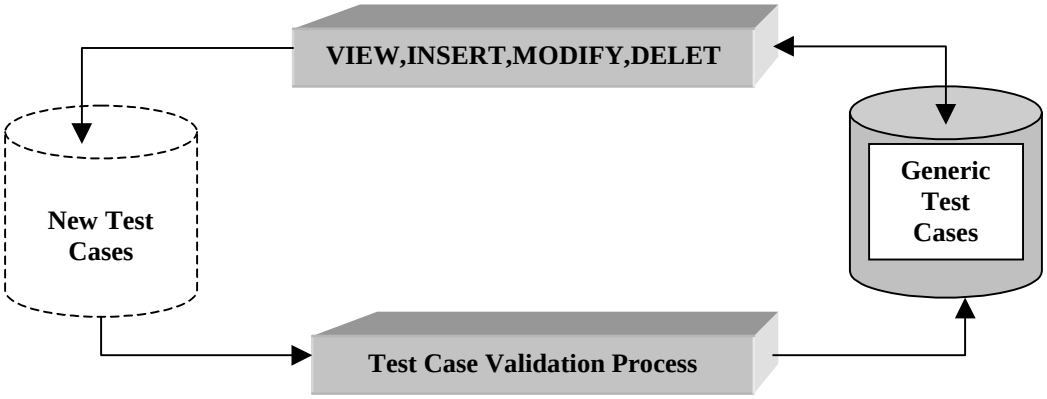


Figure 5. Generic Test Case Generation Process

3.3.3 Administration (ADMIN)

This module provides administration tools to manage the System Test Database. The database administrator can manage the database tables through these tools without going through SQL command lines. It also provides a direct method for a ***Builder*** to create a new project. We also defined the test case validation process for the TCM (Test Case Manager) here.

3.3.3.1 Process to Create a New Project

We define the process flow to create a new project as shown in Figure 6. The procedures are:

Step 1. Create the project header. This includes the project name and date.
Step 2. Create the project team. The ***Builder*** assigns the team members and grants the members different access privileges.
Step 3. Input the requirement tracking numbers for this release. The requirement tracking number is associated with a test case during the test plan GENERATE stage in the TPM (Test Plan Manager).
Step 4. Input the list of products and possible hardware configurations for the project. The list of products is used in the TPM (Test Plan Manager) module to associate each test case with targeted products.

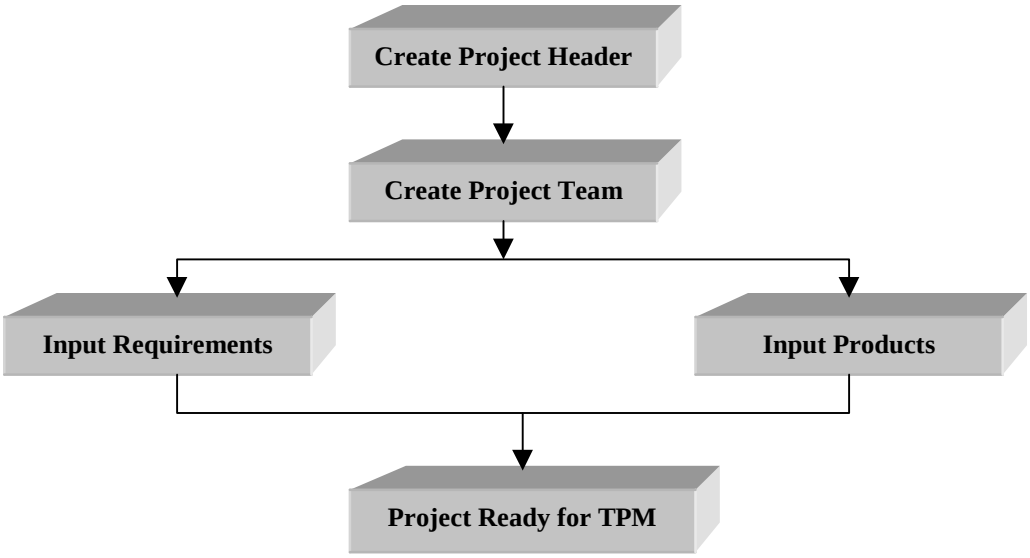


Figure 6. Project Creation Process

3.3.3.2 Process to Validate Test Case

New test case can be inserted using either the TPM (Test Plan Manager) or the TCM (Test Case Manager), and it is necessary to guarantee that these test cases are valid. We define a process called Test Case Validation to validate a new test case and a ***Test Case Validation Team*** which is responsible for executing the ***Test Case Validation Process***.

Test Case Validation Process **-** A process that validates a test case to change its state from 'NEW' to 'VALID'.
Test Case Validation Team (TCVT) – A team that is responsible for executing the ***Test Case Validation Process***. ***TCVT*** consists of senior level engineers in the Systems Test Department who meet on a monthly basis to review all new test cases and existing generic test cases. ***TCVT*** votes to **Enable** a new test case for changing the state to 'VALID' or **Disable** an existing one. ***TCVT*** also has the privilege to modify a new test case to fit into the requirements for a generic test case.

3.3.4 Resource Manager (RSM)

The System Test lab provides the complete system environment for testing and usually supports several projects/releases simultaneously. The equipment is shared among the projects/releases and with external groups. The equipment in the System Test lab also rotates out periodically.

In order to manage these dynamic resources, we have used several different tools, such as a spreadsheet, word document, etc in managing the Systems Test lab. However, we found it was necessary to centralize this kind of information; otherwise

it was not easy to provide the accessibility within either the internal group or external groups in such a manual process.

To improve this process, we developed the Resource Manager (RSM) module to manage System Test lab resources dynamically. Without an automated method to manage the lab equipment, it is very difficult to implement full test case automation and allocate resources automatically.

In this model, we define a ***Resource Allocation Process*** as shown in Figure 7. The procedure is listed below:

Step 1. Query the *Idle Resource Pool* that contains the currently available resources. The query is based on these criteria:

System components – this can be the major components of a system. For example, number of quad, size of memory, number of disks, communication cards etc. A limited system configuration is checked, such as 4 processors for a quad, minimum memory requirement for the 4 quad systems.

Equipment – this is the request for a piece of equipment like one GB of memory, five disks, etc.

Step 2. If the resource request can be satisfied from the *Idle Resource Pool*, remove the resources from the *Idle Resource Pool* and grant the resources. Exit. Else, continue on to Step 3.

Step 3. Compare the priorities of the requesting project and existing projects that hold the resources. Allocate the resources based on the project priorities. The request may be inserted into the *Resource Waiting Queue*, a queue that contains waiting resources, if the priority is lower than any current projects.

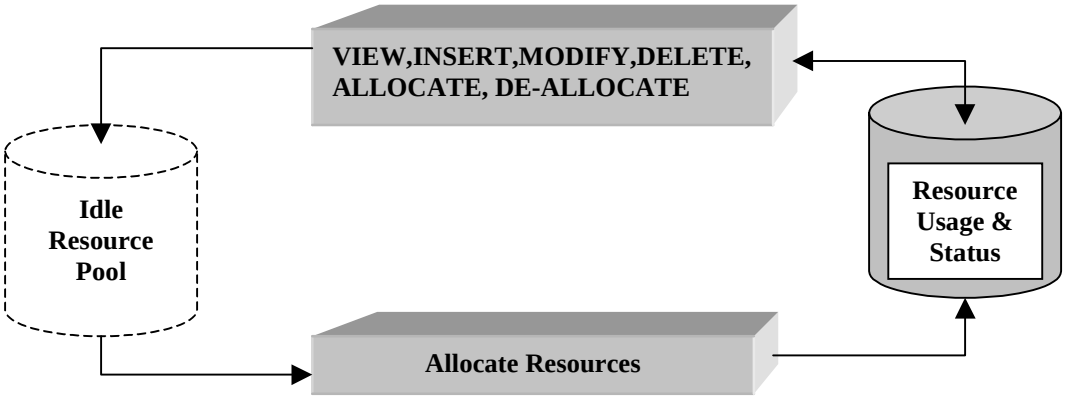


Figure 7. Resource Allocation Process

3.3.5 Database Query (DBQ)

The query mechanisms provided by the TPM (Test Plan Manager) and TCM (Test Case Manager) are not sufficient to satisfy different database query requirements.

Therefore, we need to develop the DBQ (Database Query) module to provide various interfaces for accessing the database information. We establish the query methods under these categories:

Project – Query is based on a specific project. The user can access a specified project or release information, e.g. test plan, lab resources, system configurations, etc.

Product – Query is based on a specified product. The user can access the test cases based on both product and project/release. It provides a very convenient way for a product group to access the test cases in Systems Test plan that cover a specific product.

Configuration – Query based on system configuration gives a reference to the previous test case execution environment. A configuration specified by components is also accessible from this interface.

Category – Query by Systems Test category combined with project/release creates a user interface to access the database information from different perspective.

Requirement - Query by requirement provides a way to extract the test cases associated with a specific requirement within the project/release.

We also define several database access layers by the user's ***Role***. The ***Role*** consists of:

TPL(Technical Project Leader) – The role that is able to oversee the entire database information, such as team member's privileges, other user's access views, etc.

Engineer – The role that is for System Testing engineers to have a complete view of the data in the database for the purpose of test case development.

External – The role that is for anybody outside of the Systems Test group who need to access extracted System Test Database data without being exposed to many details.

Each ***Role*** has a different view interface. Combined with the database query mechanisms, we are able to provide a flexible user interface to access database data as shown in Figure 8.

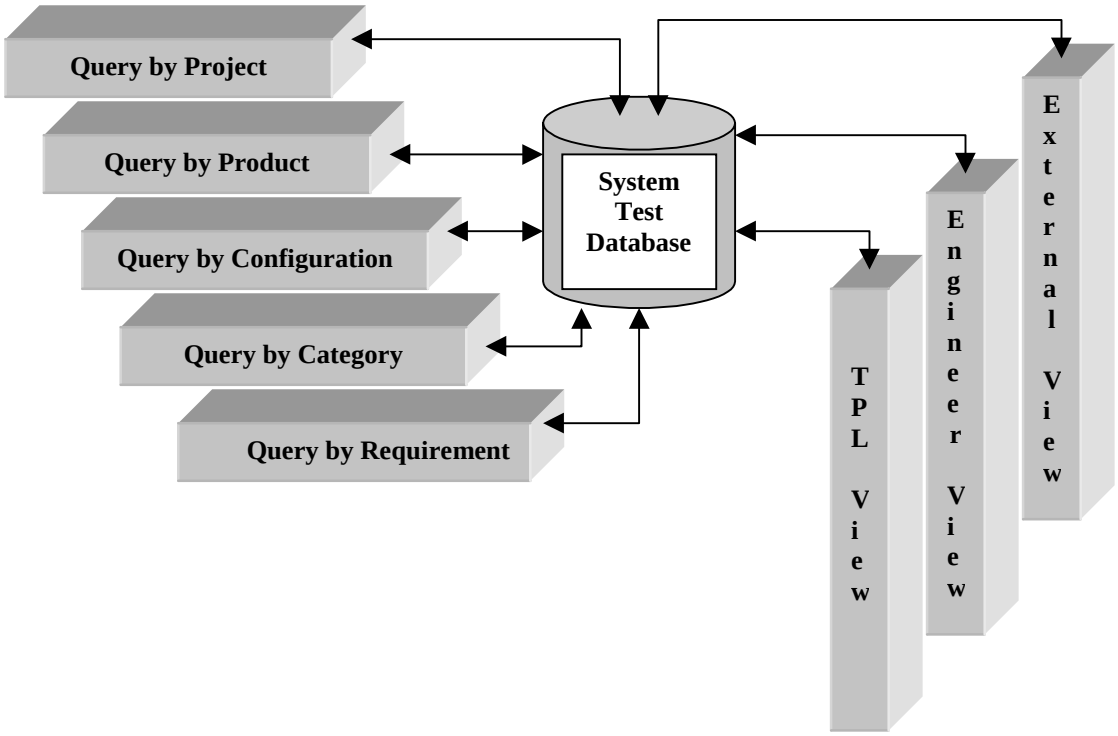


Figure 8. Query Interface

3.3.6 Test Case Execution (TCE)

Traditional test case automation is typically limited to automating the test procedures. The scripts and programs are developed to reach this goal. There are some well-known test harnesses that provide the test automation environment and tool kits for test automation, e.g. eTET/dTET, Performix, QA Partner, etc. How completed a test case can be automated is a traditional focal point. In general understanding, a *fully automated test case* means every procedure in the test case is programmable and a test case can be executed by submitting a single command. In reality, only a small portion of test cases can be automated to this degree. In Systems Test, the majority of test cases involve manual procedures, such as fault insertion, availability or installation tests. Besides, how long a *fully automated test case* survive is questionable [5]. We believe a fully automated test suite should be an integration of the ***Test Plan, Test Resources and Test Procedures***. A test case cannot be called fully automated if it lacks any of these three elements. Therefore, we focus our system test case automation on how to integrate these three elements together.

We designed the test case execution model as show in Figure 9 in order to meet the goal of fully implementing the test case automation based on the above factors.

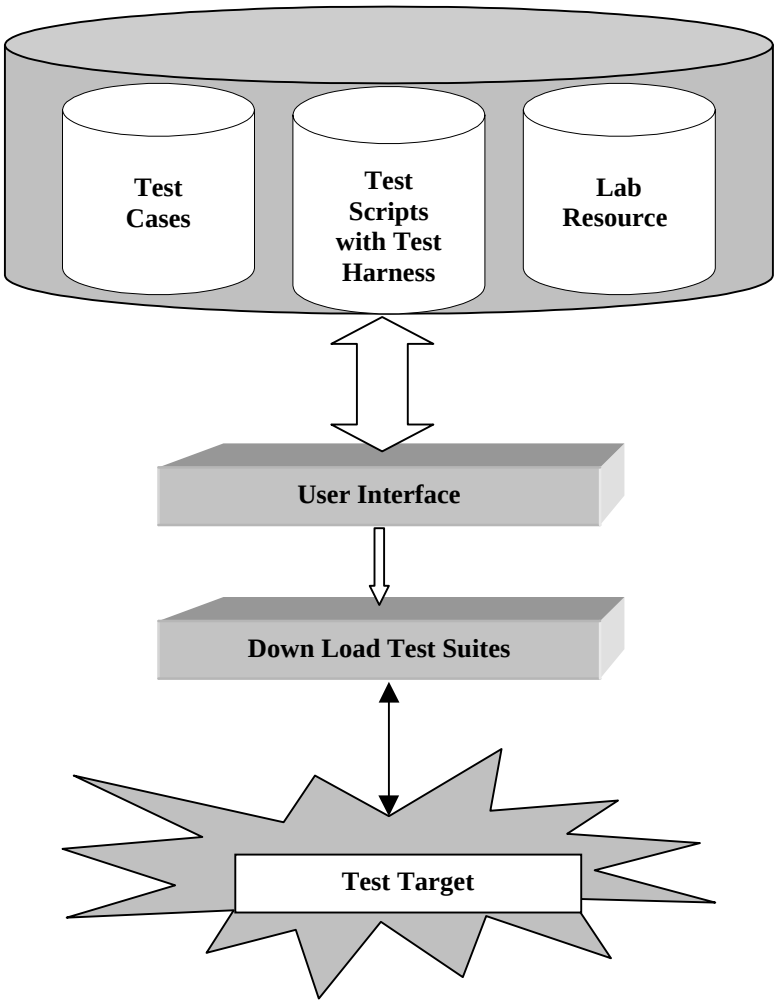


Figure 9. Test Case Execution Model

In this model, the ***Test Cases*** in the test plan, the ***Test Procedures*** implemented by scripts or programs under test harnesses, and the ***Lab Resources*** are integrated into one package. This package is downloaded onto the ***Test Target***, a testing environment consisting of system hardware and software. A test case is executed on the ***Test Target***.

3.3.7 Test Report Generator (TRG)

A System Test Report usually is a summary of testing methods, results and problems encountered during the testing phases of a release/project. It is important to list accurate data and document the significant experiences in the Testing Report as a reference for later releases/projects. Writing the System Test Report is very time-consuming and requires a lot of work to collect the statistic data and recount the project experiences. What kind of mailbugs were found during the testing phases? What did we 'learn' from each test case execution? How do we know a test case is

'good'? What do we gain for planning the current release by reading the previous System Testing Reports? The Test Report Generator attempts to answer the above questions. We design the Test Report Generator (TRG) to not only reduce the time of generating the report, but also to provide a useful and active reference for future releases/projects. The Test Report Generator (TRG) module consists of these major aspects:

Mailbug Statistics – statistic metrics of mailbug for the release/project. The typical examples are: types of mailbugs, histogram of number of mailbugs and date, ratio of Critical/Serious mailbugs and Normal/Low mailbugs, etc.
Resource Utilization – metrics of resource utilization. The information on system configuration and resource allocation is very useful for later releases/projects. The metrics give the resource overview for the release/project.
Test Result Analysis - metrics of test results for test status; for example, PASS, FAIL, etc. The metrics also give the overview of the test cases and mailbugs resulting from these test cases.

Besides the above information, the Test Report Generator (TRG) module also provides other information to the Test Report, such as product information, requirements, testing cycles, and so on. Figure 10 shows these aspects and their relationships.

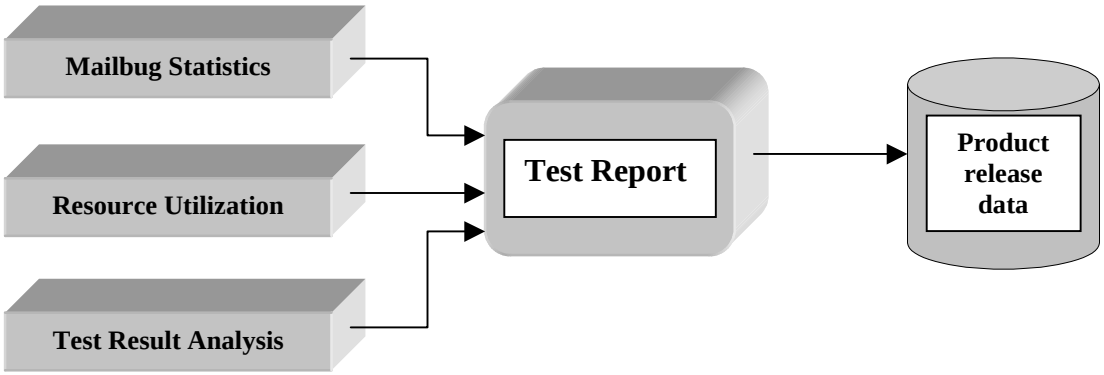


Figure 10. Test Report Generator

4 Implementation

4.1 Implementation Process

Implementation of this project was extremely challenging due to the following major factors:

- Extensive project requirements made the project have multiple and complicated goals.
- Tight project budget and time frame control the project under a very restricted scope.

- Limited project resources do not allow the project to move on in parallel in order to cut down development time.

In order to cope with the above problems, we have divided the implementation of entire project into three Phases as previously mentioned. Two to three modules are developed in each Phase. We planned each phase for a six-month period followed by complete Beta testing in a real-life product release project.

In each phase, we follow the complete product development cycle as:

1. **Plan –** initial research and writing of a plan document.
2. **Design –** a detailed functional specification is defined.
3. **Implement –** the code is developed.
4. **Test –** complete Alpha and Beta testing is conducted.

4.2 Phase I Implementation

As described in Section 3 above, in Phase I, we implemented the TPM (Test Plan Manager), TCM (Test Case Manager) and ADMIN (Administration) modules.

In the plan phase, we did the research on third-party products and investigation of our current environment. Formal planning documentation was written in this phase. The self-training of new technologies was also started in this time period. These areas are Oracle Application Web Server, Web programming, PL*SQL programming with Web, etc. Due to the tight budget and time schedule, we found in-house self-training was very efficient in reaching our goal.

The Detailed Functional Specification was written in the design phase. In this phase, we designed the process flow, functional modules for the TPM (Test Plan Manager), TCM (Test Case Manager) and ADMIN (Administration) as web-based user interfaces. The Detailed Functional Specification also contained the interface and pseudo code for each function designed.

In order to work more efficiently in developing the source code, we started the programming in the order of database manipulation, web-interface, functional module and detailed functions. We also built two databases for development and testing on different systems. During the implementation, we referenced a lot of PL*SQL examples from the books listed in the References, web sites for Java scripts and user interfaces. We found that on-line Web pages were very useful references for developing applications like this one.

We provided the testing team with an Alpha phase functional testing plan. The testing team consisted of 5 engineers, Technical Project Leaders and manager with different user-level roles. Major test conducted were functional tests, recovery tests

and stress tests that simulated the real-life environment. After the bug fixes for Alpha testing, this system was used in a major release project in Systems Test as a Beta test.

5 Summary

Phase I of this project has completed successfully. The product (STDB) is being utilized by the Systems Test engineers to develop test scenarios and test plan for the next major software system release. We have recorded a handful of enhancement requests to be implemented in phase II and III of this project. Phase II will focus on implementing the database query and resources management. A small team will be formed in the next three months to start working on requirement definitions and detailed specifications. Phase III is not scheduled at this time due to the limited resources.

Looking back, we believe that we have achieved the primary objectives of this project (phase I implementation) with the lowest investment. The primary objective of this project again is to capture the knowledge and expertise of system test engineers and store it in a database management system that is easily accessible from anywhere within the company through a web based interface. The Systems Test group is able to generate faster, more effective test cases and test plans to support system releases. Now, with this new tool, we can have sophisticated test plans (with many test cases) generated in a couple of days instead of the couple of weeks that it previously took.

References

[1] Rick Greenwald, Using ORACLE Web Application Server 3, Que Corporation, 1997, ISBN: 0-7897-0822-1.

[2] Scott Urman, ORACLE 8 PL/SQL Programming, McGraw-Hill, 1997, ISBN: 0-07-882305-6.

[3] Dynamic Information Systems, LLC, ORACLE Web Application Server Handbook, McGraw-Hill, 1998, ISBN: 0-07-882215-7.

[4] Marty Hall, CORE Web Programming, Prentice Hall PTR, ISBN: 0-13-625666-X.

[5]Brian Marick, When Should a Test Be Automated? The Eleven International Software Quality Week, 1998.

Test Automation of a GUI WEB Based Application: An Experience Developing Reusable Automated Testing

Frances Medina

Andrew Van Kraanen

GUI WEB Application

- Client/Server
- WEB server and interfaces are UNIX based
- Netscape WEB browser
- Frames
- Multiple windows can be opened
- One of the window dynamically displays information coming from interfaces to the server

Challenge

To get a test automation tool compatible with our WEB application

Strategy

- Start automating simple tests until a tool compatible with our application was available
  - GUI objects content verification
  - window traversal

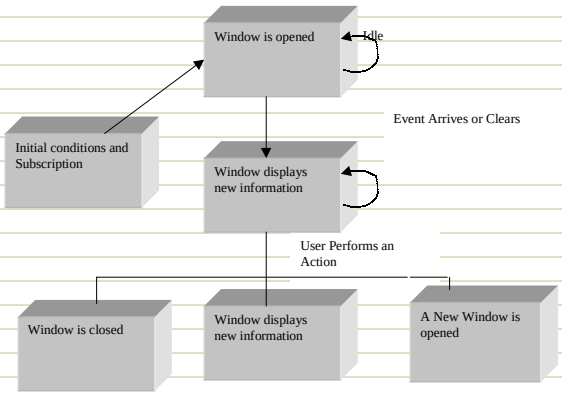
WEB Automation Test Tool

- SilkTest tool from Segue
- Was tested with our WEB application before purchasing it
- Different test methodology

Test Methodology

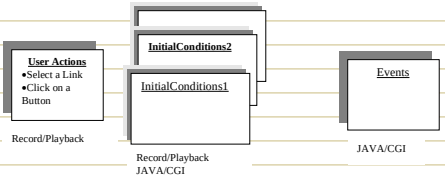
- Data driven
- Operational Profile describing the various operations (scenarios) the system will perform in the field.
- Testing Scenarios including combinations of factors and values affecting a WEB window/user interface.

Operational Test Scenario



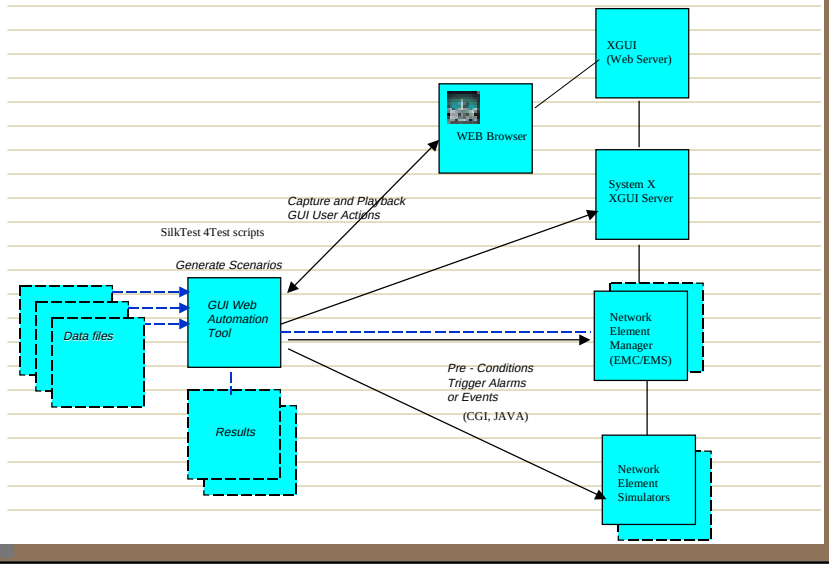
Automation of the Operational Test Scenario

DATA FILES

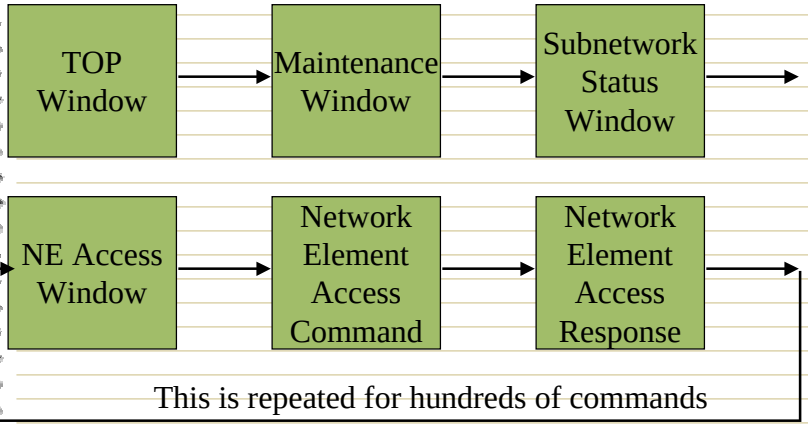


MAIN PROGRAM

Automation Test Architecture Operational Scenario



NE Access Scenario



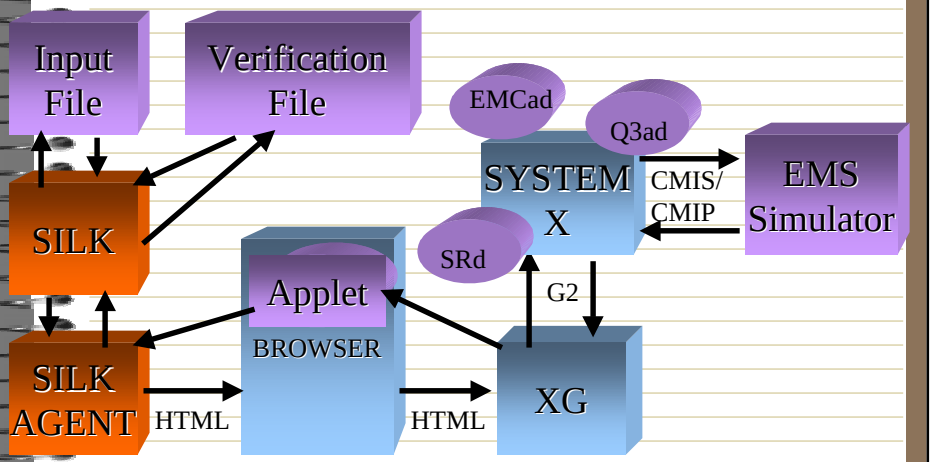
Automation with SilkTest

- Data driven testcases
- Use of verification files
- Multiple cases per command
- Recording/Playback

Automation with SilkTest

- test(Command, Case, Mode)
  - Command : createCrossConnect
  - Case : STS1-case
  - Mode : [init|verify]
- Base run with option “init”
- verify run with option “verify”
- Input File I<Command>
- Verify File V<Command>

NEACCESS Interaction



Results

- Several test scenarios have been automated and used for different software releases, including Y2K regression testing
- Multiple commands have been automated and tests have been running for several days

Lessons Learned

- Know your WEB application well before purchasing any tool
- Come with an scenario that could be automated
- Before purchasing any tool, ask the vendor to test drive the tool with your application and at least try to automate a test scenario
- When using an automation tool, come with small and realistic goals about what will be automated
- There is a learning curve associated with the use of any tool. There is no such thing as a plug and play and the problem is solved.

Lessons Learned

- Do not rely only on the capture and playback of the tool for the automation. Use it just when needed.
- In order to use automation tools a programming background is recommended
- Programming might be involved in order to create a robust automation script
- Tools are not perfect. There is software development involved on the creation of those tools.
- There is not an immediate benefit when purchasing an automation tool.

Lessons Learned

- Some time need to be invested: knowing the tool, coming with the right methodology to automate and developing the programs or script to implement the test automation.
- Automation is very useful and can be reused from release to release as long as good planning and analysis of the application under test is done.

Test Automation of a GUI WEB Based Application: An Experience Developing Reusable Automated Testing

Abstract

The GUI Interface for our application is making use of a Web Browser (Netscape). The user of this application is exposed to several WEB windows which some of them dynamically display information to the end user; others are used for provisioning or to send request or action to a server or to equipment. Several scenarios are tested for this application; submitting information in forms, dynamic changing applets triggered by an external event, clicking buttons, html links etc. To be able to tests all those scenarios for each release would be very time consuming. A commercially available tool was purchased to do the GUI automation. Then the question was what and how to automate, such that the test automation could be used for different software releases. This presentation shares our experience with web automation tools, methodology used in our testing and improvements made by using test automation tools.

1. Introduction

The information contained in this source is from purely practical experience while trying to come with a solution for GUI regression test automation in our project.

Three years ago, when we joined the project in which we are currently working, we were hired with the goal of automating regression tests. By that time the application to be tested was in the initial development stage and the GUI WEB windows of this application were continually changing. By that time there were not many commercially available tools for GUI test automation. And the few available could not satisfy our testing needs.

Most of the tools available supported Motif based GUI and our application was WEB based. Most of the WEB windows were implemented with frames and at that time the test automation tools did not handle frames very well. Also, one peculiarity of this GUI WEB application is that it contains a WEB window, which is dynamically updated. When we tried a test tool to work with that window, either one of the following happened: the application died or all the transactions happening on that window were slowed down.

Given all the limitations at that time, it was decided to limit the scope of the test automation to just GUI object content verification and what we call in our project acceptance testing[1], until a more reliable tool was available to accomplish the goal of GUI regression test automation.

2. High Level Description of the Acceptance Test for our WEB GUI Application

A typical scenario in our GUI WEB application can be described as follows:

- On a WEB browser, the user opens the location of the application
- The user logs in

[1] Acceptance Testing involves the certification of a newly delivered system or function(s) to insure that it meets minimum criteria for testing. Acceptance testing is done at the beginning of each new delivery.

- If the log in succeeds, the user gets the top window from the application and from there he or she selects to open a menu
- From the menu, he or she decides to open a window which will dynamically display information
- Before that window is opened, the user has to choose or subscribe to items for which he or she will receive information on the dynamic window
- When the subscription is submitted the dynamic window is open.

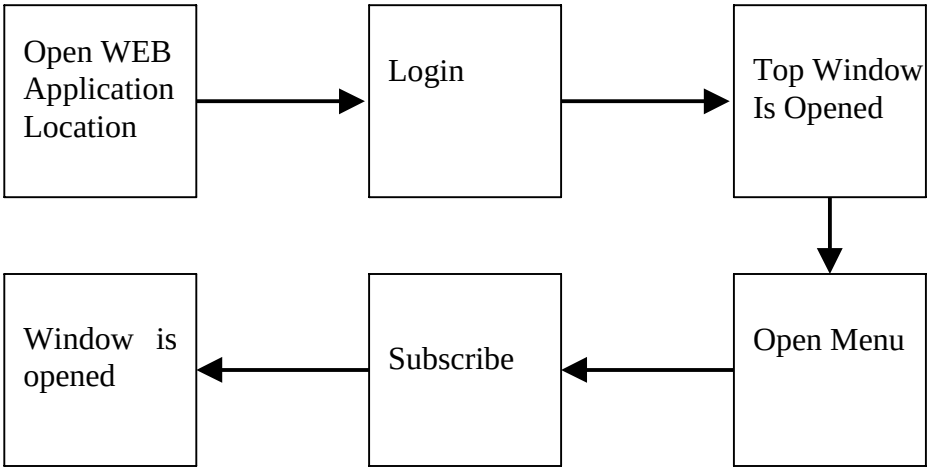


Figure 1: Typical Acceptance Test Scenario

The WEB window that is opened contains text, HTML links, a JAVA applet and links that are dynamically displayed on the WEB.

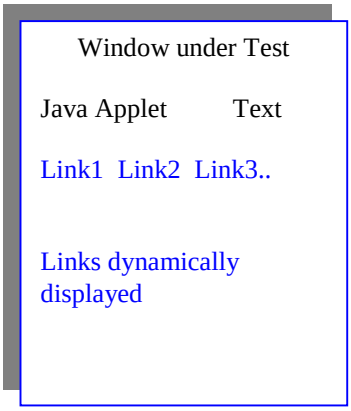


Figure 2: Objects to be verified on a WEB Window

When the window is opened the following can be verified: the content as specified in the requirements and link functionality. Some of the links open a new window but the original window remains open. The links that are dynamically displayed are triggered due to external events. For testing purposes, those events are generated making use of an in house developed simulator. To verify the dynamic links a remote shell command is sent to the simulator and then the output displayed is verified at the window.

Whenever a new software delivery was received, the above scenario was run making use of a beta version of an automation test tool. Then we learned about a new tool.

3. WEB Automation Test Tool

When we learned about the existence of that tool, we call the vendors to test drive the tool. We were allowed to test the tool for two weeks with our application. We were satisfied the way the tool behaves with our application. We started with the simple test that we were using with the previous tool. We found out that the tool handled frames very well and the transactions (play back) performed on the WEB, were not slowed down, even when the dynamically updated window was opened.

What we like the most about the tool was the object oriented functions available to do programming. Those object-oriented functions combined with the tool record/playback made the tests more robust. The tool also contains excellent I/O functions, excellent string manipulation, good error recovery and an easy to use on-line help.

4. New Scope for Test Automation

After seeing the potential of the tool, it was decided to increase the scope of the test automation. Then the question was what and how to automate. Two approaches were taken to start building an automated regression test package.

For the first approach, there were some test scenarios developed for us by other organization. Those scenarios were operational profiles describing various operations that the application will perform in the field. Those testing scenarios include combination of factors and values affecting the WEB user interface. Those factors and values were generated based on Software Reliability Engineering Test (SRET)[2]. A typical scenario might look as follows:

1) A window is opened with random initial conditions – Verify the WEB GUI content and the initial conditions
2) A random event is generated to change the state of the window – Verify the new state of the window because of the new state
3) The GUI WEB user performs a random action (like to click on a link) – Verify the expected output due to the user action

[2] The description of SRET is out of the scope of this paper, but several references are available describing this method. A good reference: Lyu Michael (Editor), Handbook of Software Reliability Engineering, McGraw Hill, 1996.

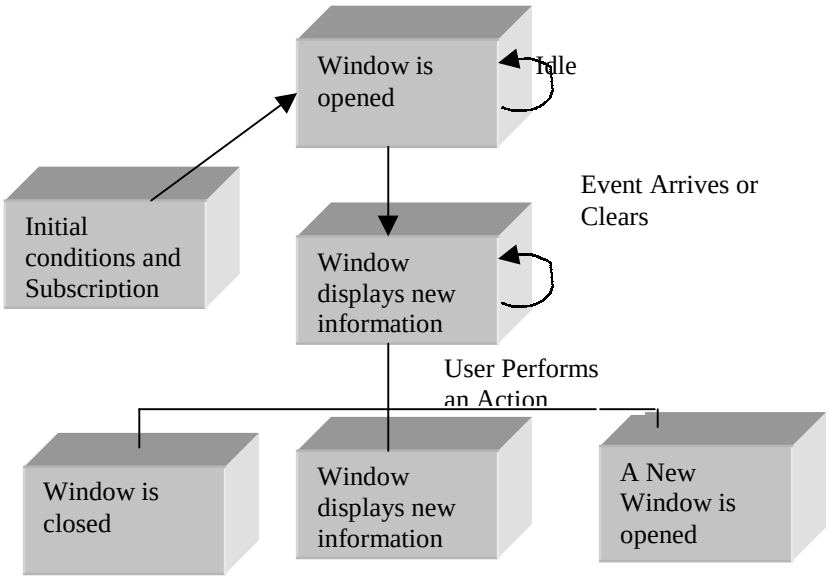
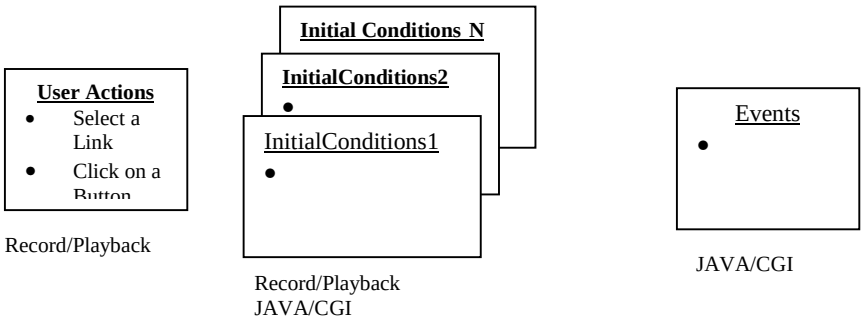


Figure 3: Operational Test Scenario

Data files are created containing initial conditions, events and user actions. A main program (using the tool programming language) is created for the window under test. That main program performs the GUI content verification. Then it randomly selects from the data files the initial conditions, events and user actions. The initial conditions and events might be generated externally by remotely executing commands on the simulator. The verification of the initial conditions and events is performed via case statements available in the programming tool. User actions[3] are playback functions originally recorded from the application. Those user actions might be the click on a link or submitting an HTML form. Verification is performed to the result of the user action. Verifications performed on the window under test might consist on the following: a change on JAVA applet, acknowledgement text due to an event or user action, the display of a new object on the window or the opening of a new window. Different programming functions in the program are called to clean up all the conditions and put the window under test in its initial state. Then on the initial state new randomly generated conditions, events and user actions can be done either on the previously opened window or in a new WEB window.

[3] User actions can be initial conditions or events also.

DATA FILES



MAIN PROGRAM

Figure 4: Automation of the Operational Test Scenario

That automation approach has been very useful for several WEB windows of our application. But there is a particular window, which is like a menu, which contains hundreds of commands (links). Each of those commands generates a different HTML form and when this form is submitted, a new window containing a JAVA applet with the result of the submitted command is opened. In other words there were a hundreds links to select from, a hundred different HTML forms with different fields to be filled in and hundreds of JAVA applets to verify. A typical Scenario will be accessing the following GUI screens (web pages):

1. Top Window
2. Maintenance Window
3. Subnet Status Window
4. NE Access Window
5. NE Access Element Command Window
6. NE Access Element Response Window
7. Steps 4,5,6 will be repeated for all NE Access commands

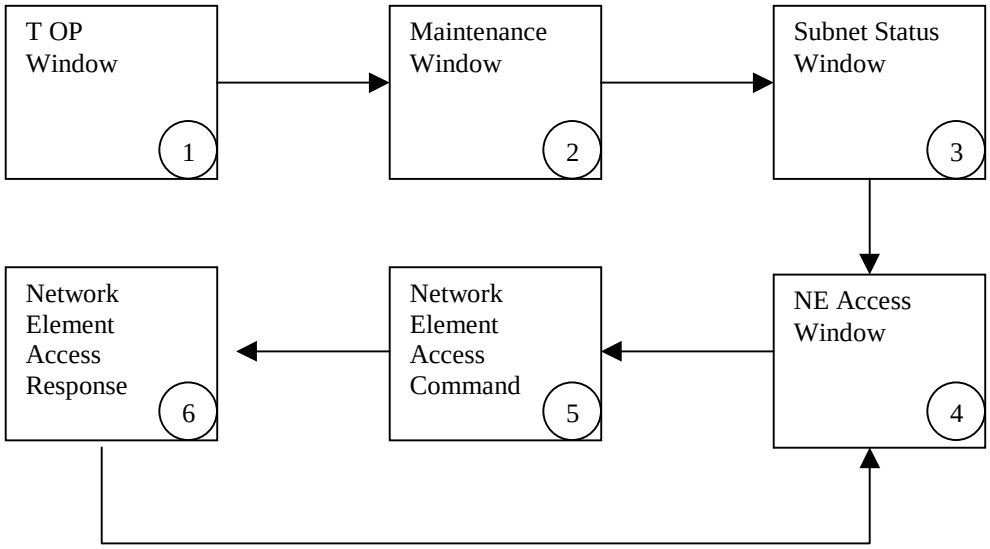


Figure 5: Network Element Access Commands Scenario

To automate that window, an input file is defined for each command. That input file contains all the parameters needed to fill out the form when the command (link) is selected. There is one verification file per command with an entry for each test case. The test scripts used to verify the commands are data driven. The main part of the test scripts is parsing the information from the data files. The data is then used to execute the test cases. The first time the test case is executed, it is done with “init” option. This option creates the output verification file. After that, the test cases are run with the verify option. Then the output is compared with what is stored in the result files.

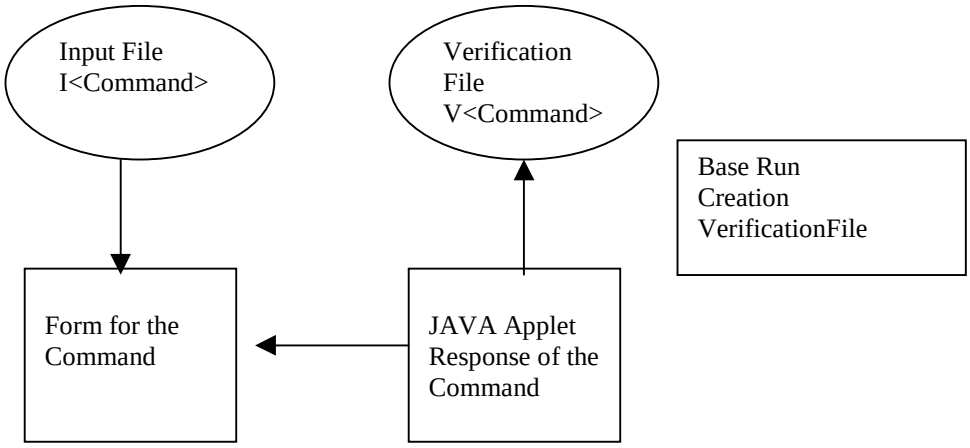


Figure 6: Automation of Network Element Access Commands Scenario (Initialization Part)

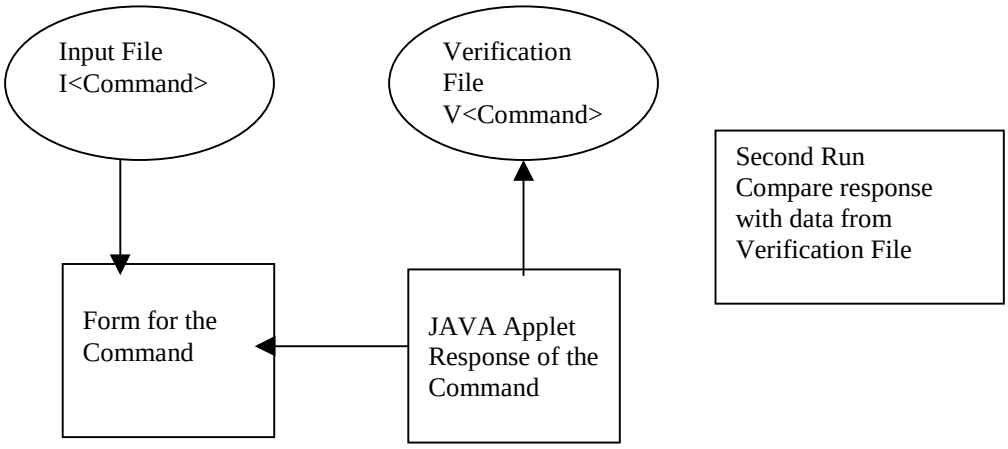


Figure 7: Automation of Network Element Access Commands Scenario (Verification Part)

5. Test Environment

The hardware environment for the test automation tool was a Toshiba TECRA 700 Laptop, 48 Mbytes.

The software environment consists of the following: Windows NT 4.0 service pack 2, SilkTest automation tool version 2.0.1[4], and WEB browser Netscape 3.0[5].

6. Architecture

The following diagram shows the architecture of the automation test:

[4] That was the version originally used for the automation, but the latest version is SilkTest Release 2.1.4
[5] Porting is being done to work with WEB browser 4.06

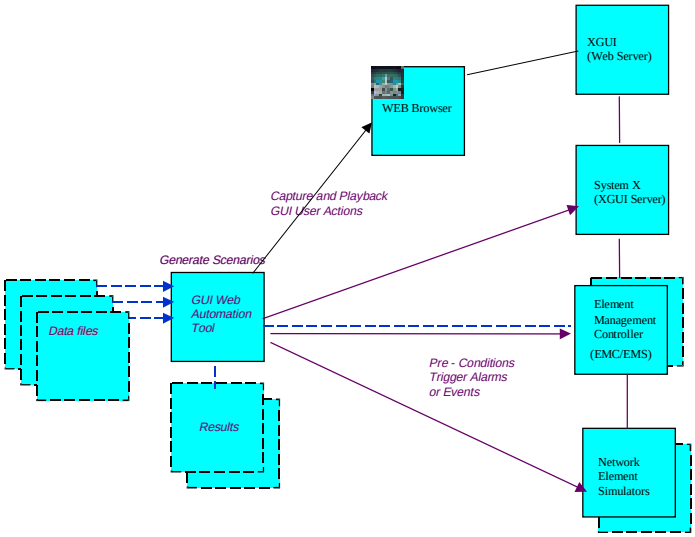


Figure 8: Automation Test Architecture

The WEB automation tool, the WEB browser, the data and results file resides on the laptop. The WEB server resides on a UNIX based workstation. The GUI server, the element management controller and the simulator resides on different HP UX machines.

The programs to perform the WEB test automation are written in 4TEST language, which is the language provided on SilkTest. To remotely execute shell scripts for initial conditions or events on UNIX based machines, CGI/PERL scripts and a small JAVA program was written. The program is executed as a DOS command via a 4TEST function available on SilkTest tool.

7. Results

The scenario-based automation has been used in 4 different application releases, including Y2K regression test for one of the releases. The advantage of using random combination of conditions, events and user actions is that multiple scenarios are created and generated via the test tool and verified at the same time. From the results obtained from the tool, we could determine not only the test that fails and passes but also the execution time for each test. By learning more about the tool, we have found a way of running regression tests on the Network Access Commands Window for more than twelve hours.

8. Lessons Learned

The following items summarized the lessons that we learned while getting the right automation tool and later while using the tool:

- Know your WEB application well before purchasing any tool
- Come with an scenario that could be automated
- Before purchasing any tool, ask the vendor to test drive the tool with your application and at least try to automate a test scenario
- When using an automation tool, come with small and realistic goals about what will be automated
- There is a learning curve associated with the use of any tool. There is no such thing as a plug and play and the problem is solved.
- Do not rely only on the capture and playback of the tool for the automation. Use it just when needed.
- In order to use automation tools a programming background is recommended
- Programming might be involved in order to create a robust automation script
- Tools are not perfect. There is software development involved on the creation of those tools.
- There is not an immediate benefit when purchasing an automation tool. Some time need to be invested: knowing the tool, coming with the right methodology to automate and developing the programs or script to implement the test automation.
- Automation is very useful and can be reused from release to release as long as good planning and analysis of the application under test is done.

9. Future Plans

Plans to continue automation efforts initiated in our project might be summarized as follows:

- Related with hardware, more powerful PCs should be purchased and have them dedicated for running SilkTest regression tests only.
- Although the scenarios automated have been re-used from release to release, there is always room from improvement in terms of getting new methodologies to automate new or existent tests. Maybe using object oriented analysis.
- A long term future plan is to study the possibility on how to interface the test automation with other tools

10. Acknowledgments

The author wants to acknowledge Teh-Hsin Wu for supporting the automation efforts in our project and supporting the presentation of this paper. Also wants to acknowledge the support and cooperation of co-worker and co-author Andrew Van Kraanen. The author also wants to acknowledge Tushar Sha and Ralph Catron for sharing information related with test automation tools.

Approaches to Testing Componentization in the Windows® CE

Patrick Copeland
Quality Assurance Manager
Windows CE Product Unit
Microsoft Corporation

Agenda

- What's Componentization?
- Component Model
- Our Approach

Part 1: What's Componentization?

- Overview
- Componentizable Modules
- Creating Custom Configurations
- Build Basics
- Samples

Overview

- Allows OS to adapt to different devices
- OS divided into Modules (EXE/DLL)
- Module can be divided into many components
- A component is a LIB file.
- Allows replacing of components
- Enables OEM's to make the tradeoffs

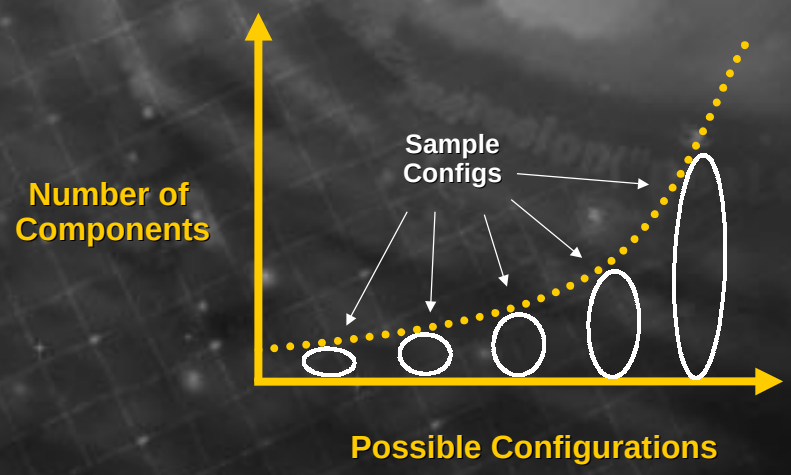
Componentizable Modules

- GWE (35 components)
  - Messages, GDI, Window Manager, Dialog Manager, Message Beeps, etc.
- File System (7 components)
  - ROM FS, RAM FS, Registry, Dbase, etc.
- CoreDLL (10 components)
  - Kernel/Win32 thunks, Serial API, etc.

Creating Custom Configurations

- Two Stage Process:
  - SYSGEN
    - compiles system modules and filter headers
    - creates a collection of about 150 LIBs that contain 1,500 functions
  - BUILD
    - creates one platform **configuration** or binary that is ROMable from the user selected components

Build Basics



Samples

- Minkern: Kernel + Filesystem
  - No Display
  - 150K -> 250K
- MinComm: + Comm stacks + COM
  - Focus only on Networking
  - 250K -> 500K
- MinGdi: + User input + GDI
  - 350K ->750K
- ...

Part Two: Component Model Basics

- Component Definition
- Object Oriented Ideals
- Reality

Component Definition

At this basic level, similar to the desktop DLL model.
Except for…

Exports

"let the world know what functions are supported"

Component
A

Imports

"tell the world what a component needs to run"

New Import Types:

•Hard – absolutely required

•Soft – only required if used explicitly

Component Definition

Component
A

```
// Code from Component A
F1() {
        H2();
        H3();
        #soft
        S4();
        #soft_end
}
F6() {
        H2();
}
```

An example…

Exports

Imports

Hard - F1 absolutely requires H2 and H3, and F6 absolutely requires H2.

Soft - (F1) requires (S4) as a soft dependency. If A is included as a component, we don't need to resolve S4. F1 will gracefully handle the error.

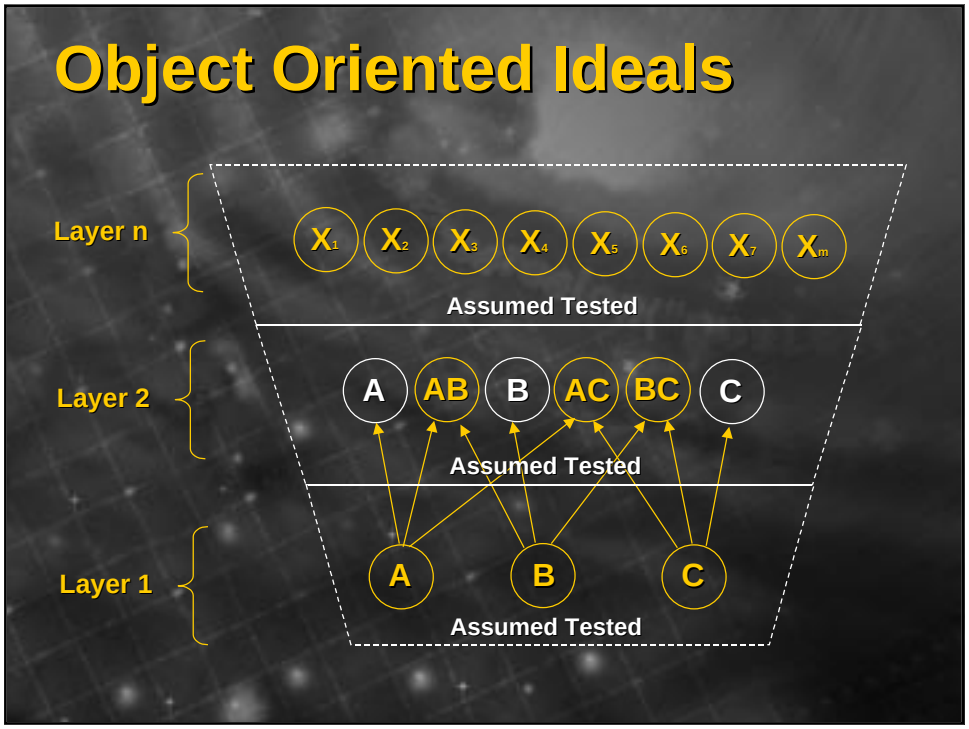
Component Connections

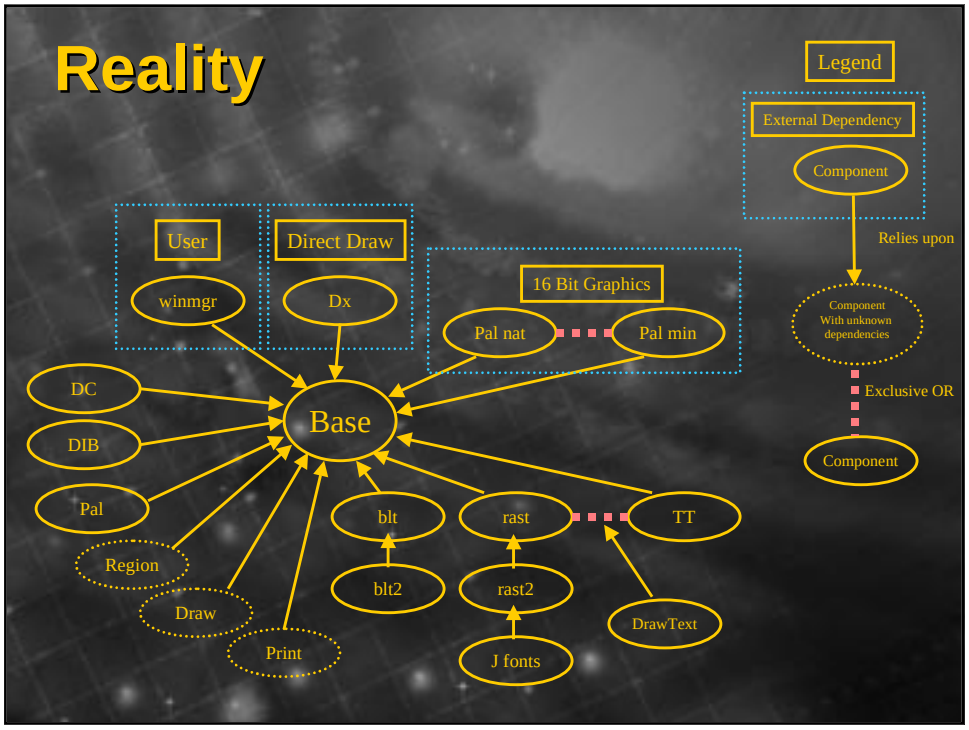
Component
A

Direct
Function calls
Resources

Component
B

Indirect
Memory
Critical Sections
Shared Resources

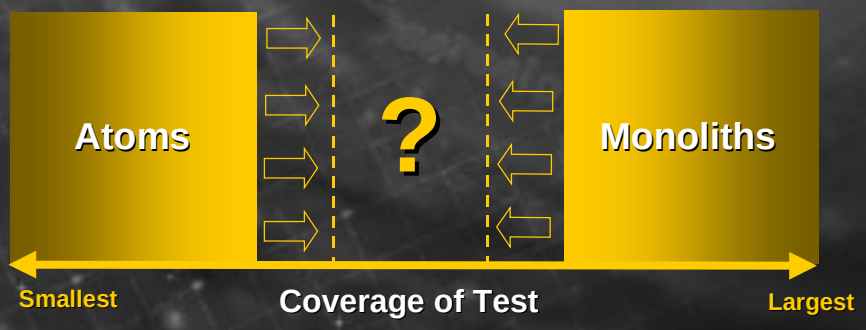

Object Oriented Ideals
Layer n
X1 X2 X3 X4 X5 X6 X7 Xm
Assumed Tested
Layer 2
A AB B AC BC C
Assumed Tested
Layer 1
A B C
Assumed Tested

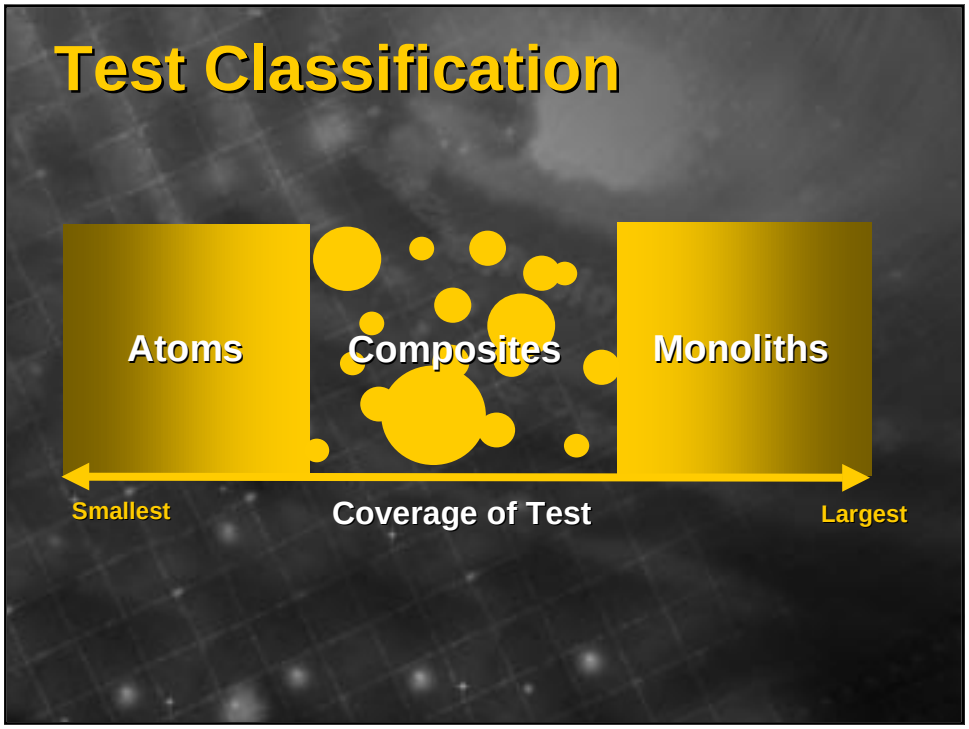

Reality
Legend
External Dependency
Component
Relies upon
Component
With unknown
dependencies
Exclusive OR
Component
User
winmgr
Direct Draw
Dx
16 Bit Graphics
Pal nat
Pal min
DC
DIB
Pal
Base
Region
Draw
Print
blt
blt2
rast
rast2
J fonts
TT
DrawText

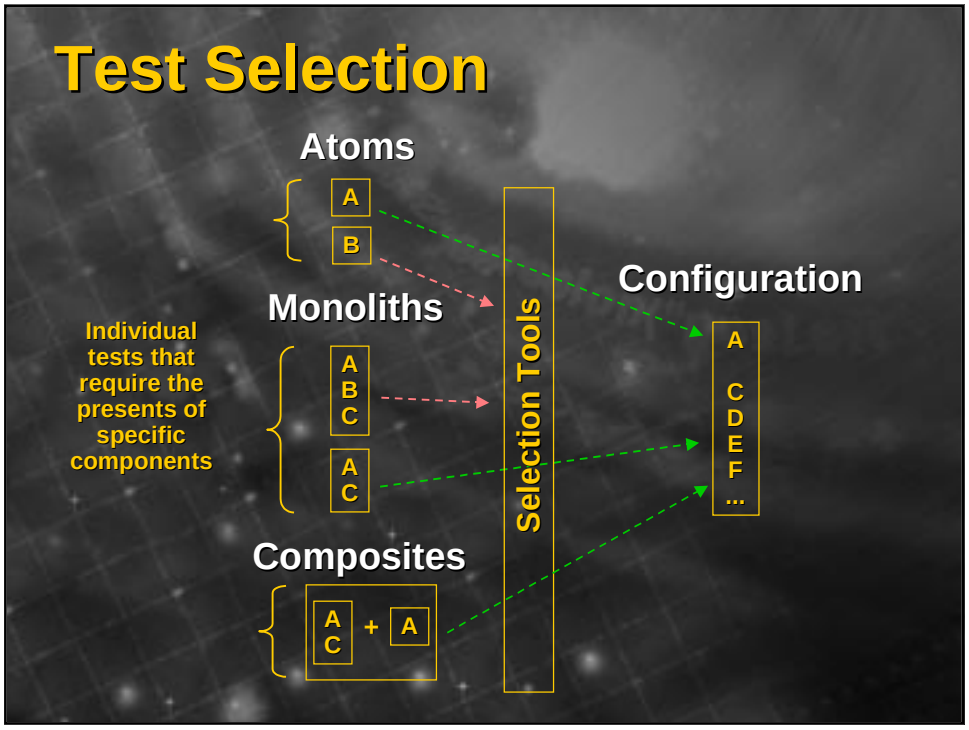
Part Three: Our Approach

- Test Classification
- Component Connections
- Test Code Groups

Test Classification




Test Classification
Atoms
Composites
Monoliths
Smallest
Coverage of Test
Largest


Test Selection
Atoms
A
B
Monoliths
Individual tests that require the presents of specific components
A B C
A C
Composites
A C + A
Selection Tools
Configuration
A
C D E F ...

Microsoft®
Start

Powered by Microsoft Windows CE

Software Quality Week 1999

Testing Windows CE 3.0 Real-Time Kernel

Sergio Cherskov
Windows CE Base OS QA Lead

Testing Windows CE 3.0 Real-Time Kernel

- Windows CE 3.0 Design Goals
- Interrupt Processing in Windows CE 3.0
- Interrupt Response Latency
- Measuring Real-Time Performance

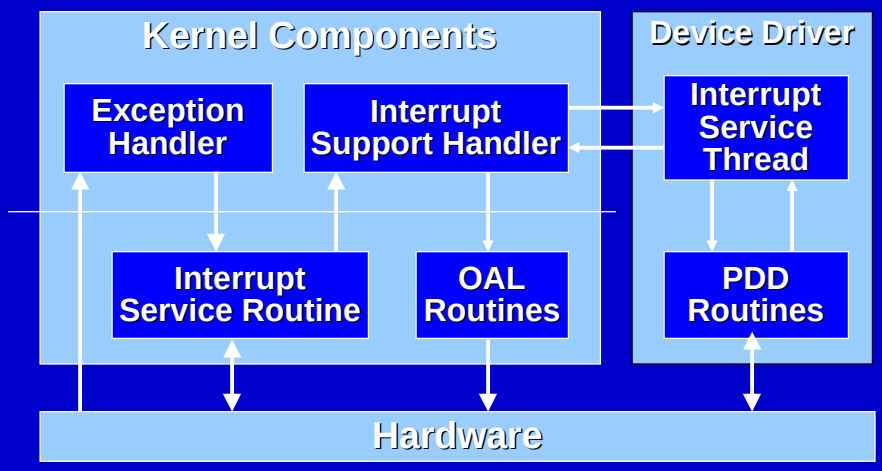
Windows CE 3.0 Design Goals

- Improve on real-time capabilities
- Nested interrupts
- More priority levels
- Finer (1ms) timing/scheduling resolution
- Priority inversion handling
- Bounded interrupt latencies

Interrupt Processing in Windows CE 3.0

- Interrupt Service Routines (ISR)
  - implemented in the OAL by OEM
  - fast, just return interrupt identifier
- Interrupt Service Threads (IST)
  - blocks on event, signaled by kernel
  - workhorse, does I/O, processes data
  - signals end of interrupt to the OS

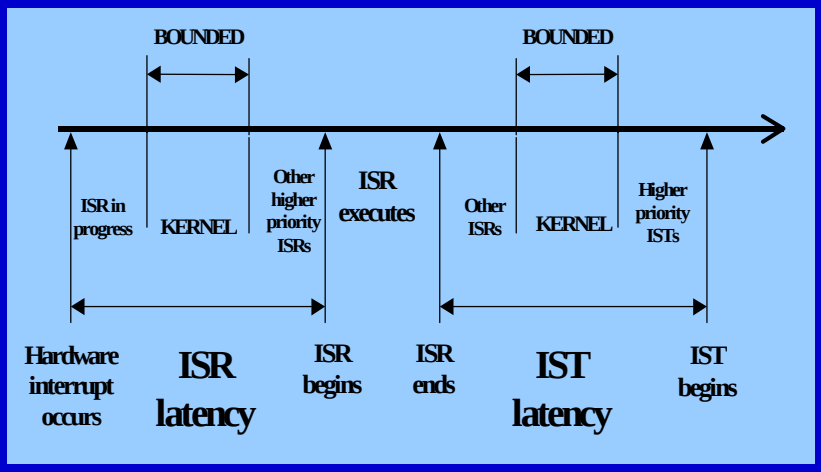
Interrupt Processing in Windows CE 3.0



Interrupt Response Latency

- ISR Latency
  - ISR in progress finish time + kernel ISR time + sum of other hi-pri ISR execution times
  - time spent in kernel ISR processing is bounded
- IST Latency
  - all ISRs in progress + kernel IST time + all higher priority ISTs in progress
  - time spent in kernel IST processing is bounded

Interrupt Response Latency



Measuring Real-Time Performance

- Kernel Functionality Testing
- IntrTime -- Interrupt Response Latency
- MRTT -- External Interrupt Response
- CeBench -- Scheduler Performance

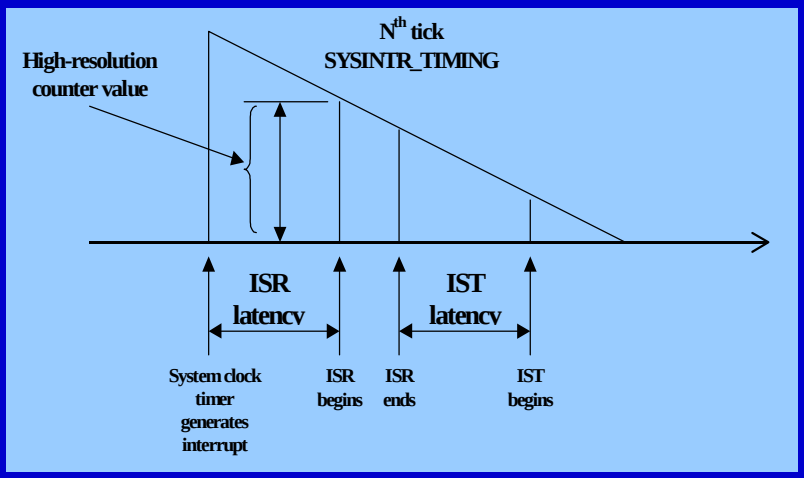
Kernel Functionality Testing

- Concentrated on functionality versus real-time performance
- Also tests compilers, linkers, loaders, processor families, hardware platforms
- Build Verification Tests
- Full Win32 kernel API functionality testing
- Windows CE Core OS Lab

IntrTime -- Measuring Interrupt Response Latency

- Runs on all hardware platforms
- Requires modifications to OAL code
- Works by stealing every n-th tick from system timer
- run-time parameters
  - cache sync,
  - ISR rate and number of interrupts captured,
  - idle threads, output redirection to file

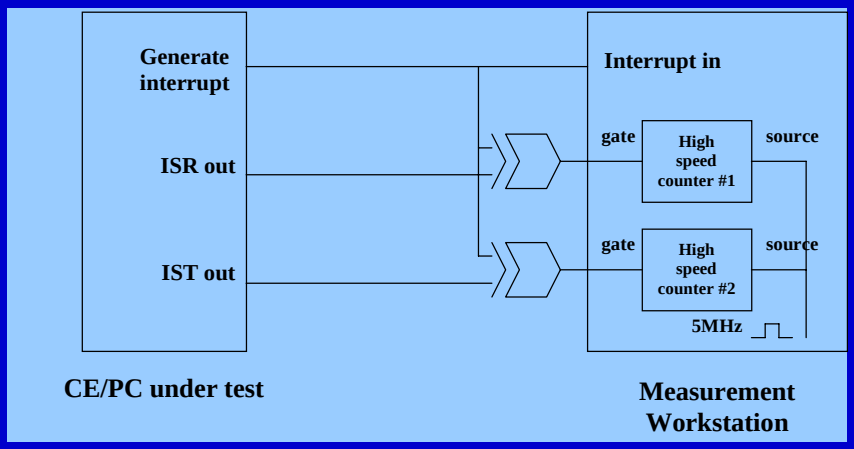
IntrTime -- Measuring Interrupt Response Latency



MRTT -- Measuring External Interrupt Response

- Measurements are done external to the hardware under test
- National Instruments PC-TIO-10 card
  - 2 sets of 5 timers, 1 hi resolution (5MHz)
  - 2 sets of 8 I/O pins, 1 can generate interrupt
- Same card used on both sides
- Improvements (PCI card for HARP testing)

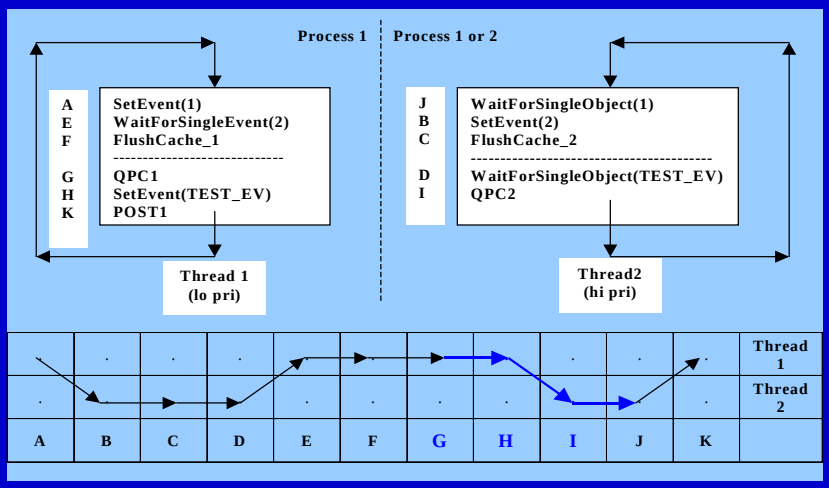
MRTT -- Measuring External Interrupt Response



CeBench -- Measuring Scheduler Performance

- No OAL changes, runs on all HW platforms
- Uses QueryPerformanceCounter()
- Critical Sections, Interlocked API's
- Event, Mutex, Semaphore, Voluntary Yield
- PSL API call overhead
- Cache Sync, intra & inter process
- writing markers for hardware verification

CeBench -- Measuring Scheduler Performance



Powered by
Microsoft
Windows CE

Testing Windows CE 3.0 Real-Time Kernel

Sergio Cherskov
Windows CE Base OS QA Lead

Copyright © 1999 Microsoft, All rights reserved.

Table of contents

Windows CE 3.0 Design Goals 3
Interrupt processing in Windows CE 3.0 3
Measuring Real-Time performance 5
Kernel functionality testing 5
Interrupt Response Latency Measurements 6
External Interrupt Response Measurements 7
Scheduler Performance Timing 9

Table of figures

Figure 1: Windows CE Interrupt Processing 3
Figure 2: Windows CE Interrupt Processing Timeline 4
Figure 3: IntrTime Timing Diagram 6
Figure 4: IntrTime command line parameters 7
Figure 5: External interrupt latency setup 8
Figure 6: CeBench command line parameters 9
Figure 7: CeBench example test output 10
Figure 8: CeBench example test execution path 10

Copyright © 1999 Microsoft, All rights reserved.

Abstract:

Presenting the testing procedures and goals that were used to test and verify real-time kernel during development of Microsoft Windows CE 3.0 operating system. Presentation details the methods used to test and validate kernel real-time performance.

Windows CE 3.0 Design Goals

Windows CE 3.0 is the latest Microsoft operating system, destined for the embedded 32-bit market, encompassing a wide range of possible devices. It runs on a number of processors from different families (currently: ARM, MIPS, PowerPC, SH, x86), and is scalable from a bare-bones kernel-only configuration, to a full-fledged portable companion OS. Windows CE 3.0 is fully multithreaded, supporting up to 32 processes, protected paged virtual memory, 256 priority levels, using round-robin preemptive scheduling.

One of the design goals for the Microsoft Windows CE 3.0 operating system was improving on real-time capabilities and approaching the standard response times as seen in the embedded microprocessor industry. This includes support for the nested interrupts, expanded number of thread priorities, better control of timing and scheduling, handling priority inversion, and guaranteed bounded interrupt latencies.

Interrupt processing in Windows CE 3.0

The modularity and layered approach of Windows CE 3.0 operating system can be clearly observed during the processing of the hardware interrupt. Real time designs use hardware interrupts as a way to ensure that external events are rapidly accepted and processed by the operating system. Within Windows CE, using layered approach, a thin OEM Adaptation Layer (OAL) and kernel share the responsibility of interrupt delivery and event dispatching to the rest of the system.

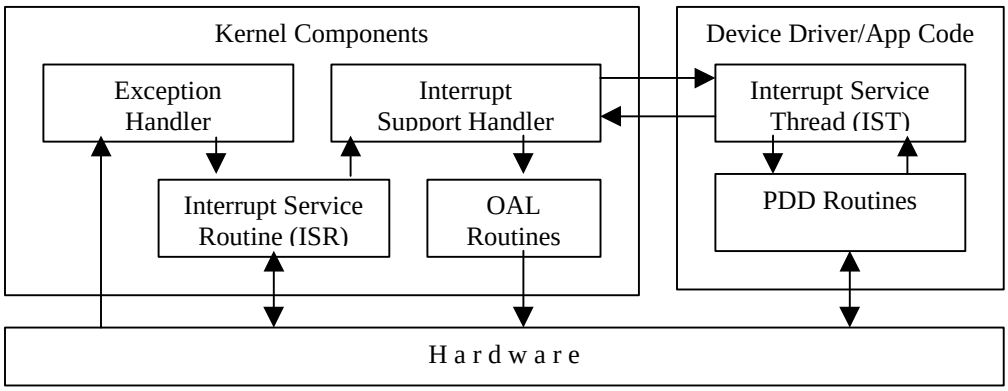


Figure 1: Windows CE Interrupt Processing

Copyright © 1999 Microsoft, All rights reserved.

The processing is split into two distinct parts: an interrupt service routine (ISR) and an interrupt service thread (IST). Each hardware interrupt request line (IRQ) is mapped to corresponding ISR. When interrupts are enabled and a hardware interrupt occurs, the kernel will merely look up registered ISR and transfer further execution to it. The ISR itself, the kernel-mode portion of interrupt processing, is kept as short as possible.

ISR is expected to return either the generic interrupt identifier value SYSINTR_NOP that indicates no further processing, or a specific value that directs the kernel to set an event that is associated with the interrupt service thread (IST). If an IST was indeed created and associated with that event, the kernel now schedules the thread. It is the IST code where the bulk of the processing specific to the device that generated the hardware interrupt is expected to take place. Upon finishing, IST calls InterruptDone() function that performs any hardware actions necessary to enable the next interrupt from the device.

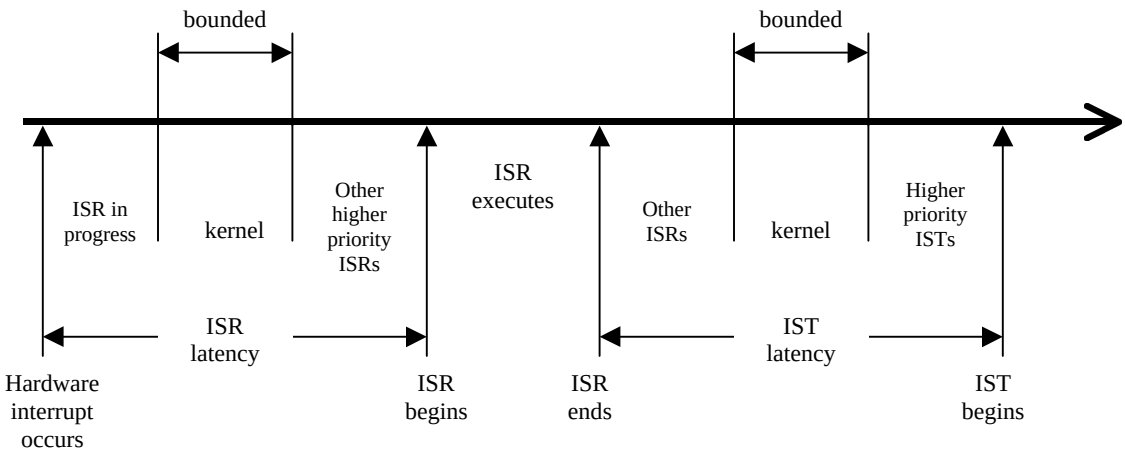


Figure 2: Windows CE Interrupt Processing Timeline

Interrupt Service Routine (ISR) latency is defined as the amount of time that elapses from the moment that an external hardware interrupt occurs and is recognized by the processor, to the moment that the kernel-mode routine that services it begins executing. This time is comprised of several parts: a) time spent in the kernel, b) the duration of an ISR in progress at the time the interrupt arrives and c) the sum of the durations of all higher priority ISRs that arrive before this ISR starts.

Similarly, Interrupt Service Thread (IST) latency is defined as the amount of time that elapses from the moment that the ISR code returns to kernel, until the user-mode thread that actually processes device data is scheduled and begins executing. Since the IST's themselves can be interrupted by one or more of same or higher priority interrupts, we try to exclude this time from the overall IST latency, which can range from 0 to the duration of the longest ISR in the system. This is also true of higher priority ISTs, which will then be scheduled before this one. The rest of the time is spent in the kernel which has to finish any Kernel Call (Kcall) that may have been in progress when signaling ISR occurred, schedule IST, and switch CPU context to the proper process space that contains IST code.

Windows CE 3.0 significant improvement over the previous versions is bounded kernel interrupt service processing times for threads locked in memory, where no paging of virtual memory occurs. As the ISR

Copyright © 1999 Microsoft, All rights reserved.

code is linked in the kernel at the time an image is built, this is not a problem. For the IST, locking of the thread code in memory is easily accomplished by moving and compiling the IST into a separate dynamic-load library (DLL) file and using the LoadDriver() API. Unlike LoadLibrary() API that merely loads the DLL into memory, LoadDriver() API also locks the code pages.

It is important to note that the Original Equipment Manufacturers (OEMs) are creating a custom version of Windows CE operating system for each specific hardware platform and operating environment. While Microsoft has shortened the time spent during kernel processing it is ultimately up to OEM to take advantage of the given platform and optimize the system design for the best interrupt response times.

Measuring Real-Time performance

In order to perform an accurate assessment of Windows CE 3.0 kernel updates, test and measure real-time performance, and validate kernel functionality, a number of test suites have been developed, using quite a different approaches. At the very base is the kernel functionality testing, which we will just briefly touch in this presentation and concentrate on real-time measurements.

Kernel functionality testing

The suite of tests that will be discussed later will concentrate more on the real-time performance aspect of the kernel, and timing of the various events that take place in such multi-tasked environment. However, the kernel exposes a number of Win32 APIs that also need to be verified and tested. Priority levels, synchronization objects, thread and process creation/deletion have all to be confirmed and verified to work for all possible configurations of the Windows CE system, on all processors. In addition, every processor carries with it a set of tools – compilers, linkers, loaders – that need to be taken into account during testing. The wide variety of hardware platforms that Windows CE runs on, makes this task even a bigger challenge.

Kernel build verification suite of tests goes through the list of APIs and kernel features and assures that the basic functionality of the kernel is intact, regardless of the changes that are constantly done underneath, be it code revisions, new compilers, new tools or new hardware reference platforms.

Full kernel functionality testing is used to exercise all of kernel API's and validate the correctness of the kernel implementation without regard as to the real-time components of such operations. This would involve for example calling an API to, say, create thread, and verifying that the thread is indeed created, that is runs on given priority, that the parameter passed during the API call is forwarded to the thread, etc. The tests do not necessarily care how fast the task was done or if the task was time bounded. The real-time aspects are evaluated with a separate set of tools.

This testing is quite involved with a number of test suites being run daily in Microsoft's Windows CE Core OS automated testing Lab. New tests are still being created and added to the rooster, as new features and kernel modifications take place. For the example of tests and test harnesses used to run them, please take a look at Device Drivers Test Kit (DDTK) that routinely accompanies the MS Platform Builder.

MS Platform Builder is a main vehicle for distribution of Windows CE, containing all the necessary tools, libraries and object files to successfully create a customized version of Windows CE to fit the requirements of a given hardware platform. The tools include cross-compilers, assemblers, remote debugging tools, code generators, operating system loaders, ROM image building tools, sample device drivers, sample application code and documentation.

Copyright © 1999 Microsoft, All rights reserved.

Interrupt Response Latency Measurements

The measurements of ISR and IST latencies described earlier have been combined in IntrTime test tool that is freely available in source code and also distributed with MS Platform Builder. The measurements are done using the system clock timer in an effort to make it available to all hardware platforms that Windows CE runs on, since some of them do not provide for a separate available unused timer.

Under normal circumstances, system clock interrupts the kernel in regular intervals. The associated system timer ISR then processes the tick and returns either SYSINTR_NOP directing kernel to just ignore the tick, or SYSINTR_RESCHED to wake up scheduler.

IntrTime test tool does the latency measurements by 'stealing' every n^{th} tick of the system clock (defaults to every 5th system tick) and signaling a special SYSINTR_TIMING interrupt identifier event. The IntrTime application's main thread waits on the SYSINTR_TIMING interrupt event, thus becoming the IST. ISR and IST measurements are derived from time-stamps i.e. the counter values of the high-resolution timer since the last system tick.

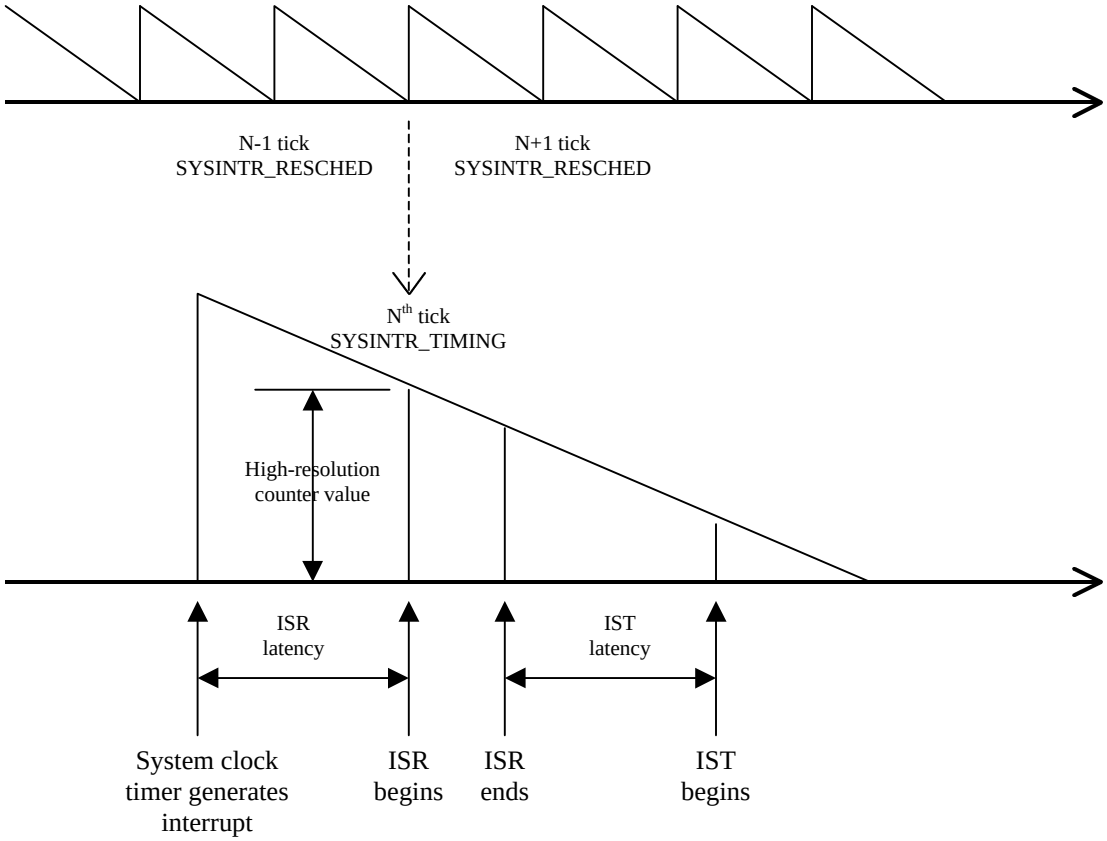


Figure 3: IntrTime Timing Diagram

In order to provide these values, a modification of the OAL is required and OEM exports a special function PerfCountSinceTick() via the KernelIoControl() IOCTL_HAL_INTRTIME. The IOCTL call returns a

Copyright © 1999 Microsoft, All rights reserved.

pointer to the function that will be called (in Kmode) directly from the application to minimize overhead. This same IOCTL call is also used to start and stop system tick from returning a SYSINTR_TIMING interrupt identifier and to return the stored start and end of ISR response times.

Because it requires special modifications to the OAL only, but not the kernel, IntrTime could be easily adapted and should run on any of the sample platforms and configs. As a special note, it is recommended that the OEMIdle function be set not to reprogram the timer during IntrTime measurements.

```
Usage: intrtime [ options ]
Options:
  -p num   Priority of the IST (default 0 ; highest)
  -ni      no idle priority thread (default: idle priority thread
spins)
  -ncs     no CacheSync() call (default: flush cache after each
interrupt)
  -i0      no idle thread (same as -ni)
  -i1      Run idle thread type 1
  -i2      Run idle thread type 2
  -i3      Run idle thread type 3
  -i4      Run idle thread type 4
  -sp      Starts a secondary process
  -t num   SYSINTR_TIMING interval (default 5)
  -n num   number of interrupts (default 10)
  -all     print all data (default: print summary only)
  -o file  output to file (default: output to debug)
```

Figure 4: IntrTime command line parameters

The IntrTime command line parameters allow for introduction of number of variations where the IST can be set to run on various priorities, cache flushed after each interrupt or not, change ISR rate and number of interrupts captured, or print/output to file the collected results. IntrTime can also create one or more idle threads running in the background, which affect the IST latencies by allowing the kernel to be in a non-preemptible kernel call that must finish before the IST is run.
.

External Interrupt Response Measurements

For quick assessment of the day-to-day real-time performance of the system, interrupt timing analysis tool (IntrTime.exe) is quite enough to determine the ISR and IST interrupt latencies. This convenient method works across all supported processors but relies on the timer on the device itself that may affect the measurements.

Thus, a more elaborate setup has been used to accurately measure ISR and IST latencies. Two machines have been set up: a workstation that generates an external interrupt and measures the time it takes to receive acknowledgements from ISR and IST routines, and device-under-test that receives the external interrupt and toggles output lines when ISR and IST routines are reached. Testing is performed under various stress levels, running anywhere from one to hundreds of threads of varying priorities on the test device.

The Windows NT 4.0 workstation, equipped with a National Instruments PC-TIO-10 digital I/O timer/counter card, is used to generate interrupts and time responses, and a CE/PC equipped with an identical card to respond to those interrupts. The NT software takes advantage of the driver library supplied by National Instruments, while the CE software is written completely by Microsoft.

Copyright © 1999 Microsoft, All rights reserved.

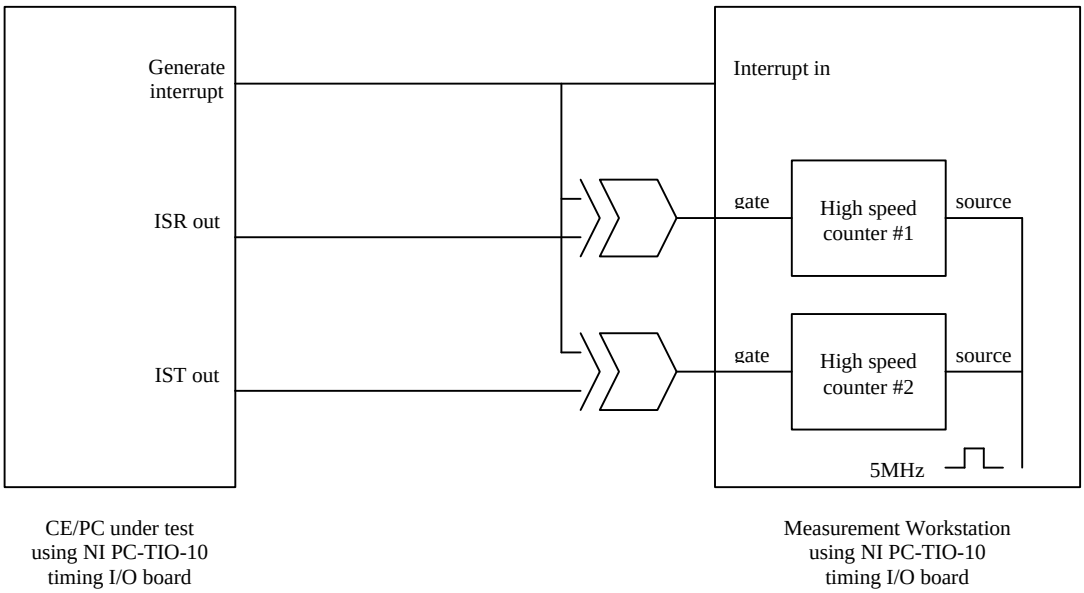


Figure 5: External interrupt latency setup

The theory of operation is quite simple: the PC-TIO-10 card has two sets of 5 timers, each set containing one timer that provides 200ns resolution, rest of them being 1us granularity. In addition, it contains two sets of 8 digital I/O lines, again each set providing one line that can be used to interrupt on edge or level triggering. One output line from the NT machine is wired both to the external interrupt pin of the CE/PC and back to the timers on the NT's card. As NT machine asserts one of its output lines, it generates an interrupt on CE/PC and starts ISR and IST timers on NT card. The ISR on the CE/PC merely acknowledges the receipt of the interrupt by asserting an output line on the card (which stops the ISR timer on NT side), and notifies the kernel to schedule the IST. When the IST starts running, it asserts a different output line, stopping the second timer on the NT side.

At this point, the NT machine can read the values on the timer counters, to determine the intervals between an interrupt being generated and CE/PC's responses. As soon as the NT machine has read the counter values, it issues another interrupt which CE/PC uses to bring all output lines to the 'standby' state, ready for another cycle.

Preliminary results gathered using above measurements confirm the accuracy of the intrtime testing results. Currently under development are somewhat less stringent versions of the setup above that allow for the greater utilization of the available Lab hardware. In one version, a workstation is replaced with a CE/PC and the intermediate circuitry duplicated. This allows us to have automated Lab setup utilizing both CE/PC's as active participants in measurements, generating and responding to interrupts at the same time. Also being investigated is the single CE/PC solution, where the lines are looped back to the CE/PC itself.

As mentioned before, Windows CE 3.0 supports not only x86 family of processors, but also a number of other processor families. The kernel development and testing of such processors is done on Hitachi D9000 reference platform (commonly called ODO) which does not provide either ISA or PCI interface. Work is also under way to port the external hardware verification test to D9000 based processors, by utilizing the available I/O lines of one of its serial ports (DCD as interrupt source, RTS and DTR for ISR/IST response).

Long-term, the replacement is sought for the current National Instruments ISA based card used for measurements, as Microsoft's next generation Windows CE Hardware Reference Platform (HARP) is PCI based. The greater variety of cards used, on several development platforms would further reinforce and

Copyright © 1999 Microsoft, All rights reserved.

verify the validity of Windows CE 3.0 real-time kernel implementation. For further information on Microsoft Hardware Reference platforms please visit our Web site at: http://www.microsoft.com/windowsce/Embedded/resources/refplat/default.asp .

Scheduler Performance Timing

Scheduler performance timing tests (CeBench.exe) are focused on measuring the time required to perform basic kernel operations (see below).

```
Usage: cebench [ options ]
Options:
  -all     Run all tests (default: run only those specified by -t option)
  -t num   ID of test to run (need separate -t for each test)
  -n num   Number of samples per test (default = 100)
  -m addr  Virtual address to write marker values to (default = <none>)
  -list    List test ID's with descriptions
  -v       Verbose : show all measurements
  -o file  Output to CSV file (default: output only to debug)

CeBench -list
TestId  0 : CriticalSections
TestId  1 : Event set-wakeup
TestId  2 : Semaphore release-acquire
TestId  3 : Mutex
TestId  4 : Voluntary yield
TestId  5 : PSL API call overhead
TestId  6 : Interlocked API's (decrement, increment, testexchange, exchange)
```

Figure 6: CeBench command line parameters

The bulk of the testing consists of timing the kernel synchronization objects: critical section, event, semaphore, mutex and voluntary yield. It also contains test measurements of PSL API call overhead.

As with IntrTime measurements, QueryPerformanceCounter() (QPC) function call has been used to obtain timing information. In addition, at every timing point where QPC() is invoked, user can specify that a write of specific “marker” value is performed to the virtual address. This hardware verification feature is enabled by providing the virtual address on the command line at the time when CeBench is started. Markers written at the virtual address can then be monitored by analyzer, independently timed by external device and results used to double-check the QPC() timing accuracy. The setup similar to the external measurements of interrupt latency can be used for this purpose.

Please note that QPC() function call to get time stamps does not come for free. The frequency of the counter at particular platform and the overhead of calling this function has to be taken into account when analyzing the results. Care has been exercised to provide for proper exclusions of the measuring overhead in the final timing numbers. The QPC() call is looped for a number of iterations before every test and the average is subtracted from the final result.

In cases where the operation takes a very small time to complete, the overhead of the QPC() function call becomes significant. In those cases, the operation is looped for a fixed number of iterations per sample (IPS), clear indication of which is provided with every test, and the result is then averaged. A special sub-marker value is provided for these cases if hardware verification was enabled. A side effect of this looping is that the cache can not be flushed between each iteration of the operation. For other tests where the IPS is equal to 1, test is run twice, once with and once without cache flush for each iteration.

Copyright © 1999 Microsoft, All rights reserved.

```
=================================================================
|  1.00  |  IP =  NO  |  CS =  NO  |       1 IPS
-----------------------------------------------------------------
Event intra-process :
Time from SetEvent() in one thread to a blocked WaitForSingleObject()
waking in another thread in the same process.
-----------------------------------------------------------------
|  Max Time =         10.057 us
|  Min Time =          5.867 us
|  Avg Time =          6.823 us
=================================================================
```

Figure 7: CeBench example test output

In the example of the test number 1.00 whose output is shown above, the operation was timing of the intra-process event synchronization object. The number of Iterations Per Sample (IPS) was one; Cache Sync (CS) was not done after each run; Inter-Process status (IP) shows that second process was not used (both threads were in the same process). The maximum, minimum and average results for 100 operations (default, if nothing is specified on the command line) are given in microseconds.

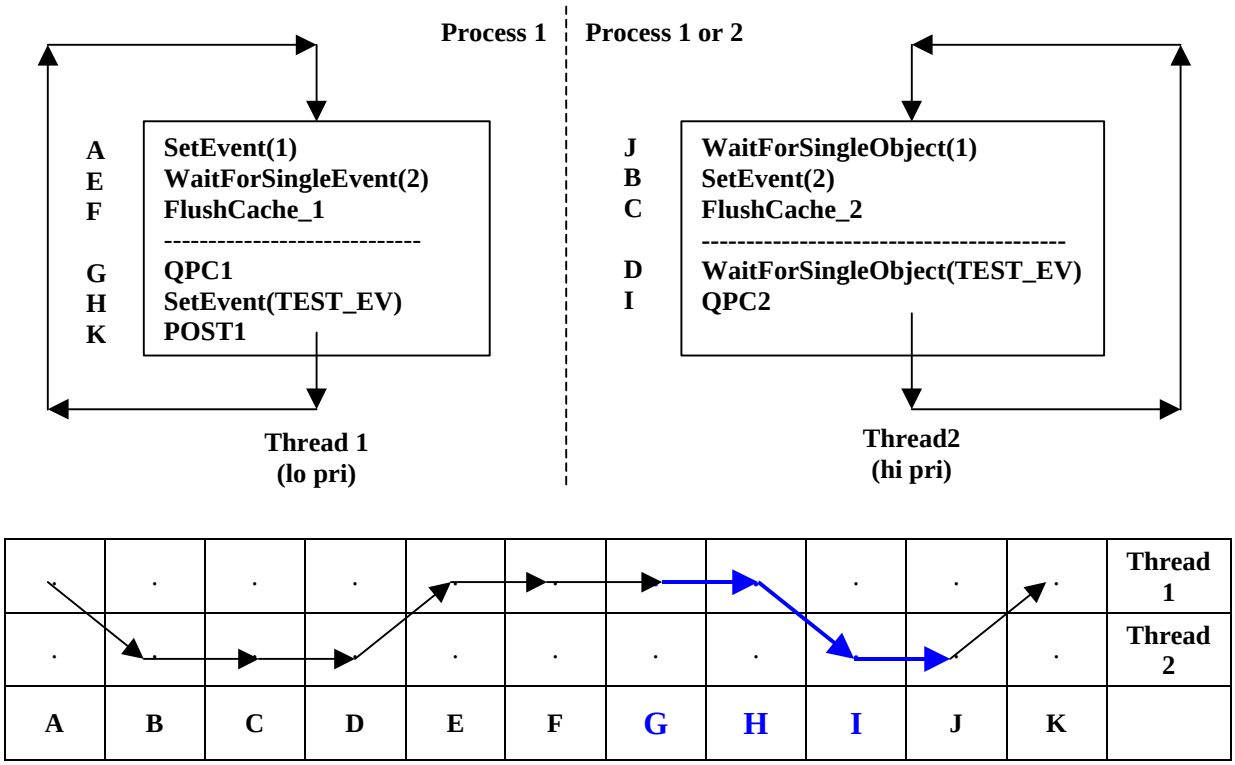


Figure 8: CeBench example test execution path

The basic suite of tests and the overall layout of the CeBench program allow for easy additions of new testing cases and measurements, augmenting the implementation for particular kernel function that might be of special interest to various customers.

Copyright © 1999 Microsoft, All rights reserved.

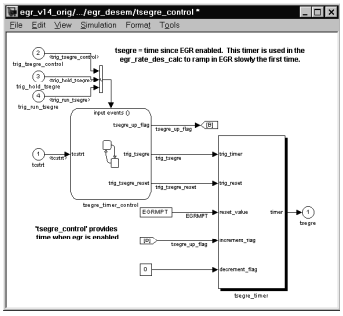
An Automated Inspection Tool For A Graphical Specification and Programming Language

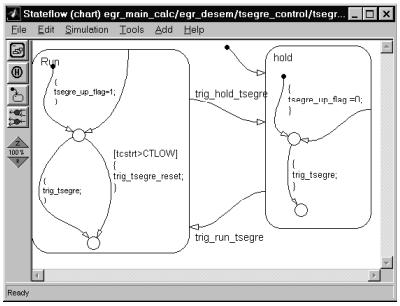
Steve Toeppe
Scott Ranville
Ford Research Laboratory
Ford Motor Company

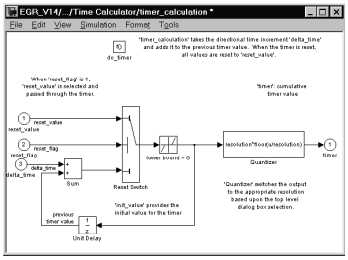
Graphical Programming Tool Description

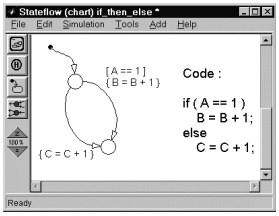
- Matlab Simulink and Stateflow Graphical Specification and Analysis Tool Utilized
- Stateflow is Similar to Unified Modeling Language Harel Statecharts
- Simulink is a Block Diagram Modeling Tool Similar to Labview, Matrix$_x$, Beacon, MGA and other Control System Modeling Tools
- Graphical Languages Have Syntax Rules With Subtle Impacts If Improperly Used
- Syntax Checking Tools Are Needed

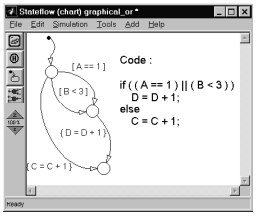
Graphical Modeling Examples







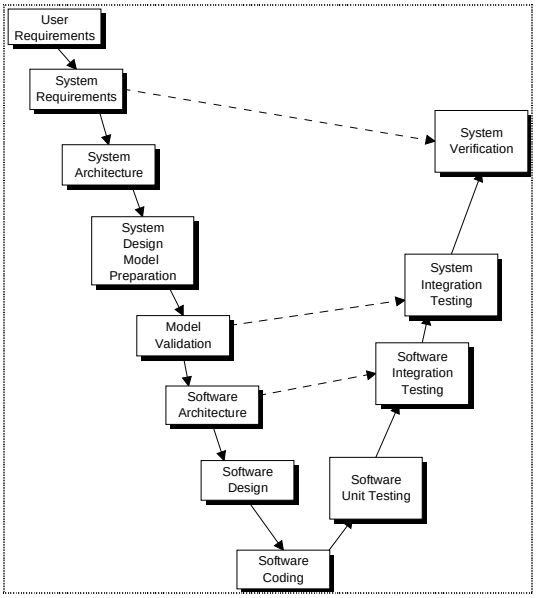




Process Description

- Iterative Development Process Intended To Support Maintenance and New Feature Development
- Experimental Stages Critical For Algorithm Validation
- Physical Dynamics Must Be Considered
- Simulation Used For Algorithm Validation and Implementation Verification
- Software Verification Expected Results Based Upon Algorithm Validation Experiments
- Graphical Models Used For User Documentation

Ford System V Process



Style Guideline

- Context Diagram Interfaces and Scheduling
- Subsystem Hierarchy and Interconnections
- Subsystem Notation Usage
- Scheduling Syntax
- Signal Connectivity
- Individual Block Usage
- Stateflow Diagram Syntax
- Software Design Information
- Documentation
- Naming Conventions

General Test Categories

Context Subsystem Name
Context Diagram First Subsystem Name
Context Diagram Input and Output Wiring
Context Diagram Scheduling Inputs
Execution Context Diagram Format
Execution Context Subsystem/Trigger Names
Execution Context Trigger
Execution Context Does Not Require Matlab® Scheduler
Simulink® Block Consistency Checks
Port To Signal Compare
Initialization of Data Store, Delay Blocks, etc
Verify Sampled Blocks Use Inherited Sample Times
Check For Appropriate Usage of Multiport Switch Block and Standard Switch Block
Check For Non Stateflow™ diagram Generated Switch Controls
Check For Constant Expressions in Dialog Boxes
Subsystem Consistency Checks
Properly Triggered Subsystem
Subsystem Held/Not Held Output
Unwired Ports and Subsystem Ports
Identify Any Non Masked Subsystems
Verify That Masked Subsystems Have Something That Will Transform Inputs To Outputs
Check For Proper Masked Subsystem Entries
Check For Labeled Blocks Instead of Masked Blocks
Verify Correct Prefix on Subsystem Names
Detect Virtual Subsystem With Only Annotations or Text Mask
Data Consistency Checks
Data Dictionary Type vs Usage
Proper View of Calibration Parameters
Verify Calibration Parameters in Model vs Data Dictionary (Check Dimensions)
Check for Unnecessary Data Stores
Trigger/Event Consistency Checks
Verify Triggers Are Function Call Triggers (handle exceptions)
Verify do_<P_SPEC_NAME> and trig_<HIERARCHY_NAME> Are Correctly Named
Unique CSpec Names and Event Signal Names Within Feature
Verify do_<> ends in a PSpec
Verify trig_<> ends in a Stateflow™ diagram
Unique Application Signal Names Within Feature
Signal Wiring Consistency Checks
Check if Tags Are Local
Check For Matching GoTo and From Tags
Check Signal Name Propagation
Check For Unnecessary Signal Names
Check Switch Variable Names Go To a Switch
Merge Block Rules
Vector Signal Propagation and Naming
Vector Signal Termination and Sources
Verify Vector Signal is Named With Attribute

State Chart Consistency Checks
Check Each State Chart For A Default Path For Flow Control
Check For Unguarded Path As Default and Invocation of SF Clockwise Rule
Verify That State Chart Is Triggered With A Function Call
Check For Flow Logic Classifications Into Acceptable Patterns
Check For Recursion SF Constructs
Check For Action Only At End of Classified Pattern
Check For Valid Compound Conditionals
Check For Proper Types For "==".
Check While/For Loops For Termination
Check For Unstructured Loops
Check for Valid Multiway Branch
Delete Check
Verify Events Are Local or Function Calls
Verify Broadcast Rules
Verify State Hierarchy Rules
Verify Stateflow™ Event Wiring Rules
Verify Stateflow™ Input and Output Data Wiring Rules
Check For Write To Workspace From SF
Check For Data Transformation in Stateflow™ diagrams
Check For ML Operation in Stateflow™ diagram
Check for SF Keywords
State Machine Data Flow Rules
Event Broadcast Check
Check Model For Unique Names When Not Considering Case
Verify Stateflow™ diagram Flow Diagrams Do Not Use Event "/".
Verify Stateflow™ diagram Does Not Invoke Any C Library Utilities or Use C Code
Software Design Consistency Checks
Check For Vectorization
Function name Rules
File Name Rules
Documentation Consistency Checks
Verify Correct Font Attributes
Check For Unneeded Drop Shadows in Annotations

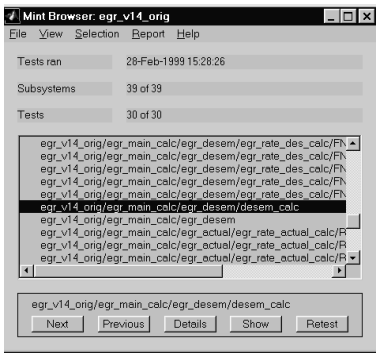
Consistency Checker Usage Scenario

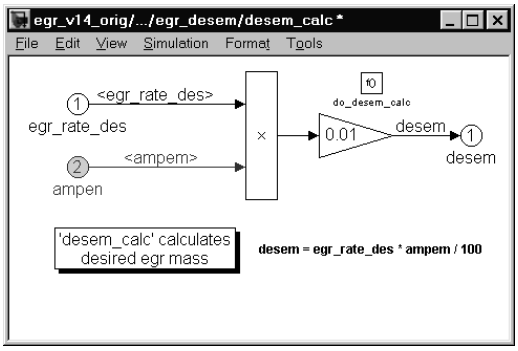
- Current Practice: Manual Inspection of Several Thousand Points of Review Per Model
- Future Practice: Automatic Inspection of 10 to 20 Thousand Points of Review Per Model
- Modeler Interactively Corrects All Anomalies
- Modeler Presents Anomaly Report at Inspection
- Anomaly Report is Put Under Configuration Management With Model, C Code and Test Vectors/Results

Tool Operation

- Extract Model Structural Connectivity and Information Content
- Analyze Structural Information to Verify Selected Style Guideline Conformance Checks
- Report Results Via Browser and Reports
- Rerun Selective Checks During Corrective Modeling
- Save/Restore Results and Analysis Session
- GUI Based Model Selection, Configuration and Filtering
- HTML Report Generation Provided

Anomaly Example





- Browser Indicates List Of All Anomalies Organized By Failure or Model Location
- Hot Links Permit Model To Be Opened and Anomaly To Be Highlighted

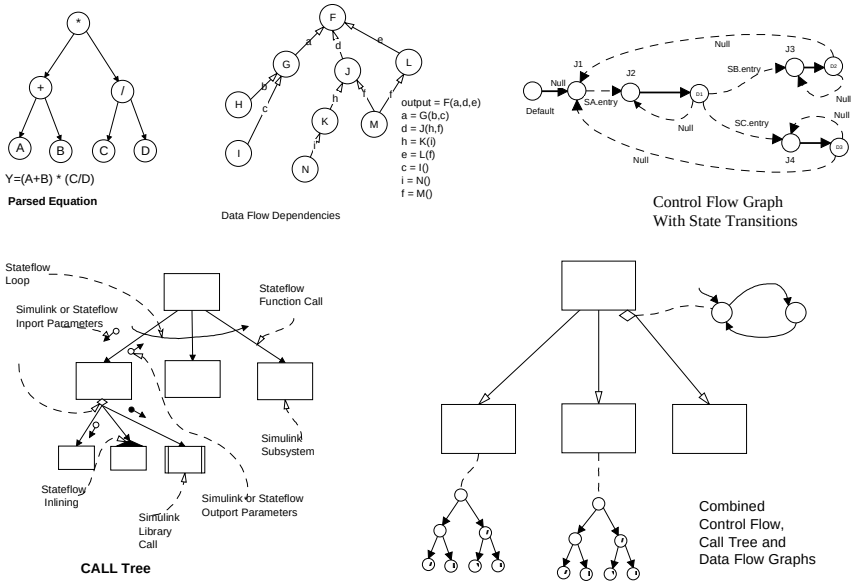
Analysis Methods

- Extract Data Flow and Control Connectivity By Traversing Directed Graphs Representation
- Establish Call Tree Relationships
- Conduct Control Flow Pattern Matching
- Detect and Analyze Loops
- Directed Graphs Can Be Modeled Within The Matlab Workspace
- Recursive m-Scripts Permit Graph Traversal
- Utilize Extensive Information Links Provided Via Matlab Handles For Graphical Objects, Data Elements and GUI Components

Analysis Methods

- Intermediate Representations Used To Manage Information For Analysis
- Multidimensional Linkages Between Representations Are Required To Correlate Data
- Directed Graphs and Search Methods Used
- Data Flow Connectivity Used For Data Dependencies
- Call Trees Used To Establish Execution Sequences Used For Tracing Enabled Data Flow
- Depth First Search Algorithms Used To Traverse Model and Directed Graphs

Intermediate Representations



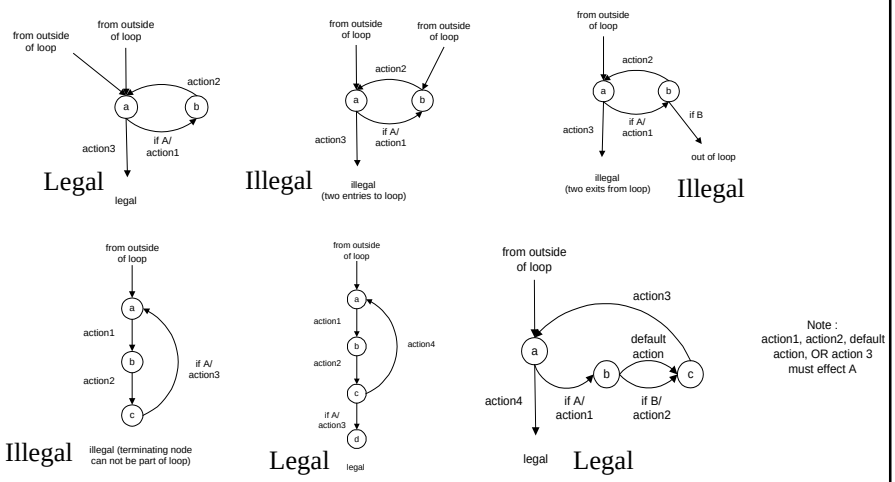
Anomaly Detection Methods

- Extract Intermediate Representation of Some Connectivity or Topology of Interest
- Associate the Relevant Model Information With the Intermediate Representation
- Detect Valid Control Flow Patterns and Collapse Into a Hierarchy of Representations
- Cross Couple Different Intermediate Representations
- Search the Multidimensional Intermediate Representation for a Specific Topology or Connectivity Pattern of Interest
- Check If Special Conditions Associated With the Pattern of Interest Exist
- Report Results

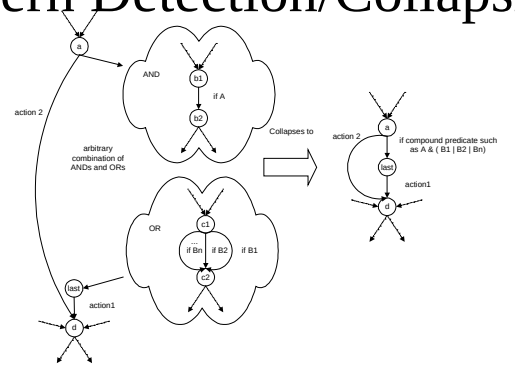
Loop Pattern Detection and Analysis Example

- Need To Detect Illegal Loop Patterns and Determine If Exit Is Possible
- Must Handle Loops Within Loops
- Testing Must Be Bounded For Practical Considerations
- Pattern Detection Utilizes Pattern Collapsing Algorithm To Simplify Analysis
- Steps:
  - Traverse Control Flow Models and Detect Loop Patterns
  - Detect Entry/Exits
  - Collapse Patterns To Simplify For Pattern Matching
  - Determine If Basic Pattern Is Valid
  - Trace Exit Predicate Equation Variables
  - Determine If Equations Are Activated By Loop Actions
  - Report Status

Sample Loop Structures

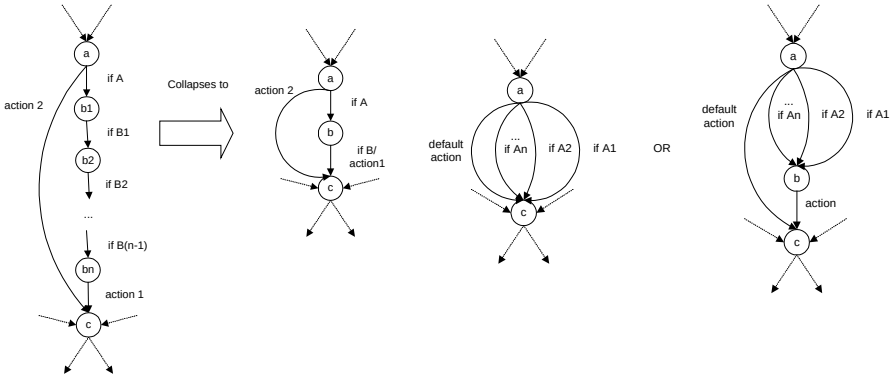


Pattern Detection/Collapsing



- Perform Graph Classification To Detect Fundamental Loops
- Collapse To Simpler Form
- Reapply Classification/Collapsing Until Fundamental Pattern Are Validated

Pattern Collapsing Example



Graphical Compound AND Collapsed To Single AND

Graphical OR Example

Future Work

- Complete Development of Tool At Tarragon Embedded Technology Ltd
- Conduct Acceptance Testing and Introduce Into Production Usage
- Develop Additional Suite Of Tests For Plant Models, Software Design and Specification Anomaly Analysis
- Establish As A Commercial Product
- Integrate With Automatic Code Generation and Automatic Unit Test Tools

Conclusion

- Graphical Based Modeling Languages Require Syntax Checking
- Automated Syntax Checking Is Required Due to Large Number of Items of Interest
- CASE/CACSD Tools Should Provide an Open Interface With Analysis Capabilities
- Intermediate Representations Reduce The Complexity of Analysis and Information Management
- Automation is Critical For Quality Improvement Within Resource Constrained Organizations

An Automated Inspection Tool For A Graphical Specification and Programming Language

Steve Toeppe, Ford Research Laboratory, Ford Motor Company, Dearborn, MI
Scott Ranville, Ford Research Laboratory, Ford Motor Company, Dearborn, MI

Abstract

Graphical programming and specification languages are becoming increasingly popular. The graphical languages are generally developed by a specific vendor and lack any rigorous conformance to independent standards. A large organization needs to standardize language usage regardless if it is textual or graphical. Commercial products exist to support coding standard compliance of textual languages. However, graphical languages generally lack such tools. Ford Motor Company has adopted a graphical language for powertrain control development specifications and design. An internal style standard has been developed and an automated consistency checker tool has been developed that is used to verify that the graphical models conform to over 250 style standard rules. The consistency checker tool is used as an automated inspection tool and is integrated into the development process. The tool provides a significant coverage and productivity improvement over manual inspection methods. This paper discusses the style construct standards, consistency checker tool features and some of the general analysis methods used by the tool.

1 Development Environment, Process and Style

Ford Motor Company has developed a control system and software development process that utilizes the graphical modeling languages provided by Matlab®[1] Simulink® and Stateflow™ [1, 2, 4]. The graphical modeling languages serve as the primary means of requirements and design specification capture. The graphical specifications support multiple uses including analysis, simulation, design, coding, testing and documentation. The multiple use approach requires that models are prepared with uniform syntax and style. A style standard has been developed that is used to constrain the rich notational capabilities of Simulink® and Stateflow™. The usage of the consistency checker has been factored into the process. Tool requirements have been established based upon usage scenarios.

1.1 Graphical Programming Tool Description

Matlab® Simulink® is a traditional block diagram tool popular with control systems engineers. The block diagrams have numerous similarities to traditional data flow diagrams popular in CASE tools. Simulink® also provides significant simulation capability that is particularly useful for control systems development. Stateflow™ supports (with some notational variation) the popular Harel statechart notation [3, 5] that is also part of the Unified Modeling Language (UML) standard [7]. Simulink® and Stateflow™ are tightly integrated. The diagrams in **Figure 1** to **Figure 3** provide examples of graphical models currently prepared at Ford Motor Company.

A complete powertrain control application may involve 5000 to 10000 diagrams that are “wired” together with control flow (event triggers) and data flow connections. A typical model contains a context diagram that defines external interfaces. Then there are multiple levels of hierarchy to

[1]MATLAB, Simulink and Handle Graphics are registered trademarks and Stateflow is a trademark of The MathWorks, Inc.

organize the model. The first level of hierarchy separates the model into single rate tasks. All subsequent lower layers of the hierarchy are added to properly manage the complexity of the model.

Figure 1 represents a hierarchy level that has control and data flow representations. The three input ports (2, 3 & 4) are each individual function call triggers used to start the Stateflow™ diagram ("tsegre_timer_control"). The Stateflow™ diagram is expanded in **Figure 2**. The Stateflow™ diagram has two output function call triggers used to start individual process specifications located within the "tsegre_timer" subsystem. The values in rectangular boxes represent constants. The input port 1 serves as a data value and output port 1 serves as a data value. The Stateflow™ diagram (**Figure 2**) represents a 2 state statemachine. Each state has a default transition (noted by large dot). The overall Stateflow™ diagram also has a default transition. Transitions can occur whenever the Stateflow™ diagram is activated or called by one of the input function call triggers.

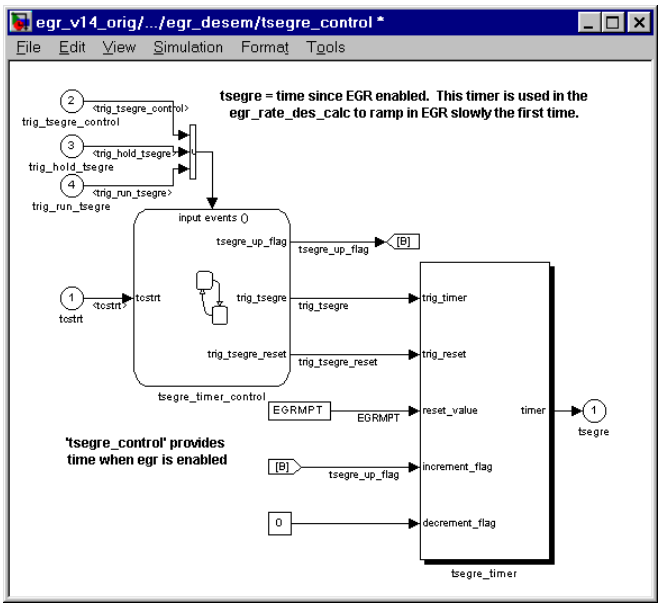


Figure 1 Control and pSpec Combination

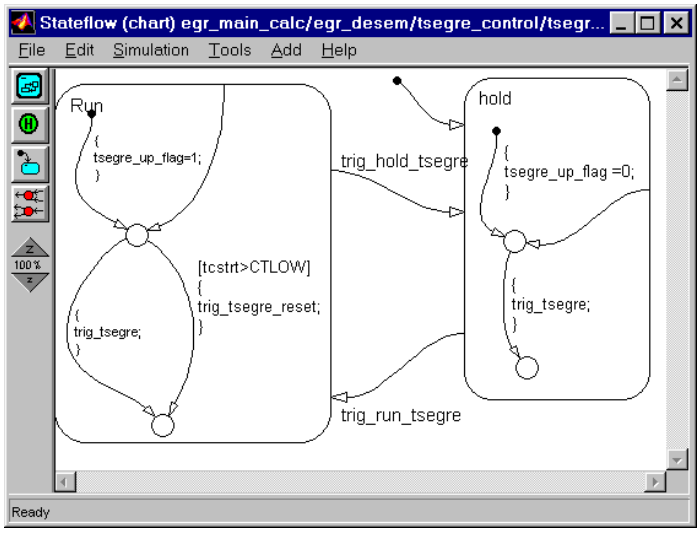


Figure 2 State Machine Example

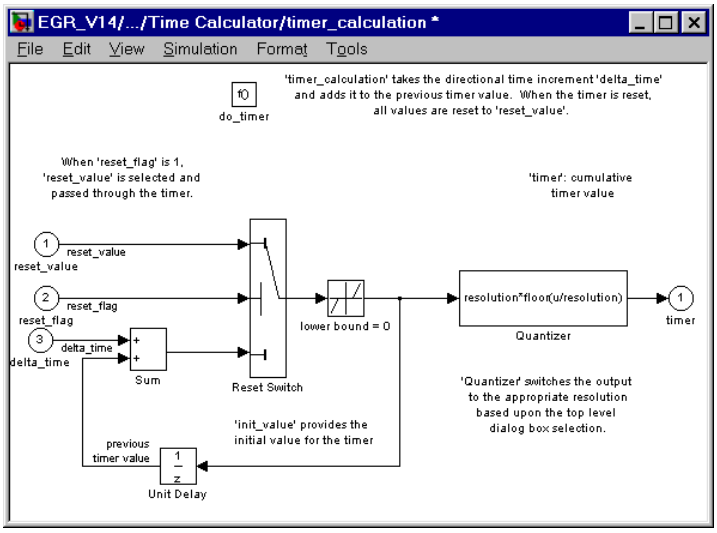


Figure 3 pSpec Example

The Stateflow™ examples in **Figure 4** to **Figure 6** illustrate some typical control logic that does not involve states. **Figure 4** illustrates a straight line action segment that does not involve predicate logic. **Figure 5** illustrates "If Then Else" logic. The top most node or junction is the decision point where the predicate equation is evaluated. The flows emanating from the decision junction to the terminating junction provide the actions. **Figure 6** illustrates a more complex "IF Then Else" logic involving a "graphical OR" in the predicate equation. It is important to note that some flows do not have action statements. Flows that do not have action statements can in many cases be collapsed into predicate equations.

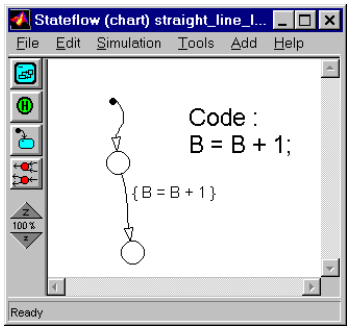


Figure 4 Straight Line Action Segment

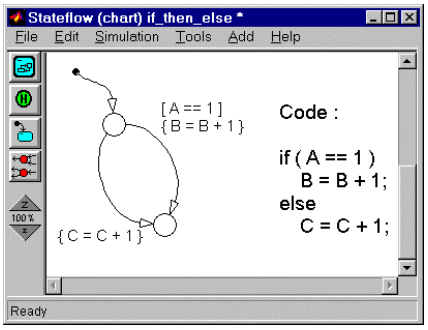


Figure 5 "If Then Else" Control Flow Logic

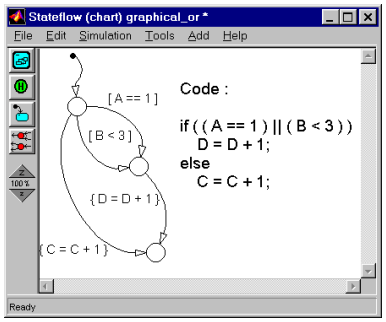


Figure 6 Graphical Or Inside "IF Then Else" Control Flow

Graphical programming languages intended for embedded control systems are made up of a few major components. Algorithmic operators are used to capture mathematical equations. Control flow operators are used to capture predicate equation logic to permit alternate algorithms to be executed for a given operating condition. A third design operator is used to control rates of execution for particular algorithms.

The algorithmic operators provide all of the standard algebraic blocks and a number of special purpose blocks that are oriented towards control system design. Some examples of the special purpose blocks include the following:

- Lookup Tables
- Logic
- Matrix Math
- Trigonometric
- Saturation
- Switch
- Memory
- Delay

The control flow operations support flow chart control operations and state machine control operations. The notation is made up of a few basic symbols that can be combined in numerous ways to create a rich notational capability. An action language also provides numerous textual operations that can be used to specify predicate logic or action sequences.

The design operators permit specification of execution rates for many blocks to create a "multi-rate" model. Most Matlab® Simulink® applications use multi-rate modeling. However, the Ford style minimizes the usage of multi-rate modeling in order to establish a closer mapping to the production software task behavior. The software task execution behavior is established with Stateflow™ diagram schedule models.

1.2 Process Description

The development process [6] is a production development process for Ford Motor Company powertrain (engine/transmission) controllers. The development process has numerous steps following the "V" system/software process as shown in **Figure 7**. The Matlab® Simulink®/Stateflow™ diagrams support the system requirements through software design steps of the process. The models are prepared in a specific style dictated by an internally developed Computer Aided Control System Design (CACSD) style guideline. The guideline specifies specific graphic modeling or graphical programming rules and guidelines.

The CACSD development process calls for several major steps that are then broken down into numerous smaller steps. The primary process steps are as follows:

- System Architecture
- System Design
- Model Preparation
- Model Validation
- Software Architecture
- Software Design
- Software Coding
- Software Unit Testing
- Software Integration Testing
- System Integration Testing

1.3 Style Guideline

The CACSD style guideline ensure that consistent, easily understood graphical specification and programming models are prepared that also permit automation to be applied. The primary objective is readability. The models serve as documentation for a large base of users who are not directly involved with software engineering and may not have a software background. However, it is assumed that they can understand a traditional engineering block diagram. The models must be easily prepared, completely specify the desired software behavior and be capable of simulation and analysis. This is critical to fully leverage the models for all phases of development.

The CACSD style addresses the following major areas:

- Context Diagram Interfaces and Scheduling
- Subsystem Hierarchy and Interconnections
- Subsystem Notation Usage
- Scheduling Syntax
- Signal Connectivity
- Individual Block Usage
- Stateflow™ Diagram Syntax
- Software Design Information
- Documentation

Figure 7 System "V" Development Process

1.4 Consistency Checker Usage Scenario

The CACSD process calls for a graphical model inspection as part of the model validation process prior to software design. The inspectors review the models conformance to the style and algorithmic content. Manual style guide inspection methods require significant time to complete. The consistency checker tool automates the style review process to enable the inspectors to focus more on the algorithmic portion of the inspection.

The modeler uses the tool to determine what style guideline inconsistencies are present in the graphical model. An interactive browser with hot links back to the model permits each anomaly to be easily inspected. The modeler must correct each anomaly or establish a justification for the anomaly. Once all of the anomalies are corrected or justified, the user can print out a report that is presented at the inspection and placed under configuration management. The style portion of the inspection is limited to reviewing the consistency checker report and reviewing the more subjective issues associated with the style standard.

A significant portion of the powertrain controller algorithm development is maintenance with incremental modifications. When a small change has been made to a model that has been previously validated as style compliant, the consistency checker tool is then only applied to the portions of the model that have changed.

2 Tool Operation

The Consistency Checker general principles of operation include the following basic categories of operation:

- Extract model structural connectivity and information content
- Analyze structural information to verify selected style guideline conformance checks
- Report results via browser and reports
- Rerun selective checks during corrective modeling
- Save/Restore results and analysis session

It is necessary for the tool to traverse the entire graphical model to obtain the relevant information. Some checks require extensive information from many subsystems while other checks derive the necessary information from within a single subsystem. The tool extracts the information with an efficient and organized approach. Since it is necessary to rerun single selected checks, the tool is able to handle small scale extraction as well as large scale extraction.

The tool analyzes the structural information to verify that the checks have either passed or failed. There are three major approaches to the analysis. The first approach is to fully extract all information and then perform the analysis. The second approach is to perform the analysis on the fly during the information extraction phase. A third hybrid approach is a combination of the first two. The tool extracts core information that may be needed by several checks. Once enough core information has been extracted, specific rules are checked.

After the tool has analyzed the information, it reports any failures. Since the number of problems can be high, it is necessary to manage the information with a browser or report. The tool supports printout, textual file dump, HTML, and workspace dump reporting. Once failures have been identified, it is necessary to resolve the problems. In order to quickly determine if a correction is satisfactory, it is possible to rerun a selected test. This permits problems to be eliminated one by one. In some cases, it is necessary to only partially complete the correction of problems and finish the effort at a later point in time. A save and restore feature is provided to address this usage.

Numerous feasibility experiments have been conducted to determine if it is possible to extract the necessary information to implement the checks. The Mathworks provide extensive capability to extract information at all levels of the model including Simulink® and Stateflow™. The commands “get_param” and “sf” provide access to all required information. However, some information is not always directly accessible and may require several steps to obtain the desired information.

The consistency checker supports the checks listed in **Table 1**. There are over 250 individual specific checks that are contained within the sub-components of the checks in **Table 1**. Major categories of checks are identified in bolded text.

The "**Context Subsystem Name**" checks are intended to verify that the top level context diagram conforms to specific naming, wiring and triggering conventions. The naming conventions are intended to ensure that top level signal, subsystems and triggers are named in a uniform manner. This aids understandability, system integration and subsequent automation.

The "**Execution Context Diagram Format**" checks are intended to verify that the execution context diagrams are properly formatted, triggered and scheduled. The execution context diagram contains one or more virtual subsystems that each represent a single rate subsystem hierarchy (or possibly tasks/ISRs). Each subsystem should only have one function call trigger or a vector of triggers entering on the left side of the subsystem block. The function call trigger should trace to either a Stateflow™ diagram or a triggered subsystem. If a vector of triggers is defined, the consistency checker will verify that all triggers have the same destination subsystem. The intent is that a subsystem can be called from multiple sources, but the calls must constitute a single entry. Special cases exist for Stateflow™ diagrams. This suite of checks is intended to ensure that the modeling style is consistent with legacy task schedules, independent of the Simulink® multi-rate scheduler and presented in a uniform manner.

The "**Simulink® Block Consistency Checks**" are intended to verify that the individual Simulink® blocks are properly configured and used appropriately per the style guide. Issues regarding naming conventions, initialization, switch usage and expressions are checked.

The "**Subsystem Consistency Checks**" are intended to verify that Simulink® subsystem diagrams are properly named, configured, scheduled and wired. The subsystem scheduling checks are intended to determine if the Matlab® multi-rate scheduler is involved with block execution order. It is desired to keep all rate based scheduling under the control of the modeler specified Stateflow™ diagrams. Several checks are intended to determine if the subsystems are actually configured or wired to produce a result. The check will not determine that the result is correct, it only determines that something will happen.

The "**Data Consistency Checks**" are intended to verify that the data definitions and memory usage are consistent with external definitions. The checks will interact with the data dictionary that contains all data definitions. Some of the checks are concerned with calibration data definitions and declaration rights. Other checks are concerned with verifying consistent usage of data and data type matching between model components.

The "**Trigger/Event Consistency Checks**" are intended to verify that subsystem triggers are properly wired, named and configured. Proper naming is critical to understandability of calling hierarchies. Name uniqueness is checked. Wiring and routing checks ensure that scheduling will behave per the defined rules in the style guide.

The "**Signal Wiring Consistency Checks**" are intended to verify signal wiring is properly named, configured, routed and utilized. Several checks are in place to determine if all signals are used or have a proper termination. Name propagation rules are checked. Connectivity is verified on Simulink® virtual connectors (Goto/From) tags.

Context Subsystem Name	Check For Unnecessary Signal Names
Context Diagram First Subsystem Name	Check Switch Variable Names Go To a Switch
Context Diagram Input and Output Wiring	Merge Block Rules
Context Diagram Scheduling Inputs	Vector Signal Propagation and Naming
Execution Context Diagram Format	Vector Signal Termination and Sources
Execution Context Subsystem/Trigger Names	Verify Vector Signal is Named With Attribute
Execution Context Trigger	
Execution Context Does Not Require MATLAB® Scheduler	**State Chart Consistency Checks**
Simulink® Block Consistency Checks	Check Each State Chart For A Default Path For Flow Control
Port To Signal Compare	Check For Unguarded Path As Default and Invocation of SF Clockwise Rule
Initialization of Data Store, Delay Blocks, etc	Verify That State Chart Is Triggered With A Function Call
Verify Sampled Blocks Use Inherited Sample Times	Check For Flow Logic Classifications Into Acceptable Patterns
Check For Appropriate Usage of Multiport Switch Block and Standard Switch Block	Check For Recursion SF Constructs
Check For Non Stateflow™ diagram Generated Switch Controls	Check For Action Only At End of Classified Pattern
Check For Constant Expressions in Dialog Boxes	Check For Valid Compound Conditionals
Subsystem Consistency Checks	Check For Proper Types For "==".
Properly Triggered Subsystem	Check While/For Loops For Termination
Subsystem Held/Not Held Output	Check For Unstructured Loops
Unwired Ports and Subsystem Ports	Check For Valid Multiway Branch
Identify Any Non Masked Subsystems	Delete Check
Verify That Masked Subsystems Have Something That Will Transform Inputs To Outputs	Verify Events Are Local or Function Calls
Check For Proper Masked Subsystem Entries	Verify Broadcast Rules
Check For Labeled Blocks Instead of Masked Blocks	Verify State Hierarchy Rules
Verify Correct Prefix on Subsystem Names	Verify Stateflow™ Event Wiring Rules
Detect Virtual Subsystem With Only Annotations or Text Mask	Verify Stateflow™ Input and Output Data Wiring Rules
Data Consistency Checks	Check For Write To Workspace From SF
Data Dictionary Type vs Usage	Check For Data Transformation in Stateflow™ diagrams
Proper View of Calibration Parameters	Check For ML Operation in Stateflow™ diagram
Verify Calibration Parameters in Model vs Data Dictionary (Check Dimensions)	Check For SF Keywords
Check for Unnecessary Data Stores	State Machine Data Flow Rules
Trigger/Event Consistency Checks	Event Broadcast Check
Verify Triggers Are Function Call Triggers (handle exceptions)	Check Model For Unique Names When Not Considering Case
Verify do_<p_spec_name> and trig_<hierarchy_name> Are Correctly Named	Verify Stateflow™ diagram Flow Diagrams Do Not Use Event "/".
Unique CSpec Names and Event Signal Names Within Feature	Verify Stateflow™ diagram Does Not Invoke Any C Library Utilities or Use C Code
Verify do_<> ends in a pSpec	**Software Design Consistency Checks**
Verify trig_<> ends in a Stateflow™ diagram	Check For Vectorization
Unique Application Signal Names Within Feature	Function name Rules
Signal Wiring Consistency Checks	File Name Rules
Check if Tags Are Local	**Documentation Consistency Checks**
Check For Matching GoTo and From Tags	Verify Correct Font Attributes
Check Signal Name Propagation	Check For Unneeded Drop Shadows in Annotations

Table 1 Consistency Checker Checks

The "**Stateflow™ diagram Consistency Checks**" are intended to verify a number of rules relating to Stateflow™ . Numerous issues regarding control flow are addressed. Control flow patterns are checked to determine if they are valid. Checks on loop usage are provided.

The "**Software Design Consistency Checks**" are intended to verify software design and automatic code generation information is properly formatted.

The "**Documentation Consistency Checks**" are intended to verify that select issues important to documentation generation are properly setup. Checks for proper font attributes and unneeded drop shadows are provided.

The graphical user interface for the consistency checker is shown in **Figure 8**. The expanding hierarchy of information illustrates a problem involving signal names. Each error has a hot link directly to the Matlab® model that has the offending anomaly. When the error is selected, the Matlab® diagram blocks involved are highlighted. The next /previous keys permit rapid traversal through all errors.

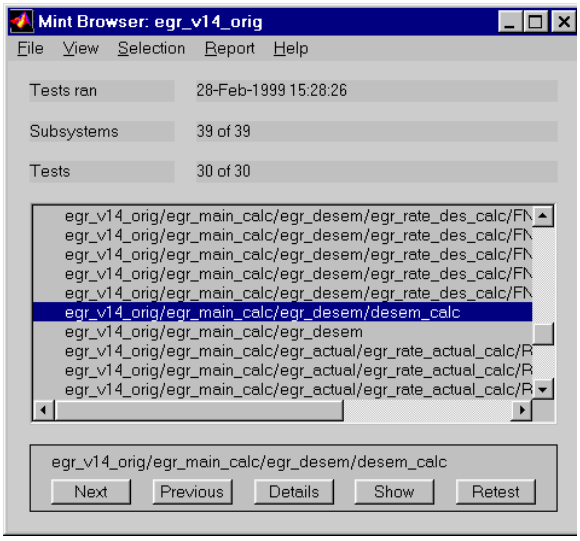


Figure 8 Consistency Checker Browser

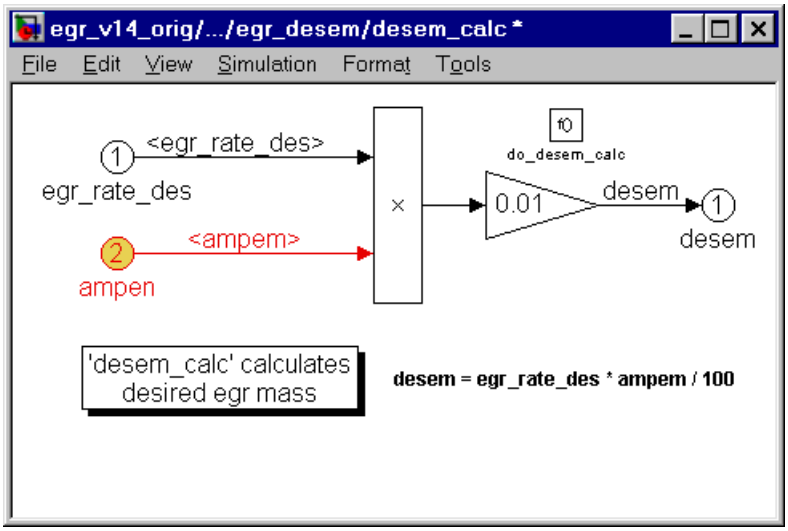


Figure 9 Example of Anomaly Error and Highlight

The GUI shown in Figure 8 is the primary consistency checker browser. The use of expandable hierarchies permits information to be easily viewed and hidden when not required. The anomoly highlighted in Figure 9 shows a mismatch between a port name and its associated signal flow. In this case, the signal flow name does not exactly match the port name. The highlighted anomaly in the model is displayed simply by double clicking on the browser line of interest. Additional details are available for each error via a details popup dialog. Additional check/rule information is available via HTML help files.

The consistency checker is configurable to permit specific checks or model segments to be disabled. All operations are available via application programmer interfaces (api), thus enabling automated regression testing and large batch jobs.

3 Analysis Methods

The Matlab® Simulink® and Stateflow™ products have open application programmer interfaces that permit access to all modeling information. The signal flow connectivity internal representation can be viewed as a special directed graph. It is possible to traverse the model based upon signal flow

connectivity within Simulink®. Most of the signal flows correspond to data flow. However, trigger signals provide control flow scheduling. The Stateflow™ diagram control flow and state-machine internal representation can also be viewed as a special directed graph.

Matlab® provides numerous diagram interrogation methods that typically operate on one diagram object at a time. However, most of the consistency checks require the analysis of large diagram objects that are made up of elemental objects. It is therefore necessary to establish data structures for the large objects. The Matlab® workspace environment supports a wide variety of array, vector and structure data elements and manipulation methods. The use of the Matlab® utilities and data representations simplify the analysis efforts. The use of directed graphs is particularly useful for representing the state-machine representations captured in Stateflow™. Directed graphs can be represented as adjacency matrixes or as sparse connectivity matrixes within the Matlab® workspace.

Data flow analysis is critical for establishing data flow dependencies between algorithms and between algorithms and control flow logic. Loop analysis algorithms require data flow detection in order to determine the data dependencies between loop actions and loop termination logic. With the Ford style guideline, data flow traversal occurs primarily within the Simulink® block diagrams. The connectivity from input ports, transformational blocks and output ports can be easily established with a succession of calls that traverses each path and branch.

Diagram traversal is easily accomplished with recursive m-scripts. Matlab® permits controlled levels of recursion to be supported. Provided the problem is properly partitioned and bounded, it is possible to use recursive techniques for data flow and control flow traversal.

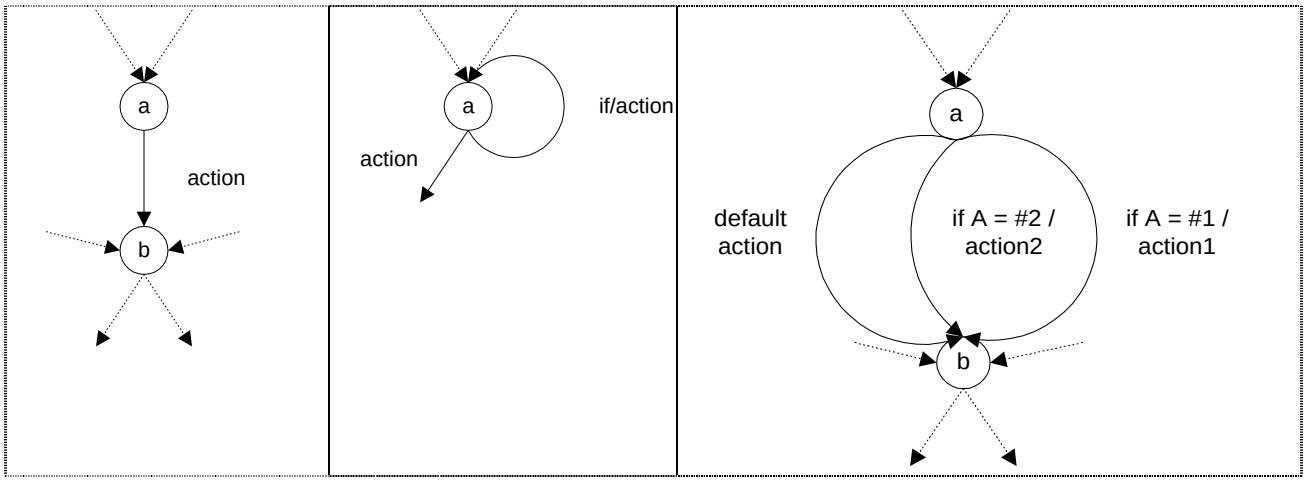


Figure 10 Straight Line Action Sequence **Figure 11 Simple Loop** **Figure 12 OR Switch**

The examples in **Figure 10** to **Figure 12** illustrate basic control representations found within Stateflow™. Many more variations are possible. **Figure 10** illustrates a straight line action that does not have any branching. While it does not contain control flow, it is fundamental to the control flow representations. **Figure 11** illustrates a simple loop with a loop action and an exit action. **Figure 12** illustrates an "OR Switch" which is analogous to a "C" switch statement (assuming "if" logic involves "A" in each flow).

4 Anomaly Detection Methods

Detection of anomalies varies from simple to complex. The simple checks generally involve testing for the occurrence of some condition associated with specific blocks or notations. The simple check generally involves the following steps:

- Find all occurrences of the specific block of interest
- Check if special condition associated with block is present
- Report results

A simple example requires that the names associated with subsystem port names need to match the signal flow names. This check is performed by finding all subsystem ports within the model. Then the associated signal flow is matched to the port. Finally, the name of the port and the signal flow are compared. If they do not match, an error is reported.

Several of the checks are complex. They generally involve looking for some topological connectivity pattern that may be coupled with numerous special cases and then determining if an invalid usage scenario has resulted. The topological connectivity patterns can include any or all of the following:

- Data flow connectivity between blocks
- Control flow graphs
- Call trees
- Hierarchical organization

The complex checks generally involve some or all of the following steps:

- Extract intermediate representation of some connectivity or topology of interest
- Associate the relevant model information with the intermediate representation
- Detect valid control flow patterns and collapse into a hierarchy of representations
- Cross couple different intermediate representations
- Search the multidimensional intermediate representation for a specific topology or connectivity pattern of interest
- Check if special conditions associated with the pattern of interest exist
- Report results

A complex check example is a check that determines if a loop has only a single entry node and single exit node. The style does not permit loops with multiple entry or exit nodes. The examples in **Figure 13** through **Figure 18** illustrate various loop scenarios that are possible using Stateflow™.

The loops illustrated in **Figure 13**, **Figure 16** and **Figure 18** illustrate style guide compliant (valid) loops of various types. The loops in **Figure 13** and **Figure 18** illustrate loops with a common single entry node and single exit node. The loop in **Figure 16** illustrates a case where the exit node is a different node than the entry node. The loops in **Figure 14**, **Figure 15** and **Figure 17** are illegal due to entry or exit path violations. **Figure 14** illustrates two entry points to the loop. **Figure 15** has two exit points from the loop. **Figure 17** lacks an exit. **Figure 18** illustrates a legal case where a “If Then Else” pattern resides within the loop.

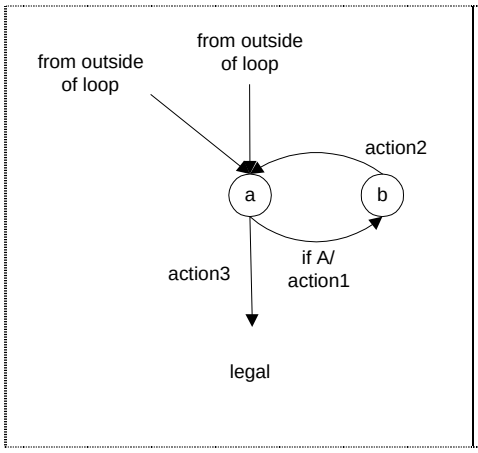


Figure 13 Legal Loop Representation

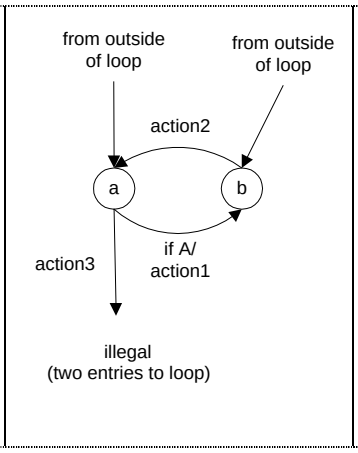


Figure 14 Illegal Loop: 2 Entries

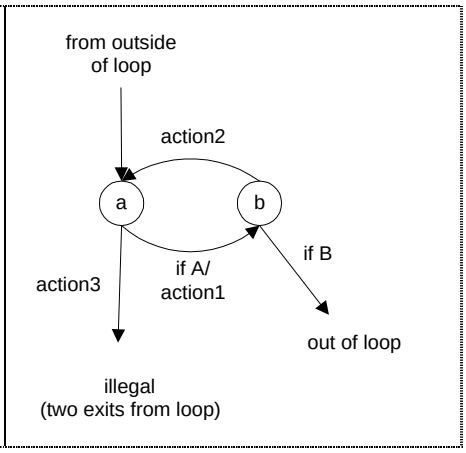


Figure 15 Illegal Loop: 2 Exits

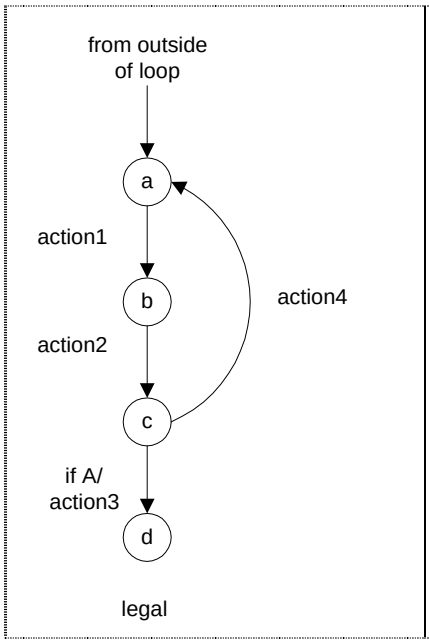


Figure 16 Standard Legal Loop With Internal Pattern

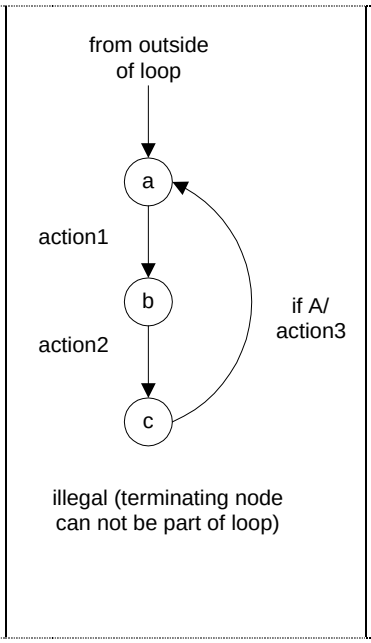


Figure 17 Illegal Loop Due To No Terminating Node

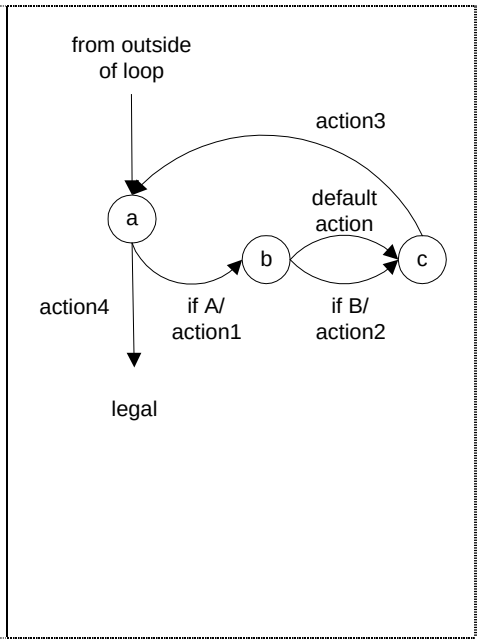


Figure 18 Legal Loop With Internal Logic and Action Pattern

A second complex check example is a check that determines if the variables used in the control logic associated with a loop can be changed or updated directly or indirectly by the loop. The intent of the check is to determine if the loop has any potential of exiting. This requires that the loop variable is updated either within the loop or in a function called by the loop. If neither case is true, then there is an error. The check is performed by first detecting the presence of a loop. Once a loop has been detected it is necessary to determine if the loop has any loop variables. Once the loop variables have been detected, it is necessary to determine if the loop variables are written by the loop directly or indirectly. This check does not prove that an exit is possible. It only detects when it is impossible to have an exit due to no loop variable writes.

The two complex loop checks have several common components. It is necessary to extract the control flows, detect loops and determine entry/exit. The model information structures are arranged so that they may be reused by these two checks. Additional information is needed for the loop variable update check.

The two complex loop checks can be implemented by extracting the control flow logic from Stateflow™ and setting up a directed graph representation. An exhaustive search of all paths with a coloring stage can be used to detect a loop. It is assumed that the total size of the control logic is containable due to previous partitioning. Once a loop has been detected, it is necessary to determine the entry and exit transitions. This can be accomplished by searching for an inbound or outbound flow.

The base of common information is established prior to additional information extraction or checking. The common base of loop information extraction is enough to permit the entry/exit check to be performed. Each valid loop is checked. If there is one and only one inbound and outbound flow, then the loop is valid. A missing inbound flow indicates dead control logic. A missing outbound flow indicates an infinite loop has been specified. Multiple inbound or outbound flows may result in intended system behavior, but they represent poor design and present maintenance and testing problems.

In the case of the loop variable check, additional information is needed in addition to the common loop information. Once the outbound flow is detected, the predicate equation associated with the transition can be parsed out. The variables within the predicate equation are then identified. The action portions of the loop are searched to determine if any assignment statements exist using the variables associated with the loop. The action portion can also include triggering additional processing external to the loop. The external processing sequence also needs to be analyzed in a similar manner. The case of the external processing (via a function call trigger) requires further analysis that may require extraction of data flow relationships and additional control flow relationships. This is potentially an unbounded problem. In order to remain practical in the tool usage and execution time, a limit is placed on the extent that a search is made. If the problem is not solvable within a bounded search, an advisory is raised indicating that the check failure may have occurred and further engineering analysis is warranted.

An additional complication associated with the loop checking occurs when loops are within loops and there is “If Then Else” logic embedded within the loop. In order to solve this issue, it is necessary to detect control logic patterns that are valid. Once a pattern is detected, it is possible to either ignore it or collapse it during analysis and checking. There are also additional style checks that require control flow logic to be classifiable to valid patterns as listed below:

- IF – THEN
- IF – THEN – ELSEIF
- Compound Conditionals (AND/OR Groups)
- Multiway Branch
- While Loop
- Do While
- For Loop

In order to combine the valid pattern check and the complication of the loop check, it is necessary to perform a graph classification prior to the analysis steps. This involves a search for fundamental patterns that can be collapsed into larger patterns. Conceptually this is shown in **Figure 19**. Graphical "AND" and "OR" cases can be collapsed into simple predicate equations. A compound "AND" example is illustrated in **Figure 20**.

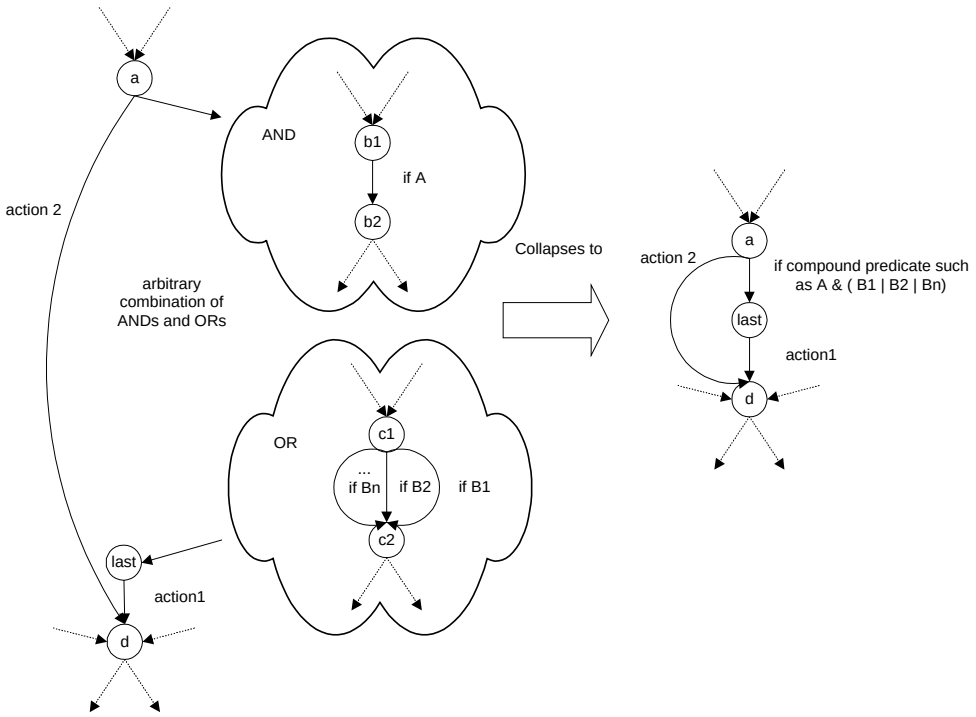
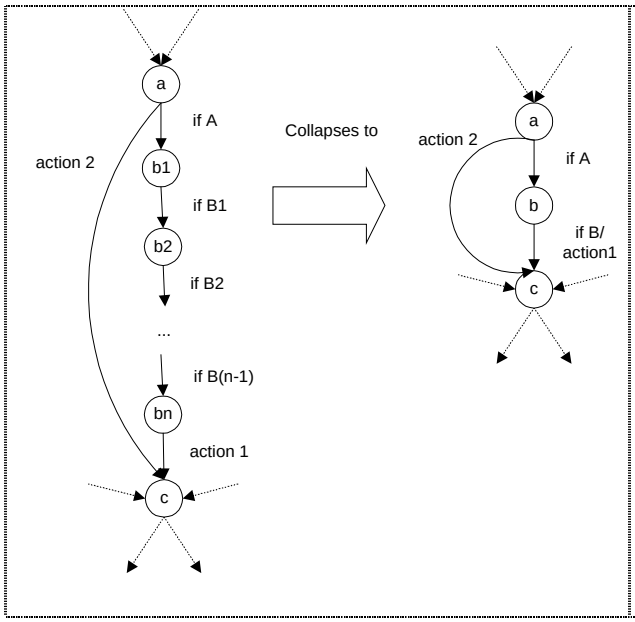


Figure 19 Control Collapsing

The primary complication for the graphical AND construct is to distinguish it from an embedded if construct. The embedded if construct is an if-then-else construct with another if-then-else construct on the true path of the original if-then-else construct. If there is a series of embedded ifs, but the default action on all these is the same, then it is an illegal construct, because it should have been a graphical AND construct with a single default path.

A complication to identifying these topologies is when legs of different patterns merge together. This merging may occur because each segment has common actions, and this is easier for the modeler to maintain. Another complication is that, almost to an arbitrary degree, any construct can be embedded within any leg of another construct. The tool needs to handle this complexity.

One approach to the issue of constructs embedded in other constructs is to identify some legal, self-contained construct, and "eliminate" it from the control flow. This elimination, or collapsing of the control flow, will make the overall flow diagram smaller and easier to analyze.



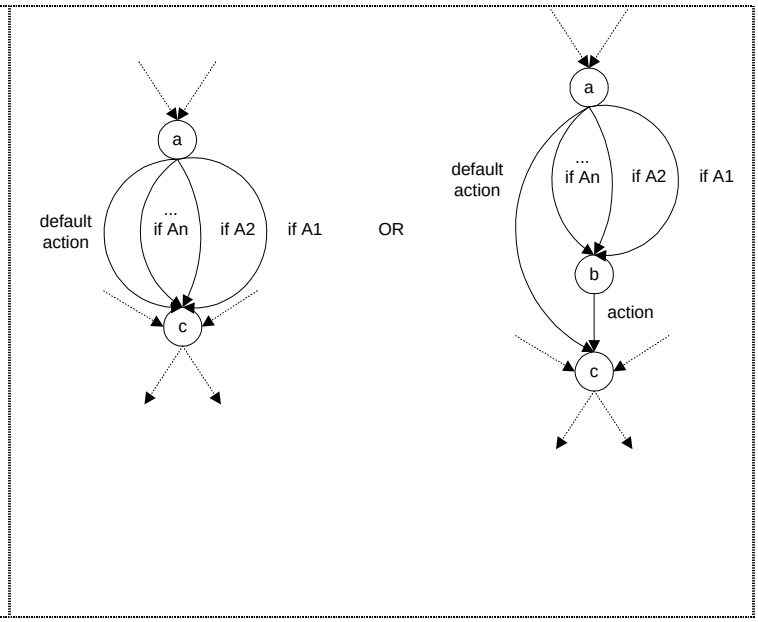


Figure 20 Graphical Compound And Collapsed To Single And Predicate Equation

Figure 21 Graphical OR

The graphical OR, as shown in **Figure 21**, is one of the more complicated constructs, because it must be distinguished from a multi-way branch and not mistakenly flagged as a violation. If there are actions on the conditional legs of the construct, then it is a multi-way switch construct and not a graphical OR. However, if there is a single common action for all the condition legs, then it is a graphical OR construct.

5 Future Work

The tool is being developed by Tarragon Embedded Technology Ltd[2] with extensive requirements preparation, acceptance checking and validation at Ford Motor Company. During the preparation of the acceptance check cases, several special case situations occurred that required refinement of the CACSD style guideline. The refinements require further analysis to determine how they fit into the inspection process. Extensive checking is underway to demonstrate that the tool has adequate robustness for production product development application.

In order to minimize life cycle product costs, it is generally preferred to use commercial off the shelf (COTS) products at Ford Motor Company. An effort is underway to establish the consistency checker as a commercial product. This would permit a COTS solution to emerge that can satisfy the needs of other Matlab® Simulink®/Stateflow™ applications. Ford Motor Company is also advocating uniformity of Matlab® graphical programming styles and processes within the automotive industry. Efforts are underway with The Mathworks, several major automotive OEMs and Tier 1 suppliers to establish uniform styles and methods.

[2] Tarragon Embedded Technology Ltd
Cambridge United Kingdom
44 1223 323336

The first suite of checks for the consistency checker were intended for the graphical programming style. Additional check specifications are under development for physical plant modeling (engine/transmission physics), specification/design anomalies and detailed software design consistency.

6 Conclusion

A highly automated inspection process will improve the overall productivity of a software engineering organization. The productivity improvements will be achieved in reduced time to detect and correct style violation, thus permitting increased focus on algorithmic review. In order to automate an inspection process it is necessary to have a well defined style and specification capture method. If CASE or CACSD tools are used, an open interface is essential to automation. Matlab® Simulink® and Stateflow™ provide a very flexible environment for analysis and development. The development of consistency checker checks have been built up entirely upon the Matlab® script language and "Handle Graphics®".

The consistency checker provides an automated suite of checks that check the graphical programming style guide compliance. The checks are implemented to support reuse of common information interrogation methods. The information is presented to the user in multiple formats to suite the process steps and user needs.

In order to develop the check algorithms, it is necessary to establish analysis and detection methods. Directed graph intermediate representations are useful for control, data flow and state machine behavior analysis.

References

1. Butts, K., "An Application of Integrated CASE/CACSD To Automotive Powertrain Systems" Computer-Aided Control System Design, 1996. Proceedings of the 1996 IEEE International Symposium on, Pages 339-345
2. Toeppe, S., Butts, K., Ranville, S. "Specification and Testing of Automotive Powertrain Control System Software Using CACSD Tools" 1998. Proceedings of the 17th AIAA/IEEE/SAE Digital Avionics System Conference.
3. Harel, D., "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming 8,m 1987 Pages 231-274
4. Hatley, D and Pirbhai, I., "Strategies For Real Time System Specification", Dorset House Publishing 1988
5. Stateflow™ Users Guide, Version 1, The MathWorks., 1997
6. Butts, K,. et.al., "Automotive Powertrain Control Development Using CACSD", Perspectives in Control: New Concepts and Applications, Tariq Samad (ed.), IEEE Press, To be Published in 1999.
7. Jim Rumbaugh, Ivar Jacobson, and Grady Booch, *Unified Modeling Language Reference Manual*, Addison Wesley, December 1997

Advanced Automated Testing Architectures

John Kent M.Sc.

CISS Ltd
29 Abbey Rd
Woking
Surrey GU21 4PQ
Tel: 01483 768992
Email: john_kent@compuserve.com
Web:
http://ourworld.compuserve.com/homepages/john_kent

© CISS Ltd 1999

Overview

- The Promise of Test Automation
- Where Are We Now?
- How Did We Get Here?
- Current Automated Test Methods
- Reasons For New Approaches
- Advanced Architectures
- CISS's Test Automation Architecture

© CISS Ltd 1999

The Promise of Test Automation

Use Automated Testing to increase ‘testedness’ of the system

- Large numbers of automated tests
- Tests to run unattended
- Moving manual testing to more complex test scenarios

© CISS Ltd 1999

Where Are We Now?

- The Promise of Automation not fulfilled for most
- Lack of progress blamed on tools
- Lack of progress blamed on approach
- Tes automation community realising that the architecture is the problem.

© CISS Ltd 1999

How Did We Get Here?

- Mainframe Automated Test Tools: terminal/processor conversations
- Automated Testing of PCs and GUIs: complex user/software interaction
- New Tools are Script-Based

© CISS Ltd 1999

Automated Test Creation Methods

- Record/Playback
- Test Translation
- Data-Driven

© CISS Ltd 1999

Paradigm Lost: Record/Playback

- Recording does not work
- It produces unmaintainable code
- We are interested in what ***may*** happen, not what ***has*** happened
- An automated test should be an *interaction* between the test tool and System Under Test(SUT)

© CISS Ltd 1999

Test Translation

Manual Test Procedures

© CISS Ltd 1999

Test Translation

Manual Test Procedures

Add customer "Bloggs"

Create an order for product X

Translated into test tool language

Automated Test Code

Menuitem

"customer"

Type("Bloggs")

.....

Menuitem

"Order"

© CISS Ltd 1999

Test Translation

Disadvantages:

- Test data is hard-coded
- Test Procedure: test program ratio =1:1
- Many test programs to maintain
- Many test programs to test
- The more tests, the more test programs to maintain and test

© CISS Ltd 1999

Reasons For New Approaches

- Automation not fulfilling its promise
- Current architectures not delivering
- Scripts are really programs
- Automated tests must be tested
- Importance of maintenance

© CISS Ltd 1999

Myth:

"Automated tests are *scripts*"

Enter Hamlet

To be or not to be-that is the question;

Whether 'tis nobler in the mind to suffer.....

```
if (audience looks bored)        {
        Drop trousers
}
```

© CISS Ltd 1999

Scripts are really programs

- Automated Test programs cannot be sequential instruction lists
- Tests must expect the unexpected
- They have conditional processing and loops
- Automated tests are *programs*

© CISS Ltd 1999

Automated Tests are Programs

Automated testing is *software engineering*.

Test code must be:

- Developed
- Maintained
- *Tested*

© CISS Ltd 1999

Re-definition Of Terms

- Test Program - the part of an automated test written in the test tool language (e.g. Test BASIC, TSL, SQA BASIC)
- Script - a list of actions to be performed in order to exercise a test. Usually written using a word processor spreadsheet or test management tool

© CISS Ltd 1999

Testing Test Programs

When running automated tests we are testing two things:

- The System Under Test (SUT)
- The Automated Tests themselves

© CISS Ltd 1999

So Far:

- Automated Tests are *programs*
- Automated Testing is software engineering
- Tests must be *maintained*
- Tests must be *tested*
- Test Translation method means many tests to maintain and test

© CISS Ltd 1999

Measuring Maintainability

- Ratios
- Test: Test Program
- SUT Screen/window: test program
- UI Object-to-Object Reference Ratio
- Test Procedure: Test Program
- Not always comparing like with like
- Also measures testability

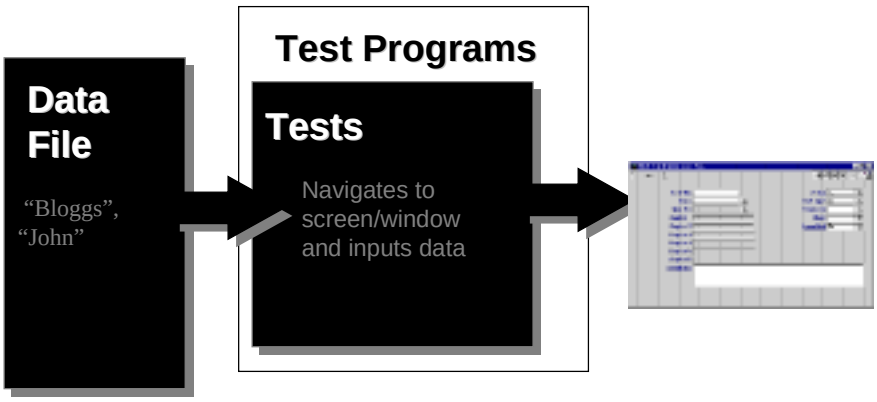
© CISS Ltd 1999

Advanced Architectures

- Basic Data driven
- Framework Based
- Advanced Data Driven (with Navigation/Actions)

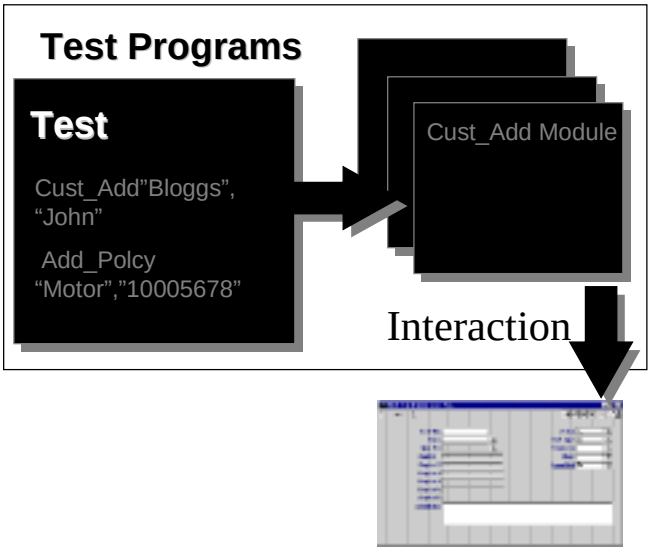
© CISS Ltd 1999

Basic Data-Driven



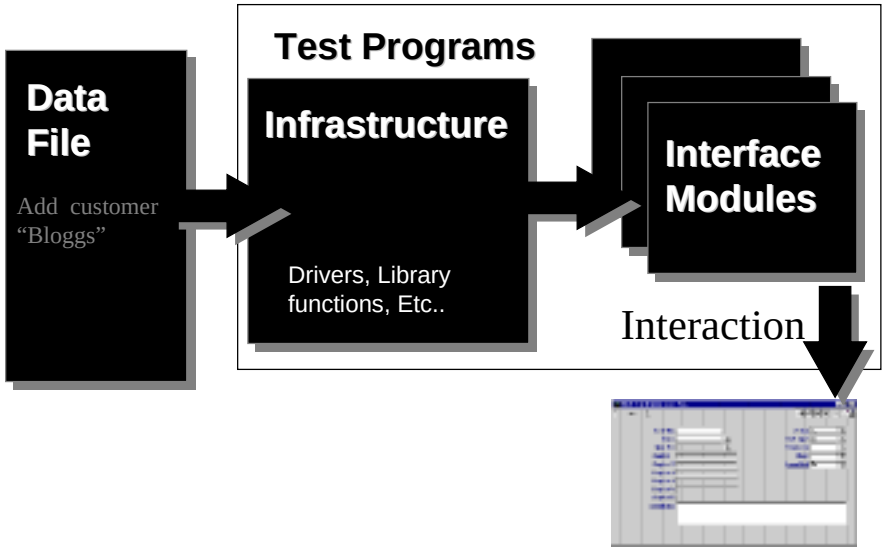
© CISS Ltd 1999

Framework



© CISS Ltd 1999

Advanced Data-Driven



© CISS Ltd 1999

Advanced Data-Driven

- Actions in Test Data
- Navigation in Test Data

CUST	ADD	Smith	John	Parregren	29 Abbey Rd
ORDER	ADD	Widget	Green	2inch	
CUST	UPDATE	Smith	Jon		30 Abbey Rd
CUST	CHECK	Smith	Jon	Parregren	
CUST	ADD	Bjloggs	Jonhansen	CUST-01	

- <u>Business Object Level</u>
- <u>Screen/window Level</u>
- Wrappers

© CISS Ltd 1999

Business Object Level

- Business tasks are automated
- Tests specified at business language level
- End-users can understand
- Object-to-Object Reference ratio reduced

© CISS Ltd 1999

Screen/Window Level

- Screen/windows are ‘wrapped’ by test programs
- Test Program: Screen/window ratio = 1:1
- Object-to-Object Reference ratio of 1:1 or 1:2

© CISS Ltd 1999

CISS’s Automated Test Architecture

- Screen/window level data-driven
- Screen/window to test program ratio of 1:1 - screen/window is wrapped
- Object-to-Object ref = 1:1 or 1:2
- All navigation and actions in data
- Each Class of UI object is wrapped by one function

© CISS Ltd 1999

CISS's Automated Test Architecture

- CISS's TestDev tool for test management, test data management and test code generation
- Implemented in TSL, Test BASIC, SQA BASIC, AutoTester
- For AS/400, VB and PowerBuilder SUTs

© CISS Ltd 1999

Test Data

#Screen	IMOM00 =>IMACS/400 Main Menu							
#	TEST	TESTSTEP	AUTOSTEP	RUNDATE	ACTIONKEY	TESTACTION	DFMOPT	
#							B L=2	
IMOM00	POLGIT01	1	1		ENTER		21	
#Screen	IMOM21 =>Underwriting Maintenance						===>	
#	TEST	TESTSTEP	AUTOSTEP	RUNDATE	ACTIONKEY	TESTACTION	DFMOPT	
#							B L=2	
IMOM21	POLGIT01	1	2		ENTER		2	
#Screen	IMPL02 =>Register New Policy						Renewal/ New Policy	Policy No.
#	TEST	TESTSTEP	AUTOSTEP	RUNDATE	ACTIONKEY	TESTACTION	DFOPTN	PLPLNO
#							B L=1	B L=7
IMPL02	POLGIT01	1	3		Enter		N	
IMPL02	POLGIT01	1	4		Enter			
IMPL02	POLGIT01	1	5		SHIFT+F2			

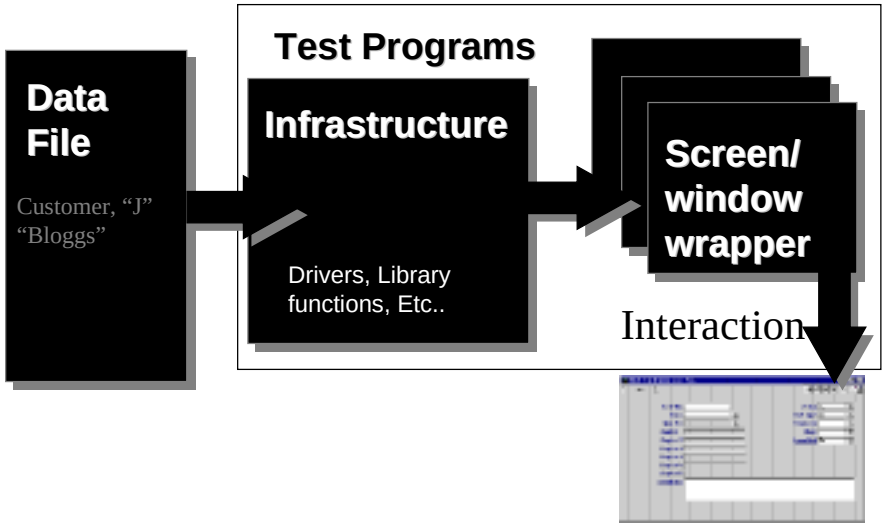
© CISS Ltd 1999

How It Works

- Driver reads first line of data
- Passes data to test program for the specified screen/window
- Test program plays actions
- Driver reads next line
- …..etc.
- Navigation in data

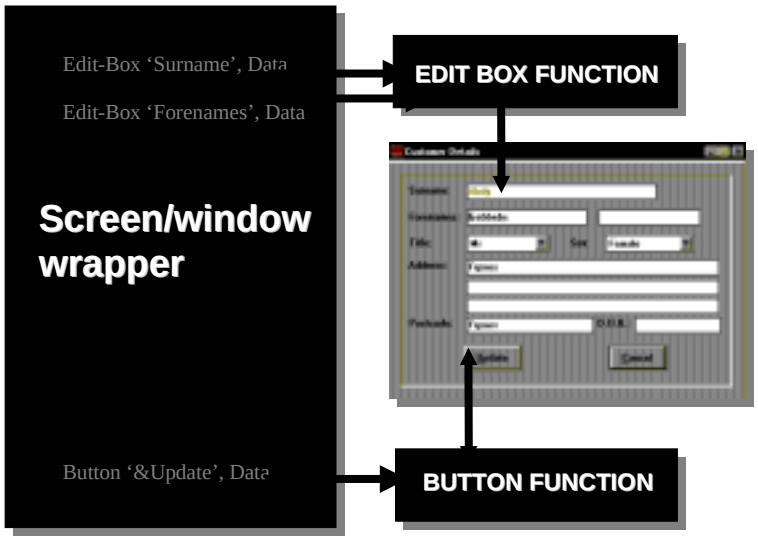
© CISS Ltd 1999

Architectural Structures



© CISS Ltd 1999

Architectural Structures



© CISS Ltd 1999

Test Program Generation

- UI object Wrappers for each object *type*
- Screen/window descriptions used to generate test programs (screen wrappers)
- Close to 100% test
- Generates TSL, Test BASIC, AutoTester, SQA BASIC

© CISS Ltd 1999

SUT Model

- Screen/window Descriptions loaded into TestDev from source code or design document
- Descriptions used to generate test programs
- Descriptions used to generate test data layout
- Gives model of the system

© CISS Ltd 1999

Test Object Re-use

- Business object level architecture tests are at business language level
- Screen/window level architecture tests at lower level
- *Templates* of business tasks reused in tests
- Not OO re-use at the moment - no inheritance

© CISS Ltd 1999

Other Features

- Library functions
- Checks
- Key translation
- Recovery and error trapping
- Reporting routines
- Performance monitoring
- Test data management support

© CISS Ltd 1999

The Advantages Of Advanced Architectures

- Tests specified by test analysts in data files
- Object-to-Object ratio reduced
- Reduced maintenance
- Reduced testing of tests
- Infrastructure components can be used to test other systems

© CISS Ltd 1999

Conclusion

- Wrappers for screen/windows and wrappers for each type of UI object - wrappers within wrappers
- Gives Object-to-Object Reference ratio= 1:1
- *Screen/window level architectures mean test program generation*
- *Generation is from SUT model*

© CISS Ltd 1999

Advanced Automated Testing Architectures

John Kent M.Sc.
Email: john_kent@compuserve.com
29 Abbey Road
Horsell
Woking
Surrey GU21 4PQ
Tel +44 1483 768 992

Abstract

Many organisations have encountered difficulties when attempting to create realistic automated tests using modern, script-based automated testing tools. As a consequence, the way these tools are used has begun to change and more effective automation *architectures* are beginning to be developed. Test automation is now often *data-driven* with actions and navigation in the test data. Test programs are *wrappers* for either business level or screen/window level objects. This paper reviews progress in this field and offers an architectural approach that may indicate future directions. The architecture is based upon *screen/window level wrappers* and *UI object wrappers* which allow for test program generation from *models* of the system under test.

The Promise of Test Automation

Automated testers are in a fortunate position in that they can have their tests running overnight while they are at home with their feet up, having a beer. Automated testers work when they are not at work. That is a productivity level which no one else can match and a great reason for being in this exciting area of software development. A large number of tests that used to have to be painstakingly done manually at a snail's pace and probably involving many testers, can now be done by an automated tester whilst he or she is at home sound asleep.

This is the promise of automation: greatly increased productivity during test execution. Software is becoming so complex that the cost of adequately testing every release of a product manually is becoming prohibitive for many development organisations. Good software is extremely expensive to create and testing it 'fully' could cost just as much again. Because testing does not have a visible deliverable in the way that development does, this is a cost that many organisations are not prepared to meet. What is needed to help test our software properly is a great increase in tester's productivity. The industry needs software to do the repetitive, boring test execution tasks and the vendors have not been shy in supplying the basic tools to do this.

Where Are We Now?

It is often a good idea to ask "where are we?" and "how did we get here?" Let's do that here for automated testing.

Has the promise of test automation been fulfilled? Unfortunately, no. The promised increased in testing productivity has been attained only by very few of those who have taken up the challenge of automated testing. This has been for a variety of reasons. Test automation is a very young branch of computing and it is easy to get it wrong. Automated test tools are sometimes believed to offer 'testing at the press of a button' and in extreme cases they are even thought of as a way to squeeze testing schedules even further. These are complete misconceptions which have not always been discouraged by the tool vendors. Many people feel that this lack of success has been due to use of the wrong tools and there has been a lot of noise about tool evaluation in the testing community. But surely we can't all be using the wrong tools, can we?

Automated testing can give real benefits but it involves an up-front and ongoing investment and the software development industry is slowly coming to realise this. Rather than simply a way of squeezing testing, automated testing should be seen as a way to do more thorough testing[1].

Many of the problems with automated testing are primarily due to the *approach* adopted rather than the fault of any particular tool[1].

The test automation community is coming to realise that the structure or *architecture* of the automated test programs and data has a major influence on the success or otherwise of an automation project[1, 2, 3]. Building automated tests with certain structures (or the lack of them) is doomed to failure within the resource constraints of most software projects. This paper explains why certain approaches don't work and describes the new architectures and the key factors in building an advanced architecture.

How Did We Get Here?

Automated Testing of Mainframe Systems

Tools for automated testing of user interfaces were first built for text-based, mainframe systems and were called 'Capture/Playback' (or 'Record/Playback') because recordings of each screen were captured when the user pressed the Enter key (or a function, 'F' key). These could then be used to play the data back into the system in order to retest the software. In a mainframe system, the user communicates with the processor (and thus the software to be tested) only when he or she presses Enter or a function key. In between each press of the Enter key, data can be input into the terminal but it is not sent to the processor until the function key is pressed. Thus the mainframe text-based automated tools did not actually record user actions, they recorded terminal/processor conversations. The tests were recordings of what was sent to the processor on pressing the Enter key and what was sent back by the software under test.

These tools were 'true' record and playback – as we will see later tools were not. The better tools would stop the test on a difference between the current run and the recording and allow you to accept the changes or record some more screens so that you could, for example, change incorrect reference data. Other tools would simply terminate the test run when a difference was found.

Although some of these tools were quite basic, they could be made to work, and work well. The author was involved in a test project on a mainframe system at an insurance company in the late 1980's where recordings of over 100,000 screens were successfully used to find bugs during regression testing.

Automated Testing On PCs

With the introduction of PCs and then GUIs, the user/software interaction became much more complex. With PCs, each user has his or her own processor rather than having to share one processor with many others, as with mainframes. This means that on PCs the user/software interaction is much closer. For example, in a GUI the application software is invoked not just when the user presses Enter—a GUI system can respond to any key press or mouse click or even just putting the mouse over a particular part of a window. GUI software is said to be *event-driven* because the application responds to a variety of user actions including mouse clicks, mouse movements and key presses—i.e. it is driven by user created events. Because of this closer user/software interaction, automated test tools could no-longer simply play-back the text on a screen; rather, they needed to be able to play-back the users' *actions*. The only way to do this is to have a list of actions which are performed on the SUT (System Under Test). These lists were called *Scripts* and thus the tools became *script-based*.

How Scripted Tools Work

Scripts contain actions such as *Type 'Smith' in the surname text box* and *Click the OK button.* In Visual Test, for example, a script might look like this:

```
Play "Smith"
WButtonClick("&OK")
```

You can record scripts with most of the test tools on the market. You set the test tool into *record* mode and then perform actions on the SUT. The recording will consist of a list of your actions.

These scripts are actually written in programming languages and you can write scripts manually as well —this is often called *Scripting* but perhaps ought to be called programming, because that is what it really is—more on this later.

Object Identification

Early on it became apparent that the tools needed to 'understand' the objects in the user interfaces (called *controls* in Windows—e.g. text boxes and list boxes) and to be able to interact with them quite closely. Selecting an item in a drop-down combo could be recorded as a series of mouse clicks:

```
Play "{Click 211, 55, Left}"
Sleep(7.168)
Play "{Click 184, 134, Left}"
```

but this is not very helpful or maintainable. It is better to have something like:

```
WComboItemClk("@1", "Mr")
```

which means select the item "Mr" in the combo box identified by the symbolic name "@1". It is equally important that the automation tools can retrieve data from these controls as well as input it.

The tool vendors have spent a lot of their time improving what some of them call 'object recording' like this. The automated test tool you use should include a library of functions you can call from the test tool's programming language which will interact with the objects in the system you are testing; for example, PowerBuilder's Data Windows or non-standard Visual Basic controls. The tool vendors have to produce these libraries for any new, non-standard controls. Some tools have generic functions which interact with non-standard controls but this functionality is often limited.

Automated Test Methods

Because these tools come with their own programming languages, they can be used in a variety of ways and that is what this paper is really about—how do you use the tools for the best results?

Paradigm Lost: Record/Playback

Automated testing tools are often called 'Capture/Playback' or 'Record/Playback'. It sounds great doesn't it? I've heard it a thousand times: "…all you have to do is simply record your tests and replay them whenever you like." It is a sad fact that this is not the whole truth. Record and playback looks great as well, but it only really does what is it is supposed to do in one situation: the tool vendor's sales demo. It is the ideal sales feature—you will see a great demonstration of a robot doing the work a highly paid technician used to do. It's simple and you don't need expertise to do it.

There is a growing body of opinion in the test automation community that recognises that it is not practical to create automated tests by simply recording them with script-based tools[2, 3, 4, 5, 6]. Recording does not work for a number of reasons, the most important of which is that re-runnable tests must cope with what *may* happen, not simply what *has* happened. For example, one of the objectives of running tests against a system is to find errors that were not there before. The automated tests must have code that recognises errors and this must be *programmed* into them. A test cannot simply be a recording played back into the SUT; rather it is an *interaction* between the test tool and the SUT. You can use recording as a way of helping to create automated tests but you will have to take them and modify them by programming. Record/Playback tools are not really Record/Playback at all, they are Program/Playback tools.

Recording also produces unmaintainable scripts that consist of a long list of references to objects on your user interface. For example, I recorded typing the above sentence with Visual Test and this is what the recording looks like:

```
CurrentWindow = WFndWndC("Microsoft Word - QW99Pap.rtf", "OpusApp",  FIND_AND_MAX,
Timeout)
Play "{Click 538, 0, Left}"
Sleep(2.374)
Play "{ENTER}Recording also pr"
Play "oduces i{BS}unmaintaai{BS}{BS}inable scripts"
Play " that"
Play " "
Play "consist of a long list of references to objects on you ue{BS}ser interface"
Play "."
```

The '{BS}'s are where I have typed a backspace. I'll try to play this back (it should type the next line) and see what I get:

Recording also produces unmaintainable scripts that consist of a long list of references to objects on your user interface.

Brilliant, so it works. So now I'll underline the word *unmaintainable* here and italicise it by selecting it with the mouse and clicking the Underline and Italics buttons. This is what I get when recording with Visual Test:

```
Play "{BtnDown 409, 136, Left}"
Sleep(5.232)
Play "{BtnUp 513, 137, Left}"
Sleep(2.106)
Play "{Click 302, 53, Left}"
Play "{Click 320, 53, Left}"
```

I tried to replay this and eventually, after some editing of the script got it to work. I had to remove the first sleep statement in order ensure that the word was selected and of course I took the underline and italics off in order to return to the initial state. It worked after editing, but look at this script. Not only is it unpleasant code, it is also unmaintainable (well, at least very difficult to maintain) because it contains XY co-ordinates which makes it almost impossible to identify what it is doing and to which controls. More importantly for re-running this, the SUT—in this case Word 97—has to be exactly the same state as when I recorded. That means the document should be open with the word *unmaintainable* in exactly the same place. Whilst this is not impossible, it is rather difficult and would be even more so in the middle of a large automated test. What is needed is a different approach here. For example, it may be better to select the word using the 'Find' dialog box (Edit|Find menu item) in order to test the Underline and Italics buttons.

Another reason that record/Playback does not work is that the script can easily get out of sequence with the SUT. The script may need to wait for a window to appear—if it doesn't appear in time the script may just carry on clicking and keying anywhere. 'Wait' conditions may need to be programmed into the script.

If an error is encountered during a test run it is likely that the whole test schedule will come to a stop. It is indeed ironic that, although the objective of automated testing is to find bugs, if the SUT behaves in an unexpected way—i.e. there are bugs—then these automated tests will not work.

Finally, with Record/Playback you end up with data hard-coded into your test program … which we know from software engineering is not a good approach to building software[6].

Record/Playback may be OK for knocking together short, simple demonstrations but for large, real-life automated regression tests it is a non-starter. On the old, text-based mainframe tools it was possible to create replayable tests by recording hours of user input from more than one terminal. Try recording just one hour of input with one of the modern GUI script-based tools and see how well it replays. Without knowing anything about your system I can guarantee that it will not run all the way through. Another enlightening exercise is to try to fix the recorded script in order to see what you would have to do get it to replay correctly and how long it would take you.

Disadvantages of Record/Playback (script-based tools):

- Does not work for realistic tests
- Tests need to expect the unexpected
- Replays can get out of sequence with SUT
- Recordings cannot recover from an error
- Scripts are unmaintainable jumble of object references
- Test data is hard-coded

Test Translation

Once you accept that Record/Playback is primarily a sales gimmick for script-based automation tools, you have to accept that you will need to do some programming in order to create automated tests. You may use Recording as a starting point for these programs (to give you the basic object references), but no more than this.

An alternative approach to Record/Playback is to 'translate' manual test procedures into automated test scripts written in the test tool's scripting language. Each step in a *manual* test procedure is the basis for one or more automated test scripts. This is the approach that many automated testers originally took, is still widely used, and is about as advanced as most test tool vendors get in their understanding of automated testing. The 'Test Translation' method can work but it has some fundamental limitations. It ultimately can become a victim of its own success in that the more tests that are automated, the more programming and maintenance that has to be done. If you have 50 windows in the GUI system you are testing, you may wish to create 500 tests for a system test. So now the development group have 50 programs to maintain and you have 500. Say that you get to be very good at automated testing; you may create 1000 automated tests. When you wish to test the next release of the system, the development group still has about 50 programs to maintain but now you have 1000.

The 'Test Translation' method leads to large numbers of automated test scripts which have test data hard-coded into them. You wouldn't hard-code data into programs so why hard-code test data into automated tests? Automated test scripts need to be developed, maintained and *tested* in exactly the same way as any other software. Thus the Test Translation method becomes a victim of its own success. In the example above you would be testing a system with 50 programs using 1000 programs which would require maintaining and testing (more on this later).

Disadvantages of the Test Translation method:

- Test data is hard-coded
- Script to test procedure ratio is 1:1 or worse—the more tests, the more scripts to maintain
- Many test programs (scripts) to maintain
- Many test programs (scripts) to test

A Note on Horses

Well, not really horses but horses for courses. Microsoft have used the test translation method to good effect for some time. This is due to many factors but there are perhaps three major ones. First they have a great deal of resources to throw at testing and have developer-to-tester ratios[7] of about 1:1. Second, they have an iterative development life cycle[7], repeating tests many times, and so it is more cost effective to use Test Translation than to do manual testing. Third, and perhaps more importantly, they are testing different types of applications from most of the rest of us. Most testers in corporations are employed to test database systems. Testing an insurance database for example, requires a very different approach from that used for testing a Word Processor. The main factor governing insurance database system testing is that lots of test data is required which is used in Time Travel tests because these systems manipulate data over time. This Time Travel involves—resetting the business date for different parts of the system test. It is very different from testing the functionality of a software product such as a word processor.

Generally the testing world needs a better way to automate tests than this.

Reasons For New Approaches:

I have listed some of the reasons for taking a new approach above; but there are more, so I will list them all here:

Automation Not Fulfilling Its Promise

As we have seen, lots of organisations buy automated testing tools, but the number of those who have benefited from them is low and the number who have used them to fulfil the promise of automation—

large automated tests—is even lower. Any new approach to automated testing must provide the facilities to be able to:

- Run large numbers of repeatable tests
- Run those tests unattended

As I mentioned above, when re-running a test you must cope with what *may* happen, not simply what happened when you ran the test the last time. Many testers build automated scripts in a way that is in conflict with what they are trying to achieve. They are looking for unexpected behaviour in a system, i.e. bugs, and yet they build automated tests that will stop (or worse) when something unexpected actually happens. This means that an automated test script could stop at any point and the tester cannot rely on it completing. One of the greatest benefits of automated testing will be lost—that is, having your system tested while you are at home tucked up in bed. One of the primary objectives of automated testing should be to create automated tests that can be run unattended. When your automated test finds that the SUT has behaved in an unexpected way, it should be able to *recover*. Recovery means getting the system back to the starting state (closing all the children of an MDI window for example) and then continuing with the next test.

Testing The Tests

When a new release of a system is handed to the test team and the automated tests are run against it, two things are being tested. Of course the SUT is under test as you would expect but, importantly for us here, the automated tests themselves are also being tested. In the real world most failures in automated tests are due to bugs in the test code or test data—not in the system being tested. There is no way around this; it is a fundamental fact about automated testing. We as automated testers need to embrace this fact rather than ignore it, hoping it will go away. Automated tests need to be built in a way that takes this fundamental truth into account. A major problem with the Test Translation approach is that the more tests you have, the more tests you have to test.

This is so important that we'll summarise it here:

When you run automated tests against a new release of a system you are testing:

- The system under test
- The automated tests themselves

The Importance of Maintenance

Automated testing is a bit like trying to hit a moving target. The target, the SUT, keeps changing. There are changes between subsequent releases as well as changes during the development of a release. This is another fundamental fact about automated testing that we need to embrace[6]. Automated tests need to be light on their feet so that they can be changed quickly in order to cope with their constantly changing target.

Object-to-Object Reference Ratios

The key factor in the maintenance of automated tests is the number of times the they refer to an individual object on the user interface. The more references you have to User Interface (UI) objects, such as test boxes and combo boxes in Windows, the higher your maintenance burden (and incidentally your 'testing the tests' burden). This is the *Object-to-Object Reference Ratio*—the object to the number of times it is referenced in the test automation.

Record/playback may have references to an individual object many times as the user interacts with that object, thus its Object-to-Object Reference Ratio is one to many. Test Translation also scores low here because the test scripts are based upon the tests themselves and an individual user interface object may be referred to by many tests. With either of these methods increasing the number of tests your

automation performs is likely to make the Object-to-Object Reference Ratio more unfavourable. In the example given above, where the number of tests is increased from 500 to a 1000 the Object-to-Object Reference Ratio could, if all of the tests use that UI object, go from 1:500 to 1:1000.

An advanced approach tries to minimise the number of times an object or control on the user interface is referenced in the automated tests—bringing the Object-to-Object Reference Ratio as close to 1:1 as possible, thus minimising maintenance and testing of the tests.

Scripts Are Really Programs

The term *script* is rather unfortunate. Automated test *scripts* are really *programs*[3, 5, 6]. A script is a list of step-by-step instructions for something, but automated test scripts need to contain conditional processing (if statements and loops—do whiles etc.). An actor in a play has a script which is a list of sequential instructions. There aren't any plays where the script says "if the audience look bored skip to Act 3." In a play, one scene follows another, whereas an automated test should make *decisions* which are dependent upon the behaviour of the system being tested. The test cannot simply be a recording played back into the system, rather it is an *interaction* between the test tool and the SUT. The SUT may behave differently on the next test perhaps due to an error or a change and the script should generally be able to handle new responses from the system. Thus automated test scripts are really *programs* and have to be developed and maintained in the same way as any other software.

Once you accept that automated test are really programs, you, as testers, should not have difficulty with the idea that they need to be tested. You will know that the more you test your automated tests, the more confident you can be that they will work under all possible conditions.

Redefining terms:

- Test Program—the software part of an automated test, usually written in the test tool language (e.g. Test BASIC, TSL, SQA BASIC).
- Script—a list of sequential actions to be performed in order to exercise a test. Often written using a spreadsheet, word processor or test management tool.

A Software Engineering Approach To Test Automation

Test automation has been sent down some blind alleys by the Record/Playback paradigm, encouraged along by the idea that automated tests can be 'scripts'. Test automation is a software development process and should conform to the disciplines of software engineering. The part of the automated tests written in the so-called scripting languages provided by these tools are really *programs* and must be *developed, maintained and tested* in the same way as any other software.

We need to take a software engineering approach to test automation. We certainly will not improve things if we start writing test programs from scratch each time we want to automate the testing of a new system. No, we should be taking what we already have and improving on it. We need a structured approach that aims to get the *Object-to-Object Reference Ratio* as close to 1:1 as possible. We should also have a library of functions that plug into the test automation in the same way that C++ developers get Microsoft Foundation Classes.

Advanced Architectures

In recent years a number of test automation architectures[3, 4, 5, 6, 8, 10, 11, 12, 13] have been proposed. The whole thrust of advanced automation techniques has been to move the creation of automated tests to higher levels of abstraction. These architectures allow test analysts to specify tests without having to know anything about the automated test tool and how it interacts with the SUT UI objects. The test programs deal with the low-level interaction—the test analyst uses symbolic names for objects used in the tests. Advanced architectures also move the *Object-to-Object Reference Ratio* closer to 1:1 than

with standard architectures by referencing objects in test programs rather than in the tests themselves.

An advanced automated test script architecture should provide:

- An Object-to-Object Reference Ratio as close to 1:1 as possible.
- A test program structure that promotes easy maintenance.
- A test program structure that is easy to test (testing the tests).
- Ability to recover from unexpected conditions.
- Basic infrastructure components.
- Tests specified at higher level of abstraction—using symbolic UI object names.

Framework Based Architectures

One way to improve upon the Test Translation Architecture is to use a Framework Architecture. In a Framework[6, 8, 9]—called *The Functional Decomposition Method* by Zambelich[3]—system functionality is automated by functions held in the test library. Tests can then be created by calling these functions from a test program which passes parameters to them. The functions can automate low- or high-level tasks. They can be business task based—for example, Add_Cust("Smith", "John")—or system screen/window based—for example, OpenFile("xxx.doc"). These functions act as *wrappers* around what they are automating. 'Wrapper' is an Object Oriented term which is used loosely here to mean a module that handles all the interaction with the thing it 'wraps' around. Thus the Add_Cust function is a wrapper for the process of adding a customer to your SUT. Similarly the Open_File function is a wrapper for the open file dialog box. Individual objects on the open dialog box are only accessed by the Open_File function. The tests themselves do not actually interact directly with the object only with the function.

Data-Driven Architectures

An architectural feature that has been around for a while is that of *data-driven* tests where the test data is held in a separate file which is read and interpreted by the automated test code. A data-driven architecture reduces the burden of maintenance and 'testing the tests' because functions are repeatedly used to interface with the SUT; thus the Object-to-Object Reference Ratio is closer to 1:1 than with Test Translation methods.

In early data-driven Architectures all of the navigation was handled in test programs. The test programs would navigate to a screen, read the first line of data, input it, read the second line of data and input it and so on until all of the data had been input. The program might then go to another screen, open up another file and do the same thing. Thus the test data was just that—data. The sequence of actions was governed by the test program.

Data Driven Architectures With Navigation Extracted From The Test Programs

The next step in the development of an automated testing architecture was to remove the navigation from the test code and move it , or at least some of it, to the test data[3, 4, 5, 10, 11, 12, 13]. The test programs are instructed what to do by the test data in the same way a manual tester is by a test procedure document. This means that it is possible to put behaviour into a test data file. Thus test data files can control the behaviour of the SUT and navigate around it as well as simply providing data to be input. Zambelich calls this *Totally Data-Driven Automated Testing*[3].

With this approach the test programs are no longer tests at all; rather they are wrappers that interface the tests to the system under test. The test data becomes the test script in that it is a list of instructions that drive the software.

This is really a combination of basic data-driven and framework architectures. The tasks are automated by wrappers that have parameters—test data—passed to them, only this time the test data is held in files rather than in a test program. As with framework based architectures this can be done at the *business object level*—for example, Buwalda's ActionWords[10] and Ottensooser's Data Independent Test Scripts[13]—or the *screen/window* level, as in the author's approach[4, 5] which will be discussed in depth below.

Business Object Level Architectures

In Business Object Level Architectures, business *tasks* are automated. Test data for an automated test of this type might look like:

Create_Cust "Smith", "John", …
Create_Account "Smith", "John", "Cheque",….
Debit_Account "Smith", "John", "$1000",….

Thus much of the navigation is in the test data. As you can see from the above example, test data in Business Object Level Architectures is at the business language level and therefore understandable by end users which is a great advantage[10]. This approach also gives a lower Object-to-Object Reference Ratio than for Test Translation because the UI objects are only referenced on a per business task basis.

Screen/Window Level Architectures

Figure 1 shows the Screen/window Level Architecture approach. In this architecture there is one test program that deals with each screen/window in the system—it acts as a *wrapper* for that screen/window—it handles all of the input and output (getting values) from it. The test data is held in a separate text file and is passed to each test program as it is needed. Thus the navigation is in the test data. This structure gives a ratio of Window-to-test program of 1:1, and thus also an Object-to-Object Reference Ratio of 1:1, so if a change is made to a window in the SUT, only one program needs to be changed and tested.

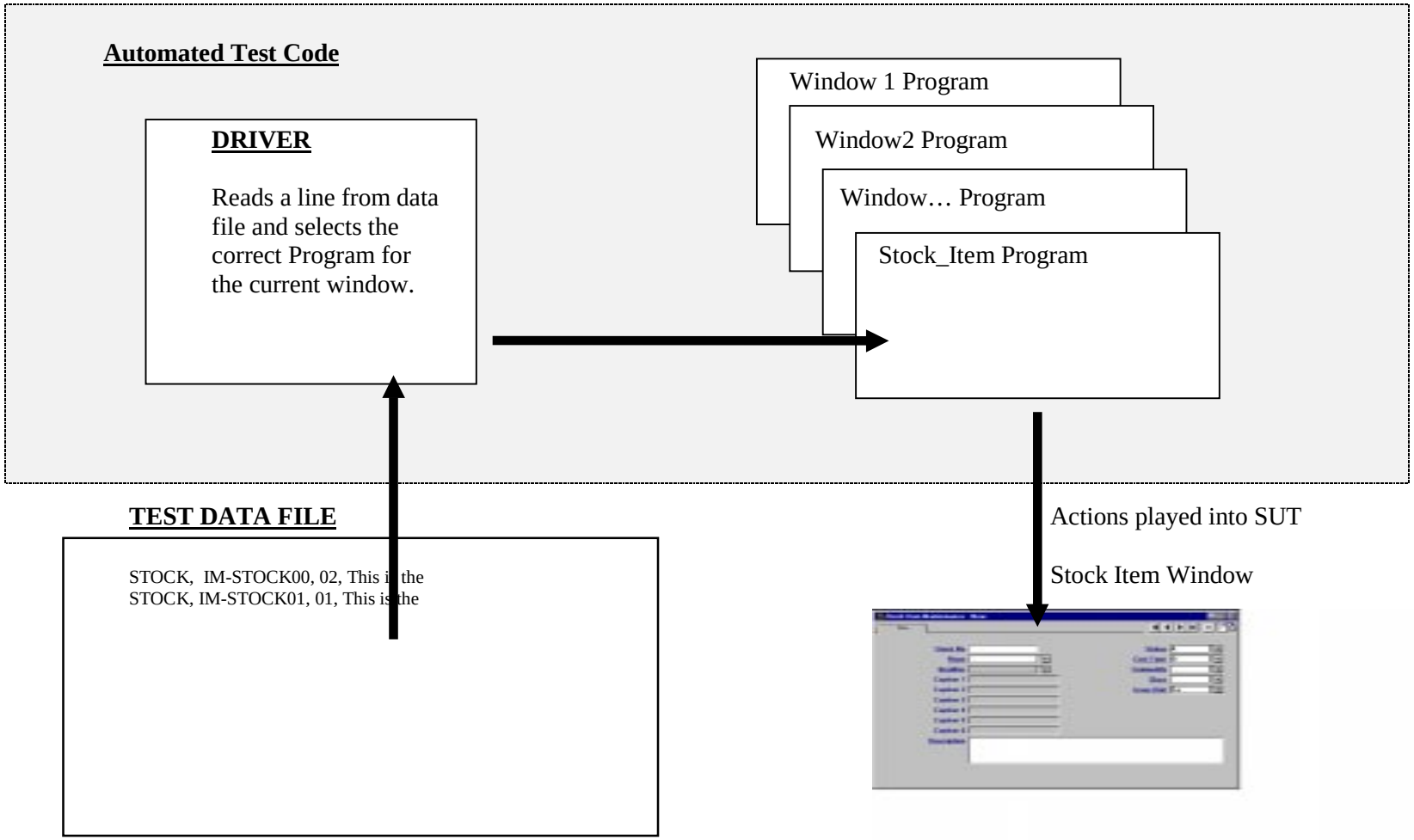


Fig. 1 Data-driven tests: **One test program per window in the SUT in this case.**

An Advanced Automated Test Architecture

The author has been developing automated test architectures since 1992 for both GUI and textbased SUTs using script-based automation tools. The author has developed a tool called *TestDev* which is used to implement this architecture and incorporates test management, system modelling and test code generation facilities (more on this later). The architecture incorporates all the advanced architectural features we have discussed so far.

- Object-to-Object Reference Ratio is 1:1 or 1:2—minimising maintenance and testing
- Completely data-driven including all navigation
- All test and user actions are in the test data
- It takes the Window/screen-level approach
- Each *class* of UI object is 'wrapped' in just one (or sometimes two) functions
- It has infrastructure components; e.g. recovery, etc.

This architecture is implementation-independent—i.e. it can be implemented with any script-based test tool.

Architectural Overview

The Data

The data needed for each test is held in a comma-separated file created using Excel spreadsheets. Figure 2 shows an example of part of an Excel spreadsheet with test data for an actual test carried out recently on a textbased AS/400 system. Although this example is for a textbased system, automated tests of GUI systems have been done using the same principles of this architecture but implemented in slightly different way.

#Screen	IMOM00 =>IMACS/400 Main Menu							
#	TEST	TESTSTEP	AUTOSTEP	RUNDATE	ACTIONKEY	TESTACTION	DFMOPT	
#							B L=2	
IMOM00	POLGIT01	1	1		ENTER		21	
#Screen	IMOM21 =>Underwriting Maintenance						===>	
#	TEST	TESTSTEP	AUTOSTEP	RUNDATE	ACTIONKEY	TESTACTION	DFMOPT	
#							B L=2	
IMOM21	POLGIT01	1	2		ENTER		2	
#Screen	IMPL02 =>Register New Policy						Renewal/ New Policy	Policy No.
#	TEST	TESTSTEP	AUTOSTEP	RUNDATE	ACTIONKEY	TESTACTION	DFOPTN	PLPLNO
#							B L=1	B L=7
IMPL02	POLGIT01	1	3		Enter		N	
IMPL02	POLGIT01	1	4		Enter			
IMPL02	POLGIT01	1	5		SHIFT+F2			

Fig. 2 Data-driven tests: EXCEL spreadsheet Test data for a textbased system.

The lines in the spreadsheet that start with '#' are comments and ignored by the automation program's

outlines. All other lines are actual test data and instructions. The comments in this case are *screen descriptions* which have three lines. The first line in the screen description has the screen name and the field labels on the AS/400 screens to be tested. The second screen comment line contains the internal AS/400 program names of the fields (e.g. "DFMOPT", "PLPLNO" and "TESTSTEP"). The third line contains B or O for each field meaning "**B**oth input and output" or "**O**utput only" for each field. This line also has the maximum length of the field (e.g. "L=7" means length is 7 characters maximum). The comment line's only function is to give information on each screen to the person creating the test data.

The first seven columns of the actual test data are standard. Column 1 contains the name of the AS/400 screen to which that particular line applies. Columns 2, 3 and 4 contain the test name, test step number and automation step number. Column 6 contains the ActionKey—this is the function key that is to be pressed after all of the data for that line has been keyed in or checked. For example "ENTER" or "F3". Column 7 contains the TestAction which can be set to perform actions like 'CHECK' which checks the values in the spreadsheet against values on the screen.

How The Test Automation Works

This data is read by a driver program which calls the test program for the screen named in column 1 of that line of the data. It passes all of the data to this program which completes its tasks and returns processing to the driver. The driver then repeats the process for each line of test data (see Fig. 1 for an overview of this process). So, for the data in Fig. 2, the driver will read the first line and call the test program IMOM00 (in this case the test program has the same name as the screen it tests) which deals with the IMOM00 screen—the main menu of the SUT. The IMOM00 test program keys '21' into the DFMOPT fields, presses Enter (the ActionKey) and returns control to the driver. The driver reads the next line of data and calls the test program specified in that line—IMOM21 and so on. Thus, it can be seen that there will be one test program for each screen in the system (this is a slight simplification for the sake of clarity—it's one program per logical screen).

Architectural Structures

Each test program is a wrapper for the screen it tests. It contains all the information it needs on each UI object in the screen/window so that it can do the tasks required of it, such as input test data and check values. Thus the Object-to-Object Reference Ratio is the ideal 1:1.

When a test program runs it does not actually input test data or check data values itself; rather, it calls functions that deal with each type of UI (see fig. 3), These functions are object wrappers for each *class* of UI object. There will, for example, be one function for text boxes that will handle all of the test data input into all text boxes in the SUT.

Thus there are wrappers within wrappers. Wrappers at the screen/window level—the test programs which use wrappers for each type of user interface object—the functions for that object class.

In the large AS/400 test that the test data example is taken from there were only four functions that input or checked test data values—two for each of the user interface object types:

- 2 functions for text fields
- 2 functions for Subfile fields (AS/400s version of scrollable list boxes)

For the whole of a large regression test lasting of over 60 hours of automated testing, there were only four functions actually inputting the test data and getting screen values.

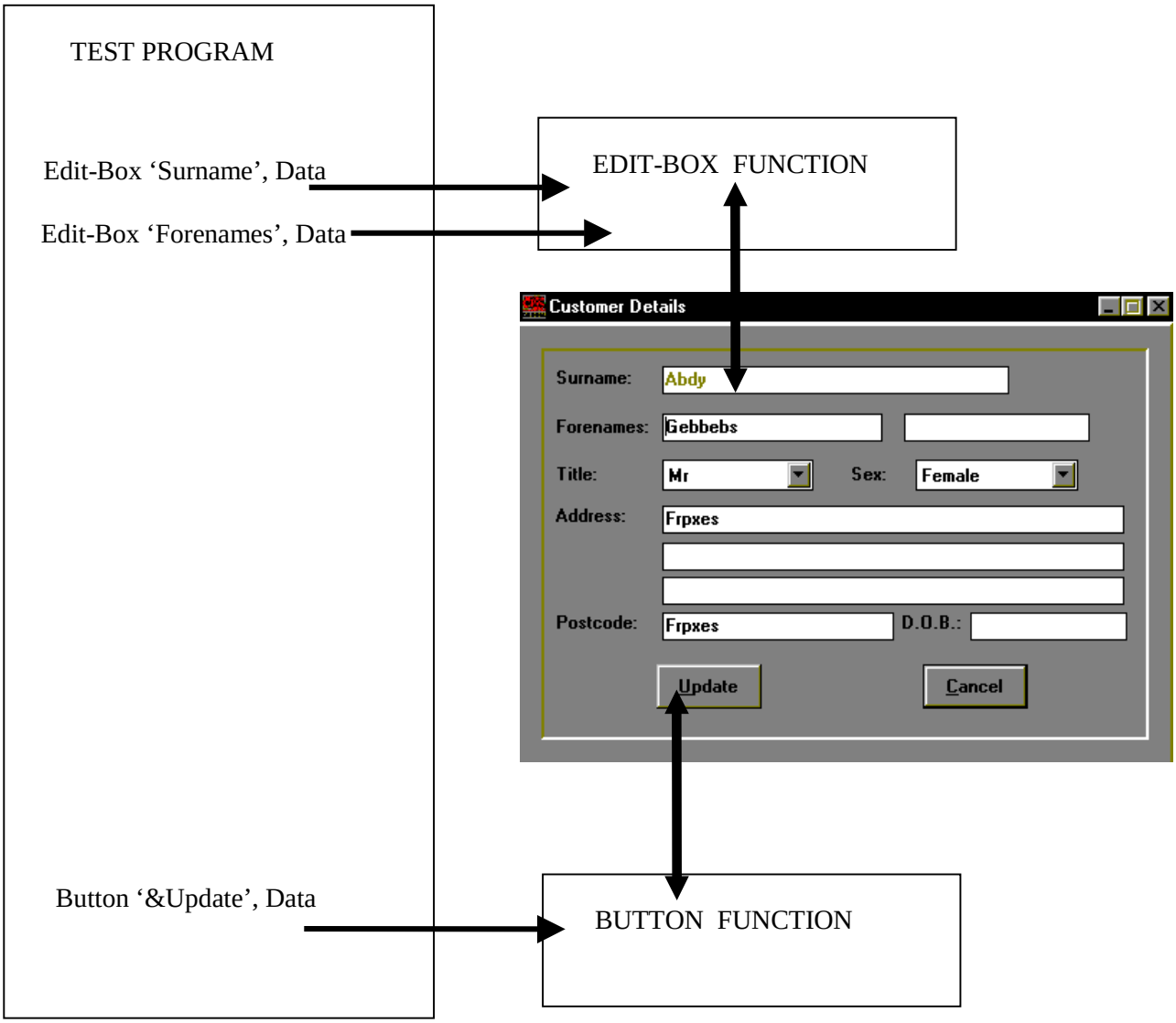


Fig. 3 Test programs call functions for each UI object class.

UI Object Wrappers

The behaviour of UI objects—text boxes, combo boxes, etc.—is well understood because development build the system using these components. Thus generic wrappers for these objects can be built based upon their behaviour. Functionality can be built to get and set their properties and initiate their methods—i.e. do things to an object that a user might do.

The Object-to-Object Reference is still determined by the test programs because they contain the references to each individual object (each instance), but the program code that actually interacts with those objects is held in the function that deals with that object class. Because the input of test data and the checking of results is handled by one (or maybe two) functions per UI class—e.g. one function inputs data into all of the Windows text boxes in the system—a high level of confidence can be attained in these functions. They are repeatedly used and thus tested.

Test Program Generation

A screen/window level approach has been chosen for this architecture because it offers one major

advantage over the business object level architecture. Because the behaviour of the UI objects within a system are defined and known and the behaviour of the screens or windows can be defined, it is possible to generate the test programs for each screen/window from descriptions of those screens. The *TestDev* tool is used to generate the test programs. It does not actually run the tests itself; it generates test code for the chosen automation tool. Currently it has been used to generate Test BASIC(MS-Test and Visual Test), TSL (WinRunner) SQA BASIC (SQA Robot) and Autotester Outlines. The generators are constantly being improved and *TestDev* is now close to generating 100% of the test code and in some cases actually 100%. Some manual programming effort has been necessary in order to cope with unpredictable system behaviour. This unpredictable behaviour is usually in the area of system generated keys (see the section on Key Translation below).

The basic architectural components can be used to test many different types of system; only the actual screen/window programs must be generated for each SUT, but this generation can be based upon descriptions of the SUT that can come from the system's source code, or from automatic inspection of the actual running executable. Alternatively, these descriptions can be derived from system design documents in order to start building test automation for a system that has not yet been delivered. In the AS/400 example, the test programs were generated from the AS/400 screen definition files (DDSs).

Actions At The User Interface Level

Another reason for choosing the screen/window level for this architecture is that all of the UI objects (and therefore their methods) that a user can access on a screen or window can be made available to a tester when creating the automated test data and actions. This includes selecting any button or menu item available. Business Object Level automated testers believe that this level of detail at the test creation stage is too much of an overhead and impossible to automate, but if your test programs are *generated* from the SUT it need not cost you anything to have access to any object in the SUT. The benefit of having all of the possible UI objects available to the test analyst at the test data creation stage is that it gives them complete freedom to perform any action on the system. With the Business Object Level only the business tasks that have been automated can be performed.

Automation Is Based Upon A Model of The System

The screen/window descriptions provide a model of the system which is used not only to generate the test programs, but also to create the spreadsheet format for test data creation. The screen descriptions in the test data in figure 2 were generated from AS/400 source files (DDSs) downloaded onto a PC. The TestDev tool provides system modelling and test program generation facilities as well as test management and test *data* management. It has functionality which loads the screen/window descriptions, generates the test programs for each screen/window and generates the test data spreadsheet format for each screen/window.

The automated tests (the test data) are based upon *test procedures* which are documented within the TestDev database in the same way that manual tests might be. The test procedures consist of *test steps* and the automated test data for each test step is contain in a series of 'AutoSteps' (see col. 4 in fig. 2).

Test Object Re-use.

One of the great advantages of the Business Object Level Architectures is that the test data is created at the business language level and therefore understandable by end users. The architecture we are describing here is built at a lower level—the screen/window level—and so the actions in the test data are at this low level. You can see from figure 2 that the actions in this real test data are at the screen level and quite detailed. This provides the advantage that *all* of the UI objects—and therefore the possible user actions—are available for the tester to specify in the test data but it also means that the test data cannot be specified at the business language level.

Some facilities have been provided in the TestDev tool so that tests can be re-used. Basic business tasks

are automated and the test data is saved off as test objects or *Templates* which can then be used to create tests. This is not re-use in the Object Oriented sense at all—there is no inheritance here; it is really just copying but it currently serves its purpose of allowing a test analyst to create tests by stringing together business level test objects. One of the more important tasks for the future is to build true inheritance into test objects thereby allowing us to build tests at higher and higher levels.

Other Features of The Architecture

Because each component of the architecture has a standard structure, it is relatively easy to include lots of additional test facilities in it. As structured architectures such as this one mature, they start to become very feature rich in terms of test automation features. A good source for information on such features is provided by Bach in *Useful Features of a Test Automation System*[14]. Here's a list of some of the standard features currently included in this test automation architecture:

Checks

It is not enough to test a system by simply inputting data. A test consists of input and output—checks against expected values. This is especially true in test automation where inputting data to a system is often much easier than getting output from it. This can be done using the 'CHECK' TestAction where a screen value is checked against a value in the test data. In figure 4, below, the test automation will check that the risk class field contains the value 'APL' and the Underwriting Year field contains '98'.

#Screen	IMPL02 =>Register New Policy						Risk Class	Underwriting Year
#	TEST	TESTSTEP	AUTOSTEP	RUNDATE	ACTIONKEY	TESTACTION	RISKCL	UWYEAR
#							B L=3	B L=2
IMPL02	POLGIT01	1	23		F3	CHECK	APL	98

Fig. 4 A Check in test data.

Having the expected values for a test in the test data is a much better way to do things than having them hidden in some 'check file', as is suggested by many tool vendors' user manuals. The standard automated check functionality provided by many tools does not lend itself to easy maintenance and is similar to hard-coding the test data.

Key Translation

The architecture can cope with system-generated fields. This usually applies to key fields—for example Policy Number or Customer Number—where the SUT generates a value for them. While writing tests, a test analyst may wish to create a business object (e.g. a Policy or Customer entry) and then later in the test refer to it for some reason. If a business object uses a generated key, the test analyst has no way to of knowing in advance what the value will be. These values can, however, be represented using symbolic names. The names are specified in the test data and, once set, can be used later in the tests. The test programs save the symbolic name and its system generated value—looking it up when-ever it is used in the test data. This is known as Key Translation and is extremely useful for some SUTs.

Recovery

Automated tests should, as much as possible, be built in a fault-tolerant manner so that they can run unattended. If the system behaves in an unexpected way, the test programs should be able to recover and continue. *Recovery* functionality provides a means of dropping a test and getting the SUT back to a known base state so that the next test can be run—e.g. closing all the children of an MDI window. It is used whenever the automated test programs discover that they are in the wrong place—i.e. that the wrong screen/window is displayed and the SUT is in an unexpected state.

In this architecture the SUT is checked at the beginning of every test program in order to ensure that the

screen/window displayed is the appropriate one for the test program. If it is not, recovery is invoked.

Reporting

Large automated tests can generate massive report logs especially if you use the vendor supplied logs which detail events at a very low level. The generated test programs within the architecture described above report events in a way that allows the tester to view results at different levels of detail. The resultant reports can be tied back to the test procedures on the TestDev database. If any AutoSteps within a test step fails then the test step itself fails.

Performance Measures

Because the architecture is screen/window-based it is relatively easy to incorporate performance measuring functionality in the generator so that a standard set of measures can be applied to each SUT action.

Test Data Management Support

TestDev provides support for the test data spreadsheet creation process in the form of EXCEL macros. These provide facilities to:

- Copy screen layouts into test data spreadsheets
- Copy Templates for test object 're-use'
- Roll dates forward (much used recently in Y2K projects)
- Scan data for specified values
- Scan data for occurrences of specified fields

Once you have overcome the major problems of automated testing and actually build and run large automated tests you find different problems. Test data management becomes more important as the failures in the automated tests tend to be related to incorrect test data.

Implementation of The Architecture For GUI Systems

The test data examples given (see figures 2 and 4) are from a text-based system. Most of the actions in this case are specified in the ActionKey field. There is only one ActionKey column per screen in the spreadsheets because the possible user actions for a textbased system of this sort are essentially at the screen/window level—i.e. pressing Enter or function keys. For GUIs this will not do, as we need to set actions for each object on a window, and therefore the implementation of the architecture is different in terms of test data creation though the fundamental principles remain the same.

Conclusion

Test automation is in its infancy and there is a long way to go before we can say that most organisations benefit from it. Although the tool vendors have delivered some helpful new features over the past few years the main problem has been due to the wrong architectural approach when implementing a test automation tool. Test automation needs to be recognised for what it is—a software engineering activity—and well understood software engineering principles apply just as much to the creation of automated tests as to the system under test. Once advanced architectures are fully understood, realistic, large-scale automation will gain wider acceptance.

Test automation is already moving away from simple 'scripting' towards using the test tools to create wrappers for the objects on the user interfaces under test. Test actions have been moved out of the test program code and into the test data. Along with this the actual tests themselves are being specified at a higher level.

This main thesis of this paper has been that architectures that create wrappers at the screen/window level can allow for test program generation to be used to create most of the actual test program code. This can be done because the behaviour of UI objects is well defined and the automation functionality required

can be provided in a wrapper for each class of UI object. The generation is based upon a model of the system which can be derived from source files such as screen or window definitions.

Another advantage of building automation architectures at the screen/window level is that all of the UI objects and possible user actions can be made available to the test analyst at test data creation time. Although this also means that the actions are not at the business object level, test creation can be moved up to this level using test object re-use facilities. This is not yet fully provided for. Once this level is reached, automated test creation will be very similar to the creation of *manual* test procedures except that the format and names of the user interface objects will be derived from the system model.

The key to the effective implementation of new architectures is to them base on a model of the system to be tested. The model can then be used both to generate the object wrappers (the test programs) and as a basis for specifying the test procedures in a format that the automation programs understand. These models can be created from the definitions of the system's user interfaces or can based upon function specifications if the system is still under development..

References

1. Bach, James: *Test Automation Snake Oil*—http://www.stlabs.com/

2. Paul Gerrard: *Automated Testing: Past, Present and Future*—Presentation at EuroStar'98.

3. Zambelich, Keith: *Totally Data-Driven Automated Testing* http://www.sqa-test.com/

4. Kent, John: *Overcoming the Problems of Automated GUI Testing.* Presentations to STAR 1994 and British Computer Society SIGST Dec 1993.

5. Kent, John: *An Automated Testing Architecture*. Presentation to the British Computer Society SIGST July 1997.

6. Kaner, Cem: *Improving the Maintainability of Automated Test Suites*—Quality Week 1997

7. Cusmano, Michael A. and Selby, Richard W. *Microsoft Secrets: how the World's Most Powerful Software Company Creates Technology, Shapes Markets and Manages People*. Harper Collins 1997

8. Arnold Thomas, R.II: *Building an Automation Framework with Rational Visual Test*—ST Labs Report 1997

9. Arnold Thomas, R.II: *Software Testing with Visual Test 4.0*—IDG Books 1997

10. Buwalda, Hans *Testing with Action Words*—From *Automating Software Testing* (Chapter 22)—Addison Wesley Longman (to be published 1999)

11. Pettichord, Brett: *Success with Test Automation*, Proceedings of the Ninth International Quality Week (Software Research) 1996.
http://www.io.com/~wazmo/succpap.htm

12. Thomson, Jim: *Business Object Scenarios: a fifth-generation approach to automated testing*—From *Automating Software Testing* (Chapter 22)—Addison Wesley Longman (to be published 1999)

13. Ottensooser, Avner: *Data Independent Test Scripts*—paper presented at STAR 1997.

14. Bach, James: *Useful Features of a Test Automation System*—printed in Thomas Arnold's *Software Testing with Visual Test 4.0*—IDG Books. Also http://www.stlabs.com/

A State-Based Testing Approach Providing Data Flow Coverage in Object-Oriented Class Testing

Bor-Yuan Tsai*, Simon Stobart,
Norman Parrington and Ian Mitchell

School of Computing, Engineering and Technology
University of Sunderland, UK
E-mail: {cs0byt, cs0sst, cs0npa, cs0imi}@isis.sunderland.ac.uk

*Also of Department of Information Management,
Tamsui Oxford University College, Taiwan

Contents

- **Introduction**
- **Background**
- **The Framework of MACT**
- **A Queue Class Example**
- **Test Cases Generated from MACT**
- **Data Flow Testing**
- **Test Cases Selected Based on Data Flow Testing**
- **Data Flow Testing Employed in MACT**
- **Other Functions of MACT**
- **Conclusions**

Introduction I

A Novel Class Testing Approach

=

A Functional Testing Technique + A Structural Testing Technique

▼ State-Based Testing ▼ State Machines

▼ Data Flow Testing ▼ Occurrences of Data

▼ Implementation State Machines

▼ Test Case Trees

↙ Test Messages ↓ Test Oracles ↘ Data Occurrences

↘ State-Based Coverage ↙ ↓ Data Flow Coverage

Introduction II

Functions of MACT:

1. Test Case Tree Generation

2. Test Message Generation

All Possible Adequate Test Massages

3. Data Flow Techniques Used in MACT

(1) Code Optimization
(2) Anomaly Detection

4. Test Driver Generation

5. Test Result Inspection

Test Messages Selected Based on Data Flow Analysis

(1) Difficulty
(2) Inadequate
(3) Ambiguous

Background

State-Based Testing examines the state values.

State Machines define the set of states and transitions.

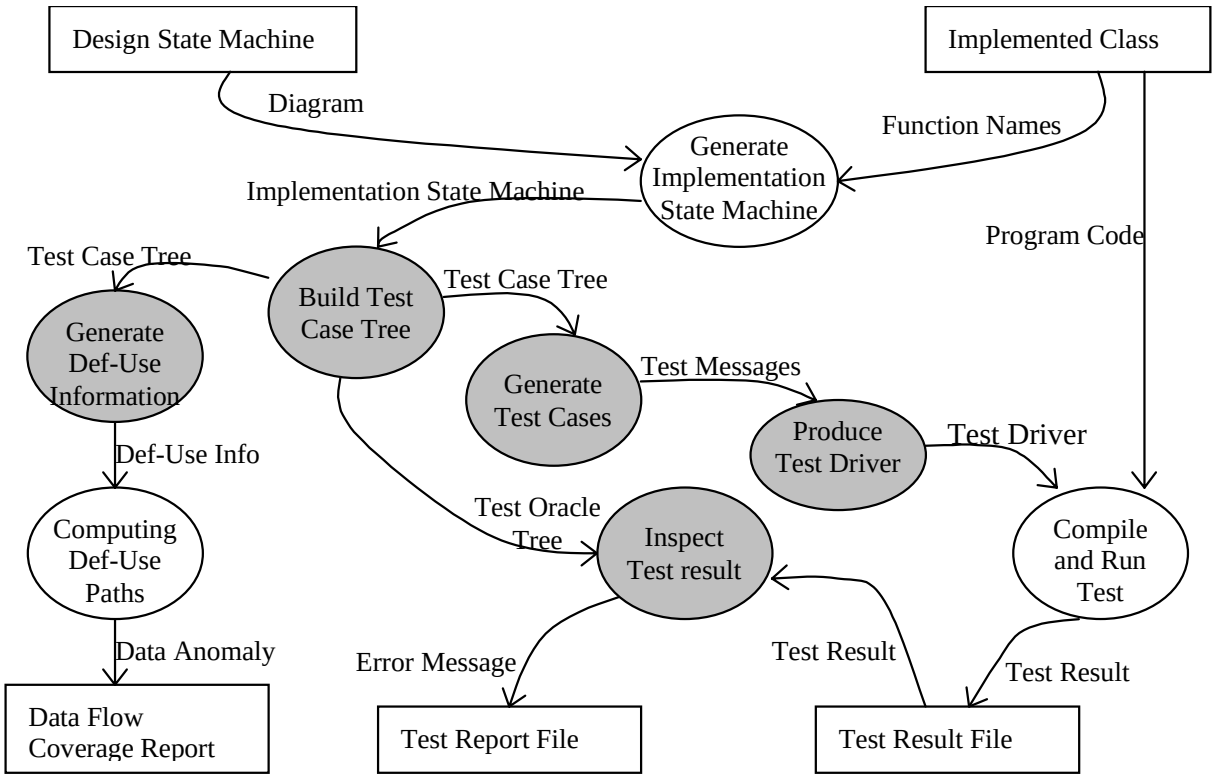
Implementation State Machines reveal the behaviours of the implemented class.

Data Flow Testing exercises paths from definitions to uses of data members.

Test Cases create a starting state, transitions and the expected next state.

Test Messages are sent to test the objects.

The Framework of the MACT Tool



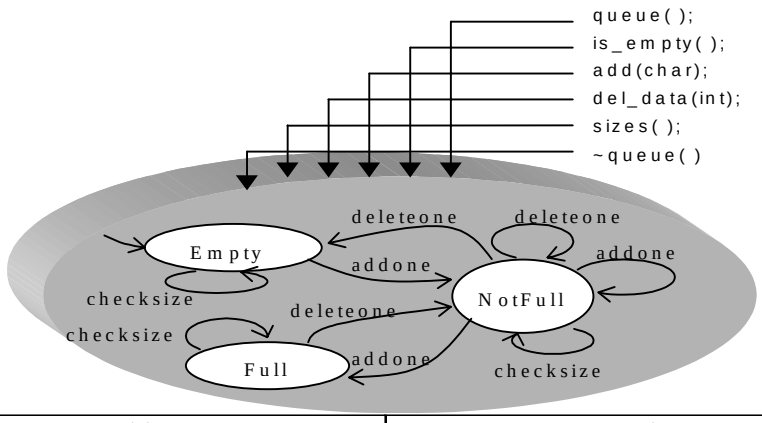
The components of the automated class testing framework (MACT)

The implemented C++ code of the queue class

```
const int Size = 5;
class queue{
protected: char    Q[Size];    // bounded array
           int  f, r;          // Front/Rear index of queue
           int  count;         // a counter of the array
public: queue(void);           // default constructor
           void     is_empty(void);
           int      add(char); // add data to the queue
           char     del_data();//delete data from the queue
           void     sizes(void);// get the size of the queue
           ~queue(void); // destructor
};
queue::queue(void){
     r = -1;  f = 0;
     count = 0;
     for (int i = 0; i < Size; i++){
        Q[i] = ' ';}
}
int queue::add(char data){
     if (count == Size){
        cout << "Not room for adding new data to stack\n";
        return (0);}
     r++;
     if(r==Size){
        r=0;}
     Q[r] = data;
     count++;
     return (1); }
```

```
char queue::del_data(){
     char data;
     if (count == 0){
        cout << "can't delete data from an empty stack\n";
        return('0');}
     data = Q[f];
     f++;
     if(f==Size){
        f=0;}
     count--;
     return (data);
}
void queue::is_empty(){
     if (count == 0)
        cout<<"The queue is empty\n";
     else
        cout<<"The queue is non-empty\n";
}
void queue::sizes(){
     cout << "the size of the queue :" << count << "\n";
     return;
}
queue::~queue(){
     cout << "\nfinishing the class test\n";
}
```

An Implementation State machine of the queue Class



Transition Names	*Function Names*
checksize	is_empty() | size()

The implementation state machine of the class **queue** is defined with finite sets ISM = (**V**, **F**, **S**, **T**) where

V = **{**char Q, int f, r, count**}**

F = **{**queue(), is_empty(void), add(char), dele_data(void), sizes(void), ~queue()**}**

S = **{(Empty**, count=0), (**NotFull**, 0<count<5), (**Full,** count =5)**}**

T = **{** S_0, queue(), **Empty,** (count=0); {[count, r, f, Q]}

Empty, is_empty(), **Empty,** (count=0), {<count>}

Empty, sizes(), **Empty,** (count=0), {[count]}

Empty, add(data), **NotFull,** (0<count<5), {<count, r>, [r++, r=0, Q[r]=data, count++]}

NotFull, is_empty(), **NotFull,** (0<count<5), {<count>}

NotFull, sizes(),**NotFull,** (0<count<5), {[count]}

NotFull, add(data), **NotFull,** (0<count<5), {<count, r>, [r++, r=0, Q[r]=data, count++]}

NotFull, add(data), **Full,** (count=5), {<count, r> [r++, r=0, Q[r]=data, count++]}

NotFull, del_data(),**NotFull,** (0<count<5), {<count, f>, [data=Q[f], f++, f=0, count--]}

NotFull, del_data(), **Empty,** (count=0), {<count, f>, [data=Q[f], f++, f=0, count--]}

Full, is_empty(), **Full,** (count=5), {<count>}

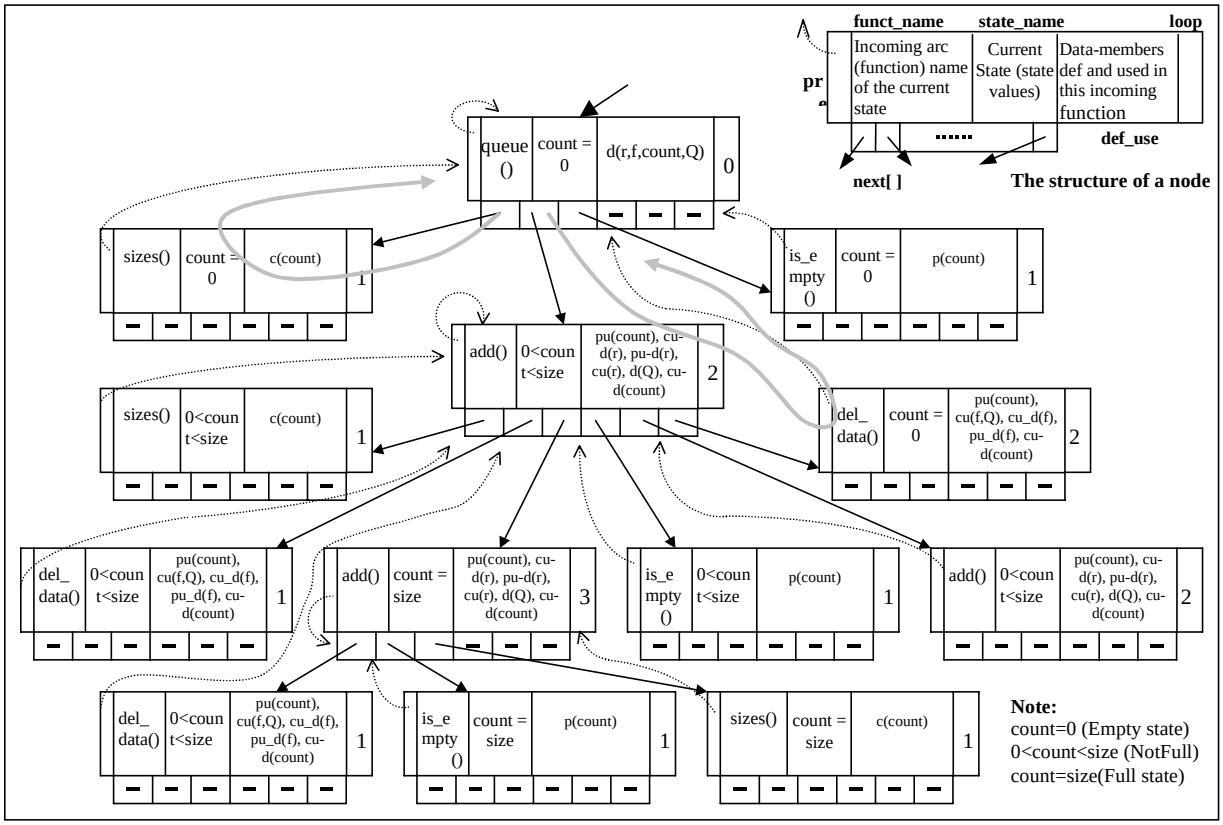
Full, sizes(), **Full,** (count=5), {[count]}

Full, del_data(),**NotFull,** (0<count<5), {<count>, [data=Q[f], f++, count--]}

Empty|**NotFull**|**Full,** ~queue(), S_t, (Ø), {Ø}

}

A Tree Contains the Test Messages, Test Oracles, and Data Member Def-Use Information of the queue Class.



[count=0], Que.is_empty(), **[count=0].**
[count=0], Que.sizes(), **[count=0].**
[count=0], Que.add(), **[0<count<5]**, <Que.add(),...,Que.add()>, **[0<count<5]**, Que.sizes(), **[0<count<5].**
[count=0], <Que.add(),...,Que.add()>, **[0<count<5]**, Que.del_data(), **[0<count<5].**
[count=0], Que.add(), **[0<count<5]**, Que.del_data(), **[count=0].**
[count=0], <Que.add(),...,Que.add>, **[0<count<5]**, Que.add(), **[count=5]**, Que.sizes, **[count=5].**
:
[0<count<5], Que.add(), **[0<count<5]**, Que.add(),**[count=5]**, Que.is_empty(), **[count=5].**
[0<count<5], Que.add(), **[0<count<5]**, <Que.del_data(),…,Que.del_data()>, **[0<count<5].**
[0<count<5], <Que.add(),...,Que.add()>, **[0<count<5]**, Que.add(), **[count=5].**
[count=5], Que.del_data(), **[0<count<5].**
:

Test cases of the Que object

```
:
Que.add();
Que.sizes();
Que.add();
Que.del_data();
Que.add();
:
Que.del_data();
Que.add();
Que.sizes();
Que.add();
:
Que.sizes();
Que.is_empty();
:
```

A test message file for the Que object

Data Flow Testing Criteria

Each instance of a variable in a program is classified as a **definition** or a **use.**

A **use** of a variable is **C-use** or **P-use.**

if (x > 0) {x = y + 10;}

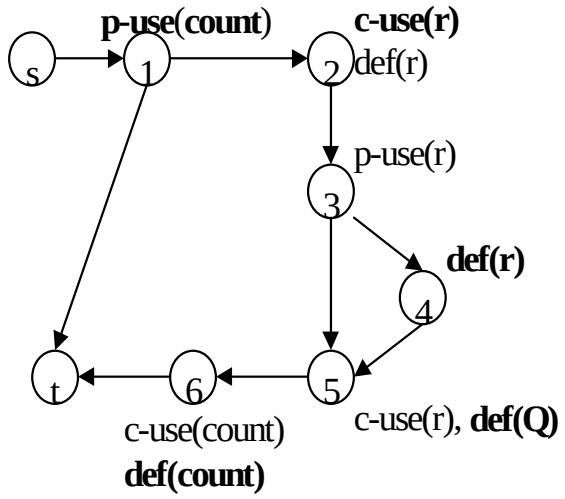
The **Global Definition** and **Global Use**

dcu(v,i) = { j | v $\in$ c-use(j) }

dpu(v,i) = { j | v $\in$ p-use(j) }

A **data flow anomaly** occurs when something unreasonable is done with data in a path (i.e. double defined, undefined or unused).

```
int queue::add(char data){
1.    if (count == Size){
         cout << "Not room for new \n";
         return (0);
      }
2.    r++;
3.    if(r==Size){
4        r=0;}
5.    Q[r] = data;
6.    count++;
7.    return (1);
}
```



The C++ code add() member function on the left and its directed flow graph on the right. Nodes in the graph represent statements in the function; start and terminate nodes are added for analysis.

Definitions and uses of count, f, r, Q in class Que; only called functions are listed

Data members	count	f	r	Q
definition	queue(), add(), del_data()	queue(), del_data()	queue(), add()	queue(), add()
c-use	add(), sizes(), del_data()	del_data()	add()	del_data()
p-use	add(), del_data(), is_empty()	del_data()	add()	

The def, c-use and p-use information associated with called functions

Function Names	**def**inition	**c**omputation-**use**	**p**redication-**use**
queue()	{count, f, r, Q}		
add()	{r, Q, count}	{r, count}	{count}
is_empty()			{count}
sizes()		{count}	
del_data()	{f, count}	{f, Q, count}	{count}

The maximum du-paths across only two member functions with respect to every data member are $\mathbf{N\times(M_d\times(M_{cu}+M_{pu}))\Rightarrow 2N\times M^2}$. In this ***queue*** example, there are $3\times(3+3)+2\times(1+1)+2\times(1+1)+2\times(1+0)=28$

Difficulty: *N!* sequences of function calls

Inadequate Test Cases,
queue()→del_data()→add()

***Infeasible* Test Paths,**
queue()→add()→del_data()→del_data()

The def-use information of data members generated by MACT

d(r,f,count,Q), cu(count). // *constructor*, ***empty***, *sizes()*, ***empty.***

d(r,f,count,Q), *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count),* pu(count),cu(f,Q), cu-d(f), pu-d(f), cu-d(count).
// *constructor*, ***empty***, *add()*, ***notfull***, *del_data()*,***empty.***

d(r,f,count,Q), *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count), pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count),…, pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count),* pu(count),cu(f,Q), cu-d(f), pu-d(f), cu-d(count), pu(count).
// *constructor*, ***empty***, *add()*, ***notfull***, *add()*, ***notfull***, *...,add()*, ***notfull***, *del_data()*, ***notfull***, *is_empty()*, ***notfull.***

:

d(r,f,count,Q), *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count), pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count), …, pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count),* pu(count),cu(f,Q), cu-d(f), pu-d(f), cu-d(count), cu(count).
// *constructor*, ***empty***, *add()*, ***notfull***, *add()*, ***notfull***, *..., add()*,***full***, *del_data()*, ***notfull***, *sizes()*, ***notfull.***

:

:

Note: d(x): x is defined
cu(x): x used for computation
pu(x): x used in predicate
cu-d(x): cu(x) and then d(x)
pu-d(x): pu(x) and then d(x)

Data members definition-use pairs in the queue Class

The du paths with respect to each data member in queue

Functions / Data Members	**queue()**	**add()**	**del_data()**
r	def(r)	cu-d(r), pu-d(r), cu(r)	
f	def(f)		cu(f), cu-d(f), pu-d(f)
count	def(count)	pu(count), cu-d(count)	pu(count), cu-d(count)
Q	def(Q_1,…,Q_5)	d(Q_1)	cu(Q_1)

Note: The arcs emerging from table show the defined data member could be used in succeeding functions

Code Optimisation: ***f*** defined in ***del_data()*** has not been used.

An anomaly of ***Q*** occurs in the ***queue() → add()***

An example of test driver program

```
// File name: queue.h
// The class under test, called queue,
// shows  in Figure 3
const int Size = 5;
class queue{
protected:  char  Q[Size];
            :
}
```

inherited

inherited

```
// File name: amt.h
// An abstract class and amt() shows
// state value
class amt : public queue{
public:
   void amt(){
   cout<<count<<'\n';
  }
};
```

```
// Test Message file
add('z)
add(y)
del_data()
is_empty()
          :
```

```
// file name: testdriv.cpp
#include <iostream.h>
#include "queue.h"
#include "amt.h"

class Queue :  public amt {};
void main(void) {
fstream outfile;
outfile.open("result.txt", ios::out);
if (!outfile){
cerr <<"Cannot open test result file\n";
abort ();
}
Queue Que;
          :
Que.add('z');
outfile<<"add()\t"<<Que.amt()<<'\n';
Que.add('y');
outfile<<"add()\t"<<Que.amt()<<'\n';
Que.del_data();
outfile<<"del  data()\t"<<Que.amt()<<'\n';
Que.is_empty();
                    :
}
```

Test message file-1	Test message file-2
Que.add(a)	Que.add(a)
Que.add(b)	Que.add(b)
Que.add(c)	Que.del_data(1)
Que.add(d)	Que.del_data(1)
Que.add(e)	Que.del_data(1)
Que.del_data(1)	:
:	:

Test Message files

Test result via test message file-1		Test result via test message file-2	
Function name	Value of Count	Function name	Value of Count
add()	1	add()	1
add()	2	add()	2
add()	3	del_data()	1
add()	4	del_data()	0
add()	5	del_data()	-1
del_data()	4	:	:
:	:	:	

Test Result files

Error found in the test result file-1			Error found in the test result file-2		
Function name	Value of Count	Ok/Error	Function name	Value of Count	Ok/Error
add()	1	Ok	add()	1	Ok
add()	2	Ok	add()	2	Ok
add()	3	Ok	del_data()t	1	Ok
add()	4	Ok	del_data()	0	Ok
add()	5	Ok	del_data()	0	Error
del_data()	4	Ok	:	:	:
:	:	:	:	:	:

The Test Reports of the* queue *Class

Conclusions

The steps of object oriented class testing are:

(1) Intra-method and inter-method levels: Data-flow Testing.

(2) Intra-class level: State-based Testing + Data Flow Testing.

A State-Based Testing Approach Providing Data Flow Coverage in Object-Oriented Class Testing

Bor-Yuan Tsai*, Simon Stobart, Norman Parrington, and Ian Mitchell
School of Computing, Engineering and Technology
University of Sunderland,
St. Peter's Way, Sunderland SR6 0DD UK
Tel: +44 191 5153201
E-mail: {cs0byt, cs0sst, cs0npa, cs0imi}@isis.sunderland.ac.uk

*Also of Department of Information Management,
Tamsui Oxford University College,
32 Jen-Lee Street, Taipei County 251, Taiwan

Abstract

A novel Object-Oriented class testing approach, proposed in this paper, combines functional with structural testing techniques. Based on state-based testing, test cases generated from the MACT (Method for Automatic Class Testing) tool can be used to execute functional testing. The definition-use information of data members, occur in public member functions of a class under test, is generated from MACT to facilitate data flow analysis. Testers can compute definition-use path with the information in order to ensure that the class is also satisfied with data flow coverage at intra-class level. The discussion with a queue class example to reveal that using a hybrid testing technique benefits class testing.

1 Introduction

Most computer applications can be tested in one of two ways: (1) functional (black-box) testing, and (2) structural (white-box) testing. The supporters of structural testing argue that functional techniques may not provide sufficient coverage of the code. Their opponents contend that structural approaches do not consider the requirements of specifications at all, since test cases are entirely generated from the implementation [1] [2] [3].

The new class testing approach for Object-Oriented classes, discussed in this paper, is adopted by MACT [4] [5] [6] [7]. In which the test case tree, created from the state machine of a class under test, is used to produce test messages for functional testing as well as the intra-class definition-use information for structural testing. In functional testing, each class is tested as a unit based on state-based testing techniques, to verify the behaviour of an object, such as state changes. Data flow techniques are used to perform this structural testing that enables us to detect whether every data variable (data member, or parameter in public methods) has been defined prior to being used, and whether all defined data variables have been used in the class.

The functional testing approach in MACT needs a (an implementation) state machine as a specification. Following the state machine, the test case tree generator builds a threaded mutli-way tree. After tracing the tree (also called test case tree), all possible test cases are generated. The tree completely duplicates the behaviour of the state machine of the class under test and it comprises all possible expected states of all transitions of the class [3] [7] [8]. Therefore the tree is also used as a test oracle, with which the test results are inspected by the test result inspector of MACT. Eventually, the pass/error messages generated by the MACT tool show whether the class under test is satisfied with the state-based coverage.

In the MACT tool, the various associations of data members with definitions and uses across the public methods in the class are taken into account, in order to compute intra-class definition-

use pairs. The definition-use information across public functions is examined whether or not any data anomalies exist in the sequence public methods called by clients. Inter-method definition-use pairs can also be computed by MACT, if the interaction of private and public functions can be described as a sub-state machine and embedded in the super state machine that the public methods are depicted in. However this is excluded from this paper. In data flow coverage, other pairs of definition-use (e.g. parameters, alias and local variables) also need to be concerned in the class under test. At the intra-class level, however, only the occurrences of data members within all possible sequences of function calls, which can be executed by the class under test, are considered.

The framework of MACT is illustrated in section three. A *queue* class example and the technique of test case generation are introduced in section four and five respectively. The concept of data flow testing and the approach of MACT, which is used as a structural test tool, are described in section six and seven. In which the *queue* example is also used to show how the definition and usage of data members are generated by MACT. The rest functions of MACT are briefly discussed in section eight. Following that is our future work and conclusions.

2. Background

The main point of *state-based testing* [9] [10] [11] is to examine the values, which have been stored in the object at a particular time. Those particular values represent the state of the object. State-based testing also validates the interactions that occur between the transitions and the state of an object. The changing states rely on the values that are changed by the transition. After executing a transition, it validates the final state that has been achieved by the object.

A *state machine* defines the set of states and transitions, and depicts the dynamic behaviour and the state changes of an object. Therefore the state machine can be used as an aid in state-based testing [12]. Programmers refer to the state machine of a class as a specification to code an implemented class, and the expected results are the requirements that should be followed. Therefore the states of the implementation object should be the same as the states that are shown in the state machine. Nevertheless, a transition name in the state machine could be different from its member function name in the implemented class. If any differences exist, then it is necessary to use an *implementation state machine* to reveal the behaviour and state changes of the implemented class.

Data flow testing techniques [13] [14] [15] are based on data flow analysis, and require that the test data (cases) exercise paths from definitions to uses. These techniques are also code-based testing techniques, and they can be used in code optimisation, anomaly detection, and test data (cases) generation [2] [15]. Extending the techniques to test object-oriented classes, the test cases can be classified into intra-method, inter-method and intra-class three categories [2] [16]. Intra-class testing examines the definition-use paths of the class's variables which across public methods when they are called in various sequences.

Object-oriented classes are usually designed as an independent small (sub) system. They respond to service requested from outside. If a whole class is tested as a unit, then it can be similarly performed as an application system test. Systems can be tested using *unit (intra-method*), integration (*inter-method*), and system (*intra-class*) testing techniques. The state-based testing technique covers the entire object rather than just individual methods and so is much more appropriate for testing objects. The *test cases* generated with state-based testing methods will be used to execute intra-class testing.

A test case can create a starting state, the expected action causing transition to the next state, and the expected next state [17]. The test cases considered in this paper are defined for class testing by sending messages to objects and estimating the results. The test message generator in

MACT only produces the expected messages using state machines. Those messages are directly sent to the object class under test.

3 The Framework of MACT

Based on the new object-oriented class testing approach, an automatic object-oriented class test tool, called MACT, is built. Which consists of five components: Test Case Tree Generator, Test Message Generator, Test Driver Generator, Test Result Inspector and Def-Use Info Generator. The first four are used to achieve state-based testing, and the last is a mechanism for data flow testing. The framework of MACT is shown in Figure 1.

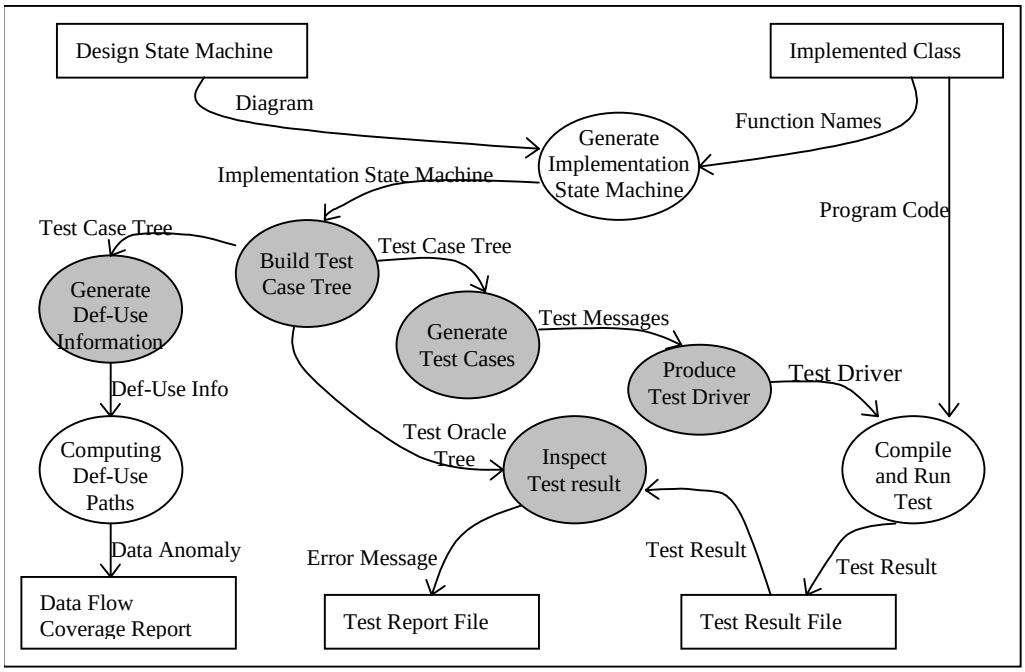


Figure 1 *The components of the automated class testing framework (MACT)*

Designing test cases for a unit test, testers usually require the specification and source code of the unit [18]. Programmers follow the specification (e.g. state machine) to implement the program code of the class. Testers also need to review the specification and implementation of the class to design test cases, in order to test the implemented class with the test messages directly. Therefore, the implementation state machine (e.g. in Figure 4) may be required in MACT. The *test case tree generator* can generate various test case trees according to the various implementation state machines. Traversing the test case tree, test messages for the class under test are automatically produced by the *test message generator* of MACT. The *test driver generator* in MACT receives test messages and generates a C++ *main()* format function as a test driver in a C++ program code. The class under test is included as a user-defined class and test messages are listed in the *main()* function. An example of the test driver is illustrated in Figure 12. In a driver, a test result file is declared as an output file, in order to record the test result (state) values while the program is executing. The tree also contains test oracles, which are based on state-based testing, consists of function names and state information. Hence the test results are inspected by the *test result inspector*, which parses the test result records (containing messages and resultant states of the messages) one by one using the test oracle tree. The *def-use info generator* also traverses the same tree, which contains the occurrences of data members in each member function, to produce definition-use information. This intra-class level information facilitates computing definition-use paths and detecting data anomaly.

4 Example: A Queue Class

Assume a circular bounded *queue* class that can only store five units of data. An object of the *queue* has six state spaces, which are the empty state, contains 1 unit, contains 2 units, ... , to the full state. The state of the *queue* object is defined by the value of the *count* data member. We can classify the states into *Empty*, *NotFull*, and *Full* sub-states. The *Que* object is at the start (*Empty*) state when it is defined and the constructor function sets its *count* to *0*. Moreover the state of *Que* will change to *NotFull*, when a piece of data is added into it, and *addone* transition is executed. The behaviour of *Que* is depicted with the state machine in Figure 2. In which the transition *checksize* does not cause any state change. When *Que* is at the *NotFull*, *add* and *del_data* methods (transitions) may not cause any state change until *count = 5* or *count = 0*.

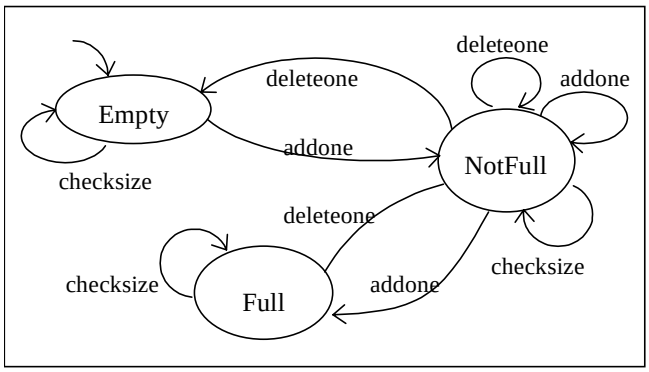


Figure 2 *The Design State Machine of the queue*

Suppose the *queue* class has *f*, *r*, *count* and *Q[SIZE]* four data members. The *Q* can only contain five units of data, and *queue* has a constructor, a destructor and four public member functions. The C++ code template of *queue* is given in Figure 3.

```
const int Size = 5;
class queue{
protected: char    Q[Size];    // bounded array
           int  f, r;          // Front/Rear index of queue
           int  count;         // a counter of the array
public: queue(void);           // default constructor
           void    is_empty(void);
           int  add(char);     // add data to the queue
           char    del_data();//delete data from the queue
           void    sizes(void);// get the size of the queue
           ~queue(void); // destructor
};
queue::queue(void){
     r = -1;  f = 0;
     count = 0;
     for (int i = 0; i < Size; i++){
        Q[i] = ' ';}
}
int queue::add(char data){
     if (count == Size){
        cout << "Not room for adding new data to stack\n";
        return (0);}
     r++;
     if(r==Size){
        r=0;}
     Q[r] = data;
     count++;
     return (1); }
```

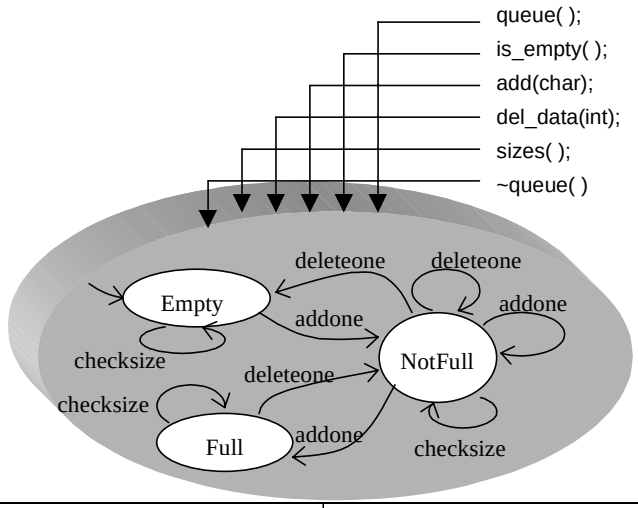
```
char queue::del_data(){
     char data;
     if (count == 0){
        cout << "can't delete data from an empty stack\n";
        return('0');}
     data = Q[f];
     f++;
     if(f==Size){
        f=0;}
     count--;
     return (data);
}
void queue::is_empty(){
     if (count == 0)
        cout<<"The queue is empty\n";
     else
        cout<<"The queue is non-empty\n";
}
void queue::sizes(){
     cout << "the size of the queue :" << count << "\n";
     return;
}
queue::~queue(){
     cout << "\nfinishing the class test\n";
}
```

Figure 3 *The implemented C++ code of the* queue *class*

4.1 Implementation State Machine of the *queue* Class

The *chechsize* transition in Figure 2 is performed by the *is_empty()* or *size()* functions in the implemented class, see Figure 3. Moreover, the detailed information of transitions and member functions, in which data members are accessed, are required to facilitate data flow analysis. Hence an *implementation state machine* of the *queue* is illustrated in Figure 4 to display the information of implemented class code and the behaviour of the object class.

The design state machine (Figure 2) is encapsulated as a class. The attached member functions show the possible messages, which can be sent to the objects by invoking these member functions declared inside the class. The table following the oval diagram describes the mapping between transitions and member functions. Function names with the | symbol in the table means OR.



Transition Names	*Function Names*
checksize	is_empty() | size()

The implementation state machine of the class **queue** is defined with finite sets ISM = (**V**, **F**, **S**, **T**) where
V = **{**char Q, int f, r, count**}**
F = **{**queue(), is_empty(void), add(char), dele_data(void), sizes(void), ~queue()**}**
S = **{**(**Empty**, count=0), (**NotFull**, 0<count<5), (**Full,** count =5)**}**
T = **{** S_0, queue(), **Empty,** (count=0); {[count, r, f, Q]}
Empty, is_empty(), **Empty,** (count=0), {<count>}
Empty, sizes(), **Empty,** (count=0), {[count]}
Empty, add(data), **NotFull,** (0<count<5), {<count, r>, [r++, r=0, Q[r]=data, count++]}
NotFull, is_empty(), **NotFull,** (0<count<5), {<count>}
NotFull, sizes(),**NotFull,** (0<count<5), {[count]}
NotFull, add(data), **NotFull,** (0<count<5), {<count, r>, [r++, r=0, Q[r]=data, count++]}
NotFull, add(data), **Full,** (count=5), {<count, r> [r++, r=0, Q[r]=data, count++]}
NotFull, del_data(),**NotFull,** (0<count<5), {<count, f>, [data=Q[f], f++, f=0, count--]}
NotFull, del_data(), **Empty,** (count=0), {<count, f>, [data=Q[f], f++, f=0, count--]}
Full, is_empty(), **Full,** (count=5), {<count>}
Full, sizes(), **Full,** (count=5), {[count]}
Full, del_data(),**NotFull,** (0<count<5), {<count>, [data=Q[f], f++, count--]}
Empty|**NotFull**|**Full,** ~queue(), S_t, (Ø), {Ø}
}

Figure 4 *An Implementation State machine of the* ***queue*** *Class*

In the implementation state machine, V is a set of data members declared in *queue* and the parameters of the member functions; F implies a set of member functions declared in *queue*. S is a set of all states which an object of *queue* has, and T is a set of all transitions between states, and each element of T is a quintuplet = {*current*, *function*, *result*, *predication*, *data access*}. Where

current, *result* ∈ S, are states having a transition outgoes and incomes respectively. *Function* ∈ F, which triggers the transition from current to result. The *predication* causes state change and it is the post/pre-condition of the current/next transition. A set of <predication use> and/or [definition or computation use], bracketed within {} in Figure 4, indicates the data access of the data members in the functions. For instance, the *<count, r>* exposes the *count* and *r* are used at condition statements (predication use), and, *[r++, r=0, Q[r]=data, count++]* indicates that the *r, Q and count* are defined in the *add()* function, as well as the values of *r* and *count* are referenced*.

The state change of an object is determined by the execution of the next member function at the current state. The current state value is the pre-condition/post-condition of the next member function/previous member function will be/has been executed. For example the *Que* object is at *Empty* state when *count = 0* is true. This is unsatisfied with the pre-condition of *del_data()* member function, so that the *del_data()* function cannot be triggered when the object is at the *Empty* state. The transition "*Empty, add(data), NotFull, (0<count<5)*" described in Figure 4 is *Empty* and *NotFull* as a current state and a result state respectively. This transition occurs when an object is at the *Empty*, pre-condition *(count=0)* is true, and the member function *add()* is called. When the *add()* function is executed and *count++* is performed, then the state changes to *NotFull*. Hence, the pre-condition of this transition is *count=0* and its post-condition is *0<count<5*.

4.2. The test case tree of the queue class

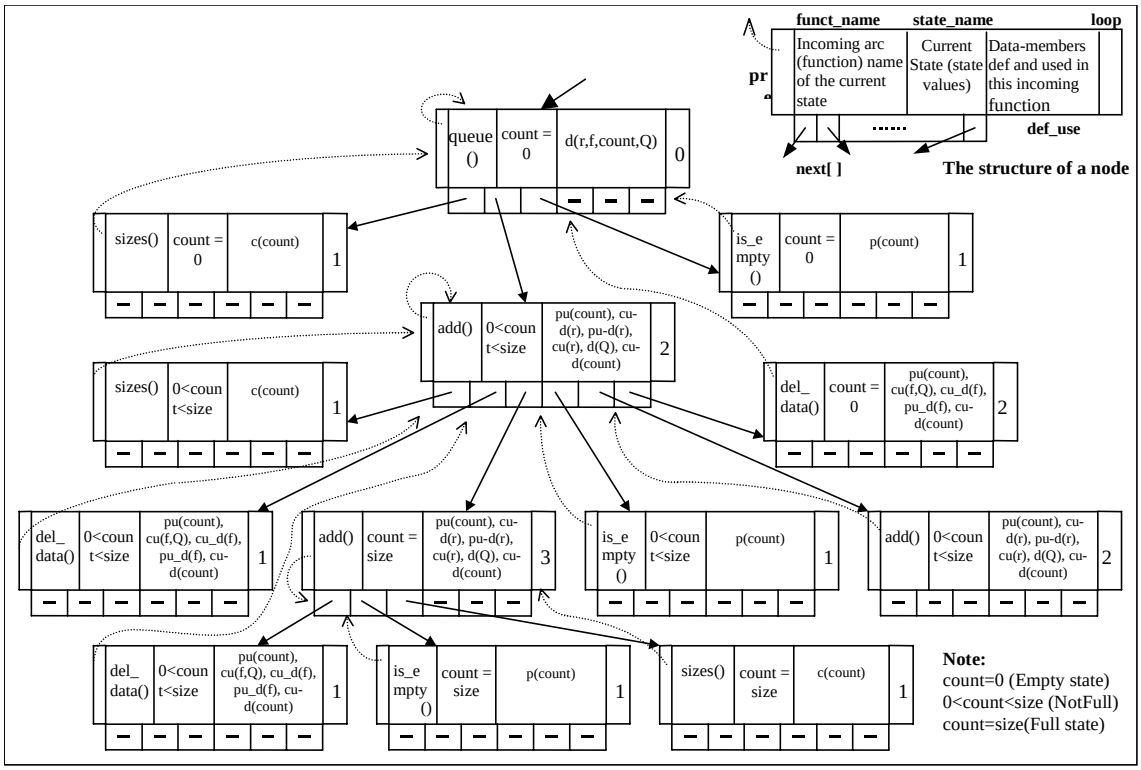


Figure 5 *A Tree Contains the Test Messages, Test Oracles, and Data Member Def-Use Information of the queue Class.*

Following the implementation state machine in Figure 4 and based on state-based testing, the *test case tree generator* (see Appendix A) of MACT produced the threaded multi-way tree in

* In C or C++, *int x;* is a definition statement and *x=10*; is an assignment statement. However, in data flow testing, the definition of a variable *x*, when it is at the left side of an assignment (e.g., *x=10*). The value of *x* is referenced, when it is at the right side of an assignment (e.g. *y=x+10;)*

Figure 5. The tree completely duplicates the graph of the state machine. The pointers and the threads in the tree can simulate any cyclic links in the state machine. Each (circular) path is a sequence of test cases. The core of MACT is the threaded multi-way tree, which simulates the behaviour of a state machine. Even more complicated state machines (featuring hierarchies, concurrency, and nested states) can also be represented by this approach [7]. The structure of each tree node is shown at right up corner in Figure 5.

5 Possible test cases of *queue* generated from MACT

After tracing the tree in Figure 5 with *Que*, the test cases of *Que* are produced by loops in the tree from the root node and shown in Figure 6. The condition in bold square brackets followed by messages is a pre-condition. The state value (post-condition) within bold square brackets following the messages means the expected resultant state, after the messages being executed. For the sake of simplicity, repeatedly executing a member function is represented with "*Oue.f(),...,Oue.f()*" form, and messages in a pair of arrow brackets represent the state is still at same after executing the messages. The *test message generator* (its algorithm shows in [5]) traverses the tree from the root node down to the leaves. A generated message file, consists of a sequences of function calls, for *Que* is produced, see Figure 7.

[count=0], Que.is_empty(), **[count=0].**
[count=0], Que.sizes(), **[count=0].**
[count=0], Que.add(), **[0<count<5]**, <Que.add(),...,Que.add()>, **[0<count<5]**, Que.sizes(), **[0<count<5].**
[count=0], <Que.add(),...,Que.add()>, **[0<count<5]**, Que.del_data(), **[0<count<5].**
[count=0], Que.add(), **[0<count<5]**, Que.del_data(), **[count=0].**
[count=0], <Que.add(),...,Que.add>, **[0<count<5]**, Que.add(), **[count=5]**, Que.sizes, **[count=5].**
:
[0<count<5], Que.add(), **[0<count<5]**, Que.add(),**[count=5]**, Que.is_empty(), **[count=5].**
[0<count<5], Que.add(), **[0<count<5]**, <Que.del_data(),…,Que.del_data()>, **[0<count<5].**
[0<count<5], <Que.add(),...,Que.add()>, **[0<count<5]**, Que.add(), **[count=5].**
[count=5], Que.del_data(), **[0<count<5].**
:

Figure 6 *Test cases of the Que object*

:
Que.add();
Que.sizes();
Que.add();
Que.del_data();
Que.add();
:
Que.del_data();
Que.add();
Que.sizes();
Que.add();
:
Que.sizes();
Que.is_empty();
:

Figure 7 *A test message file for the Que object*

6 Test cases of *queue* generated based on data-flow criteria

A class is a basic unit of testing in an object-oriented program, and most of this test work has centred on black-box approaches. Harrold [2] developed a class control flow graph to connect all methods in the class, and adapted Pande's [19] data flow analysis algorithm to compute the data flow information required for data flow testing. Hong [20] demonstrated the Class State Machine (CSM), extended from Finite State Machine, to specify the behaviour of classes. The CSM was then transformed into a Class Flow Graph to show data flows of the state machine. Eventually, selected intra-class test cases using data flow testing techniques.

Harrold [2] and Hong [20] describe that intra-class def-use information can guide testers in the selection of sequences of methods (function calls). In the following, the *queue* class is used as example to discuss the difficulty of generating intra-class test cases for the *Que* based on data flow testing techniques. Moreover, the selection of sequences of function calls, which satisfy

certain data flow coverage criteria, may also contain some unnecessary (ambiguous) method sequences. Testers need to filter them by referencing the functionality of the class.

6.1 Overview of Data Flow Testing Criteria

Data flow testing techniques need directed flow graphs to facilitate the computation of def-use pair information, the selection of test cases, and the detection of anomalies within the program under test. The indication of anomalies may include: (1) defining a variable twice with no intervening use, (2) referencing a variable that is undefined, and (3) not referencing the variables that are defined [1] [21].

Based on data flow testing, each instance of a variable in a program is classified as a **def**inition or a **use**. A definition of a variable is that a variable is assigned a value. A use of a variable is that the value of the variable is used (referenced). Uses of a variable are further divided into two classes as either **c**omputation **use**s (c-use) or **p**redicate **use**s (p-use) [22]. A c-use occurs when the value of a variable is used in a computation or output statement, and a p-use occurs when the value is used in a condition (predicate) statement. For instance, the *if (x > 0) {x = y + 10;}* statement contains p-use of *x* and c-use of *y*, followed by a def of *x*.

The du path of each data member is from its definition to every use that is reached by the definition. Let *i* be any member function and *v* any variable such that $v \in$ def(*i*). Hence, *dcu(v,i)* is the set of all functions *j* such that $v \in$ c-use(*j*), and in this path there is a def-clear sub-path with respect to *v* from *i* to *j*. The *dpu(v,i)* indicates the set of all functions *j* such that $v \in$ p-use(*j*) and in this path there is a def-clear sub-path with respect to *v* from *i* to *j*.

Data members in a class are also global variables. In data flow testing at intra-class lever, a global use (c-use or p-use) of a data member *x* if and only if the definition of the *x* preceding its use does not occurs within the same member function. Otherwise, it is a local use. A global definition of a data member *x*, if and only if the last definition of the *x* occurs in a member function *i*, and there is a def-clear path with respect to the *x* from the *i* to another member function *j*. In the function *j*, there is a global use of the *x* [23].

The criteria of testing-path-selection are based on data flow, and focus on variables to be defined and used. When executing a test case, the test case is said to exercise a def-use (sub) path if the (sub) path is traversed. Therefore, tracing the flow of data members among member functions in the class rather local variables within an individual function is concerned in this paper. Some data flow testing concepts criteria can be referenced in [2] [14] [15] [22].

6.2 Def-use information in the *queue* example

Selecting intra-class test cases, the interest is only the data members in the public member functions of *queue*. In this example, moreover, each element in the data member *Q* array will be dealt with individually.

For simplicity of def-use presentation in each member function of *queue*, each code statement is a unit, in which the definition and/or use of data members occur. For example, in the data flow graph of *add()* function (see Figure 8), the **p-use(*count*)** and **c-use(*r*)** at the node 1 and 2 cause concerned as to whether or not the *count* and *r*, used in this function, have been properly defined in the preceding functions. Moreover, we also need to examine if the defined *count* and *Q*, **def(*count*)** and **def(*Q*)**, will be able to be used in the succeeding functions.

```
int que::add(char data){
1.    if (count == Size){
         cout << "Not room for new \n";
         return (0);
      }
2.    r++;
3.    if(r==Size){
4        r=0;}
5.    Q[r] = data;
6.    count++;
7.    return (1);
}
```

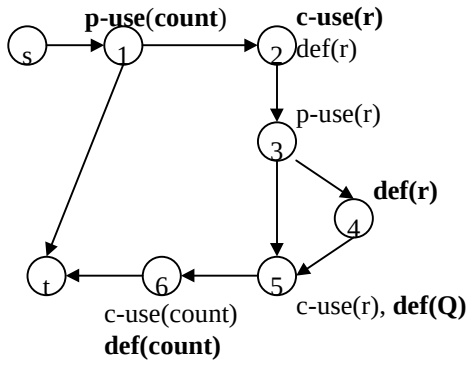


Figure 8 *The C++ code add() member function on the left and its directed flow graph on the right. Nodes in the graph represent statements in the function;* ***s****tart and* ***t****erminate nodes are added for analysis.*

As in Figure 8 the bold def and use of data members should be computed, and also the non-bold data members. The dpu(*r*, 2)={3}, and dcu(*r*, 4)={5} have formed sub paths within the *add()*, and the definition of the *r* at statements 2 or 4 could be either used in the succeeding functions. Figure 9 illustrates the intra-class definition-use information of data members in *queue*.

functions (*queue(), is_empty(), add(), del_data(), sizes()*).

global_defs (*queue()*, [count, f, Q, r]).	global_defs (*del_data()*, [f, count]).
global_cuses (*queue()*, []).	global_cuses (*del_data()*, [Q, f, count]).
global_puses (*queue()*, []).	global_puses (*del_data()*, [count]).
global_defs (*is_empty()*, []).	global_defs (*sizes()*, []).
global_cuses (*is_empty()*, []).	global_cuses (*sizes()*, [count]).
global_puses (*is_empty()*, [count]).	global_puses (*sizes()*, []).
global_defs (*add()*, [r, Q, count]).	
global_cuses (*add()*, [r, count]).	
global_puses (*add()*, [count]).	

Figure 9 *Global definition-use information of data members within* ***queue***

The definitions and uses of the data members among the functions of *queue* are shown in Table 1. The test cases can be generated to cover associations between definitions and uses of each data member from the table, based on du-path criteria. The arcs in Table 1 show that some pairs of sequence methods are used for intra-class testing. For example, du-paths with respect to the *count* are in *queue()→add()*, *add()→add()*, *add()→del_data()*, and *del_data()→is_empty()*. There are d-cu paths of the *r* in *queue()→add()*, and *add()→add()*. A d-cu path of *Q* and *f* exists in *add()→del_data()* and *del_data()→del_data()* respectively, Furthermore, a sequence of messages *queue()→add()→add()→del_data()→del_data()* which has all defs coverage, can be established. It is time-consuming work to compute this sequence.

Table 1 *The def, c-use and p-use information associated with functions*

Function Names	**def**inition	**c**omputation-**use**	**p**redication-**use**
queue()	{count, f, r, Q}		
add()	{r, Q, count}	{r, count}	{count}
is_empty()			{count}
sizes()		{count}	
del_data()	{f, count}	{f, Q, count}	{count}

6.3 Generating Data Flow Test Cases

It is difficult to discover all possible test cases to achieve all defs, all uses, or all du-paths. For example, a class has N data members (denoted V_1, V_2, …, V_n) and M public member functions, in each of which every data member is defined (denoted M_d), computation used (denoted M_{cu}), and predicate used (denoted M_{pu}). Then the maximum du-paths across only two member functions with respect to every data member are $V_1\times(M_d \times(M_{cu}+M_{pu})) + V_2\times(M_d \times(M_{cu}+M_{pu})) + \ldots + V_n\times(M_d \times(M_{cu}+M_{pu})) \Rightarrow N\times(M_d\times(M_{cu}+M_{pu})) \Rightarrow 2N\times M^2$. In this *queue* example, there are $3\times(3+3)+2\times(1+1)+2\times(1+1)+2\times(1+0)=28$ du-paths within two member functions in the sets with respect to data member, *count, r, f* and *Q* individually. Additionally, a pair of ordered functions may not only contain a data member. Such that *count* and *r* two data members occur in the *queue()→add()* test path. Selecting a test case, the du pairs of all possible data members in each test case ought to be considered.

6.4 The Complexity of Data Flow Criteria for Test Case Selection

Weyuker [24] [25] proposed that all du-paths require an exponential number of test cases in the worst case. If *d* is the number of (two way) decisions in the program, then the *all uses* data flow criterion requires $O(d^2)$ test cases, and *all du-paths* requires $O(2^d)$ in the worst case. For example, suppose a program (procedure) comprises a sequence of *d* IF-THEN-ELSE statements, and each of them defines and uses a variable *x*. All-du-paths may, then, require 2^d test paths. However, this is simply the worst case and in practice both criteria would require only at most $d + 1$ test cases [24]. That means many test cases of the worst case are not necessary.

In a class, each method has to be tested individually. Each class will then have N associated test cases when there are N methods in the class. Nevertheless, to check the validity of calling sequences within the class, such as intra class coupling, *N* methods implies the order *N!* test cases [26]. In those cases there could be some redundant sub sequences. For instance, the first element of the *Q* is defined in the first *add()* function (called F_{add1}), and the element is referenced in the first *del_data()* function (called F_{del1}). There can be several member functions with different performance orders in the interval of F_{add1} and F_{del1}. Such as *add()→del_data(); add()→is_empty()→del_data(); add()→ … →del_data();* and each of the test cases has a du-path with respect to *Q[0]*.

The result, found by Bieman in [27], shows that 80% of the procedures need to be tested by ten or fewer complete paths to satisfy all du-paths criteria. However, it is a tedious task to find the redundant test cases.

6.5 Inadequate and Ambiguous Test Cases

Test cases can be produced following the du-paths. Some of the paths could be inadequate. For example in the queue, a *queue()→del_data()→add()* can achieve the du-paths with respect to *count*, *f*, *r*, and *Q*. However, it is unreasonable to perform the *del_data()* member function following the constructor, *queue()*. In addition, there is a du-path of the *count* and *r* data members in *queue()→add()*, but a double definition anomaly occurs on *Q*. If we avoid the ambiguous test cases, then some of data members may not be tested, such as the *queue()→add()* which is needed to test *count* and *r*.

6.6 Feasible and Infeasible Test Paths

An *infeasible* path is that no input data exists which can cause such a path to be executed. However, a path is *feasible* if there are some input data, which will cause the path to be traversed during execution. The sequence methods calls from outside of the class under test can be specification infeasible or implementation infeasible. Infeasible sequence methods (subpaths) should not (or cannot) be executed according to the specification. For example, *queue()→del_data()* sequence should not be required in the specification, and an infeasible implementation is such as *queue()→add()→del_data()→del_data()* sequence. Because *queue* cannot accept the second delete data message from the client object when its state value is zero (i.e. *count == 0*).

Harrold [2] and Hong [20] demonstrated their techniques are also useful for determining which sequences of methods should be executed to test a class, and pointed out error sequences with examples that need not be run. However, they did not discuss the technique to select infeasible sequences from *N!*. In fact, Weyuker [25] found that the non-executable (infeasible) path problem was the primary practical difficulty in using the all du-paths criterion, because there are many infeasible paths to contend with the criterion. In automatically generating sequences of calling methods to satisfy data flow criteria, the problem of generating infeasible sequences is impossible to avoid [28].

To gain all possible useful test cases, we need to remove redundant paths and to eliminate the infeasible (such as inadequate, ambiguous, or non-executive) test cases from the generated intra-class level test cases based on data flow testing criteria. However, in practice, it is not an easy task to detect them from *N!* test cases.

7 Data Flow Testing Approach in MACT

At the different point of view, if we already have test cases for all possible feasible paths of the class. We, then, use data flow analysis technique to detect whether or not these feasible sequences violate data flow criteria. On this assumption, intra-class test cases are generated by MACT based on state-based testing technique, and then the sequence paths within these test cases are analysed with data flow criteria to conclude whether these test cases are satisfied with data flow coverage.

The data flow testing approach of MACT is to firstly generate a test case tree of the class, which is derived from the design (or implementation) state machine. This is the same as the discussion in section 4.2. Secondly, by tracing the tree, intra-class definition-use information of data members can be produced for data flow testing. Finally, the def-use information of the data members is analysed to detect whether or not the class under test is satisfied with data flow coverage.

7.1 Generating Def-Use Information from MACT

Reviewing the implementation state machine, we then fill the occurrences of data members into the *def-use* field of the respective node in the test case tree, while the test case tree generator is executing. After the test case tress of the *queue* contains def-use information is produced, see Figure 5. Each node of the tree has the function name, current state value (name), and def-use information of data members. Based on the tree, we are

(1) to identify definitions and uses of each data member in the transition of the implementation state machine;

(2) to review these definitions, uses, and the sequences of test messages (generated by the MACT) to compute the sequences of definition-use pairs; and

(3) to analyse the information of these definition-use pairs to show what kinds of the data flow criteria those test messages can achieve and if any anomaly occurs.

The test case paths (see Figure 10), generated by tracing the tree, are used to detect whether they cover def-use paths. For example, the "*constructor*, ***Empty***, *sizes()*, ***Empty***" test case has a sequence of member functions, *queue()→sizes()*, and we can analyse the def-use information to find if the def-use path with respect to data members occur in this test case, and thus detect if data anomalies exist.

7.2 Computing DU Paths

The def-use information of data members in each member functions of the several test cases, generated by the MACT, is given in Figure 10.

d(r,f,count,Q), cu(count). // *constructor*, ***empty***, *sizes()*, ***empty.***

d(r,f,count,Q), *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count),* pu(count),cu(f,Q), cu-d(f), pu-d(f), cu-d(count).
// *constructor*, ***empty***, *add()*, ***notfull***, *del_data()*,***empty.***

d(r,f,count,Q), *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count), pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count),…, pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count),* pu(count),cu(f,Q), cu-d(f), pu-d(f), cu-d(count), pu(count).
// *constructor*, ***empty***, *add()*, ***notfull***, *add()*, ***notfull***, *...,add()*, ***notfull***, *del_data()*, ***notfull***, *is_empty()*, ***notfull.***

:

d(r,f,count,Q), *pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count), pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count), …, pu(count), cu-d(r), pu-d(r), cu(r), d(Q), cu-d(count),* pu(count),cu(f,Q), cu-d(f), pu-d(f), cu-d(count), cu(count).
// *constructor*, ***empty***, *add()*, ***notfull***, *add()*, ***notfull***, *..., add()*,***full***, *del_data()*, ***notfull***, *sizes()*, ***notfull.***

:

:

Note: d(x): x is defined
cu(x): x used for computation
pu(x): x used in predicate
cu-d(x): cu(x) and then d(x)
pu-d(x): pu(x) and then d(x)

Figure 10 *Data members definition-use pairs in the* ***queue*** *Class*

The definition-use of the data members in the "*queue()*, ***Empty***, *add()*, ***NotFull***, *del_data()*, ***Empty***" is shown as an expression in Figure 10. For the sake of explanation, the expression has been divided into four rows with different data members and they are shown in Table 2. In the first row, for example, the du paths can be computed with respect to the data member *r*. The *r* is defined in the *queue()* and *add()* functions, and then is used in the *add()* function. This shows no anomaly occurred on the *r* in this test case.

Table 2 *The du paths with respect to each data member in* ***queue***

Functions / Data Members	**queue()**	**add()**	**del_data()**
r	def(r)	cu-d(r), pu-d(r), cu(r)	
f	def(f)		cu(f), cu-d(f), pu-d(f)
count	def(count)	pu(count), cu-d(count)	pu(count), cu-d(count)
Q	def(Q_1,…,Q_5)	d(Q_1)	cu(Q_1)

Note: The arcs emerging from table show the defined data member could be used in succeeding functions

7.3 Code Optimisation

There are several du paths with respect to *f* and *count* in the second and third rows, and the last definition of *count* and *f* in this case is not against the du path criteria. Because the *count* and *f* are global definitions in the *del_data()*, they may be used in the succeeding messages, such as *is_empty()*, *sizes()*, or *del_data()*. However, if the all test cases generated from the test case tree (see Figure 10) are carefully reviewed, we can find out the last defined *f* will never be used. The sequence member functions following the *del_data()*, in which a definition of the *f* occurs, do not reference the defined *f*, except another following *del_data()*. The following *del_data()* references the *f* defined in the previous *del_data()*, will define the *f* again. This shows that the last defined *f* within the last *del_dat()* function in every sequence test case does not be used. That *del_dat()* has a "not referencing the variable that are defined" anomaly.

The *f* data member is used as an index when the front element in the Q*[SIZE]* array is removed. The last defined *f* can only be used in the *sizes()*, *is_empty()* or *~queue()* functions. In the *Que***,** the *count* is used to reply to the messages of the *sizes()* and *is_empty()* methods. However, the *count* can be replaced with *f* and *r*. By calculating *f* and *r*, the result can indicate whether the *Que* is at the empty, not-full or full state, even show how many units of data exist in the *Que*. This is a code optimisation problem and will not be discussed further in this paper.

7.4 Anomaly Detection

An anomaly of data member *Q* occurs in the *queue()→add()* test case is found in the last row of Table 2. In fact, programmers are used to initialise variables as they are defined in the constructors of a class. In this case, it is necessary to initialise the *count*, *r* and *f*, but it is not necessary to empty the *Q* in the *queue()*. The test cases generated based on state-based testing could not detect this kind of anomaly.

Moreover, theses anomaly problem, the *Q* has double definition and the last defined *f* may not be referenced, can never be detected in the intra/inter method testing. Because intra-method testing is performed to test each function in the class individually and inter-method testing is executed to test a function together with other functions (in the same class) that are called directly or indirectly.

8 Other Functions of MACT

8.1 Test Driver Generator of MACT

The *test message generator* writes the function names into a test message file, which is part of the test driver. The *test driver generator* in MACT reviews the message file, and then

generates a C++ *main()* function as a test driver. The driver will include the class under test, and then test the class with the messages. After executing the test driver, the test results can be recorded into a test result file.

The class, *queue*, under test is included in the program as a user-defined class (e.g. *#include "queue.h"*) [29], and it is inherited by a subclass called Queue. Moreover, an additional member function, called *amt()*, is declared in the subclass in order to access the state values (the type is private) stored in data member *count*, which is also inherited from the *queue* class. Obviously, the *amt ()* can be declared as a friend function. Therefore, the implemented *queue* class (from the developers) is insulated from change and will not be modified at all.

A test result file is declared as an output file in this main function (test driver), so that it records the result of each test message execution. The driver is to store the test result of each message into the test result file, while the program is executing.

Test messages are embedded in the main function in the test driver example (see Figure 12). Alternatively, a test message file, containing those test messages, can be declared as an input file in the main function, and each test message is accessed as an input record, while the file is read. Test message files are ASCII text type files and each test message is stored in the files as a string. Therefore, a program code generator is utilised to read the test message file as an input file, and then produces the source code of a test driver program.

A test driver program, in which a class under test is included and a test message file and a test result file are declared, can be manually/automatically produced by a *program editor* or a *program source code generator*. If a source code generator can play as a test driver generator. A complete C++ test driver program can be automatically built by answering the prompts asked by the generator, see Figure 11. The prompts may ask to enter the names of (1) a class under test (e.g. *queue.h* in Figure 12), (2) a test message file, (3) a test result file, and (4) a function to reveal state value, e.g. *amt()* in Figure 12.

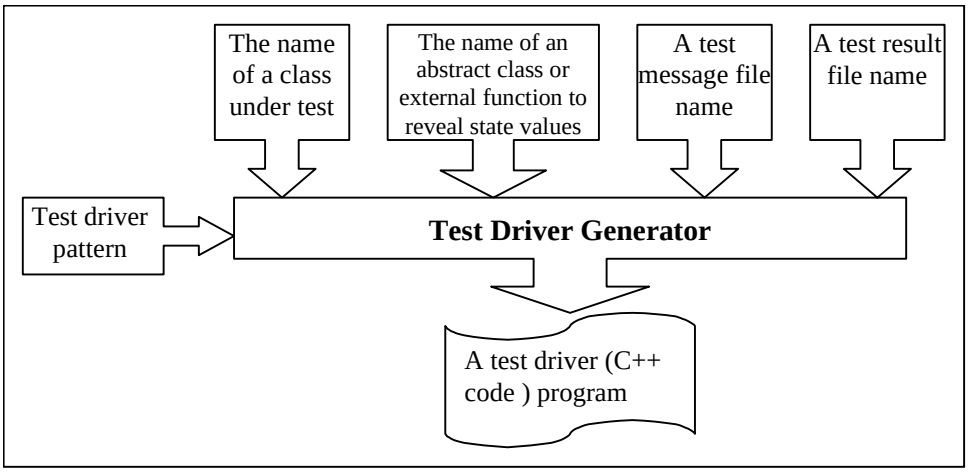


Figure 11 *Possible requirement to generate test driver in MACT*

Of course, the class under test and the function to show state values should be developed at first. The class is stored as a user-defined class with '*.h*' sub-program name in proper directory, such as *'c:\borlandc\include\'* sub-directory in Borland C++*. Then, the class is included in the generated test driver program and the function (friend function) is linked as an external function [29] with the program.

* Borland C++ for Windows, Version 5.0, Borland International, Inc.

Another way is that the function can be defined in an abstract class [29], which may also be stored with '*.h*' sub-program name in *'c:\borlandc\include\'* sub-directory. In order that the class can be included and inherited by the test driver program and subclass in the test driver program respectively, see Figure 12. In which, the right hand side program is a test driver program example. The parts high lighted in grey are inlayed when testers enter proper names in response to the prompts of the test driver generator. The rest parts without high light is the test driver pattern, see Figure 11.

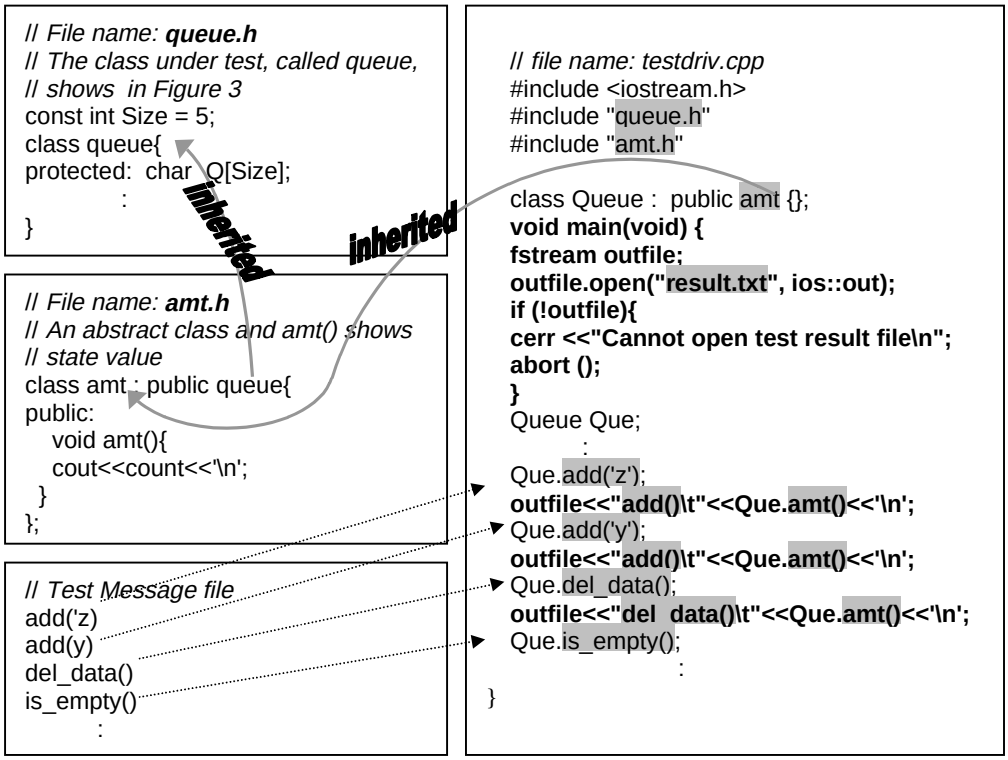


Figure 12 *The example of test driver generation.*

The diagrams from left top to down in Figure 12 are the class under test that is stored with *queue.h* file name, an abstract class which inherits the class under test and contains a function to show state value, and the test message file respectively.

8.2 Test result inspector

The *test result inspector* in MACT will parse the test result records (the messages and the resultant states of the messages) one by one using the test oracles in the tree. The oracles of *queue* (Figure 5) contain the expected state for each expected message. The inspector opens the test result file as an input file, and it can sequentially read each record, and then compare the function name and state value in the record with the oracles in the tree. If the state value in a record mismatches the expected state value in the tree, then an error is deemed to have occurred. The algorithm of inspecting the test result file is given in Figure 13.

```
STEP 0 set root to current
STEP 1 while not end of test_result file
STEP 2   read the test_result record and store data to funct and state two variables
STEP 3   if all children nodes of the current node have been traced go to STEP 5
STEP 4     if a child node's funct_name = funct and state is satisfied with its state condition
               then the test_result record is correct;
                    assign the child node's pre to current and
                    go STEP 6
               else go to STEP 3
STEP 5   print "this test result record is in error"
STEP 6   go to STEP 1
```

Figure 13 *The algorithm to inspect the test results with the oracle tree*

For example, when the first record (function name = "*add()*", and count = "1") of the test result file has been read at *STEP 2*. The inspector, at *STEP 3*, will look for a child node of the root (it is also the *current* node at the moment) in Figure 5 which contains the *add()* function name and its state condition is satisfied with the state value (count = 1). The middle child node of the root matches the requirement. Therefore, the *current* node moves to the middle node of the root when the statements in *STEP 4* have been executed.

Following the above algorithm, the Test Result Inspector can detect errors when state values in the result file cannot be matched with the state conditions in the oracle tree. For example, if count = 6 after executing an *add()* function, then an error message should be reported by the inspector. Because the state value 6 will not be satisfied with any state condition in the oracle tree, see Figure 5. If the inspector can not respond this with an error message, then the error existed in the *Que*. That intends the *Que* may not have ability to handle the over flow problem, or the *count* data member in the *Que* may be accumulated improperly.

9 Future Work and Conclusions

The core of MACT is a threaded multi-way tree (test case tree) which simulates the behaviour of a state machine. Test cases can be generated from the tree, which contains the test oracle of the class under test. A key feature of the *test case generator* is its ability to generate test cases even when the state machine has circular references (see Figure 5). More complicated state machines (featuring hierarchies, concurrency, and nested states) can also be represented by this approach [7], and provides a topic for future study. The def-use information of each data member in the test case tree nodes can be listed in a data file by the same test message generator (discussed in section 5.) Using stacks to automatically detect the data anomaly is another topic. Moreover, it is also worth studying if this du path information can support test data selection for the test cases, generated from MACT.

The MACT tool can generate all possible intra-class test cases has been discussed, and we have also followed *data-flow* criteria to compute the du paths with respect to the data members in the intra-class test cases. In order to insure the class under test is satisfied with *data-flow* coverage. We suggest that the steps of object oriented class testing are:

(1) The *data-flow* testing is employed at intra-method and inter-method levels.

(2) At intra-class level, each class is tested as a unit with test cases based on state-based criteria.

(3) The definitions and references of the data members within the sequences of test cases need to be computed and detected.

References

1. Beizer, Boris "Software Testing Techniques," Van Nostrand Reinhold, 1990.

2. Harrold, Mary J. and Rothermel, G. "Performing Data Flow Testing on Classes," 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dec. 1994, 154-163.
3. Siegel, Shel 1996, "Object Oriented Software Testing-A Hierarchical Approach," John Wiley & Sons, Inc. 1996.
4. Tsai, Bor-Yuan; Stobart, Simon and Parrington, Norman; “A Method for Automatic Class Testing Object-Oriented Programs Using A State-Based Testing Method,” 5th European Conference Software Testing Analysis and Review 1997, (EuroSTAR '97,) Edinburgh, UK, 403-415.
5. Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman and Mitchell, Ian "An automatic Test Case Generator Derived from State-Based Testing," 5th Asia-Pacific Software Engineering Conference 1998, (APSEC '98,) Taipei, Taiwan, 270-277.
6. Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman and Mitchell, Ian "A Hybrid Object-Oriented Class Testing Method - Based on State-Based and Data-Flow Testing," accepted for inclusion in 7th Annual International Conference on Software Quality Management (SQM '99), March, 1999, Southampton UK.
7. Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman and Mitchell, Ian "Automated Class Testing Using Threaded Multi-way Trees to Represent the Behaviour of State Machine," to appear at Special Volume on Software Reliability, Testing, and Maturity, Annals of Software Engineering, Vol. 8, 1999.
8. Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman and Mitchell, Ian "Performing State-based Testing and Data-flow Testing on Object-Oriented Classes," accepted for inclusion in The 15th International Conference on Advanced Science and Technology (ICAST '99), April, 1999, Argonne, Illinois
9. Chow, T. S. "Testing software design modeled by finite state machines", IEEE Transactions on Software Engineering SE-4(3): 1978, 178-187.
10. Turner, C. D. and Robson D. J. "A Suite of Tools for the State-Based Testing of Object-Oriented Programs," Tech. Report: TR-14/92, U. of Durham 1992.
11. Binder, R. V. "State-based testing", Object Magazine, July-Aug 1995, 75-78.
12. Turner, C. D. and Robson, D. J. “The State-based Testing of Object-Oriented Programs,” Conference on Software Maintenance 1993, 302-310.
13. Harrold, Mary J. and Soffa, Mary L. "Interprocedural Data Flow Testing," 3rd Testing, Analysis and Verification SYMP (SIGSUFT89), 1989, 158-167.
14. Laski, J. and Korel, B. "A Data Flow Oriented Program Testing Strategy," IEEE Transactions on Software Engineering, SE-9(3), May 1983, 347-354.
15. Rapps, S. and Weyuker, E. J. "Selecting Software Test Data Using Data Flow Information," IEEE Transactions on Software Engineering, SE-11 (4), April 1985, 367-375.
16. Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman and Mitchell, Ian "Selecting Intra-class Test Cases Based on State-Based Testing and Data Flow Testing Methods," Occasional Paper, CIS-4-98, School of CIS, University of Sunderland, UK, 1998.
17. Kit, Edward "Software Testing in the Real World," Addison-Wesley, 1995.
18. Myers, Glenford J. "The Art of Software Testing," John Wiley & Sons, 1979.
19. Pande, H., Landi, W. and Ryder, B. "Interprocedural def-use associations for C systems with single level pointers," IEEE Transactions on Software Engineering, 20(5); 1994, 385-403.
20. Hong, H. S., Kwon, Y. R. and Cha, S. D. "Testing of Object-Oriented Programs Based on Finite State Machines," Asia-Pacific Software Engineering Conference '95, Australia, Dec. 1995, 234-241.

21. Spillner, Andreas "Control Flow and Data Flow Oriented Integration Testing Methods", Software Testing, Verification and Reliability, Vol. 2, 1992, 83-98.
22. Frankl, Phyllis G. and Weyuker, Elaine J. "An Applicable Family of Data Flow Testing Criteria," IEEE Transactions on Software Engineering, 14(10), 1988, 1483-1498.
23. Rapps, S. and Weyuker, Elaine J. "Data Flow Analysis Techniques for Test Data Selection," 6th International Conference on Software Engineering, 1982, pp 272-78.
24. Weyuker, Elaine J. "The Complexity of Data Flow Criteria for Test Data Selection," Information Processing Letters, 19(2), 1984, 103-109.
25. Weyuker, Elaine J. "The Cost of Data Flow Testing: An Empirical Study," IEEE Transactions on Software Engineering, 16(2), February 1990, 121-128.
26. Duncan, I. and Doake, J. 'The Use of OO Design Metrics to Indicate Adequate Testing and Maintenance Costs," TR-9801, Dept. of Computer Science, Anglia Polytechnic University, UK, 1998
27. Bieman, J. M., Schultz, J. L.: "Estimating the Number of Test Cases Required to Satisfy the All-du-paths," TAV3-SIGSOFT '89, 1989, 179-186.
28. Parrish, A. S., Borie, R. B. and Cordes, D.W.: 'Automated Flow Graph-Based Testing of Object-Oriented Software Modules,' Journal of Systems and Software, 1993, 23, 95-109.
29. Lafore, Robert " Object-Oriented Programming in C++," Waite Group Press, ISBN: 157169160X, 1998.
30. Tsai, Bor-Yuan; Stobart, Simon; Parrington, Norman "An automatic Test Case Generator Derived from State-Based Testing," Occasional Paper, CIS-1-98, School of CIS, University of Sunderland, UK, 1998.

Appendix A

```
Step 1    Create a pointer queue which can temporarily store tree nodes
Step 2    Create the head node of the tree
Step 3    Add the head node into the queue
Step 4    While not stop building the tree
Step 5        Create a new node and fill the required information in each field
Step 6        While the new node is not a child of the first node in queue
Step 7            Delete the first node from the queue
Step 8        Link the new node as a child of the first node in the queue
Step 9        If the state_name of a node in the queue is the same as the new node's
Step 10           the pre of the new node threads to the node in the queue
Step 11       Add the new node into the queue
```

This algorithm can build test case trees, such as a tree illustrated in Figure 5. A complete C++ program, coded following the algorithm, is shown in [30].

A Software Engineering View of Data Quality

Mónica Bobrowski
Universidad de Buenos Aires

Joint Work with
Martina Marré and Daniel Yankelevich

Outline

- Motivation
- The Data Quality Problem
- What is Data Quality?
- Software Engineering and Data Quality
  Measuring Data Quality
  Testing Data Quality
  Data Quality in the Software Development

  Conclusions and Future Work

Which one is my name?

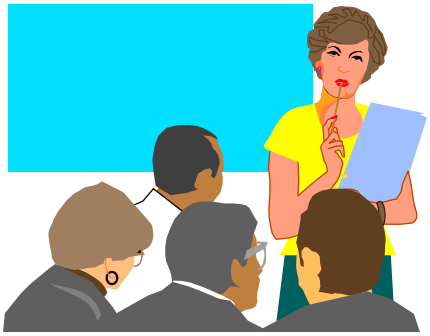
- Mónica Bobrowski
- Monca Bobrowski
- Mónica Bobrowsky
- Mónica Bobrovsky
- Mónica Brobrovsky
- Mónica Bobrosky
- Mónica Bovrosky
- etc.

The client point of view

I could not sign for a new house because I was found in Fidelitas records as not trustable. Of course it wasn't me!! I've put a lawsuit against them. They made me lose a lot of money!

The company point of view

We are having troubles with our people identification system. We are loosing our clients confidence and we are loosing lots of money too!



The Data Quality Problem

- Even with good software systems (and marketing staff) we can get bad results
- Organizations cannot use their systems because of the data
- Economic impact of poor quality data
- *Bad data is a problem that has to be*

What is Data Quality?

- Quality: a relative concept
- Data attributes:
  - Accuracy
  - Timeliness
  - Usability, etc.
- Quality Data does not necessarily mean

Software Engineering and Data Quality

- *"The system works perfectly. Of course, if wrong data is being loaded, what can the*

- Poor system design may lead to bad data

Software engineering has to deal with data quality problems!!!

Software Engineering and Data Quality

- Software Engineering has been dealing for long with quality problems

Product and Process Quality

How can we deal with data quality

Data Quality in Software Engineering

- Measuring Data Quality
- Testing Data Quality
- Data Quality in the Software Development

We want to use existing techniques!!!

Measuring Data Quality

- “If you can’t measure it, you can’t manage

  Value of the information

  How and what to improve

Measuring Data Quality

- Identify interesting attributes (dimensions, Wang, Strong and Lee, 1997)

  Use existing techniques (e.g., GQM,

  Measure the quality of the data instance and

Testing Data Quality

- “Data is not a problem for software

  System testing concentrates on system

  But systems may use data generated by

Testing Data Quality

- Verify data quality independently of the systems that may use it
  - Complete validation of all data
    Statistical indicators
    Testing techniques to define and execute test

Using Known Software Testing Techniques

- Define testing criteria based on quality

Notion of coverage

Construct test cases

Need for an Oracle

≈ Structural Testing?

Data Quality in the Software Development Process

- We have functional and non-functional requirements in traditional software

We define them at the early stages of the development process, and verify them

Data Quality Requirements are Non-functional Requirements

Data Quality Requirements

- Define them at the early stages of the development process

  Using formal notations

  Incorporate them to the final product

  Verify them using:
  - Data Quality Metrics
  - Data Testing

Conclusions and Future Work

- Organizations need quality data in order to

  Some data quality dimensions may be incorporated to software systems

  Software engineering is able to contribute to

Conclusions and Future Work

- Research program:
  - Define a set of metrics (what and how to

    Define data testing criteria based on dimensions

    Define data quality requirements as non-functional requirements

- *Empirical Validation is Mandatory!!!*

A Software Engineering View of Data Quality

Mónica Bobrowski, Martina Marré, Daniel Yankelevich

Departamento de Computación, FCEyN, Universidad de Buenos Aires, Argentina

{monicab,martina,dany}@dc.uba.ar

Abstract

Thirty years ago, software was not considered a concrete value. Everyone agreed on its importance, but it was not considered as a good or possession. Nowadays, software is part of the balance of an organization. Data is slowly following the same process. The information owned by an organization is an important part of its assets. Information can be used as a competitive advantage. However, data has long been underestimated by the software community. Usually, methods and techniques apply to software (including data schemata), but the data itself has often been considered as an external problem. Validation and verification techniques usually assume that data is provided by an external agent and concentrate only on software.

In this work, we present different issues related to data quality from a software engineering point of view. We propose three main streams that should be analyzed: data quality metrics, data testing, and data quality requirements in the software development process. We point out the main problems and opportunities in each of them.

Keywords: Software Quality, Data Quality, Software Engineering.

1. Introduction

Thirty years ago, the software owned by an organization was not considered a concrete value. Everyone agreed on the importance of software, on its virtual value, but it was not considered as a good, as a possession. In those days, the value of software was defined by its cost.

Nowadays, software is part of the balance of an organization, it contributes to its value, and for almost every software project the ROI is calculated. Data is slowly following the same process. In fact, people is now talking about "the value of information." Many organizations want to possess information. Managers know that having the right information at the right time may lead them to obtain great benefits. Moreover, organizations have information that may help them to improve their work, make decisions, and increase their profits. This information is usually stored in large databases accessed via software applications. However, it is not enough to have good applications; an organization needs good data in order to achieve its goals.

Now, how could an organization know that it has the right information at the right time? How could an organization evaluate its information? That is a matter of *data quality*. In fact, the quality of information is crucial when determining its usefulness. When quality is not achieved, information is not used, or leads to incorrect decisions, and even loss. As it is known, "decisions are no better than the data on which they are based" [Red98]. But, what does information quality mean?

In recent years, researchers have been studying data quality problems from the perspective of the data generation processes [SLW97, SLW97, WW96]. They have identified problems in data, and tried to associate them with problems in the process that lead to this data. The underlying idea is that improving the process

may lead to an improvement in data.

In [Red96], Redman gives a deep introduction to DQ issues. He points out many aspects of data quality: definition, management, policies, experiences, requirements, measurements, etc. Although his approach differs from ours (it is mainly statistical, and concentrate on the data generation process), his book offers clear examples, motivations, definitions, and useful tips.

Data quality problems are well known to practitioners. In fact, many failures of software are not due to poor quality of the systems, but to inconsistencies or other problems in data. The quality of data has a great impact on the usefulness and overall quality of software systems.

However, the mainstream of Software Engineering ignored data quality issues up to day. Validation and verification techniques exist and have been used for software processes and products, but few has been done related to data. The only concern for computer engineers regarding data quality has been the extraction of data for data warehouses. In the context of Data Warehousing, an European project investigated quality and in particular the requirements on data needed to implement a data warehouse [JV97].

In our view, software engineers must take into account data quality issues in the design, validation, and implementation of software systems. Moreover, standard techniques can and should be applied to these problems. In this paper, we present different issues related to data quality from a software engineering point of view. We point out three main streams that should be analyzed and the main problems and opportunities in each of them. These themes are:

- Data quality metrics

Measuring the quality of the information will help us to know its value, and also its pitfalls. We will be able to know how valuable our information is, and also what we need to improve in order to increase quality. Moreover, measuring quality would clarify the goals of a quality improvement strategy or process. We agree with the well-known proverb: "if you can’t measure it, you can’t manage it."

- Data quality and testing

Usually, testers and engineers assume that the data (in a production environment) is correct, and test the system considering its behavior. However, as we have said, this is not the case in the real world. When a new system is incorporated to an existing environment, the data it uses must be analyzed to understand its usefulness. Moreover, an old system may be using corrupted data. We believe that the verification of a system must include the verification of the data it works on. Besides, we believe that many testing techniques can be adapted in order to be used to test data.

- Data quality in the software development process

The dimensions in which data quality is analyzed (for instance consistency, accuracy) can be considered data quality requirements for a project, and can be assessed from the beginning of the software development process in the same way that we have functional and non-functional requirements. In fact, they are a particular sort of non-functional requirements. So, we want to deal with them from the beginning of the process, and incorporate them to our specification, our design, and our system implementation. In this case, our system should give us a warning when the data is in close or already in a “bad quality” situation with respect to our requirements. And hence we can prevent our system from entering such a situation.
In Section 2 we discuss what data quality means. In Section 3 we describe typical data quality problems. Section 4 presents data quality metrics: why, what, and how to measure. Section 5 is devoted to data testing. Section 6 points out why data quality requirements should be incorporated to the software development process. In section 7 we present our conclusions and future work.

2. What is Data Quality?

It is difficult to give a universal definition of what quality means. When we talk about quality we do not always refer to the same concept. We will try to exemplify this issue. Suppose that you are planning a trip to a foreign country. You have to choose an airline to fly. Which one do you prefer? Of course, you will prefer the airline that offers the *best* quality. But, what does quality mean? You want to arrive in time, you want comfortable seats, you want a helpful crew, you want a quiet trip, and you want low prices. These attributes

(punctuality, comfort, helpfulness, peace, low prices) constitute your notion of quality in this particular context. Even in the trip situation, someone else may not be concerned about the price, but be very worried about the meals served. So his notion of "airline quality" is different from yours. It may differ not only in the attributes taken into account; the relevance of each item may be different. Moreover, you could have different notions of "airline quality" for different trips.

This example shows that quality is not an absolute concept. The word *quality* by itself has not a unique connotation. We have to make assumptions on which aspects apply on a particular situation. In the case of data quality, we may want to take into account only specific attributes with some specific relevance, depending on the particular context we are analyzing. In our view, the quality of data in the context of software systems is related to the benefits that it might give to an organization.

As we have said, the quality of data depends on several aspects. Therefore, in order to obtain an accurate measure of the quality of data, one have to choose which *attributes* to consider, and how much each one contributes to the quality as a whole. In what follows, we present several attributes that we think may determine the quality of our data. These attributes or *dimensions* have been taken from [WW96, SLW97] following the point of view of the value of the data, i.e., our pragmatic view of data quality.

We present an informal definition for each of the attributes considered. This selection is not exhaustive, but is representative enough for our purposes.

Completeness	Every fact of the real world is represented. It is possible to consider two different aspects of completeness: first, certain values may not be present at the time; second, certain attributes cannot be stored.
Relevance	Every piece of information stored is important in order to get a representation of the real world.
Reliability	The data stored is trustable, i.e., it can be taken as true information.
Amount of data	The number of facts stored.
Consistency	There is no contradiction between the data stored.
Correctness	Every set of data stored represents a real world situation.
Timeliness	Data is updated in time; update frequency is adequate.
Precision	Data is stored with the precision required to characterize it.
Unambiguous	Each piece of data has a unique meaning.
Accuracy	Each piece of data stored is related to a real world datum in a precise way.
Objectivity	Data is objective, i.e., it does not depend on the judgment, interpretation, or evaluation of people.
Conciseness	The real world is represented with the minimum information required for the goal it is used for.
Usefulness	The stored information is applicable for the organization.
Usability	The stored information is usable by the organization.

Notice that dimensions may be related to others. For example, the amount of data may be important only in conjunction with correctness (lot of incorrect data has no sense, and even may damage the organization), usability (inefficient access to data due to the size of the database is worthless), and so on. In some way, these attributes complement each other.

3. The Data Quality Problem

- I can´t use this application. Look, I know that wells in this field are at most 3000 feet, and the depth of this well in the system is 4500! This system is useless.

- The system works perfectly. Of course, if wrong data is being loaded, what can the computer do?

- I don´t know. I just say that it is not good for me. It makes me loose more time looking for data than before.

- It is not our problem. The system works, we detect wrong data when it is loaded –and in the cases YOU specified-. It is a problem of the users: you should tell them to use it right.

This dialog, at least in spirit, happened in many places many times. Data quality problems are real problems in most information systems. With different degrees of criticality and deepness, these problems are being treated in many organizations. In most cases, in an ad-hoc way.

For instance, let us consider mailing lists. How many times do you usually receive a brochure for a conference? How many combinations of first, second, and last name have you seen your name on envelopes? This simple example shows how expensive data quality errors can be: mailing can be quite expensive, and using a faulty list, a mailing campaign can be many times more expensive. Moreover, the lost caused by wrong advertising goes long beyond the cost of mailing: customers and potential customers do not trust someone that is not even capable of keep his/her data right. The image of the organization suffers offering and using wrong data.

However, the particular case of names (and, mainly, occidental names) has been extensively studied and many heuristics have been proposed for the problem of determining whether two names correspond to the same person (in general, in the presence of more data, like date of birth, addresses, etc.). Commercial products and algorithms are available to attack (not to solve) this problem. Even though, this particular problem is cause of misuse of systems in several different contexts.

For instance, criminal identification systems determine if a person has criminal records. This information is used, for example, by judges (to decide whether the person has to be punished, and how), and by organizations (to decide whether to hire him). These systems are critical because, in some sense, our future may depend on the quality of the data and the procedures used to recover it. Although a high quality application is used to access the data, the identification is based on several attributes, and sophisticated algorithms are used to match them, it turns out that wrong conclusions can be obtained when bad quality data is present in the system. It has been found that 50-80% of computerized criminal records in the U.S. were found to be inaccurate, incomplete, or ambiguous [Tay98]. This poor quality data may imply send people to jail or not to hire them.

Data quality problems are not only related to pattern matching of persons or organizations. Such problems arise in many different contexts, and the consequences can be disastrous. The cultural change imposed by the use of computers in many different environments, only makes the problem worse. In fact, people trust computers and utilize them as the main source of data: digital information is used minute by minute to take important decisions that affect people lives.

Recently, data warehouse and data mining projects exposed many data quality problems in big enterprises. When the information collected was analyzed or was checked for integrity, some “hidden” problems were detected. For instance, data from different sources was detected to be inconsistent in data warehousing.

The usual (implicit or explicit) position of software professionals facing data quality issues is that “this is not our problem. ” Somehow, information professionals are not responsible of dealing with information.

On the other hand, we have two ideas that contradict that belief. First, some data quality issues are caused by poor design of software systems. In particular, the effect of poor interface design on data quality is direct. For instance, in many cases users of a complex interface with mandatory values have the tendency to choose a random value. If there is a list of values available, users choose the first of the list or the default value.

For example, by studying last year information the managers of a hospital discovered that most of the patients suffered from hemorrhoids. The resources of the next year were assigned on this basis. The number of beds,

nurses, and other resources needed were determined using this information. However, it came out that "hemorrhoids" was the default choice at the check-in application, and clerks selected it because it was difficult to look for the correct choice. This bad data -due to a poor interface design- had terrible consequences on the hospital finances [Tay98].

Another way in which poor design may affect the quality of the data is by failing in a complete analysis of business rules or by not taking into account data quality issues during the requirements analysis phase. In fact, if data quality is a risk, the design of the system must take measures to minimize that risk.

A rule of thumb [Orr98] proposes to improve data quality by increasing the use of the data. Data that is not used cannot be maintained. We agree with this rule. However, several times it has been used to illustrate that problems in the quality of information are not caused by poor design. This is not true. The process of creating and using data must be subsumed in the design of the system. The data flow, the organization of the processes, and the overall design must be created with this data life cycle in mind [Red96]. Not to do so is a modeling and design fault. During the analysis phase, the processes that are automated must be analyzed not only for efficiency: data quality is also a driver when designing the processes and the use of the applications.

The second idea is that many data quality problems can be prevented and deal with by using standard software engineering techniques – adequately adapted or revisited. For instance, configuration management techniques could be used to solve problems with out-of-date data or versioning of information. Standard metric definition techniques could be used to define useful data quality metrics. These ideas are addressed in more detail in the following sections of this work.

4. Measuring Data Quality

The first step to improve data quality and to define methods and techniques is to understand what "good quality" and "bad quality" is. Hence, we need to measure data quality to be able to know how valuable the information is, and how to improve it. Measuring the quality of the information will help us to know its value, and also its pitfalls. We will be able to know how valuable our information is, and also what we need to improve in order to increase quality. Moreover, measuring quality would clarify the goals of a quality improvement strategy or process. We agree with the well-known proverb: "if you can't measure it, you can't manage it". In fact, it is not possible to make serious empirical analysis of techniques or strategies if there is no agreement on how the results will be evaluated.

We propose to measure information quality using metrics defined using traditional software engineering techniques. Metrics have been deeply studied in software engineering [FP97], so we want to take advantage of it.

In [BMY98] we present a framework for defining and using data quality metrics. The outcome of this work is a suitable set of metrics that establish a starting point for a systematic analysis of data quality. We identify the attributes we want to measure, and obtain a set of metrics and techniques to calculate them. This is a starting point for a systematic analysis of data quality, that may lead to improve the quality of the data in an organization.

We base our work on the GQM methodology [BR88]. GQM is a framework for the definition of metrics. GQM is based on the assumption that in order to measure in a useful way, an organization must:

- specify goals,
- characterize them by means of questions pointing their relevant attributes,
- give measurements that may answer these questions.

We have chosen this framework because it is a top down approach that provides guidelines to define metrics, without a priori knowledge of the specific measures. There are other approaches for metric definition, e.g., [BBL76, MRW77]. We have chosen GQM because of its simplicity, its adequacy to our problem, and because it is well known and proven in software engineering applications [Van98].

Following GQM, we first are able to state which dimensions characterize our notion of data quality. Then, we can ask questions characterizing each dimension, without giving a precise (formal) definition -that is

sometimes impossible-, only focusing on their relevant characteristics from our point of view. Finally, we give metrics (some objective, some others based on people appreciation) to answer these questions, giving us a more precise valuation of the quality of our data.

We cannot measure data and ignore how it is organized. Certain quality characteristics are related to the organization of data, i.e., to the *data model*, and not to data itself. The data model might affect some data quality attributes, since it defines the way data is accessed and maintained. We want to identify and measure those attributes too, and complement measures of data with information on how it is organized. As a consequence, we defined two kinds of metrics: set of data metrics, and data model metrics.

Once we have defined our data quality metrics (i.e., what and how to measure) we want to use them. We can simply take our relational database, identify the dimensions we are interested in, choose the appropriate metrics and techniques depending on specific considerations, apply them, and analyze the results. This is a useful approach, specially when the system is already in production, the database is implemented, there is a lot of data loaded, and we want to have a picture of the current situation in order to decide what to improve. We may even add information about the quality of the data to the meta model, as part of its definition. This way it may be easier to check and evaluate the quality of the data at a certain point. In [JV97], this approach is followed in the data warehouse case.

Once we have measured the quality of our data with respect to the chosen dimensions, we can decide whether or not our current data satisfies our quality expectations. Moreover, we will know in which dimension it fails (although we do not know why), with respect to which specific aspect, and we have a measure of the "badness." So we can concentrate our efforts in solving that particular problem, and we can decide if it is convenient to do so - may be data is not so bad, and the solving effort is worthless.

This procedure only deals with measuring the quality of data at certain points, and can help in deciding which corrective or preventive actions to implement. In order to reach and maintain high levels of data quality, it has to be part of a broader plan, that takes into account all the aspects of data quality in the organization (see [Red96]).

Another approach is to see the dimensions we are interested in as data quality requirements (see Section 6). These requirements can be assessed from the beginning of the software development process, in the same way that we have functional and non-functional requirements. So, we want to deal with them from the beginning, and incorporate them to our specification, our design, and our system implementation. In this case, our system should give us a warning when the data is in close or already in a "bad quality" situation with respect to our requirements. And hence we can prevent our system from entering such a situation. Metrics may be used here to establish the requirements and check them at different stages of the software development process.

5. Testing and Data Quality

Software systems were often analyzed as if they start from scratch. Only recently the idea of using COTS is being incorporated in formal description of the development process. This is even stronger in the case of the data used by these systems. The idea of testing a system concentrates on testing its functionalities: never the data that it is supposed to work with – even if it makes assumptions on what is the state of the data. The phrase "garbage in/garbage out" only expresses the idea of "data is not a problem of software systems."

If a system is started from scratch, some of these assumptions can be accepted. However, in the daily practice of our profession, most systems are incorporated on top of existing systems or collaborating with existing systems. Many projects use data generated by other systems, in many cases by systems that are not operative anymore.

In our view, it is important to check whether the data satisfies the requirements of the system or, in other words, that the quality of data reaches the minimum level required for the system to work properly. This activity can be done before the system is developed (in order to take corrective measures or include extra components during the development), before installing the system (in order to check how it will work and to prevent problems during its use) or, independently of any system, just to measure the quality of the data.

This verification can be done in three different ways:

- Complete validation of all data.

- Statistical indicators of mean, variance, intervals, etc.; or random selection with an associated confidence.
- Use testing techniques to define and execute test cases.

The first way is clear: validate the whole data, using automatic and manual verification. This is not equivalent to clean the database or files, because the cost of repairing a data error can be many times greater than the cost of detecting it. However, in most cases complete validation is unfeasible. In this extent, it is not different of complete validation of programs [How76]: in many cases the domain of programs are finite and could, in theory, be validated for all inputs. Even though, complete verification is not done, because it is too expensive, too complex, or unnecessary. Only the thought of checking a 2,000,000 registries database to see if any customer has changed his address is scaring.

It is important to discuss the difference between the last two options. There are testing techniques based on statistics, and the activity of testing is strongly related to statistical analysis. However, there is a subtle difference between taking values that describe distribution of data on one hand, and choosing particular cases that satisfy particular criteria on the other. When we propose testing as a technique to validate data quality, we think that it is possible to define *testing criteria* for data quality based on the quality dimensions of interest. In some cases, it might be even possible to define the notion of *coverage*, and to construct *test cases* to satisfy a particular coverage criteria [My79].

For example, suppose that we are interested in measuring how accurate our data is with respect to time (timeliness). Let us assume that we know which attributes are time dependent. If we have a way to know if specific values are outdated, we may define a test over the data to estimate the number of records that are outdated. Hence, in order to implement these testing activities we need to use a selection criterion (to reduce the number of test cases to be evaluated) and an *oracle* (to know if specific values are outdated). We believe that in this case, the selection criterion should use the specific information about timeliness of data, improving the results obtained by using sampling. Testing for other qualities should use different information to select data.

Data testing has the flavor of structural testing –because the structure of data will probably play a basic role in defining the criteria- and aspects of functional testing. The type of coverage used to check data quality will be fundamental to create new testing techniques –or adapt existing techniques for new goals. A lot of work must be done to define adequate notions, and those notions must be validated by empirical data (and, possibly, by high quality data!) before proposing concrete techniques. However, it is clear that testing, as presented in this section, has many advantages over statistical analysis. One of the advantages is that we do not need to define the rules that guarantee that a particular piece of data is of high quality explicitly: for each case we can determine whether the output passes or fails the test. The only difficulty is to choose the right tests. But we know how to do that to test programs: the same ideas should apply here.

6. Data Quality in the Software Development Process

The requirements of a software system are usually divided in two groups: functional requirements and non-functional requirements. Functional requirements include the services the system is expected to provide, while non-functional requirements place constraints on the way those services must be provided [Som94]. Examples of non-functional requirements are programming languages ("the system must be implemented using C++"), performance ("the expected response time is 2 seconds"), standards ("the development process must be ISO compliant"), interoperability ("the system must communicate with the accounting system"). Moreover, non-functional requirements may be classified according to the kind of constraints they impose. So, we have process requirements (constrain the development process), product requirements (constrain the final product), and external requirements [Som94].

Besides, when describing non-functional requirements at early stages of the software development process, we assume that they should be verifiable, that is, we want to be able to decide whether the system architecture, the design, the implementation, the process model, etc., satisfy them.

As mentioned in previous sections, we claim that data quality issues are non-functional requirements that may be incorporated into the software development process. In fact, we may add a fourth kind of requirements: *data requirements*. These requirements must be placed at requirements and specification time, and they will constrain the following steps of the development process. In this way, we the system constructed will satisfy

the expected levels of data quality, and we may be able to verify these data quality requirements in the system.

Very often, system developers claim that their job do not consist in understanding what the systems they develop are used for, neither the context in which the systems will be used. They just build systems that meet the requirements of the users; the users have to ensure the quality of the data in the databases [Orr98]. According to our view, if data quality requirements are formulated together with other non-functional requirements, the developer has to guarantee that those requirements are met, and consequently, that the expected levels of data quality are achieved and maintained. Of course, it is not always possible to have an a-priori knowledge of all the aspects regarding data quality, but at least a subset should be available. And the analyst is responsible for asking and obtaining this information.

Users have expected levels of data quality in mind. In fact, they obviously want data in the systems to be used, and this alone constitutes a requirement. Sometimes they have more specific demands, concerning accuracy, timeliness, security, accessibility, etc., of data. These requirements are functional by no means, since they are not related to the services the system provides. However, they are related to how the services will be implemented. For example, if a requirement is placed on the security of the data so that certain data is not accessible to every one, the system must comply with this security requirement in order to satisfy the user expectations. Hence, data quality requirements are non-functional requirements. To include data quality requirements from the beginning of the development process may help to improve the quality of the data during system usage, because the system will be designed to take care of the quality of the data according to the user needs.

Moreover, in information systems the expectations on data quality can be even stronger than the expectations on a particular functionality or operation that the system may perform.

Existent approaches consider data to be independent from the applications that use it [Red96]. This is essentially true. Organizations have information that may be used by many systems, although the data has entity by itself. However, systems are build to use this data in agreement with the rules and needs of the organization. Thus, they have to preserve the consistency of the data, make it accessible, extract useful information, maintain its quality, etc. It follows that applications must take care of data quality. And, as we already know from software development models, it is better to have them in mind from the beginning. It is always more expensive to modify existent systems in order to deal with the quality of the data, to preserve it, or improve it. It is cheaper to include them at the starting point of the development process and verify them at each stage, including the final implementation.

In order to have verifiable requirements, we would rather use formal notations that may allow us to use automatic tools to perform verification of requirements. As opposed to other requirement languages, a language for data quality specification should be decidable and quite simple.

Data quality requirements may be formulated in terms of data quality dimensions [SLW97]. Different requirements may be placed over the same set of data and data model. Hence, we want to be able to decide whether a set of requirements is sound. Moreover, we may place different requirements over different subsets of data.

There are other approaches that deal with requirements on the data at early stages of the software development process [Red96]. However, they do not follow a software engineering approach. In fact, they do not incorporate them as part of standard software development processes; they do not look for a formal, simple, and verifiable notation to describe data quality requirements; they do not apply traditional software engineering techniques to data quality problems.

Strong, Lee, and Wang [SLW97] describe common existing problems with data. They identify their source, the dimensions affected, and the impact on the organization. They propose general solutions to these problems, for instance, as guidelines to the process development and management. They do not formalize the expected data quality levels, and cannot verify if they are achieved. This analysis can be done when the problems are detected, and the experience could be used in future developments.

Redman [Red96] proposes to understand the customer needs prior to the software development. He translates user requirements into technical requirements written in natural language. Some of these requirements are formulated in terms of specific conditions over certain data (for example, “the new address must be in the

system within two weeks"). He determines which dimensions are affected by each requirement. To do this, he uses an impact matrix, where the impact of each requirement is rated as "high", "medium", or "low". It is hard to verify if the technical requirements really correspond to the user requirements, since they are both informal. Also, it is difficult to verify if the technical requirements are satisfied. Moreover, with such a limited scale, it is hard to determine the desired quality levels precisely, and consequently, to verify their achievement.

7. Conclusions and Future Work

Problems in the quality of information are real problems in almost all organizations that use large databases. In this work we have discussed the characteristics of the data quality problem, in particular related to other quality topics usually considered in the software engineering field. Moreover, we have proposed three specific lines in which particular techniques could have a direct impact on how organizations deal with these problems.

This work presents more problems than solutions, it is just a particular point of view to attack data quality problems. In order to obtain concrete results, more work must be done in each of the three themes proposed. In particular, empirical validation is mandatory to check the adequacy of the methods and techniques proposed. Actually, this work can be used as a research agenda, and the lines presented are the basis of our research program on data quality.

The start point of this research program is the definition of metrics for data quality. Without a clear knowledge of what and how to measure, it is difficult to attack the underlying problems or to define objective experiments to check improvement after the use of new techniques [BMY98]. A particular issue related to this point is the value of data. At some point in our program, we would like to have a notion of value of information (in the sense of dollar value or market value), probably related to its use.

Data testing and the incorporation of data quality in the software development process are both issues that must be investigated before defining practical techniques.

One of the main conclusions of this work is that software engineers cannot ignore data quality in the day to day practice, and that many among the best practices of the field can be adapted to work with data quality.

Acknowledgements

This research was partially supported by the ANPCyT under ARTE Project grant PIC 11-00000-01856, and the ANPCyT under grant PIC 11-00000-0594.

References

[BR88] Basili, V.R., Rombach, H.D.: The TAME Project: Towards Improvement-Oriented Software Environments, *IEEE Transactions on Software Engineering*, vol. 14, no. 6, June 1988.

[BMY98] Bobrowski, M., Marré, M., Yankelevich, D.: *Measuring Data Quality*, submitted for publication.

[BBL76] Boehm, W., Brown, J.R., Lipow, M.: Quantitative Evaluation of Software Quality, *Proceedings of the Second International Conference on Software Engineering*, 1976.

[FP97] Fenton, N.E., Pfleeger, S.L.: *Software Metrics - A Rigorous & Practical Approach*, 2nd edition ITP Press, 1997.

[How76] Howden W. E.: Reliability of the Path Analysis Testing Strategy, *IEEE Transactions on Software Engineering*, vol. 2, 1976.

[JV97] Jarke M., Vassiliou Y.: Data Warehouse Quality: A Review of the DWQ Project, *Proceedings of the Conference on Information Quality*, MIT, Boston, October 1997.

[MRW77] McCall, J.A., Richards, P.K., Walters, G.F.: *Factors in Software Quality*, Rome Air Development Center, RADC TR-77-369, 1977.

[My79] Myers G. J., *The Art of Software Testing*, Wiley, New York, 1979.

[Orr98] Orr, K.: Data Quality and Systems Theory, *Communications of the ACM*, Vol. 41, No. 2, pp. 66-71, Feb. 1998.

[Red96] Redman, T.: *Data Quality for the Information Age*, Artech House, 1996.

[Red98] Redman, T.: The Impact of Poor Data Quality on the Typical Enterprise, *Communications of the ACM*, Vol. 41, No. 2, pp. 79-82, Feb. 1998.

[SLW97] Strong, D., Lee, Y., Wang, R.: Data Quality in Context, *Communications of the ACM*, Vol. 40, No. 5, May, 1997.

[SLW97] Strong, D., Lee, Y., Wang, R.: 10 Potholes in the Road of Information Quality, *IEEE Computer*, August 1997.

[Som94] Sommerville, I.: *Software Engineering*, Addison-Wesley, 1994.

[Tay98] Tayi, G.K.: Research Seminar on Data Quality Management, Universidad de Buenos Aires, July 22th, 1998.

[Van98] Van Latum F., Van Solingen R., Oivo M., Hoisi B., Rombach D., and Ruhe G.: Adopting GQM-Based Measurement in an Industrial Environment, *IEEE Software*, pp. 78-86, January-February 1998.

[WW96] Wand, Y., Wang, R.: Anchoring Data Quality Dimensions in Ontological Foundations, *Communications of the ACM*, Vol. 39, No. 11, November, 1996.

Minimizing Testing while Maximizing Failure Detection

Graham Thompson
Vice President of Marketing

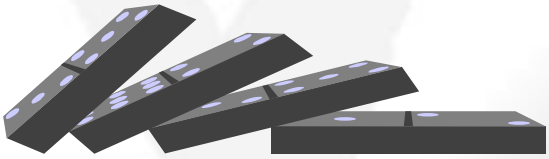
InCert Software Corporation
201 Broadway
Cambridge, MA 02139
617 621 8080

Agenda

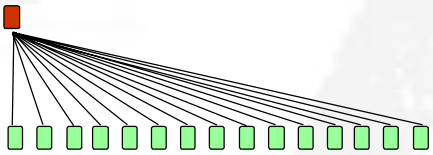
- The Y2K Problem……again!!!!
- Testing Lessons Learned from Y2K
- The New Science of Failure Management
- Looking beyond Y2K
- Summary

Y2K Remediation

- Fixing Date Issue is Simple
- Finding them is Not!
- Making a Single Change is Simple
- Making Thousands of Changes is Not!
- Understand Impact within a Program is Simple
- Understand Impact across the Enterprise is Not!



Program->Impacted Fields



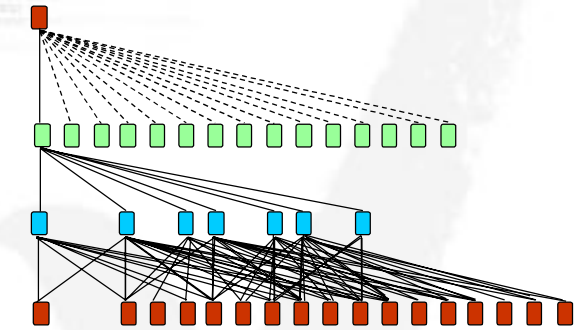
Program

Year2000 Field

Copybook

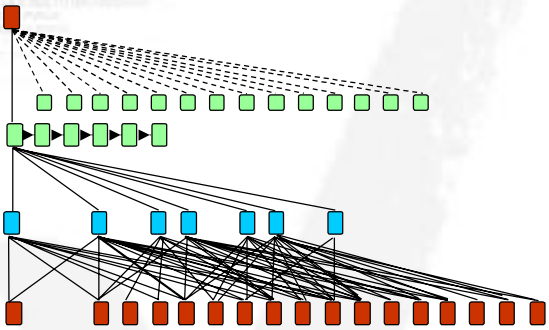
File or Database

Fields ->Copybooks ->Programs



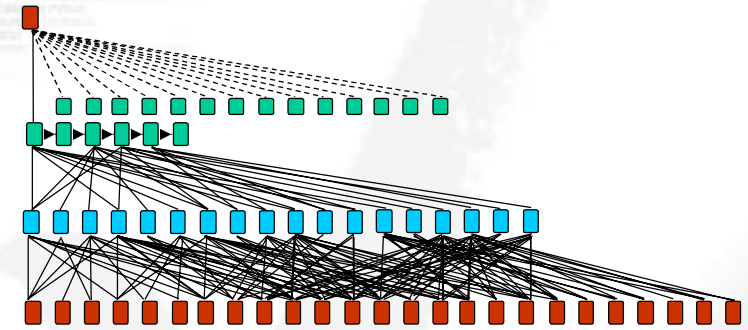
Program Year2000 Field Copybook File or Database

Synonyms within a Program



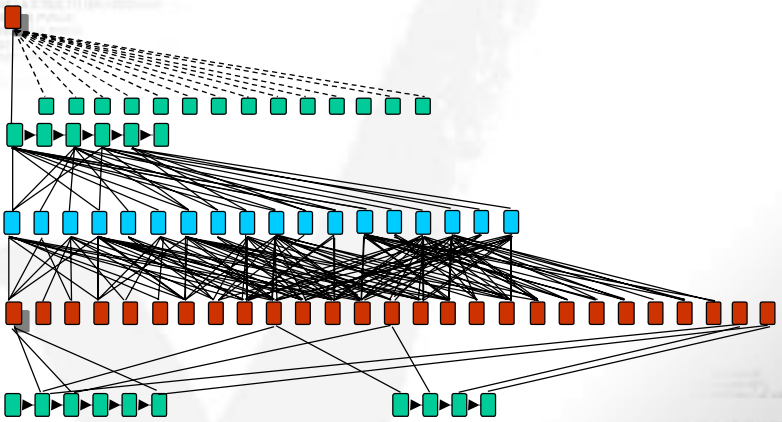
Program Year2000 Field Copybook File or Database

Relationships to the Synonyms



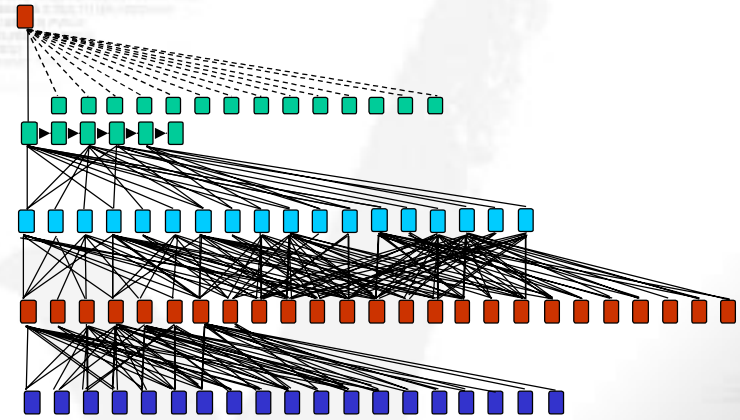
Program Year2000 Field Copybook File or Database

Synonyms through Program Calls



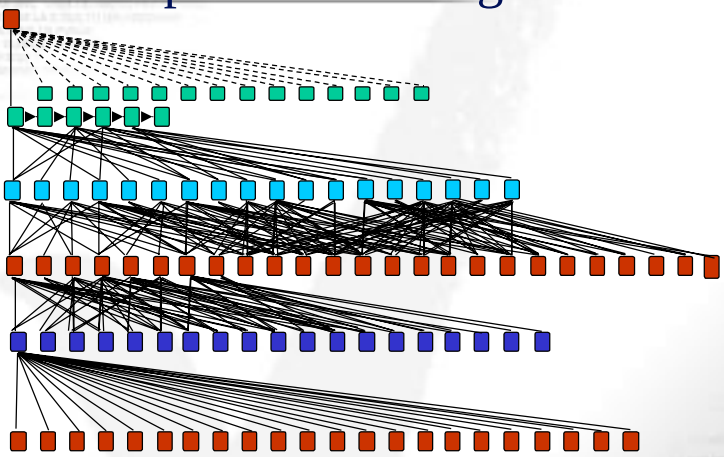
Program Year2000 Field Copybook File or Database

Relationships to Files/Databases



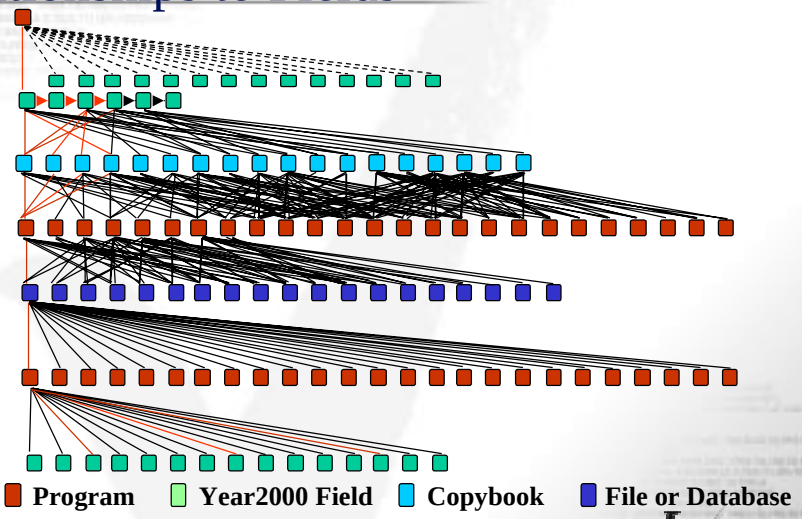
Program Year2000 Field Copybook File or Database

Relationships to other Programs



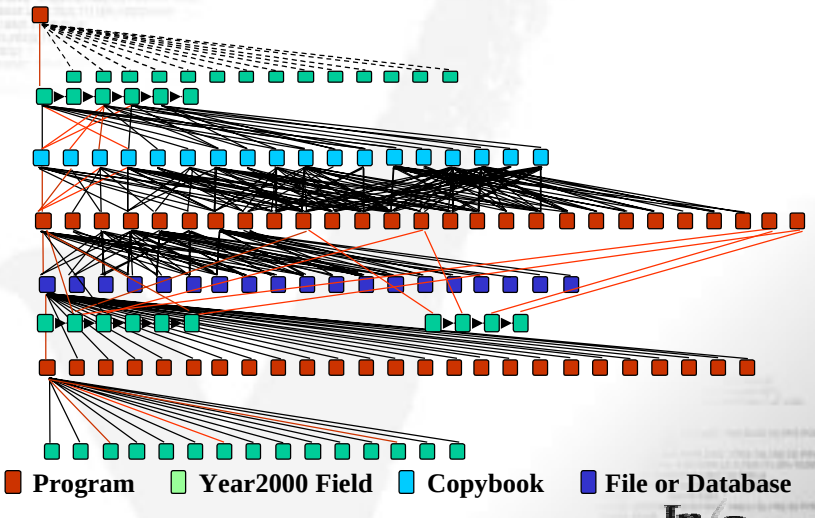
Program Year2000 Field Copybook File or Database

Relationships to Fields



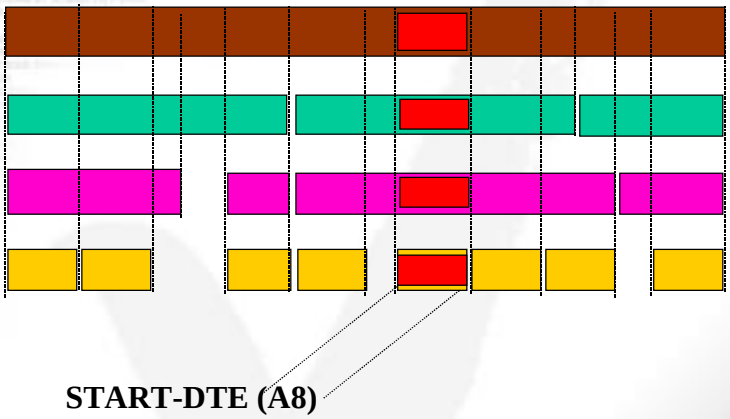
InCert

Relationships through Calls, Databases/Files and Synonyms

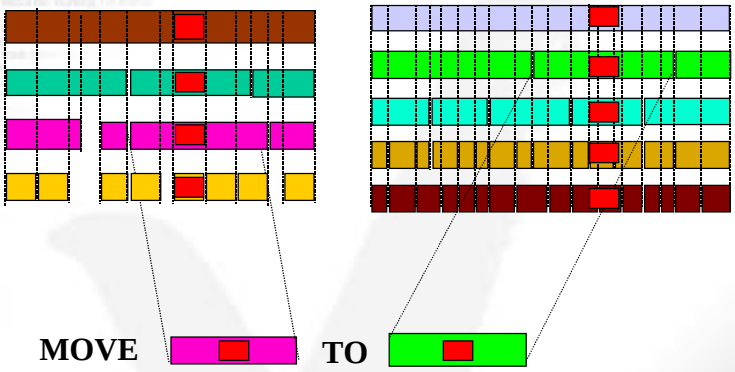


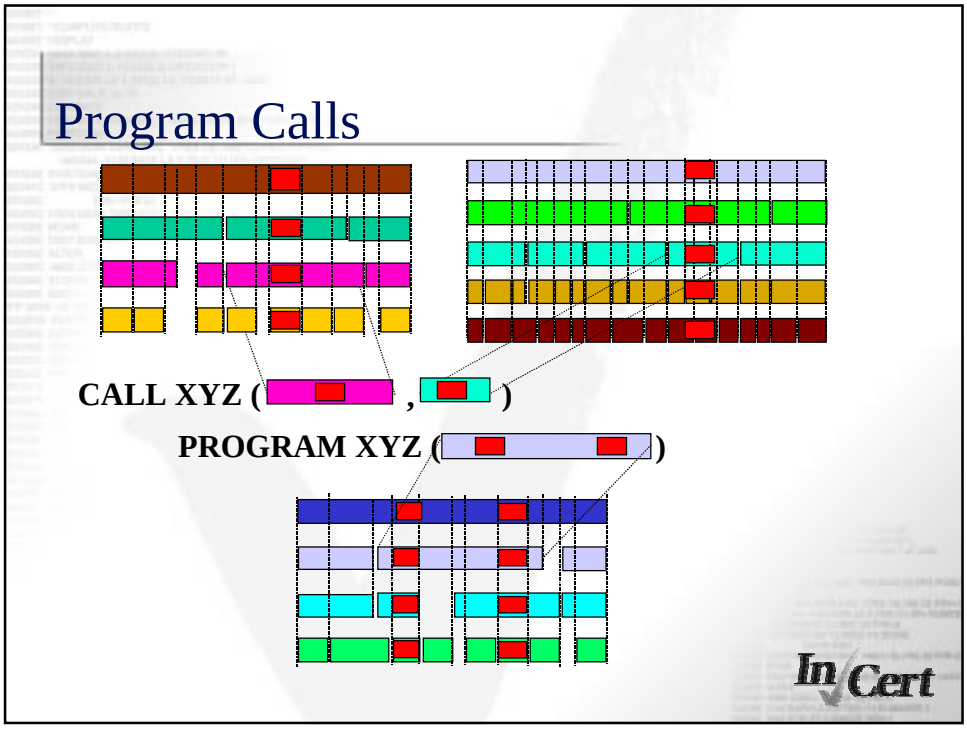
InCert

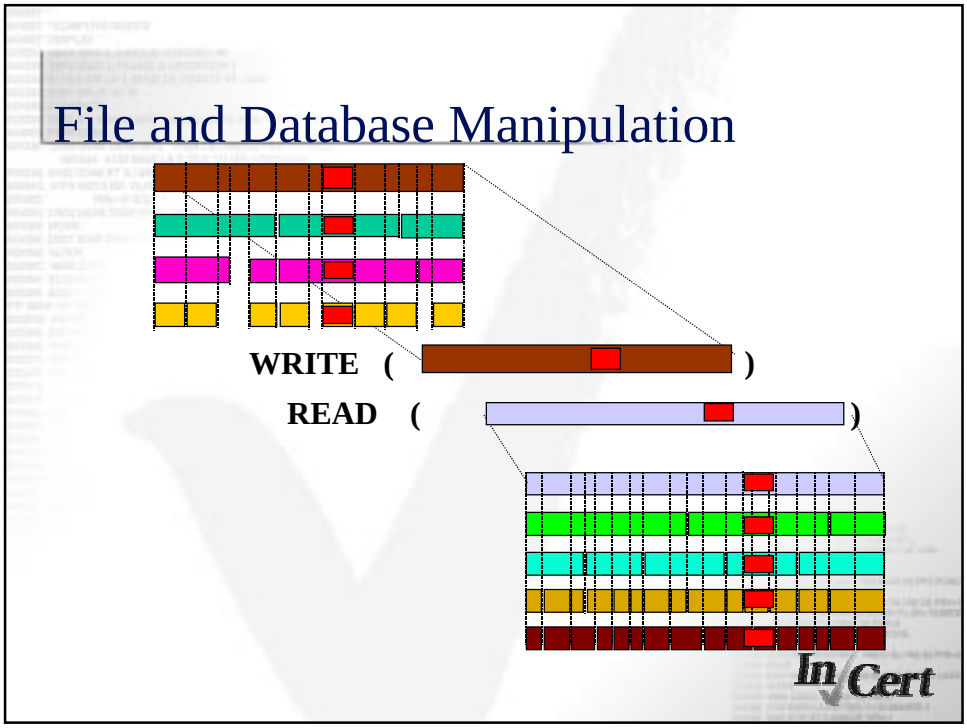
Program Data Structures

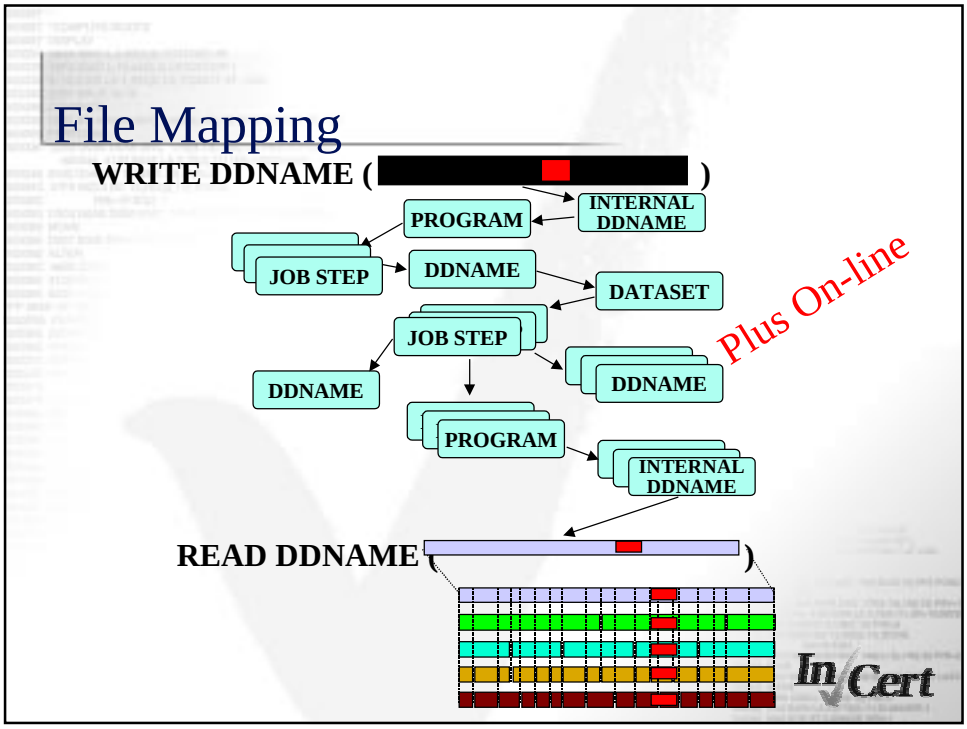


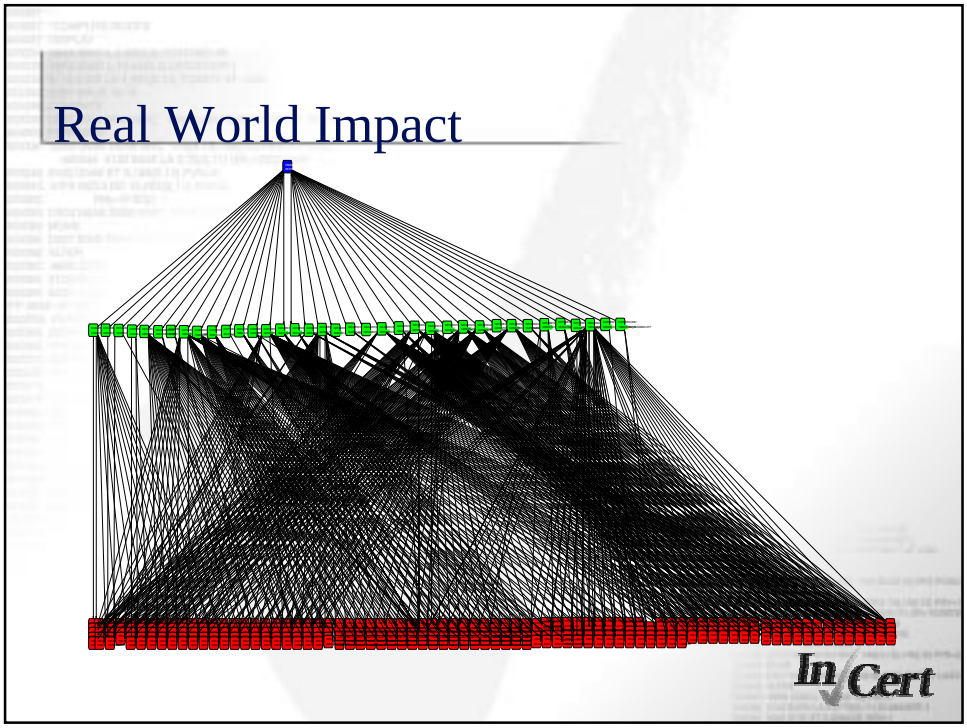
Group Assignments and Comparisons




Program Calls
CALL XYZ (,)
PROGRAM XYZ ( )
InCert


File and Database Manipulation
WRITE ( )
READ ( )
InCert


File Mapping
WRITE DDNAME ( )
INTERNAL DDNAME
PROGRAM
JOB STEP
DDNAME
DATASET
JOB STEP
DDNAME
DDNAME
PROGRAM
INTERNAL DDNAME
Plus On-line
READ DDNAME ( )
In Cert


Real World Impact
In Cert

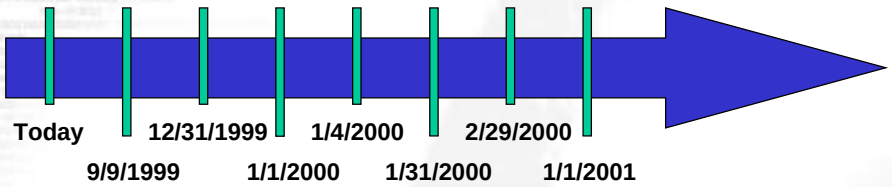
Lessons Learned from Y2K Testing?

- Warp Testing
- Code Coverage
- Regression Management

The Alternative Approach

InCert

Warp Testing

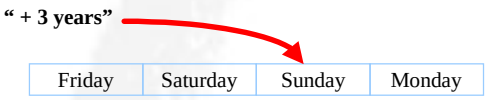


- Common Dates
- Industry Dates
- Instrument Dates

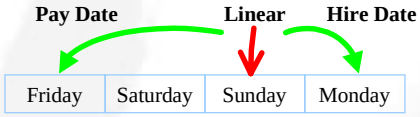
InCert

Aging Your Data

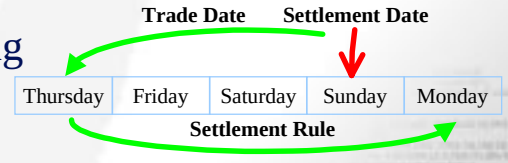
- Linear Date Aging



- Semantic Date Aging



- Business Rule Aging



- Date Classes
  - Holidays - Payday - Special

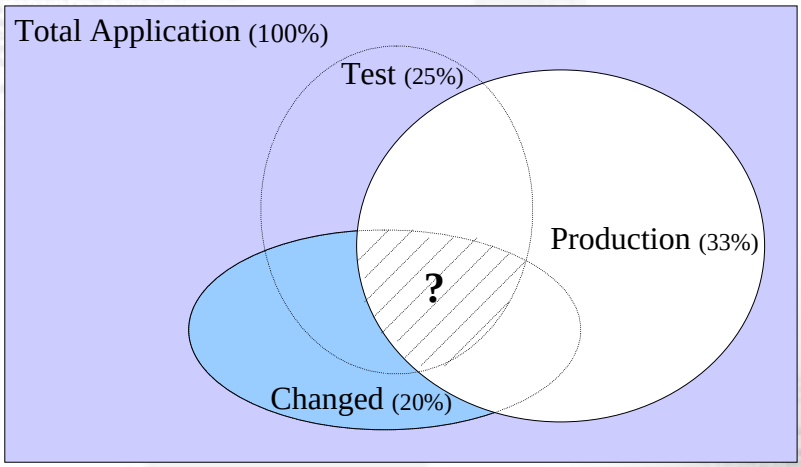
Year 2000 Testing Coverage

- Need to test the very code that has changed
- Need to test the very code impacted by the changes
- Need to regression test the code that runs in production
- Need to minimize the data to achieve the coverage
- Need to repeat testing for each warp date

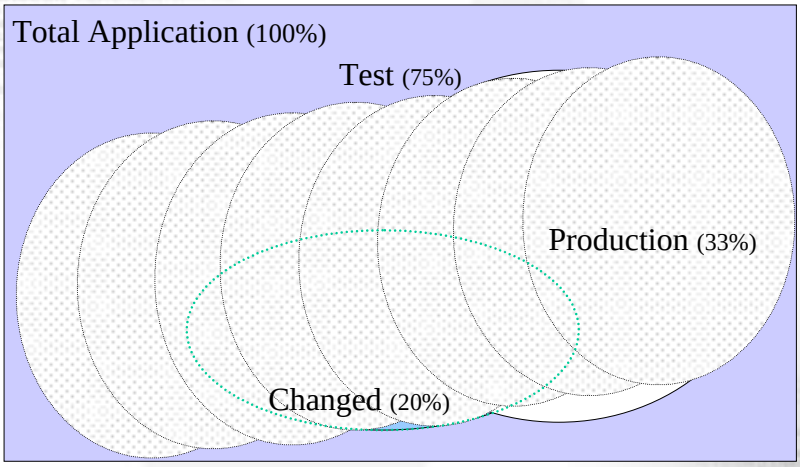
Why testing will not find all failures

- Warp testing only tests identified dates

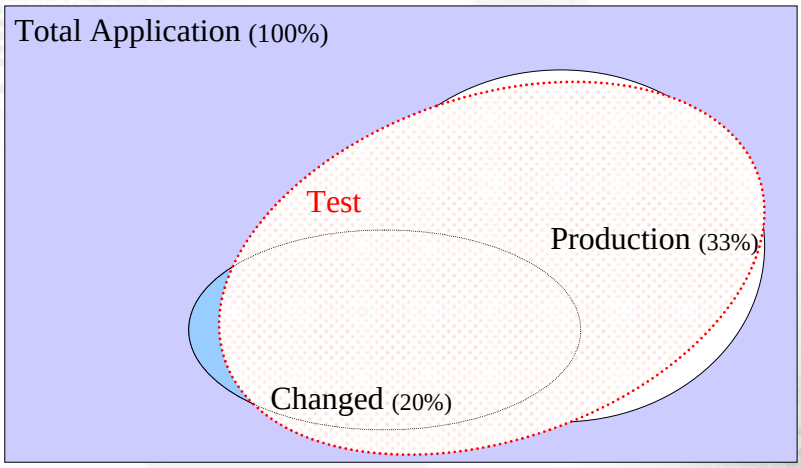
Test Optimization - *Typical Scenario*



Test Optimization - *Scenario 2*

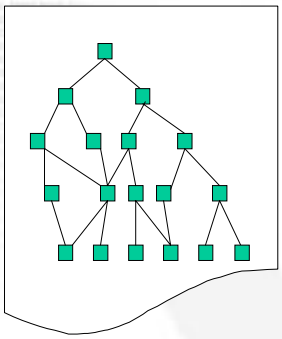


Test Optimization - *The Goal*

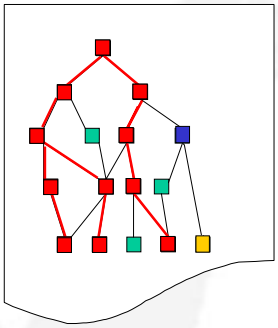


Testing Changed and Impacted Code

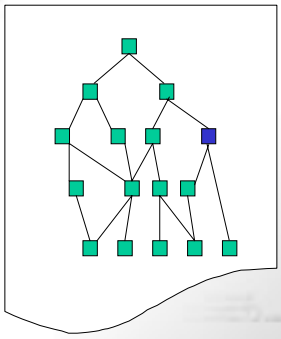
Before Binary



Untested Changed

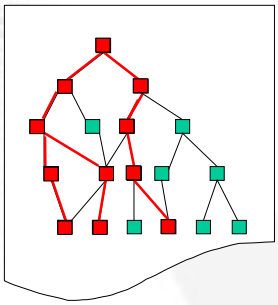


After Binary

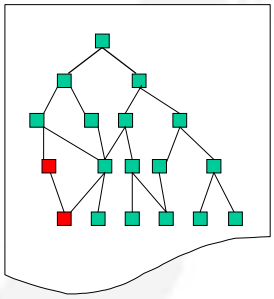


Testing Code Exercised in Production

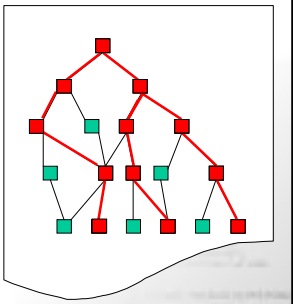
Production



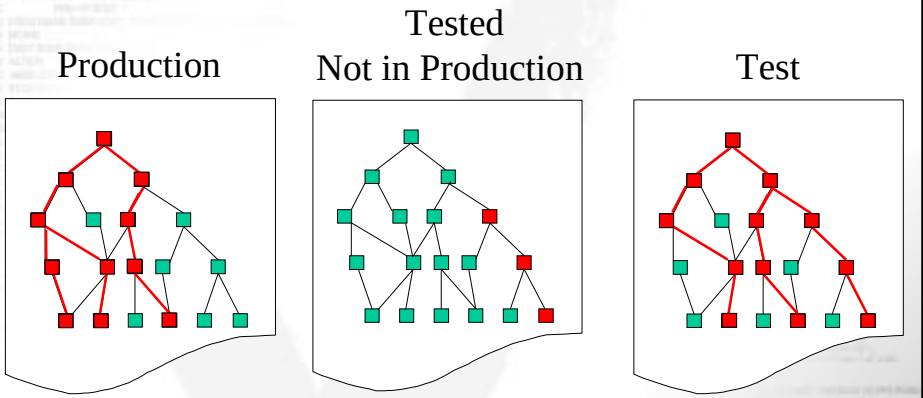
Untested Production



Test

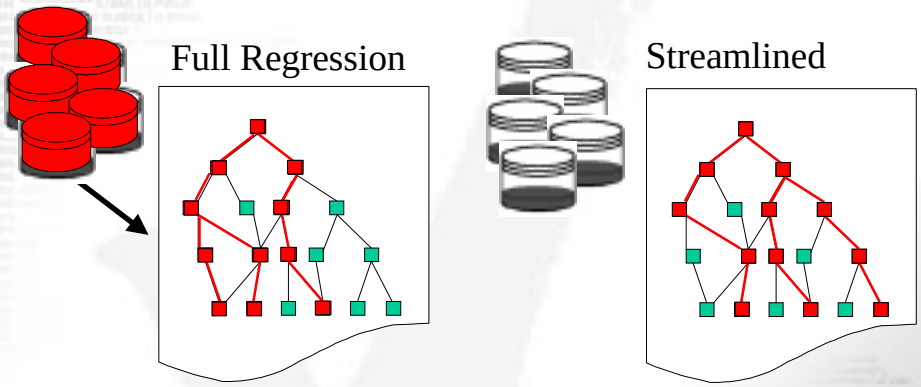


Eliminating Unnecessary Testing



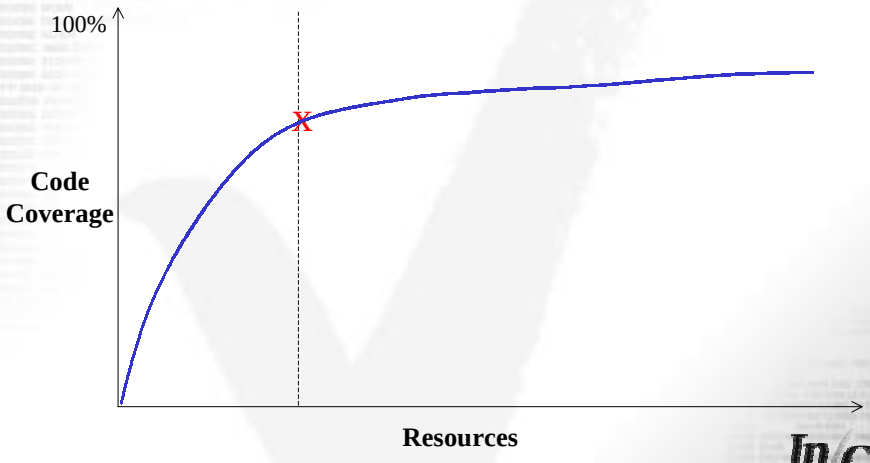
InCert

Minimize Testing while Maximizing Results...



InCert

Test Optimization



InCert

Key Lessons Learned

- The Process
- The Testing
  - Test Regression Suites
  - Suite Coverage determined by Warp Date
  - Need to Accumulate Suite Coverage
  - Total Coverage
  - Test vs Production
- The Results
  - Fixes
  - Comparisons

Interest =
Interest_Rate * (Today_Date - Deposit Date)

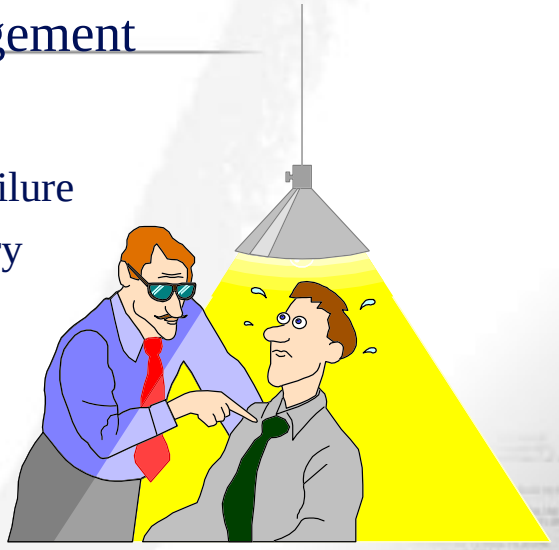
InCert

Audit your Year 2000 testing process

- Determine the levels of:
  - Total Code Coverage
  - Changed and Impacted Code Coverage
- Did you operate the 28 year rule?
- What results comparisons took place?
- What data inspection techniques were used?
- Do you have more work to complete here?

Failure Management

- Failure Today
- Recognizing Failure
- Failure Recovery



Failure Today

- IDC Survey Commissioned by InCert Software
- Over 170 responses from Fortune 2000 Companies
- 60% Management - 40% Technical
- All Large MVS Mainframe Shops
- Survey at www.rresults.com/recover

Number of Failures Today (12 Months)

None	5%
1-4	21%
5-10	15%
11-25	14%
>25	44%

Source: IDC

Recover Time from Failure

	Mean Recovery Hours
Minimum	2
Maximum	16
Average	5

Source: IDC

Highest Maximum was 120 Hours

Available Batch Window

< 1 hour	8%
1-2 Hours	50%
3-4 Hours	26%
> 4 Hours	15%

Source: IDC

Increased Failures due Y2K

More Than	37%
Same	39%
Less Than	24%

Source: IDC

Cost of Application Downtime

- Average Cost of Application Downtime was $70,000 per hour
- 42% of Organizations have cost of over $5M per annum from Application Downtime
- 14% of Organizations have cost of over $2.5M per annum from Application Downtime
- 15% of Organizations have cost of over $1.5M per annum from Application Downtime

Returned to Test Environment

- 76% having read the dump (native or with Abend-AID) returned to the test environment
- ? What proportion of the 5 hours was replicating the problem in the test environment?
- Provide business card for full survey

Recognizing Failure

- The Dump
- The Phone Call!
- Sampling
- New Code Exercised
- Software Behavior Triggers

SYMPTOM DUMP OUTPUT
USER COMPLETION CODE=1022
ASID=001F
PSW AT TIME OF ERROR 078D1000 00007754 ILC 2 INTC 0D
LE ADDRESS=00006C80 OFFSET=00000AD4
NAME=ABEND1
F4
GPR 4-7 00006DAC 00007244 00006D20 0000BFB0
GPR 8-11 000070D0 00007344 00006C80 00006C80
GPR 12-15 00007100 00006EC8 500072D8 500075F4
END OF SYMPTOM DUMP
15.02.11 JOB09249 IEF450I AYERSK STEP - ABEND=S000 U1022 REASON=00000000
TIME=15.02.11
15.02.11 JOB09249 IEF404I AYERSK - ENDED - TIME=15.02.11

Y2K Failure Example

$$\text{Pension\_Multiple} = \frac{\text{Emp\_Retirement} - \text{Emp\_First}}{60}$$

IF Pension_Multiple =< 2/3

THEN Pension = Pension_Multiple * Emp_Salary

ELSE Pension = 2/3 * Emp_Salary

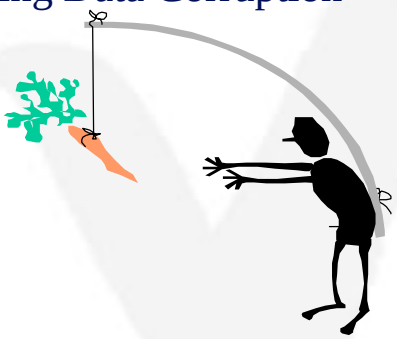
Emp_First = 80 Emp_Retirement = 00 Emp_Salary = 50,000

$$\text{Pension\_Multiple} = \frac{00 - 80}{60} = -1.33$$

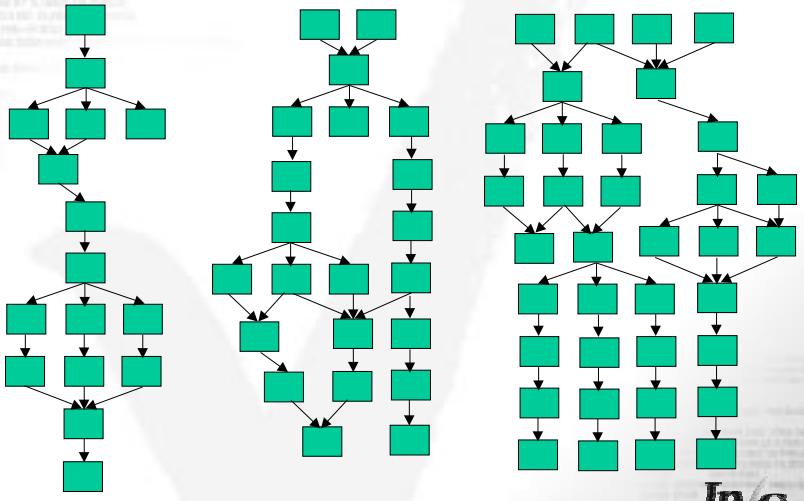
Pension = -1.33 * 50,000 = 66,666

Failure Recovery

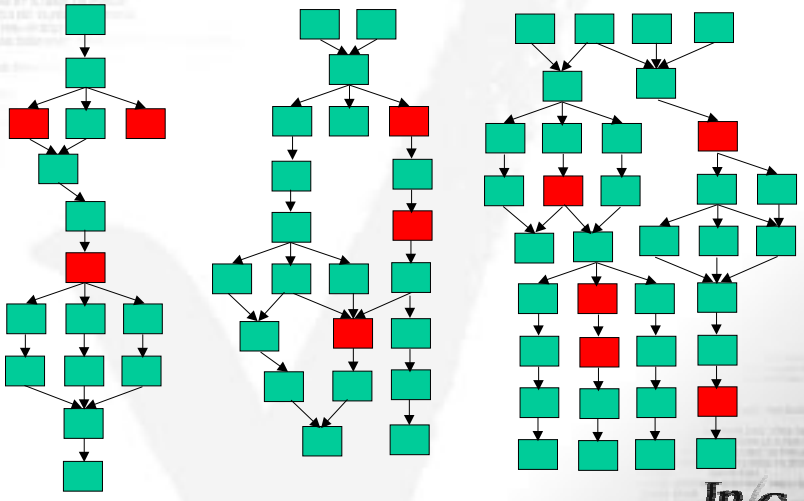
- Failure Manifestations
- Debugging Crashes
- Fixing Data Corruption



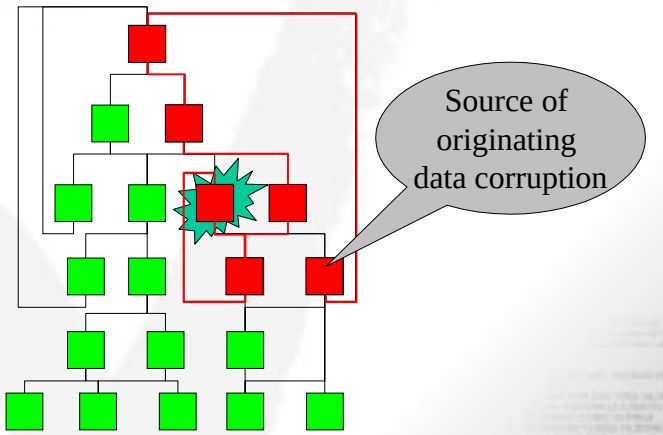
Job Dependencies



Failures in Parallel and Series



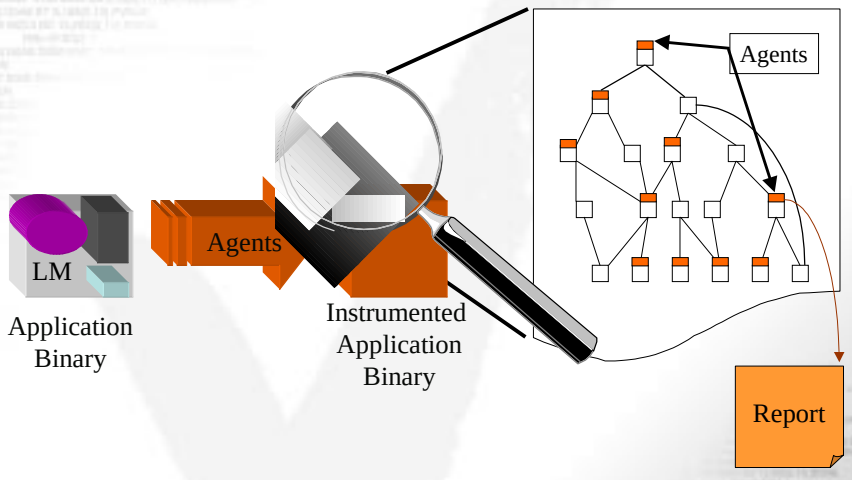
Finding the Root Cause



Need For Dynamic Failure Management

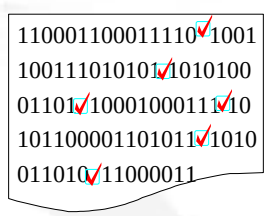
- Dynamic tracing of statement execution
- Trigger writing of traces:
  - Abends
  - Abnormal behavior
  - Negative value in unsigned field
  - Data trigger rules
- Requires light weight - no overhead implementation

Binary Instrumentation

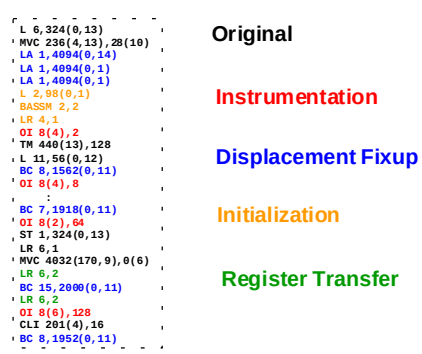


Agent Insertion

Analysis



Instrumentation

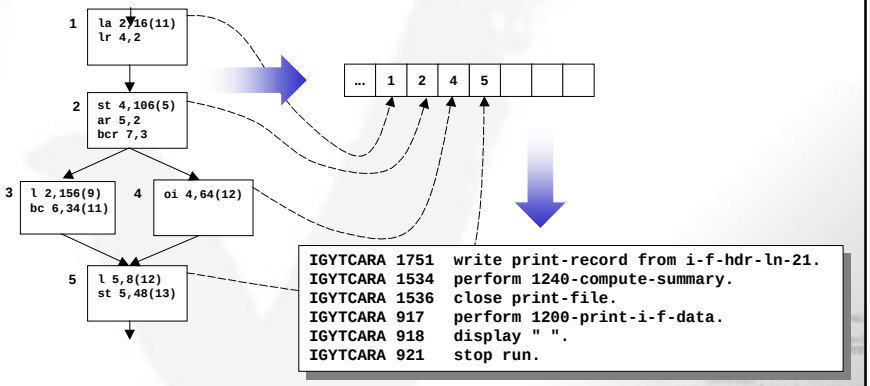


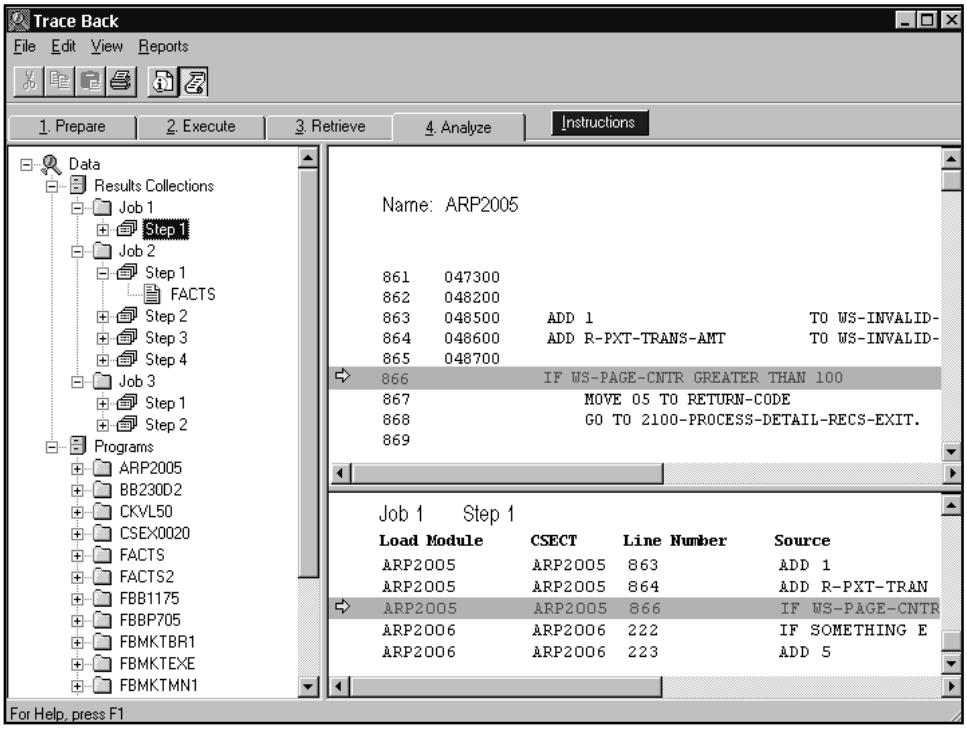
Binary Instrumentation for Failure Management

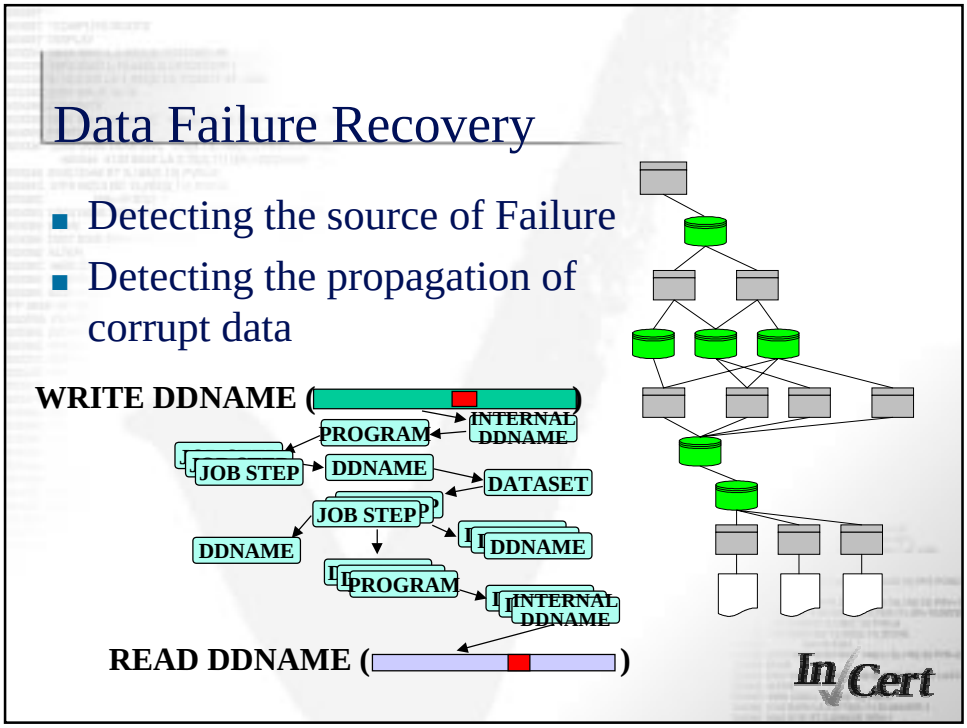
- Create a circular buffer of executed blocks
- At point of crash can walk developer back from point of failure to the root cause of failure
- Complement failure management tools (Abend-AID)
- Negate the need to replicate failure in a test environment
- Expedites the recovery process and reduce application downtime

Dynamic Failure Management

- At each program block, record the progress of program execution.

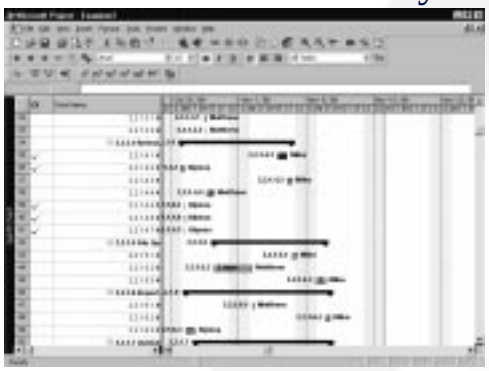



Trace Back
File Edit View Reports
1. Prepare
2. Execute
3. Retrieve
4. Analyze
Instructions
Data
Results Collections
Job 1
Step 1
Job 2
Step 1
FACTS
Step 2
Step 3
Step 4
Job 3
Step 1
Step 2
Programs
ARP2005
BB230D2
CKVL50
CSEX0020
FACTS
FACTS2
FBB1175
FBBP705
FBMKTBR1
FBMKTEXE
FBMKTMN1
Name: ARP2005
861 047300
862 048200
863 048500 ADD 1 TO WS-INVALID-
864 048600 ADD R-PXT-TRANS-AMT TO WS-INVALID-
865 048700
866 IF WS-PAGE-CNTR GREATER THAN 100
867 MOVE 05 TO RETURN-CODE
868 GO TO 2100-PROCESS-DETAIL-RECS-EXIT.
869
Job 1 Step 1
Load Module CSECT Line Number Source
ARP2005 ARP2005 863 ADD 1
ARP2005 ARP2005 864 ADD R-PXT-TRAN
ARP2005 ARP2005 866 IF WS-PAGE-CNTR
ARP2006 ARP2006 222 IF SOMETHING E
ARP2006 ARP2006 223 ADD 5
For Help, press F1


Data Failure Recovery
Detecting the source of Failure
Detecting the propagation of corrupt data
WRITE DDNAME ( )
PROGRAM
INTERNAL DDNAME
JOB STEP
DDNAME
DATASET
JOB STEP
DDNAME
DDNAME
PROGRAM
INTERNAL DDNAME
READ DDNAME ( )
InCert

Failure Contingency Planning

- Plan for Failure
- Plan for Failure Detection
- Plan for Failure Recovery



Looking Beyond Y2K

- Disciplines are being added for Y2K
- Infrastructures are being built
- Build on this investment to reap the rewards well beyond 2000
- Make the process repeatable
- Capture metrics where you can
- Make Fault Management an going task

Summary

- Testing can not be perfect
- Testing without coverage increases the risk
- Organizations should plan for failure
- New tools are required for dynamic failure recovery

Provide business card for Failure Management Research

InCert

Measuring Fault Density in the Real World

A. Frank Ackerman
a.f.ackerman@izdsw.org
Institute For Zero Defect Software

Cherie McKinney
mckinney@adobe.com
Adobe Systems

Audiences and Purposes

- Software Development Managers
  - Motivate to use fault density
  - Information on challenges and solutions
- Software Development Tool Vendors
  - Motivate to develop COTS fault density measurement tools

Agenda

A. The Importance of Fault Density

B. A Case Study

C. Wish List

Institute For Zero Defect Software 3/24/99 3

A. Fault Density for Software Quality Management

- Terms
  - software failures (problem reports)
  - software faults
  - fault density
- Relationship between fault density and failure rate

Institute For Zero Defect Software 3/24/99 4

A. Fault Density for Software Quality Management (cont)

- Faults translate to failures via usage
- Operational profiles not used
- Expected fault density
- Using fault density to manage system test.

Institute For Zero Defect Software 3/24/99 5

B. Case Study

- The product:
  - major commercial product on Unix, Windows and Macintosh
  - globally distributed, Uis in a variety of languages
  - new releases every four or five months
  - Code base of million+ lines
  - C and C++ predominate, but more than 30 different types of files
  - ClearCase™ and Vantive

Institute For Zero Defect Software 3/24/99 6

B. Case Study (cont)

- Want fault density for
  - retired releases
  - currently fielded releases
  - the release currently in system test
- Calculate:

$$\frac{\text{Number Problems}}{\text{Number Code Changes}}$$

- Use of reliability models

Institute For Zero Defect Software 3/24/99 7

B. Case Study (cont)

- Counting code changes
  - problems with eliminating comments
  - problems with formatting changes
  - problems with `diff` "changes"
  - weighting adds and deletes
  - weighting different types of source
  - finding the files

Institute For Zero Defect Software 3/24/99 8

C. Wish List

- Support for Project Work items in the CM system:
  - Product
  - Developer
  - Type
  - Dates
  - Subsytem
  - Files
  - Problems fixed
- Accumulate change counts incrementally

Institute For Zero Defect Software 3/24/99 9

C. Wish List (cont)

- Counting changes
  - between two file versions
  - parse for statements in canonical form
  - processing determined by file attributes
  - user control over new version changes
- Fault Density Report User Interface:
  - problem data by month
  - usage data by month

Institute For Zero Defect Software 3/24/99 10

Measuring Fault Density in the Real World

A. Frank Ackerman
a.f.ackerman@izdsw.org

Institute For Zero Defect Software
Holmdel, NJ
732-264-2671

and

Cherie McKinney
mckinney@adobe.com

Adobe Systems
San Jose, CA
408-536-6240

Acknowledgments

The work reported on here is largely the result of on-going quality improvement efforts at Adobe Systems Inc. The intellectual concepts and techniques draw mainly from the work of John Musa [R1] and other software reliability researchers [R2]

1. Purpose

This paper is addressed to software development managers in medium to large development organizations (10 to 100+ software developers) that develop "versioned" products or a family of similar products.

The purposes of the paper are:

a) to motivate development managers to routinely determine, analyze, and utilize fault density data to improve customer satisfaction,

b) to describe our experience in establishing procedures for determining product fault density of a large commercial software product, and

c) to propose capabilities for source management tools that will simplify the routine collection of fault density data at several levels of detail.

2. Definitions

software failure - any specific instance in which a software product fails to meet customer expectations. Some percentage of observed failures are reported to the product supplier and are eventually registered as problem reports, trouble reports, or some other term. Usually there is a weeding out process that tries to identify repeated reports for the same failure so that only unique failures are logged. Unique failures are often categorized as "enhancement requests" if the problem results from a capability not included in the product requirements, or as a "bug." Bugs are often further classified by three or four levels of severity in term of their effect on customers' operations.

software fault - the source code (including resource files) changes that are made to eliminate a reported failure. All faults are defects that are placed in the product during

the software development process. They are all caused by human error during this process.

fault density - Since software faults are the result of human error during the development process the number of faults that can be expected is directly related to the extent of that process. The number of changes made to the source base is the most often used measure of development process extent. The most common measure is New/Changed Lines of Code (NCLC). So fault density is then

(number of faults)/NCLC

A better measure is discussed below in section 7.

3. References

[R1] Musa, John D., *Software Reliability Measurement, Prediction, Application*, New York, McGraw-Hill Book Company.

[R2] Lyu, Michael (editor), *Software Reliability Engineering Handbook*, New York, McGraw-Hill Book Company.

4. Introduction

One of the software development manager's major concerns is customer satisfaction. A number of factors influence customer satisfaction and many of them cannot be influenced by development management. But one factor, the number of unique failures experienced by customers during the software's lifetime, is usually directly related to the fault density of the delivered software. It is almost certain that the kinds of software we are considering here will have a non-zero fault density. The subject of this paper, then, is the measurement and use of software product fault density to improve customer satisfaction. We discuss

5. Fault Density and Problem Reports
6. Measuring Fault Density for a Large Commercial Software Product
7. Proposed Capabilities for New Tools for Determining Fault Density

5. Fault Density and Problem Reports

An approximate, a priori mathematical relationship between fault density and expected problem reports is described in [R1]. This mathematical model may not apply exactly to the execution environment faced by may software products, but it provides a useful initial estimate that can be improved over time if the relationship between its predicted failures and observed failures remains fairly stable. Typically, for commercial products the number of acceptable problem reports is a fairly broad range, for example 30 to 60, 500 to 700, and so forth depending on the size and importance of the product, so mathematical exactness is not required. It is usually understood that at least a few customer problem reports can be expected, but that there is also a tolerance range beyond which the number of problem reports will eventually adversely affect the development organization's bottom line.

In small development organizations variations in the capabilities of individual developers overwhelms statistics so quality management is largely a question of hiring and retaining the right people, and the hands-on management of details. In larger development organizations organizational and personal experience is generally sufficient for guiding management if the development and business environment is static. In such a static situation the energy and money required to establish more rational and quantitative methods can often be better spent elsewhere. But there are few static situations in today's business environment, and achieving and maintaining high levels of customer satisfaction is essential for survival. In today's business environment the methods discussed here provide a competitive edge for early adopters. In the future they will become a competitive necessity.

It is important to realize that the number of problems reported for a software product depends on the usage of that product. Thus, if a product receives very little use few problems will be reported, if it is used by lots of different people for a few years many more problems will be reported.

Strictly speaking, the number of problems reported for a delivered software product will depend not only on the overall amount of usage, but on the statistical distribution of that usage across the features and capabilities of the product. This distribution is called an *operational profile* [R2]. The development and use of operational profiles is important for the more precise measurement and prediction of problem reports but in many commercial situations the assumption that usage patterns with respect to triggering failures (with the creation of certain data set conditions) is reasonably constant from release to release, even though most of the problems reported will be for new rather than old features. In any case, development management has very little influence on customer usage patterns and fault density thus becomes one of the key factors related to customer satisfaction that is under the direct control of management.

Two fault density measures are important: (1) the fault density of the product when it is initially delivered for system level testing (as opposed to unit or developer level testing), and (2) the fault density when it is released for general customer use.

The <u>process</u> of software development, especially in the design and construction phase, has changed very little over the last five decades of pell-mell industry growth and usually results in a delivered fault density that is closer to 1 fault per 100 NCLC than to 1 fault per 1000 NCLC. The release of software with fault density of 1 per 100 usually results in an unacceptable number of problem reports and so in most organizations system level testing is used to move the released fault density close to 1 per 1000, or even better. Fault density, then, is a key factor not only in determining customer satisfaction, but also in the cost-effective management of system test. Since typically the system test schedule is equal to the design and development schedule, and often takes as much as 50% of the total development budget, the on-going, routine use of fault density provides, for perhaps a 1% increase in development cost, a significant opportunity for a cost/benefit approach to managing the system test effort.

6. Measuring Fault Density for a Large Commercial Software Product

Our purpose in this section is to describe the challenges we faced and the procedures we used to determine fault density on a large commercial software project with current tools. In the next section we propose new tool capabilities that could make fault density determination not only easy and routine, but also provide this key measure for individual subsystems and developers.

The product we measured has the pertinent characteristics listed in the following.

- It is a major commercial product that is available on Unix™, Windows, and MacIntosh platforms.
- It is a globally distributed product that provides user interfaces in several different western and eastern languages.
- The product has a multi-year, multi-owner history with subsystems that were developed both internally and externally.
- There is a major release of the product about every six months and at any given time there are three or four versions in various stages of development.
- The code base is more than a million lines distributed over more than five hundred directories and eight thousand files.
- The predominate languages in the code base are C and C++ but substantial use is made of many other languages and tools (assembly, yacc, etc.) and substantial use is made of resource files that are processed by a number of different support tools.
- The code base in maintained in ClearCase™ from the Rational Software Corporation and the problem reporting system is from the Vantive Corporation.

Conceptually, measuring the fault density in such a product is quite simple. One just needs to know (1) the total number of faults in the product, and (2) the size of the product. In real life, with the tools available in most commercial development organizations, this is not a trivial task. In the next two subsections we described how we obtained these two values.

6.1. Counting the total number of faults

We counted faults simply by counting the number of bugs reported against the product by the development organization from the start of system test, and by customers from the release date onward. This approach overstates the number of faults slightly since it sometimes happens that correcting one failure also corrects another [R1], but the difference is small enough in our case to be ignored. To obtain this count from a problem reporting data base problem records must have fields for:

- date problem reported,
- who reported the problem (development organization or customer),

- the products the problem applies to, and
- whether or not this is a duplicate report.

In the type of environment we are considering development managers typically need to know:

- the fault density of retired releases,
- the fault density of currently fielded releases, and
- the fault density of releases currently in system test.

Assuming that the product size has been determined, to determine the fault density of a retired release that has seen extensive use one needs to know only the total number of problems reported against the product.

For currently fielded releases most of the organization reported bugs are known at the time of release but the number of customer reported bugs will continue to grow significantly as the product is used. Since it is usually important for management to have a good estimate of the fault density of recently released products before the products currently in system test are released, an estimate needs to be made of the total life-time number of bugs of recently released products. Such an estimate can be provided by a software reliability model (we used the Logarithmic Model in [R1]) provided one can estimate product usage. The most accurate measure of software usage is total execution time (adjusted for processors of varying speed) but in many commercial applications it is quite impossible to get a reasonable estimate of execution time. We found that simply taking, month-by-month, the cumulative number of installed system months (estimated by monthly product shipments) gave us a well-behaved software reliability model.

A key question during system test is: How many more faults can be expected to be uncovered and corrected before the product is shipped? If we know the number of Problem Reports logged by system test so far, the size of the product, and the average fault density in recently released products then we can make a reasonable estimate of the number of bugs that still need to be found and corrected before the product will have a fault density at least as good as recent releases. This allows a rational, quantitative approach to managing resources during system test and provides important information for a cost/benefit analysis of releasing early.

6.2. Determining the size of the product

Initially we determined the fault density of a recently released product. Our biggest challenge was determining the size of the product. To do that we needed:

a) to determine the ClearCase view of the base product, *i.e.*, what development started with,

b) to determine the ClearCase view of the product at the time it was released,

c) to determine which subtrees were relevant since in our case a single ClearCase view was used for a number of development purposes,

d) to determine the relevant directories and the relevant files within the relevant subtrees,

e) to determine our unit of size for each type of file,

f) to determine how we would recognize new and changed units,

g) to determine how we would track files from the base view to the release view, and

h) to determine how we would get a single composite NCLC from separate deleted and add counts.

Since ours was an exploratory project in which the major objective was to establish the cost/effectiveness of fault density measurement for managing quality, we sought the most expedient resolution to each of the above issues that would provide what we felt was reasonable data.

Items (a) and (b) were resolved by consulting one of the lead development managers and having him select the ClearCase views most suitable for our purposes. This manager also selected the relevant subtrees in these views for item (c).

A further constraint on our effort was that we wanted to place as little burden as possible on the development staff, who were deep in the throes of producing the next release. Thus we choose not to try to collect all the build specifications (*i.e.*, `makefiles`) for the base and released product and to determine the content of each product from these specifications. The build process for this product was exceedingly complex. Taking this approach would have added a week or two of additional effort and would have required guidance from the buildmeisters.

To resolve item (d), within the selected subtrees the relevant files were selected by using obvious file extensions (like `.c`, `.C`, `.cpp`, *etc.*), manually examining other likely looking files relevance, and excluding certain obvious files (those whose names contained strings like "TEST" or "JUNK"). All of the relevant ClearCase directories were explicitly designated in an indented input structure for a tool that was developed to "pull" copies of product files from ClearCase to the counting directories. The files to be pulled from each directory were designated by file extensions. Altogether we used approximately thirty different file extensions.

Resolving item (e), the size unit, was non-trivial. The most common unit reported in the software engineering literature is non-commentary source lines. But we had thirty different types of files with a number of different ways of syntactically identifying comments, and in at least a few cases a fully correct program for recognizing and removing comments would be non-trivial. Furthermore, a spot check of some of the files revealed that block comments were not used extensively so we settled on using non-blank lines as our size unit. The ideal unit would be source statements. This is discussed below in section 7.

Resolving item (f), recognizing new and deleted units, also gave us pause. Developers often reformat source code to ensure conformance with a standard or to enhance maintainability (*i.e.*, to reduce the probability of error in making subsequent modifications). Extensive reformatting had been done on many of the source files in

this product. To address this we chose the simple expedient of removing all spaces from all source lines. This would not eliminate counting as a change a reformatting that, say, moved a brace from one line of C code to another, but it did address a number of other re-formatting changes. Thus, for us a "line of code" was any non-blank line of source with all spaces removed.

The standard Unix `diff` program was selected to determine the difference between a base file and release file. `diff` produces output that describes how an old version can be changed into a new version by deleting, adding, and changing blocks of lines. However, the exact methods used by `diff` to distinguish between these three cases was not know to us, so we choose to consider any of the blocks reported by `diff` as changed as a deletion followed by an add. This choice was captured in scripts we wrote that provided a count of lines deleted and lines added for each file that had both a base and a release version. All the lines in base files that did not appear in the release view were counted as deleted lines and all the lines in release files that did not appear in the base view were counted as added lines. We built a number of Perl and C-shell scripts to perform many of the detailed tasks just described, but since this was an initial, exploratory effort, we executed all of our scripts manually and verified our results as we went though our million-line-plus code base subsystem by subsystem.

Resolving item (g), tracking files from the base to the release view, was complicated by the fact that sometimes the name (especially the full pathname) of the file would change. To counter this we added the capability to our tools to fetch and use the ClearCase file database ID to distinguish between new files and those whose names had been changed.

The net result of the above was that for each subsystem we obtained the number of old lines deleted and the number of "new" lines added. But to resolve item (h), we need to decide whether or not these two values were equivalent from the perspective of fault density. Since the faults in the release would occur much more often in the added lines than in the deleted lines we answered this question in the negative. We chose, then, to again consult our lead development manager who decided to weight the delete count by 0.25. Thus a file with 100 deleted lines and 100 added lines would receive a NCLC count of 125.

Since we were dealing with a multi-language product with a large number of displays a significant percentage of the code base was resource files that were constructed by special development tools from tables that were less complex, and thus much less prone to human error than code. Weights for various groups of these files were again selected by our lead development manager. Thus, if the example just given was a resource source file that was in a group that had a weight of 0.2, then the final contribution of this file to the release size would be 25 NCLC.

As we mentioned, the effort just described was first applied to a previously released, but still current version. Much of our effort involved developing the Perl and C-shell scripts and the detailed procedural instructions for their use. A couple of person-months was therefore expended to obtain our first release code count. The procedures and tools we developed were then applied to the product being tested by another software engineer. It took her about two weeks to make the count for the this product.

By writing a master program to invoke the various scripts, and with specific points for user verification and intervention, we estimate that we would be able to make a count for future versions in two or three days. Similarly, the tools and procedures for this product could be move to other products in this organization with similar characteristics in a week or two. However, a much more cost effective approach would be to embed the required data collection and calculation machinery directly into the source management and problem tracking systems and then to develop procedures that would require very little effort to obtain the current fault density estimate for any release. Some of the tool capabilities that would make this possible are described in the next section.

7. Proposed Capabilities for New Tools for Determining Fault Density

Although the capabilities of individual software development management tools has improved in recent years, not much improvement has been made in the integration of these tools. To easily obtain and effectively use fault density data the tools for problem tracking, source code management, and project management need to be integrated. One way to do this is would be via Project Work Item (PWI) entities.

A PWI would contain the following information:

- Product name
- Responsible developer
- Type (New Capability, Fixes, Assigned Problems, Unassigned Problems)
- Tracking dates
- Subsystems modified
- Files (by subsystem) modified and new, deleted, and changed statement counts
- Problems fixed (for PWIs of type Fixes)

A product (version of a system or application) would be exactly a union of New Capability or Fixes PWIs, which would drive the automated product "make" procedures. All changes to the source repositories would be made under the aegis of a PWI and the file change statistics would be updated whenever a change was made.

The major effort involved in determining fault density, *i.e.*, determining the code change counts, would thus be performed automatically and would be available continuously. (New techniques for making such counts are discussed below.)

The other two pieces of information that are needed for determining fault density for shipped products, namely, total monthly usage and total monthly problems over ten or more months could be obtained from the problem tracking system and elsewhere. We envision that PWIs would be maintained on the source code management system, and that this system would have a user interface for entering usage and problem counts and obtaining current product fault density values.

The use of PWIs would allow fault density to be estimated for individual subsystems and individual developers. Extensions to the PWI concept, in conjunction with other data collections tools, could support other useful product management functions such as test coverage and test progress tracking.

New tools should be developed to replace the NCLC count defined in section 5. Such tools should have the following attributes:

a) they should count statements rather than lines,

b) counts should be given separately for new, deleted, and changed statements,

c) they should work at the file level, not the program level, *i.e.*, counts should be made just by examining the old and new version of a file,

d) the way in which each file was processed would be determined by the file extension (or some other syntactical attribute of the file name, or a file attribute maintained in the source management system),

e) changes in the way a statement was formatted would not be counted as a change, *i.e.*, statements would be put in a canonical form before they were compared, and

f) the criteria for deciding whether a statement was new or changed could be externally controlled.

Clearly, different counting programs would be necessary for different programming languages, or at least different syntax tables would be required for different languages. Counting changed lines of specified types in a text file would be special case.

The current situation is that each software development organization that wants to use fault density measures for quality management must develop its own ad hoc solutions to the problems we have just described. Many such local solutions have been developed. We hope that some of the source code management system vendors will see this problem as a business opportunity, and offer products that would make fault density measurement routine. As we discussed in section 5, fault density is an essential measure for managing software quality and improving customer satisfaction.

Lucent Technologies
Bell Labs Innovations

A Tester's Top 10 List

R. C. Silacci

5/26/99 RS

Number 10 - Metrics

Lucent Technologies
Bell Labs Innovations

"You can't control what you can't measure"

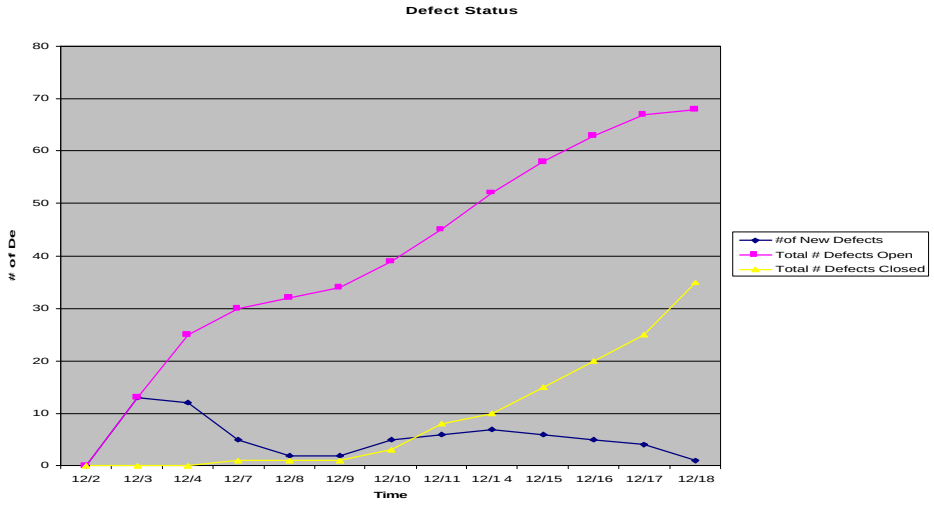
- **What to measure?**
  - **Defects**
  - **Testing Progress**

?

5/26/99 RS

Tracking Defects

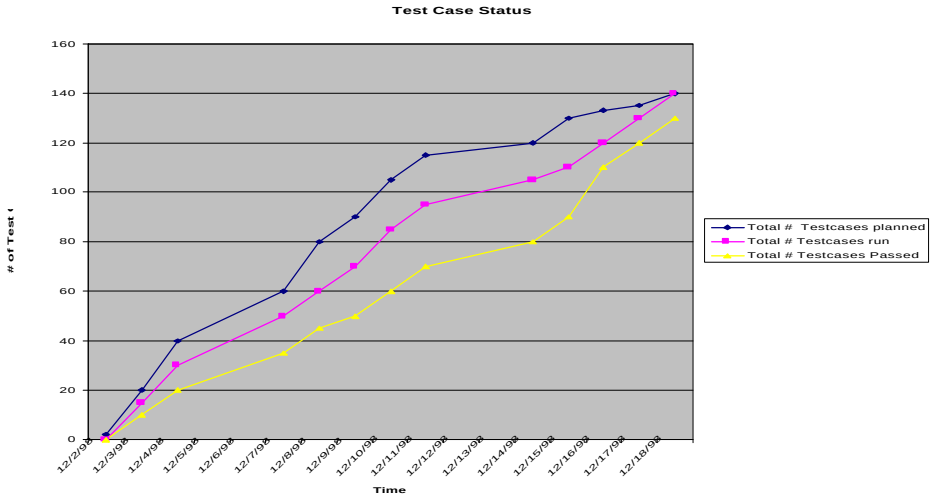
Lucent Technologies
Bell Labs Innovations



5/26/99 RS

Testing Progress

Lucent Technologies
Bell Labs Innovations



5/26/99 RS

Number 9 – Involve Testers Early

Lucent Technologies
Bell Labs Innovations

"The early tester gets the bug"

- **Do Bring Testers in Early to Develop Test Plans Based on Architecture and**
- **Catch Bugs During the Reviews/inspections**
- **Do Not Wait Until Testing Starts to Start the**

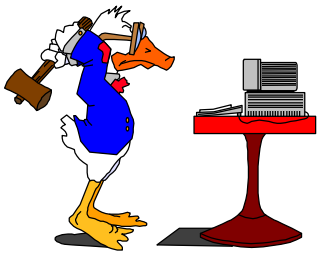
5/26/99 RS

Number 8 – Development Testing

Lucent Technologies
Bell Labs Innovations

"Developers test the code;
Testers code the test"

- **Unit Test**
- **Integration Test**



5/26/99 RS

Number 7 - System Builds

Lucent Technologies
Bell Labs Innovations

"To build or not to build"

- **Build Numbers**

{dev/test} {release #}-{incrementing#}

- **Build report**
  - **List of Defect Fixes in the Build**

5/26/99 RS

Build Report

Lucent Technologies
Bell Labs Innovations

MYSYS BUILD REPORT

Build Number: dev2.0 - 143
Total: 4 MRs included in this list.
This data is updated - Wed Oct 7 01:14:09 EDT 1998

Defect	Severity	Submit Date	Create Date	System	Developer	Abstract
981580	3	10/06/98	06/07/98	mysys	Dev1	Soft keyboard should be available for all applications
983052	3	10/06/98	09/25/98	mysys	Dev2	peculiar characters show up on screen
983153	3	10/06/98	10/06/98	mysys	Dev2	product evolution support thru boot files
983159	3	10/06/98	10/06/98	mysys	Dev3	update build scripts

5/26/99 RS

Number 6 – Deliveries and Packaging

Lucent Technologies
Bell Labs Innovations

"Bugs are delivered in all sizes and packages"

- How do you want the software delivered to
- How do you want it packaged?
- What is the easiest way to install it?
- Can you back it out , if it is garbage?

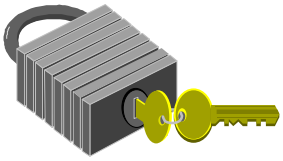
5/26/99 RS

Number 5 – Test Environments

Lucent Technologies
Bell Labs Innovations

"Save the environment"

- An Isolated Separate Environment
- Resembles, As Closely As Possible,
- Is Under Change Control.
- Controlled by the Testing Organization



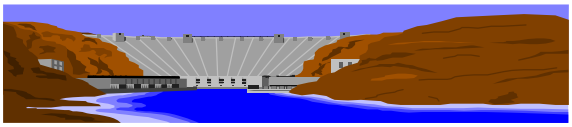
5/26/99 RS

Number 4 – Quality Gates; Entrance/Exit

Lucent Technologies
Bell Labs Innovations

"Watch out when you open the gates!"

- **Agree Upon Before Testing Starts**
- **Tells Testers When They Can Officially Start and When They Have Completed**
- **Puts the Pressure on the Developers to Meet the Criteria or Test Rejects the**



5/26/99 RS

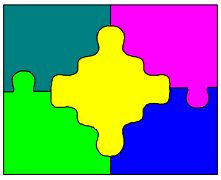
Number 3 – Test Planning

Lucent Technologies
Bell Labs Innovations

"If you fail to plan you are planning to fail"

- **Planing Is Crucial to the Success of**

- **Use a Template**
- **Review the Plan**
- **Don't Forget to Plan the Environment**



5/26/99 RS

Number 2 – Status Meetings

Lucent Technologies
Bell Labs Innovations

"Oh NO, More Meetings!"

- **Attendees:**
  - **Development – 1 Rep From Each Development**
  - **Testers**
  - **Project Management**

- **Agenda:**
  - **Defect Tracking Status and Fix Delivery Status**
  - **Test Case Progress**
  - **Issues and Action Items**

5/26/99 RS

Number 1 – Configuration Management

Lucent Technologies
Bell Labs Innovations

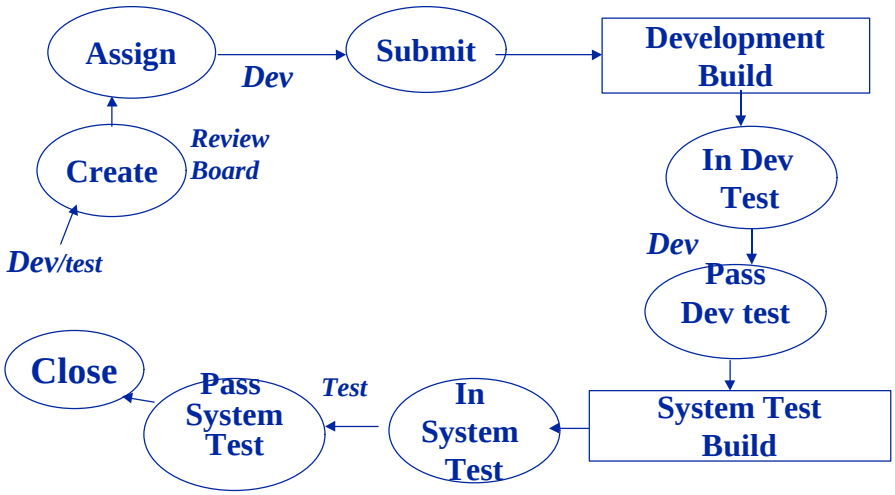
"Go Configure "

- **CM = The Overall Process for Management of the Source Code, Defects and Software**
- **Change Control**
- **One of the Early Decisions a Project Must**
- **Tie Defects and Source Code Changes**
- **Agree on a Defect Life Cycle**

5/26/99 RS

Defect Life Cycle

Lucent Technologies
Bell Labs Innovations



5/26/99 RS

Conclusion

Lucent Technologies
Bell Labs Innovations

- Use This Top 10 List to Help Your Project
- What About Process?
- From These 10 Key Factors of Success You
  - *Testing Process*
  - *Configuration Management Process*

5/26/99 RS

A TESTERS' TOP 10 LIST

R. C. Silacci
Lucent Technologies Inc.

Abstract

System testing is a vital component of the service/product development cycle at any company. The current competitive environment requires our project teams to deliver higher quality products faster while meeting or exceeding customer expectations. In an effort to meet this challenge, more and more pressure is being put upon the testing effort. From practical experiences in testing many Telecommunication systems a testers top 10 list has been compiled. This list contains the top 10 factors that will make testing a product successful and will make a testers work life easier. The paper discusses these top 10 items as the key factors of success. The following is the Top 10 List:

Number 10 - Metrics

"You can't control what you can't measure"

The two basic metrics a tester has to track are the defects and the testing progress.
Defects reports should be generated daily and contain the following data:

- Defect id number
- Defect severity
- Defect Status (open, being worked, fixed, tested)
- Who created it
- Who is assigned to fix it
- An abstract of the defect
- Date it is due to be fixed and delivered to system testing

Defects should also be plotted on a graph so that the defect backlog can be managed. See figure 10-1.

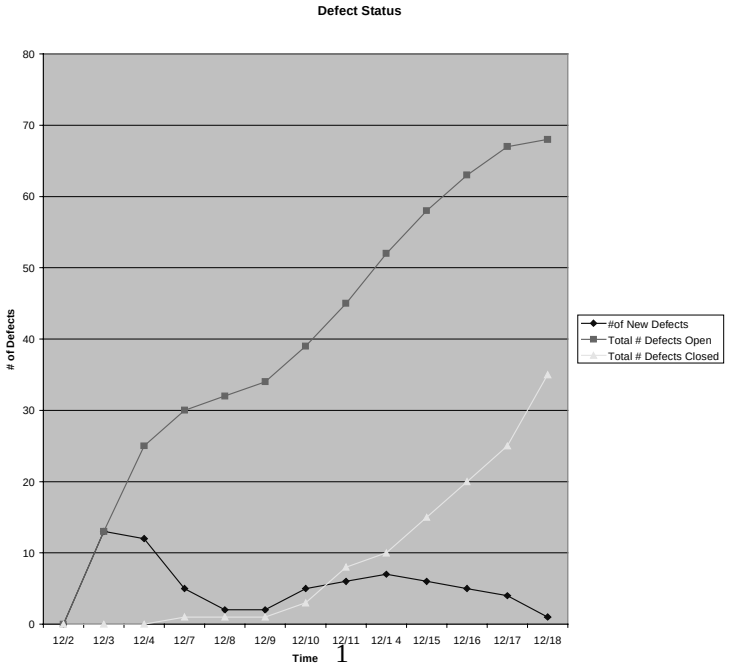


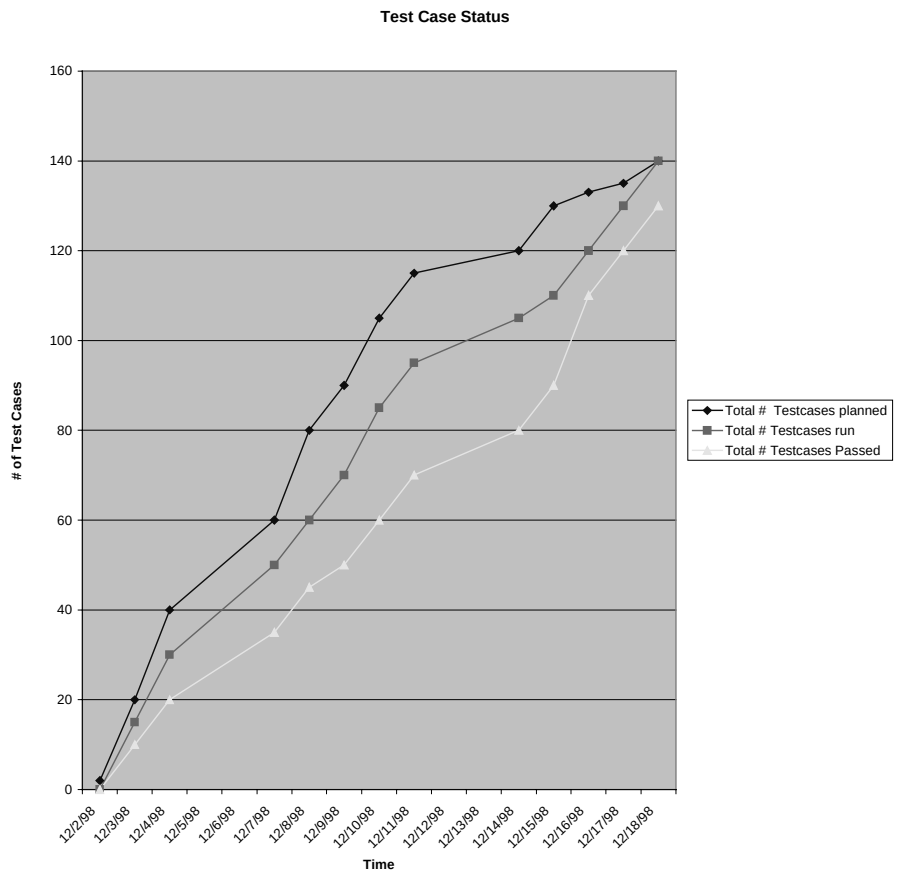
Fig 10-1

Test Progress reports should be generated daily and contain the following information:

- Number of test cases planned for a particular functional area
- Number of test cases executed
- Number of test cases passed
- Number of test cases failed
- Number of test cases blocked
- Number of test cases deferred

Test progress should be plotted on a graph to manage the execution of test cases. See Figure 10-2

Figure 10-2



Number 9 – Involve Testers early in Reviews/Inspections

"The early tester gets the bug"

Testers should be involved as early as possible in the project so that they can understand the architecture, requirements and design of the project. Reviews are critical to any projects success. The testers must be

involved in requirements and design reviews in order to build their test plans and develop their test cases. Do not wait to bring testers in just in time to perform testing.

Number 8 – Development Testing

"Developers test the code; Testers code the test"

Make development responsible for Unit and Integration testing. Integration testing, in this sense, means integrating the units together and making sure the interfaces between the units work correctly.

Number 7 - System Builds

"To build or not to build"

Builds whether development or test should be numbered and uniquely identified. A scheme one can use is {dev/test}.{release #}.{incrementing number}. A build report can be produced from the list of the defect fixes going into the build. This helps developers and testers know what is in the build. Development builds can occur daily, if desired, while test builds can occur at a frequency determined by the testing organization. Once the defects have been verified in the development build, a test build can occur. The development build should be tested by the development organization via a unit and Integration development test.

Number 6 – Deliveries and Packaging

"Bugs are delivered in all sizes and packages"

Software deliveries refer to how software is being delivered to testing. Is it delivered as binaries, source code or a list of defect fixes to build? This depends on the following circumstances:

- One or many development locations. In this case, the testing organization may want to receive binaries from each location.
- Large or small project. In a large project, each module could be compiled separately and binaries delivered to system test. In a small project, just source code or a list of defect fixes may be delivered.
- Third party deliveries – If an outside party is delivering software, make sure it is binary so that they are responsible for the compiling and integrity of the software.

Packaging refers to how deliveries into and out of system test are bound together. Examples are install shields, compressed (zip) files, binary, self-extracting files, etc. Testers must know how to extract and install these files and bring up the system. The same goes for customers.

Number 5 – Test Environments

"Save the environment"

This is obvious, but it takes preparation and thought to configure a suitable test environment. A few basic requirements for a test environment are as follows:

- An isolated, separate environment
- Resembles, as closely as possible, production.
- Is under change control.
- Controlled by the testing organization

Number 4 – Quality Gates, a.k.a., Entrance/Exit Criteria

"Watch out when you open the gates!"

Entrance /exit criteria is critical in controlling the quality of the deliveries testing receives. This should be developed and agreed upon before testing starts. This tells testers when they can officially start and when they have completed. The following are examples of entrance criteria:

1. Supporting documentation in the form of Functional Requirements, Methods and Procedures, User Manuals, for the delivered software must be baselined and made public. The SIT Plan documents must be reviewed and base lined. This includes the completion of a document review with team members from all asset teams and BA systems.
2. Release notes must accompany all deliveries into System Test. Release notes contain a list of know problems, what has been fixed and any start up or shut down procedures.
3. The source code for all deliverable executables must be under Source Code Control. *
4. All development Unit Tests and Integration tests must be executed and completed before delivery to System test. *
5. All known unresolved defects (severity 3s) resulting from the above tests (5) must be documented and made available to the Test Team. *
6. No severity one or two defects exist out of previous tests (5). *
7. Any Environmental changes such as setting or changing of Environment variables must be noted and made public via release notes to the SIT Team.
8. The delivered code must be properly packaged according to packaging specifications. *
9. The Test Team must have access to a process or system for recording and tracking defects.
10. System installation must be complete with a proven startup and shutdown capability. *
11. The Test Environment must remain stable after installation of software.
12. The Testing Environment is separate from any development, demo or production environment, so as not to introduce contentions for resources. This includes servers, clients, networks and databases.
13. Any tools that are required for executing tests must be up to date in terms of version number and must be accessible by the Test team from within the Test Environment.
14. Connectivity to external systems must be available and have been tested.

    That is, Lab-to-lab connections have been verified. *
15. Network for client/server connections has been verified. *
16. The Test Support team (i.e., Application Developers/Testers, etc.) has been identified and is available to provide support.
17. ALL approved exceptions to the above criteria are fully documented and communicated as such by management to testers.

* - Failure to meet anyone of these criteria will result in rejection of the software back to development and a severity 1 MR generated.

The following are three categories in which the systems can be certified when time is a factor. It allows management to understand completely the condition of the systems when making deliverable decisions. When **time is not a** factor, the ultimate goal is to target Green for the certification status.

A. Green

1. All test cases for the software shall be executed.
2. The software shall pass at least 90% of test cases.
3. The software shall have no defects with severity of 1 or 2.
4. The software shall have no more than one severity 3 defect for every five- (5) test cases.
5. All known problems are documented in the release letter sent to the Customer.

B. Yellow

1. All high priority test cases for the software shall be executed.
2. The software shall pass at least 90% of the high priority test cases.
3. The software shall have no severity 1 defects.
4. The software shall have no more than one severity 2 defect for every fifty (50) test cases
5. No limit to number of severity 3 defects. Work arounds will be provided where possible.
6. All known problems are documented in the release letter sent to the Customer.

C. Red

1. The conditions for Yellow in **B** are not met.
2. All known problems are documented in the release letter sent to the Customer.

Number 3 – Test Planning

"If you fail to plan you are planning to fail"
A Test plan is crucial to the success of testing. In developing a test plan, a template is invaluable. The IEEE standard on test case documentation is a good one to follow and perhaps add to. It is important that development and requirements personnel are involved in the test plan review. In the test plan a high level "1 liner" for each test case should be included to give and idea of the type of test cases being developed.

Number 2 – Status Meetings

"Oh no, more meetings!"

Test Status meetings are crucial to controlling the testing effort and understanding the progress or lack thereof.
Attendees:

- Development – 1 rep from each development area.
- Testers
- Project Management

Agenda:

- Defect tracking status and fix delivery status.
- Test case progress
- Issues and action items.

The best way to run the meeting is to prepare a matrix of the defects, which would look like figure 2.

Defect ID	Status	Severity	Orig	Assigned to	Date Due	Abstract
Y2k980158	Created	2	Don	Diego	1/1/2000	Contents of Y2K Readme file is empty
Y2k980159	Assigned	1	Don	Juan	1/1/1999	Daylight savings time doesn't work in 2000

Figure 2

This should be updated daily and be available at least 1 hour before the meeting so that everyone can review it. The web is a convenient mechanism for providing this. The facilitator should go through the newly created defects and assign a developer and a due date. For the assigned defects, check any overdue fix deliveries.

Test case progress can be tracked by updating daily the graph in Figure 10.2. A matrix of test case number, description of the test cases and its status should also be produced daily. To take it one step further, if a test case fails the defect that caused the failure should be shown on a test case matrix.

An issues list or action item list should be available to review. This meeting should occur daily during the critical part of the testing cycle.

Number 1 – Configuration Management

"Go Configure "

Configuration Management is the overall process and management of the source code, defects and software builds. It starts with a source code control system and for best results, one that is tied to defect tracking. That is, the source code can't be changed unless there is an accompanying defect that institutes that change. The status of these defects and their appropriate fixes are tracked throughout the development and testing cycles. One of the first things a project must do is to decide what statuses they want to use to track defects and their fixes. Typical sets of statuses follow:

- created
- assigned
- submitted for testing
- in development testing
- out of development testing
- in system testing
- out of system testing
- Delivered to customer.

The build process can be built around these statuses. For example, a development build would take the defect from submitted to development tested. Once development passes it, the status would change to passed development testing.
The system test build would bring it to system testing. When it passes system test, it would go to "passed system test". See figure 1 below.

Defect Life Cycle

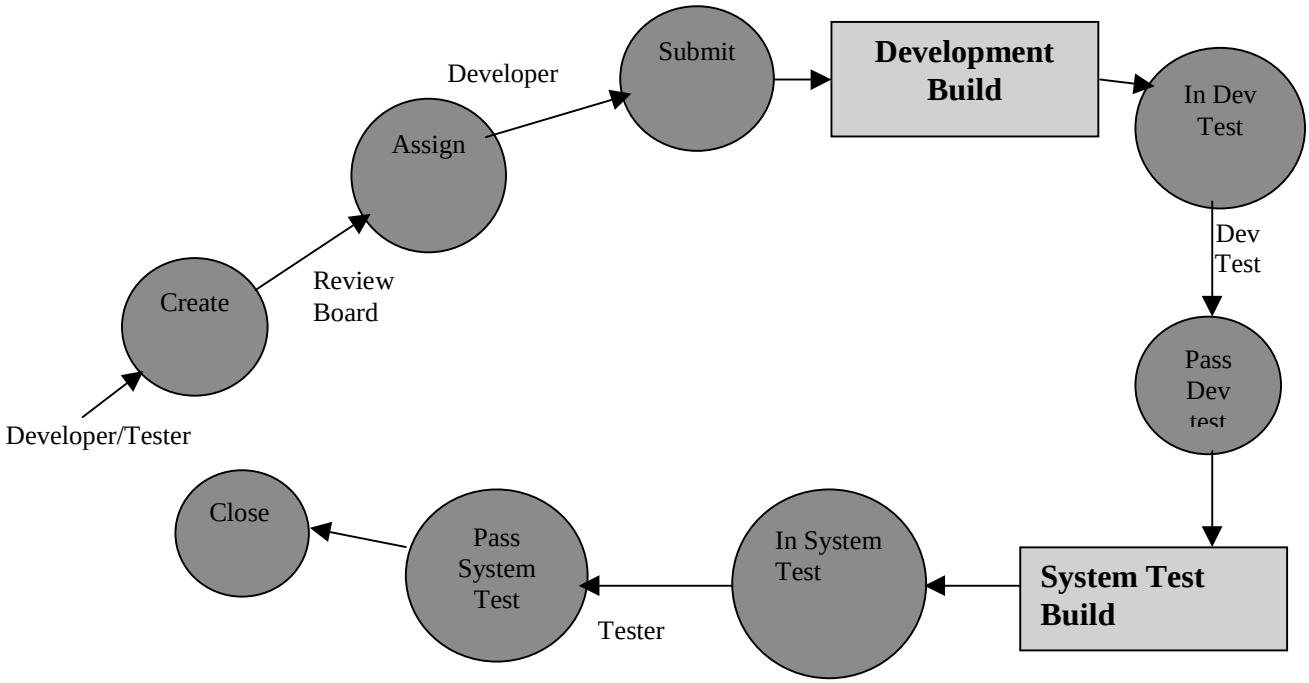


Figure 1.

Conclusion

There are obviously more than 10 key factors of success, but the factors presented here based on the experience of the author. The bottom line is how these key factors are all tied together. The answer is process! Two processes to be exact. One is a Testing Process that includes all phases of testing including walk-throughs/inspections, development testing, system testing and performance testing. The second one is a Configuration Management Process that includes change control, defect status and tracking, build processes, etc. These 10 key factors of success can be packaged in these two processes to produce quality improvements by prevention as well as detection of defects. The 10 factors would map as follows:

Process	**Key Success Factors**
Testing	2, 3, 4, 5, 8, 9, 10
Configuration Management	1, 6, 7, 10

The payoff is to develop processes that contain these factors of success.

How to Build a 20-Year Successful Independent Verification and Validation (IV&V) Program for the Next Millennium

Jon Hagar and Lisa Boden
Lockheed Martin Astronautics Company
Mail Stop H0512
P.O. Box 179
Denver, CO 80201
303-977-1625
fax 303-977-1472
jon.d.hagar@lmco.com and lisa.m.boden@lmco.com

Lockheed Martin

Topics

- What is IV&V?
- When is IV&V beneficial?
- What are the key elements of a successful IV&V?
- What are the experiences from a history of IV&V on a variety of projects?

Lockheed Martin

IV&V Tested Products

Mission Requirements

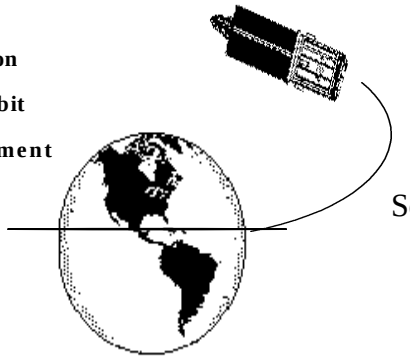
Apogee

Perigee

Inclination

Park Orbit

Deployment



Vehicle Characteristics

Rocket Motor

Reaction Control

Payload

Software
- Code
- Documentation
- Load/Uplink Products

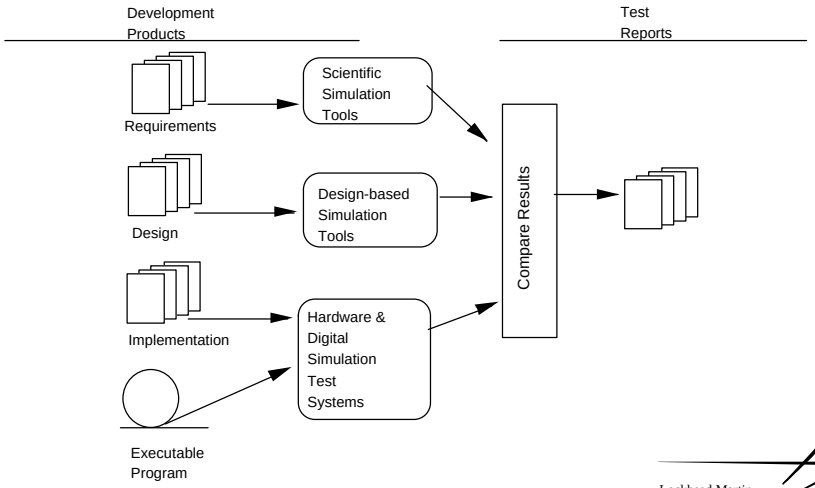
Lockheed Martin

Three Tenants of IV&V

- Verification
  - Check lifecycle products for consistency
  - Unit level testing for coverage
  - Requirements to Design to Code to Load Products
- Scientific Validation
  - Analysis and modeling of algorithms
- Hardware Based Validation Testing
  - Test software in realistic environment

Lockheed Martin

Multiple Levels of IV&V



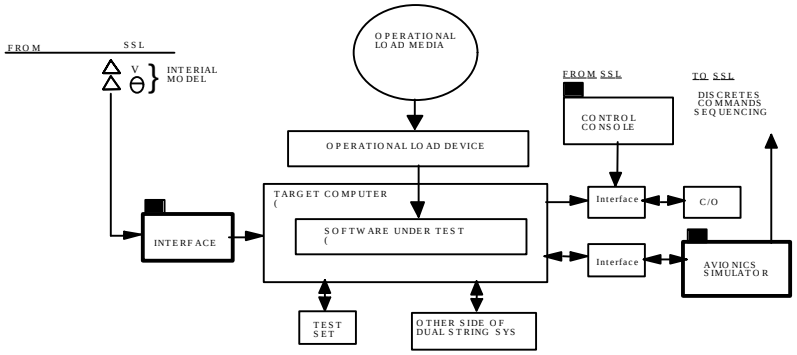
Lockheed Martin

Standard Sample Tools

Activity	Tool	Function	Benefit
Verification	AdaTest by IPL	Coverage	Measurement of test
Scientific Validation	Booster Utility Program	3-degrees of freedom simulation in Matlab	Assessment of data values testing
Validation	Test Environment	Execution of software supported by Web tools	Assessment of software realistically

Lockheed Martin

One Possible Lab Configuration



SSL = SGI Computer Running Real-Time Closed Simulation Models
C/O = Check Out Computer Station

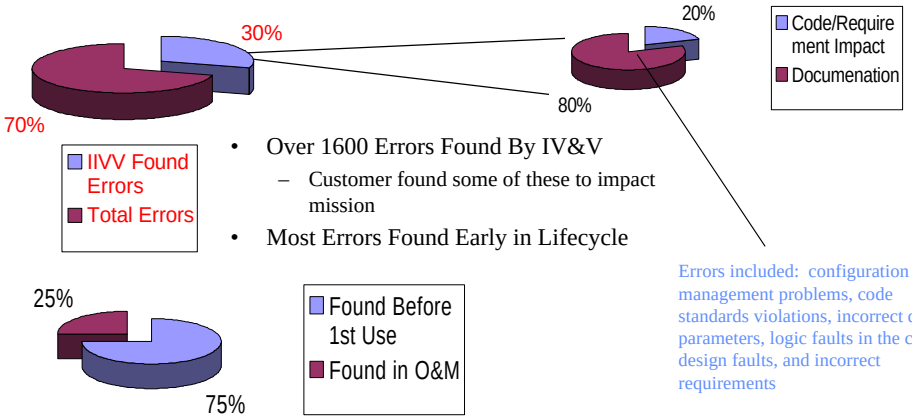
Lockheed Martin

Experiences

- Full testing with hardware in loop is better than review
- Test to find errors
  - Negative Testing
  - Break the software - try a lot of different “hammers”
- Full life cycle testing - *from unit up to full system level Testing*
- Must have:
  - People - training and diverse skills
  - Process - defined, repeatable, documented, measured, and managed
  - Tools and Techniques - manual and then automated if costs are returned
  - Management - schedules, planning, budgets, supervision, and communication
  - Continuous Improvement - of all of the above with small measurable steps

Lockheed Martin

Data Results



- Over 1600 Errors Found By IV&V
  - Customer found some of these to impact mission
- Most Errors Found Early in Lifecycle

Errors included: configuration management problems, code standards violations, incorrect data parameters, logic faults in the code, design faults, and incorrect requirements

- Problems found by inspection, test, and analysis using both manual and automated approaches
  - equally distributed in Verification, Validation, and Scientific Validation

Lockheed Martin

Lessons Learned

- Define scope of work and exchange of data/products
  - Configuration manage all products
- Good customer interface and relationships
- Work and rework schedules - *planning of new testing is recurring*
- Work communications - *developer, testers, management, customer, and engineers*
- Databases - *error report, product configuration management, traceability, and tests*

Lockheed Martin

Lessons Learned

- Interface problems are the hardest to find and most common that remain
- Automation of tests - *good if you get your development costs back*
- Staff - *have mix of experienced, new, and diversely skilled*
- Criticality of software dictates the strictness of testing
  - Requirements, standards, coverage, analysis, modeling, inspection, etc.
- Modeling and analysis of test inputs, environment and outputs is important

Lockheed Martin

When to Consider IV&V

- One of many approaches that can find errors
- Benefit when involves high risk, “ultra-reliable”, safety related, large cost factor, regulation
  - Aircraft, Medical, Nuclear, Space, etc.
- Can be used when there is no customer acceptance test
- Added cost can be justified by removed errors for critical applications

Lockheed Martin

How to Build a 20-Year Successful Independent Verification and Validation (IV&V) Program for the Next Millennium

Jon Hagar and Lisa Boden
Lockheed Martin Astronautics Company
Mail Stop H0512
P.O. Box 179
Denver, CO 80201
303-977-1625
fax 303-977-1472
lisa.boden@lmco.com and jon.d.hagar@lmco.com

Abstract: : IV&V is common on many critical software government programs. IV&V can save missions and improve the product quality when done right. Done right means that there must be a balance of people with the right skills, management, as well as processes aimed at verification of each life cycle step, validation with simulation/analysis tools, and a hardware based test facility. Automation of testing and IV&V helps to reduce cost, but automation is only part of a complete program. While IV&V is not for every software program (because of redundancies), high risk projects where there are critical cost or safety factors can benefit from some level of IV&V.

Keywords: IV&V, Verification, Validation, Hardware-based Test Environment

Introduction

Lockheed Martin started supporting Independent Verification and Validation (IV&V) programs in 1979. Projects supported have included both booster and spacecraft vehicles. A 1991 report from an Air Force review concluded that one Denver area IV&V program had found errors that would have impacted mission performance of the subject software. Errors found by IV&V tend to be more numerous during preliminary development efforts, though review of data shows that some errors have continued to be found by IV&V well into operations and maintenance efforts.

This paper examines some of the lessons learned from almost 20 years of testing and how these will carry the programs forward in the next millennium, with the following considerations:

1. What is IV&V?
2. When is IV&V beneficial?
3. What are the key elements of a successful IV&V?
4. What are the experiences from a history of IV&V on a variety of projects?

What is IV&V and What Are We Testing?

In Denver, Lockheed Martin's IV&V approach involves different levels of testing and analysis. IV&V testing concepts originated in Air Force programs wishing to achieve reliable software. In our IV&V, Verification is defined as the iterative process of determining whether the product of each step of the life cycle fulfills all of the requirements levied by the previous step. Validation is defined as the evaluation, integration, and test activities carried out at the system level, to ensure that the final developed system "works" [Wallace 89]. The need for IV&V sometimes is questioned. NASA studies have concluded that its effectiveness can be low [McGarry 82]. However, our experience leads us to conclude that IV&V can be successful and effective, where effectiveness is defined as the ability to find errors, if the testing of software is conducted from a user and/or system perspective to assess attributes such as function, performance, usability, quality, and reliability. This systems view of the software test process allows the consideration of all classes of errors that can be practically detected in software. These errors or faults, as defined by [Howden 91], include programming, deductive (e.g., translation from design to code), and abstraction errors. To accomplish this, software evaluation must be done both at a Verification as well as a Validation, level by engineers skilled in more than just the software disciplines.

The software-systems we are using IV&V on have the following characteristics: real-time; booster flight control; minimal human intervention possible; embedded within the system it controls, and numerically intensive calculations of such critical items as, trajectories, flight dynamics, vehicle body characteristics, and orbital targets. The development programs usually are small consisting of less than 50,000 source lines of code in two separate computer systems, yet these programs are critical to the control and success of the entire flight system. An example mission profile is depicted in Figure 1.

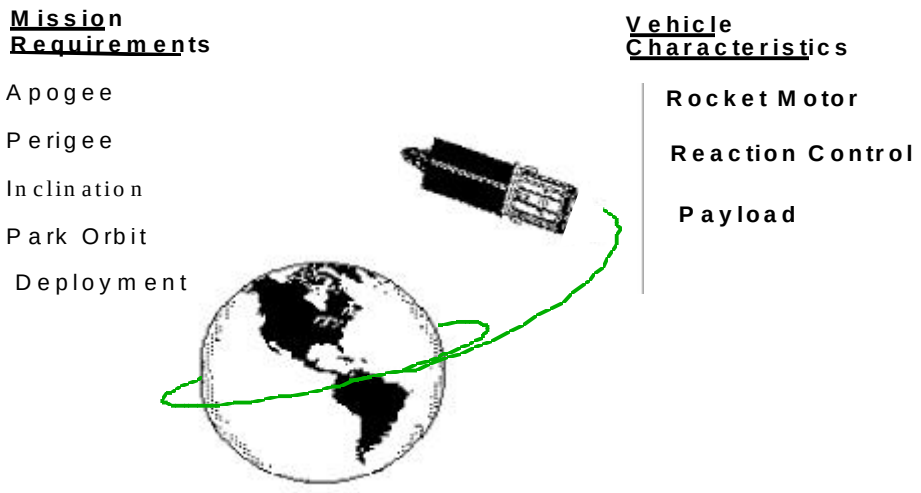


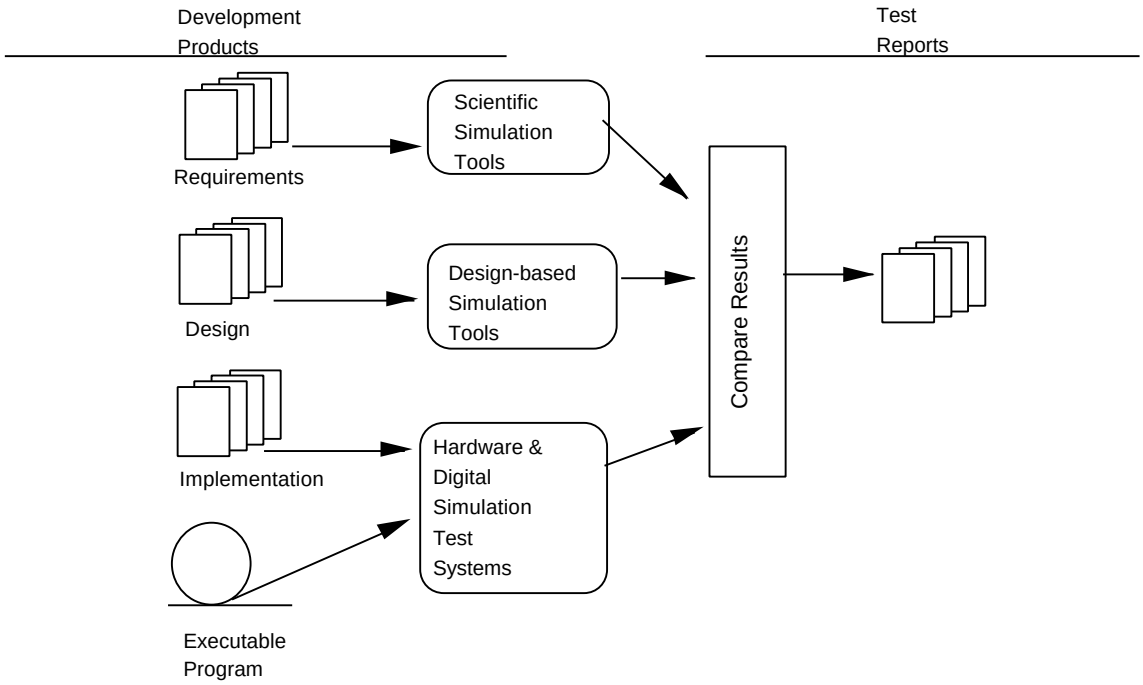
Figure 1 - Complex Software Requirements

Our test approach spans from the software unit level to integrated hardware and software (a system). Additionally, we have issues of configuration control, change management, documentation, and business management. In all of the test activities, we use a variety of supporting software tools and metrics. The goal of our IV&V is to show that the flight software is ready for use and the chance of catastrophic mission loss due to software is minimized. The software under test is the guidance, navigation and control software of the system, and so test programs that involve the fully integrated system are not possible. For example, we cannot fly the software in a "beta" test on an actual system to see if it will work in real use.

Three Main Process Elements of our IV&V Process and Tools

Our product area's testing tools address various levels of abstraction (Figure 2) of the software-system. In our approach, the lowest testing level is structural verification testing conducted with a digital simulation and/or hardware system. At this level, verification testing is done to ensure that executable programs implement such things as requirements, design information, and software standards. This testing is done at a module level with small segments of the code being executed in isolation from the rest of the system. A tool executes code in a simulator to support analysis of individual equations and simple logic structure. The comparison and review of results at this low verification level uses models constructed to replicate the code unit and an automated comparison tool (Adatest by IPL) to reduce human interaction. Results information is captured in a Web based test procedure tool/system [Hagar et al] as well as a requirement traceability tool, RTM by Marconi. This process is still human intensive and additional modeling techniques such as Tvec by Tvec Technologies, are being explored in some Lockheed Martin areas.

.



2.2-1 Figure -- Tested Products

The next level of testing involves what we call scientific validation, which demonstrates requirements compliance using scientific-simulation tools. These simulations are done in both a holistic fashion and on an individual functional basis. For example, a simulation may model the entire boost profile of a launch rocket with a full 6-degrees of freedom simulation, while another simulation may model the specifics of how a rocket thrust vector control is required to work. This allows system evaluation starting from a microscopic level up to a macroscopic level.

At the system level, software is tested with actual hardware in the loop. An extensive real-time, continuous, digital-simulation modeling and feedback system of computers is used to test the software in a realistic environment. Testing in a realistic environment means testing the software as a "black box", with the same interfaces, inputs, and outputs as an actual flight system. In the test bed, the computer is surrounded with a first level of electrically-equivalent hardware interfaces. Signals are input into the test bed to simulate the performance of the system and hardware interfaces. Since the test system runs in real time, there is no way to speed-up or slow-down the system. We call this level real-time Validation, since it is primarily concerned with testing the integrated end product against "system" requirements.

Numerous tools support the tiers described above, and many of these tools are simulation-models based on requirements, design information, or the computer architecture. The tools are stand-alone and often custom-built software programs that execute on separate platforms from the software under test. These tools take data that could be the input to the system under test, and produce expected outputs. The tool and simulation outputs can then be compared to the results generated by the actual software being tested. Some of the tools simulate individual equations or logic sequences, while other tools simulate aspects of the entire system. Scientific simulation-based tools provide success criteria or analysis capability that allow engineers to judge the success of the software under test without relying entirely on human judgment.

Overall, this approach and tool set has been successful in taking input from the development products, performing IV&V and generating test results. An important part of our IV&V culture is to look for improvements in our testing processes and tools for both time and cost savings. Recent improvements

have included the use of unit level COTS automation, tools like RTM instead of custom built tools, peer based inspections, reusable assets with tools like Matlab, and Web-based technologies (See Table 1).

Table 1 - Sample of Standard Tools

Activity	Tool	Function	Benefit
Verification	AdaTest by IPL	Coverage	Measurement of test
Scientific Validation	Booster Utility Program	3-degrees of freedom simulation in Matlab	Assessment of data values
Validation	Real-time closed loop hardware test bed	Execution of software supported by Web Tools	Assessment of software realistically

Key Element: Hardware-Based Lab Testing with Real-Time Closed-Loop Simulation

The hardware-based test bed (lab) is the primary validation test facility for our IV&V, and it is integrated with the verification and scientific validation efforts. The lab is used to execute full and partial mission runs, up to eight hours long, designed to validate mission requirements. It provides a certified (the lab itself has been tested) test bed environment for validation of flight load products under simulated real-time conditions using avionics hardware and software simulators. The lab's flight hardware and avionics simulator interfaces with an Silicon Graphics (SG) computer, which simulates the mission environment and vehicle subsystems including propulsion, mass properties, and flight controls hardware. This provides real-time, closed-loop inputs to the test bed. Thus, the lab provides the capability to perform simulated full-mission runs from pre-launch or pre-deployment through spacecraft release and booster deactivation.

The lab consists of a two sub labs: Guidance and Control Lab (GCL) and a Scientific Simulation Lab (SSL). The GCL is made up of flight-prototype hardware and custom-built avionics simulators. The SSL consists of support equipment, such as a SG computer and real-time data linkages. The GCL avionics simulator is used to emulate flight hardware not available in the test bed. Also residing in the GCL is the Checkout Station (COS), which has the capability of issuing uplink commands to the computers, as well as recording downlink telemetry and checkout status of the airborne system. The two areas work together to provide a realistic environment for the flight software. The SSL is used to simulate rotational and translational dynamics of the vehicle. The SSL/GCL combination is then used to make real-time 6 Degree-of-Freedom (DOF) trajectory runs, with the added capability to create restart images along the way. The SSL/GCL test bed is functionally shown in the lab Validation Concept.

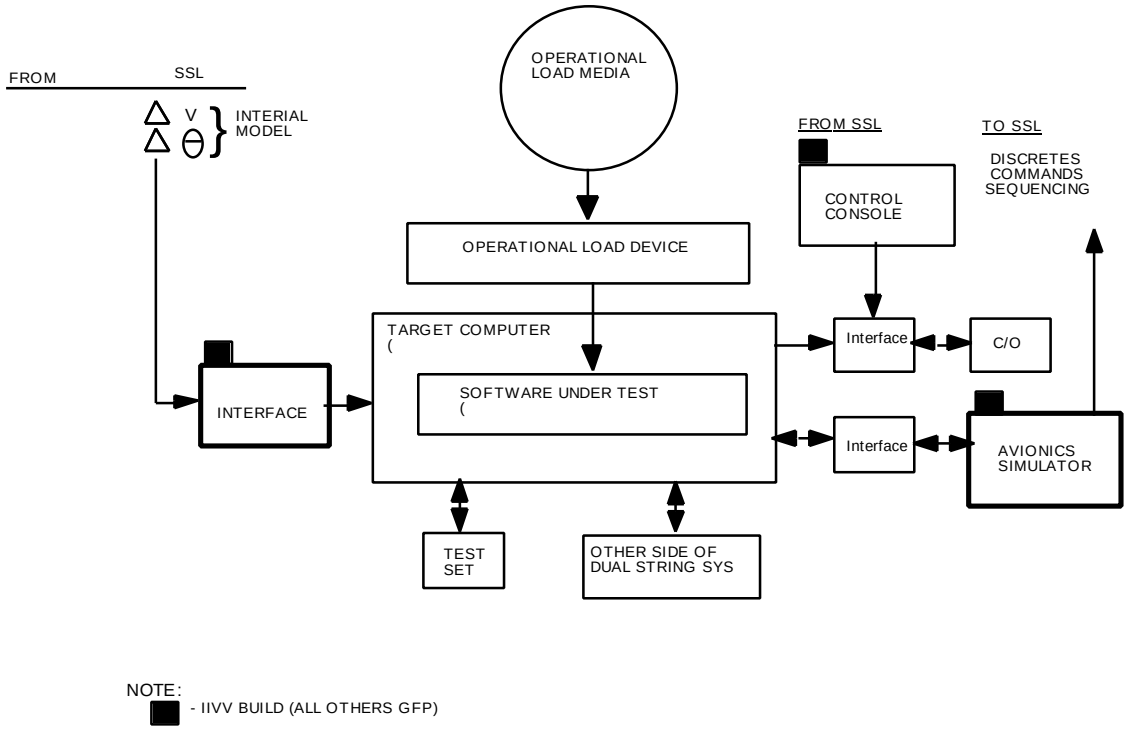


Figure 3 Real-time Hardware-based Validation Concept

When to consider IV&V

IV&V is not right for every program. Comprehensive, fully independent, IV&V is justified when the risk from software failure is high. Risk can be from cost or safety factors. IV&V must also be considered from the view point of the customer and management. If customers and management are willing to do active engineering with software, where other people can provide "outside eyes" of independence, then less rigorous IV&V may be acceptable. IV&V certainly can find errors, but if the cost of the errors being found do not offset the cost of the IV&V, then benefits can be questioned. IV&V can be considered in medical, air safety, nuclear, space, government, or other high risk software programs.

There are now companies and IV&V centers like NASA's IV&V center[ref] that can provide resources and expertise on IV&V. While not for everyone, these centers can provide knowledge bases even to more standard test approaches.

Experience and Data Found Over a 20 Year Life Cycle

IV&V is common on many critical software DOD programs, but there are different levels of IV&V. The range is from review, audit, and/or inspection to full and complete test programs with hardware-in-the-loop test beds. The latter seem better at finding errors and improving products, but represent a cost factor that can be equal to the cost of developer-based testing.

IV&V should be focused on negative testing, in which finding errors is the goal. It should do this by having all levels of testing (unit to full software qualification), multiple tools (simulations, instrumentor, oracles, etc.), and knowledgeable people. IV&V should be viewed as serving in place of the customer performing detailed acceptance testing as defined in ISO 9001.

The keys to exercising a successful IV&V program are tools, people, knowledge about the test domain, defining processes with metrics, and an interest in improving each of these. While other papers that we have written, addressed tools, people, and process, we would like to note here that continous improvement

efforts have contributed both to IV&V success and longevity. Our IV&V area with the support of the customer, has continuously and in small affordable steps upgraded tools, people, and processes in line with Softare Engineering Institute concepts. Trade studies and upgrade plans were prepared at many points along the line of improvement. We have moved from large mainframes to micro computers and workstations and from custom-built tools to commercial products and new languages. Test processes and tool improvement is measured and evaluated against planned expectations. Not every improvement or tool has been successful, but we learn from our failures as well as successes.

Another experience we have found is that test automation works but up front costs and risk issues must be considered. During automation efforts, we first established what our manual processes were. If a manual process was repeated often enough to justify the cost of automation, then we budgeted development effort to test tools. The trade off point between manual and automation was basically to calculate the total cost of manual efforts and then do the same for automated efforts. If automation had less cost and time or there was some major technical advantage, then we would elect to automate. Automation costs include the up front effort to buy, integrate, and then set up tests. It is important to note that not everything we do is automated, but when repetition or some risk factor— like the chance for human error, indicated automation is viable, we develop it using a software development plan, just like any software effort.

Data From IV&V Detected Errors

Over the years, we have found that IV&V finds the most errors and results in the best products when done with full testing and analysis. Part-time or review/audit types of IV&V, which we have done, are not as comprehensive and have missed critical errors. They provide some level of product improvement, and so their costs may be worthwhile, but to obtain the higher levels of error detection, the program must spend the money for full IV&V testing and analysis. Quality is not free, but the lack of quality can cost in the long run too. It is a trade off that must be considered during program planning by management and customers. To do this, we consider the following statistics from the IV&V problem report database:

- Over 1600 error reports have been filed by IV&V in one area, the number represented about 60% of all errors found by IV&V since many error were reported informally (outside of the tracking database via memos or meeting action items) on pre-release engineering products to save costs by analyzing products while they were still under development. This was an important cost reduction factor.

- 30 percent of all software problems were found by IV&V on products we test.

- 20 percent of IV&V defined problems resulted in code or requirement changes, and included errors that could have resulted in degraded mission performance.

- An Air Force review estimated that at least one mission over the last twenty years had been impacted by IV&V, which more than paid for the costs of IV&V.

- Over 80 percent of problems found by IV&V are in documentation. While not impacting code, finding these kinds of errors can reduce maintenance costs and help in process improvement related to concepts such as the Capability Maturity Model and ISO.

- The kinds of IV&V logic errors found include: configuration management problems, code standards violations, incorrect data parameters, logic faults in the code, design faults, and incorrect requirements that impact the design and code. These are equally distributed over the remaining 20 percent of error reports.

- Over 75 percent of the total errors were found during initial development. However, after almost 17 years of operations and maintenance on one program, some 30 error reports were filed during a major system

upgrade. Two of these reports impacted flight products. As the software matured, the shift in problem reports was from code and requirement errors to documentation problems.

- Problems are found by inspection, test, and analysis using both manual and automated approaches. Problems are equally distributed for the verification, validation, and scientific validation areas

Lessons Learned

The following represent some of the main lessons learned by the IV&V program during the past twenty years of testing:

- The IV&V team should be involved at the earliest possible time in the software development cycle. For established programs, the developer should provide extensive technical information and documentation of the system design.

- ·The IV&V scope must be established and agreed to by all involved parties. Scope would include statement of work, reporting structure, interface requirements for formal deliveries and support, and final deliverable software procedures.

- It is best and most independent if a separate budget controlled by the IV&V contractor is established so that changes in scope and deliverable products can be negotiated.

- Establish a customer for the IV&V effort external to the software development group. In order for the IV&V effort to have an influence on the software design, findings and recommendations should be reported to an interested third party with organizational responsibility for the software development group.

- A formal data exchange list should be created between the developer and the IV&V organization, along with a format to control interchange of documentation and products.

- Establish combined schedules for all tasks and maintain active review and notification of schedule impacts.

- Accomplish active planning for all tasks including review of schedules, metrics (trending and projection), and deliveries.

- Establish a forum for exchange of technical and schedule information on a regular basis. Weekly seems to work best.

- There should be an independent development of a requirements traceability matrix to ensure all requirements are testable and to track changes and requirement closures.

- Complete configuration management of all delivered products should be implemented by the IV&V team (know what you are testing).

- Increasingly with the aging of a project, a primary focus of testing should be to validate the interfaces between software and hardware elements. Compatibility problems are hard to find and often remain even after extensive testing and IV&V. Good interface definition helps, but systems of any complexity hide interface problems.

- There is a hidden cost associated with using reuse and heritage software. Typically, the requirements documentation and the interfaces can have problems that complicate their use.

- The cost and time to develop the test software, process, and tools can often be underestimated. This can result in failure of these types of efforts. Also part of the cost, that is often overlooked is the training of people to do the work.

- For critical or safety related code, it's important that no lines of code exist for which there is no corresponding requirements information or objective evaluation criteria (design level). A requirement that only states what to do and is not clear is also incomplete. To develop tests and test requirements, it is important that the software requirement be stated in such a way that a test or IV&V engineer can infer how the code is supposed to function.

- If present, it is important that software quality assurance group be staffed with personnel who are experts in software development engineering, since they need to make decisions about the validity of tests, tools, processes, and IV&V efforts.

- Establish a common problem reporting and tracking database with formal use and access procedures between all users.

Summary

IV&V has contributed to mission success within the Lockheed Martin Denver Astronautics group, but we have found that IV&V has to be done correctly. A good working relationship with the developer and customer has to be maintained. IV&V had to define a plan that used methods, tools, and techniques to test the products and find errors. Once the IV&V plans were defined, these had to be managed and controlled to produce results. This includes obtaining and training the technical staff to do the job right. And, finally, over time the plans and associated elements had to be optimized to ever improve technical and budget performance. The role of independence in testing may be changing but we should not forget some of the lessons learned for IV&V.

References

D. Wallace and R. Fujii, "Software Verification and Validation: An Overview," IEEE Software, May 1989

F. McGarry and G. Page, "Performance Evaluation of an Independent Software Verification and Integration Process,: Tech. Report SEL 81-110, NASA Goddard Space Flight Center, Greenbelt, MD., Sept. 1982.

W. Howden, "Program Testing versus Proofs of Correctness," Journal of Software Testing Verification and Reliability, Vol 1, Issue No. 1, 1991.

M. Deutsch and R. Willis, "Chapter 2: Overview of the Book", Software Quality Engineering A Total Technical and Managerment Approach, Prentice Hall, Inc. 1988, p33.

Hagar, Burba, Wittekind, and Bell, "World Wide Web Based Tools to Support Testing", 1997 Qual Week.

eCommerce Benchmarking Methodology and Criteria

Steve Rabin
Chief Technologist
InterWorld Corp.

InterWorld Confidential - Do Not Duplicate

Commerce Server Benchmarking Issues

- simultaneous connections is misleading
  - depends on size/type of documents being delivered
- what does pages/second mean?
  - determine the total data rate and the number of transactions/second the server can process
- understand units of commerce work
  - loading modules
  - building templates
  - round trips (ie. database)
  - transaction types

InterWorld Confidential - Do Not Duplicate

eCommerce Testing

- **Quality**
  - **Unit Test - thread safety, code coverage**
  - **Smoke Test - quick validation**
  - **BVT - build validation test**
  - **Full Functional Test**
  - **Regression Test - consistent functionality**
  - **Load Test - high availability**
  - **Benchmark - by Platform**
  - **Installation Test - by Platform**
- **Automated Tools**
- **Repeatability**
  - **Consistent and Isolated Environment**
- **Metrics**
  - **Measure real world scenarios**
  - **Load test specific subsystems**

InterWorld Confidential - Do Not Duplicate

Performance & Stability Testing

- Simultaneous Users**: Concurrent usage testing. Threaded commerce software guards against serialized resources. However, that requires careful management of common services to avoid deadlock.**
- Transactions per second**: Stress testing for peak performance. Turnaway capabilities based on estimated peak availability.**
- Server Sizing**: Memory? Connections? CPUs?**
- Database Sizing**: Capacity planning using sample data. Foot-print per user. Trigger threshold notices.**
- Real-world Simulations**: Server log analysis? Classify types of transactions. Example: Search versus Orders?**

InterWorld Confidential - Do Not Duplicate

Top Commerce Site Performance

Site	HomePage Delivery Rate (seconds)	Availability (percent)
Amazon	4.9	99.4
Barnes & Noble	7.3	99.2
Dell	8.3	99.1
Gateway	6.4	86.7
Lands End	9.0	91.5
Toys R Us	5.7	98.1
Business 40 index	9.2	97.5

InterWorld Confidential - Do Not Duplicate

Procedure for Site Sizing and Characterization

1. Collect site characterizations and performance/load requirements.
2. Translate these requirements to server throughput requirements.
3. Perform virtual or actual benchmark

InterWorld Confidential - Do Not Duplicate

Site and Load Characterization

revenue/period:	user input
avg$/order:	user input
visitors placing an order:	user input
pages/visitor:	user input
pages/order:	user input
pages/day:	user input
order ratio (peak hour):	user input
pages/peak hour:	user input
dynamic to static pages:	user input
browsers to buyers	user input/calculated
transactions/second:	calculated
pages/second:	calculated
server/CPU config:	calculated

InterWorld Confidential - Do Not Duplicate

Site/Load Comparisons

	Base Line	Site 1
Hits/day:	**10,000,000**	**14,000,000**
Orders/day:	**600 (4,000)**	**2,500**
Visitors/day:	**70,000 (650,000)**	**75,000**
App Servers:		**24 (NT)**
Conversion Ratio:	**1-1.5%**	**0.97%**
Avg. pages/visit:	**6-8**	**11**
Avg. order size:	**$190.00**	**NA**
Concurrent visitors:	**1200**	**2500**
peak business:	**12PM-5PM (50-60%)**	**5PM-10PM (62%)**

InterWorld Confidential - Do Not Duplicate

Benchmark Configurations

Scenario 1:	Search to Buy Ratio 99 : 1	Static to Dynamic Pages 1 : 99 5 : 95 10 : 90
Scenario 2:	Search to Buy Ratio 95 : 5	Static to Dynamic Pages 1 : 99 5 : 95 10 : 90
Scenario 3:	Search to Browse to Buy Ratio 60 : 39 : 1	Static to Dynamic Pages 1 : 99 5 : 95 10 : 90
Scenario 4:	Search to Browse to Buy Ratio 50 : 45 : 5	Static to Dynamic Pages 1 : 99 5 : 95 10 : 90

InterWorld Confidential - Do Not Duplicate

eCommerce Benchmark Transactions

- Home Page Transaction
- Product Page Transaction
- Section List Transaction
- Shopping Basket Transaction
- Buy Transaction
- User Registration Transaction
- Search Transaction
- Check Status Transaction

Static and Dynamic Units of Work - Simple and Complex Transactions

InterWorld Confidential - Do Not Duplicate

Transaction Type Percent Frequency (bookseller)

- Search on product characteristics (e.g. title 43%)
- Display particular product 32%
- View Homepage 10%
- Display Shopping Cart Contents 8%
- Add Item to Shopping Cart 5%
- Buy a product 1%
- Register new user .3%
- Display order .3%
- Error .3%

InterWorld Confidential - Do Not Duplicate

Benchmark Scripts
(Search-Buy)

Home Page
Search
Basket
Order Process

OP1 =	ObjBuilder, OrderProcess	initialize
OP2 =	" "	verifyOrderDialog
OP3 =	" "	verifyBillingDialog
OP4 =	" "	verifyShippingDialog
OP5 =	" "	setShippingMethodDialog
OP6 =	" "	confirmationDialog
OP7 =	" "	paymentMethodDialog
OP8 =	" "	orderCompleteDialog
OP9 =	" "	deleteOrder

InterWorld Confidential - Do Not Duplicate

Benchmark Scripts
(Browse-Buy)

Home Page
Section List
Product Page
Basket
Order Process

OP1 =	ObjBuilder, OrderProcess	initialize
OP2 =	" "	verifyOrderDialog
OP3 =	" "	verifyBillingDialog
OP4 =	" "	verifyShippingDialog
OP5 =	" "	setShippingMethodDialog
OP6 =	" "	confirmationDialog
OP7 =	" "	paymentMethodDialog
OP8 =	" "	orderCompleteDialog
OP9 =	" "	deleteOrder

InterWorld Confidential - Do Not Duplicate

Benchmark Scripts
(Search-Browse-Buy)

Home Page
Section List
Search
Product Page
Basket
Order Process

OP1 =	ObjBuilder, OrderProcess	initialize
OP2 =	" "	verifyOrderDialog
OP3 =	" "	verifyBillingDialog
OP4 =	" "	verifyShippingDialog
OP5 =	" "	setShippingMethodDialog
OP6 =	" "	confirmationDialog
OP7 =	" "	paymentMethodDialog
OP8 =	" "	orderCompleteDialog
OP9 =	" "	deleteOrder

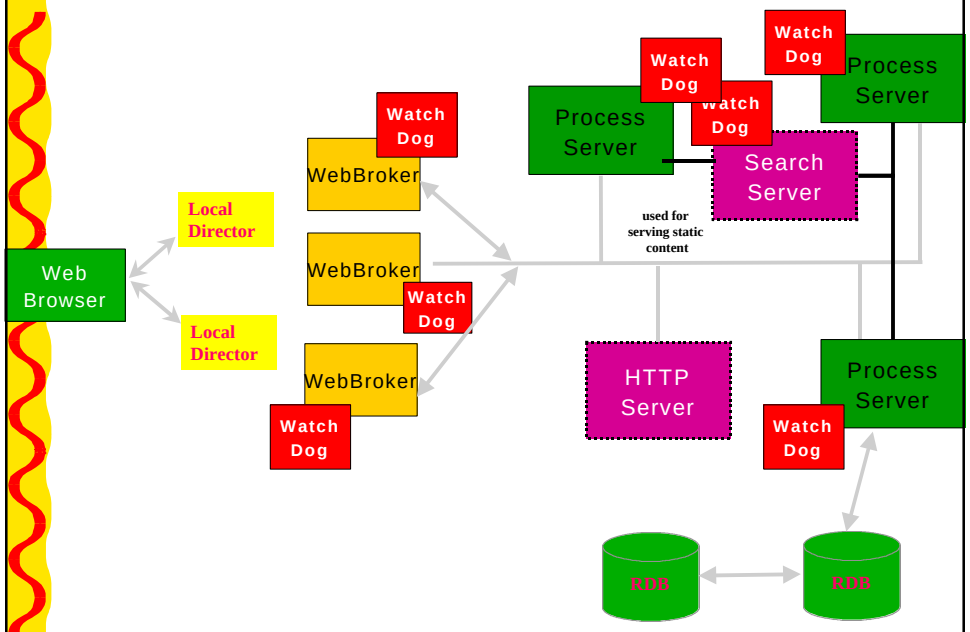
InterWorld Confidential - Do Not Duplicate

The Math

- TotalThroughput = WebBrokerEfficiency * (ThoughputPS1 + ThroughputPS2 + … + ThroughputPSn)
- WebBrokerEfficiency = F(number of process servers, network bandwidth, network latency)
- Throughput PS1 = (ThroughputPS1$_{browsing}$ * BrowsingPercent)+ (ThroughputPS1$_{shopping}$ * ShoppingPercent)+ (ThroughputPS1$_{purchasing}$ * PurchasingPercent)

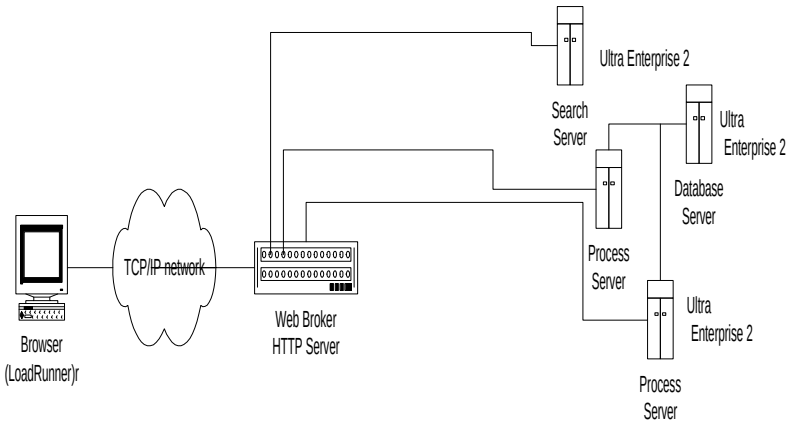
InterWorld Confidential - Do Not Duplicate

Sample Configuration



InterWorld Confidential - Do Not Duplicate

Sun Benchmark Configuration



InterWorld Confidential - Do Not Duplicate

Benchmark Results - Order Process Transaction

Test 1 – Order Process

Transactions	Average Response (sec)	Test Duration (sec)	Transactions/ Second	CPU (%)/ Memory Usage
5,017	NA	296	16.9	90% - 70MB

Sun Ultra Enterprise 2 - Oracle, 266 MHz, 1 CPU, 512 RAM

InterWorld Confidential - Do Not Duplicate

Benchmark Results - End to End

Number of Transactions- 4502 **static- 225, dynamic- 4277**
Duration in Seconds- 900 **15 minutes**

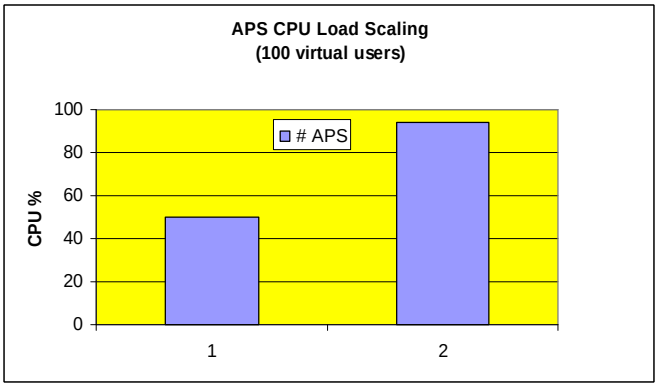
Transactions/Second- 5.1

search to browse to buy- 65 : 39 : 1
static to dynamic pages- 5 : 95
100 virtual users

Sun Ultra Enterprise 2 - Oracle, 1 APS: 266 MHz, 1 CPU, 512 RAM

InterWorld Confidential - Do Not Duplicate

Benchmark Results - CPU Utilization

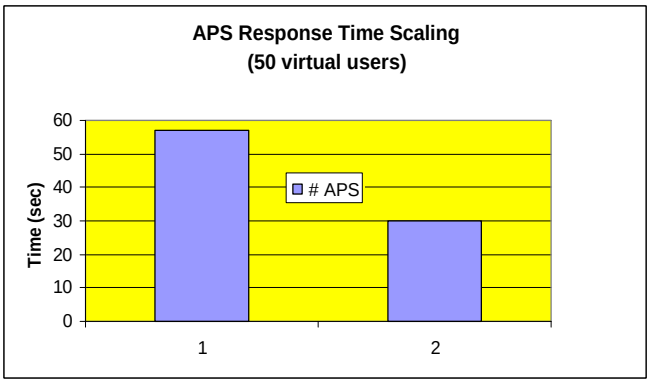


search to browse to buy- 65 : 39 : 1
static to dynamic pages- 5 : 95

Sun Ultra Enterprise 2 - Oracle, 266 MHz, 1 CPU, 512 RAM

InterWorld Confidential - Do Not Duplicate

Benchmark Results - Response Time Scaling

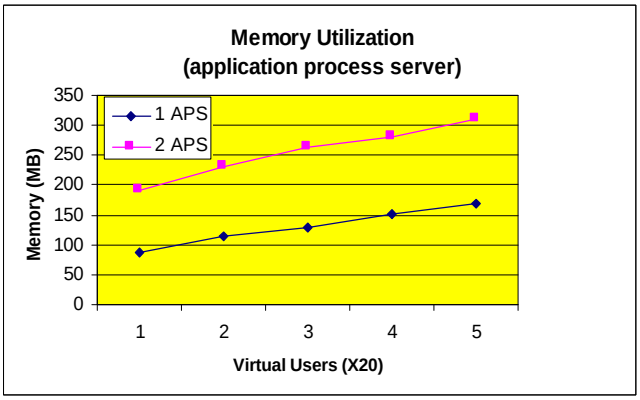


search to browse to buy- 65 : 39 : 1
static to dynamic pages- 5 : 95

Sun Ultra Enterprise 2 - Oracle, 266 MHz, 1 CPU, 512 RAM

InterWorld Confidential - Do Not Duplicate

Benchmark Results - Memory Utilization



search to browse to buy- 65 : 39 : 1
static to dynamic pages- 5 : 95

Sun Ultra Enterprise 2 - Oracle, 266 MHz, 1 CPU, 512 RAM

InterWorld Confidential - Do Not Duplicate

Database Server - Memory Utilization

Test 1 - Basket Operations (view, select, insert, remove, delete)
100 virtual users, 2 APS

CPU	Memory
20%	78Mb

Test 2 - Order Process - 100 virtual users, 2 APS

CPU	Memory
38%	140Mb

Test 3 - Search - 100 virtual users, 2 APS

CPU	Memory
55%	230Mb

Sun Ultra Enterprise 2 - Oracle, 266 MHz, 1 CPU, 512 RAM

InterWorld Confidential - Do Not Duplicate

Operating System: Solaris
Number of CPU's: 2
Memory: 1.4 GB RAM

Element	1 user load	5 user load	10 user load	20 user load	50 user load
Home Page					
CatalogList					
SectionList - Cached					
CatProduct - Cached					
SectionPage - Cached					
ProductPage - Cached					
Basket (IWX)					
BasketSteps					
OrderProcess (totals)					
- Initial Step					
- Verify Order					
- Verify Billing					
- Verify Shipping					
- Shipping Method					
- Confirmation					
- Payment Method					
- Order Complete					
- Delete Order					
ObjectBuilder					
- NOOP process					
- Static Template					
- Static Template + Database access					
Search					
Category Search					
Customer Registration					

Mix 85% to 15%	
Mix 95% to 5%	

Results vary based on site customizations

Use benchmarks as a guide. Your mileage may vary!

Units = Dynamic Pages/sec (transactions/sec)

InterWorld Confidential - Do Not Duplicate

Web Site Performance Q&A

- Do customers experience consistently good performance from your web site?
- How does your web site compare to your competitors or to industry benchmarks?
- Do customers in certain cities have more trouble with performance than from other cities?
- How does your site deal with heavy traffic and/or peak load periods?
- How reliable is your site in terms of connections refused, connection time outs and page time outs?
- Are certain pages consistently slow?
- Should geographic mirroring be considered for your site?

InterWorld Confidential - Do Not Duplicate

Benchmark and Performance Technology Assessment

Site Characteristics
- Determine customer value of each feature
- Remove non-essential steps
- Dynamic page analysis
- Turn off features (based on peak load scenarios)
- Remove or disable personalized pages (peak load scenarios)
- load characterization, site characterization

Assessment
- Determine customer expectations and evaluate the impact
- Who are you selling to and what do they expect to see next?
- Does the current infrastructure support the future?
- Cost and benefit analysis of new features vs. performance

Simulate Real World Load
- Number of users
- Types of transactions
- Database access
- Legacy connection(s)
- Network/infrastructure issues

InterWorld Confidential - Do Not Duplicate

Hardware Redundancy

- Separate ISPs**: Are you using two separate ISPs to handle the unlikely event that one ISP is down?**
- Mirrored Site**: Does your organization have a system to automatically reroute traffic to a mirrored site?**
- Multiple Commerce Servers**: Do you have multiple Internet commerce servers? [Cisco has LocalDirector]**
- Redundant Disk Drives**: Is transaction information stored on multiple drives? RAID arrays?**
- Dirty Power**: Do you have a uninterruptible power system for the commerce server? How long can you go without power?**

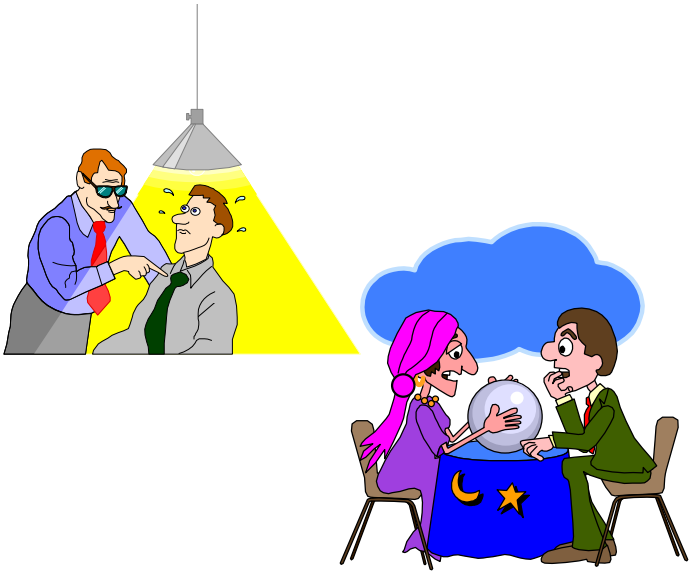
InterWorld Confidential - Do Not Duplicate

Contingency Planning

- Contingency Plan Preparation**: Do you have a documented contingency plan?**
- Contingency Plan Testing**: Do you test your contingency plan on a periodic basis? Do you employ a stress testing product that simulates disastrous or overload conditions in order to prevent them?**
- Penetration Testing**: Did you perform penetration testing of your commerce server before publicly announcing it's availability?**

InterWorld Confidential - Do Not Duplicate

Comments or Questions



InterWorld Confidential - Do Not Duplicate

Usability Analysis in E-Commerce Systems

Quality Week '99 Track Presentation

Federico I. Pacquing, Jr.
Director, Enterprise Support Systems
PhotoDisc, Inc - Seattle, WA

- Overview/Background
- Analysis -
  "What is the Problem"
- Design -
  "What is Usability"
- Implementation -
  "How do you practically get Usability??
- Test -
  "What were the results?"
- Maintenance -
  "Where do we go from here?"

Personal Background

- DOD QA and Process Improvement Manager
- General Manager QA of an e-commerce company
- Currently tasked with the development, test and delivery of enterprise systems for PhotoDisc

Company Background

•Publisher of High Resolution Digital Photography

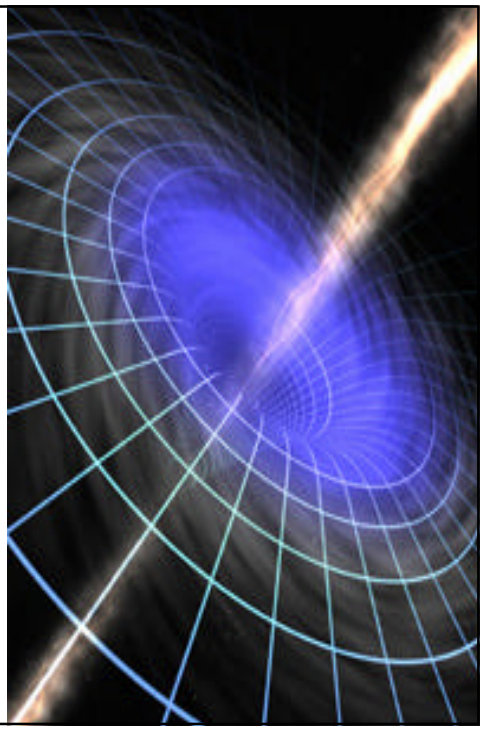
•Collection of 75,000 unique images

•Recently merged with Getty Images (GETY)

•E-Commerce enabled at www.photodisc.com

Project Background

- Consolidated results of four Seattle-based e-commerce companies
- Practical Applications of “Text book” approaches
- Oldest Implementation is 12 months old, newest is 4.
- Varying degrees of success (though there are more positive aspects than negative)



Analysis -

"What is the Problem"

- Unique issues with Usability
- Importance of Usability

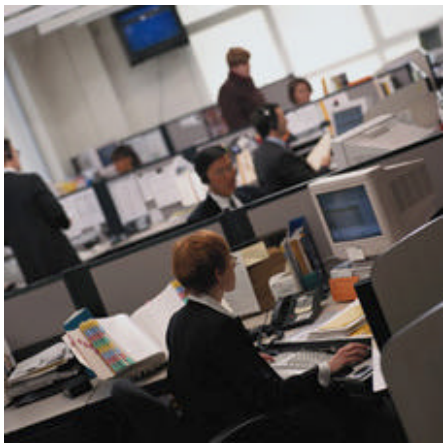
Usability <u>isn't</u>...

- ...a "fuzzy marketing" function where "clueless" users sit around a table and needlessly criticize developer's work…
- ...a thoroughly understood and well-represented software development activity…
- …something that can easily be sold to executive management…

Usability Analysis (UA) is a systematic approach used to gauge the efficiency and effectiveness of a system or product from the perspective of the end-user.

The Problem...

- Frustration among end-users
- Loss of Customers/ Revenue
- Increased Customer Service and Technical Support Calls



Resulting from...

- Inadequate “Requirements definition”
- Poor “review process”
- Improper implementation of initial requirements and design

Design -

“What is Usability”

- Text Book definitions
- “High level” look at activities

“User-Based Usability”

(Category I)

- Requirements Analysis
- User Profile Analysis
- User Context Analysis
- Alpha and Beta Groups

Design-Based Evaluations

(Category II)

- Design Guidance
- Heuristic Evaluation

Product/System Evaluation Methods

(Category III)

- Heuristic Inspection
- Usability Lab Experimentation
- Performance Measurement Method
- User Subjective Assessment
- Model Based Assessment
- Supportive Evaluation

Implementation -

"How do you practically get Usability?"

Program Buyoff and Setup

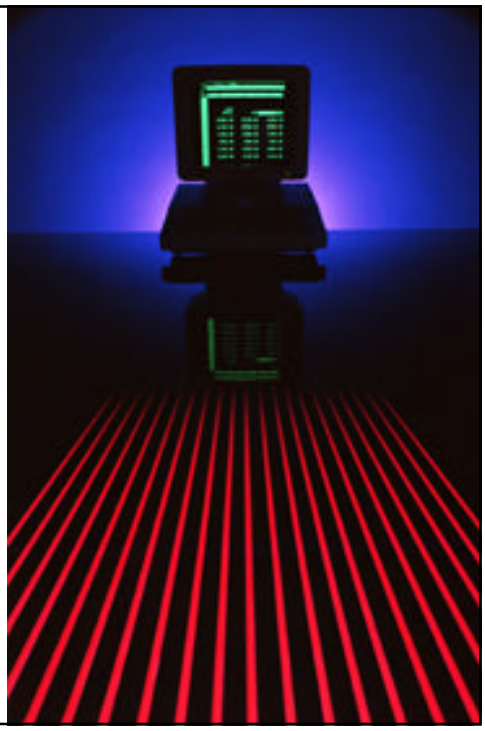
(Phase I)

- Find an Expert
- Initial Data
- The Business Case
- The Internal Champion

Initial Activities

(Phase II)

- Initial Programs
- Team Building



Growth

(Phase III)

- The Organization
- Integration with Current Processes

Full Program Implementation

(Phase IV)

- Expanding the Scope
- Automate/Formalize the Process
- Metrics

Test -

"What were the results?"

- What went right
- What went wrong

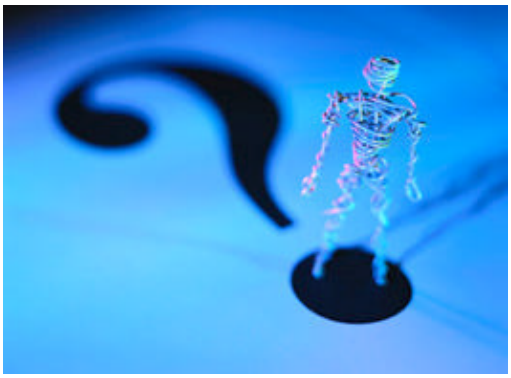
Maintenance -

"Where do we go from here?"

- Current Projects
- Future Expansions
- What I'd do differently

Usability Analysis in E-Commerce Systems

Quality Week '99 Track Presentation



Federico I. Pacquing, Jr.

Freddy.Pacquing@PhotoDisc.com

(206) 695-3466

Quality Week '99 Track Paper
Federico I. Pacquing, Jr.
Director, Enterprise Support Systems
MSCS, BSCS, CQE
PhotoDisc, Inc., Seattle, WA
Freddy.Pacquing@PhotoDisc.com

Usability Analysis in E-Commerce Systems

Abstract

Usability Analysis (UA) is a systematic approach used to gauge the efficiency and effectiveness of a system or product from the perspective of the end-user. While this paper and presentation will briefly discuss and help clarify what usability is, the true purpose is slightly more unique. Its primary focus is to describe from a practical perspective exactly how a Usability Analysis program can be implemented and integrated into a technical Quality Assurance process, what the benefits are to an e-commerce company and most importantly – how to sell it to executive management.

The information included in this paper was obtained from personal interaction with 4 Seattle-based e-commerce companies (most of which have asked to remain "unnamed" due to the proprietary nature of the processes and results). The methods and practices represent a "best of breed" approach to UA implementation.

This paper is organized and follows the logical flow of the specific phases typical to a software development lifecycle. An analysis of the problem set is included, as well as those tools necessary to "design" a proper UA program. Specific implementation tasks are listed, as well as brief results obtained after initial program implementation.

Prior to the implementation of a formal Usability Analysis Program, PhotoDisc was frequently criticized both internally and externally for not developing the "right product" or developing a product that was technologically superior, but incorporated a substandard "user experience". Our company came to the realization that in the end most e-commerce ventures develop a store – and that our goal was to provide the user a shopping experience on-par with Sax Fifth Avenue …not K-Mart.

ANALYSIS – "What is the Problem"

The Problem

Twelve months ago I had only a vague idea of what "usability" was.

If you had described the activities and processes involved with a typical usability program, I would have thought it a waste of valuable development and quality assurance dollars. What benefits could be gleaned by spending 5 hours watching tape after tape of

“clueless users” fumble around a perfectly good e-commerce site? Or, worse still, sitting in a cramped room staring at mockups of user interfaces imagining what “could happen” even before it’s coded?

Common test practices like test automation, code coverage, and even complexity analysis these, I could discuss and see the obvious benefits. Code and design reviews were a necessary evil, and development methodologies and concepts such as SEI CMM, RUP, or Object Oriented design were quite clear and obviously necessary (well, maybe not the SEI CMM). However, what were the benefits of evaluating the “user-friendliness” of a web site to any degree of depth?

And I was not alone. The implementation of a *Usability Analysis Program* is often overlooked in the development of a system, site, or product. I have found that this oversight can have an incredible impact on a product’s success – or lack thereof.

To put it succinctly, usability is the study and analysis of products with respect to ease of use, operating procedures, and user experience. While development can gain many advantages from the implementation of this process, a larger benefit can be derived by other organizations (such as Customer Service, Technical Support, or Marketing). In one instance, marketing was able to produce prototypes and receive almost immediate customer feedback on changes to the product line or concepts. In another instance, over 25% of the calls received from a Customer Service department were asking procedural questions about our product due to **poor usability**.

But though these and other benefits can be realized from a properly designed and customized Usability Analysis program, I found that “selling” the process is even harder than promoting the benefits of a typical QA program. The concept of a Marketing Department running focus groups is something companies are typically able to comprehend. The idea of a Quality Assurance group running a similar appearing function is many-times alien to upper-management and perceived as an unneeded expense and more importantly the loss of valuable time.

What is Usability?

So what is Usability? Usability Programs are used to help define consumer requirements and refine our product base. Web Application usability is complex, combining concepts from Technology, Marketing, Software Testing/Analysis and Cognitive Psychology.

However, the mastering of this concept has given our company an undeniable competitive edge in a fast growing and aggressive market.

The fundamental questions the Usability Analysis (UA) program answers are "Who are our customers?" "What are their buying, shopping and web surfing habits?" and most importantly, "How can our company capitalize on these behaviors?" This program specifically provides feedback for the development, marketing and the strategic placement of our systems. The Usability Analysis Program also furnishes technical evidence of site and product performance, timing and general "user-friendly" measurements and metrics. The power of this program comes from its ability to quantify the vague, user and customer related aspects of our products. The following section describes specific examples of activities and results seen in designing a typical usability program.

DESIGN – "What is Usability"

Type of Activities

To design a proper UA program, it is necessary to understand the various activities that are typically associated with it. Usability activities vary dramatically, in terms of effectiveness, resources, and overall purpose. Further, I have divided the activities into three categories based on similarity of function. Specific examples of the effectiveness of each will be discussed in the implementation section of this presentation. The intention of this section is to briefly provide a background of UA and provide a common frame of reference and set of terminology.

Category 1 – "User-Based Usability"

This stage consists of a strict analysis of the main user groups. In this category, the various target user groups' tasks, working environments, and overall requirements are analyzed. The statistics and profiles of the targeted consumer groups can be garnered from a company's marketing or business development groups (there should be a target audience for a set of goods or services or you have bigger problems than the lack of a Usability process!). Furthermore, in this area, usability and acceptability goals also are specified and design constraints identified.

It is important to the success of a venture to understand thoroughly the customer and his habits. This in spirit is the same type of activity that Analysts and QA professionals strive to accomplish – a complete analysis, with documentation, of the specific and exact requirements that need to be implemented. In this case, however, the UA activity focuses on the requirements and workflow as pertaining to the interface or the web.

Usability Context Analysis

This technique elicits detailed information about a product and its users. It also allows for an in-depth look at the product's uses from which a usability evaluation plan is derived. Stakeholders attend a facilitated meeting to fill in a questionnaire about a

product, what user group it will be used by, and how it will be employed. Context Analysis should be performed as early as possible in the design of a product and before evaluation work begins. The results of Context Analysis can be referred to and updated throughout a product's development.

Deliverables can include a Product Report (a description of the product in a standard format), a completed Context of Use questionnaire, and/or a Context of Evaluation document (a summary list of those factors in the questionnaire considered important for the usability of the product).

Alpha and Beta Groups

In my experience, Alpha and Beta testing in the e-commerce field has been under-utilized. True, these forms of testing provide Quality Assurance with valuable feedback on the technical validity of the site. It also provides marketing with the answers to "how does the audience respond to the product". However, in most e-commerce companies, formal UA processes and procedures are not used to analyze the user's patterns, problems or profiles.

For this discussion, consider *Alpha* groups as those groups developed using corporate personnel (both technical and non-technical) evaluating initial deployments and approving product prototypes. *Beta* groups, comprised of "Friends of the Company" or external personnel, can be created that mirror target audiences. Both user groups are surveyed, questioned and are allowed a "first look" at products and services.

Though typically the purpose of these groups in e-commerce is –

1) To provide feedback to the development and marketing organizations

2) "Shake out" defects prior to launch SQA has not find due to being "too close" to the project, or due to the sheer number of unique browser, platform and ISP configurations available

- Two additional and equally important intentions of properly architected Alpha and Beta groups should be to –

3) Allow a thorough analysis of the usability of the site

4) Provide feedback on the user.

Category 2 – "Design-based Evaluations"
This group of activities deals with concept development, testing and initial evaluation of a product's flow and implementation. In this category, simulation and prototyping methods are used to generate potentially high-level solutions and to test their feasibility.

Design Guidance

The method of Design Guidance classifies the application according to a number of key characteristics. These characteristics assist in defining the design. Furthermore, they help establish the main usability goals (in essence, this is a key requirement-gathering phase for UA activities).

Typically, design guidance activities provide direction on overall appearance, interface styles, dialogue structure, navigation and user support. The method is used as a reference guide to check points and ideas in relation to the current interface design and an evaluation against organizational design criteria. These organizational criteria can be contained in design guidelines and standards.

A design guidance session consists of internal Usability or QA personnel evaluating the potential design of a product or site. If referencing back to the typical development and QA lifecycle, it would map directly to standard design reviews of a program's architecture, data schema or program flow.

The deliverables from design guidance activities can include standard guidance on interface design, questions regarding workflow, task oriented procedures and a site design guide.

Heuristic Evaluation

Heuristic evaluation can be a powerful tool in the UA assessment of an e-commerce system. A Heuristic Evaluation can be applied to specifications, or more commonly, to user interface examples that can be assessed by a panel of "experts". This form of evaluation is applied to paper prototypes, running prototypes or different versions of a current production system.

But what exactly is a "Heuristic Evaluation"? Well, according to the dictionary, a heuristic is something that relates "to exploratory problem-solving techniques that utilize self-educating techniques (as the evaluation of feedback) to improve performance." Or, in plain English (and specifically for our discussion), a Heuristic is a predefined set "performance" criteria used to measure a potential product's usability. Examples would include flexibility and efficiency of use, consistency and standards, and help available. A Heuristic evaluation is therefore an evaluation of the product against performance criteria (typically a "walkthrough" or review process).

Applying Heuristic Evaluation to any system or prototype results in a detailed list of potential problems users are expected to encounter when using the application.

Problems are rated for severity and/or safety criticality – in essence, an evaluation of the risk involved with each problem. It is possible to extract the total number of problems that might be found in a system by extrapolating from the count of actual problems found in an application. Using this method gives quick and relatively inexpensive feedback to designers and business groups for user interface improvement, site performance and consumer experience. The process also allows stakeholders in the User Interface to comment on the system development in an open and accessible forum.

Deliverables from Heuristic Evaluations can include a list of problems identified by expert or user observation rated by severity.

Category 3 – "Product/System Evaluation Methods"
This category of activities deal with both the consolidation and rationalization of user requirements. Critical to a properly designed UA Program, evaluations of the product functions and features (based on the user context and the tasks that to be performed), methods of interaction, hardware interfaces, user documentation needed, and required user support. Plans for implementing the user requirements and defining how usability is built and developed into a system is crucial.

Heuristic Inspection

A Heuristic inspection can be applied to specifications, or to UI examples assessed by a panel of users. Like Heuristic Evaluations, a Heuristic inspection is usually applied to paper prototypes, coded prototypes and different versions of a production system. Deviations from Heuristic Evaluation will be stressed and examples from different case studies will be given. It is important to note that while Heuristic Evaluations are typically internally based, Heuristic Inspections are external – based on actual user feedback. The cost/benefits of the possible solutions to different user groups can also be analyzed.

.

Usability Lab Experimentation

Lab experimentation activities are the cornerstone of the Usability Analysis program – and those activities most typically associated with Usability. Using specifically developed questionnaires and select users from our target consumer groups, a UA group can conduct controlled experiments within a tightly controlled environment. A room or location can be allocated for this activity and be fully utilized. The power of this type of experimentation is that it allows companies to test the performance and user experience, and to gain immediate feedback on new product prototypes, UI changes, design issues and even customer concerns and suggestions.

Performance Measurement Method

This technique enables a detailed analysis of the workflow and patterns of actual target users. In this technique, the real world working environment and the product under development are simulated as closely as possible. Users undertake real work tasks while observers make notes, timings are taken, and video/audio recordings made. These observations are subsequently analyzed in detail. Usability metrics for site efficiency, effectiveness, problems encountered, and overall productivity are produced. User-based evaluation can be used whenever a working product - either a high quality prototype or fully released product - is available.

A potential deliverable can include a report, which incorporates a problem list and associated severity ratings, and recommendations for improvement. This report may also include usability metrics for user effectiveness, efficiency, productive period and satisfaction.

User Subjective Assessment

This is a method for analyzing the subjective opinions of users with practical experience of using a software product or service. These informed users complete a standard psychometric questionnaire. After receiving the questionnaire, their answers are analyzed with the aid of computer programs (to provide statistical consolidation of all feedback – manual methods can be tedious and can lead to inaccuracies). This User-based evaluation can be used to provide feedback at any stage of design.

Key outputs from this process can include a report which encapsulates the following measurements: global subjective measurements, perceived inefficiency, affect (likeability), control, learnability, helpfulness. Furthermore, an assessment of how these results compare with historical results for similar systems can be inferred (or deduced from a database of past results). Detailed analysis of how answers to individual questions differed from the norm may also be included.

Model Based Assessment (prototyping)

This method supports the evaluation of a prototype system, an associated task, workflow requirement or description. A simple test procedure evaluates and produces a list of ranked problems associated with a product. The Model Based Assessment can be applied early in the design process. This UA method can be seen as an alternative to more extensive trials, which commonly take place when a more completed product is to be evaluated prior to release.

Supportive Evaluation

The Supportive Evaluation is an activity that utilizes participatory workshops in which users examine prototypes and provide feedback in a structured manner. Scenarios and

exercises are used to create a more realistic context and simulate the hands-on experience. Product workflow management and integration aspects are also addressed.

IMPLEMENTATION – "How do you practically get Usability?"

Though the previously mentioned techniques and activities are critical to the understanding of a properly constructed UA program, the most important part of any new program is its implementation (for without the program buy-off and realization, complete knowledge of the tools and activities are futile). The following is a "best of breed" approach used in 4 Seattle-based e-commerce companies (most of which have asked to remain "unnamed" due to the proprietary nature of the processes and results). Though this presentation and method is e-commerce-centric, many of the concepts and methods relate to other industries and market segments.

Phased Approach

Unless your CFO is remarkably different than mine and every other I've encountered, he'll "flinch" at a proposed 5-15% increase in operating or development expenditure during any given quarter (if not laugh – or glare - you out of his office). Therefore, I've found that to effectively "sell" the program a "phased" approach is best for implementation. From a financial perspective the concept of a phase UA program is important – from the program's success, it is critical. In many instances, UA programs are even more difficult to sell than traditional QA Programs. QA results are easily quantified – for instance, the amount of code covered is "measurable" and simple to understand in concept. Test Automation scripts can be seen and their results are Boolean. However, most executive managers are reluctant to spend money on programs with little perceived short-term benefit to the bottom line. The solid usability program proponent's challenge is therefore to tie in it into either a direct reduction of cost (due to an elimination of operating inefficiencies) or a marked increase in revenue inflow.

Phase I – Program Buyoff and Setup

Phase I is critical to the success of a corporate UA program, and often the most difficult. The criticality of creating a business case that ties directly into the corporate financials and the "politicking" necessary to launch the program can make it questionable as to whether its worth spending the time. However, rest assured that after the analysis of user statistics and initial research the benefits will become clear.

Find an Expert

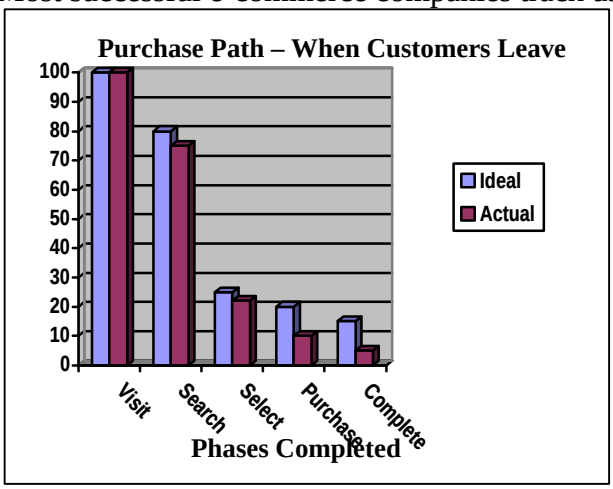
Finding an expert in the field of usability to mentor or teach was critical to the success of each of the programs studied. Either an industry leader who was able to provide valuable feedback on our plans, or an acadamian who is well versed in the theory of usability was crucial for an understanding of what how to quickly implement a UA program.

In the case of PhotoDisc, we developed relationships with a number of universities that have programs in usability in one form or another. The UA program utilized their skill, knowledge and facilities to help us define our process, procedures and questions. These

universities included the University of Washington, Central Washington University and Seattle University.

Analysis of and Gathering of Initial Data

Most successful e-commerce companies track user activity using one method or another



This data can be used by marketing organizations to track specific behaviors, or Quality Assurance groups to develop specific test cases (i.e., browser compatibility tests, platform, ISP, or connection rate statistics). It can also be used to identify problems with e-commerce site usability.

The above chart demonstrates an example of concerns that were found during an initial analysis of PhotoDisc's e-commerce statistics early this year. It was discovered that a startling number of potential customers left during a specific step of the purchase process. A full 17% of customers, for unknown reasons, left the site after choosing product and price but prior to completing the transaction. If even a small percentage of these customers were leaving due to usability issues, the team theorized that the cost associated with the problem would justify initial limited expenditures.

Another source of potential data is to "piggy-back" on Technical Beta tests. Questionnaires can be issued and informal analysis of user actions can be completed with minimal resources and impact on the schedule. This method has been effective to gather informal or ad-hoc information that can verify other sources of data. For instance, in the above example, to help verify that it truly was a usability issue that was causing the sharp decrease in user activity during the purchase step, informal UA activities were instituted.

A Heuristic inspection demonstrated that there were fundamental flaws in the design and implementation of the purchase procedure. What was thought as intuitive to the UI designers was too complex to allow for proper completion of the task (in this case, buying the product). To complete this Heuristic inspection, a total of 10 man-hours were necessary for setup by QA personnel. Added to this total, 10 Heuristic inspectors (who received 30 minutes of training) spent 2 hours in a Heuristic review session. In essence, to solve a

critical problem to the profitability of the site cost approximately 35 man-hours – a potential Return on Investment (ROI) of magnitude proportions.

The Business Case – "SHOW ME THE MONEY"

The creation of the business case to create a formal UA process was critical to the success of the project. From a financial perspective, it was key to effectively tie the program and it's benefits to the corporate bottom line. Though informal tactics and activities provide a limited ROI, the importance of creating a formalized UA program with the associated budget and resources was critical to get it effectively integrated into the rest of the development and QA processes. In essence, the best way to achieve project "buy-in" was to tie it, and to demonstrate that it made good business sense.

The obvious first step was to outline the phases of the project, the financial investment and return. To accomplish this, the creation of a budget and list of expenditures was in order. Personnel (added or reallocated), computers, video equipment, and even facilities should be included to demonstrate the total combined cost of an organization's UA program. In many instances, starting off with a limited budget and scope proves to be an easier sell than attempting to promote the benefits of a full-blown UA program. For equipment alone, the price tag to set up a single small room can equal $40,000 or more. When adding manpower requirements to run and administrate simple UA sessions, the cost can skyrocket. In the end, a limited win is better than no win at all.

Tying the project results to the financial baseline can be tricky. It is possible to use data from an initial analysis (mentioned above) and extrapolate potential paybacks. Furthermore, it is also helpful to use industry data from your research to provide executive management with a typical historical "payback" that others in the industry or market have achieved. This tactic needs to be closely monitored and the correlation made as close as possible to your specific market segment.

Find an Internal Champion

Finding a champion within the business community within your company who will provide support for the program is extremely important. I knew that in order to develop a successful program I would not only need the financial buyoff, but the "intellectual" buyoff from management on the fundamental need for the UA program. This champion needed an understanding of what the customer would need as well as a basic idea on how sites were technically implemented. In one case, this person was the Vice President of Marketing. In another, it was the CFO.

Phase II – Initial Activities

Initial Programs

Choosing the initial programs to demonstrate the power of the UA is a vital step. If you're able to achieve buyoff of the implementation of the program, many individuals within the company may still be skeptical. Therefore, it is important that the first initial programs are both high visibility and low risk.

Two specific activities initially implemented were the Heuristic Inspection and a simple model based assessment. Though these activities do not provide as much "bang for your buck" as a more requirements based analysis technique, they are both highly visible. A Heuristic Inspection can incorporate many company people – including program skeptics. A Model-Based Assessment can incorporate key users, the review of which can be seen by business executives. Both are highly interactive and dynamic activities, uncovering basic flaws in site and product execution. Furthermore, both activities have a tendency to be relatively inexpensive.

The cost, in terms of manpower, for a simple Heuristic Inspection has already been addressed. In fact, in this instance, no additional computer or equipment resources were needed to accomplish the task. Assuming that a working HTML prototype exists with limited functionality, a Model-based Assessment is also inexpensive. The only additional resource recommended would be a borrowed video camera and tripod to document the sessions for future analysis (and as a selling point for program expansion.)

Team Building – Profile of a UA Team

The profile of the perfect usability professional is difficult to determine. Likewise, the skills necessary for an ideal team is also problematic to exactly determine. Due to its very nature, the skills needed for proper UA include cognitive and behavioral psychology, statistics, User interface theory and design, simple coding knowledge and knowledge of hardware, networks and platforms (to name a few). Administration skills are needed, as well as group facilitation. All these skill sets are required (in some combination) to properly run and organize a UA program. The dilemma, of course, is to meet all these requirements and still stay within budget.

To overcome this dilemma, there are many alternatives. For an initial implementation of a UA program, I managed the process myself and included a QA Tester as the project lead. To help facilitate and run the program, we hired an intern (with a background in cognitive psychology) and a temporary person for the administration work (usability

programs tend to be extremely paper and scheduling intensive). This staff arrangement provided the basic necessary skill sets necessary for the initial UA program activities.

Phase III - Growth

The maturation and growth phases of a program are often times difficult to manage effectively. A maturing UA program is no different – the ability to gauge the program's effectiveness and control the growth and expansion in scope is difficult to anticipate even under the best of circumstances. There are two areas that are necessary to consider during this phase: Who should control the group and program long-term, and how the process is integrated with the development, quality assurance and business processes.

The Organization

One key danger is that due to its very nature, UA groups and programs can have a tendency to align more toward UI design groups, marketing or business development organizations, or even pure engineering rather than QA. For instance, due to the criticality of the data gathered, there is a strong "business twist" to the group. Also, the interface-intensive activities lend it to be very appealing for UI design groups to control the organization. However, it has been my experience that keeping UA activities within the scope of a QA group is critical to the program's success.

The reasons for the tight integration between UA and QA are many and varied. The chief rationale is that both UA and QA are evaluation-based activities (from a test-centric perspective, not process-centric). In essence, the goal of a properly designed and implemented UA program is to ensure that all interface, procedural and customer centered activities are evaluated and reported in a controlled and centralized manner. The goal of QA is essentially to do the exact same function for the technical aspects of the system, site or product. The synergism between the two specific groups can be a valuable tool in the successful implementation of cross-organizational "usability".

Integration with the Current Process

Integration with the current Business, QA and Development processes is key for the long-term viability of the program. An integrated UA set of procedures will help foster a change in organizational culture and allow for a more complete and thorough development of sites and products. The customer-centric processes and procedures add benefits in a variety of areas – even if a minimal amount of usability errors are located. Other functions can add value to the UA process or receive value from it. This areas or activities include competitor analysis, user profiling, and surveying.

Incorporating UA into current processes is relatively simple – if you have your current processes formalized and documented (if this is not the case, then again, you have bigger problems than the lack of a Usability program). Integration into an ad-hoc procedure is near-to impossible. The first step to integrate UA into your organizational processes would be to document exactly what your UA processes are. Included in the documentation would be the different activities associated with various phases of your lifecycle, the pre- and post-conditions of each activity, the deliverables from the activity, and the interaction with the diverse parties (Marketing department, UI Designers,

Developers). The next critical step would be to provide the exact workflow for a given project, as well as provide project or program management with the tools they need to insert the activities into their formal project planning (i.e., schedule and resource estimations). Finally, providing the cross activity infrastructure needed (as seen in the next section) such as the tools needed to track the information, a common defect severity scale (integrated with the QA scale) and the triage and fix procedures is critical to long-term success.

Phase IV – Full Program Implementation

Full program implementation is the obvious goal for the entire phased approach. If the previous phases have been implemented correctly, the business case for UA expansion should be relatively simple. If properly accomplished, the former phases would have enough "wins" and revenue-making examples to justify the additional expenditure.

Expanding the Scope

If not previously accomplished, the scope of the UA program should be expanded in this phase from a limited subset of projects to the entire product line or series of sites. It is beneficial to begin with a simple set of products – it allows for a controlled and structured approach to UA with a limited pool of resources. However, once complete buy-in is achieved, it is important to integrate the UA process and activities into all new and even legacy projects.

For instance, at PhotoDisc, the obvious place to begin the UA processes was with our front-end e-commerce systems. The benefits of UA could easily be seen, were highly visible to executive management, and had a high impact on both customer and revenue. This also allowed for an "easy win" for the program. Later, when the scope expanded, other systems such as financials, HR-Systems and the call-center were added. The focus of UA in these cases was not to make a significant revenue impact – it was to reduce the operating inefficiency through improved internal user usability. One key example is that, due to UA processes in the development of a single application, call-center response time improved by 30%.

Metrics

A strong metrics program associated with UA is important for a variety of reasons. The tracking and reporting of UA metrics can assist in the "sale" of the program, and also provide a valuable resource for the improvement of the program long-term. Standard

severity levels, and associated defect rates should be measured. In essence, a typical software test-like metrics should be measured, recorded and reported against to maximize the investment in time and resources.

Automate Process

In order to become more efficient, it is necessary at some point to partially automate the process. Programs which track the different information associated with UA are available off-the-shelf, however in our research we discovered that it was necessary to create a custom database application to consolidate and analyze the data. Project-based results were kept in a repository that tied to our automated test, project, process and defect control applications. This automation consolidation allowed for a tighter integration and level of operational efficiency.

TEST – "What were the Results?"

Post-implementation Statistics

The results of the implementation of UA within each of the companies is proprietary, however, it is possible to share some brief "high level" outcomes.

Positive Results

- One implementation improved profitability across the e-commerce site by an estimated 2% within 2 weeks.
- Another activity caused a decrease in Technical Support calls by 25%.
- At PhotoDisc, a simple heuristic evaluation and correction of key defects freed up 20-hours of work per week for a key financial analyst.
- A heuristic evaluation allowed for an improvement in search efficiency by 25%.
- Finally, a Model Based Assessment allowed for a more effective redesign and "rethinking" of a $250,000 e-commerce initiative

These are just a small subset of examples on how UA improved e-commerce corporate-wide. There are many more instances. In fact, for PhotoDisc, a 5% estimated improvement in site performance and a 10% increase in operating efficiency could be attributed to UA's effect. There were, however, some negative results that were associated with the implementations.

Not-so-Positive Results

- In one UA implementation, a project was delayed by 2 weeks due to UA involvement.
- In another, an increase in expenditure by 5% of the annual budget was attributed to UA.
- One site experienced a decrease in search effectiveness by 3%, and a decrease in sales by 2%.
- In one instance, Technical Support calls increased by 5% due to a change requested by a Heuristic Inspection.

Summary

Though there were a few "glitches" in each of the implementation, this program was seen to provide a real and tangible benefit to e-commerce companies. Usability Analysis activities were found to be versatile enough to "fit in" to any organizational structure or budget constraint. Once the practical constraints and problems were identified and controlled, the UA program was definitely able to provide a positive ROI.

In summary, UA programs provide valuable feedback to development, marketing and business executives to help them make informed, strategic decisions based on facts – not on assumptions.

BIBLIOGRAPHY/SUGGESTED READING

While the following books/articles are not specifically noted in this paper, I found them to be instrumental in my understanding of the issues relating to usability and/or implementation or procedural concepts and suggest their reading.

- Boehm, B. W. (1981). Software Engineering Economics. Prentice-Hall.
- Card, S. K., Moran, T. P., and Newell, A. (1983). The Psychology of Human-Computer Interaction, Lawrence Erlbaum Associates
- Flemming, Jennifer. Web Navigation: Designing the User Experience. O'Reilly & Associates
- Forsythe, Chris. Human Factors and Web Development. Lawrence Erlbaum Assoc
- **Nielsen, J. (1993). Usability Engineering. Academic Press.**
- Rubin, Jeffrey. Handbook of Usability Testing : How to Plan, Design, and Conduct Effective Tests. Wiley Technical Communication Library
- Spool, Jared M. (1998). Web Site Usability : A Designer's Guide (The Morgan Kaufmann Series in Interactive Technologies). Morgan Kaufman Publishers.

ABOUT PHOTODISC, INC.

A pioneer in its field, PhotoDisc, Inc. was the one of the first publishers of high-resolution digital photography on CD-ROM and online. The company's image collection is available to creative professionals in the advertising, graphic design, marketing communications, and web and multimedia design markets.

PhotoDisc's constantly refreshed 75,000-image collection is available in its entirety for search, purchase and download on the company's award winning web site located at **www.photodisc.com** . As well, 32,000+ of those images are available on more than 175 thematic CD-ROMs. CD-ROMs are available on the company's website, and worldwide through the company's catalogs and at major software retailers.

Founded in 1991, the company pioneered the royalty-free licensing model for stock-photography. Using technology to enable hassle-free, rapid-access image licensing, PhotoDisc has become a valuable resource for deadline driven creative professionals. Working with more than 100 noted photographers from around the world, the company carefully selects compelling images to add to the PhotoDisc collection. Each photograph selected must meet the company's high standards for exceptional technical and artistic quality. Advanced drum scanning technology is used to digitize the images, which are then color-corrected and color-managed throughout the production process to closely match each original both digitally and in print. The resulting PhotoDisc images are high resolution, 24-bit color, and ready for use with all popular desktop publishing software, graphics programs and multimedia applications.

The company's products are available from its 9 offices (London, Paris, New York, Hamburg, Tokyo, Stockholm, Dubai, Sydney, Seattle) as well as worldwide through its website (www.photodisc.com) and distributors in the European Community, Asia, Latin America, Canada and the Middle East. Recently merged into Getty Images, a NASDAQ-traded company (GETY), PhotoDisc joins Tony Stone Images, Hulton Getty, Gamma-Liaison, Energy Film Library and Allsport to provide a full range of creative content solutions to graphics, multimedia, web and publishing professionals around the globe.

Investigating Test Effectiveness on Object-Oriented Software - A Case Study

M. H. Kao, M. H. Tang and M. H. Chen
Computer Science Department
University at Albany
State University of New York
SUNY at Albany

Outline

- Object-Oriented Pitfalls
- Traditional testing techniques
- State-based testing
- Empirical studies
- Object-Oriented Metrics
- Conclusions and Future Work

Object-Oriented Pitfalls - Inheritance

A subclass may re-define its inherited functions and other functions may or may not be affected by the re-defined functions.

```
Class foo {
int local_var;
⋮
int f1() { return 1; }
int f2() { return 1/f1(); }
}
Class foo_child :: Public foo {
// child class of foo
int f1() { return 0; }
}
```


Object-Oriented Pitfalls

An object may be bound to different classes during the run time.

```
// beginning of function foo
⋮
P1 p;
P2 c;
⋮
return (c.f1()/p.f1());
// end of function foo
```

Traditional Testing Techniques

Are traditional testing techniques for the imperative programs adequate for testing object-oriented programs?

- Black-Box Testing
  - Program specification
  - Operational profile
- White-Box Testing
  - Statement coverage
  - Branch coverage
  - Data-Flow coverage
  - Path coverage

Class Testing Strategies

- Testing inheritance
- Testing polymorphism
- Function dependence class testing
- State-based testing
- Sequence testing
- Data-Flow testing

Empirical Studies

Applications:

1. System A - GUI
2. System B - Data logging system
3. System C - Network communication program

Faults

- *Type I*: Object-oriented faults
- *Type II*: Object management faults
- *Type III*: Traditional faults

Empirical Studies

	System A	System B	System C
Lines of code	5.6k	21.3k	16.0k
Number of classes	20	45	27
Number of known faults	35	80	85
Type I faults	5	15	10
Type II faults	6	13	7
Type III faults	24	52	68
OO faults(%)	31%	35%	20%

Functional Testing

	A	B	C
Number of test cases	100	403	326
Number of faults detected	17	42	52
Type I faults detected	1	4	5
Type II faults detected	1	6	3
Type III faults detected	15	32	44
Total faults detected(%)	48%	52%	61%
Total OO faults detected(%)	18%	35%	47%

Testing Strategies

Strategy I:

1. *functional testing*
2. *statement testing*
3. *decision testing*

Strategy II:

1. *functional testing*
2. *state-based testing*

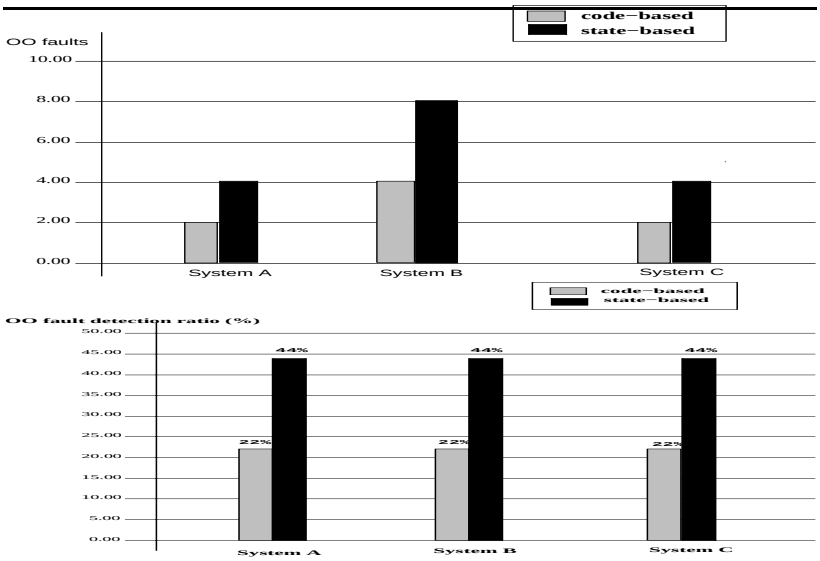
Data Analysis

Strategy	*code-based*			*state-based*		
System	A	B	C	A	B	C
Coverage	Statement			All-states		
NOT	46	55	52	21	37	38
Type I	0	1	0	1	3	2
Type II	1	1	1	0	2	1
Type III	5	6	4	4	8	10
CU	92%	91%	92%	87%	95%	90%
IC	8%	9%	8%	13%	5%	10%
Coverage	Decision			All-transitions		
NOT	23	41	33	43	79	75
Type I	0	1	1	1	2	0
Type II	1	1	0	2	1	1
Type III	1	6	5	4	6	8
CU	89%	89%	92%	89%	88%	85%
IC	11%	11%	8%	11%	12%	15%

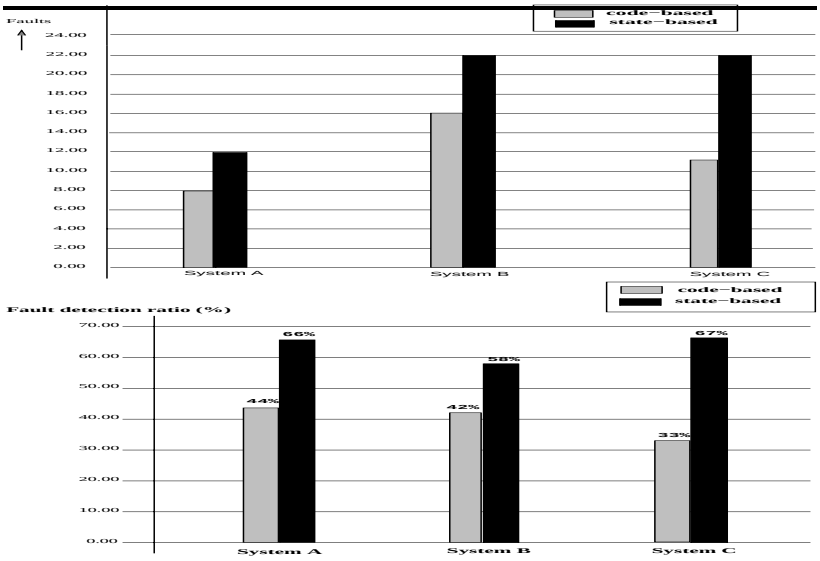
Data Analysis – Summary

System	A		B		C	
Strategy	I	II	I	II	I	II
NOT	69	64	96	116	85	113
OO faults	2	4	4	8	2	4
OO fault	22%	44%	22%	44%	22%	44%
Total faults	8	12	16	22	11	22
Total fault	44%	66%	42%	58%	33%	67%

Data Analysis



Data Analysis



Object-Oriented Metrics

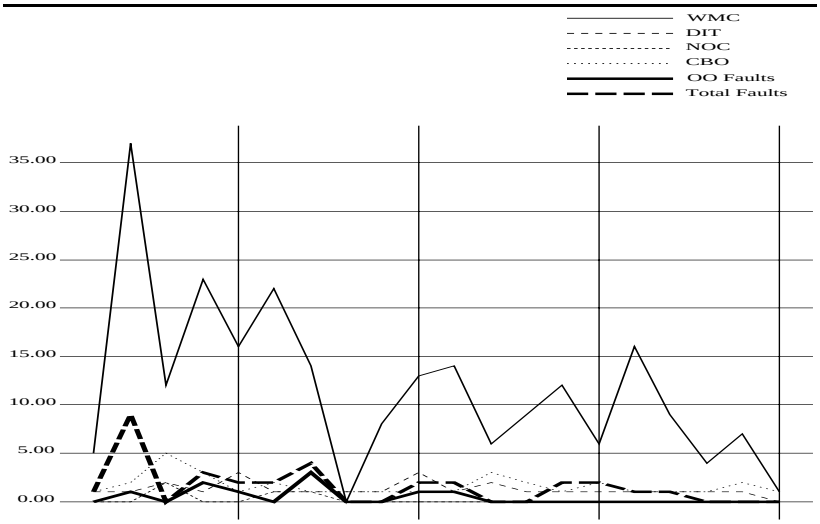
WMC : Weighted methods per class

DIT : Depth of inheritance tree of a class

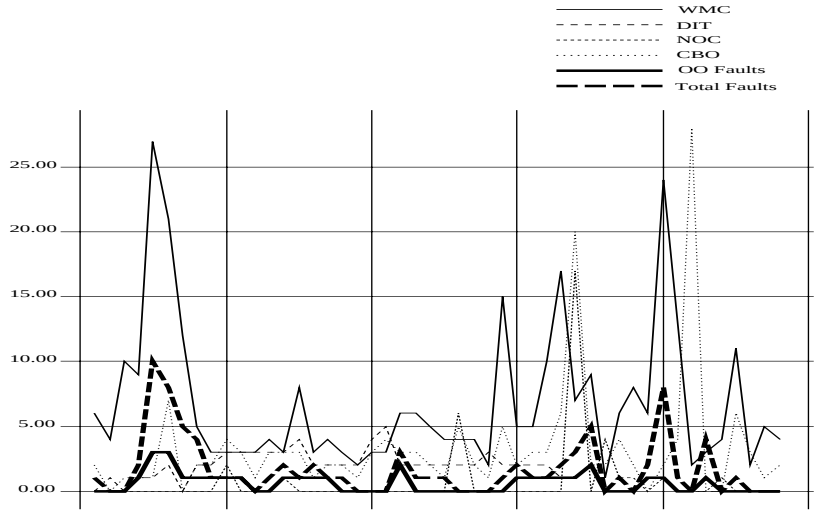
NOC : Number of children

CBO : Coupling between objects

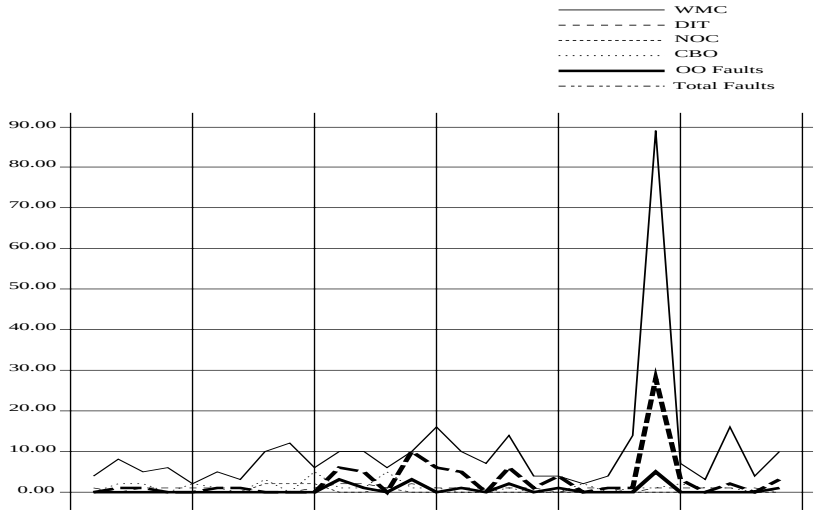
CK Metrics - System A



CK Metrics – System B



CK Metrics – System C



CK Metrics

	WMC	DIT	NOC	COB
$MEAN_{sysA}$	11.85	1.25	0.2	1.65
$MEAN_{sysB}$	6.81	1.54	0.71	3.29
$MEAN_{sysC}$	10.38	0.9	0.24	1.17
Type A	28%	37%	57%	16%
Type B	36%	27%	7%	49%
Type OO	50%	39%	14%	83%

Conclusions and Future Work

In our study:

- Code-based and State-based testing strategies are not reliable w.r.t. OO faults.
- CK metrics are not good predictors of fault-prone classes.

Future Work:

- Developing an object-oriented testing strategy
- Identify new design metrics for object-oriented programs

Investigating Test Effectiveness on Object-Oriented Software - A Case Study

Ming-Hung Kao and Mei-Huei Tang and Mei-Hwa Chen
Computer Science Department
SUNY at Albany
Albany, NY 12222
(kao, meitang, mhc)@cs.albany.edu

Abstract

We present a case study that investigates the effectiveness of traditional testing techniques as well as existing state-based testing strategy on detecting faults in object-oriented programs. In this study, we applied a black-box approach, functional testing and two white-box approaches on which statement and decision coverage criteria were used in test case selection and as a guideline for determining test adequacy. To apply the state-based testing strategy on object-oriented programs instead of on each class individually, we introduce two coverage criteria: all-states and all-transitions. For each execution of a test case, the visited states and the transitions to enter these states for each class are recorded. New test cases are designed to traverse the uncovered states and transitions. These testing techniques were applied to three industrial systems with sizes ranging from 5.6k to 21.3k LOC. The investigation began with classifying faults found in these three systems over the past three years. Based on their relevance to the object-oriented features, the faults are classified into three types where *type I* is strongly related to the object-oriented features such as inheritance and polymorphism; *type II* is related to object management, and *type III* is the type of faults that can be found in the non-object-oriented software as well. After applying these testing techniques to the faulty systems, we observed that the majority of *type I* and *type II* faults still remained in the systems. This result implies the likelihood that traditional testing techniques are not adequate for detecting object-oriented faults and the state-based testing is not good enough to address OO faults. Furthermore, we investigate the feasibility of using existing OO metrics to estimate the percentage of OO faults in a given system. Our results show that three out of the four CK metrics we used might not be sufficient as predictors of fault-prone and OO fault-prone classes. Our observations suggest that other metrics, that can capture the dynamic behavior of the program and the scenarios on which the instances of the classes are referenced, might be better indicators for the fault-prone and OO fault-prone classes.

Keywords: statement testing, decision testing, state based testing, object-oriented metrics.

1 Introduction

Object-oriented programming has been increasingly adopted for software development. OO features such as information hiding, encapsulation, inheritance, polymorphism and dynamic binding facilitate reuse and component-based development. As a consequence of the emerging use of OO techniques, testing object-oriented software has become a critical issue.

Although much work has been proposed at the class level on OO testing, very few studies are concerned with the program level. Thus, practitioners often question whether traditional testing techniques that are designed to test imperative programs can be applied in the object-oriented paradigm. In this paper we present a case study to address this issue. We obtained three industrial object-oriented systems ranging from 5.6K to 21.3k to be used in our study. These three systems have been used by many manufacturing companies for a number of years. They were tested using a functional testing in the system test phase and released to the customers. In addition, we obtained the test reports and trouble reports of these systems which record all the faults detected in functional testing as well as those reported by the customers over the past three years.

We first conducted a fault analysis on these faults and classified them into three different categories, where *Type I* faults are those introduced by the object-oriented features, such as inheritance and polymorphism; *Type II* faults are encountered in object management and *type III* are the traditional faults such as missing statements. *Type I* and *Type II* faults are considered OO faults, since they are strongly related to objects and object-oriented features; while *Type III* faults are considered non-OO faults they can also be found in traditional non-OO programs. The first investigation is to identify how reliable functional testing is with respect to these three types of faults. We observed that this testing technique is much more sensitive to non-OO faults than to OO faults. The second is to investigate if, by applying a stronger testing technique on the systems after functional testing, this phenomenon can be changed. Two strategies were used to enhance the testing process: *Strategy I* using statement and decision testing; *Strategy II* applying the state based testing. The results show that with *strategy I* the majority of OO faults still remained undetected, with less than half of the total faults detected. While with *Strategy II* two-thirds of the total faults and 44% of OO faults were detected. These results suggest that neither strategy is adequate for testing object-oriented programs and further research on developing a new strategy to take OO features into account is necessary.

Furthermore, we measure the CK metrics of the three systems aiming at identifying a correlation between these metrics and the distribution of each type of fault, so that, based on the structure and the complexity of the system, one can allocate test resources more effectively. However, the results from our study show that among the four CK metrics we used, only NOC can be useful for identifying fault-prone modules, the others can not be used as indicators. To this end, further research into the identification of new metrics is essential.

The remainder of this paper is organized as follows: in Section 2 we give a description of

the systems used in this study. The results obtained from the fault analysis are presented in Section 3. Section 4 describes the experiments conducted in this study and Section 5 presents the CK metrics of the systems. Our conclusions and the direction of future research appear in Section 6.

2 Descriptions of the Applications

The study was conducted on three industrial applications that were tested using tradition functional testing techniques in the system test. A description of these applications and the details of this study are presented in the following sections.

The applications that were used in this empirical study are subsystems of an HMI (Human Machine Interface) software which is a fully networked Supervisory Control and Data Acquisition system. This software is based on client-server architecture consisting of servers and clients. Servers are responsible for the collection and distribution of data. Clients connect into servers and have full access to the collected data for viewing and control actions. This software, which consists of more than 200 subsystems and 3 million lines of code, has been used by many manufacturing companies for several years. These subsystems are briefly described below.

Although each subsystem selected plays a different role in the system and performs different functionality, they share some similar characteristics that meet with our selection criteria. All the subsystems are implemented using C++ and possess certain object-oriented features such as encapsulation, inheritance and polymorphism.

System A is a user interface-oriented program that allows customers to configure the basic product operations and device communications. It consists of 20 classes that define 256 new, re-defined or virtual functions; its complexity is approximately 5,600 lines of code. System B is a real time data logging process that collects data as needed and logs data into the database based on the user configuration. This system defines 46 classes and 353 new, re-defined or virtual functions and with its complexity approximately 21,300 lines of code. System C is a communication-oriented program that acts as a router and not only delivers message between processes within the same host but also forwards messages to other hosts. This system defines 27 classes and 293 new, re-defined or virtual functions and contains approximately 16,000 lines of code. Among these three systems, Systems A and B were designed and implemented in a certain object-oriented methodology, whereas System C was ported from an older operating system and partially redesigned and enhanced to support the new requirements under a new environment.

In Table 1 we summarized the results obtained from the analysis of these three systems. The table lists the sizes of the systems in thousands of lines of code and the number of classes defined in the systems. The total number of known faults with their distributions in the three fault types is presented and the percentages of *type I* and *type II* faults over the total known faults are also included. We observed that among the total faults found, one third of the faults in Systems A and B are OO faults and in System C about one fifth of these faults

	System A	System B	System C
Lines of code	5.6k	21.3k	16.0k
Number of classes	20	45	27
Number of known faults	35	80	85
Type I faults	5	15	10
Type II faults	6	13	7
Type III faults	24	52	68
OO faults(%)	31%	35%	20%

Table 1: Summary of the faults in the three systems.

are of this type. The detailed design metrics of these systems are presented in Section 5.

3 Fault Analysis

We analyzed the trouble reports of these systems recorded for the past three years and classified the faults found in the system test and the maintenance phases along with a few new faults that were detected in our experiments. The classification scheme was based on the nature of these faults and their relevance to the object-orientation. Below we describe the detailed classification of these types among which *Type I* and *Type II* are OO faults and *Type III* faults are non-OO.

Type I: Object-oriented faults that are strongly related to the OO features and are introduced by these features such as inheritance and polymorphism. This type of fault can be further divided into two sub-categories: inheritance faults and polymorphism faults. A typical inheritance fault occurs when a derived class modifies the data members of the base class, which in turn changes the behavior of the base class and then causes the fault. In other words, the derived class changes the environment of the base class which causes the faults encountered in either the derived class or the base class. Polymorphism faults are the faults encountered in the OO program when an object can be bound to different classes during the runtime. For instance, if there are X possible bindings of an object which sends a service request and Y possible bindings of the other object which provides the service, then totally there are X x Y different combinations of bindings during the runtime. If some of them are not tested during a system test phase, then a failure, caused by a polymorphism fault, might occur.

Type II: The object management faults that are related to object management such as object copying, dangling reference, object memory usage faults and etc. A typical object copying fault would be encountered, if the implementation of the method for the copying of an object is either the duplication of the original object or the generation of a reference to the

	System A	System B	System C
Number of test cases	100	403	326
Number of faults detected	17	42	52
Type I faults detected	1	4	5
Type II faults detected	1	6	3
Type III faults detected	15	32	44
Total faults detected(%)	48%	52%	61%
Total OO faults detected(%)	18%	35%	47%

Table 2: Results of the functional testing.

original object. If the copied method is used incorrectly, some unexpected faults or memory corruption will occur. The dangling reference object fault means that an object, say "object A," tries to reference another object, "object B," which was destroyed by a third object, "object C." The object memory usage fault normally refers to the situation where an object allocates the memory during the runtime and does not free up the memory. when is is no longer needed.

Type III: The traditional types of faults that are not related to objects. They fall into the fault classification of traditional software [10].

In Table 1 we summarized the results obtained from the analysis of these three systems. The upper half of the table lists the sizes of the systems in thousands lines of code and the number of classes defined in the systems. The total number of known faults and their distributions in the three faults types is presented with the percentages of *type I* and *type II* faults to the total known faults also included. We observed that among the total faults found, one third of the faults in Systems A and B are OO faults and in System C , these faults types comprise about one fifth of the total faults.

Table 2 shows the results of the functional testing which were the data recorded in the test documents obtained from the company that developed the systems and from our analysis. The data we obtained includes total number of test cases used and total number of faults detected during the functional testing phase. In addition, we listed the number of faults detected for each fault type, the percentages of the total faults detected and total OO faults detected against the total number of known faults in the systems. We observed that in functional testing 48% to 61% of faults were detected, while only 18% to 47% of OO faults were detected, i.e., 53% to 82% of OO faults remained in the systems when the systems were delivered. This analysis suggests that traditional functional testing is less sensitive to the OO faults than the non-OO faults.

4 Description of the Experiments

The objective of these experiments is to investigate the fault detectability of traditional code based testing strategy and the existing state based testing strategy. Two experiments were conducted in parallel, in which each applied one testing strategy. *Strategy I* is a code-based testing using statement and decision coverage criteria to design test cases. The state-based testing was adopted as *strategy II* in which we define all-states and all-transitions coverage criteria to perform the state-based testing at program level. For each execution of the program, the visited states and executed transitions in the state diagram are recorded. Test cases are designed to exercise the unvisited states and transitions of the classes. This testing, unlike the existing state testing [2, 4, 5, 6, 7, 11], is not performed at the class level. Thus, there is no need for test stubs and test drivers. Both strategies were applied after the functional testing; therefore, the version of the systems we used in these experiments is the one released which still contains the faults reported by the customers as well as those detected in these experiments.

4.1 Code-based testing

In this experiment, two commercial tools, Visual PureCoverage from Rational Software Corporation and DeepCover from Reliable Software Technologies, are used to measure the statement and the decision coverage, respectively.

The steps of these experiments are described below; at each step we intended to reach 100% of the target coverage criterion, whereas with the existence of infeasible paths we tried to obtain the maximum possible coverage.

Step 1: using Visual PureCoverage to re-execute the test cases used in the functional testing to measure the cumulative statement coverage.

Step 2: Constructing test cases to execute all the uncovered feasible statements.

Step 3: Using DeepCover to re-execute the test cases used at Step 1 and 2 to measure the cumulative decision coverage.

Step 4: Constructing test cases to execute all the uncovered feasible decisions.

Faults that were found at Step 2 and Step 3 were corrected; among them, a few were not yet reported by the customers, i.e., these faults were not recorded in the trouble reports. The corrected programs were re-executed using the previous test cases to measure the cumulative coverage.

4.2 State-based testing

To perform state-based testing, we first create a state diagram for each class based on its specification. Each state diagram is maintained using a state list indicated by the state

Strategy	*code-based*			*state-based*		
System	A	B	C	A	B	C
Coverage criterion	Statement			All-states		
Number of test cases	46	55	52	21	37	38
Number of detected *Type I* faults	0	1	0	1	3	2
Number of detected *Type II* faults	1	1	1	0	2	1
Number of detected *Type III* faults	5	6	4	4	8	10
Cumulative coverage achieved	92%	91%	92%	87%	95%	90%
Infeasible coverage	8%	9%	8%	13%	5%	10%
Coverage criterion	Decision			All-transitions		
Number of test cases	23	41	33	43	79	75
Number of detected *Type I* faults	0	1	1	1	2	0
Number of detected *Type II* faults	1	1	0	2	1	1
Number of detected *Type III* faults	1	6	5	4	6	8
Cumulative coverage achieved	89%	89%	92%	89%	88%	85%
Infeasible coverage	11%	11%	8%	11%	12%	15%

Table 3: Summary of the two experiments.

id and each state is associated with a transition list that records a list of ordered pairs in which a transition id and the id of the destination state are maintained. We introduce two coverage criteria-all-states and all-transitions- to observe which states and which transitions have been executed under some tests. To keep track of the all-states and the all-transitions coverage, a state checker subroutine is instrumented to each member function in each class, so that, when the state of the object is changed and a transition is fired, the state and the transition list can be updated. In addition, the state checker is inserted into the program right after each statement that explicitly changes the value of a data member of an object.

The steps of the experiment are described in the following.

Step 1: Re-executing test cases used in the functional testing to identify which states and which transitions have been traversed.

Step 2: Creating new test cases to traverse all the uncovered feasible states.

Step 3: Creating new test cases to traverse all the uncovered feasible transitions.

4.3 Data analysis

Data obtained from this case study are presented in Table 3, where the numbers of test cases used to fulfill the requirement of every coverage criterion, the numbers of faults detected for each fault type, the cumulative coverage achieved and the percentage of infeasible paths for the three systems using three different strategies are presented. The results obtained from

System	System A		System B		System C	
Strategy	I	II	I	II	I	II
Number of test cases	69	64	96	116	85	113
Number of detected OO faults	2	4	4	8	2	4
Number of detected faults	8	12	16	22	11	22
OO fault detection ratio (%)	22%	44%	22%	44%	22%	44%
Total fault detection ratio(%)	44%	66%	42%	58%	33%	67%

Table 4: The fault detection ratios of the two strategies.

each strategy are summarized in Table 3 and For all three systems the code based strategy, *Strategy I*, detected 22% OO faults and the state based strategy, *Strategy II* detected 44% of OO faults. Therefore, the strength of detecting OO faults can be compared as 1 : 2, i.e., the state-based is two times more effective than the code-based strategy. However, neither of them detected half of the OO faults. Figure 1 and 2 depict these results.

The second comparison was based on the total fault detection ratio, which is the total number of detected faults, including all three types, against the total number of faults remaining in the systems after the functional testing prior to these experiments. The code based strategy detected 33% to 44% of faults; the state based detected 58% to 67% of faults. We observed that for all three systems, the state-based strategy detected more faults than the code-based, yet only up to 67% of faults were detected. These results are depicted in Figure 3 and 4.

5 Object-Oriented Metrics

It is often desirable that the fault-prone modules and the types of residential faults can be estimated based on some quantitative measurement of a given system. Object-oriented metrics are developed to realize the structure and the characteristic of object-oriented programs. Some metrics, such as CK metrics [3], have been proven empirically to be useful for the prediction of fault-prone modules [1].

In this study, we measured the CK metrics of the three systems, described in Section 2, and analyzed their distributions in the systems.

5.1 CK metrics

The object-oriented metrics proposed by Chidamber and Kemerer [3] are described in the following.

Weighted methods per class (WMC): This measures the complexity of an individual class. Two different weighting functions are considered: WMC1 uses the nominal

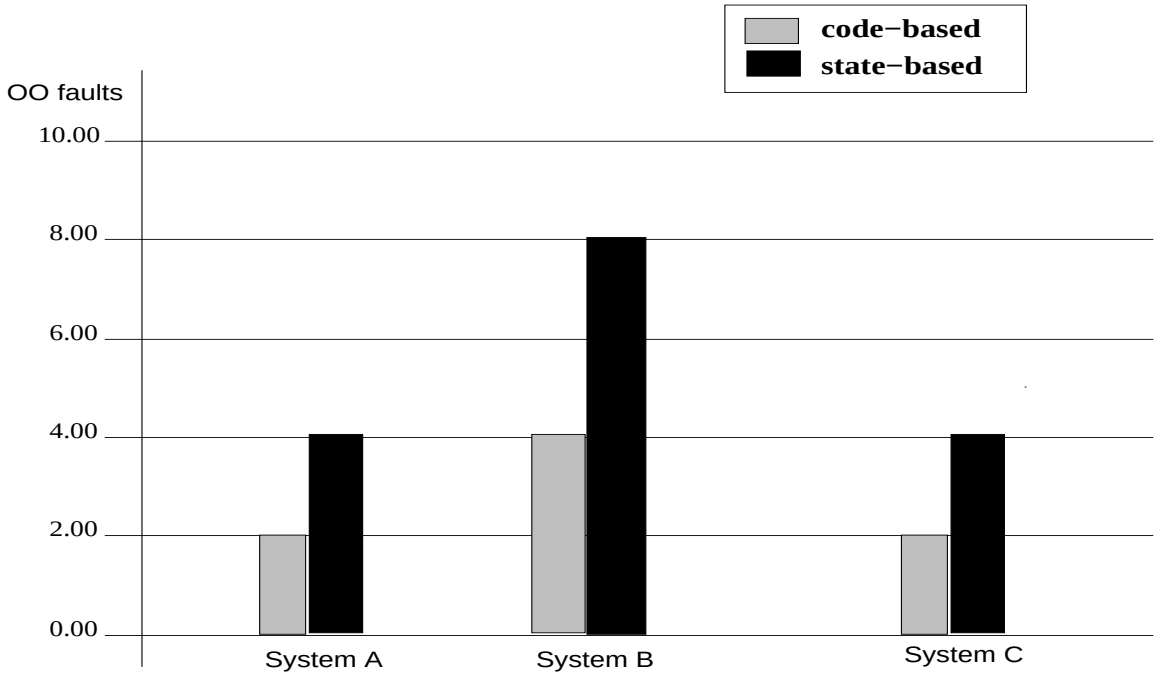


Figure 1: Summary of the OO faults detected.

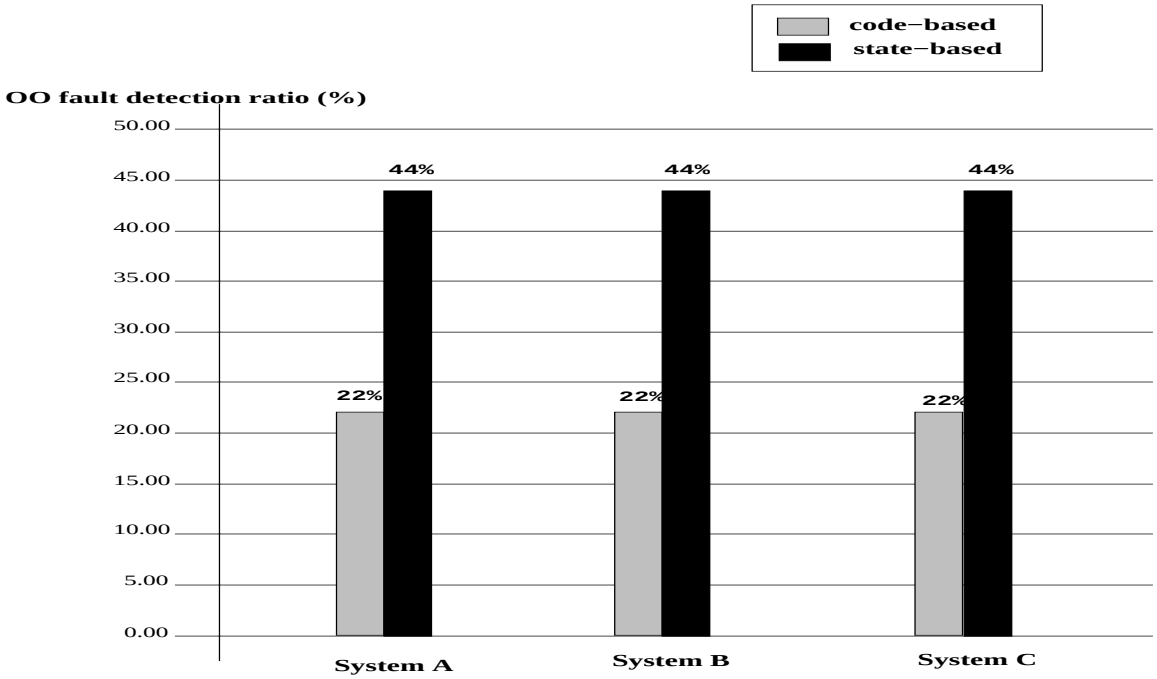


Figure 2: Summary of the OO fault detection ratios.

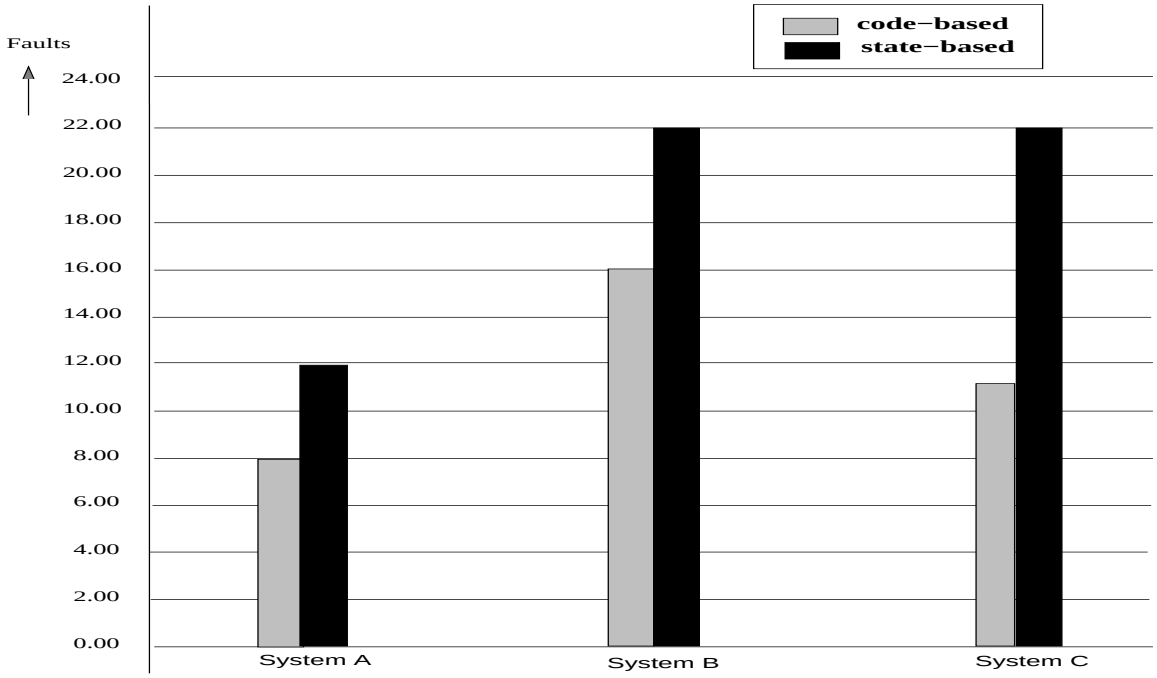


Figure 3: Summary of the total faults detected.

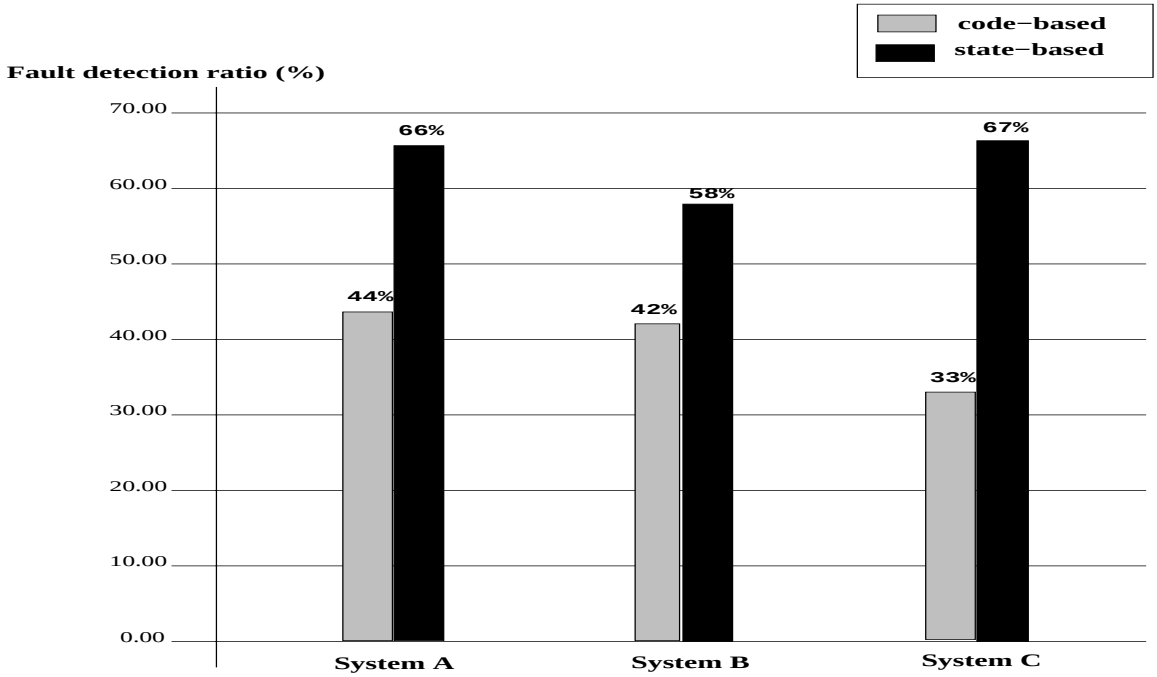


Figure 4: Summary of the total fault detection ratios.

weight of 1 for each function, and hence measures the number of functions, WMC uses a weighting function which is 1 for functions accessible to other modules, 0 for private functions. In this study, we adopted the first approach to simplify the factors. In another word, we consider all methods of a class to be equally complex.

Depth of inheritance tree of a class (DIT): It is defined as the length of the longest path of inheritance ending at the current module. Intuitively, the deeper the inheritance tree for a class, the harder it might be to predict its behavior due to the interaction between the inherited features and new features.

Number of children (NOC): It represents the number of classes that inherit directly from the current class. Moderate values of this measure indicate scope for reuse; however, high values may indicate an inappropriate abstraction in the design. Furthermore, a class with large number of children has to provide more generic service to all the children in various contexts and must be more flexible. We believe that this tends to introduce more complexity into this class.

Coupling between objects (CBO): This provides the number of other modules that are coupled to the current module either as a client or a supplier. A class is coupled to another one if it uses its member functions and/or instance variables. Excessive coupling indicates weakness of module encapsulation and may inhibit reuse. The assumption behind this metrics is that highly coupled classes tend to introduce more faults caused by inter-class activities.

5.2 Analysis

In figure 5-7 we list the four metrics measured, OO faults and total faults detected in each class, for the three systems. The results obtained from the analysis of the CK metrics on the three systems are presented in Table 5. The three mean values: $MEANsysA$, $MEANsysB$ and $MEANsysC$, are the average of each metric measured from the three systems, respectively. For WMC, DIT and COB, Type A denotes the percentage of classes, over the total number of classes, with metrics greater than the mean, which are faulty. Type B, on the other hand, is those classes that have metrics less than the average, yet contain faults. For OO faults, we compute Type OO which is the percentage of classes, over the total classes that contain OO faults, which have metrics less than the average of the system. For NOC, on the contrary, Type A are the classes with an NOC value which is less than the average. Type OO classes have a greater than average NOC value.

Type A classes follow the statement suggested in [1], whereas, Type B classes are those which would be overlooked if these metrics were used as the predictors of fault-prone classes. We noticed that up to 49% of classes would be overlooked if COB is used, 36% and 27% would be overlooked if WMC and DIT are used. Only NOC can be a good indicator for our system. For OO faults, 50%, 39% and 83% of classes would be overlooked if WMC, DIT and COB are used. Only 13.9% would be overlooked if NOC is used.

	WMC	DIT	NOC	COB
$MEAN_{sysA}$	11.85	1.25	0.2	1.65
$MEAN_{sysB}$	6.81	1.54	0.71	3.29
$MEAN_{sysC}$	10.38	0.9	0.24	1.17
Type A	28%	37%	57%	16%
Type B	36%	27%	7%	49%
Type OO	50%	39%	14%	83%

Table 5: Summary of the CK metrics analysis.

Therefore, based on our study, we observe that it is necessary to identify other metrics with which these metrics can be used as indicators for the fault-prone classes and for the OO fault-prone classes.

5.3 Discussion

In [1] shows that WMC is related to the number of faults reported for the class. In general, the larger the WMC, the larger the probability of fault detection. However, we observe that 36% of the classes are faulty yet have low WMC values. This scenario can be explained by the following two observations:

1. The complexity of the method is not considered. For example, a method with 1000 lines of code is likely to introduce more faults than a method with 100 lines of code.

2. The dynamic behavior is not considered. For example, a class which is used more frequently than the other classes tends to have more faults reported than the other classes.

On the other hand, by examining the class level *type I* fault distribution in our study, we failed to identify any single metric that is strongly related to the probability of *type I* fault detection This observation can be explained by considering the scenario under which this type of fault was introduced. Since the *type I* faults are caused by OO features such as inheritance and polymorphism (dynamic binding), we feel that not only the DIT, but also the following information, contribute to the probability of *type I* fault detection:

1. The number of child classes, including those who inherit directly and indirectly from the current module. The existing NOC metric reflects only the number of direct descendants for each class. Therefore, the additional complexity introduced by indirect descendants is not considered.

2. The function dependency relationship between the inherited methods and the new/redefined methods in child classes. A method is *function dependent* on another method if this

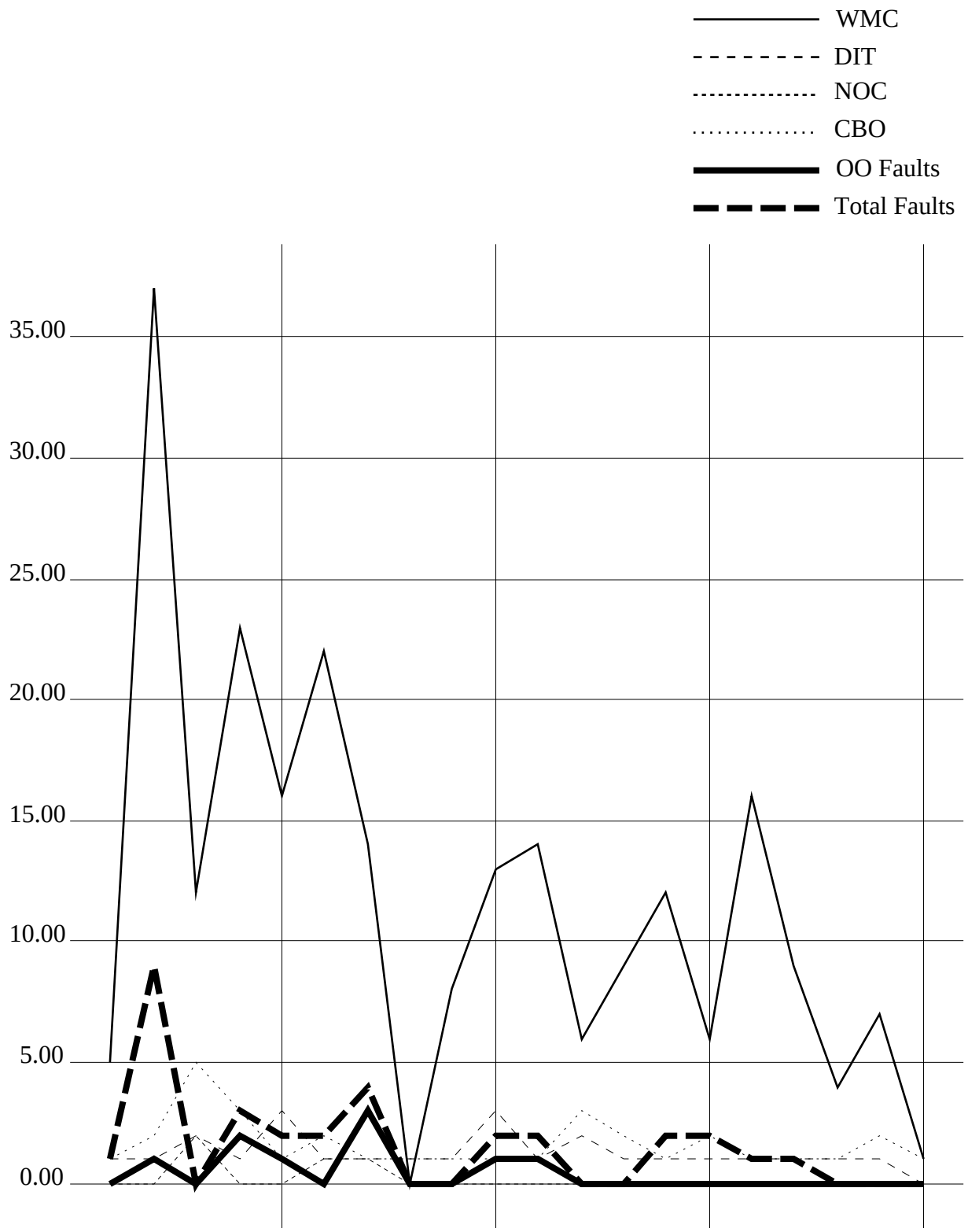


Figure 5: Summary of the CK metrics for System A.

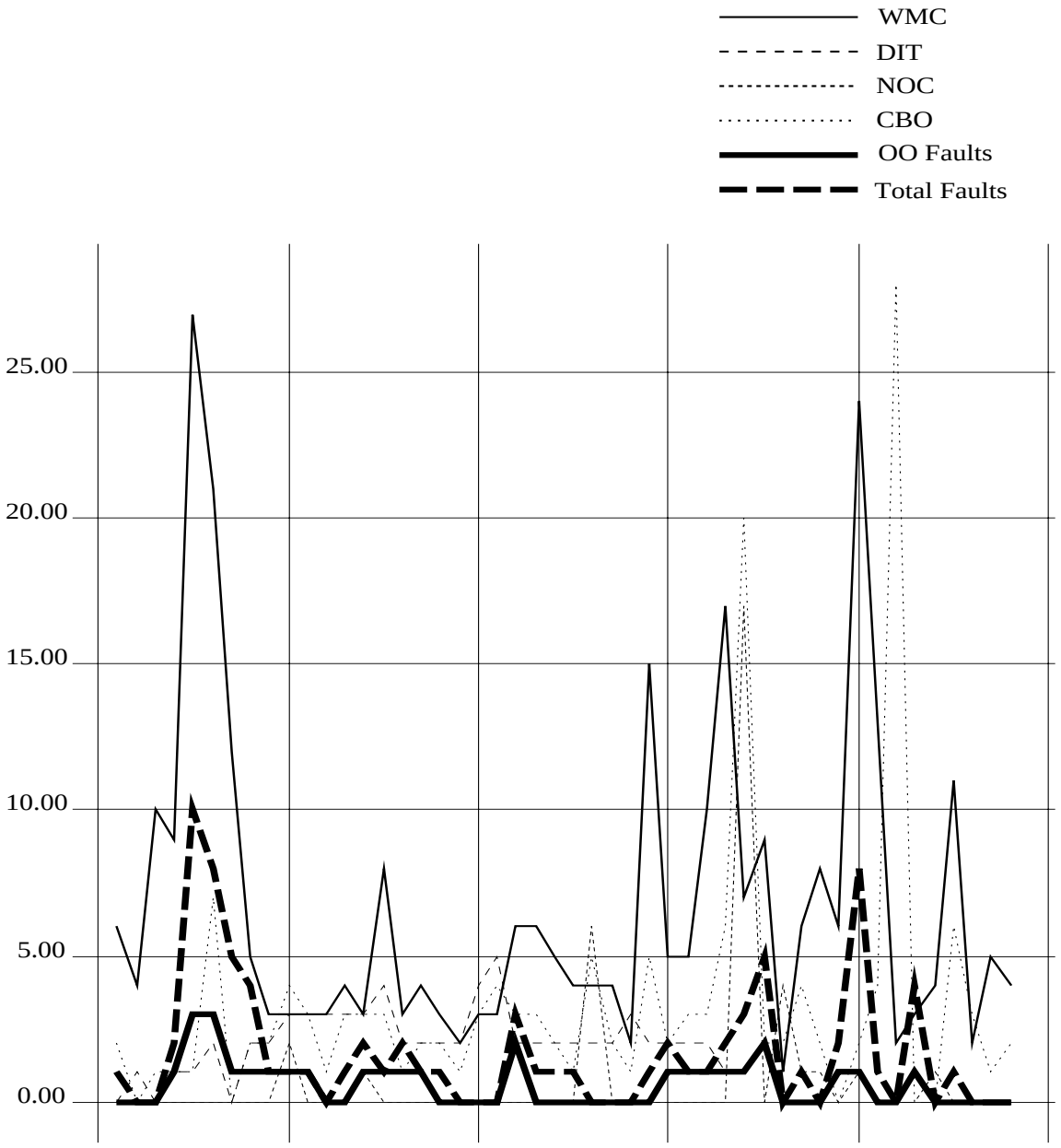


Figure 6: Summary of the CK metrics for System B.

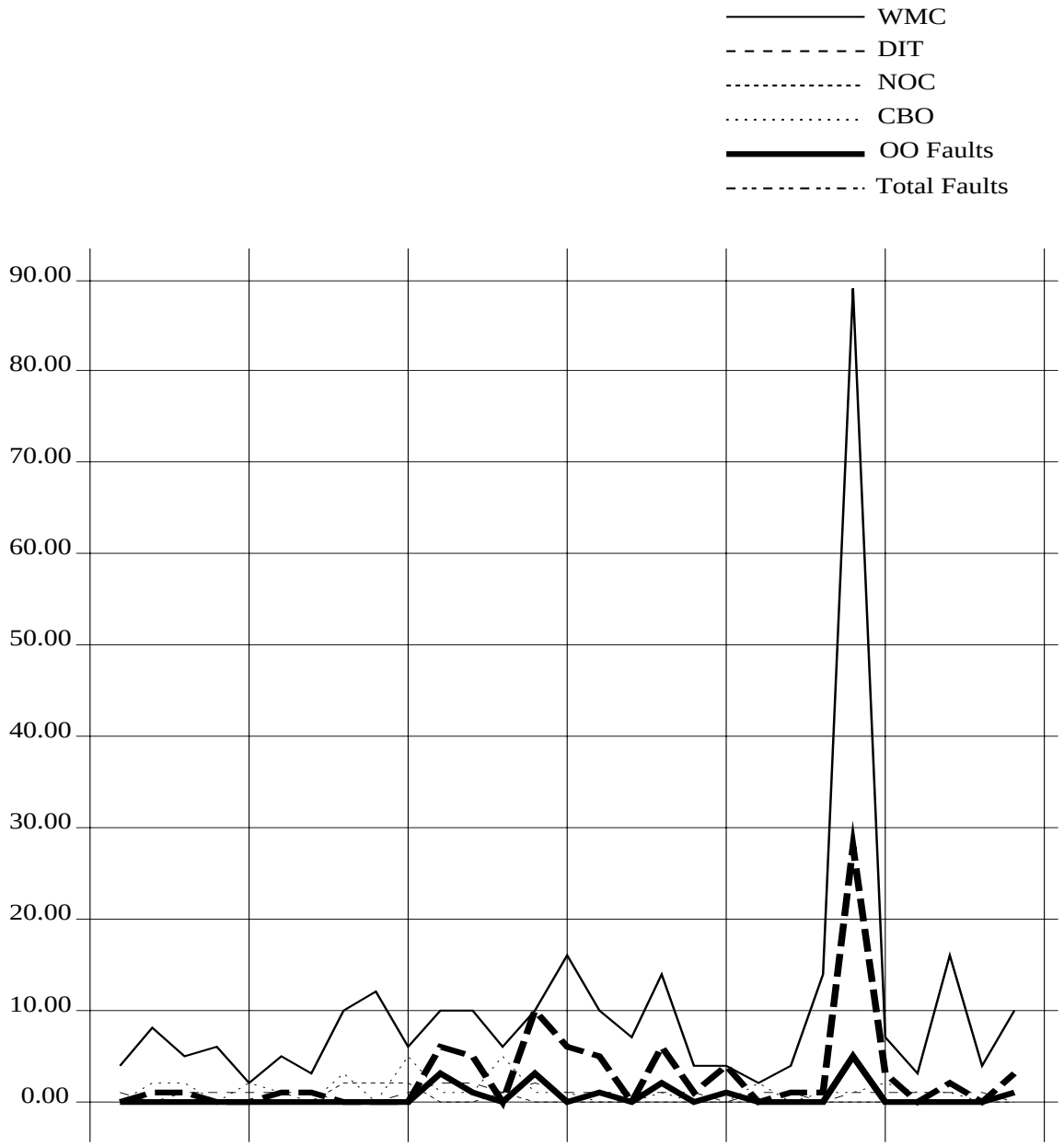


Figure 7: Summary of the CK metrics for System C.

method uses any data which is defined/modified by another method. The motivation behind this factor is that when a data member, which is used by the inherited methods, is modified by the new or redefined method, new faults tend to be introduced into the inherited methods.

For *type II* faults, we also can not relate any of the CK metrics to the distribution of this type of fault. This observation can be explained by considering the causes of the type II faults. The *type II* faults are the object management faults which might be caused by dangling object reference and object copying. We observe that a class with more object/memory allocating activities tends to introduce more *type II* faults. Therefore, the number of object/memory allocating statements within a class should be taken into account. Furthermore, we feel that a complicated copy/assign operator of a class tends to introduce more *type II* faults than the default or simple one. Therefore, the complexity of the copy/assign operator implementation of a class should be also considered.

To summarize what we observed, we think that the existing CK metrics are not sufficient to identify the fault-prone classes with OO faults. For this purpose, other metrics are needed which take into account (1)the dynamic behavior of the program; (2) the scenarios that the instances of the classes are referenced in the program.

6 Conclusions and Future Work

We have shown what types of faults are likely to be encountered in object-oriented programs. The results of our empirical study suggest that traditional testing techniques, such as functional testing and white-box approaches that utilize the statement and decision coverage criteria, are not adequate for detecting object-oriented faults. Although the state-based testing performs better than the code-based strategy, the results are still insignificant, particularly, in detecting OO faults. Therefore, to have confidence in testing object-oriented programs, there is a need to develop an object-oriented testing technique that takes object-oriented features into account and is reliable with respect to OO faults. Moreover, we applied four CK metrics on the three industrial systems and we observed that three out of the four metrics we selected can not be used as fault-prone indicators for our systems.

Much work remains to be done in this research area. We are currently investigating new design metrics which can be applied to identify OO fault-prone modules in object-oriented programs. It is expected that such metrics can be used to select and prioritize effective testing techniques, so that most faults can be detected with an affordable cost.

References

[1] Victor R. Basili, Lionel C. Briand, and Walcilio L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, October 1996.

[2] R. V. Binder. The free approach to object-oriented testing - an overview. In *http://www.rbsc.com/pages/Free.html*.

[3] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[4] J. Gao, D. Kung, P. Hsia, Y. Toyoshima, and C. Chen. Object state testing for object-oriented programs. In *Proceedings of the 19th Annual International Computer Software & Applicaitons Conference*, 1995.

[5] D. M. Hoffman and P. A. Strooper. A case study in class testing. In *CASCON'93*, 1993.

[6] D. M. Hoffman and P. A. Strooper. The testgraphs methodology: automated testing of collection classes. *Journal of Object-Oriented Programming*, 8(7):35–41, 1995.

[7] D. Kung, N. Suchak, J. Gao, P. Hsia, and Y. Toyoshima. On object state testing. In *Proceedings of the 17th Annual International Computer Software & Applicaitons Conference*, 1993.

[8] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New York, 1988.

[9] S. Siegel. *Object-Oriented Software Testing - A Hierarchical Approach*. John Wiley & Sons, Inc., New York, 1996.

[10] IEEE Computer Society. Ieee 1044 - standard classification for software errors, faults and failures. *IEEE Computer Society*, 1994.

[11] C. D. Turner and D. J. Robson. The state-based testing of object-oriented programs. In *Proceedings of the Conference on Software Maintenance*, pages 302–310, 1993.

[12] B. F. Webster. *Pitfalls of Object-Oriented Development*. M&T Books, New York, 1995.

An Industrial Case Study of Quantitative Management for Object Oriented Software Testing

Glen Xia, Deloitte Consulting

May, 1999

1

Deloitte Consulting

Content

- Company & Project Background
- Implementation Process
- Test Planning
  -
  - Estimating defects numbers
  - The novel approach
- Test Monitoring, Tracking
- Defects Causal Analysis
- Process Improvement Initiatives

Summary and Conclusions

2

Deloitte Consulting

Company & Project Background

- Large Australian insurance and financial

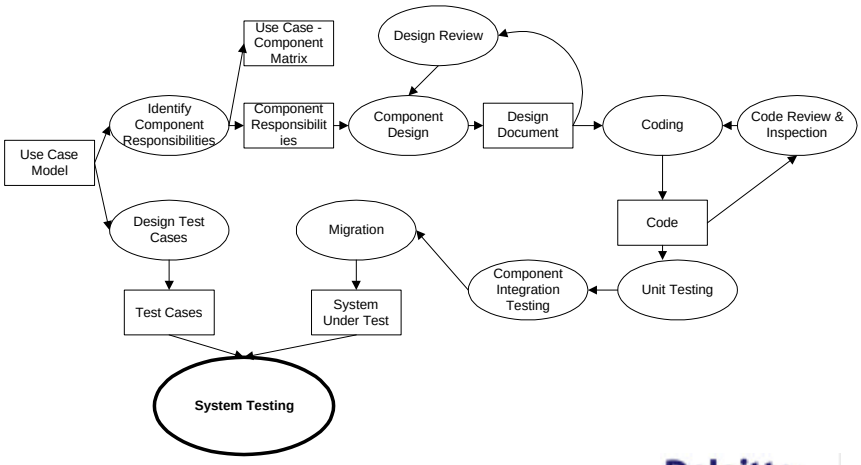
The project was to replace the legacy products administration systems
OO Insurance Framework was purchased
Project consists of 10 releases
The information reported here is from one release

50-75 people was on the project
About 30K man hours were consumed
45K SLOC were modified and added

3

Deloitte Consulting

The Implementation Process



4

Deloitte Consulting

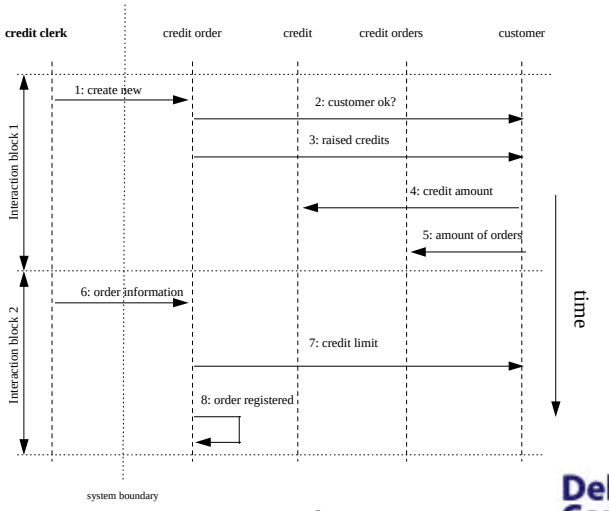
The Implementation Process - Use Cases Components Matrix

Use case/Components	C001	C002	C003	C004
UC1000		x	x	
UC1001			x	x
UC2000	x			
UC2001		x		
UC2002			x	
UC2003			x	x
UC2005		x	x	x
UC3001			x	
UC3002			x	x

5

Deloitte Consulting

The Implementation Process - Interaction



6

Deloitte Consulting

Test Planning - Estimating Testing Effort

Using high level effort distribution among major development phases (Top down)

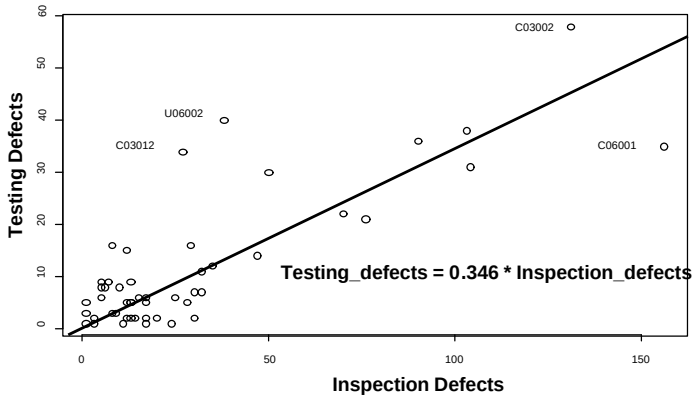
Estimating the number of test cases required

Use Case Type	Estimated Number of Test Case Required
Simple	10
Medium	14
Complex	20

7

Deloitte Consulting

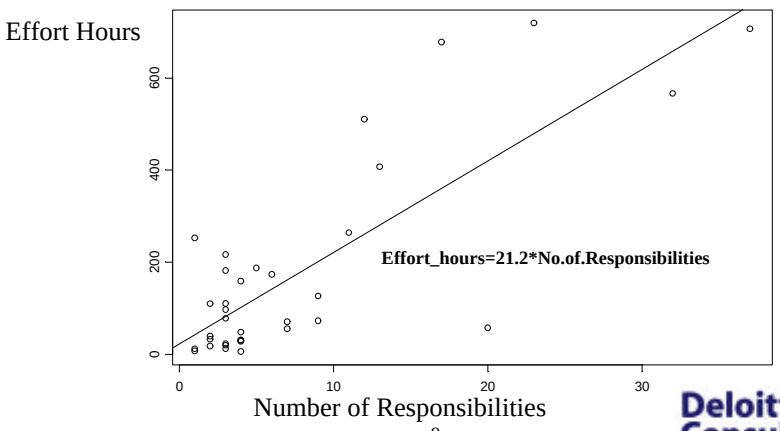
Test Planning - Estimating Defects Numbers



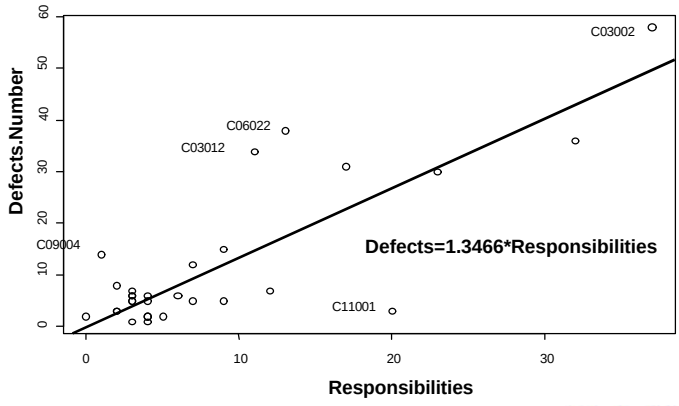
8

Deloitte Consulting

New Approach - Estimating Development

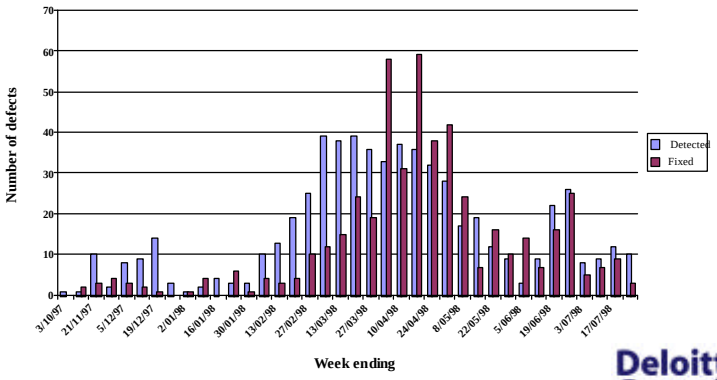


9



10

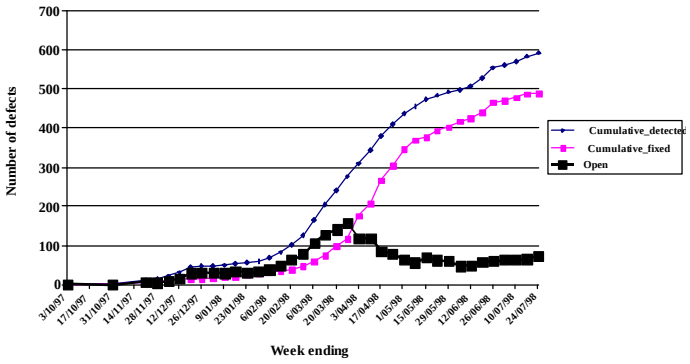
Defects Monitoring, Tracking and Status



11

Deloitte Consulting

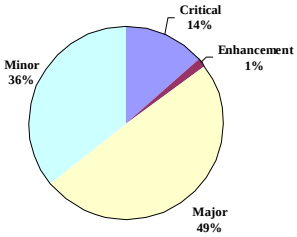
Defects Monitoring, Tracking and Status



12

Deloitte Consulting

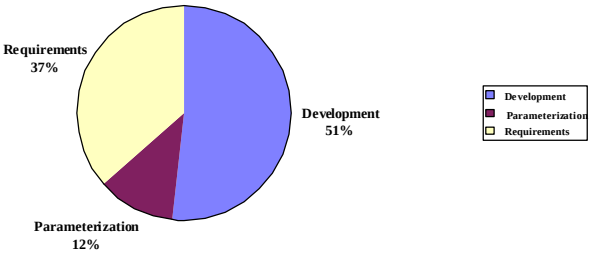
Defects Monitoring, Tracking and Status



13

Deloitte Consulting

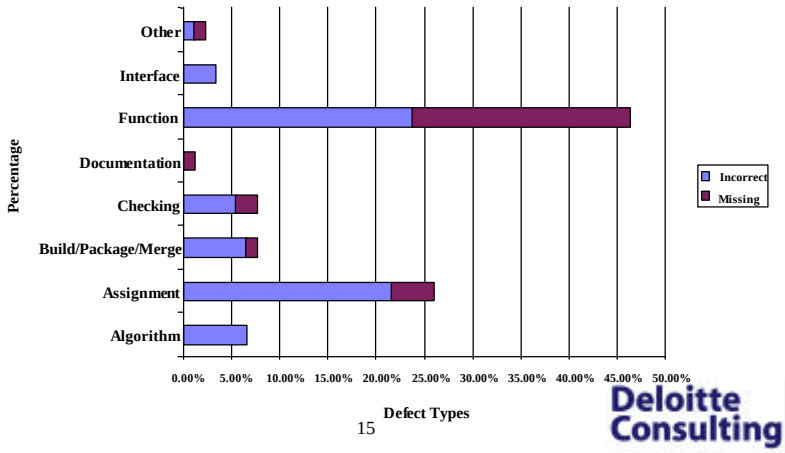
Defects Causal Analysis-Where Defects Were



14

Deloitte Consulting

Defects Causal Analysis - Distribution among



15

Deloitte Consulting

- The functional defects are dominant and most defects were introduced in the development phases. The possible causes are as follows:
  - The requirements are not clear, not detailed enough

    Lacking of communication between developers and requirements modellers

    Poor design and ineffective design review processes

    Inadequate and ineffective functional unit testing and component integration testing

16

Deloitte Consulting

17

Deloitte Consulting

- Allocate more time to enable unit testing and component integration testing

  Testers were required to be involved in component integration testing around use cases by providing use case test cases and to be responsible for

- Quantitative tracking the effectiveness of development and verification processes by using

Deloitte Consulting

Summary and Conclusions

- Use case driven approach makes it possible to estimate project effort and defects number accurately and reliably

  Use case driven approach makes project progress tracking simpler and more effective

  The approach developed in estimating project effort and defects number using the number of added responsibility is found to be accurate and easy to use, however external validations are

19

Deloitte Consulting

Summary and Conclusions

- The analysis and visualization techniques presented here are pragmatic but useful

  Orthogonal defects classification schema is practically useful

  Quantitative test management is not only essential but is also possible

20

Deloitte Consulting

An Industrial Case Study of Quantitative Management for Object Oriented Software Testing

Glen Xia
Deloitte Consulting
225 George Street, Sydney, Australia 2000
Email: glen_xia@deloitte.com.au

Abstract

In this paper, we will report our experience in managing the testing aspect of a large object oriented software development project through quantitative analysis. The novel approach, we developed in estimating development effort and the number of defects will be presented. We will demonstrate the analysis and visualization techniques used in tracking, monitoring and controlling the testing process. Finally, the process improvement initiatives, based on the Orthogonal Defects Classification (ODC), will also be reported.

1. Introduction

Software testing has becoming a mainstream software development and maintainenece activity in recent years. Effectively managing the testing aspect of a project is essential for any software project to be successful, as testing related activities normally consume 35-50% of the project budget [2]. The proportion of testing cost is growing with the rapid increase of utilizing components based software development approach, implementing package based enterprise solutions and rapid application development. There is a stronger than ever requirement for software testing methodologies, testing processes, formalized test planning, tracking and monitoring mechanisms to effectively control the process and to achieve cost effectiveness.

Object oriented technology has been perceived by many people as the better way, than the conventional development approach to achieving higher reusability, higher development productivity, better product maintainability and quality. It may still be debatable if the perception is true or not. Nevertheless, it is certain that object oriented technology is becoming the major software development paradigm, and as a software testing professional, we will be required to test more and more object oriented software systems. While there are a number of inherent difficulties associated with testing object-oriented systems as reported by many authors [3,4], we found that object oriented development process does lend us useful mechanisms for effectively estimating and planing testing effort, tracking, monitoring and controlling testing processes.

In this paper, we will report our experience obtained in managing the testing aspect of a large object oriented software implementation project. We will introduce our approach in estimating and planning testing resource requirement and the method we developed in estimating the number of defects at different stages of software development processes. The measures and visulization techniques we used in tracking, monitoring and controlling the testing processes will also be presented. Finally we will report our experience in devising and driving process improvement initiatives by utilizing defects causal analysis techniques and Orthogonal Defects Classification (ODC) scheme [1]. The paper will be concluded by summarizing the important

lessons learned from the project and proposing plans to improve our capability in managing object oriented testing.

2. Background

2.1 The Company

The company is a large insurance and financial service organization operating in Australia. The key markets in which the company operates are reinsurance, corporate insurance, personal general insurance, business insurance, accident compensation, collective investments, superannuation, life insurance, savings, and loans and wholesale funds management. It has been providing these services using a variety of disparate software systems which had a high maintenance cost, were inflexible and had a high product time-to-market. In order to maintain and increase market share in an increasingly competitive market, the company decided to replace the systems using up to date technology. An object-oriented insurance software framework (workbench) was purchased. However, substantial work was required to develop new components, to modify and enhance the existing ones, in order to tailor the system to align with the Company business practices, the Australian insurance market, and Australian legislative compliance.

2.2 The Project

The project consisted of four main teams—requirements, development, testing and deployment. The total team size was between 50 -70 members at any time during the implementation phase. An Object-Oriented Development Process was followed and several leading OO consultants were brought in to set up the process, to train the project managers and the team members.

2.3 The Development Process

Overall, the project followed the Use Case Driven and Architecture-centric, iterative and incremental approach. It includes Requirements Modelling, Design, Implementation, and System testing. Figure 1 is an overview of the software development processes we followed.

1. **Requirements Modelling**: It involves eliciting and documenting requirements and requirements walkthroughs. The deliverables produced in this phase include a set of use cases and specifications for user interfaces and reports. For detailed description, please refer to [5].

2. **Design**: The design phase starts with translating use cases into responsibilities of each component of the system. The deliverables of design phase is the use case and component matrix with responsibilities identified and a set of interaction diagrams (one use case has at least one interaction diagram associated with it). Table 1 and Figure 2 provide examples of a use case component matrix and an interaction diagram.

3. **Implementation**: Implementation phase involves detailed design, design review, coding, code review and inspections, unit testing and component integration testing.

4. **System Testing**: In this phase, scenario testing and full system integration testing are carried out.

Figure 1: Overall Software Development Processes

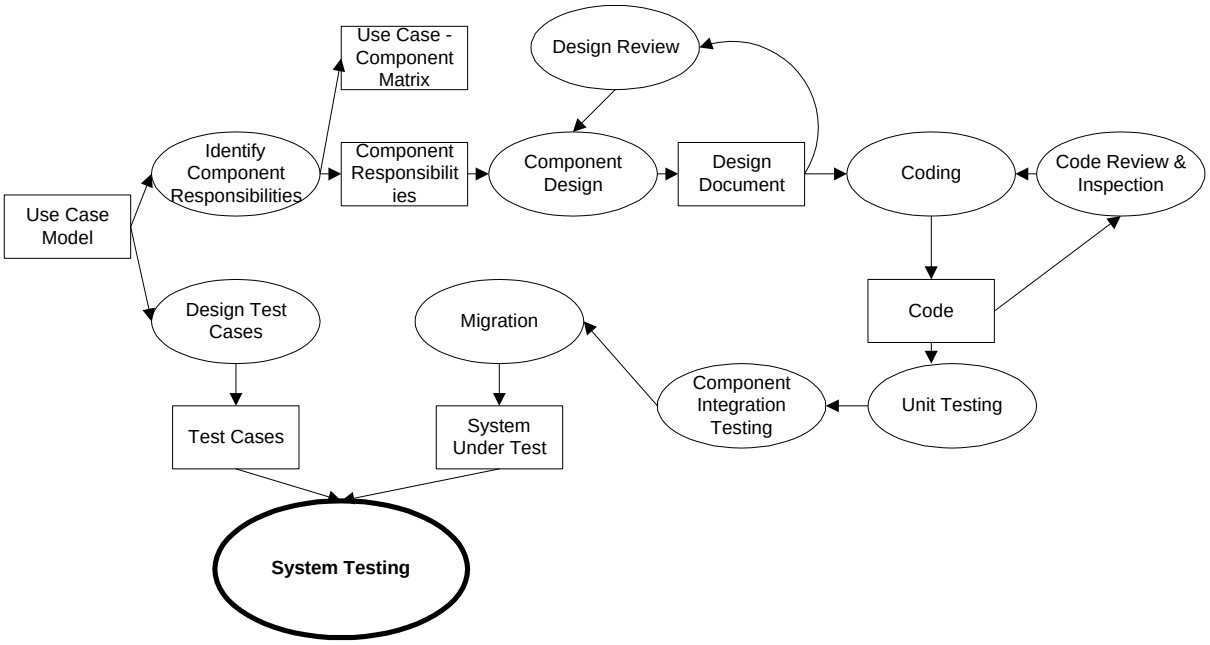
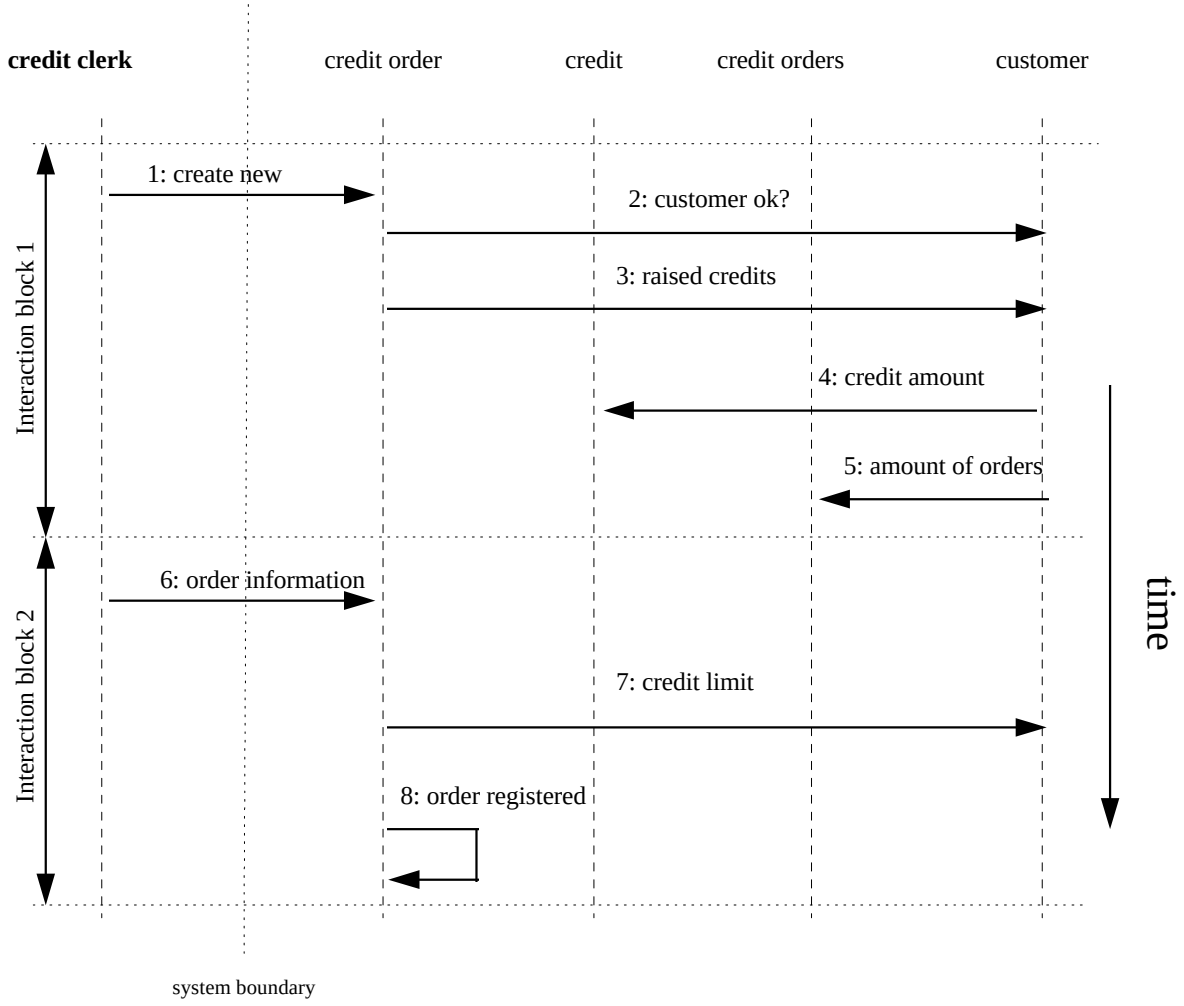


Table 1 : Example of Use Case and Compnent Matrix

Use case/Components	C001	C002	C003	C004
UC1000		x	x	
UC1001			x	x
UC2000	x			
UC2001		x		
UC2002			x	
UC2003			x	x
UC2005		x	x	x
UC3001			x	
UC3002			x	x

Figure 2: Example of Interaction Diagram



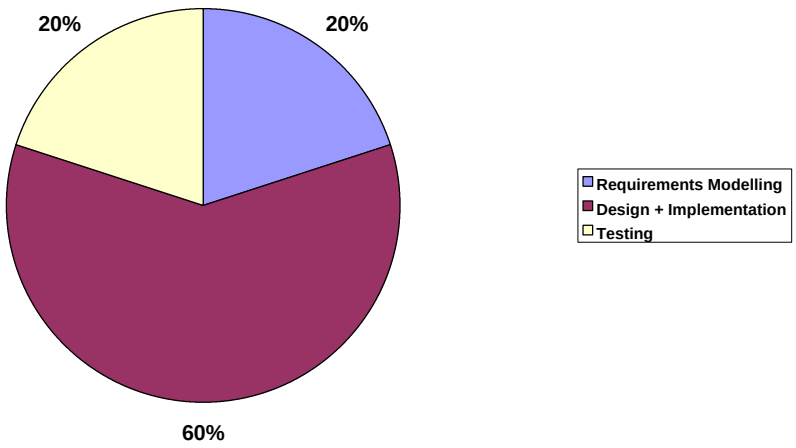
3. Test Planning

Test planning is critical for any software project to be successful [2]. Without plan, we do not know what we are going to do, how we are going to do it, how long it will take, how many testers required and how much it is going to cost us. However, to have a realistic and achievable plan is still very challenging for most of us. This is mainly due to the fact that testing effort and the number of defects to be detected are difficult to be estimated accurately and reliably, as software testing is still an immature software engineering discipline, no codified knowledge is available. Collaborative effort within the software testing community is required to establish and validate the techniques and to share experiences obtained. Test planning involves estimating testing effort required and the number of defects to be detected. The former is useful for staffing purpose and the latter is important to determine when it is ready to stop testing.

3.1 Estimating testing effort

We estimate our testing effort using both top down and bottom up approach. The top down approach is based on the high-level effort distribution among major development phases. In our project, the following distribution as illustrated in Figure 3 was identified.

Figure 3: Effort Distribution among Major Development Phases



The bottom up approach is based on the estimate of the number of test cases required and the average time required to design, executes a test case. For our project, the number of test cases is estimated using the following parameters as shown in Table 2 and 2 hours is used as an average effort for designing and executing a test case.

Table 2 : Estimating number of test cases required

Use Case Type	Estimated Number of Test Case Required
Simple	10
Medium	14
Complex	20

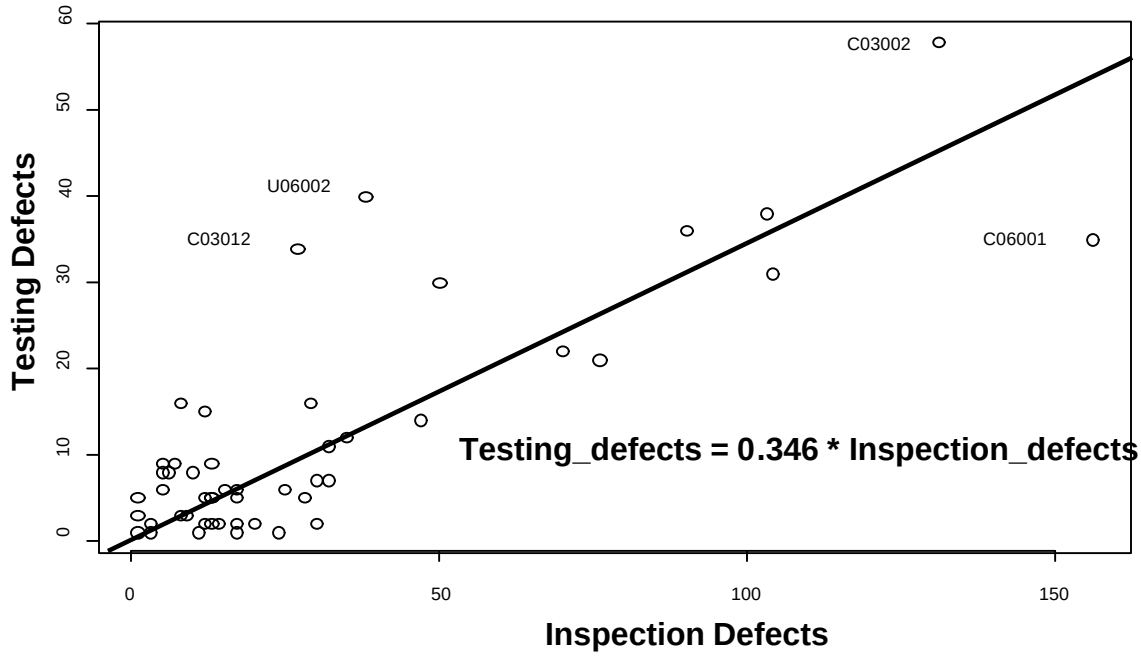
3.2 Estimating the number of defects

The number of defects was estimated using the following two approaches:

1. Using the estimated defects rate and the number of Source Lines of Codes (SLOC) modified and added to the system

2. Using the number of defects captured in design reviews and code inspections as the basis for estimating the number of defects to be detected in system testing phases. Figure 4 demonstrates the relation we used.

Figure 4: Relationship between the Number of Defects Captured in Reviews and Inspections and the Number of Defects Found in System Testing



It is clear that the number of defects to be found in system testing can be estimated by the number of defects captured in reviews and inspection. For our project, for every three defects found in reviews and inspections, we expect one defect will be found in system testing. Please note that from process improvement perspective, we expect the line will be flatter each time, that is: we expect more defects are found in reviews and inspections, and less will be found in system testing phase. On average, it only took 0.5-1 hour to fix a defect found in reviews and inspections, whereas, it took 15 hours for fixing that found in system testing stage.

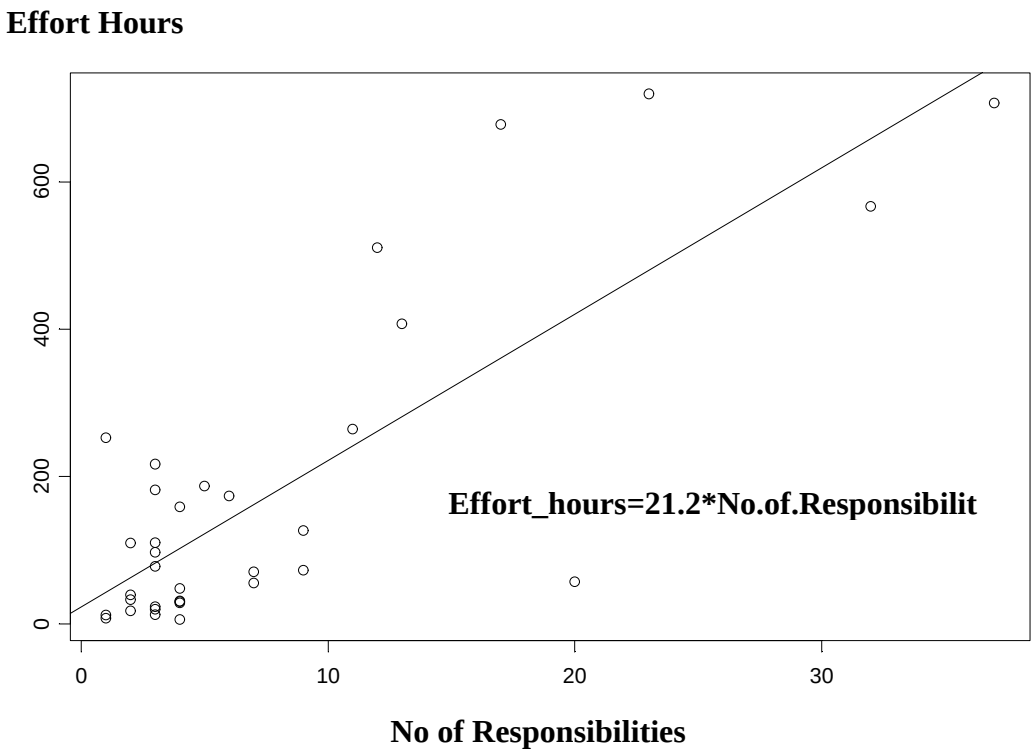
3.3 The novel approach

Both Line of Code (LOC) and Function Points Analysis (FPA) have long been used as predictors for estimating development effort and defects number, however with limited success. Although, it is still arguable if they can be used in OO context, we found some inherent difficulties associated with these two methods. Estimation based on LOC requires the LOC as the input, however, in reality this will only be available at the late stage of the project, where estimation is not so critical. For methods based on FPA, FPA counting is very subjective and not so visible therefore project tracking based on FPA is very difficult. For use case driven approach, we found the number of responsibilities identified in the design stage is a good predictor for development effort and for the number of defects to be found in system testing stage. It also makes project progress tracking very easy by simply comparing the actual number of responsibilities implemented and tested versus the planned ones. However, it is important to be aware that to maintain the uniform granularity is important for this measure to be useful.

3.3.1 Predicting component development effort

Strong correlation has been found between the number of responsibilities assigned to the component and the development effort required. The prediction accuracy is found to be better than that of using the LOC as the predictor. The accuracy can be improved when the number of collaborators is also considered, alone with the number of added responsibilities. The collaborator is the component, which provides assistance for the component under consideration to fulfil the responsibilities assigned.

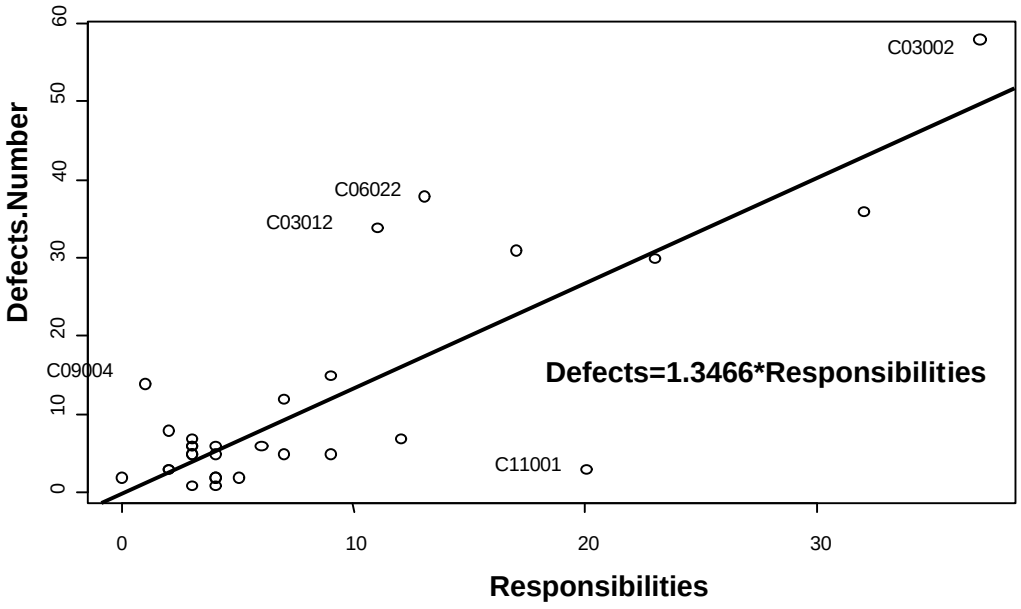
Figure 5: Component Development Effort Prediction Using the Number of Added Responsibilities



3.3.2 Predicting the number of defects associated with each component

The number of defects can be predicted using the number of responsibilities associated with the components. For most components, the prediction tends to be reasonably accurate. However, some outliers are also found. Further investigation needs to be carried out in terms of what it is the case. For example for component C06002 and C03012 illustrated in Figure 6, these are the two components which subjected many changes and many developers have actually touched them. It indicates that proper design is needed to minimise the changes required. For component C11001, it is a utility component and all the responsibilities are very simple.

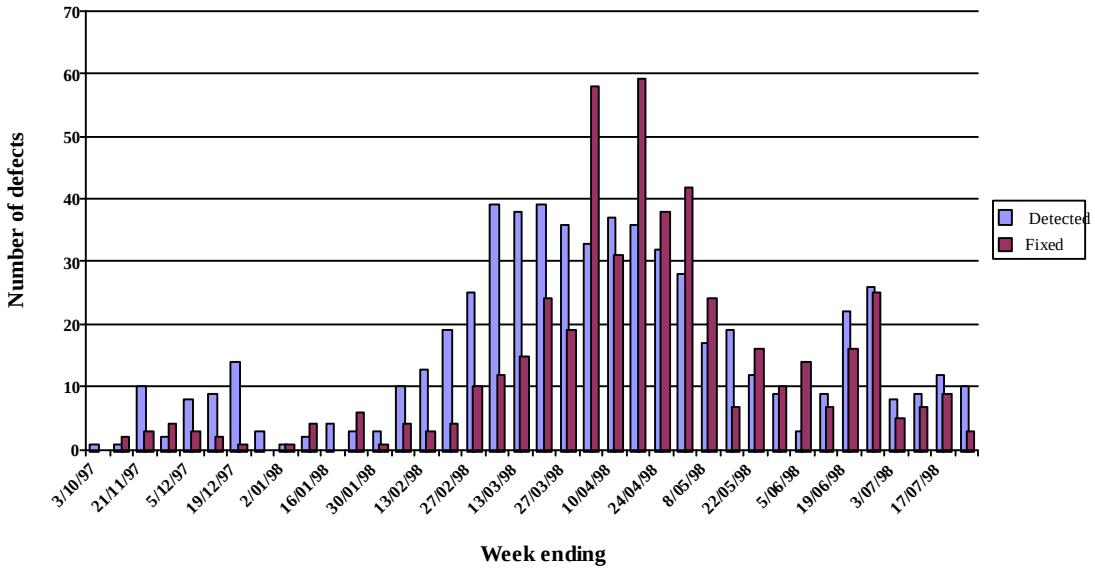
Figure 6: Predicting Number of Defects Using the Number of Added Responsibilities Assigned to the Component



4. Defects Monitoring, Tracking and Status Reporting

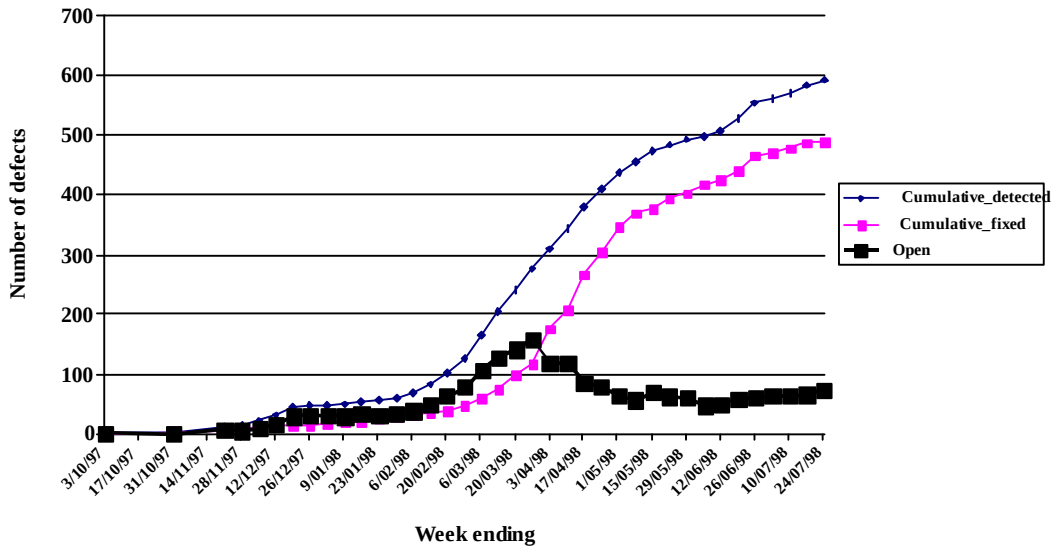
In this section, we present the visualization techniques we employed to track testing progress and defects status.

Figure 7: Weekly Number Defects Detected versus Fixed



The number of defects and their status are monitored on regular basis, therefore, test progress can be tracked and the status of testing can be evaluated.

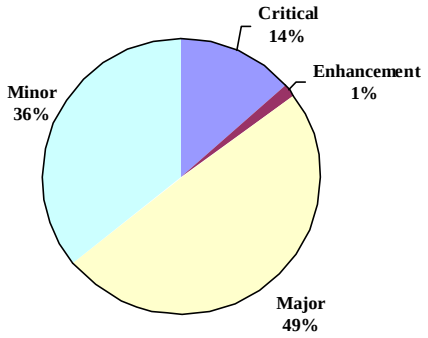
Figure 8 : Accumated Defects Detected, Fixed Over Time



The accumulated defects can be used to analyse where the test progress is, how many more defects need to be detected and fixed, before the system can go on live with confidence. The release decision was made on the basis of the estimated defect number and the defects trend analysis.

4.1 Distribution of defects in each severity category

Figure 9: Defects Distribution in Severity



The severity of a defect was classified based on the assessment of potential impact it may cause on business functionality. The majority of defects detected in system testing stage are either

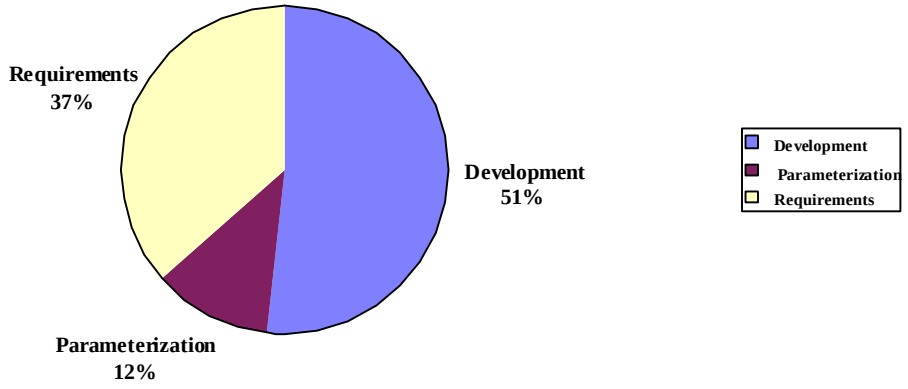
Critical or Major in nature. Although, there is no clear correlation between the severity of the defect and the effort required fixing it, it is important to track them as it was one of the acceptance criteria that all critical and major defects need to be fixed before the system can be accepted.

5. Defects Causal Analysis and Process Improvement

As demonstrated in Figure 8, we captured 600 defects in the system testing stage for one particular release under investigation. The rework still consumed 30% of whole development effort, although we invested substantial time in design reviews and code inspections (12% of development effort). There were certainly needs to improve the development process to minimise the introduction of defects in the first place, and to maximise early defects detection effectiveness. To improve the process, we need to understand where we have problem, such that we can design mechanisms to improve it.

Figure 10 shows the source of defects. The majority of the defects were actually introduced in the development (design and implementation) phase.

Figure 10: Where Were Defects Introduced?



To understand why the defects were introduced and why they were not captured in the development phase, we analysed the nature of the defects. The Orthogonal Defects Classification (ODC) schema was used to assist us to understand the nature of the defects, in order to take right corrective actions. Figure 11 and Figure 12 below demonstrate the distribution of defects in each defect type. It is clear that incorrect and missing functionalities are the major defect types.

Figure 11: Defects Type Profile

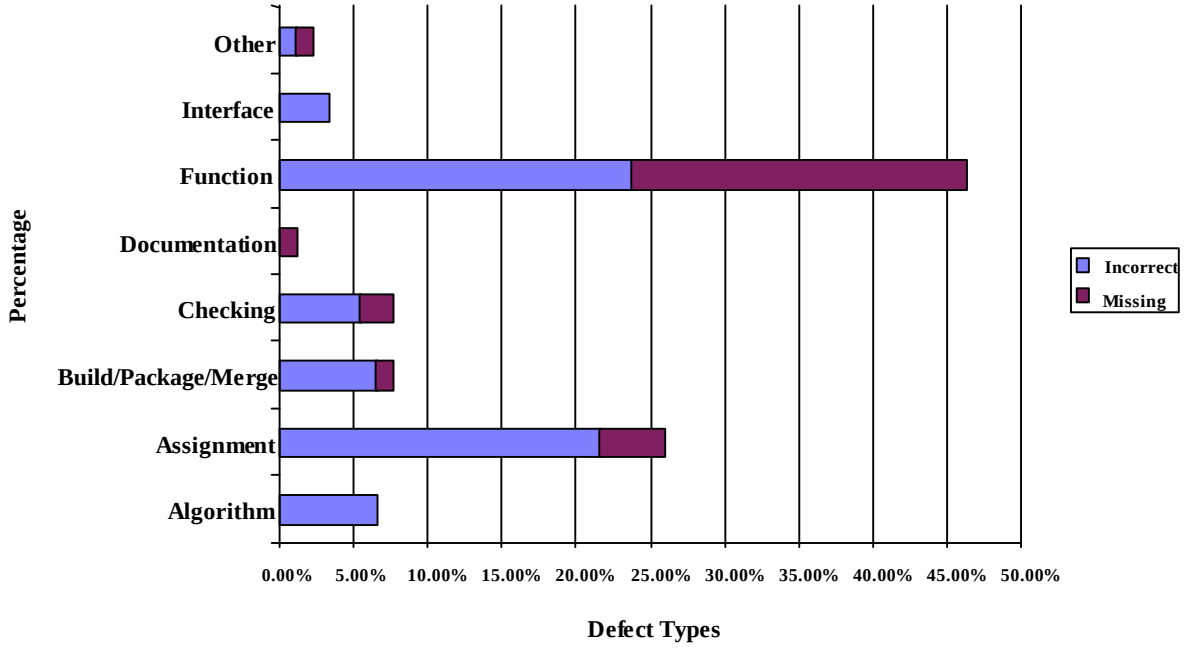
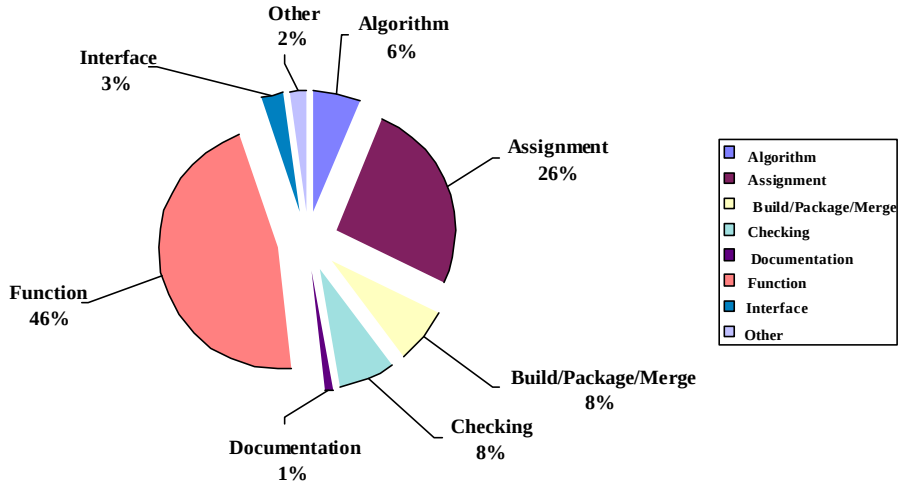


Figure 12 : Distribution of Defects in Each Defect Type



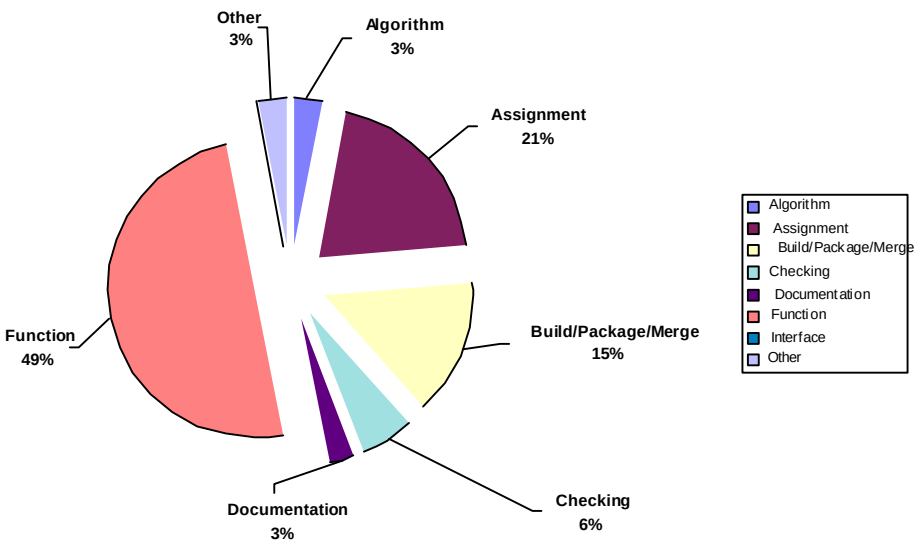
5.1 Defects Causal Analysis

5.1.1 Requirements modelling

As analysed above, 37% of defects were injected from requirements modelling phases. Most of them are associated with either missing or incorrect functions. There are substantial number of

defects are assignment type in nature, especially those associated with dates (effective date, etc), which reflect that the requirements were not detailed or clear enough for implementation. Configuration management was poor and circumstances such as how to handle abnormal input (error messages, etc) were not specified in requirements. Details for calculation were also not given in some circumstances, which made it difficult to implement and test.

Figure 13: Distribution of Defects Injected from Requirements Modelling Phase



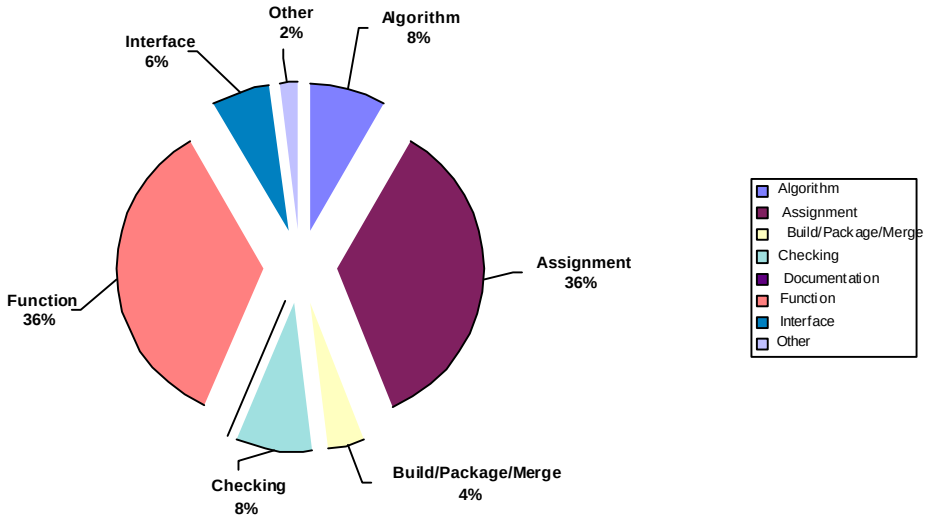
5.1.2 Development

The dominant type of defects is function type (incorrect). This was mainly due to:

1. Poor requirements quality
2. Poor design process
3. Ineffective design review
4. Inadequate and ineffective component integration testing

Substantial number of defects is associated with assigning wrong number to certain parameters. This reflects the fact that the developers interpret requirements themselves rather than checking with requirement modellers. This also reflects the fact that the design review was not as effective as we expected, due to inadequate number of people who were familiar with the system. The substantial number of checking type of errors point to the areas of unit testing, where the developers have to do some negative testings. The most important problem identified is inadequate functional unit or component integration testing, where most incorrect or missed functionalities should be detected.

Figure 14: Distribution of Defects Injected from Development Phases

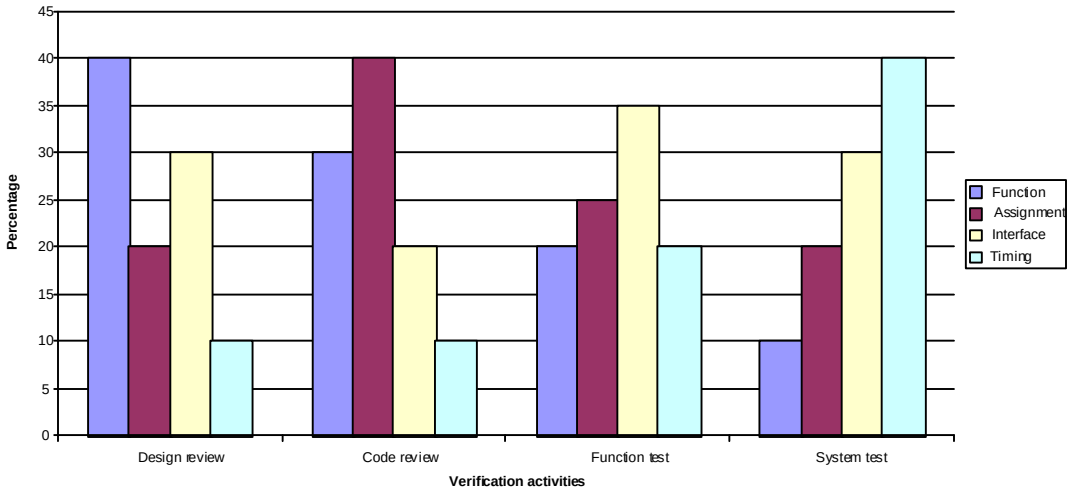


5.2 Process Improvement Initiatives

In order to reduce the number of defects escaped from the development phase, the following process improvement initiatives were taken:

1. Allocate more time for developers to conduct functional unit testing and component integration testing. For each use case, a use case owner was assigned to be responsible for carrying out functional testing whenever the code is ready to do it. The system testers were required to work closely with the developers by providing use case test cases and to be responsible for signing off use case testing.
2. Improve communications between requirement modellers, developers and testers. For each use case, a contacting person in each team was appointed and "use case" teams were formed. These who work in the same use case team will be involved in reviews at different phases of the project.
3. Track the effectiveness of development and verification processes by capturing defects information, such as: where the defects injected and captured the defect type and defect mode. The objective is to monitor and track the defects pattern, such that timely feedback can be provided. For more information, please refer to [1].

Figure 15: Expected Change of Defects Patterns over Time



6. Conclusions

In this paper, we reported our experience in managing the testing aspect of a large object oriented software development project through quantitative analysis. We found that the use case driven approach lend us opportunities to estimate, track, monitor and control the process effectively. The novel approach, we developed in estimating development effort and the number of defects is easy to use. However, theoretical investigation and external validations from other sites are required. We also reported the analysis and visualization techniques used, which are parametric and are found to be very useful. The defects causal analysis based on ODC are practical and easy to use. The effectiveness of the process improvement initiatives will be evaluated.

7. Acknowledgments

I would like to thank Tony Batten, David Cliff and Robert Burnelek for support and encouragement. Assistance from Stephan Meyn, Philip Haynes and Alex Jouravlev is also acknowledged.

8. References

1. R. Chillarege (1996). Orthogonal Defect Classification, in *Handbook of Software Reliability Engineering (M. R. Lyu, Editor)*, pp 359-400, McGraw-Hill.

2. E. Kit (1995). Software Testing in the Real Work –improving the process, Addison-Wssley.

3. I. Jacobson, M. Christerson. P. Jonsson and G. Overgaard (1992). *Object-Oriented Software Engineering – A Use Case Driven Approach*, Addison-Wesley.

4. S. Siegel (1996). *Object Oriented Software Testing – A Hierarchical Approach*, John Wiley & Sons, INC.

5. J. Tassopoulos and G. Xia (1998). Process Improvement to Overcome Use Case Weaknesses: An Experience Report, *OOPSLA'98 Pre-Addendum*, pp. 58 – 69, Vancouver, British Columbia, October 18 - 22, 1998.

Developing Load and Performance Requirements

For Web Sites

By
Leon A. Slota
Neoglyphics Media Corporation

Figure 1: Example Load and Performance

"Response time for the web site will be less than 0.75 seconds for simple web pages and less than 2.5 seconds for data driven pages, not counting internet delays (i.e. measured on the LAN attached to the server)."

"At 260,000 customers, each one visiting the site every other month on average for ten minutes, with everyone visiting within the same 15 hour window of time ... then we have something less than 50 customers who might be on-line at any given time. It is unlikely, then, that we will get more than a few page views per

- The maximum number of simultaneous users the site

  The maximum response times for the types of transactions or pages on the site.

  The maximum error rate allowed at peak load.

Figure 3: Development Situations.

- New Site, new market. Neither the organization nor anyone else has a site that does what the proposed site will do and the market in which it will function is

  New Site, existing market. An organization is developing a site to enter and existing market on the internet. However, it doesn't currently work in that market

  New site, existing function. An organization is replacing a current function with a

  Replacement site. An organization is replacing an existing site with a new site.

Figure 4: Sources of Information on Load

- The organization's existing business processes.

  Industry research and data.

  Competitive or comparable web sites.

  The organization's existing web site.

Table1: Sources of Information for

Information Source	New site, new market	New site existing function	New Site existing market	Replacement site
Existing business process		X	X	X
Industry research	X	X	X	X
Competitive sites			X	X
Existing site				X

Table 2: Number of Simultaneous

Not-Simultaneous	Simultaneous
2,000 per day	14
4,042 per hour	34
6,252 per hour	68
8,287 per hour	89

Table 3: Average Internet Response Time, March 15 to

(source Keynote.com).

5 Fastest Sites	Overall Average	Slowest site
3.75 seconds	8.00 seconds	21.42 seconds

Figure 5: Weekly Performance from Keynote.com.

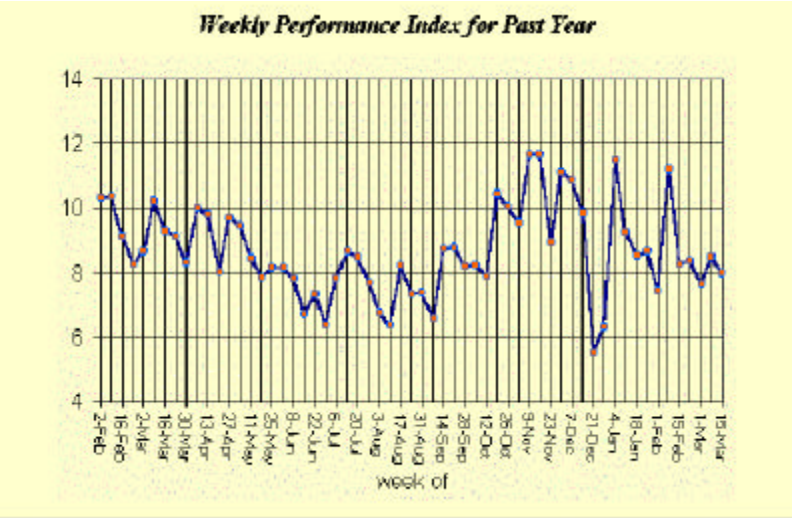


Table 4: Rate of Web Page Download Failures

	% of Failures	% of Time Outs	Total failures
Best in Class	1.31	1.92	3.23
Industry Average	2.15	3.03	5.18

Figure 6: Sample from Access Log.

```
198.69.241.2 [26/Jan/1999:07:25:05 0500] "POST /servlet/com.servlet.SearchServlet HTTP/1.0" 302 221
198.69.241.2 [26/Jan/1999:07:25:25 0500] "POST /servlet/com.servlet.SearchServlet HTTP/1.0" 302 221
198.69.241.2 [26/Jan/1999:07:25:28 0500] "GET /catalogs/searchresults.jhtml HTTP/1.0" 200 22786
141.185.72.145 [26/Jan/1999:07:25:38 0500] "GET / HTTP/1.0" 302 205
141.185.72.145 [26/Jan/1999:07:25:38 0500] "GET /splash.jhtml HTTP/1.0" 200 15526
198.69.241.2 [26/Jan/1999:07:25:51 0500] "GET /catalogs/acrosgroup.jhtml HTTP/1.0" 200 18179
```

Table 5: Users Per Second.

Time	Users
8:30:19	89
8:19:50	84
8:54:49	84
8:22:15	83
8:20:25	81
8:36:44	80
8:23:20	79
8:39:04	78
8:22:45	77
8:30:54	76
8:51:54	75
8:44:54	74
8:45:29	73
8:29:45	71
8:37:19	71
8:37:54	69
8:48:59	69
8:19:15	68
8:35:34	68
8:29:44	67

Table 6: Simultaneous Users 5 – 9 AM.

Hour	Log Records	Minimum	Maximum	Mean
5 AM	3601	2	34	14.58
6 AM	3580	3	68	24.52
7 AM	3600	12	73	30.07
8 AM	3591	4	89	35.25

Figure 7: Formula for Load Requirement.

$$\text{Future Load} = \frac{\text{Future Sales/Users} * \text{Current Load}}{\text{Current Sales/Users}}$$

Table 7: Page Hits

Page	Hits	Transaction
/	21628	home
/splash.jhtml;	15182	splash
/servlet/com.servlet.SearchServlet;	11832	search
/servlet/com.servlet.CatalogServletImpl;	11099	catalog
/catalogs/searchresults.jhtml;	9256	search
/dhtml.jhtml;	8098	home
/catalogs/fcatgroup.jhtml;	6748	catalogs
/catalogs/fcathiertabndhtml.jhtml;	6495	catalogs
/ndhtml.jhtml;	4275	home
/catalogs/fcathier.jhtml;	3847	catalogs
/catalogs/ncatgroup.jhtml;	3111	catalogs
/Login/;	2522	login
/catalogs/proxy.jhtml;	2491	catalogs
/ordering/shopping-cart.jhtml;	2487	orders
/catalogs/root.jhtml;	2438	catalogs
/servlet/com.servlet.ShoppingCartServlet;	2371	orders
/catalogs/;	2206	catalogs
/;	2060	home
/catalogs/acrosgroup.jhtml;	2007	catalogs
/servlet/com.servlet.LoginServlet;	1955	login
/ordering/rapidorder.jhtml;	1805	orders
/catalogs/chemgroup.jhtml;	1692	catalogs
/personal.jhtml;	1450	personal
/catalogs/fcat.jhtml;	1397	catalogs
/catalogs/powersearch.jhtml;	1387	search
/servlet/com.servlet.RefererServlet;	1193	referer
/Login/register.jhtml;	968	login
/servlet/com.servlet.OrderPlacementServlet;	956	orders
/ordering/;	909	orders
/servlet/UserAdmin;	871	admin

Table 8: Transaction Frequency

Transaction	Hits	% of Hits
catalogs	57490	36.43
home	36061	22.85
splash	15186	9.62
search	11832	7.50
orders	11601	7.35
login	7902	5.01
support	3728	2.36
about	2296	1.45
admin	2224	1.41
supplier	1741	1.10
personal	1450	0.92
referrer	1193	0.76
register	1157	0.73
whatsnew	1145	0.73
help	896	0.57
sitemap	804	0.51
hotlist	661	0.42
template	228	0.14
error	223	0.14
ttracker	2	0.00

Developing Load and Performance Requirements For Web Sites

By

Leon A. Slota
Neoglyphics Media Corporation

Introduction

Performance and reliability have become some of the most important considerations in web site development. Web developers and clients both know that when long waits or the site being down frustrates users, opportunities for sales and building customer loyalty are lost. And it's not just a theoretical problem; examples of sites buckling under load are easy to find. The ready availability of alternatives provided by competitors makes this even more critical in the web development world.

The way to avoid these problems is to develop a workable set of load and performance requirements. It's widely accepted that performance and load capacity must be tested. But testing is comparing what is to what should be, and the more difficult task, in our experience, has proved to be deciding how fast it should be and how many users it should be able to support so when it comes time to do the testing, we'll know if the site has passed them.

In rare cases, the client will know what their performance and load requirements are. Figure 1 is an example from an RFP.

Figure 1: Example Load and Performance Requirement

"Response time for the web site will be less than 0.75 seconds for simple web pages and less than 2.5 seconds for data driven pages, not counting internet delays (i.e. measured on the LAN attached to the server)."

"At 260,000 customers, each one visiting the site every other month on average for ten minutes, with everyone visiting within the same 15 hour window of time … then we have something less than 50 customers who might be on-line at any given time. It is unlikely, then, that we will get more than a few page views per second (say 10) at peak time."

This is about as good as it gets and there are some obvious flaws in logic here.

Typically, clients don't know how fast they want their site to work (except that they want it to be fast) or how many users they want to support (except that they want to support all of them). So we've found that it's often up to us as the experts to help them develop appropriate performance and load criteria and, of course, seek their approval for our tortured calculations.

On the broad level, much of what I have to say about the process of doing this is common sense and is probably known to many of you. There is certainly much that I don't know. This is partly because every situation is unique in some ways and defies the application of strict formulas. One of the most important things

I've learned is there are many ways to do a good job of developing load and performance criteria, and what follows are suggestions about some of the kinds of information and analysis which might be helpful in certain common situations.

Components of load and performance requirements

Although these might vary with the system being developed, I've found three minimum components of load and performance requirements listed in Figure 2 most useful.

Figure 2: Minimum Load and Performance Requirements

1. The maximum number of simultaneous users the site has to support.

2. The maximum response times for the types of transactions or pages on the site.

3. The maximum error rate allowed at peak load.

If the longest response times and highest error rates are within required tolerances at the maximum load, then they will most likely be within tolerance at lower loads. It's hard to imagine a site that's designed to perform well under the most stressful conditions not performing well under average conditions.

Although these three seem basic, you might also want to specify response times and error rates at an average load. These determine how the site will perform most of the time.

Stress requirements for running large loads over long periods of time might be useful. Tests of stress requirements are aimed at finding issues with storage, memory, and queues that might run out of space or connections over time.

Past, Present, and Future

Above, I denied any claim to universality for this paper. I will, however, claim that the basic method to employ in developing load and performance requirements is universally applicable, mostly because it's so broad and widespread in every kind of business and academic analysis. This is that the past and present are your guides to the future. The more you know about current business practices and volumes, or those of the industry the better you'll be able to predict what they will be in the future. See told you it was going to be pretty much common sense.

For the load and performance requirements of web sites, what we know about current sales, volumes, and processes; plus what we know about future goals, will tell us a lot about what volumes and performance we need to support in the future. This paper is about the kinds of information about the past and present that can be used to develop load and performance requirements in web sites. The kinds of available information will be different depending on the situation.

Before we get any further, there are a couple caveats about the accuracy of this kind of forecasting.

- Its usefulness depends in part on how similar the two things we're trying to compare are. If the plan is to turn an existing marketing site into an e-commerce site, then there's little point in using the current site's usage rates to forecast the new site's usage rates because their functions are too different. However, the volume of a traditional call center could be used to forecast volume for an Internet replacement because the functions are the same.

- Accuracy depends in part on how consistent underlying conditions are from time to time. The state of the economy obviously affects disposable income and sales. If this changes, projections are likely to be wrong. The same might be true of changes in technology. So, it's best to keep the time horizons short, perhaps inside 2 years. This doesn't seem unreasonable given that retail store displays, catalog layouts, and now web site designs change within much shorter periods of time.

In sum, the basic formula for developing load and performance requirements is:

Current or Past load/performance + future goals = forecast of future load and performance.

General Situations and Sources of Information

Depending on the situation, there will be different sources of information on the present and past. Presumably, there will always be some statement of future goals, but it might not be in the form of load and performance numbers for a web site, but in more general business terms that need to be translated. In fact, it's better that goals are stated in business terms, since whatever site is being build should serve some business purpose and have goals like every other aspect of an organization.

Figure 4 shows the three common situations in which new web site development takes place.

Figure 4: Development Situations.

1) New Site, new market. Neither the organization nor anyone else has a site that does what the proposed site will do and the market in which it will function is new to the internet. (I tried to think of a market that doesn't exist on the internet. If I could I'd be working on getting into it and not talking here). However, the point is that there will be little or no internal organizational or external market information on which to draw.

2) New Site, existing market. An organization is developing a site to enter and existing market on the internet. However, it doesn't currently work in that market either online of offline.

3) New site, existing function. An organization is replacing a current function with a web site to perform it.

4) Replacement site. An organization is replacing an existing site with a new site.

The main sources for data on current and past volume and performance are:

Figure 5: Sources of Information on Load and Performance.

1) The organization's existing business processes.

2) Industry research and data.

3) Competitive or comparable web sites.

4) The organization's existing web site.

Table 1 below details how the type of data available differs according to the situation in which the site is being constructed.

Table1: Sources of Information for Requirements

Information Source	New site, new market	New site existing function	New Site existing market	Replacement site
Existing business process		X	X	X
Industry research	X	X	X	X
Competitive sites			X	X
Existing site				X

As far as information on future goals, most organizations will have a business plan or a set of goals for the site that can be used to help forecast load and performance. This information can take the form of sales dollars, online customers, percentages of existing sales or volumes that should be transferred to the web site, or decreases in customer response time.

These are useful for measuring the success of the site at fulfilling its intended organizational function. But, they have to be translated into requirements for load and performance that can be measured on the web site by automated tools so that its readiness to meet those business goals can be tested. For load, this measure is the number of simultaneous users the site can support, and for performance the response times for html requests. Automated tools, like Mercury Interactive's LoadRunner that we use at Neoglyphics, are designed to generate such loads and measure response times.

The business plan will also suggest how important load and performance requirements are relative to other considerations. For a new site in a new market, it might be more important to get out there first and to prove the concept than to provide best in class speed and support 100,000 simultaneous users. In replacing an existing site, efficiencies in back office systems integration which reduce overhead compared to the current site might be a better investment than halving it's response time or doubling it's capacity. On the other hand, responsiveness may be an important part of a company's strategy for beating the tar out of a competitor whose site shows problems in this area. Next line, no waiting draws as much of a crowd on the net as it does at the supermarket. However, in every case, you do want to be sure that your site's performance doesn't impede its ability to serve its ultimate business purpose.

The following sections suggest how to do this in the various situations outlines above.

New Site, New Market

Developing a new market is inherently uncertain and risky, and the information that can be used to help develop load and performance requirements can't help but be uncertain too. And maybe just taking your best guess is as good as anything else in this situation.

Still, some useful information on load should be in the business plan and Internet industry research can provide some on performance. Since the market is new, there wouldn't be any directly comparable competitive sites, however, there might be some sites that perform similar functions or serve similar users where

you could go to get some comparative information. For example, a shopping cart is pretty much the same wherever it may be and whatever goes in it.

Load Requirements

As in most cases, sales or service goals can be a starting point for setting load requirements, but eventually must be expressed in terms of simultaneous users. If the sales goal is $20,000 a month and projected sales per order are $50 ($55 was average order in Shop.org study), that means 400 orders a month. Since the only 1.6% of visitors to web sites actually buy (State of Online Retailing. Shop.org), we can expect the site to have to host 2,666 visitors a month to generate the sales target. This can be broken down into visitors per week, day, or hour. The number of people who will be on the site simultaneously (at the same second) will be much smaller. The figures in Table 2 that are drawn from several sites show that the number of simultaneous users can be deceptively small. It would be nice to know if there's some standard ratio for particular sites, but that's research for another paper.

Table 2: Number of Simultaneous Users.

Not-Simultaneous	Simultaneous
2,000 per day	14
4,042 per hour	34
6,252 per hour	68
8,287 per hour	89

Performance Requirements

For performance requirements there is a good deal of industry data to draw on which is getting better all the time. This will become even more useful as better standardized measures of Internet commerce are developed.

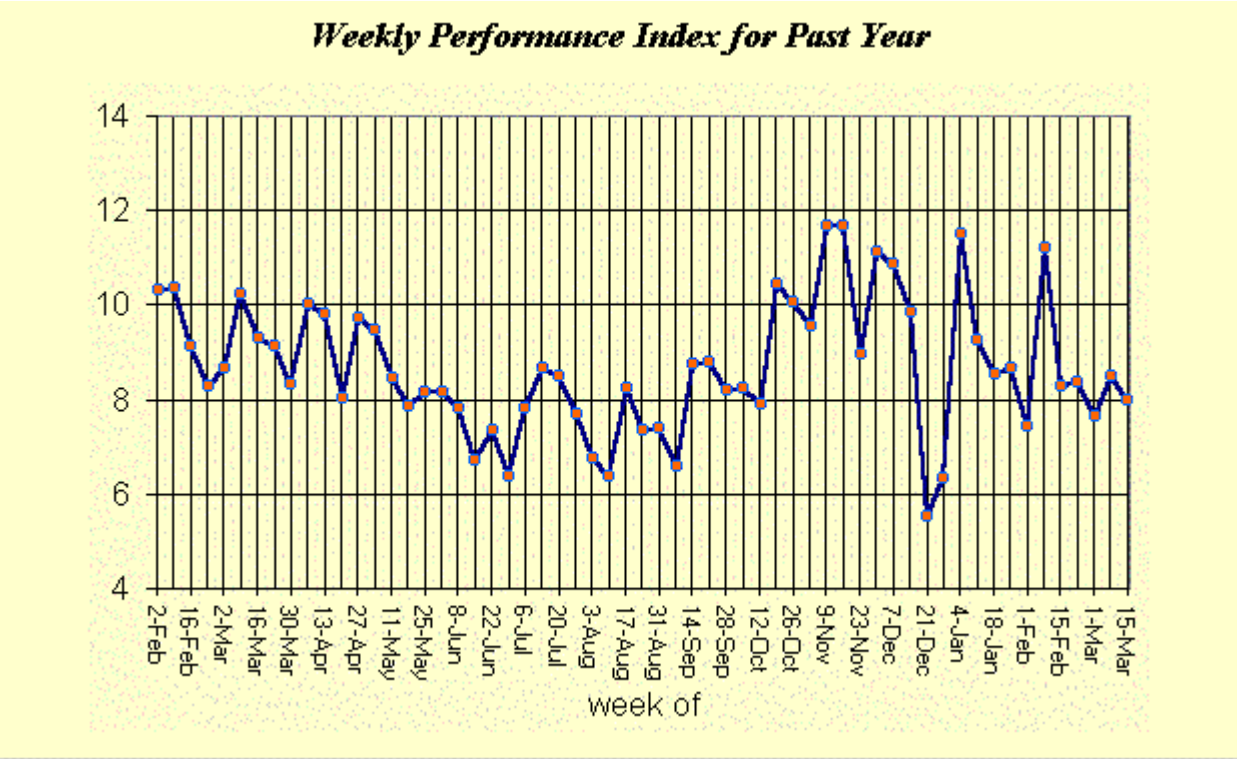
One good source for broad data on performance is Keynote (www.keynote.com). Their very interesting reports show what kind of performance is really possible on the net because one of the goals of the service is to show how Internet performance, as opposed to server performance, affects the delivery of information to end-users. Table 3 is adapted from Keynote's Business 40 Internet Performance Report for 3/15/99.

Table 3: Average Internet Response Time, March 15 to 19, 1999.

5 Fastest Sites	Overall Average	Slowest site
3.75 seconds	8.00 seconds	21.42 seconds

The fastest sites downloaded their home pages in an average of 3.75 seconds and the average download time for the entire sample of sites was 8 seconds. So So, if you wanted your homepage to download with the best in class, it would have to respond in about 3 seconds.

Figure 6: Weekly Performance from Keynote.com.



One of the problems with this kind of information is that it does include Internet overhead in the measures that will vary from time to time, as the graph shows. If you've set a goal of delivering the home page in about 4 seconds, when you test to verify it, the state of Internet performance may be bad enough to make it fail.

There are some ways to allow for this variation. You could do multiple runs over several days to correct for peaks and valleys in Internet performance and average the results. You could also note how the Internet performance when you ran the test varied from average performance and correct the results for the difference.

Another problem is that the Keynote index uses only home pages in constructing its index. This is perfectly fine for looking at variations in internet performance

over time, but much less useful for setting performance standards because we can't tell to what extent the Keynote sample will match our own pages. The home pages Keystone uses might do more than serve text and images, but we don't know which and to what extent. So if our site uses dynamic html, reads and writes databases, or runs background applications, then response time for a sample page is not a standard these pages will meet on a consistent basis. We have to consider what function pages perform in setting performance requirements. This isn't possible using industry data like this.

Still it shows, roughly, what good and bad performance is on the web.

Error Rates

Some information on error rates compiled by Inverse Network Technology has been published on Internet World's Statistics Toolbox site (http://www.iw.com/daily/stats/1998/05/2702-downloads.html). Table 4 below summarizes this information.

Table 4: Rate of Web Page Download Failures

	% of Failures	% of Time Outs	Total failures
Best in Class	1.31	1.92	3.23
Industry Average	2.15	3.03	5.18

The best performing site listed had combined error rates of about 3.23% and the industry average is about 5.18%. These are intended to measure ISP performance, so they are probably not directly applicable to the error rates you should expect from the web site alone. But, they are certainly an upper boundary for the number of errors you would want to generate at your maximum load.

There are other organizations that compile and sell performance data that's more specifically targeted. For example, ActiveMedia (www.activemedia.com) offers a report on the top 100 E-commerce web sites that includes information on performance. Others are Media Metrix Inc. (www.mediametrix.com), Internet.com's CyberAtlas and Stats Toolbox (www.internet.com). However, these all heavily emphasize research on the business and marketing end of the web. Still, it might be worth the cost to acquire research that addresses load and performance issues. More and more of this information will make it into the public domain in the future. The more the research is broken down into the types of functions that your site does, the better the performance requirements you can extract from it.

If you're breaking new ground on the web, the information isn't well targeted or voluminous.

New Site, Existing Function or Existing Market

If the organization is taking an existing offline function to the web, then the volumes of that function can be used to develop load projections. If a traditional call center takes an average of 5,000 calls a day and you want to move that function on-line, then it's average projected load would be based on 5,000 users per day. You would also likely know about peak demand times, or the number of people on the phones and waiting in the queue at the same time. So you could plan for peak usage and know how many simultaneous users you've had to service using the traditional system. These are good guides to maximum simultaneous users for the web site. Mail order catalogs or help desks would have the same characteristics. However, you would also need to plan for new users since research shows that when a business moves on-line, it doesn't just transfer users from it's offline equivalent, but also attracts a new population of users.

Since basic business functions are similar across organizations, there might be comparable sites online that you could benchmark. This can be done easily with automated test tools or you could contract with an organization like Keynote to run the benchmarks. If you're going to do this yourself, make sure your benchmark transactions do not affect the performance of the site or disrupt it by placing false orders for goods or services.

One of the uncertainties of benchmarking competitive sites is that you won't know what the load on the site was when the response time were measured. You can control for this somewhat by taking several measurements on different days and times.

Replacement Site

Load and performance requirements for a replacement site are more fun to develop than the other cases because there's a lot of interesting analysis you can do for the existing site. The data in the access logs tell you everything about the load that site sustained and you can benchmark it in a very controlled way to learn about access times. Of course, we still need to develop estimates for the

growth of site traffic, but these too can be based, in part, on the growth in the existing site over its life span.

There are many tools for analyzing log files. There's a useful, but probably somewhat out of date list in a good book on log analysis Web Site Stats by Rick Stout. Most of these tools are geared toward a marketing audience and will not, for example, break the data down by the second.

I prefer to extract what I need from the raw logs with a sed script and then read the data into a standard statistical analysis package. These will slice, dice, and filter the data any way you can think of. Also, I have the comfort of knowing exactly what's in my data set and exactly how it's being handled and counted. You can also set up batch jobs that will run the analysis automatically once you get it defined.

Load

The access logs from the existing site contain all the information needed to determine the maximum number of simultaneous users. Figure 7 shows a section from an access log.

Figure 7: Sample from Access Log.

```
198.69.241.2 [26/Jan/1999:07:25:05 0500] "POST /servlet/com.servlet.SearchServlet
HTTP/1.0" 302 221
198.69.241.2 [26/Jan/1999:07:25:25 0500] "POST /servlet/com.servlet.SearchServlet
HTTP/1.0" 302 221
198.69.241.2 [26/Jan/1999:07:25:28 0500] "GET /catalogs/searchresults.jhtml HTTP/1.0" 200
22786
141.185.72.145 [26/Jan/1999:07:25:38 0500] "GET / HTTP/1.0" 302 205
141.185.72.145 [26/Jan/1999:07:25:38 0500] "GET /splash.jhtml HTTP/1.0" 200 15526
198.69.241.2 [26/Jan/1999:07:25:51 0500] "GET /catalogs/acrosgroup.jhtml HTTP/1.0" 200
18179
207.31.232.160 [26/Jan/1999:07:25:56 0500] "GET /support/flit.jhtml HTTP/1.1" 200 8879
```

Each line is a request to the server that records the IP address of the requester, the time the request was made, and what file was requested. Actually, it records more information, but this is all we need for our purposes. Since the time is recorded to the second, it's just a matter of counting the number of unique IP addresses that made requests during each second, and then finding what the maximum value is. Table 5 shows the raw values for number of users per second and Table 6 the minimum, maximum, and average simultaneous users at the busiest time of the day for a sample site. As mentioned above, it may also be useful to find the average value.

Table 5: Users Per Second.

Time	Users

8:30:19	89
8:19:50	84
8:54:49	84
8:22:15	83
8:20:25	81
8:36:44	80
8:23:20	79
8:39:04	78
8:22:45	77
8:30:54	76
8:51:54	75
8:44:54	74
8:45:29	73
8:29:45	71
8:37:19	71
8:37:54	69
8:48:59	69
8:19:15	68
8:35:34	68
8:29:44	67

Table 6: Simultaneous Users 5 – 9 AM.

Hour	Log Records	Minimum	Maximum	Mean
5 AM	3601	2	34	14.58
6 AM	3580	3	68	24.52
7 AM	3600	12	73	30.07
8 AM	3591	4	89	35.25

For some sites, there may be multiple servers. The most common situation of this type is having both a secured and unsecured server, but there may be an image server or multiple virtual sites. In this case, all the logs need to be combined and all of them have to be recording IP addresses in the same way, that is resolving them into names or not. Otherwise, the same user could be counted as 2 users because the IP address is being recorded differently.

If the logs are very large, it might be easier to select a peak period as the data for the maximum value, as was done in Table 6, and an average period or random sample of records as the data for the average value.

This analysis produces the load numbers for the existing site, which still need to be adjusted for growth. If logs are available from some time in the past, the growth rate of the existing site can be determined and then projected into the future. This kind of forecasting can be done very rigorously using various methods of time series analysis. However, in most cases, such rigor isn't

necessary since we're interested mainly in planning for the peak load, so when that occurs and how fast the site grows (the shape of growth) is unimportant as long as we know what the load goal is. As cautioned before, the accuracy of these projections diminishes the farther out the forecast goes.

If the business plan sets certain goals for online customers or sales, these can also be used along with the current customers or sales and current load to estimate the load requirement that would be necessary to support those goals. The formula is shown in Figure 8.

Figure 8: Formula for Load Requirement.

$$\text{Future Load} = \frac{\text{Future Sales/Users} * \text{Current Load}}{\text{Current Sales/Users}}$$

Having data from an existing site is a big advantage and makes estimating load requirements much more certain.

Performance

Existing sites can provide a baseline from which to estimate performance requirements. One straightforward way to do this is to benchmark the existing site. This builds a performance profile that highlights problem areas and, for areas that are performing well, sets the performance requirement for the revised version of the site. Changes aimed at enhancing response time in the replacement site can be checked against the old version to see if they actually accomplish that goal.

Benchmarking an existing site also allows us to intelligently analyze different types of transactions that use different technologies that might affect performance. For example, some pages may use dynamic html, do database queries, link to legacy systems, or search catalogs. These might all rely on different technologies to do what they do so performance standards should be different. Transactions or pages that are critical to customer satisfaction or closing sales can also be timed to make sure their performance isn't inhibiting their function.

To time transactions accurately we need a transaction profile which lists the kinds of transactions the site performs and how often they are performed. The access log record of page requests is the source for this information. However, the frequency of requests for specific pages may not be define the transactions well enough, since a transaction may take several page requests to complete.

So you have to do some hand coding and group the page requests together into transactions.

Table 7 is a frequency table of page requests from an access log.

Table 7: Page Hits

Page	Hits	Transaction
/	21628	home
/splash.jhtml;	15182	splash
/servlet/com.servlet.SearchServlet;	11832	search
/servlet/com.servlet.CatalogServletImpl;	11099	catalog
/catalogs/searchresults.jhtml;	9256	search
/dhtml.jhtml;	8098	home
/catalogs/fcatgroup.jhtml;	6748	catalogs
/catalogs/fcathiertabndhtml.jhtml;	6495	catalogs
/ndhtml.jhtml;	4275	home
/catalogs/fcathier.jhtml;	3847	catalogs
/catalogs/ncatgroup.jhtml;	3111	catalogs
/Login/;	2522	login
/catalogs/proxy.jhtml;	2491	catalogs
/ordering/shopping-cart.jhtml;	2487	orders
/catalogs/root.jhtml;	2438	catalogs
/servlet/com.servlet.ShoppingCartServlet;	2371	orders
/catalogs/;	2206	catalogs
/;	2060	home
/catalogs/acrosgroup.jhtml;	2007	catalogs
/servlet/com.servlet.LoginServlet;	1955	login
/ordering/rapidorder.jhtml;	1805	orders
/catalogs/chemgroup.jhtml;	1692	catalogs
/personal.jhtml;	1450	personal
/catalogs/fcat.jhtml;	1397	catalogs
/catalogs/powersearch.jhtml;	1387	search
/servlet/com.servlet.RefererServlet;	1193	referer
/Login/register.jhtml;	968	login
/servlet/com.servlet.OrderPlacementServlet;	956	orders
/ordering/;	909	orders
/servlet/UserAdmin;	871	admin

A request for a file in the /catalogs/ directory generally means a that the catalog is being viewed by the user. So all those requests become part of a catalog browse transaction. The exception is the search function, whose applications and pages are also located in the catalogs/ directory. These page hits constitute part of the search function.

Once all the pages are categorized into transactions, the frequency of transactions can be determined. Table 8 is a sample transaction frequency table.

Table 8: Transaction Frequency

Transaction	Hits	% of Hits
catalogs	57490	36.43
home	36061	22.85
splash	15186	9.62
search	11832	7.50
orders	11601	7.35
login	7902	5.01
support	3728	2.36
about	2296	1.45
admin	2224	1.41
supplier	1741	1.10
personal	1450	0.92
referrer	1193	0.76
register	1157	0.73
whatsnew	1145	0.73
help	896	0.57
sitemap	804	0.51
hotlist	661	0.42
template	228	0.14
error	223	0.14
ttracker	2	0.00

The transaction profile determines how many of each type of transaction you set up using your automated test tool. So, the benchmarks you r u n will mirror patterns of user behavior and when you increase the number of test users to check load, you won't distort those patterns.

If the results show numbers that you suspect might be a problem, you can check them against available industry performance data or benchmark similar transactions on other sites. If the site collects user feedback, then there may be information on areas of the site that are particularly slow, which you can correlate with the transaction responses. The conclusions drawn form this will help guide the site revision into areas which need attention and make sure that other areas perform as well as they did before.

Assuming similar transactions are implemented on the new site, the benchmarks can be used to see if the performance of slow transactions has improved and to regression test others. The old automated scripts might not work because the pages and objects on the pages probably changed. But most of the transactions, like logging in, searching, browsing, and buying should be the same if the redesigned site does the same things the old one did.

Conclusion

To have decent, testable load and performance requirements we have to at least fill in the three basic blanks of maximum load, maximum response time, and maximum error rate. The basic method for doing this is to get as much information on current load and performance as possible and combine it with future goals to forecast the requirements. Depending on the situation, which can range from a brand new site to a replacement for an existing one, the quantity and quality of available data will vary. I've tried to suggest some sources of information and methods that anyone might find useful in tackling this task. However, I'm sure that the knowledge and experience you bring to the task will suggest other sources and methods that will enhance the set of tools we have to deal with developing load and performance requirements.

Agenda

- Complexity of Java environments
- Complexity of testing Java
- Develop for testability

RATIONAL SOFTWARE

What *is* Java software, anyhow?

- Where does it fit on the Windows desktop?
- What's an Applet and how is it deployed?
- What's an Application?
- What is Java software made of?

RATIONAL SOFTWARE

Java's Promise

- It is used in many different places
  - Java Applets -- in browsers, applet-viewer
  - Java Applications -- Sun and Microsoft VMs
  - Java Runtime Environment (from Sun)

WRITE ONCE, RUN EVERYWHERE!

RATIONAL SOFTWARE

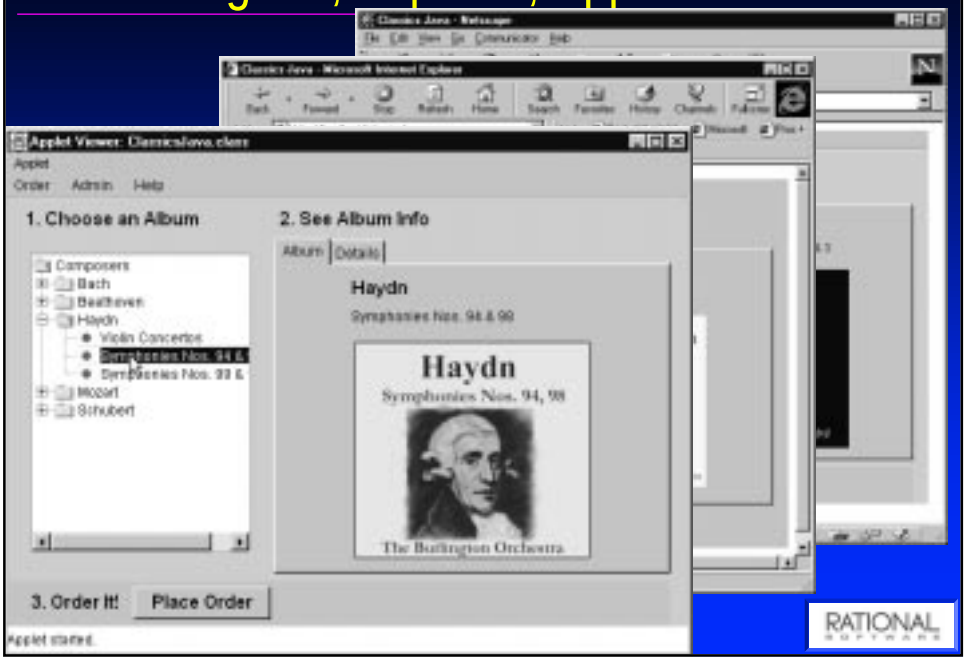
Java's Reality

- It is used in many different places
  - Java Applets -- in browsers, applet-viewer
  - Java Applications -- Sun and Microsoft VMs
  - Java Runtime Environment (from Sun)

WRITE ONCE, TEST EVERYWHERE!

RATIONAL SOFTWARE

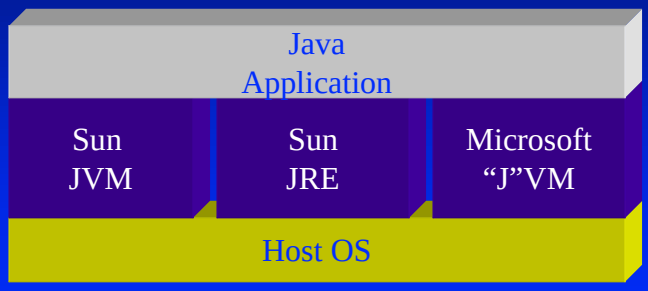
Navigator, Explorer, Applet Viewer



RATIONAL SOFTWARE

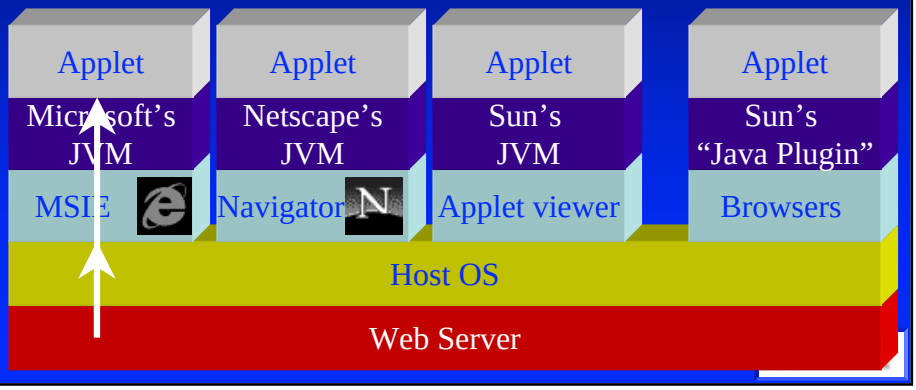
Java Apps - programs on the desktop

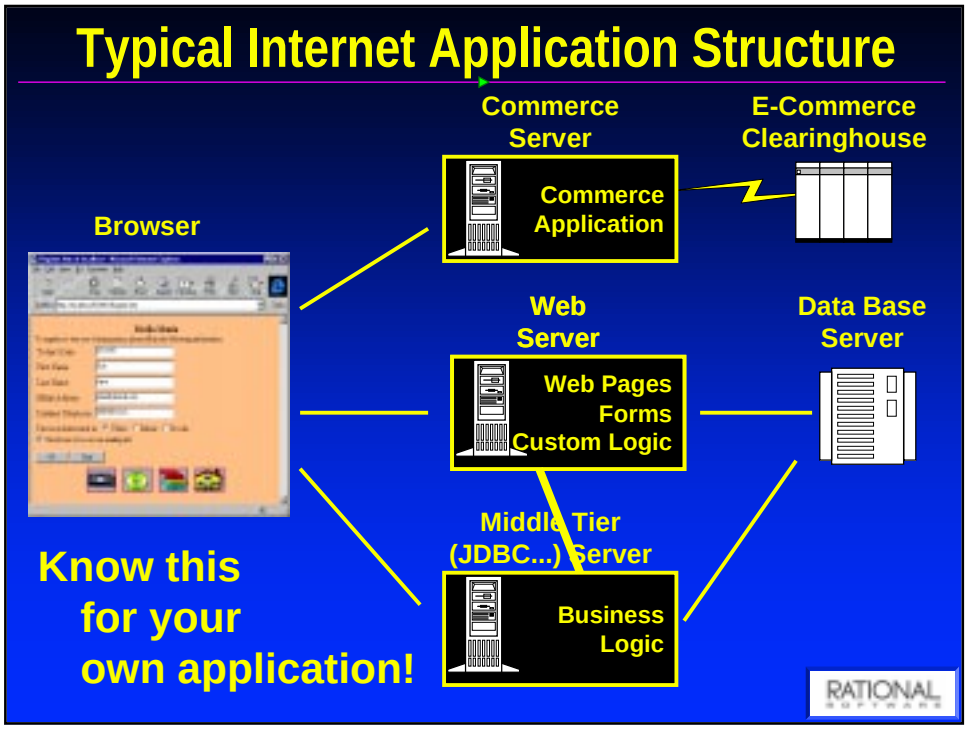
- *Several possible environments.*
- *Each application is likely to be deployed in just one (or maybe two) environments on Windows*

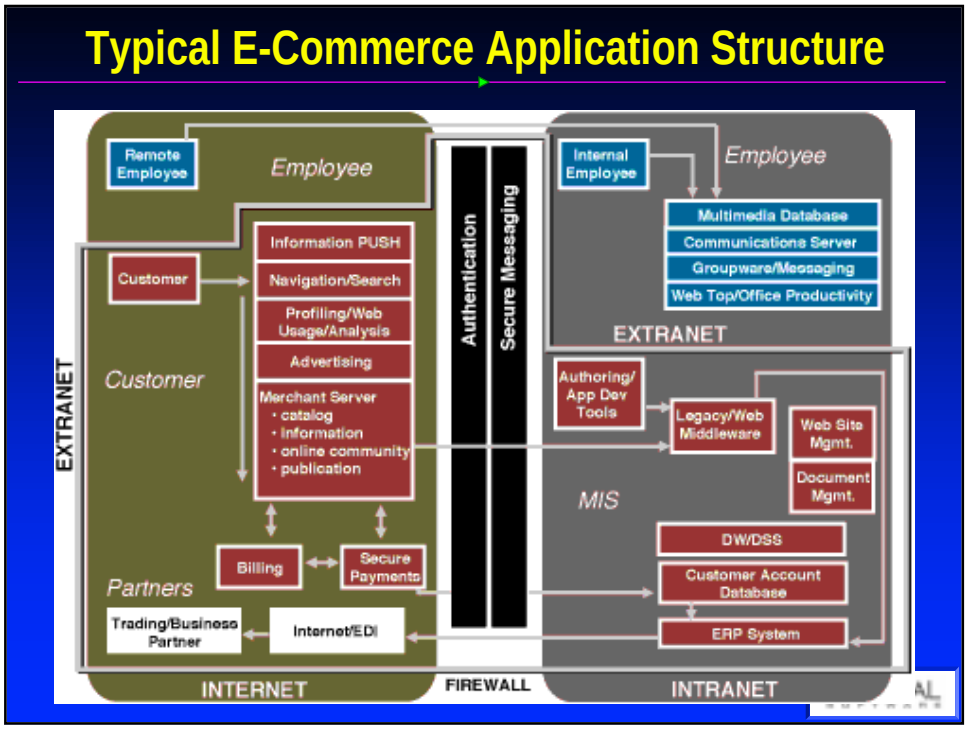


Applets - Web apps running in a browser

- *Many pieces to this configuration*
- *Before testing, know the target environments*
- *Regression testing is necessary*

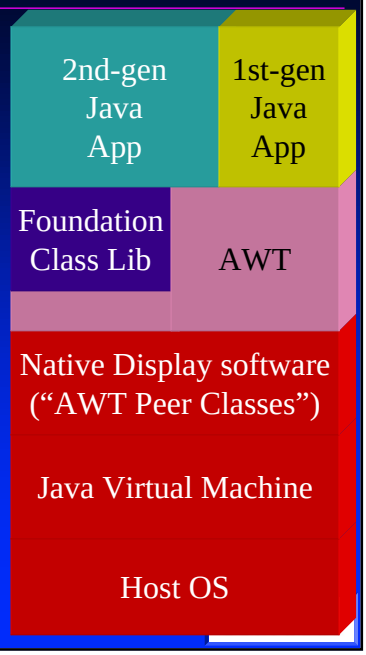



Typical Internet Application Structure
Commerce Server
E-Commerce Clearinghouse
Commerce Application
Browser
Web Server
Data Base Server
Web Pages
Forms
Custom Logic
Middle Tier (JDBC...) Server
Business Logic
Know this for your own application!
RATIONAL


Typical E-Commerce Application Structure
Remote Employee
Employee
Internal Employee
Employee
Multimedia Database
Communications Server
Groupware/Messaging
Web Top/Office Productivity
EXTRANET
Customer
Information PUSH
Navigation/Search
Profiling/Web Usage/Analysis
Advertising
Merchant Server
• catalog
• information
• online community
• publication
Authentication
Secure Messaging
Authoring/ App Dev Tools
Legacy/Web Middleware
Web Site Mgmt.
Document Mgmt.
MIS
DW/DSS
Customer Account Database
ERP System
Billing
Secure Payments
Partners
Trading/Business Partner
Internet/EDI
INTERNET
FIREWALL
INTRANET

Java software -- what it's made of ?

- GUI toolkit, drawing software
- Foundation Class choices:
  - Custom classes
  - Sun's JFC / Swing
  - Symantec Café
  - MS AFC
  - MS WFC
  - Abstract Windowing Toolkit underneath other display classes

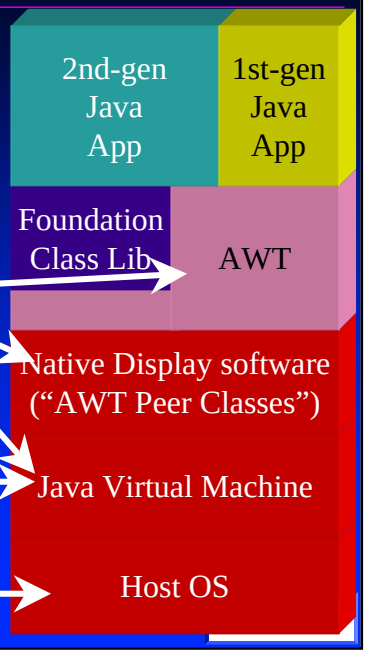


Java software -- what it's made of ?

Java Development Kit version (1.0.3, 1.1.6, etc)

Java Virtual Machine version (1.0.3, 1.1.6, Microsoft, etc)

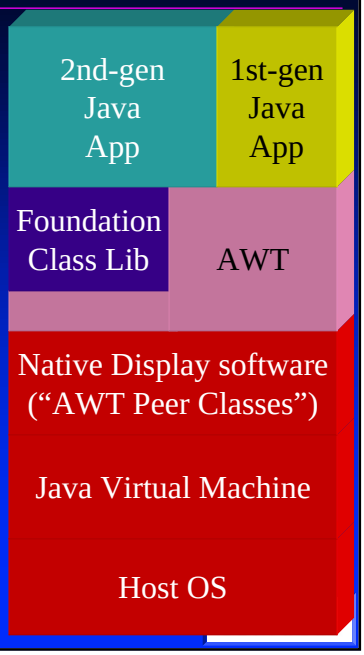
Host Environment



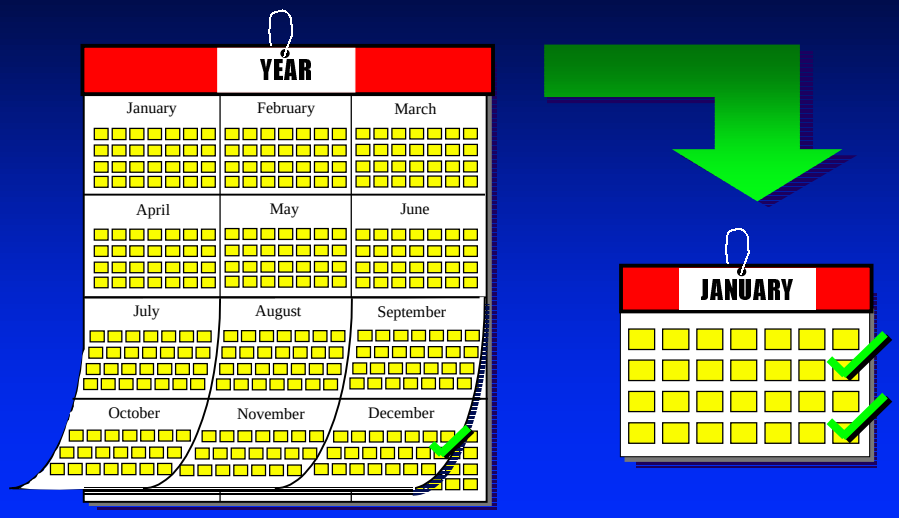
Java software -- what you have to know!

Before beginning a testing project, get some answers about the software

- **Application? Applet?**
- **What Foundation Class set?**
- **Any custom display classes? (What development IDE?)**
- **What JDK version ?**
- **What deployment browsers (applets)?**
- **What OS platforms?**



Internet Time



RATIONAL SOFTWARE

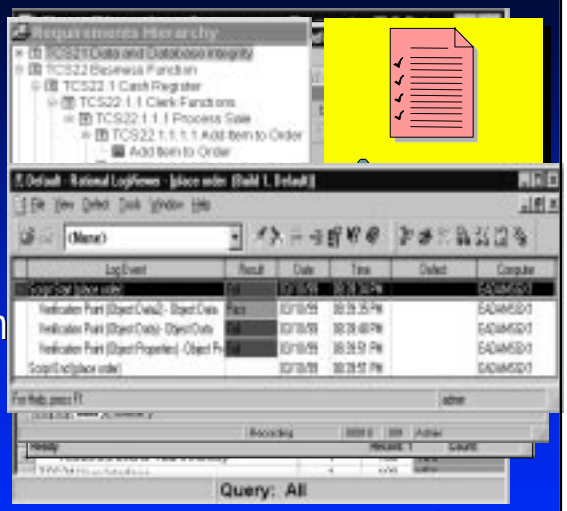
Agenda

- Complexity of Java environments
- Complexity of testing Java
- Develop for testability

RATIONAL SOFTWARE

Recall what you have to communicate...

- Requirements
- Test Plan and coverage
- Test Scripts
- Defects - including state, priority, system config., etc.
- Regression results



Are we ready to release?

RATIONAL SOFTWARE

To effectively test Web Apps, you MUST

- Be able to tell how far we had tested against the current system requirements in the test plan
- Check for reliability automatically with each new build
- Create test scripts for new functionality quickly and easily
- Make test scripts on one iteration and keep using them on all of them
- Use one set of test scripts for all configurations
- Load test with confidence and ease before deploying
- Communicate the results and share the test assets and metrics

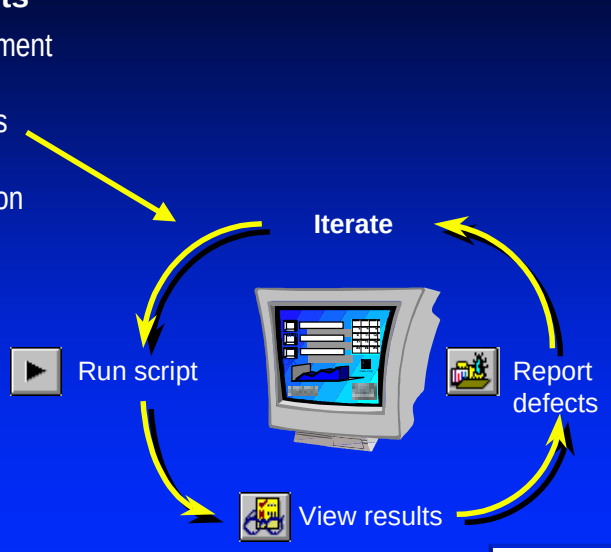
RATIONAL SOFTWARE

Steps in Functional Regressions Testing

1. Create test scripts

- Select requirement
- Record actions
- Insert validation
- Edit script (optional)

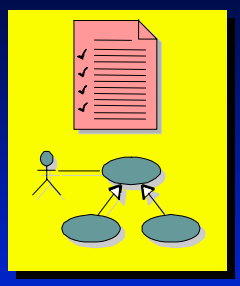
2. Run tests



RATIONAL SOFTWARE

Track dynamic requirements

- Understand the requirements to create the test
- Report coverage against current requirements
  - Where do results still apply?
- Track and document

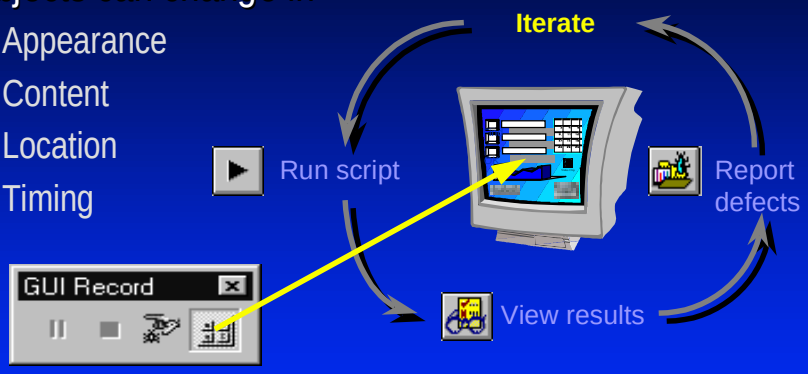


Are we ready to release?

RATIONAL SOFTWARE

Track System-Under-Test Changes

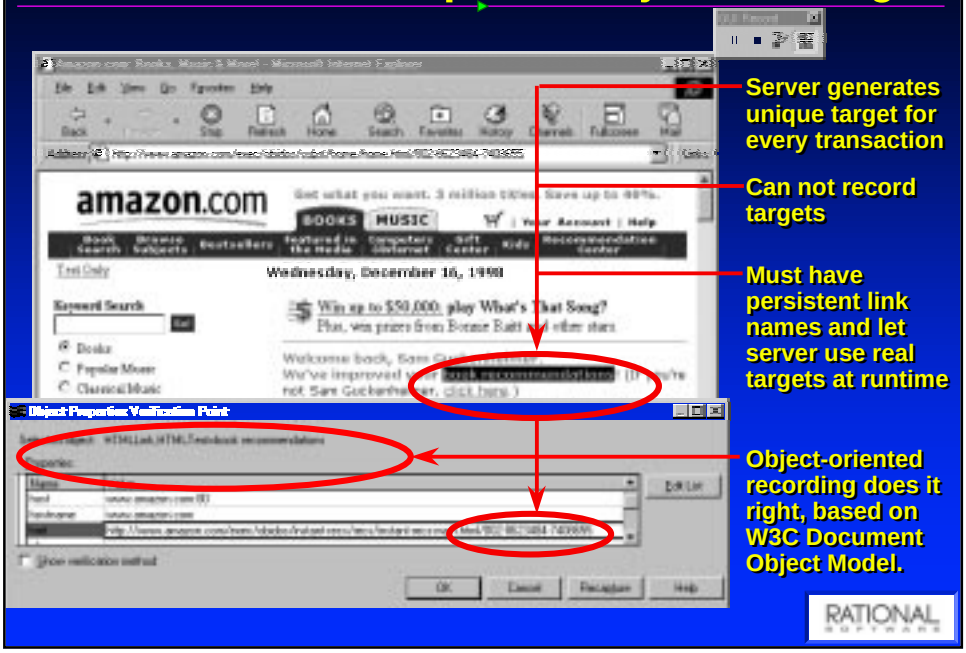
- Objects can change in
  - Appearance
  - Content
  - Location
  - Timing



- **Object recording required to make test scripts robust**
  - When application changes
  - Across configurations

RATIONAL SOFTWARE

Web Content Requires Object Testing



Server generates unique target for every transaction

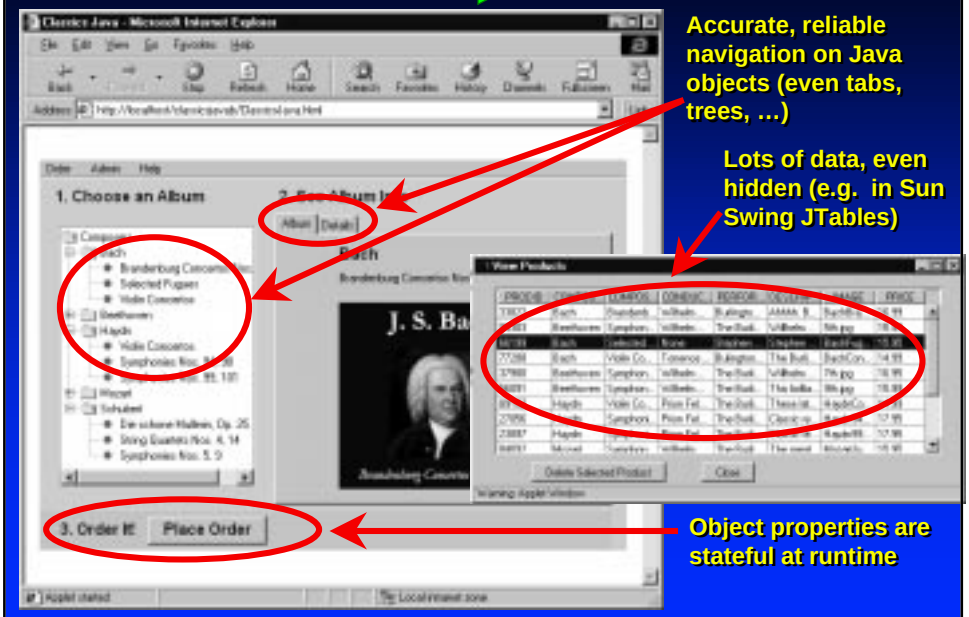
Can not record targets

Must have persistent link names and let server use real targets at runtime

Object-oriented recording does it right, based on W3C Document Object Model.

RATIONAL SOFTWARE

Object Testing on Java

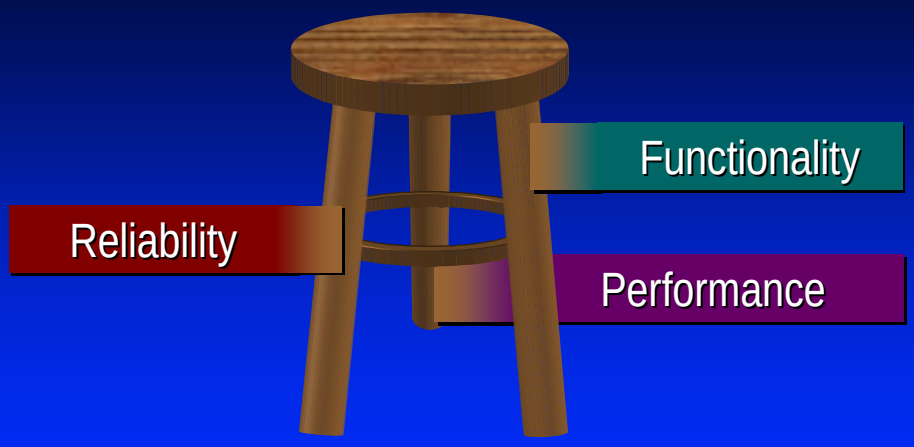


Accurate, reliable navigation on Java objects (even tabs, trees, …)

Lots of data, even hidden (e.g. in Sun Swing JTables)

Object properties are stateful at runtime

Test All Dimensions of Quality



RATIONAL SOFTWARE

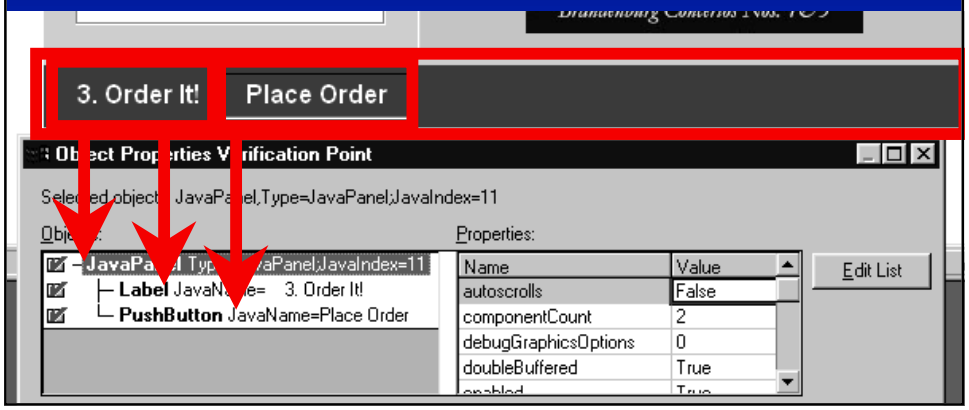
Agenda

- Complexity of Java environments
- Complexity of testing Java
- Develop for testability

RATIONAL SOFTWARE

Developing Java to enhance testability

- Naming of Java objects: flatten out the subdivision hierarchy; do not depend on inheritance
  - E.g. "JavaPanel; JavaIndex=11" include Label & PushButton

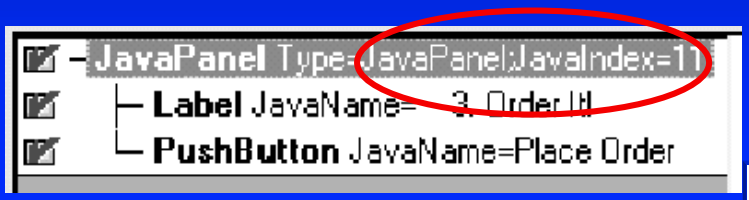


Developing Java to enhance testability

- Give the Java objects names; set the **name** member variable -- optional -- increases testability by allowing object recognition by name.
  - For example, in AWT:

  orderButton.name = ".Place Order";
  - ... in JFC:

  orderButton.accessibleName = "Place Order";

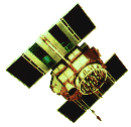


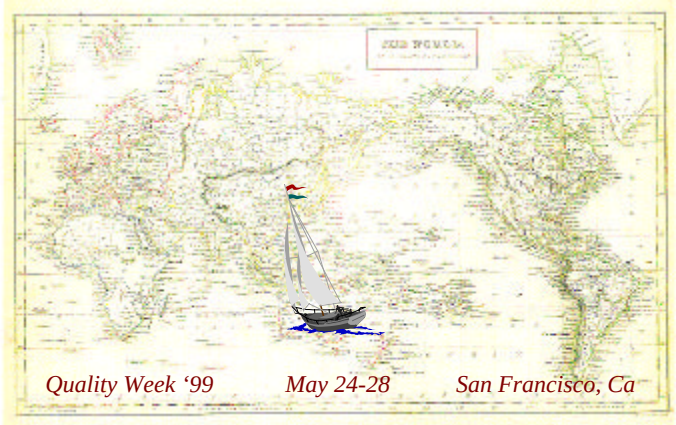
RATIONAL SOFTWARE

Developing Java to enhance testability

- Derive your custom Java object classes from standard JFC / AWT classes -- a good GUI testing tool recognizes the object as an instance of the standard object from which it is derived
- Put testability information into public member variables in your Java objects -- a good GUI testing tool sees these public members as object properties.

RATIONAL SOFTWARE





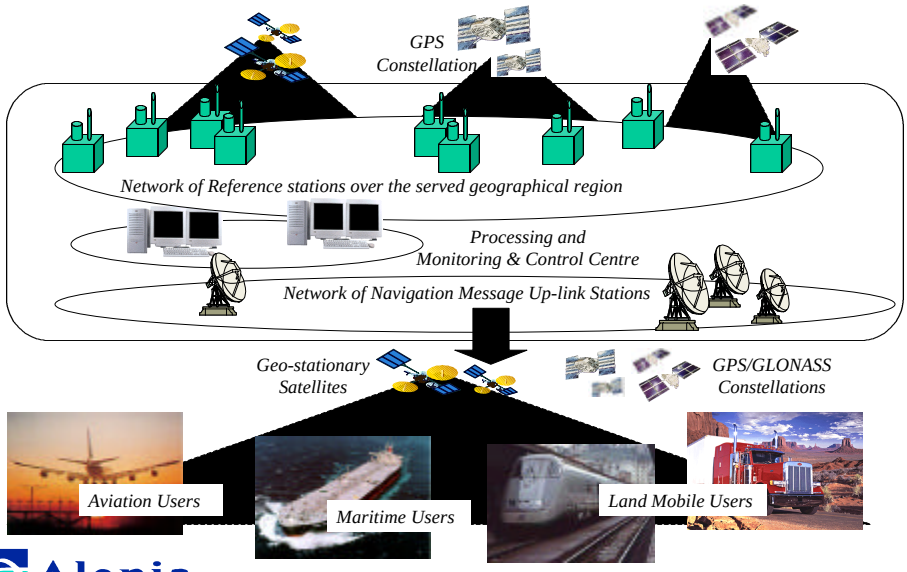
Quality Week '99 *May 24-28* *San Francisco, Ca*

Authors: L.Lattanzi, M. Musmeci

Alenia
AEROSPAZIO

A Finmeccanica Company

Overview of a Global Navigation Satellite Augmentation System

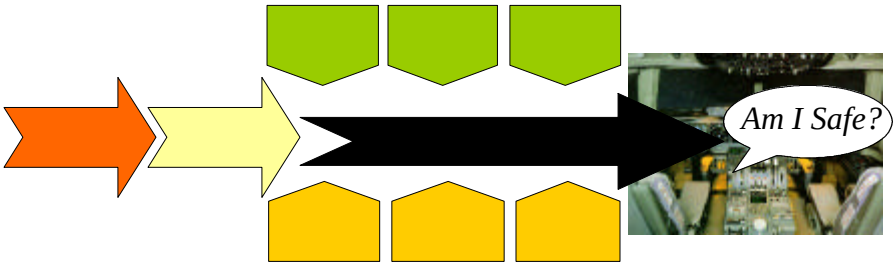


Alenia
AEROSPAZIO

A Finmeccanica Company

Approach to Safety Critical S/W Systems

For Critical Real-Time S/W with high level of automation, the size, the complexity and the required effort necessarily increase.

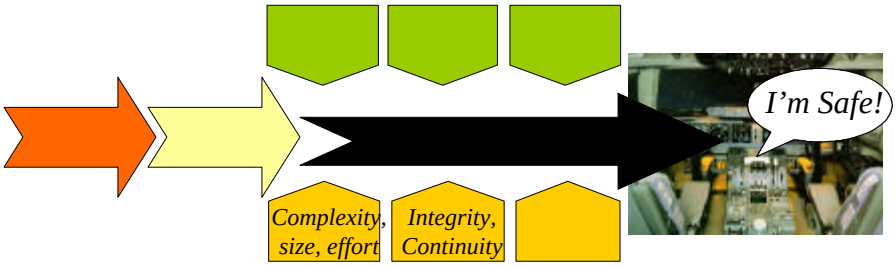


How to approach ?

Alenia AEROSPAZIO

Approach to Safety Critical S/W Systems

A rigorous process, from specifications to final deployment and maintenance, becomes mandatory to guarantee the



required level of safety in accordance with applicable certification requirements. This goal has to be achieved

Alenia AEROSPAZIO

Methodology

Dependability and Safety issues have to be considered in all their aspects and in all phases of the project.

The goals of RAMS activities are:

- *to demonstrate that reliability, availability, maintainability, and safety targets are met;*
- *to identify all technical risks;*
- *to define risks reduction and control measures;*

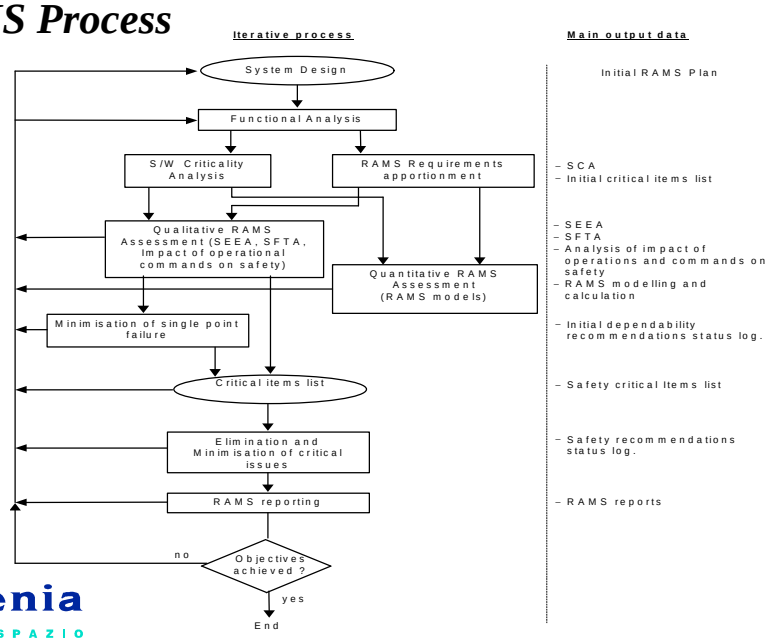
RAMS main assessments:

- *Qualitative Requirements (for integrity and continuity);*
- *Quantitative Requirements;*

Alenia AEROSPAZIO

A Finmeccanica Company

RAMS Process



Alenia AEROSPAZIO

A Finmeccanica Company

Mitigation Techniques

The Mitigation Technique (MT's) are possible course of actions with the goal of removing, or at least to reduce, the criticality level of a system or part of it.
There are different techniques for each phase of the software life cycle and for each specific activity;
A possible classification is the following:

- *General guidelines and methodologies;*
- *S/W Specification and Design techniques;*
- *S/W Development techniques;*
- *S/W AIV techniques;*

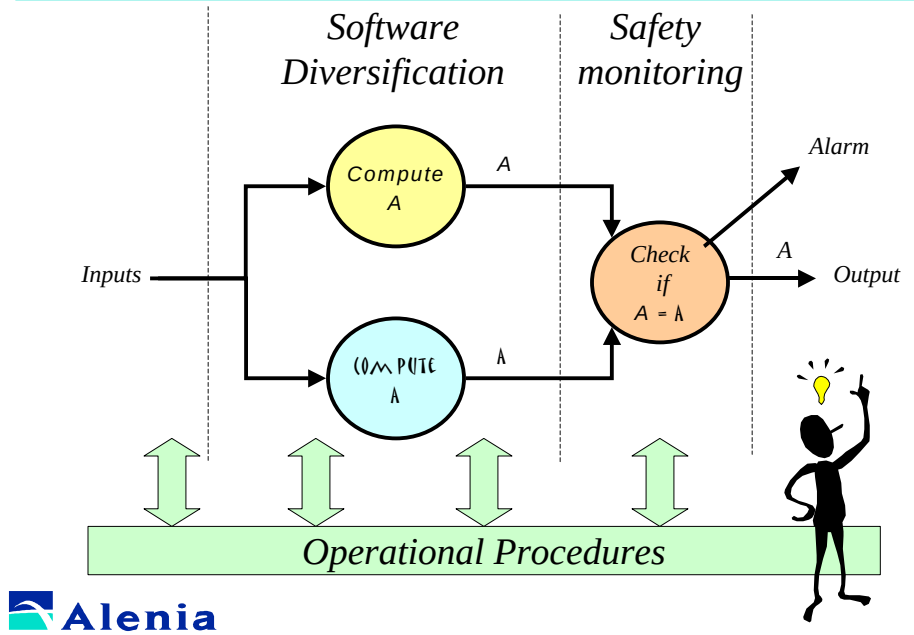
Alenia AEROSPAZIO

Examples of Mitigation Techniques on S/W Specifications and Design

- *Constraints on S/W Engineering methodology;*
- *Design and Functional Redundancy (diversification);*
- *Consistency Check (safety monitoring);*
- *Partitioning (isolation of critical functions);*
- *Constraints on used S/W COTS and re-used S/W;*
- *Prototyping and Simulation;*
- *Specific Operational Procedures;*

Alenia AEROSPAZIO

A Finmeccanica Company

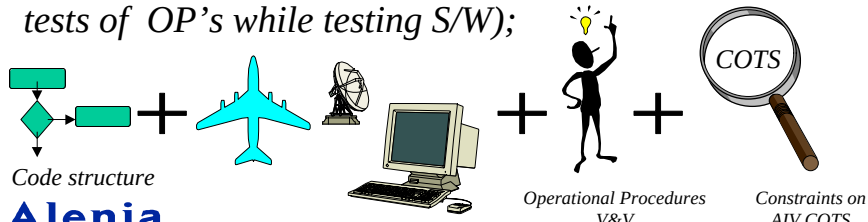


Alenia AEROSPAZIO

A Finmeccanica Company

Examples of Mitigation Techniques on AIV

- *Code structure verification coverage;*
- *Representative AIV Environment (real S/W instead of simulator or stubs, real platform, real scenario);*
- *Constraints on AIV tools (simulators, emulators, use of COTS test tools);*
- *Operational Procedure (OP's) verification and validation (operator checks, alarms acknowledgement, tests of OP's while testing S/W);*



Alenia AEROSPAZIO

A Finmeccanica Company

Software Verification

To detect and report errors that may have been introduced during S/W development process:

1) Reviews and Analysis Activities

- *Technical and Management reviews;*
- *Walkthrough and Inspections;*
- *Reporting;*

2) Audits to S/W development team;

- *Through the whole S/W life cycle and system*

4) Testing Activities

- *Compare expected and actual results;*
- *Demonstrate the absence of unacceptable errors;*

Management Issues

The management is fundamental in both making decisions and in monitoring the project progress.

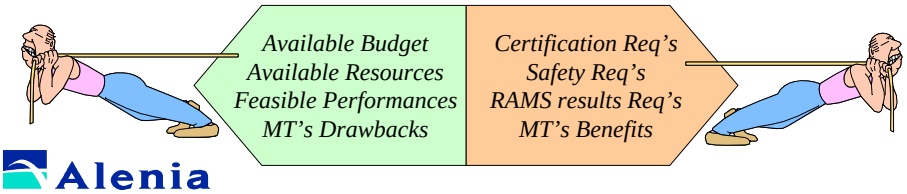
There two main branches:

1)- Trade-off analysis in making decisions;
2)- Risk Management activities for the whole S/W life cycle;

Trade-off analysis in making decisions

The objective is to select the "best" solution to meet the safety requirements while controlling the growing complexity and corresponding costs/resources.
The selection is driven by two needs:

- *Improve the reliability and robustness of the system;*
- *Keep under control the cost increase, affected performance and increased size of the system itself;*



Alenia AEROSPAZIO

Risk Management:

Risk Definition

Risk is a measure of the probability and consequence of not achieving a defined program goal.

Risk Management goal

Risk Management is a systematic approach to support program management in the optimisation of program resources with the purpose to identify, assess, reduce, prioritise, control, document and communicate the risks involved in a program with reference to cost, schedule

Alenia AEROSPAZIO

Risk Management process

The Risk Management Process includes the following

1) Definition of the Risk Management Policy;
2) Identification and Assessment of the Risks;
3) Reduction of the Risks and/or their Acceptance;
4) Monitoring, Communication and Acceptance of the Risks.

Although there is a logical flow in the process, Risk Management is an iterative process with considerable interaction and feedback among the process elements.

Alenia AEROSPAZIO

A Finmeccanica Company

Conclusions

For Safety critical S/W systems some of the normal engineering practices have to be improved, expanded and constrained by certification and safety requirements.
The Management is in charge to adopt specific mitigation techniques and to set-up risk management process, by means of a complex trade-off between cost issues and technical needs, without compromising on the quality and timeliness of the project.

In doing this exercise the already gained experience and the already consolidated company standards have to be followed to the maximum possible extension.

Alenia AEROSPAZIO

A Finmeccanica Company

1. The Challenge of Safety Critical Developments

There are projects where safety critical real-time software is a strategic issue. Automated systems like auto-pilots, fly by wire, breaking systems, speed control systems are dramatically increasing in size and complexity. It is therefore becoming more and more necessary to develop them using a rigorous process from specification to final system deployment. With this level of complexity only developers who master such a process can be expected to develop systems which will guarantee the required level of safety in accordance with certification requirements.

In addition, the formal process allows to control costs and personnel resources. Then the real challenge for the project leaders will be to increase the productivity of their development team without compromising on the quality and timeliness of their project.

1.1 Principles of a Global Navigation Satellite System

GPS and/or GLONASS, when used in an autonomous way, cannot cope with the very strong integrity requirements of civilian aviation users, and the availability and accuracy of these systems are not high enough for some of the more critical flight phases. The civilian aviation authorities requirements have therefore presently given a boost to three programs with the aim of making up for these shortcomings, namely the systems WAAS (Wide Area Augmentation System) in the USA, EGNOS (European Geostationary Navigation Overlay System) in Europe and MSAS (Multi-functional transport Satellite-Based Augmentation System) in Japan. Each of these three systems, which are all compatible and inter-operable, consists of the following main elements:

- A ground network of "Reference Stations" for observing the GPS and/or the GLONASS constellations;
- A Processing Facility/ ies to collect these data and to compute the correction and to perform the augmentation (this is the kernel of the system where ad-hoc algorithms are used to compute the navigation message);
- A Ground Network of "up-link stations" for the navigation message up-link to the Geostationary satellites;
- The Geostationary satellites in charge to broadcast the GPS-like navigation message containing the integrity-related information and additional corrections received from the observation network;

A civil aviation user, maritime or land-mobile user, is able, by means of a compatible reception equipment, to receive the navigation message with high level of accuracy, continuity, integrity and availability requirements.

These systems then will provide a regional wide-area augmentation of the current standard positioning systems service (GPS and GLONASS) to the levels required for Category 1 precision approach for civil aviation.

In the rest of this paper the generic navigation system, when used as an application example, will be called GNSS (Global Navigation Satellite System).

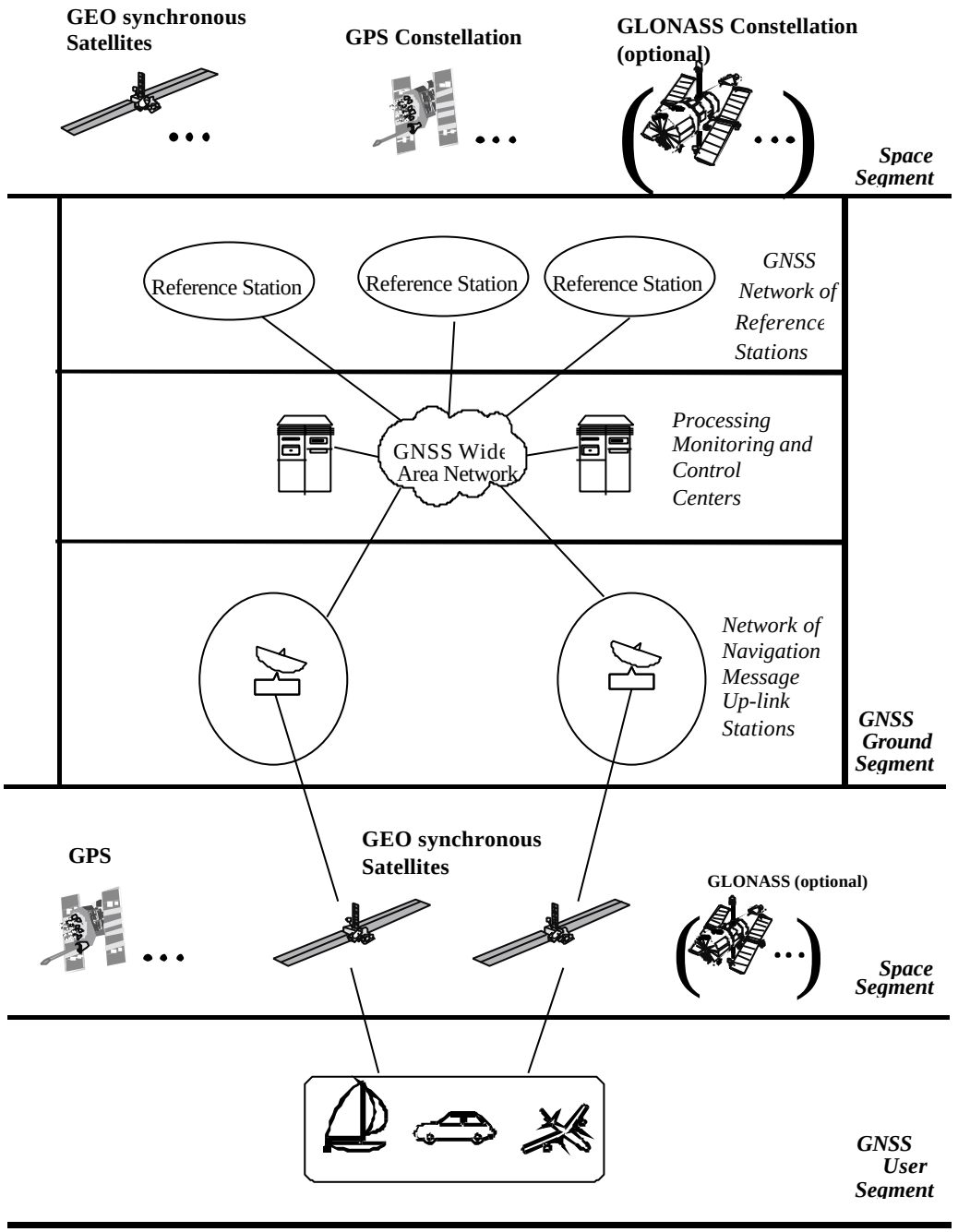


Figure 1.1-1: Navigation Satellite Augmentation System

2. RAMS Methodology

Dependability and Safety are inherent characteristics of any system, this is even more true for a safety critical system.

Dependability and Safety issues have to be considered in all their aspects, in all phases of the project. To ensure a successful mission, the objectives of Reliability Availability Maintainability Safety (RAMS) activities are:

- to demonstrate that reliability, availability, maintainability, and safety targets are met,
- to identify all technical risks,
- to define risks reduction and control measures.

This is a continuous and iterative process which has to be reiterated through the design, development and integration/validation processes. The RAMS process can be seen as the result of two branches: the *qualitative* assessment and the *quantitative* assessment. The objectives of them are described here below.

The aim of the *qualitative* assessment is to support the design (e.g. supporting software classification) and to justify the compliance with the qualitative requirements such as: in a GNSS system there must be no single software failure leading to loss of:

- system integrity (considered as a catastrophic event),
- system continuity (considered as a critical event),

The *quantitative* assessment has two essential objectives :

- to ensure compliance with the quantitative requirements,
- to identify and focus on the critical items for the RAMS parameters.

The main objective of RAMS analyses which have to be performed on the system is to ensure that both quantitative and qualitative requirements are met.

The various techniques used are shortly discussed in order to have an overview of the various steps which lead to the identification and isolation of the most critical S/W components within a system. The following flow-chart summarises the overall RAMS process, highlighting the main activities and the related outputs.

Definitions

Integrity : the assurance that all functions of a system perform within operational limits. *Integrity* is the ability of the navigation system to provide timely warnings to users when the system should not be used for navigation. The *integrity risk* is the probability of an undetected failure which will result in the loss of the specified accuracy

Continuity : the ability of the navigation system to provide its service without interruption during an operation. More specifically, continuity of (service of) a system is the probability that the system will be available for the duration of a phase of operation, assuming that the system was available at the beginning of that phase of operation.

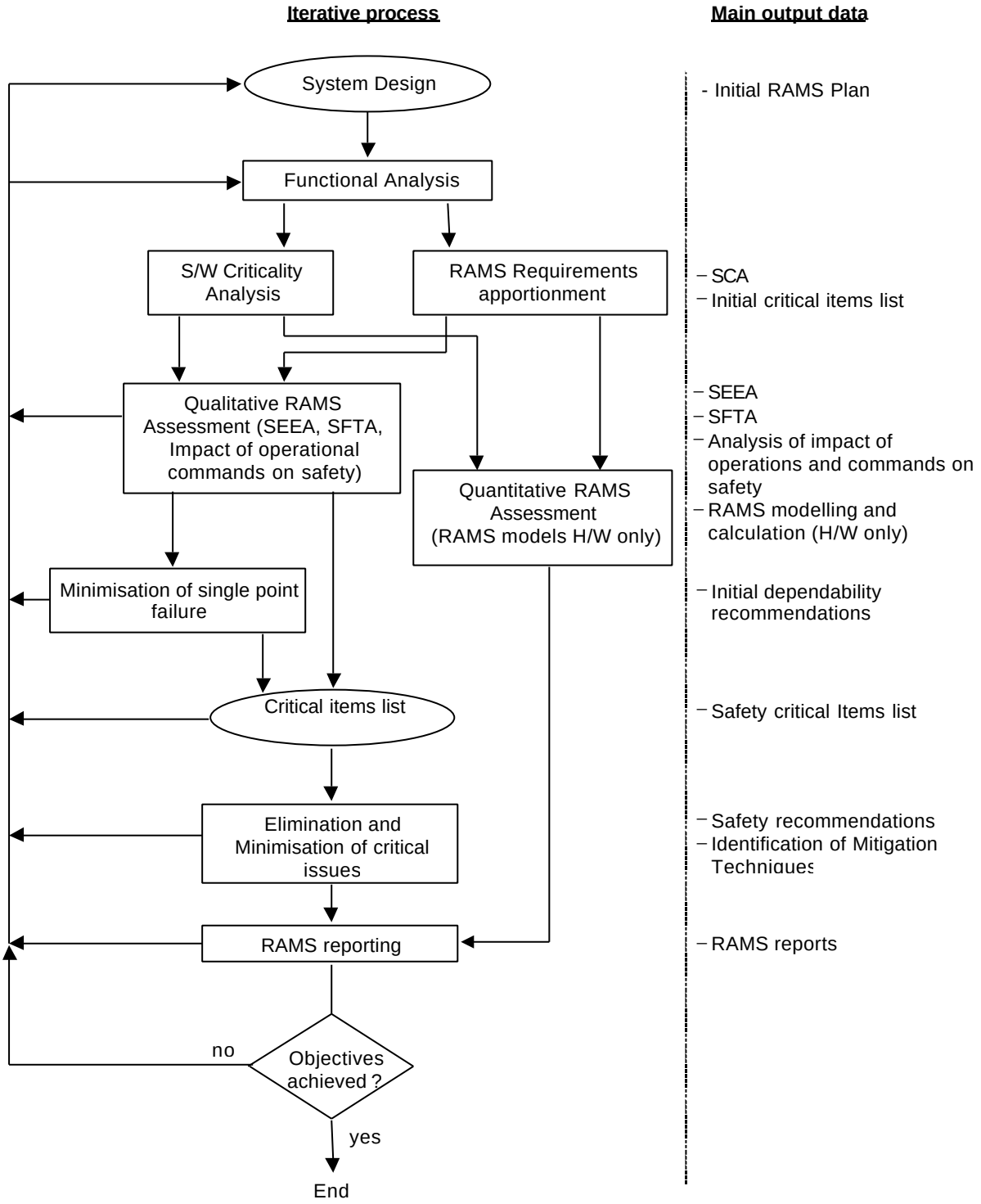


Figure 2-1: RAMS Process

2.1 S/W Criticality Analysis

The standard normally used when a software package is to be certified for usage related to safety critical airborne application is the RTCA DO178-B. According to this standard all the S/W components are classified in one of the following categories:

Level A: S/W whose anomalous behaviour, as shown by the sy stem safety assessment process, would cause or contribute to a failure of system function resulting in a catastrophic failure condition for the aircraft.
Level B: S/W whose anomalous behaviour, as shown by the system safety assessment process, would cause or contribute to a failure of system function resulting in a hazardous/severe-major failure condition for the aircraft.
Level C: S/W whose anomalous behaviour, as shown by the system safety assessment process, would cause or contribute to a failure of system function resulting in a major failure condition for the aircraft.
Level D: S/W whose anomalous behaviour, as shown by the system safety assessment process, would cause or contribute to a failure of system function resulting in a minor failure condition for the aircraft.
Level E: S/W whose anomalous behaviour, as shown by the system safety assessment process, would cause or contribute to a failure of system function with no effect on aircraft operational capability or pilot workload.

In the context of a GNSS project the above failure conditions can be categorised as follows

Failure Condition (DO178B)	**Meaning within the project**
Catastrophic	Loss of integrity of the Navigation System
Hazardous/Severe-Major	Loss of continuity of the Navigation System
Major	Continuity risk is exceeded
Minor	Loss of system budget margins for continuity
No Effects	Services Accuracy, Availability, Continuity, Integrity performed with reduction of margins

In order to obtain the proper classification for each S/W component a Software Criticality Analysis (SCA) is performed. The resulting S/W classification has a major influence on the development process.

The SCA is a joint activity between the disciplines of product assurance, engineering and operations. Specialist of these three disciplines work together to optimise the design and to improve safety and reliability of the system.

The categorisation applies to S/W components "involved" in critical functions. The categorisation scheme is Configuration Item (CI) oriented, i.e. each S/W component identified in the product tree is analysed.

Definition: for a given system design a S/W component p is involved in a function F if and only if any failure of p may result in a failure of F or in a degradation of the implemented failure tolerance for F. A S/W failure is defined as inability of a S/W component p to perform its required function within specified limits.

If the anomalous behaviour of a S/W component contributes to more than one failure condition, then the most severe failure condition category of that component determines the S/W level for that S/W component.

Initially, the system safety assessment process determines the software level(s) appropriate to the software components without regard to the system design. Only the impact of failure, both loss of function and malfunction, is addressed when making this determination.

There are various architectural strategies, which during the evolution of the system design may result in the S/W level(s) being revised.

2.2 System Architectural Considerations

If the system safety assessment process determines that the system architecture precludes anomalous behaviour of the S/W from contributing to the most severe failure conditions of a system, then the S/W level is determined by the most severe category of the remaining failure conditions to which the anomalous behaviour of the S/W can contribute.

The system safety assessment process considers the architectural design decisions like partitioning, multiple-version dissimilar software or safety monitoring to determine whether they affect S/W level or S/W functionality. Those architectural design decisions will be discussed in more details in para. 3.2.2

2.3 Software Error Effects Analysis (SEEA)

The purpose of the S/W Error Effects Analysis (SEEA) is to evaluate S/W components for potential impacts of S/W failure modes on other design elements, on interfacing components or on functions of the S/W component, especially those that are critical. The major objective of the SEEA is to evaluate the design for error propagation and error effects.

The analysis outcome will be recommendations to the design team in order to minimise the effects of S/W errors and help reducing the probability of failures propagation.

Compared to the SCA which is top-down analysis which considers the S/W components as black-boxes, the SEEA is an additional in-depth analysis that goes in much more details analysing the actual S/W design objects in a bottom-up approach starting with the assumed failure modes of the lowest objects, determining the effects on higher level components which in turn may result in failure modes to be considered for these higher level objects.

2.4 Software Fault Tree Analysis (SFTA)

Fault tree analysis (FTA) is a static deductive failure analysis which provides a method for determining causes of a predefined top-level event (usually a state that is critical from a safety or reliability point of view). The Software fault Tree Analysis (SFTA) has the objective to investigate the S/W design for possible causes of assumed unacceptable states of the S/W product.

The purpose of SFTA is to support and complete the SEEA with respect to failure combinations of design elements.

The basic procedure for the SFTA is to assume that a S/W fault has occurred in the S/W product and then to analyse backward the S/W design to determine a set of events as possible causes. It is important to understand that a fault tree is not a model of all possible causes of system failures. The choice of the top events is important for the final result of the analysis. If it is too generic the analysis becomes unmanageable; if it is too specific then the analysis fails in providing a sufficiently broad view of the S/W system. The fault tree has to be carefully tailored to the chosen top events which correspond to some particular S/W failure modes.

Basis for SFTA is the architectural design and its analysis as performed in the SEEA. The top events must be chosen to support the objective of the SFTA to investigate on critical combinations of failure modes identified in the SEEA for terminal objects.

2.5 Software Verification Process

The purpose of the Software Verification process is to detect and report errors that may have been introduced during the software development process.

Software Verification is an integral process performed concurrently with the software development processes throughout the software life cycle.

The verification activities consist of:

- Reviews and analysis activities, Technical reviews, Walkthrough, SW inspections
- Audits
- Traceability
- Testing activities

Review and analysis activities are basically performed on all products of the S/W Life Cycle:

- Users Requirements Document
- Software Requirements Document
- Architectural - / Detailed Design Document
- Source Code
- Integration process
- Test Cases, Procedures and Results
- Transfer phase and Maintenance

2.5.1 Software Reviews and Analysis

Each development process (e.g. SW Requirements Development Process) can be seen as two concurrent subprocesses:

- a process that creates the life cycle documents/data
- and the Review & Analysis process that verifies the created life cycle data for correctness, for adherence to standards, etc.

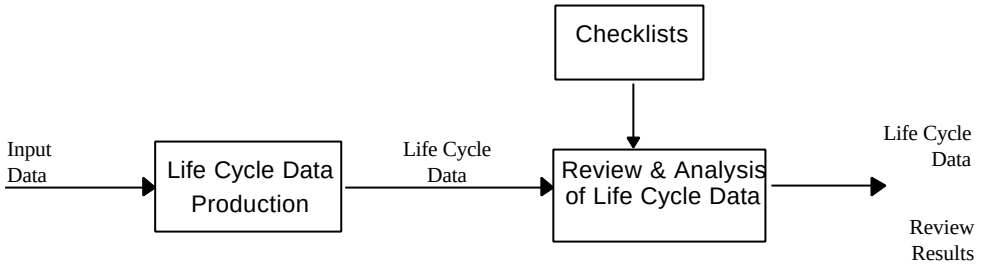


Fig. 2.5.1-1 Structure of Development Processes

One distinction between reviews and a analysis is that analysis provide repeatable evidence of correctness and reviews provide a qualitative assessment of correctness.

- A review consists of an inspection of an output of a process guided by a checklist or similar aid.
- An analysis examines in detail the functionality, performance, traceability and safety implications of a software components, and its relationship to the components within the airborne system or equipment.

Reviews and analyses are thoroughly applied to the results of each software development process and software verification process. The SW verification process is performed on all SW levels but the verification methods are tailored to take into account the assigned SW Category.

2.5.2 Tests

Testing is the process of exercising or evaluating a system or system component, by manual or automated means, to:

- confirm that it satisfies specified requirements
- identify differences between expected and actual results
- demonstrate with a high degree of confidence that errors which could lead to unacceptable failure conditions have been removed.
- demonstrate the correct functioning of the S/W together with the operational procedures in the real environment

2.5.3 Independence

For A and B S/W levels independence is required by the imposed standard. The degree of independence can vary depending upon the criticality of the performed function. The degree of independence can range from subcontracting the whole verification process for a specific S/W application to a third party to having the verification process implemented by different teams from the same organisation, provided independence is demonstrated.

2.6 Tools selection

When talking about the verification process of a complex system the need for using some automatic tool often arises. In a project governed by a standard like the DO-178B the decision of using a tool and its selection process have to be carefully planned.

When a verification tool is going to be used it is important to make a careful assessment of the tool's accuracy, both in terms of its function and its output.

Whilst the process of determining that a tool performs its intended function with the required degree of accuracy may appear to be one that most people could approach with a high degree of confidence, there are some subtleties related to tool performance that may cause unforeseen problems.

It is very important to correctly evaluate the tools functionalities in order to make sure that they actually cover the projects needs. This implies that one of the first processes associated with tool selection should be an assessment of the project objectives, defined in the applicable standard, that are intended to be implemented by using the tool compared with the vendor's description of the tool's functionalities to establish the level of coverage.

Whilst the principle stated above may appear to be an obvious one, the task of comparing the vendor's description of the tool's functionalities against the project objectives can, sometimes, be very difficult if the objectives are not readily understood.

Several vendors supply tools with additional verification functions for which no standard definition exists. These functions are often given names that have generally accepted engineering interpretations but which have no industry standard definition. If a tool is equipped with functions for which there is no standard definition it is unwise to make assumptions about the way in which the function has been implemented. Making such assumptions can lead to two main problems, incorrect interpretation of the tool's output and inadequate supporting procedures for the tool. The interpretation of a tool's output is, necessarily, based upon the user's understanding of the tool's function. If an incorrect assumption has been made about the tool's function the interpretation of its output will be equally flawed. If the output of the tool is being used for a function that requires certification credit this error can have a serious effect on the certification programme.

Another common mistake which can be done when choosing a tool is believing that buying a tool replaces the need to understand how to perform the function the tool has been bought to do. Basically thinking that buying a tool also buys a process is a very risky assumption which can seriously jeopardise the activity.

3. Mitigation Techniques

Some of the possible course of actions useful to reduce the criticality level will be presented. Design and/or development diversification, introduction of consistency checks and "ad-hoc" operational procedures are some examples of what could be done bearing in mind that the right balance between robustness and simplicity of the S/W has to be achieved.

3.1 Why and how to adopt mitigation techniques

After the criticality level identification of a given S/W system or of a component of it, the next step is represented by the adoption of specific mitigation techniques aimed to the reduction of the risk associated to the S/W element. The mitigation techniques can be also adopted with the sole aim to control the criticality and to isolate the potential risk. This can be the only feasible solution for the specific case where it is impossible to completely remove the risk itself.

It is important to underline that the Safety Analysis and the Risk Assessment process outputs may indicate, in addition to the allocation of the S/W levels to each S/W element, the acceptable failure levels for each identified hazard and for each S/W element. It can be the case that for some specific S/W element a Major failure can be accepted if limited by means of barriers to keep the effects of the faults on the rest of the system under pre-defined thresholds.

3.2 What are the possible techniques

3.2.1 General techniques and guidelines

As first set of mitigation techniques we can identify general guidelines and methodologies to be adopted with different emphasis on a case by case basis, they can be summarised by the following list:

1. Applicable Certification requirements and standards
2. Skill and experience of the involved S/W Team
3. Documentation requirements for all the S/W phases
4. Requirements full-traceability through the whole S/W life cycle

For each of these general guideline a brief description is given in the following pages.

Applicable Certification requirements and standards

This approach is adopted each time there is a specific exigency to follow an already define standard for certification purposes. This constraints is usually imposed by the

customer or by the final user of the S/W system or by a third part acting as certification authority for specific application. For this reason the only degree of freedom, in adopting the mitigation techniques stated in the standard, is represented by the identification of the S/W system components for which the standard, or its specific requirements, have to be followed. This exercise can also identify different levels of applicability of the standard itself (e.g. see the S/W levels defined in the DO178B). Each S/W component shall be then associated to a specific level of applicability and to a corresponding "severity" in respecting the standard requirements.

Skill and experience of the involved S/W Team

This is more a mandatory requirement, rather than a guideline, for the team responsible for the safety critical S/W design and development. This requirement includes, obviously, also the Management and QA/PA teams. At least the key persons of the involved team have to be already skilled and trained on the adopted standards and on the safety system development, even if on different areas of application.

Documentation requirements for all the S/W phases

The documentation requirements, even if usually already defined in the adopted standards, have to be carefully managed in order to provide the right level of documentation of the S/W and avoiding, at the same time, a proliferation of documents that can vanish the effort spent in the direction of a complete description and traceability. In addition to the basic S/W documentation, we consider the periodic reporting one of the key elements in the documentation requirements. The periodic reports are the best mean to monitor the progress of the S/W project, they provide evidence on the actual progress and allow to compare it with the predicted/scheduled one. The periodicity of the reporting activities can be tuned depending on the specific peculiarity of the project. A good practice is to adapt the reporting periodicity to the different phases, reducing the reported period when the phase is considered more critical wrt the overall project.

Full-traceability of the Requirements through the whole S/W life cycle

The full traceability of the S/W requirements through the whole S/W life cycle is fundamental. It must be possible to trace each top level requirement (i.e. the user requirement) through the S/W architecture and the test related activities, this traceability can become complex for large S/W systems in which the architecture foresees the breakdown in different hierarchical levels and the tests are performed at different levels corresponding to different steps of a progressive integration and validation. The traceability also allows to allocate requirements to the Software, to the Hardware and partially to the operational procedures to be followed in using the Software/Hardware. This will allow a full verification of the requirements involving, for each of them, the right component/s and test case/s.

3.2.2 Techniques for Specifications and Design

In the frame of the Software specification and design activities, several mitigation techniques can be adopted for safety related S/W components:

1. Constraints on S/W Engineering methodology;
2. Design diversification;
3. Functional redundancy (S/W diversification);
4. Consistency checks functions (safety monitoring);
5. Partitioning of the S/W functions;
6. Constraints on used S/W COTS;
7. Constraints on re-used S/W;
8. Prototyping (to better understand the S/W requirements and to mitigate the development and technical risks, specifically the MMI aspects are a good example for prototyping activities);
9. Specific operational procedures and requirements;

Constraints on S/W Engineering methodology

The design of the safety critical Software has to be performed with specific constraints also on the used Computer Aided Software Engineering (CASE) methodology and corresponding tool. The exigency to have a total visibility of the software structure and a fully predictable behaviour of the software can limit the use of more recent CASE methodology, as the object oriented one.

Design diversification

This is one of the most important decision to be taken in implementing the mitigation techniques during the software design. Some specific system components or software module can be diversified and designed, starting from common specifications, with different architectural solutions in two or more ways in order to minimise the risk of common failures due to the adopted design solution.

In a GNSS system, the reference stations, in charge to collect the GPS, are a good example of system elements to be diversified with different "stations types". The specifications can be common but the design and the development has to be in charge to different companies using different computers, software operating systems and programming languages. This solution allow to minimise the risk of common failures in the different designs.

The aim of diversification is to fight against common mode failures. Two different cases needs to be considered for a GNSS project:

- common mode failures leading to non integrity
- common mode failure leading to non continuity.

Diversification for integrity

In case of equipment for which common mode failure potentially leads to non integrity, the following diversification rules can be applied:

1. They can use the same hosts platform.
2. They can use the same analysis method and tool.
3. They can use the same design method and tool.
4. They can use the same programming language.
5. They **cannot** use the same algorithm.
6. They **cannot** use the same OS and libraries.
7. They **cannot** use the same CPU.

It is important to highlight that these two last items only applies if the two equipments are potentially using the same set of data. The diversification required in algorithms needs to be expressed in specification of algorithms documents. The rationale is that the main driver here is the algorithms but item 6 and 7 are to protect from bugs or flaws which are data dependent (Ex. Pentium floating point operator error).

Diversification for continuity

In case of equipments for which common mode failure potentially leads to non continuity, the following diversification rules can be applied:

1. They can use the same hosts platform
2. They can use the same analysis method and tool
3. They can use the same design method and tool
4. They can use the same programming language
5. They **cannot** use the same algorithm
6. They **cannot** use the same OS or libraries.
7. They can use the same CPU

Item 7 is there because it is considered that a potential simultaneous failure of all CPU of a given type leading to a stop of all computers using this CPU type is impossible. Here again the main driver is the algorithms but item 6 is there to protect against year 2000 or GPS time roll-over problems.

Functional redundancy (S/W diversification)

The diversification can also be restricted to the S/W without involving the Hardware in which the software is installed: In this case we speak about functional redundancy. From this point of view it can be considered a sub-case of the design diversification. The functional redundancy is introduced when the software function is considered critical and the other mitigation techniques are not considered enough to reduce the safety risk. The software function is then at least duplicated (this is the typical case) in order to obtain two pieces of software performing the same task. In particular cases, in which the software uses complex algorithms, the design diversification can imply the use of software performing the same task but using different algorithms. A cross check of the outputs provides evidence of consistency and correctness of the computed data while, in case of failure of one of the software algorithm, the other algorithm is assumed to be not affected by the same bug.

In a GNSS System the functional diversification can be adopted in the processing of the reference station data (inside the Processing facilities) and in the preparation of the

mission performance monitoring data to be provided to the ATC domain (inside the M&C facility).

Consistency checks functions (safety monitoring)

The adoption of the software diversification requires the introduction of a check function in charge of comparing the outputs of the diversified software in order to detected any deviation and discrepancy. The consistency check function can also be adopted for other purposes, independently by the diversification adoption. In specific case could be useful to check input and/or output data in order to detect any inconsistency when it may have impacts on safety. This check function can be a workaround for all the cases in which it is not possible to totally remove the risk. In this specific case can be enough to alert the data "consumer" about the possible data inconsistency without stopping the software. This notification in term of "do not use data" can be sufficient for transient periods without major safety impacts and impacting only the continuity requirements.

Partitioning

The partitioning is a technique for providing isolation, between functionally independent software components, to contain and isolate faults and potential safety risks. The partitioning can force the software designer to isolate a specific software function even if it could take benefit in sharing resources and common services with other software functions. The partitioning is then resulting on a cost increase and for this reason it has to be adopted with special care. The partitioning is usually more adopted for hardware resources when they are totally allocated to critical functions avoiding concurrency and potential overlapping of failures coming from other functions. As an additional major benefit, coming from the adoption of partitioning, it has to be mentioned the verification effort that can be reduced and optimised in focusing on the most severe failure condition category associated with each partitioned component.

A GNSS System can adopt the partitioning especially for the communication software and hardware for which dedicated and isolated resource can be foreseen for the Front-End Equipment in each Ground Segment Facility.

Another example of partitioning can be found inside the M&C Facility. The functions in charge to the Ground Segment M&C and to the Mission Performance Monitoring control have to be totally de-coupled and using separated computers.

Constraints on used S/W COTS

The use of Commercial Off The Shelf (COTS) software in implementing specific function is a good practice in order to reduce the development cost when the market offers already developed software for which only a customisation is required. Typical examples are the Relational Data Base Management System and the Simulation software. This cost reduction can not be achieved if the criticality of the function to be implemented imposes specific constraints on the test, verification and maintenance tasks. The requirement of the total source code coverage can not be met in case of use of S/W COTS. In some specific

cases the S/W COTS can be selected, if available, with special pre-requisite like the already granted certification and qualification as for the S/W system to be developed.

In a GNSS application the use of S/W COTS can be adopted in different ways depending on the safety level allocated to each S/W component. A typical example is the software code to be developed for the Reference Stations, it should be developed with Real-Time oriented coding language like the "ADA" language and with specific compilers already used in other safety critical project. The Ground Segment M&C software, usually more relaxed for Real-time performance requirements and safety levels, can be developed in C language code on UNIX/OS environment and largely using S/W COTS for RDBMS and communications services.

Constraints on re-used S/W

Another constraint, in addition to the S/W COTS use, is the restriction in re-using software already developed on a previous project. The re-use of S/W is obviously a benefit for cost and time saving. It may represent also a risk reduction when the re-used software has been already tested and verified with the same level of severity required for the target software system. Another constraints on the re-use of software is the availability of the expertise related to the performed development, this can not be always feasible (i.e. the company or the team is no more available). When these pre-requisite are not met the re-use of software can not be allowed: the re-used software has to be managed and submitted to the same requirements as for the software developed from scratch.

Prototyping

The prototyping activities are a fundamental technique to start working, in advance, with the most critical software functions. This activity will allow to identify problems as early as possible in order to minimise the risks deriving by potential specifications and/or design errors. The prototyping activity represent an additional cost that has to be spent only when a benefit is identified in term of risk mitigation.

In a GNSS application the prototyping activities can be performed for several and different issues. Ad-hoc facilities can be foreseen for the prototyping and simulation purposes, they are mainly oriented to the processing algorithms used for the navigation message and they can be used also for the tests and validation of the system upgrades. In addition there are specific prototyping activities that can be introduced in order to support the Man Machine Interface (MMI) definition mainly for the Ground Segment M&C and Mission Performance Monitoring functions. These prototyping activities can be performed in order to better understand the MMI requirements and to mitigate the development and technical risks anticipating potential problems for the S/W operability. One aspect which require specific attention is the verification of the correct integration of operational safety procedures adopted and the S/W system.

3.2.3 Techniques for Development

During the development of safety related S/W components the constraints and the requirements are those coming from the previous specification and design phase. Some additional specific indication can be given in the frame of the development environment itself. In particular the coding language choice could be limited in case of safety critical S/W development. This limitation directly implies corresponding limitation on the compiler and debugger and in general on the development environment. Other more detailed requirements are defined in the coding standards adopted for the specific project (e.g. limitation of the set of language constructs that can be used).

In a GNSS application, depending on the different allocated S/W levels, the appropriate programming language must be selected. The standards used for the development of the safety critical software (like DO-178B) recommend or require the use of a language which is well-defined, has validated tools, enables modular programming, has strong checking properties and is clearly readable. For the most critical S/W components, the use of ADA is strongly encouraged because it has properties which make it a natural choice for the development of safety critical systems.

3.2.4 Techniques for Assembling, Integrating and Verification (AIV)

In the frame of the Software AIV activities, these are the main mitigation techniques that can be adopted in doing assembling, integration and verification of safety related S/W components:

1. Code structure verification coverage
2. Representative AIV environment
3. Constraints on AIV tools
4. Operational Procedures verification

Test Coverage of software structure

Depending upon the S/W criticality level different requirements are imposed for the level of structure coverage to be reached. It can vary from the simple *statement coverage* (every statement in the code is invoked at least once), to the *decision coverage* (every point of entry and exit in the program is invoked at least once and every decision in the program takes on all possible outcomes at least once) to, for the most safety critical software, the very hard *modified condition/decision coverage* In this case every point of entry and exit in the program is invoked at least once and every decision in the program takes on all possible outcomes at least once, and each condition in a decision is shown to independently affect that decision outcome.

Due to the heavy workload required for this task automatic COTS tools are usually used, in order to speedup the test activities and to have a better confidence on the results.

Representative AIV environment

One of the fundamental requirements in performing AIV activities for safety related S/W systems is the set-up of the related integration and validation environment with special care for the following aspects aimed to obtaining a test environment as representative as possible:

- extensive use of real data, when possible, rather than simulated for the test scenarios. This is true in particular for the system interfaces with the external world;
- respect of the timing as for the real environment;
- use of real S/W components rather than stubbed interfaces;
- use of the real target H/W platform with the real resource margins and operational constraints;
- execution of tests covering the whole fault tree identified during the system analysis (i.e. considering the whole system with S/W, H/W, operators and the environment).

Specific needs in performing AIV could require "accelerated" tests for all the cases for which the time interval to be investigated covers very long periods (e.g. stress tests for low probability events).

Constraints on AIV tools

The use of AIV simulators, emulators and specific tests tools is largely encouraged for the safety related S/W. The selection and the use of these tools, usually based on S/W COTS, has to be managed with the same approach as for the developed S/W. For safety critical S/W it could be the case that the use of S/W COTS test tools is forbidden or constrained. These restrictions are aimed at avoiding the introduction of not controlled elements in the AIV environment potentially source of wrong and/or misleading data.

Operational Procedures verification

The final system robustness and capability to correctly react to failures and events potentially impacting the safety is verified only when the S/W is totally integrated with the H/W and operated in the real environment and following the defined operational procedures. This implies the need to anticipate as much as possible, in the AIV environment, the use of the operational procedures as defined for the final system in order to anticipate potential conflicts. It has to be underlined that for specific cases the operational procedure could be the only implemented barrier for the safety impacts; this is true for all the cases in which the S/W role has been limited to the failure detection and notification while the operator decision is fundamental for the failure recovery actions.

4. Management Issues

The Management role is fundamental in both making decisions and in monitoring the project progress. There are two main branches:

1. Trade-off Analysis
2. Risk Management

4.1 Trade-off Analysis

The process with which the Management can make the best decision is based on specific trade-offs performed comparing benefit and drawbacks for each potential solution in adopting a safety risk mitigation technique

In the previous paragraph the main mitigation techniques have been discussed. The management role is fundamental in deciding which techniques have to be adopted on a S/W project selecting case by case the "best" solution for the different phases/activities and components of the system. The selection of the "best" solution depends upon the results of a complex trade-off between two fundamental drivers:

- Improve the reliability and robustness of the S/W system or part of it;
- Keep under control the corresponding cost increase, the affected performances and the growing complexity;

As far as the general techniques and guidelines are concerned, there are not specific trade-offs to be performed because they are usually driven by "imposed" requirements and as consequence they have to be adopted as mandatory directives. The management role is, for these general techniques, focused on implementing the requirements in a direction that allows the maximum re-use of the already defined in-house standards and procedures. This approach is fundamental in order to gain the maximum benefit from already consolidated skills available in the company in charge of S/W system development. This "compromise", between imposed requirements and in-house procedures, can be adopted only in the cases for which flexibility exists in implementing the requirements without changing their objectives.

4.1.1 Design and software diversification

The adoption of the functional redundancy, by means of deliberate duplication of functions to improve reliability, provides a mean to avoid common-mode failure(s) while presents a series of drawbacks. In general it is a common practice in hardware design, in some case this technique could be also suitable to reach the same objectives with software, unfortunately this is not always true. As diversity is attempting to avoid common-mode failures (in particular with hardware equipment's), with software this will only work with different implementations (SW diversification).

This is a list of drawbacks deriving by the adoption of SW diversification:

1. ***Costs drawbacks;***
   Costs are to be expected to increase by ratio of N times while the advantages of diversification could turn out to be very little, as discussed under technical aspects below.
   Costs will increase for:
   - development of diversified versions;
   - verification and validation of diversified versions (on top of those activities already required for each version);
   - maintenance and operations of diversified versions;

   Any activity not duplicated (i.e. aimed to reduce the costs of this diversification) could potentially contribute to common-mode error(s), practically vanishing all the effort spent.

2. ***Technical drawbacks;***

   Software design diversity attempts to avoid common-mode failures, therefore to increase the reliability of the software. Against this positive objective we have the following drawbacks:

   - Any shared software specification can lead to common-cause failures;
   - It could be impossible to avoid shared specifications for particular cases (e.g. complex software algorithms);
   - The added complexity of providing fault tolerance in this fashion may itself cause failures. So redundancy, designed to increase reliability, may in fact decrease safety;
   - Although different algorithms could be used by the diversified versions this may not help if the problem resides in the handling of difficult (abnormal) input cases , or logic;
   - If the same software requirement specifications are used, then design diversity could be effective only against coding errors (and only a limited set of those errors). Experiences with fault-tolerant techniques based on software redundancy have shown that programmers often make the same mistake. Therefore independently coded software doesn't necessarily fail independently;
   - Most of the hazards in real time software systems are caused by timing problems. Redundancy will not be of any help in this case;
   - Redundancy appears to be most effective against random failures and less effective against design errors. It is applied to software which -of course - has only design errors. There is not an easy way to determine how different two software designs are in their failure behaviour;
   - Most of the safety problem stem from misunderstandings on system requirements (and the required operation of the software). Any redundancy, then, will simply duplicate these misunderstandings;

4.1.2 Partitioning

The partitioning, as said in the paragraph 3, is another technique aimed to increase the reliability of the system. The following are drawbacks in implementing it:

1 Costs drawbacks;

The following are reasons for cost increase in adopting partitioning:

- the need for additional dedicated H/W resource/s for the "isolated" safety critical S/W function;
- the lost of cost optimisation usually coming from shared S/W common libraries (additional effort in development and AIV);

2 Technical drawbacks;

The main technical drawback of the partitioning technique is the increase of S/W complexity due to the increase of the number of internal interfaces;

The following table summaries the trade-off between benefits and drawbacks in adopting the different mitigation techniques:

Mitigation Technique	Benefit	Drawbacks	Comment
Design and software diversification	• Increase of system and/or S/W reliability;	• Cost increase of a factor N on the specific diversified component; • Increase of complexity; • Potential impacts on performances;	To be adopted only locally and for well isolated system components.
Consistency checks functions	• Increase of S/W reliability;	• Additional cost for the check function development/test; • Impacts on performance;	A good compromise between cost and benefits. It can be used with or without diversification.
Partitioning of the S/W functions	• Isolation of safety related components; • AIV activities facilitated;	• Additional cost for resource/s dedicated to the isolated functions; • Loss of optimisation in common resources;	A very good technique, even if it lessens the optimisation, because it does not introduce additional complexity and functionality's.
Constraints on used S/W COTS or re-used S/W	• Detailed knowledge of all the S/W developed from scratch; • Independence from COTS providers if they are not adopted; • Complete S/W documentation for all the S/W developed from scratch;	• Cost increase due to development of S/W from scratch instead of S/W COTS usage; • Loss of the gained experience in case of no re-use of S/W previously developed for other similar project;	The S/W COTS adoption has to be performed only for well know products with proven quality. The re-use of already developed S/W has to be encouraged only for the case in which no major differences exit between the actual and previous system;
Prototyping and Simulation	• Risk reduction; • Speed-up the Requirements assessment; • Allows to perform accelerated tests;	• Cost for prototypes specification, development, test and use; • Requires to anticipated S/W development; • Not real world but simulated one;	Strongly encouraged for the MMI application and complex processing algorithms prototyping; To be adopted carefully for simulation activities in the frame of system analysis and design due to the impossibility to reproduce 100% the real world;
Specific operational procedures and requirements	• Reduce tasks and functions in charge to the S/W; • S/W simplification; • Clearly isolate the decisions in charge to the S/W from the ones in charge to the operator;	• Increase the probability of human errors; • Need to identify in advance all the required operational procedures;	Strongly recommended for all the cases in which the system operators have a key role in the system operational environment and in particular in solving problems and failures. Moreover on a GNSS system the operational procedures have to be harmonised and constrained by the Aviation Authorities Rules due to the direct interface with ATC domain.

Table 4.1-1: Summary of Mitigation Techniques

4.2 Risk Management

Risk is a measure of the probability and consequence of not achieving a defined program goal. Risk involves also the notion of uncertainty due to a lack of predictability and control over the events.

Risk is introduced by a potential problem situation on a program with unwanted consequence on cost, schedule and technical performance including safety and dependability.

Risk Management is a systematic approach to support program management in the optimisation of program resources with the purpose to *identify, assess, reduce, prioritise, control, document* and *communicate* the risks involved in a program with reference to **cost**, **schedule** and **performance**. Risk Management is not a separate program office activity assigned to a risk management department, but is one aspect of the technical management program.

Risk Management is an iterative process to be applied during the whole life cycle of the program in order to reflect its evolution and to verify the implementation of the risk reduction actions.

The Risk Management Process includes the steps defined below:

1. Definition of the *Risk Management Policy*;
2. *Identification* and *Assessment* of the Risks;
3. *Reduction* of the Risks and/or their *Acceptance*;
4. *Monitoring, Communication* and *Acceptance* of the Risks.

Although there is a logical flow in the process elements defined above, Risk Management is an iterative process with considerable interaction and feedback among the process elements. Initial risk assessment activities may be expanded or re-examined based on the outcome of subsequent activities to define effective risk control measures.

The monitoring of performance may suggest that initial assumptions or data were inadequate and need to be updated or rethought.

5. Conclusions

For Safety critical S/W systems some of the normal engineering practices have to be improved, expanded and constrained by certification and safety requirements. The Management is in charge to adopt specific mitigation techniques and to set-up risk management process, by means of a complex trade-off between cost issues and technical needs, without compromising on the quality and timeliness of the project. In doing this exercise the already gained experience and the already consolidated company standards have to be followed to the maximum possible extension.

It is then necessary to adopt a very careful and pragmatic approach to avoid that the introduction of functionality's specifically devoted to increase the system robustness, actually ends up with a far too complex, difficult to test and to operate S/W system. The winning blend will be the result of a trade off between the different needs.

It is important to point out that the previous experience we had in our company was in the development of S/W for space applications and this was a very useful starting point. The existence, within the company, of a well defined and correctly managed S/W development process is a must. The skills needed to successfully face the challenge of meeting the requirements applicable to a safety critical system, is therefore a very valuable company asset which needs the right level of consideration from the company management. The objective is that the acquired know-how be capitalized on future projects.

Improving DSP Software Process from Testing

Matias Vierimaa / VTT Electronics

NOKIA MOBILE PHONES

Number 1 in Wireless Handsets

World's largest manufacturer of mobile phones

Market leader in wireless data products and

Product portfolio covering all main standards and consumer segments worldwide

Over 100 million phones produced to date

is a leading founder in industry initiatives:

NOKIA
CONNECTING PEOPLE

TECHNICAL RESEARCH CENTER OF FINLAND

VTT Electronics

Research fields:

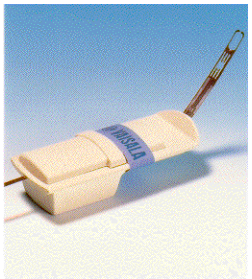
- **Microelectronics**
- **Electronic Circuits and Systems**
- **Embedded Software**
- **Optical Electronics**

Staff: 291 Employees

Turnover: FIM 129 million

Applied research based on industrial needs

Technology transfer



Development of new products for industry

Industrial Consumer Business

- **Volume products with many hard implementation limits**
- **National and international system standards, new features**

DSP Software Development

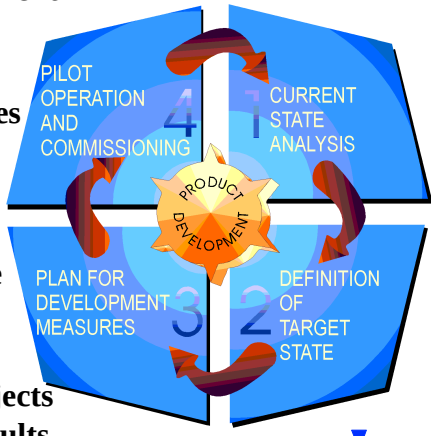
- **Based heavily on source code from previous generations**
- **Implementation mostly in assembly, some C-language**
- **Background of DSP designers in signal processing, software engineering practices not widely known**

Why process improvement?

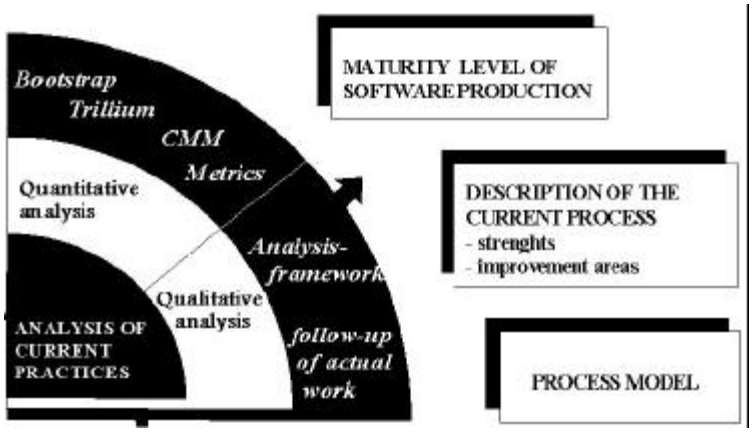
- **High quality software process increases efficiency, quality and**
- **Importance of DSP software has grown**
  - **Size of software has increased**
  - **Software engineering practices are just emerging**
- **We need to know:**
  - **How good the current process/product is ?**
  - **What are the strong points and weaknesses of current practices ?**
  - **What should be improved ? How we should improve ?**

Pr²imer

- **Pr²imer = Practical Process improvement for embedded real-time software**
- **Pr²imer is a systematic and practical approach which uses the best practices in the production of embedded SW.**
- **Fully integrates software process analysis, modelling, improvement and measurement techniques into the TQM based process development of embedded software.**
- **Pr²imer has been used in several projects with several companies with good results**



Analysis of current practices



Analysis results

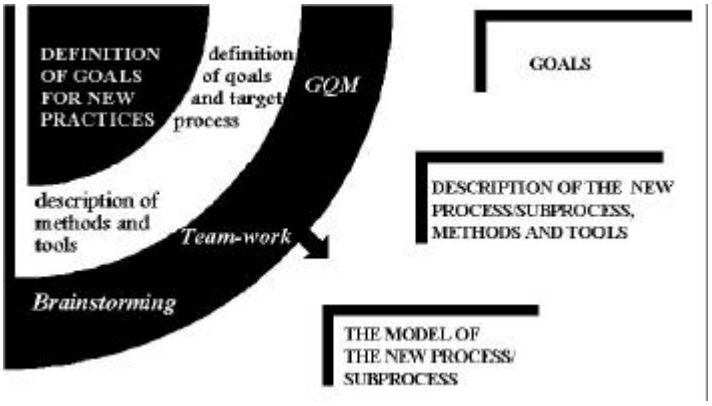
Focused interviews and questionnaires

- **75% of projects covered**
- **25% of DSP designers answered**

Analysis of DSP software development practices

- **Several improvement areas identified in testing**
  - **unit testing, software integration testing, regression testing**
- **Several improvement areas affecting testing**
  - **Specification management**
  - **Design and implementation practices**
  - **Inspections**

Definition of goals



Goals definition

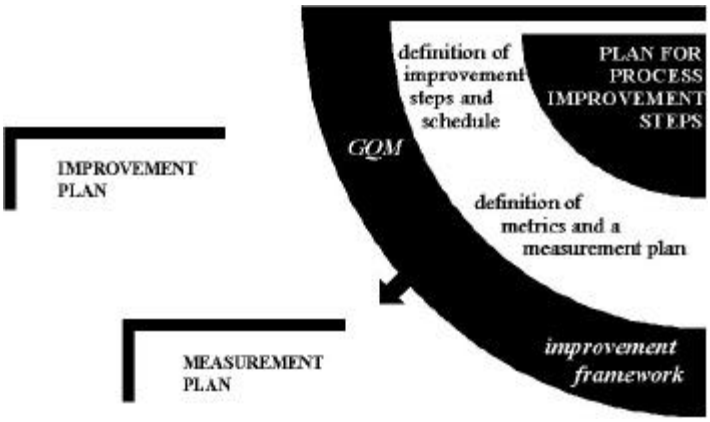
Brainstorming session was arranged

- **Goals were set based on analysis**
- **Several improvement proposals based on goals**

High level goals were set to

- **Improve software documentation and inspections**
- **Improve the efficiency and quality of testing**
- **Improve evolutive aspects of software development including regression testing**

Plan for improvement steps



DSP Characteristics That Affect Testing

- **Interfaces - DSP software is related to several components of mobile phone, on both hardware and software side.**
- **Optimisation - DSP code has tight limits for both memory consumption and execution speed.**
- **Low level implementation - Due to tight requirements, the DSP code has been largely implemented with assembly language.**
- **Code reuse - Currently projects usually base their work on code developed in earlier projects, code not designed to be reused**
- **Dependency on hardware - DSP software testing requires hardware, but this is typically not available in early phases of**

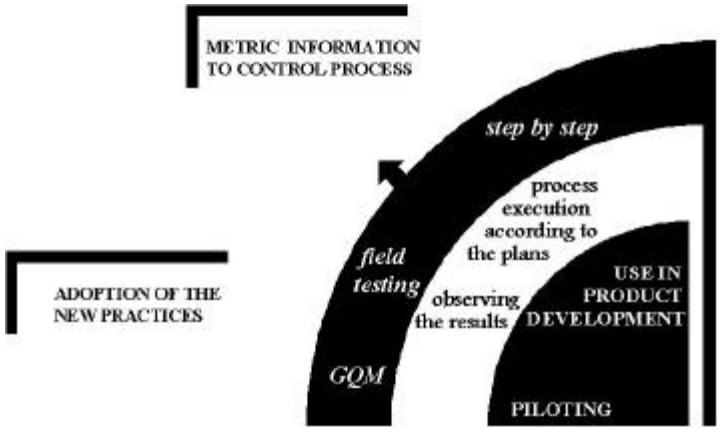
Improvement actions

Software engineering practices taken into use

- **More control to interface specifications**
- **Scattered documentation was collected under version**

- **Unit test documentation stored together with module implementation and design**
- **Inspections were improved to contain focus areas**
- **Stepwise software integration instead of "big-bang" integration**
- **Testing viewpoint - testability, naming practices**

Piloting and use in product development



Piloting

Piloting background

- **Piloting was done in four product development projects**
- **Schedule of product development projects affected which actions**

- **Positive feedback from new documentation and inspection practices - some measurements already available**
- **Testing improvements still on-going, early results promising**
- **Base for continuous improvement has been set**

Lessons learnt

Key success factors for piloting

- **Based on needs originating from the pilot project's designers**
- **Correct timing (synchronized schedules with pilot projects)**
- **Training for the new methods and tools to be piloted**
- **Regular follow-up sessions with the pilot project members**

- **Software engineering practices can be used in DSP development**
- **Testing is improved by improving processes that affect testing**
- **Pilot project resources/schedule changes ==> Piloting delays**
- **Limited scope of improvements better than too many at one time**
- **Long duration needed to measure impacts of actions**

Improving DSP software engineering processes from testing viewpoint

Matias Vierimaa

VTT Electronics
P.O.BOX 1100
FIN-90571 OULU
Finland
Matias.Vierimaa@vtt.fi
Telephone +358 8 551 2444
Fax +358 8 551 2320

Minna Mäkäräinen, Atte Kinnula

Nokia Mobile Phones
P.O.BOX 50
FIN-90571 OULU
Finland
Atte.Kinnula@nmp.nokia.com
Minna.Makarainen@nmp.nokia.com

Keywords: DSP software, software process improvement, software testing, Pr^2imer

Biographies:
M.Sc. Matias Vierimaa works at VTT Electronics, one of the largest European R&D organizations focusing on electronics. Since beginning of 1996, he has been research scientist of embedded software at VTT Electronics. He has written several publications on software process improvement. Mr. Vierimaa has worked in several process improvement projects with both industrial and research partners.

Phil.Lic. Minna Mäkäräinen works at Nokia Mobile phones (NMP) as a senior research engineer. She has worked on the area of software process improvement, with the special emphasis on software change management and maintenance issues. Currently she is doing research on distributed software architectures and mobile code.

M.Sc. Atte Kinnula is senior research engineer at Nokia Mobile Phones (NMP), the world's largest producer of mobile phones and a leading company in telecommunications industry. Since joining the company in 1993 Mr. Kinnula has worked on the area of software process improvement and quality management. From 1997 to August 1998 Mr. Kinnula was the manager of the NMP's global Software Process Improvement Program and is now working on his Ph.D.

Abstract

This paper presents the efforts to improve the DSP (Digital Signal Processing) software testing at Nokia Mobile Phones, Ltd (henceforth NMP). We describe the improvement approach used, together with the intermediate results and give a brief overview of the improvement actions initiated. The focus of this article is on the key issues that affect the testing of DSP software and indicate where and how in the DSP software engineering process the testing issues should be taken into account.

Digital Signal Processing is the application of mathematical operations to discrete-time signals. Signals are represented digitally as sequences of samples that are obtained from the physical analog signals. DSP is central to the operation of modern telecommunication systems, such as modems or digital cellular telephones. [1]

The efforts invested in the development of digital signal processing software are increasing significantly, especially in telecommunication applications. Design of optimised mathematical algorithms has dominated research and product development, but as the size of software is increasing, a need for improved engineering practices for DSP software has emerged. At NMP the Pr^2imer improvement approach has been used to guide the DSP software process improvement activities [2].

1. Introduction

This paper is based on several process improvement projects conducted at the DSP department at NMP R&D. Aim of these projects has been to increase DSP software development quality and efficiency. The projects have been joint efforts between the NMP and VTT Electronics.

NMP is the world's largest manufacturer of digital cellular phones. The company started out in 1987 with some 250 persons in two sites, both in Finland. As a result of subsequent exponential growth it has been growing rapidly and currently has over 12.000 employees and R&D sites all over the world.

VTT Electronics is a leading research center in Finland. Its R&D services are used by the electronics, telecommunications, process automation, mechanical engineering, and instrumentation industries. VTT Electronics offers contract research and development for industrial clients, design, development, and prototyping of electronic products, and consultancy in methods and process development and technology transfer.

The focus of this paper is the improvement of the DSP software testing process. The paper includes description of the overall improvement approach, describes the results of analysis of the testing process and issues that affect testing. We will also describe the key improvement areas and the actions initiated to improve the situation. This paper serves as an example how to conduct DSP software process improvement and captures experiences which we think are useful for organisations facing similar challenges.

2. Background

The importance of software as a value adding element and as a means to provide features and services to the user has been rapidly increasing throughout the 1990's. Today it can be said that software is a major factor, if not the deciding factor in the cellular phone markets. The majority of the R&D effort is, as of today, in software.

The importance of software was realised in NMP early on. Ever since the company started expanding beyond the first R&D site, there has been a dedicated ownership for 'global', i.e. company-wide software technology issues, processes included. Activities to improve processes have been ongoing since late 1980's and the R&D sites started introducing the ISO 9000 quality approach in early 1990's. All main R&D sites are currently ISO 9000 certified. Those that do not have only recently been established or acquired and are in process of getting one.

The NMP software development has been divided into different software development units within each R&D site. The reason for this is historical and stems from the fact that the importance of software over hardware has been a gradual change. Hence the software development has been divided according to the hardware platforms where the software is embedded to. The bulk of the software resides in Main Controller Unit (MCU). This is also the area where the growth of software started earliest and has been traditionally considered as *the* software of the cellular phone. It is also the furthest away from the hardware dependency and today it can be justifiably argued whether some parts of it is 'embedded' software as such any more. The MCU software provides the functionality and takes care of e.g. user interface, network interface and so forth. It is usually implemented with either C or object-oriented language.

The other major software part is the software for Digital Signal Processing (DSP) purposes. This part of the software has been very hardware-dependent, has been traditionally programmed with low-level (assembly) language, and the volume (code-wise) has only recently started to grow. However, the growth in this area is accelerating and the implementation is moving towards higher-level languages. Subsequently in a past few years the visibility and impact of the DSP software has increased to the point where it is considered a major factor of its own right.

Since the MCU software has been the main focus for NMP software effort, it is understandable that in the past the improvement activities have also been focusing on the MCU software development rather than DSP software. The need to improve the DSP software processes has only recently been realised. As a result the two areas are at very different levels as far as practices and processes are concerned.

When the case for DSP software process improvement became obvious it was decided that the differences between these two areas were sufficiently different to warrant two separate improvement programs, the other which was already ongoing for MCU software, and another to be initiated for DSP software. The differences were in the degree of process understanding, the product itself and in the fact that the DSP software development organisation was independent from MCU software organisation. For these reasons, at that point the merging or aligning of the two improvement programs would have been a major effort both organisationally and process-wise and the added value was in doubt because the differences in process maturity would still have required two distinctive set of improvement activities.

Despite of this the general policies for process improvement apply to both programs. These policies state that the process assets are owned globally by a nominated process owner, but the responsibility for process improvement and support activities is local, i.e. resides at the R&D site level. Furthermore the approach gives local level a freedom of choice as to what kind of process improvement methodology is being used or applied. Although the NMP has chosen the Capability Maturity Model for Software (henceforth SW-CMM) as a common metric for process maturity, it does not require the use of SW-CMM as a method for guiding improvement actions. In fact the NMP uses a portfolio approach e.g. for assessing software processes. The philosophy behind this is that certain methods are more suited to different situations than others; for instance for software development units that have no prior exposure to or experience in process improvement full-scale CBA-IPI assessment may be too heavy and the results discouraging rather than encouraging. In such cases a less formal method that has a higher probability for buy-in is needed to establish a positive loop for process improvement.

3. Pr^{2}imer method

VTT Electronics has used Pr2imer [3] (Practical Process Improvement for Embedded Real-time software) service package with good results with several customers. It is available in several tailorable versions for different purposes. It is used to improve the processes in developing embedded systems applications. Pr2imer framework has four phases (analysis of the current practices, definition of goals for improved practices, plan for process improvement steps, and piloting and use in product development).

Pr^{2}imer represents a practical and systematic approach for improving the quality of the software engineering process. It fully integrates software process analysis, modelling, improvement and measurement techniques in order to meet the critical quality requirements for the development of embedded

systems. Pr^{2}imer addresses software process quality improvement by supporting the following tasks (figure 1): [3]

1. Quantitative and/or qualitative analysis of current software engineering practices
2. Definition of measurable goals for improving those practices
3. Planning of successive and practical process improvement steps
4. Piloting and trial use of the improved practices in product development projects.

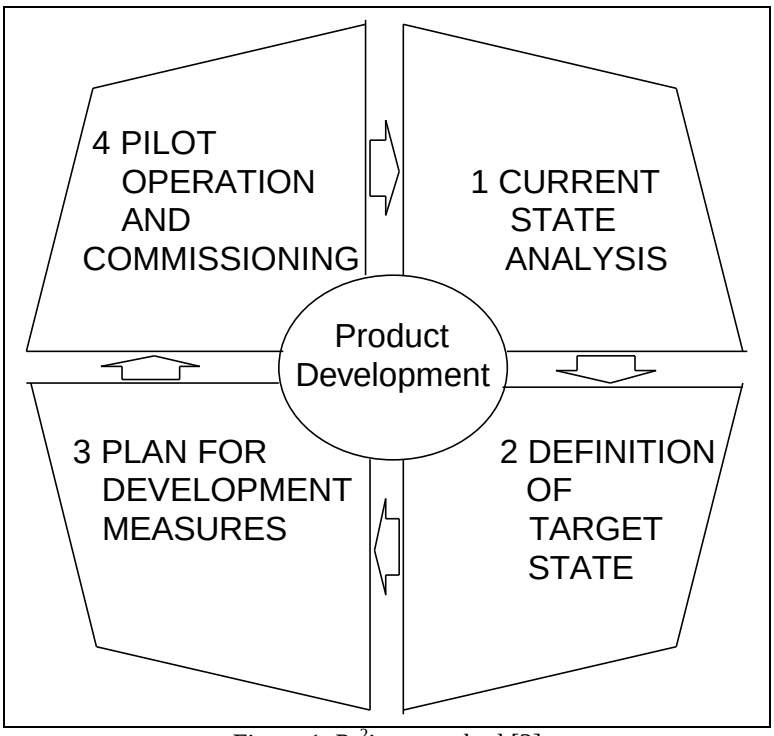


Figure 1. Pr^{2}imer method [3]

The first phase of the Pr^{2}imer method is current state analysis. In our case it was carried out by using focused interviews. Several key persons of the DSP software development were interviewed and a questionnaire was sent to DSP designers in each of the NMP R&D sites.

In the second phase of the Pr^{2}imer method the target state is defined. In our case a brainstorming session was used to identify process improvement candidates. Improvement suggestions were reviewed and later divided to practical improvement areas. Testing was identified as one of the key improvement areas. Improvement goals were set and decided to be tracked with metrics. The measurements were agreed to be defined with the Goal/Question/Metrics (GQM) software quality measurement paradigm [4].

In the third phase, stepwise software process improvement actions were planned; each improvement area including testing was divided into more practical and easily understandable actions. Improvement actions are described in chapter 6. Improving only the testing process itself was considered to be insufficient, as some of the problems in testing stem from other parts of the engineering process and can be best tackled there. For this reason, other improvements that had an impact to testing are also described in this paper.

The pilot operation and commissioning is the fourth phase of the Pr^{2}imer method. This crucial phase includes the piloting of improved practices in real projects. If the results from a piloting phase indicate improvement, the revised practices can be institutionalised within the organisation. Measurement of improvements is important in order to understand the impacts of the changes.

4. Pr^{2}imer step 1: Analysis of current practices

In the first Pr^{2}imer step, analysis of current practices was done. The improvement areas for the DSP software engineering were identified by using focused interviews and questionnaires. The focus of the interviews was the entire DSP software development process, which resulted in a wide scope of improvement areas. The coverage of the interviews was 3/4 of the projects (1/4 of the designers) from the R&D site responsible for piloting improvement actions. Approximately 40% of the DSP designers answered to the questionnaire. As a result, the strengths and weaknesses of the current process became known. Pr^{2}imer step 1 structure is described in figure 2:

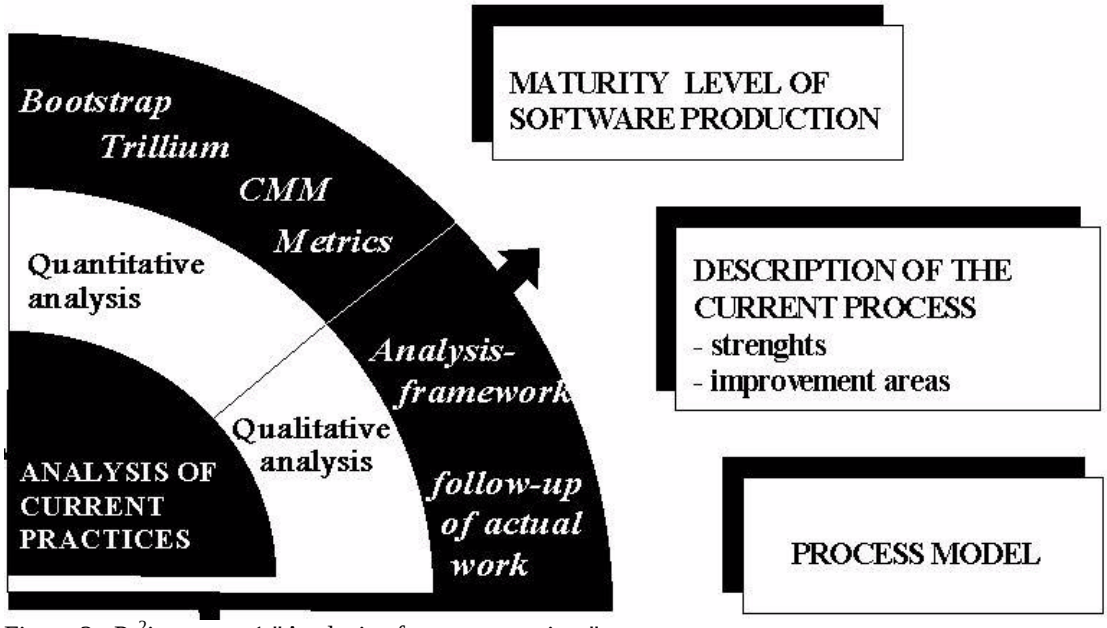


Figure 2: Pr^{2}imer step 1 "Analysis of current practices"

The analysis of the interview and survey results was done using the qualitative analysis. Several strengths and improvement areas were identified as a result of the analysis. The level of detail varied a lot; some identified improvements allowed immediate implementation while others would have required launching an extensive improvement project.

The following improvement areas were recognised to have an impact on testing :

- Specification management. Specifications did not have clear responsibilities, change control needed improvement.
- Design and implementation quality. Some of the documents were not up-to-date at end of project, documents were carried over from previous projects, but they were not designed to be reusable. Documentation was scattered to several places.
- Inspection practices. Current practices were good in DSP specific issues, but needed improvement in quality aspects such as maintainability and testability.
- Testing practices. Especially unit tests and testing support needed improvement. The following testing related problems were identified:
    - No common unit testing procedures. The unit testing procedures were found to be very person dependent. Sharing of best practices and creation of common tools and procedures for unit testing were considered to make unit testing more systematic and efficient. Guidelines and tool support were also needed for unit testing.
    - The separation between software and hardware integration testing was not clear enough
    - Lack of regression testing support. The evolutive nature of the DSP software development emphasises the role of software regression testing. More guidelines and tool support for automating regression test case management and result analysis was sought.

5. Pr^{2}imer step 2: Definition of goals for new practices

In the second step of the Pr^{2}imer improvement cycle the goals for DSP software development were defined by using a brainstorming technique. The figure 3 below describes the step 2:

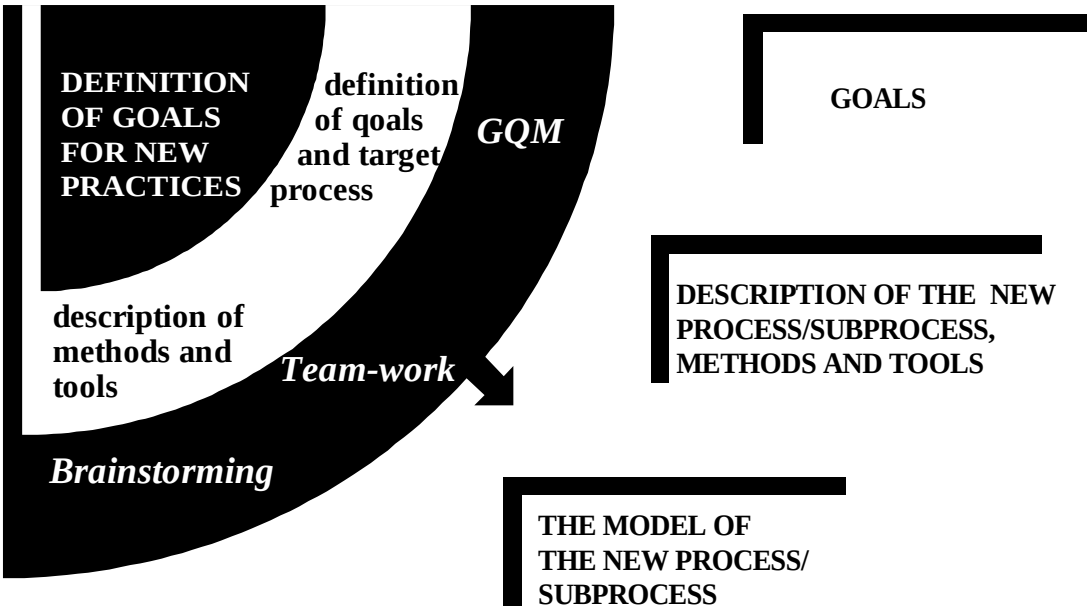


Figure 3: Pr^{2}imer step 2 "Definition of goals for new practices"

As a result of the brainstorming session, several improvement areas with improvement goals were identified and proposals of improvement actions were defined. The higher level improvement goals related to testing were agreed to be:

- Improve software documentation and inspections
- Improve the efficiency and quality of testing
- Improve evolutive aspects of software development including regression testing

The testing goal for improving the testing process was set to improve the efficiency and quality by focusing both on processes that have largest impact on testing and to testing processes themselves. The following testing related improvement actions were also proposed during the brainstorming session.

- New inspection practices including focus areas and code walkthroughs
- New documentation practices to unit testing including unit test inspections
- Improvement of the DSP SW integration testing
- Improvement of regression testing
- Identification of issues that affect testing
- Error data analysis

The software designers opinion was that the DSP code inspections were useful and necessary, but inefficient. Time was wasted in the inspection meetings to read and analyse the material, since the preparation time used by the participants had not been sufficient. There was also a tendency to focus to certain elements of DSP software development such as optimisation, often neglecting other elements of quality. A better coverage of different quality aspects would be ensured by defining focus areas. Code walkthroughs were suggested as a practice to complement code inspections. Such walkthroughs would require less people to analyse the code, and it would allow the reviewed pieces of the source code to be examined to be smaller. Code walkthroughs could be arranged for work products under development, and code inspections would be used to inspect completed work products.

Several improvements in documentation were proposed. The problems identified in the unit testing documentation were decided to be tackled by putting more emphasis on the unit testing practices. The aim was to provide general rules for unit test documentation and unit test inspections that would then guide the projects and designers towards more coherent unit testing practices. Code walkthroughs would be used in detailed examination of the source code, and the source code inspections would concentrate more on inspecting the unit test plans and results. Inspection of unit test documentation would also force the designers to document their unit tests.

The DSP integration testing was very time consuming. The process needed restructuring and better support. Also, means for stepwise integration should be studied, since the current 'big-bang' integration of all product components was found to be problematic.

The errors found in various testing phases were already recorded by the DSP SW projects, but the data was not analysed for improvement purposes. The error data could provide valuable information about the root causes of the errors, how the errors could be avoided in the future, etc.

There was no support for regression testing in DSP SW development. The possibilities for automating regression testing tasks especially during integration testing were recommended to be studied. Besides improving the processes directly related to testing, testing issues should be addressed and emphasised as an important quality factor thorough all project lifecycle phases. Testing issues should be included in all working instructions and guidelines.

6. Pr^{2}imer step 3: Plan for process improvement steps

Pr^{2}imer step 2 identified improvement areas and proposed improvement actions. Testing practices were further examined with additional interviews in the beginning of step 3. The interviews were arranged with the experienced testing experts. Our goal was to identify main issues affecting testing and current practices related to testing for more details. Two interview sessions were arranged and the total of six designers were interviewed. Following section describes the results of the interviews.

DSP Characteristics That Affect Testing

Although testing the DSP software is in many ways similar to "standard" embedded software testing, there are also certain DSP software -specific characteristics that have an impact on testing. The following list includes issues specially related to DSP software testing:

- Interfaces – DSP software is related to several components of mobile phone, on both hardware and software side. For this reason the requirements for the DSP software are extremely volatile, especially in a concurrent development environment. Software/hardware interfaces may change during product development project, thus making the testing more difficult.
- Optimisation – DSP code has tight limits for both memory consumption and execution speed. Significant degree of optimisation is needed to meet these requirements. Optimisation often has negative effect to the quality of documentation.
- Low level implementation – Due to the memory and speed requirements, the DSP code has been largely implemented with assembly language. This limits the number of potential testing environments, and sets additional requirements to code documentation to improve understandability. However, as DSP processors are becoming faster, it is possible to use more and more C-code in software.
- Code reuse - Currently projects usually base their work on code that has been developed in earlier projects, and change it to meet the requirements of the new product. Although the code is reused, it is not often designed to be reusable, which creates additional problems for testing.

- Dependency on hardware – DSP software testing requires hardware, but this is typically not available in early phases of testing as the hardware is being developed concurrently with the software. This may result to a situation where some units of the software has to be initially integrated without proper unit testing.

Defined improvement steps

Earlier on during current state analysis, current state of testing processes was identified, and its strengths and weaknesses were known. As a result of brainstorming session, improvement goals and proposals had been defined. Additional interviews gave further detail about the issues that affected the testing most. In Pr^{2}imer step 3 (see figure 4 below), the stepwise improvement actions and schedule for improvements was defined with more detail.

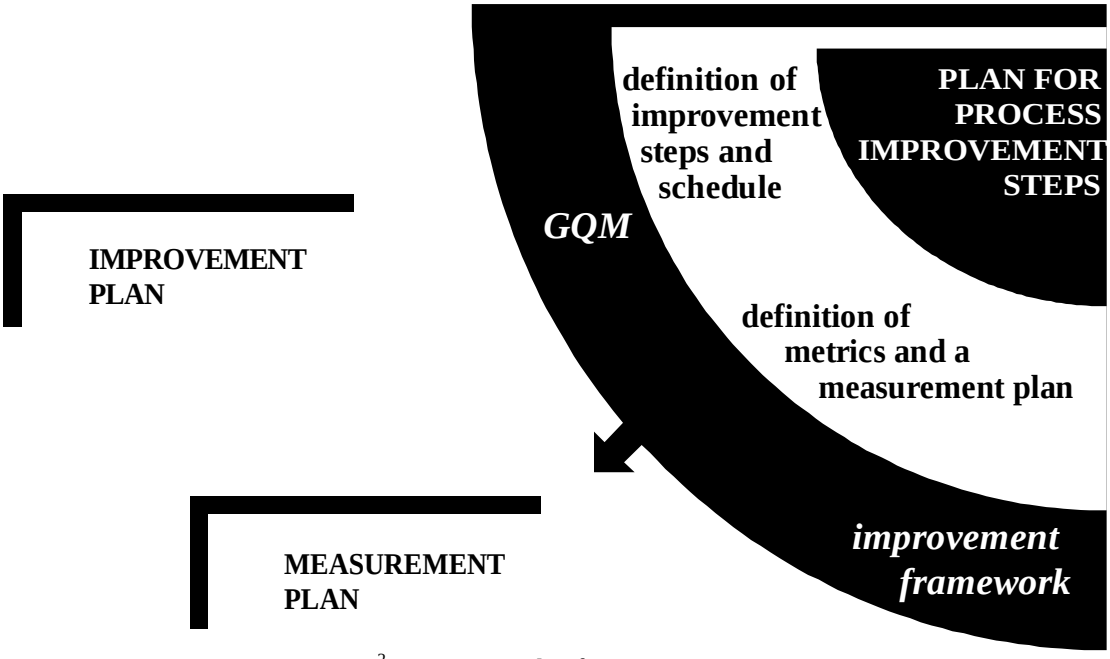


Figure 4: Pr^{2}imer step 3 "Plan for process improvement steps"

The results from previous steps indicated that DSP specific issues were at reasonable level, but general software engineering procedures and software engineering "best practices" were not widely used. Main improvements were therefore set to improve those elements. Several process improvement goals were set as already described. To meet the goals, stepwise improvement actions were defined. This section describes the actions that had an impact on testing, including improvement actions in specification management, design and implementation, inspections and testing itself.

One of the improvement actions was improved control of interface management specifications. As the interface specifications require co-operation with the group working on the other side of the interface, this issue cannot be solved by improving DSP software practices alone. Therefore, experienced resources were selected to improve interface specification documents. Each interface was given a responsible person who in turn would take care of co-operation between the different groups and also to guarantee that the interfaces in his control are stable and available as early as possible.

Quality issues in design and implementation were seen as one of the most important improvement areas. Due to amount of work required, a separate project which provided necessary effort and commissioning for the improvement actions was started [2]. Its main goal was to support documentation reusability. New tools and methods for documentation were selected. Previously scattered documentation was collected and put

under version management. Unit test documents including test plans, data and results were stored together with design and implementation of the module.

Low level implementation and optimisation at late stage of project made the code documents hard to read. Testing viewpoint was seldom considered during implementation. Focus was set to describe all the modules at design level. Code quality issues such as testability issues and naming practices were studied and taken into use.

Defects found during the inspections cost less than defects found during the testing. The DSP designers considered current inspection practices to be at adequate level, but inspections were heavily focused on functionality and optimisation. This led to over-emphasis of hardware resource optimisations (memory, speed, power) based on reading assembly source code line-by-line. Certain quality elements, such as testability or maintainability were often neglected. Inspection methods [5] were studied to improve current inspection practices. As a result, several focus areas including testability were defined. In order to assure that no aspect has been overlooked, each focus area has a specific checklists which provide consistent acceptance criteria.

In testing processes itself, improvements were set to unit testing, software integration testing and regression testing. Improvements in unit testing were focused in process and documentation, although some changes were also done to testing tools. Improvements included better visibility of test results including test parameters and data. This can be achieved by storing testing documentation into defined location, as in our case, under version management tool. The unit test procedures and guidelines needed to be improved to make unit testing more systematic and efficient. New documentation process included also unit testing documentation.

Hardware dependency and changes in interfaces limit the unit testing coverage. This lead to situation where integration of software modules was done in one single step. To divide integration into smaller steps, instructions for stepwise software integration were defined. Software integration testing was set to test module interfaces and functionality as much as possible without the need for hardware. Meetings with testing experts from previous projects were proposed to ensure necessary knowledge transfer.

Regression testing improvements were designed to better match the evolutive aspects of DSP software development. Main goal of these improvements was to create an environment which would support regression testing. During schedule planning, it became clear that improving the regression testing would require significant resources which were not then available. Therefore, improvements were limited to creating regression testing instructions for DSP designers.

Error data analysis was proposed to improve testing. If more information about the causes of errors, and their origin would be available, it would be possible to identify likely problems before testing phase in future projects.

Measurements were also defined to follow improvement actions using GQM approach. GQM measurements were defined with interviews of experienced DSP experts. Measurement goals included reliability, documentation reusability and inspection efficiency. For example, following measurements were defined for each defect/error:

- Phase when found
- Detection activity
- Origin
- Severity
- Status

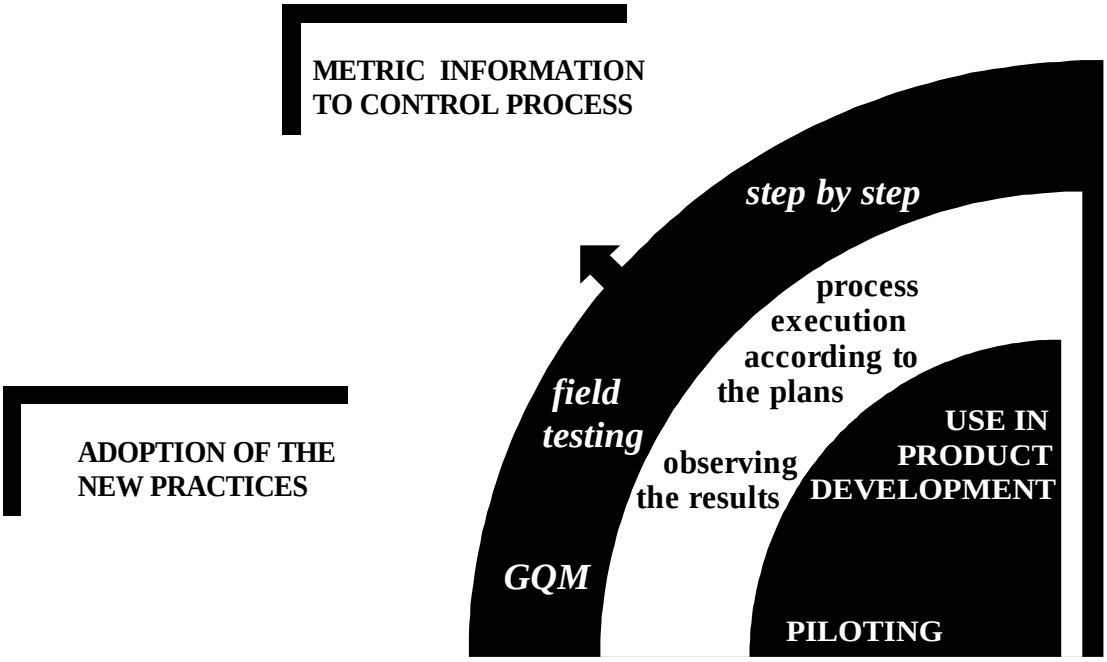
Measurements were defined for each testing phase and test cases. In addition to those measurements, we also measured project effort to find out how much effort is used to different tasks of project including specifications, design, implementation and testing. Our goal was continuous improvement based on measurements and improvement actions already initiated.

7. Pr^{2}imer step 4: Pilot operation and commissioning

Pr^{2}imer step 4, piloting, is described shortly in this paper, because measurements in pilot projects are still on-going. For example, data from testing measurements are not yet available. However, results of the improvement actions with some measurements and expected impacts can be described.

Piloting of improvements was done in several product development projects. As these projects were in different phase of their life-cycle, we divided the piloting of improvement actions between the projects. Projects that were on-going piloted only those improvement actions that had an effect to their work. Some preliminary measurements were collected from the pilot projects as baseline data.

Several improvement goals with many improvement actions led to situation were we had to define priorities. As piloting was done in product development projects with tight schedules and limited resources, process improvement actions had to be limited in size and planned carefully with project managers. Some of the improvement actions originally planned were too optimistic and heavy to implement during project work and therefore they were delayed or limited in scale. However, we were able to achieve several improvements even with fewer resources than planned. Following figure 5 describes the piloting phase:



See figure 5: Primer step 4 "Pilot operation and commissioning"

From testing viewpoint, results of the improvements in documentation process and inspection practices were the most significant. Results of the improvements included new methods and tools for documentation. Our goal is to store all documentation from software process under version control thus achieving better reusability. As a consequence, the availability of project documentation including unit test documentation was improved. Following measurements regarding documentation availability were done in the middle of a pilot project, figures will be further improved as the pilot project matures:

- Implementation, 68% available in PCMS (end of previous project 70%)
- Test results, 43 % available in PCMS (end of previous project 0%)
- Test data 55% available in PCMS, (end of previous project 0%)

New inspection practices based on focus areas have been well-received by the pilot project. The feedback has been positive. New checklists and roles used in inspections create efficiency and discipline to inspections. Code walkthroughs as proposed by step 2 have not been widely used, however. With enhanced inspection practices and new focus areas, we expect to find more defects already during the inspections. Following efficiency measurements have been made:

Table 1. Example measurements of specification inspections [2]

Metric	In pilot project	In reference project
Hours/Defect	2,7	4,6
Hours/Participant	2,6	Not measured
Hours/Document	16,4	Not measured
Average number of participants	6	6

Improved efficiency of inspections should indicate that less errors will be found during testing. We expect that measurements indicate further improvements as the pilot project matures. Better availability of test data and results yield for better testing baseline in forthcoming projects when they inherit the work done in previous project. True impact of improvements can only be measured in forthcoming projects, but we already expect significant rise in reusability of testing documentation thus making the schedule of testing shorter and effort smaller in forthcoming projects.

Improvements in interface management have been on-going in several projects. Defined responsibilities and interface stability have been improved. Moreover, importance of interface specifications is known, and interfaces are discussed with different groups with necessary detail.

Unit testing is considered to be an important part of module documentation, and as our measurements indicated, its visibility has increased. Unit test inspections assure that unit tests are properly done and that unit test documentation is an integral part of module documentation thus allowing forthcoming projects receive all documentation and results related to a module. Measurements for unit test efficiency are currently being defined with more detail.

Stepwise software integration practices are currently used instead of "big-bang" integration approach. Software integration tests cover the interfaces of the modules and functionality of the software. Although it is not possible to test everything due to hardware dependency, more and more software functionality can be tested during this phase. Testing measurements have not been collected, so impacts of the software integration testing improvements cannot be measured yet, but new practices should nevertheless improve the overall testing.

New work instructions for unit testing, software integration testing and regression testing give better support for DSP designers during testing. Work instructions alone cannot guarantee that they are actually used, therefore quality control in testing and proper training of all process changes must be conducted.

8. Conclusions

This paper has identified several issues that affect DSP software testing indicating how DSP software process can be systematically improved from the testing viewpoint. We have also demonstrated, how it is possible to improve DSP software testing, by using existing software engineering "best practices", taking into consideration DSP specific issues and not forgetting practical needs of the pilot project.

The work we have described has shown how the DSP software process can be improved using systematic approach. We hope that our experiences are beneficial to other companies facing similar challenges. The strengths and improvement areas in organisations may be different, but our work could provide example how to improve DSP software processes in general.

We would like to thank all people who participated to work described here in Nokia Mobila Phones and VTT Electronics. Special thanks for Hannu Honka and Juho Näsi for reviewing the paper.

9. References

[1] Marven C., "Ewers G. "*A simple approach to Digital Signal Processing*", Texas Instruments, 1994.
[2] Vierimaa, M., Kaikkonen T., Oivo M., Moberg M. "*Experiences of practical process improvement*", Embedded Systems Programming Europe, Miller Freeman, November 1998, Volume 2, No 13.
[3] Karjalainen J., Mäkäräinen M., Komi-Sirviö S., Seppänen V., "*Practical process improvement for embedded real-time software*", Quality Engineering, 1996, Vol. 8, no 4, pp. 565-573.
[4] Basili V., Caldiera G., Rombach D. "*Goal Question Metric Paradigm*". In John J. Marciniak, editor, Encyclopedia of Software Engineering, volume 1, pp. 528-532. John Wiley & Sons, 1994.
[5] Gilb T., Graham D. "*Software Inspection*", Addison-Wesley Publishing Co, Reading, MA, 1993.

Industrial Experiences in Establishing Laboratories and Software Models to Effectively Execute Software Test

Jon D. Hagar
Lockheed Martin - Denver
jon.d.hagar@lmco.com

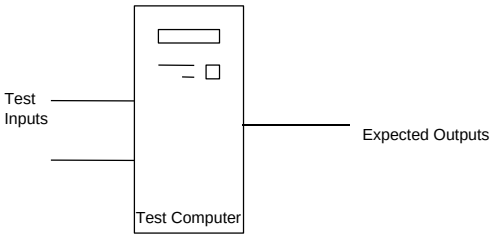
Lockheed Martin

Topics

- Problem statement
- Definition of a lab
- How to establish one
- Definition of software models
- How do labs support testing
- Experiences and types of errors found
- Lessons Learned

Lockheed Martin

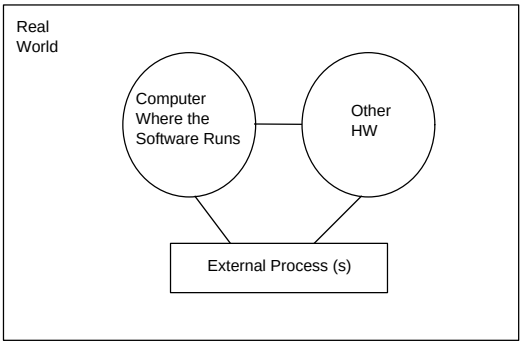
A Problem - Testing Environments



- How We Test Is Not Always How The Software Gets Used

Lockheed Martin

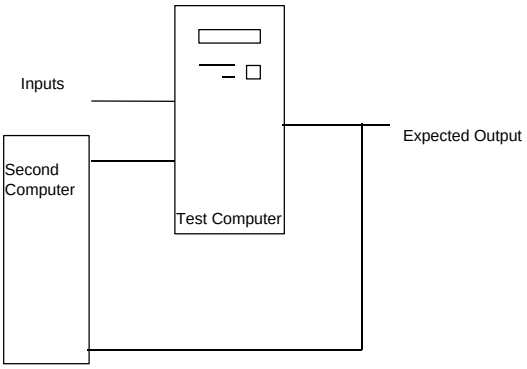
The Real World



- Computers are connected to the real world
  - hardware, software, time effects and process can be unknown

Lockheed Martin

Modeling Offers Some Help



- In many cases a test lab must be established with other computers that run models

Lockheed Martin

Test Labs

- Lockheed Martin has for years tested complex software to high levels of reliability by creating test environments (labs)
  - A Lab is a collection of computers and software supporting test of the software
    - Test Tools, Analysis, Procedures, Test Staff, Metrics
  - Philosophy: “Fly what you test” and “Test like you fly”

How to establish a lab

- Test labs range from single PCs to full hardware “mock ups”
  Define and allocate resources
  - People to develop/run the lab
  - Computer, hardware, and software to support lab development
  - Time and schedule (AKA money)
  - Test process and supporting management
- Successful and cost effective testing using good labs lead to higher CMM levels and ISO certification
  - Controlled facilities, integrated with development, process driven, managed and measured, defect removal based, optimized

Lockheed Martin

Labs have limitations

- Not all hardware, e.g., global telecom networks or rocket
  Not the real environment, e.g., space, 1000s of users or net

  Not the real use process

  yes use-case help, but the real users do things unexpected by test plans, spec, use-cases, procedures, inputs, and timing/sequencing

  Modeling can improve the capabilities of a lab

Lockheed Martin

Definition of what software models

- General Categories: **Discrete** and **Continuous**
  - Simulations -- RDD100 Matlab, Simulink, MatrixX - limited time

  Continuous -- functions in real time (like the real thing)
  - Complex and custom built
- Range
  "testers" mental model <-> software tool based <-> hardware-software tool based

use-cases
simulations
CASE tools

Special built equipment

Lockheed Martin

Models to support testing

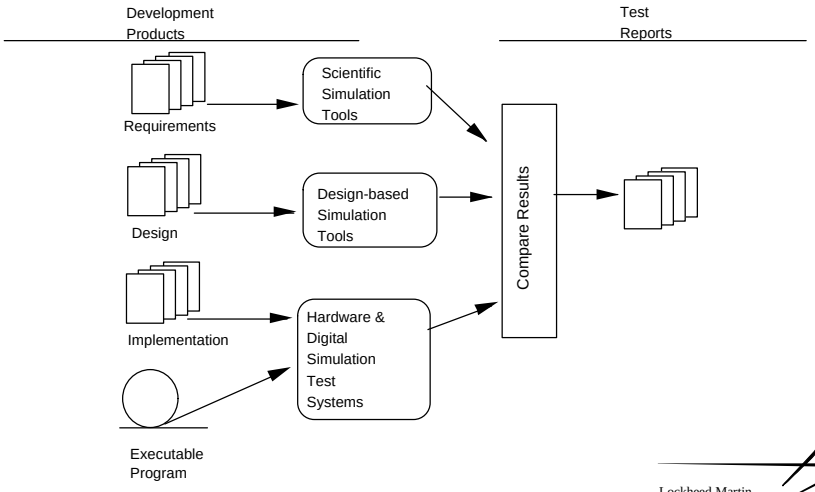
- Allow automation of defined test process
  - 6D closed loop simulation of launch vehicle
  - Simulation of hundreds of network system inputs (or users)
  - Simulation of hardware

  Allow generation of inputs
  - Run hundreds of monte carlo simulations of code (e.g., guidance) to find input conditions for execution in full test lab
  - Generation of OO or SA model based inputs
  - Requirement based for qualification or acceptance testing like ISO/IEEE requires

Risk: Certification and proof that models represent reality becomes a big problem
But the testing of the test tools often helps test the product (iterative problem)

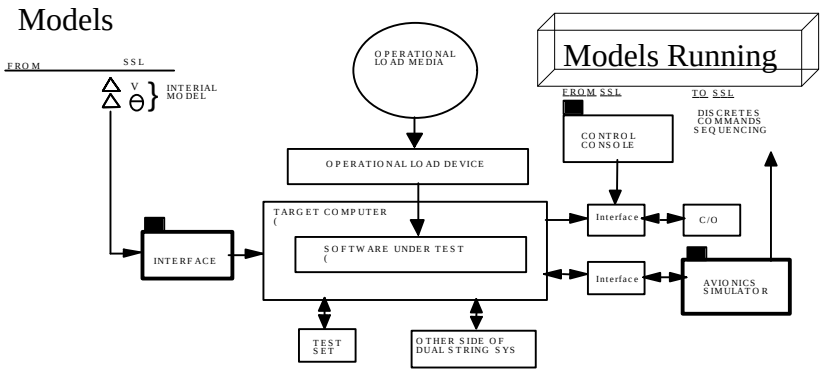
Lockheed Martin

Multiple Levels of Modeling Possible



Lockheed Martin

Sample Lab Configuration



SSL = SGI Computer Running Real-Time Closed Simulation Models
C/O = Check Out Computer Station

Interfaces - either hardware or software

Lockheed Martin

Experience and types of error found

- Programming errors - good to use tools such as battlemap, STW, IPL products, SAW, T, or logic analyzers are in use

  Design errors - logic or software interaction problems

  Requirement/functional error - missing, incomplete, poor wording, and wrong

  Interface/compatibility errors
  - Lab testing is done at integrated product level (qual/acceptance test)
- Lab and Lab Results Benefit for Peer Review - Total independence of testers not necessary, but can have some benefit

  Labs and Models can produce large amounts of data
  - Save it
  - Analyze it
  - Automate when feasible

Lockheed Martin

Lessons Learned

- Labs cost money (and more if not done right)

  Labs are a major test tool in knowing when you are ready to deliver

  Models can support testing but they are “real”

  Errors in models/labs hide real product errors

  Errors in models/labs can create “wild goose” chases

  Have trained people to create and run the lab

  Use commercial products and open architecture

  Have a process, find complementary tools for repetitive or risky parts

  Change is given, plan for it in hardware and software

  Be driven by plans and procedures

  Do realistic complete mission or use-case testing

Lockheed Martin

Summary

- Testing in many cases requires a lab
  Modeling can directly support labs
  - Simulate that which is hard to create
    Many levels of sophistication

Lockheed Martin

Industrial Experiences in Establishing Laboratories and Software Models to Effectively Execute Software Test

Jon Hagar

Lockheed Martin Astronautics Company

Mail Stop H0512

P.O. Box 179

Denver, CO 80201

303-977-1625

fax 303-977-1472

jon.d.hagar@lmco.com

Keywords: *Integrated Test Lab Environment, Test Tools, Software Modeling to Support Test*

Introduction

In recent years, there has been more interest in software testing accomplished with models as well as creating test environments and tools to realistically test the software. There has also been an increase in the number and types of support programs or tools available from commercial suppliers. ISO and IEEE standards require testing in realistic environments. Developers create software-based systems that must function in real-time environments, function directly with complex hardware systems, or add high risk/critical functions for their users. These types of systems tend to require the highest levels of quality, test, and reliability. Organizations within Lockheed Martin have many years of experience creating test facilities for these types of systems, as well as models to help define, run, and analyze tests.

Examples of the kinds of critical/high risk systems where models are used with a laboratory to do testing include: automotive sub-systems, nuclear and power system applications, aerospace avionics, medical devices, telecommunications, and factory or industrial-based systems. These systems share many characteristics and can all involve large financial consequences when they fail. Additionally, many practices, such as "beta testing", and concepts associated with "shrink wrap" development are not appropriate for these systems. And, many of these systems have regulatory agencies that have compliance standards, as well as issues with things such as Y2K. This paper reports on the creation of a test system that directly supports testing and releases of software under these constraints. Topics include:

1. An important feature of any test program is the environment in which testing is conducted. Having a dedicated test lab with software models can be an important part of successful and cost effective testing..
2. Modeling can increase the effectively of test labs by providing "missing" data in the form of inputs, supporting simulation, or output analysis. Modeling of the software can be an important tool in software testing. But what is modeling? What are the kinds of modeling? What tools can be used to implement models? Do models have to be executable?

3. The test environment and support tools must be developed, just like the software itself. This is vital for success and must happen prior to the test cycle. We examine how to set up the processes and a lab to achieve higher Software Engineering Institutes (SEI) CMM levels.
4. Given that a main goal of testing is to expose errors, the areas of software risk and types of errors must be considered in selecting tools, modeling, and test environments. Types of errors that are found when using models/test environments are presented.
5. Many tools have limitations in an integrated test environment, and this must be factored into planning and a test development (both tools and tests) effort to be successful. The limitations of models/test environments must also be considered.

To be successful, test environment (i.e., the lab) construction includes the hardware, software, support techniques, time, management, and people. And each of these must be planned, developed, implemented and controlled. This presentation outlines Lockheed Martin's 20 years of lab set up and modeling to support software testing.

Problem: Software Test Environments

Most testers, after they have gotten some experience, find that software errors leading to failures *still occur* even after they have done a good job testing. This can occur for a variety of reasons, one of which is that the software gets used differently from how the test engineer planned during test definition. This occurs because:

1) Hardware configuration can be different;

2) Users do things that the software test plan did not cover;

3) Inputs/outputs related to external sources are not what was tested; and

4) States and conditions that the software is in, are different than expected.

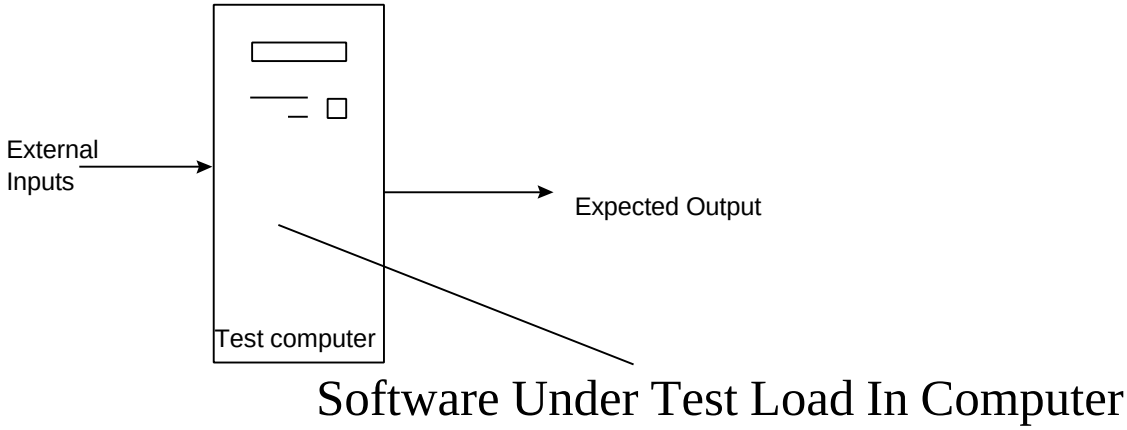


Figure 1 - Basic Test Environment

All of these come down to the point that the real world is different from the test environment. Companies doing testing of software usually establish test computer systems (test environments) in which they can run their software. These are often in laboratories that have the software under test installed on a target computer (Figure 1) where inputs

can be fed in the system and outputs obtained. This is the most basic test environment. But computer environments can vary widely from the laboratory to the real world. Testers must concern themselves with the nature of the test environment because if the fidelity of the environment is not sufficient, testing will not be complete and comprehensive. Consequently it is possible that failures will be missed.

The test environment should mirror the "real world" as closely as possible (Figure 2). For some software test plans this may not be a big problem. The users or customer's environment and system may be well understood, or the tester (you or me) may be testing on the actual system where the software will run. These cases make testing easier, but many other software applications will run in the real world where the computer on which the software is running interfaces to hardware or external processes that the test lab did not or cannot fully replicate.

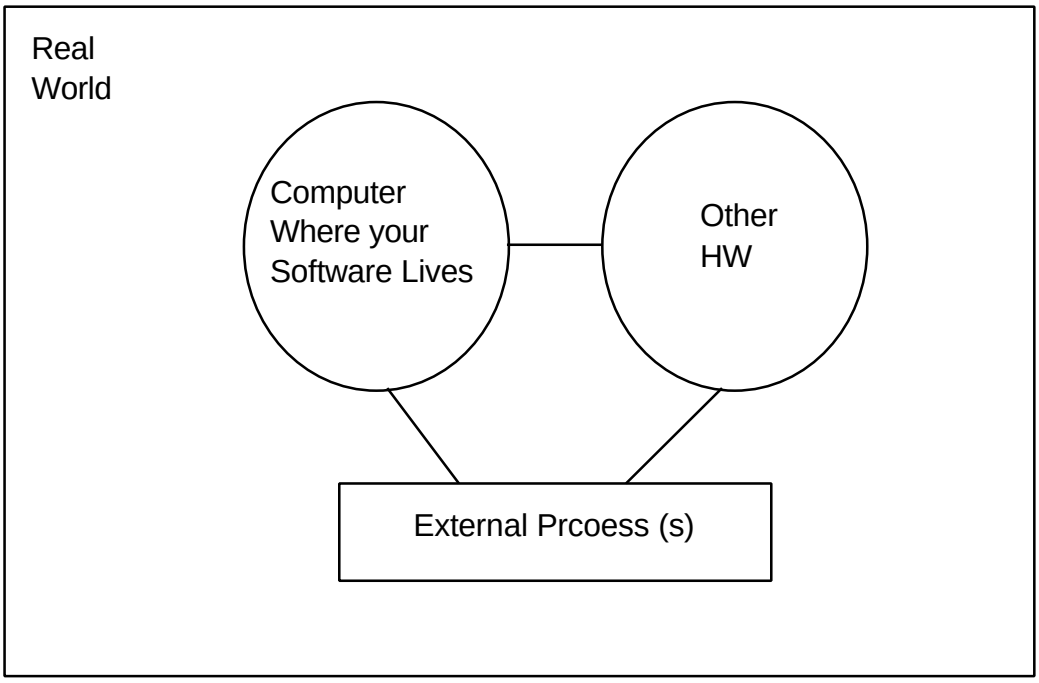


Figure 2: The world that software lives in

"Murphy's Law" reminds us that the real world is filled with stress cases that can: put the software into unknown hardware; have bizarre inputs from users; and have time effects that impact the software. All to often these expose errors that lead to system faults or even failures. Some programmers call these "features" or say "Garbage In/Garbage out" (GIGO) which does not need to be handled. While managers will cling to statements like "the software requirements or use-cases never indicated that we would function under these conditions", and while that may be true or offer some protection, it can hurt business when customers are unhappy, it costs money, takes time, and leads to rework.

Testing for any complex piece of software will never exercise all possible use cases, processes, user inputs and/or hardware configuration. This fact is accepted by most of us, but can lead some organizations to avoid doing anything but minimal test efforts. But, as

software users become more savvy and competition continues, most companies cannot afford to have software that fails.

Test Environments (Labs) Supported by Modeling

A partial solution to the problem of test environment fidelity exists in using models of the real world to handle many of the input, environment, and output limitations of labs.

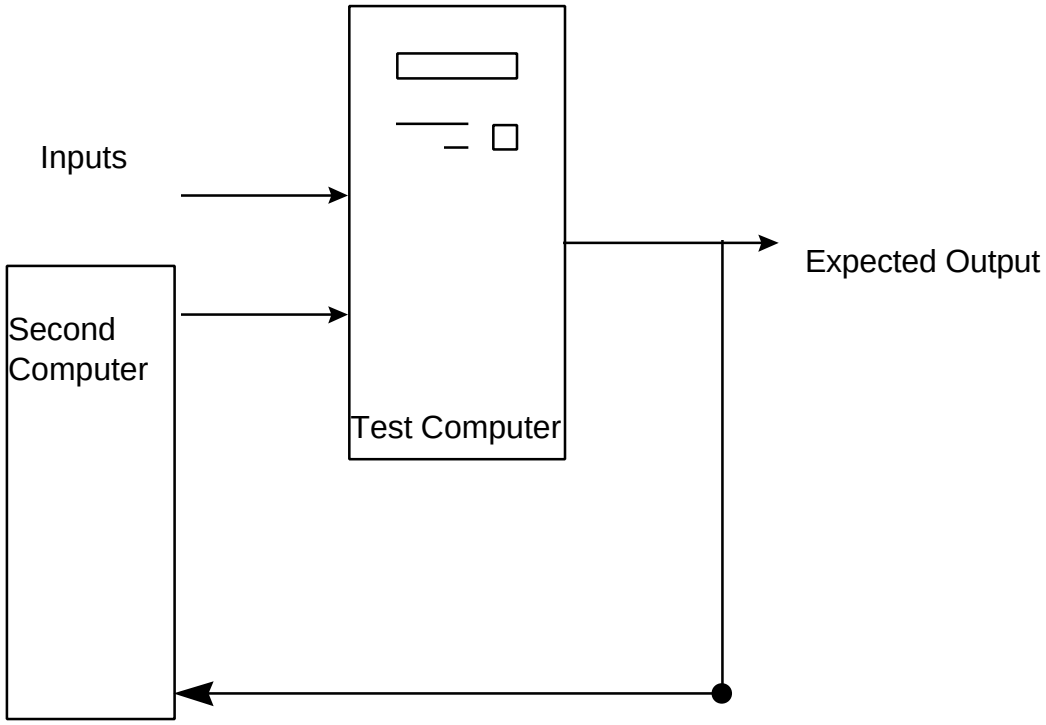


Figure 3 Test Environment with second Computer/Software in the loop

A simple kind of model exists in test lab when we connect or include a second computer to the computer with the software under test (Figure 3). This second computer can be used to simulate aspects or all of the external world that the test computer "sees".

Variations of this concept exist. The second computer and its modeling software can be used in:

- a closed or open loop simulation of the world, including time considerations;
- prior to test execution as a generator of inputs to the system under test
- a pre-execution test oracle; or;
- after the fact judge of software results.

It is possible to have two programs running in the same computer, one being the program under test, and the other being a modeling/simulation program. This approach is possible in many systems, though care must be taken on how the two programs interact with each other, since in the "field", the test programs and models are usually not installed on the actual system. This is basically what some Graphical User Interface (GUI) test tools do. For us, simulation modeling programs include capture/playback systems where a tester

uses a test tool to capture the inputs and system responses for later playback and checks during subsequent testing. Software Test Works (STW) and Mercury's Xrunner are examples. However, while useful in many settings, these are only one kind of tool that can be thought of as doing modeling. Users of these tools must factor in set up costs, maintenance, need of programming, and limited domain (GUI systems). Additionally, the tester must form "mental models" or in some other way judge the creation and results of the first capture playback test. This paper does not discuss this kind of modeling/simulation any further, but does deal with how we use other kinds of models in our test environments.

Creating a Lab

Many companies, Lockheed Martin included, have for years developed software and hardware test environments. A lab in our definition is a collection of computers, support/interfaced hardware, and software that allows realistic testing of the software we produce. The test labs can include special equipment that aids in the definition, exception, and analysis of test results. This can include specialized computers with test points, probes, oscilloscopes, input devices, and data gathering tools/recorders. Our test labs typically have specially trained staff and test procedures to aid in the test process. The labs are used to run the testing, having both the inputs to drive the software and facilities to record results.

While great cost and care may go into creating a lab, we have found over the years that most laboratories have limitations that impact how the software is tested. This impact can include testing the software in an unrealistic way that leads us to miss errors. For example, it is not possible for the lab to have outer space conditions in 1Gravity, or a 1000 different network nodes, or users where the test environment has only one or two in the test loop. Also, some test labs do not have every piece of hardware in existence or interfaces may vary from site to site and system-to-system. And, the real processes, time factors, and uses of the software are never fully known. Good requirements and interface documents help. Techniques like use-cases and detailed understanding of the system lead to much better testing. But it is not possible to fully know the future, so every lab has limitations, which means software errors get missed due to the labs limitations.

In creating a lab, we consider how the software we are going to test is to be used. We try to recreate the full world that the software will see in the test lab. Test labs range from a single standalone PC or computer board, up to multi-node networks with huge amounts of hardware and software. We have even duplicated hardware to the point of having miles of connecting cable looped on the floor to simulate long line hardware connects because of signal or timing concerns. But in creating a lab, care must be taken because everything costs money.

A tester must consider the resources for a lab to support testing. Each lab resource translates to money - an ultimate commodity which is usually in short supply. What many software program managers underestimate is the cost of developing a test program and associated lab. This development can be a very large percentage of the total project cost. Hence many efforts are forgotten or minimized on budget, so testers get the "short end of

the stick". Testers must always trade the need for a "right sized" or even "perfect lab" against cost.

In the global competitive world, many companies are realizing they must be "better faster cheaper" to be competitive. Concepts in standardization like the CMM and ISO address test issues. ISO requires that we have processes which are documented and followed. ISO 9001 requires acceptance (in a lab possibly) and field testing because of recognized limits of labs. Lockheed Martin is ISO certified and recently completed assessment at SEI Level 3 successfully, in part, because our standardized processes of having test environments.

Modeling

Testers and managers must be enlightened and allocate the resources and associated money. When money for critical items is not possible or other limitations of the test lab occur, one technique that has been used successfully is modeling. This is where a model simulates something that is "too costly" to bring directly into a lab. While modeling may be "cheaper and faster" than the real thing, modeling must be planned and developed, because it still consumes resources too.

Lockheed Martin has used modeling for each aspect of the test problem (inputs, environment, and output) to address the limits of our labs, while still meeting budget limitations and support ISO and CMM improvement initiatives. We have models that aid in test input generation and definition. We have models that run as part of the test environment. And finally we have models that aid in analysis or prediction of outputs, thus helping to determine correctness.

We classify models into a couple of basic groups. One group is discrete in that they tend to treat time and events as a quantized element. The other group, we call, continuous in that they tend to function more with time as an integral element of their function. Discrete models can run faster than real time, while continuous models usually run in real time or run by calculating time at each step of the model, so that they can run independently of time. Time effects, while less critical to some software, have impacts on all software to some degree, so its effect in modeling in many cases cannot be ignored.

Models come with differing capabilities. Simple models need not be executable (meaning able to run on a computer, change state, or do some computation). The simplest models are the ones a tester forms mentally about the software. More sophisticated techniques exist in the form of techniques such as use-cases, state diagrams, or data flows, where models take on some defined and standard form. Some of these can even become executable in more advanced CASE and programming tools.

Commercial tools such as RDD-100, MatLab, MatrixX, SimuLink, and even capture-playback tools fit into the discrete-executable modeling category. While the tools are vastly different in their use and nature in testing, they each allow the tester to model some aspect of the system. Tools like RDD-100 and Simulink allow systems engineers and testers to model sequences of events and actions that the software will see. And tools like Matlab and MatrixX allow engineers to calculate mathematical functions that the software will implement. While each of these tools has many other features and uses in testing and engineering, Lockheed Martin has successfully used aspects of these tools in software test modeling.

Testers and analysts have successfully used these tools and modeling to do the following kinds of things in Lockheed Martin in support test labs:

- Allow automation of defined test processes in a test lab environment
  - 6d closed loop simulation of launch vehicle
  - Simulation of hundreds of network system inputs (or users)
  - Simulation of hardware that can/does not exist, e.g. ordinance, safety or hazardous events, hardware items that had not been created
- allow generation of inputs
  - Run hundreds of Monte-Carlo simulations of code (e.g. guidance) to find input sensitivities for execution in a full test lab
  - Generation of random or statistically meaningful inputs
  - Generation of object oriented, data driven, or structured analysis model-based inputs to test path coverage
- allow generation and/or assessment of outputs
  - Requirement based for qualification or acceptance testing like ISO/IEEE requires
  - Design based for verification
  - System based for validation

While we have been successful in using models created from these tools, the limitations of modeling must be acknowledged. If the model and use-case of the model fails to represent reality, muany of the test results may be invalid. The validation of the tools, models, and test lab must be demonstrated before testing can be considered complete and accurate. Test models and environments have indicated software errors/faults when none were present. This created a "wild goose" chase scenario for testers and developers. On the other hand, test models and environments have hidden or missed problems that then surfaced in the real world. Each of these two cases can end up costing time and money. Thus, a key to the lab and modeling is the certification/validation of the models and lab components. Certification means that the lab and models are tested as being faithful representations of the system. Once a lab and supporting models have been shown or tested and valid, they must be maintained under configuration control, and updated as the actual system changes. If this does not happen, then testing can be invalid.

The best solution to the test lab validation is to view this effort as an iterative problem. You cannot fully test the test lab without the software and system under test. And you cannot fully test the products until the test lab is complete. So the "test a little" and "develop a little" spiral and parallel life cycle model seems to work well. This is where

testers generate their test plans, models, designs, inputs, environments, and outputs in parallel with the software developers. The developers share early versions of the software with testers and testing provides ongoing feedback to developers. Each set of product gets iteratively improved until sufficient confidence is reached that products can be released.

Lockheed Martin's experience is that release of a product is dependent on the level of risk the software posses. For example, human safety software gets much more testing/iteration than non-critical ground database software. Schedule determines release as well as when there is a decrease in the error detection trend rate for both the modeling/test environment and the products under test. Trend rates are typical plotted over time and as the curve "flattens", releases can be considered in line with schedules. These processes are documented in project software development plans and schedule documents, thus supporting the needs of concepts like CMM and ISO.

Experience

A proof of validity for any test method or tools is this: does it find errors. While modeling has aspects of the disproved N-version programming, having multiple versions/aspects of a software program has demonstrated usefulness to testers because it help find errors/failures in software, as well as showing that the software works (meets requirements either written or expected by the user). The test lab environment supported by a model has worked for years at Lockheed Martin. We have successfully qualified software for launch vehicles, spacecraft, and ground systems with database, transitional and GUI systems. The test labs have supported testing from the unit/debug level up to full system acceptance. The types of errors found include:

- Programming and data errors - sometimes requires "structural" analysis capabilities such as instrumenter tools (Battlemap, STW, IPL products) or hardware probes (T, SAW,), as well as detailed I/O results
- Design errors - fundamental logic or software interaction problems - example: conflicts between two software components
- Requirement/functional error - system behavior did not "fly" mission
- Interface/compatibility errors - user interfaces, hardware I/O, incompatibility between components, time/sequence related

Another experience is, that lab environments benefit programs because the labs are operated by people independent of the developers. These "lab rats" help provide a different set of eyes to assist in analysis of products during inspections and peer-reviews, as well as during the test processes. Boris Beizer [email communications] has observed that the need for tester independence that once was a "religious requirement" in some organizations, has been fading in recent years. However, as the success of inspections and peer review at finding errors has shown, a key is the second set of eyes and perspectives. The test lab people seem to provide this quite well for us. It is true they don't need to be "religiously" separated from developers, but they still represent some level of independence.

We find models to be very good (even necessary) in supporting testing, but both the models and test labs tend to generate large amounts of results data. While automation is not for everyone all the time, if you are automating labs with models, consider having an integrated and interoperable system of automated tools. Having models that interchange data with each other and the lab can be a big benefit. Having models that can read and process lab data with minimal effort improves test products, since without automation a tester can get buried under data.

Another consideration during lab and model set up, is to allow for hardware and software upgrades of tools lab equipment and models. Technology changes, so upgrade and migration paths should be part of any long term plans. Company resources must be allocated over time or dead end systems tend to result.

Finally, a most important point, automation is good, but it can also be a curse. Having plans, processes, people, and good management are all necessary *before* automation. Also, not every test needs to be automated or to have a full lab with automated models built for it. Consider having the right tools for the job and do not expect automation to solve every problem. There are many problems that labs and models cannot solve. In fact, as detailed earlier, they create and introduce their own problems.

Closing considerations

Over the years, Lockheed Martin has built many labs. When done right, test labs have been major players in successful deliveries of software based systems, and is a major item of review before delivery or shipment of systems. If lab results were not sufficient, Lockheed Martin has been known to delay programs.

Because labs represent major resources, ways to reuse them should be a major consideration. Reuse can be practiced at different levels. For example, whole labs can be reused on new projects. Or, models can be reused between common elements of different programs that require different labs due to hardware or system differences. Reuse can even include procedures, processes, and people on different efforts.

Reuse can be enhanced by the use of commercial tools and products. Custom built features may often be necessary but these can become dead ends. Whereas commercial tools have large use bases, and can be more open ended because many different people know how to use them. Commercial products are not an instant solution and most require some level of engineering and integration to work. But they have advantages like maintenance by the vendor, vendor support, and commercial libraries of related information. Do not expect commercials to solve all problems and remember large amounts of effort may still be needed to make them functional.

Finally, and most importantly, make the testing realistic and based on an organized plan. The automation using the lab and models can be major elements of the plan, but they are only parts - not the whole, and not the first thing testers should work on. To be successful, one must have all the elements of a good test plan and organization *in association with labs and models.*

1

Using Quality to Drive Product Development Processes

Johanna Rothman
Rothman Consulting Group, Inc.
781-641-4046
jr@jrothman.com
www.jrothman.com

2

Why develop products?

- Make money
- Trial balloons
- Rest of the world changed
- Whatever the reason, there are quality requirements

© 1999 Johanna Rothman

3

Possible project quality goals

- Minimize time to market
- Maximize feature set
- Minimize defects
- Each project has a top priority
  - Address other two goals within that context

4

Different projects require different processes

- Different processes leverage different outcomes

© 1999 Johanna Rothman

Shifting project goals inhibit project success: SmartStore

5

- Developers wanted to fix previous release's defects
- Management wanted to ship in eight months
- Inadequate project manager

SmartStore's processes never changed

6

- Project goals changed
  - Specific feature set with low defects
  - Time to market
  - Performance
- Difficult for technical staff to keep up
  - Some people did not change what they did
  - Some felt jerked around
  - Some were completely confused

© 1999 Johanna Rothman

7

SmartStore shipped, but...

- Shipped four months late
- Fewer features
- No performance improvement

8

SmartStore's lessons

- Product development process not sufficiently flexible
- Inadequate project manager
  - Unable to explain how shifting goals hurts progress
  - Unable to focus work
- Project Manager's role
  - Define goals
  - Focus the project
  - Work to accomplish the goals

© 1999 Johanna Rothman

9

How to determine primary goal

- It is 3 weeks before the ship date. The defect count is higher than developers prefer, the last features is not quite implemented.
- Ship anyway: Time to market
- Ship when features is complete: Feature set
- Ship when defects are fixed: Low defects
- The other two goals are also important

10

SmartStore's goals

- Predominant goal: Time to Market
  - Specific feature set
  - Low defects

© 1999 Johanna Rothman

11

Project steps

- Clarify project priorities
  - Use Moore's marketing model
- Choose a process based on your quality requirements
- Set milestone criteria

12

Quality Perceptions over Product Lifetime

Market Pressure	Intro-duction	Early Adopters	Main-stream	Late Majority	Skeptics
Time to Market	High	High	Medium	Low	Low
Feature Set	Low	Medium	Low	Medium*	Medium
Low Defects	Medium	Low	High	High	High

*** Medium pressure for features may not mean *new* features. It means the promised features must be in the product and working.**

© 1999 Johanna Rothman

13

Processes, Strengths, Quality Priorities

Project Lifecycle	Lifecycle strengths and necessary conditions to make the process work	First Priority	Secondary Priorities	Project Culture [5]	Lifecycle's prognosis for Success
Waterfall: A serial lifecycle that in its pure form does not consider feedback. I have only met waterfall projects that consider feedback.	Known and agreed-upon requirements Well understood system architecture Requirements stable over project Development team stable over project	Features	Low defects Time to market	Routine	Many successful waterfall projects incorporate feedback.
Spiral Staged-delivery, Evolutionary delivery	Risk management	Features	Low defects Time to market	Steering	successful
Evolutionary prototyping Design-to-schedule (frequently used in concurrent engineering)	Rapidly changing requirements or incompletely known requirements	Time to market	Low defects Features	Steering	successful
Code and Fix (not recommended, but many organizations use it. [3])	Unplanned projects	Time to market	Features Low Defects	Variable	Does not generally meet any of the desired priorities

14

Time to market example

- Messenger had a history of missing all project goals
- New project: Make a specific time to market
  - Some more features
  - Few enough defects for a usable product
- Used a design-to-schedule process
  - Elements of concurrent engineering

© 1999 Johanna Rothman

15

Messenger's Beta and Release Criteria

Beta Criteria

1. All code compiles and builds for all platforms.
2. All developer tests must pass.
3. All beta customer tests run and pass.
4. All current bugs are entered into the bug-tracking system.
5. First draft documentation is available and shippable.
6. The code is frozen.
7. Technical support is ready.
8. Less than 36 open high priority bugs.

Release Criteria

1. All code compiles and builds for all platforms.
2. Zero high priority bugs.
3. Document all open bugs in release notes with workarounds.
4. All planned SQA tests run, > 90 percent pass.
5. Number of open bugs decreasing for last three weeks.
6. All Beta site reports obtained and evaluation documented.
7. Reliability criterion.
8. Final draft documentation available.
9. A working demo runs on previous release.
10. Performance criterion.
11. At least two Beta site references.

16

Messenger results

- Project met Beta and Release Criteria
- Shipped on time
- Almost all features (all required features)
- Users only found two of the open defects

© 1999 Johanna Rothman

17

Feature set example

- ExtendIt had a history of missing dates and taking forever to complete functionality
- Inadequate change processes to cope with lengthy development schedules
- New project: Get specific set of features to market
  - Time to market close secondary consideration
  - Low defects not as important as the other two
- Used a quarterly staged-delivery process
  - Each release decomposed into several projects
  - Each project divided into series of staged deliveries
  - Each project tested against Beta and Release criteria

18

ExtendIt's Beta and Release Criteria

Beta Criteria

1. Components 1, 2, 3 must be usable by Beta customers.
2. All code compiles and builds for all platforms.
3. All system tests must run, >90% pass.
4. System tests for Components 1, 2, 3 must run and pass.
5. First draft documentation is available and shippable.
6. The code is frozen.
7. Technical support is ready to support Beta.
8. For each major component, zero major bugs, and fewer than 30 minor bugs.

Release Criteria

1. Components 1, 2, 3, must be complete.
2. All code compiles and builds for all platforms.
3. All system tests must run, >98% pass.
4. System tests for Components 1, 2, 3 must run and pass.
5. Final draft documentation is available and shippable.
6. The code is frozen.
7. Technical support is ready to support Release Product.
8. For each major component, zero major bugs, and fewer than 20 minor bugs.
9. At least three Beta site references.

© 1999 Johanna Rothman

19

ExtendIt results

- Project met all milestones
- Components 1, 2, 3 shipped
- Developers not burnt out

20

Low defect example

- ViewIt already successful focusing on time to market or feature sets
- New project: Get improved accuracy and system performance
  - Time to market
  - Feature set
- Used a concurrent design-to-schedule process
  - Included iteration cycle for requirements changes
  - Stringent development process
    - Review code and unit tests
    - Two-stage build process

© 1999 Johanna Rothman

21

ViewIt's Release Criteria

- All unit tests executed and passed.
- All code reviews complete, no outstanding issues.
- All system tests executed, >98% pass.
- Successful generation of executable images for all appropriate platforms.
- Code is completely frozen.
- Documentation review is complete.
- There are zero showstopper bugs.
- There are zero major bugs, and no more than 10 minor bugs.
- The product is ready to ship on July 1.

22

ViewIt's Results

- Project shipped, meeting all release criteria
- Won a quality award
- Other teams started to use the review and build processes

© 1999 Johanna Rothman

23

Summary

- Quality is what customers value
  - Low defects AND
  - Time to market AND
  - Feature set
- Balance all aspects of quality for your project
- Create Release criteria, trading off each aspect of quality
- Choose a product development process to provide leverage for your project
  - Subprojects may need different processes

© 1999 Johanna Rothman

Using Quality to Drive Product Development Processes
© 1998 Johanna Rothman

Abstract

Companies create a variety of products, and different releases of those products, for many reasons. These range from market-testing trial balloons disguised as 'beta tests' to releases forced by incompatible changes in operating systems. Some have many changes, some have few. Some can tolerate fairly glaring defects, others have to extremely reliable. Some are built on very speculative financial grounds, where others protect lucrative franchises.

The goals for quality, schedule, and features for a project are intimately related to the reasons for creating the release in the first place. Since the definition of quality is determined by the goals you're trying to achieve, you can't use a cookie cutter approach to the project management process in software development.

In my experience, each project has one, and only one, of these goals as its top priority, and then addresses the other two goals within that context. This prioritization determines the tradeoffs the project manager will make during the project, and should help the project manager determine the product development process to use in developing the product.

Different product development processes provide unequal leverage for the three possible top priorities of the software product development. This paper uses concrete examples to illustrate how to choose the quality priorities for a project. Once the priorities are set, it illustrates how to choose the appropriate product development process.

Define this project's quality

Every project has its own goals. To determine the appropriate product development process for a project, you must first define and understand your project goals. The end goal in almost any project is to make money, either by selling the product or by serving an internal need, such as IT projects or internal tools. Choose the goal, which fits with the rest of the business objectives of the larger project. For example, if the goal is a time-to-market driven beta-balloon, then there are no "customers" at all – yet. But there is a business imperative to have "at least these features by this date."

The three possible project goals for any given software project are [2]:

- Minimize time to market, by delivering the release as soon as possible.
- Maximize customer satisfaction, by delivering a fixed set of features, whether demanded by existing customers or believed to be required to grab the hearts and minds of potential customers
- Minimize number of known and shipped defects

In my experience, a project can have one and only one of these goals as its top priority. The manager must address the other two goals within that context. If the manager accepts and understands this prioritization, she can actively decide to shape the development process to achieve the goal. As the project proceeds, she can make tradeoffs consistent with the priorities. If the manager refuses to accept that only one of these goals can be paramount, the project is likely to not make any of its goals.

When you start a project, you have to pick one of these three[3]. as the predominant goal of the release. Choose which aspect of quality is most valuable to your customers, whether they exist or are represented by the company. The other two goals are traded off within the context of the primary goal.

This is an example of what happens when a project does not have one primary goal for a specific release:

SmartStore product developers worked on a major software release. The organization consisted of:

[3] Note that cost of development is explicitly not stated in these goals. In general, the acts taken to decrease the cost of development increase the cost of quality (overall quality): the organization simply cannot generate a product within the time/feature/defect constraints if they are not sufficiently funded. Reducing development cost without changing the time/feature/defect constrains generally extends the development cycle, and/or increases defects, and/or reduces the features in the product. [1]

- A product development group who believed that the most important thing to do in any release is to fix the bugs reported in the last one.
- A management team that wanted (a) something to ship in about eight months for market reasons, and (b) a list of new features, and (c) some performance improvements.
- A project manager who did not understand how to set project goals, nor how to track progress, or how to communicate project goals adequately.

SmartStore planned an eight calendar month time period from start to delivery. No one specified any explicit release criteria, but development was working towards a specific feature set with low defects (to maximize customer satisfaction). A month or so before the planned ship date, SmartStore's managers realized the project would not make the ship date with both the features and low defects. At this point, management chose to focus the release goals on time to market—they would be content with any set of features that the development group could complete in the next 6 weeks (to minimize time to market). They decided to drop some features from the project and they made a new plan and schedule. The schedule was not terrible, it just did not take into account that the product developers needed to change how they worked.

Management then started measuring how well the project team met the new schedule. About two weeks before the new ship date, it was clear to everyone the project would not make the new ship date with any of the features completed (the feature freeze had been slushy and the product developers pushed into Beta anyway). Since they were now going to miss the second planned ship date, with fewer features and more defects, management said "We can't make the features we want in the time we want, so let's get the performance we want" (to minimize customer-visible defects). Performance had been an implicit development goal, but had never been a focus of the project. The project goals were briefly changed to focus on product performance.

After only two weeks of working towards product performance, the SQA Manager realized they might never ship a product. The SQA Manager chose specific product goals (a certain smaller set of features, moderate defect levels including performance no worse than the previous release, and an aggressive time to market) and formulated measurable release criteria. The project was complete when the release criteria were satisfied.

With all these goal changes, some members of the development team were unaffected, some felt jerked around by the apparent lack of management focus, and some were so utterly confused, they were unable to make progress. Even though the professed project goals changed, the product development activities did not change Throughout the entire process, the product developers spent most of their time fixing bugs to make a (lower priority) market ship target.

All of the management gyrations made the project even later because the project manager was busy trying to figure out how to accommodate what management wanted in the current development process.

The project manager was unable to adequately track what was really going on in the project, because he was confused by the lack of a single project focus. Since no one asked the technical staff if they were capable of meeting the new goals, the technical staff ignored the new goals. They continued working on reducing the inventory of defects in the existing product.

The product finally shipped about four months late, without some of the planned features, without the performance, and with more defects than originally hoped and planned. The lack of management focus during a critical phase of the project (when everyone realized the estimates were inadequate) allowed some of the technical staff to drift, some to focus on the new goals, and some to remain focused on the original goals. In the end, no one at SmartStore was happy with the release.

SmartStore's product development process was not flexible enough to accommodate the shifting goals, and the project manager was not capable of explaining to the organization how the shifting goals would prevent the project's success. If you've ever found yourself in the dilemma of shifting project goals, then consider how you focus your project goals on low defects, time to market, or features[4]

---

[4] An honestly designed project has two priority thresholds: the absolute bottom line and everything else. This paper is about the second threshold. For example, even if time-to-market is king, if installing the candidate release wipes the hard drive of 25% of the people who test it, unless the people in charge are clueless, **the product will not ship**. Why? Because the organization is smarter than shipping a product that will decrease its market share. In the same way, there is an absolute limit to features. A release has to have something new, even if there is just a requirement not to wipe disk drives. It might be a very minimal requirement, but it exists.

Some people are not easily convinced that there is only one top priority. Here is a check you can use to verify that choice: If it is three weeks before the scheduled ship date, and the defect count is higher than the developers prefer, and the last feature is not quite implemented, what does management choose to do? If management chooses to ship anyway, then *time to market* is the top priority. If management chooses to ship as soon as that feature is installed, then *customer satisfaction through features* is the top priority. If management chooses to hold shipment in order to fix the defects, then *low defects* is the primary definition of quality.

Note that the primary goal does not exclude a secondary goal from being extremely important. In the case of SmartStore, the top priority was time to market, with a specific feature set as a very close second. Low defects were only important if the features didn't work at all.

If the management and the project manager had established these priorities at the start, the project manager would have had a clear goal to communicate to the developers, and a clear idea of how to measure the progress of the developers. With those, he could have brought the effort to completion on schedule.

When you are asked to manage a project with unclear priorities, your first job is to get clear priorities. You have to make it clear to your management that there's no plan without them. If your management is hopelessly confused, you have to create them yourself (perhaps with other people on the project). It is better to manage the project with priorities you create than with none at all.

When the priorities are unclear, you have to ask more questions—when someone proposes to change the overall project priorities, the product itself shifts, and product shipment is at risk. No process can eliminate the impact of shifting priorities, but an appropriate process may help prevent priorities from shifting. If there is no way to prevent a shift, a conscious approach to the importance of priorities can help you minimize the impact.

One way to choose the right quality priorities for a project is to use Geoffrey Moore's high-tech marketing model [4] to understand the market imperatives. Table 1 provides a view, based on his model, of the interaction between the definition of quality and the product lifetime. For more information, see [6]. The users (and potential users) have different perceptions of product quality over the lifetime of a product. The chart shows how the three possible priorities are sorted for different classes of users. As a product evolves, it moves from one target user group to another, and (in the model) the priorities shift accordingly.

Table 1: Quality Perceptions over Product Lifetime

Product Life/ Market Pressure	**Introduction (Enthusiasts)**	**Early Adopters (Visionaries)**	**Mainstream (Pragmatists)**	**Late Majority (Conservatives)**	**Skeptics (Laggards)**
Time to Market	High	High	Medium	Low	Low
Features	Low	Medium	Low	Medium*	Medium
Low Defects	Medium	Low	High	High	High

* Medium pressure for features does not mean pressure for *new* features. It means that the promised features must be in the product and must be working.

Choose a process based on your quality requirements

There are several well-known product development processes. Within an overall product development process, there are a number of techniques that can be used, such as cleanroom and concurrent engineering. See [3] for more explanation about product development processes. Different processes and techniques have varying effectiveness in terms of the goals of time to market, feature richness, or defect control. The following table is a comparison of process development processes and techniques, their strengths, and their product quality priorities.

Table 2: Processes, their Strengths, and Quality Priority[5]

Product Development Process	**Process strengths and necessary conditions to make the process work**	**First Priority**	**Secondary Priorities**	**Project Culture [5]**	**Process's prognosis for Success**
Waterfall: A serial process that in its pure form does not consider feedback. I have only met waterfall projects that consider feedback.	Known and agreed-upon requirements Well understood system architecture Requirements stable over project Development team stable over project	Features	Low defects Time to market	Routine	Many successful waterfall projects incorporate feedback.
Spiral Staged-delivery, Evolutionary delivery	Risk management	Features	Low defects Time to market	Steering	successful
Evolutionary prototyping Design-to-schedule (frequently used in concurrent engineering)	Rapidly changing requirements or incompletely known requirements	Time to market	Low defects Features	Steering	successful
Code and Fix (not a recommended product development process, but many organizations use it. [3])	Unplanned projects	Time to market	Features Low Defects	Variable	Does not generally meet any of the desired product criteria
Product development techniques					
Cleanroom	Known customer requirements Development team capable of maintaining reviews during project	Features	Low defects Time to market	Routine	Successful when the project maintains the work style
Concurrent Engineering	Most requirements are known, but not all Risk management by bringing in cross-functional team	Time to Market	Features Low Defects	Steering	successful

When you decide on your first priority of quality, you can then deliberately choose the most appropriate product development process to support that definition of quality. Sometimes you choose a combination of processes to meet the mix of quality attributes needed for this product at this time. In essence, you tailor the development process to produce the kind of quality that matches the overall business requirements.

If you are called upon to plan a project with unclear requirements, or other high-risk components, consider a spiral process to manage the requirements risk. To meet an aggressive time to market goal, consider concurrent engineering to make sure the whole company is ready for product shipment. To meet a specific time to market goal with low defects, consider a staged delivery process.

In the case of SmartStore, the project's criteria for success were unclear. If the project manager had realized that a specific set of features was necessary, but that time to market was critical, the project manager could have

[5] No process is appropriate for a first priority of 'low defects.' In reality, does anyone ever say, 'We don't care what is in it or when it is, so long as it has no more than X defects?' Perhaps they do say, 'The goal of this release is the reduce the defect list to size X. Here is a list of desired features to be included if time permits while stomping on defects. Any delivery date in the range of X to Y is acceptable. Y is an absolute limit so long as the defect list has shrunk by at least Z. If not, then all of you will need new jobs and your replacements will plan another attempt.

communicated these priorities to the development team. SmartStore was in the Early Adopter phase of their market. As a result, time to market was much more important than low defects. If the project manager had realized that time to market was the first priority with a certain set of features, and low enough defects to use those features, the product development process would have reflected those priorities[6]. And, then when management wanted to shift priorities, it would have been easier to explain the effects of changing the project goals.

This next case is an example of an organization choosing their definition of quality for their product in their marketplace, and then explicitly setting Beta and Release criteria to know when those goals were attained.

Example of process based on time to market

Messenger had a history of problems in the product development. All their previous projects were late, there were too many defects for the products to be sufficiently useful, and some necessary features were missing. Even at the start of this project, the Messenger organization was constantly in "catch-up" mode, trying to fix the deficiencies in previous releases. This project started as the others had, with vague project goals, and creeping requirements. Once the first project milestone was missed, the parent company changed management, including the project manager. New business and project management staff was brought in.

Messenger had an overriding goal: to make a specific time to market. They had a small number of customers, and were trying to break into the mainstream of the market. They had to have enough features in the product so that their potential customers could use the product and few enough bugs so that the product was mostly usable.

Messenger technical staff decided not to repeat the past. They worked with the project manager to define the value this project had for their current and future customers. They decided time to market was critical, with a small set of features. They built a project plan based on a design-to-schedule process, with elements of concurrent engineering. They divided the work into a set of relatively independent pieces of software. The technical staff worked in parallel on the different pieces. The plan called out specific times at which they would integrate the changes.

Messenger's Beta ship and Release ship criteria were different because they used a design-to-schedule process. They developed the requirements, designed the architecture, and worked on features in priority order. That is, the highest priority feature was designed, implemented, debugged, and integrated first. The next highest priority feature was designed, implemented, debugged, and integrated second. This repeated for all the features before the release date. (Implement features until you run out of time.) Messenger was able to work on a number of features concurrently, because the staff worked in parallel. Since all the necessary features were implemented in parallel, they were sure they would make their release date.

Table 3: Messenger's Beta and Release Criteria

Beta Criteria	Release Criteria
1. All code compiles and builds for all platforms. 2. All developer tests must pass. 3. All beta customer must run and pass. 4. All current bugs are entered into the bug-tracking system. 5. First draft documentation is available and shippable. 6. The code is frozen. (Debugging had stopped long enough to create a stable build and verify the Beta customers could use the product.) 7. Technical support training plan is in place, and the people are in place. 8. Less than 36 open high priority bugs.	1. All code compiles and builds for all platforms. 2. Zero high priority bugs. 3. Document all open bugs in release notes with workarounds. 4. All planned SQA tests run, > 90 percent pass. 5. Number of open bugs decreasing for last three weeks. 6. All Beta site reports obtained and evaluation documented. 7. Reliability criterion. 8. Final draft documentation available, complete and submitted to corporate organization. 9. A working demo runs on previous release. 10. Performance criterion. 11. At least two Beta site references.

This project met the Beta and Release criteria. The product shipped on time, with almost all the features, and very few user-visible defects. The users found only 2 of the open defects, and were able to work around those problems.

The users did not find other defects. Messenger met their quality goals with this design-to-schedule (and goal specification) process.

Example of process based on features

ExtendIt had problems in their previous projects: all their projects took much longer to complete than they expected and the projects were still missing major pieces of functionality. Since the projects took too long to complete, users started changing their requirements, frequently just before the release shipped. ExtendIt did not have change management processes that allowed them to deal with requirements changes—the only thing they knew how to do was hold the release to put the changes in. This led to even longer project schedules. It was common to have to wait for up to two years for a complete feature.

ExtendIt had an overriding goal: get a certain set of features to market. If each feature was not complete, the product would not be shipped. Time to market was a close secondary consideration—their market was growing, and they wanted to increase market share. This product had a significant number of pre-existing customers, and was firmly entrenched in the mainstream of their market.

ExtendIt went to a quarterly staged-delivery process called a Release Train. The product manager decomposed each release into several projects. Then, the product manager divided each project into a series of staged deliveries. The project manager worked with others to define the requirements for the release and with the developers to define the architecture. Then, the developers built and tested the components in parallel. This process reduced schedule risk by fixing the architecture at the outset and choosing an architecture built from components that could be developed in parallel.

As each component was integrated into the system, the system was tested to verify that it met the absolute ship criteria. If it failed the test, Project Management directed the developers to back out the feature or to wait for the next release cycle. Only projects that could meet their release criteria were allowed on the train.

ExtendIt ran short Beta test cycles. This is an example of Beta and Release criteria from a recent release:

Table 4: ExtendIt's Beta and Release Criteria

Beta Criteria	Release Criteria
1. Components 1, 2, 3 must be usable by Beta customers. 2. All code compiles and builds for all platforms. 3. All system tests must run, >90% pass. 4. System tests for Components 1, 2, 3 must run and pass. 5. First draft documentation is available and shippable to customers. 6. The code is frozen. 7. Technical support is ready to support Beta. 8. For each major component, zero major bugs, and less than 30 minor bugs.	1. Components 1, 2, 3, must be complete. 2. All code compiles and builds for all platforms. 3. All system tests must run, >98% pass. 4. System tests for Components 1, 2, 3 must run and pass. 5. Final draft documentation is available and shippable to customers. 6. The code is frozen. 7. Technical support is ready to support Release Product. 8. For each major component, zero major bugs, and less than 20 minor bugs. 9. At least three Beta site references.

ExtendIt's Beta and Ship criteria were very similar. The criteria are similar because the components had to be close to complete to make a meaningful Beta program. ExtendIt used a staged delivery process with elements of concurrent engineering, to get the required components and the company ready for the release.

ExtendIt's release was successful: the project met all its milestones; components 1, 2, and 3 shipped; and the developers were not too tired to go onto the next project. The customers were amazed that they had complete components, amazed enough so that product sales improved.

Example of process based on low defects

ViewIt did not have customer-viewable problems in their projects. ViewIt's products frequently met their customers' perceptions of value, in all of time to market, feature set, and defects. ViewIt now needed to produce a next-generation machine vision application with very little downtime and high accuracy. They needed to emphasize low defects in this follow-on release. The customer had a previous version of the product, and was not willing to use

software with more defects than the previous release. ViewIt's next priority was time to market, since the software was promised on a very early delivery schedule to the initial customer.

ViewIt chose a concurrent design-to-schedule process. They could predict that some changes in the requirements were inevitable, so they modified this process to include an iteration cycle so that they could absorb these changes and control their impact. They involved representatives across the company from the start of the project, not just to verify product features, but to make sure everyone would be ready to for product shipment.

To ensure that the code would meet the defect metrics, the development team designed, implemented, and enforced a more stringent development process than that used on other projects in ViewIt:

1. All code and unit tests had to be reviewed before it was merged into the project branch.
2. All code went through a two-stage build process. The first stage accepted the code, created a product build, and ran a minimum acceptance test. If the acceptance test passed, the code was "promoted" to the accepted area.

These project rules were different from other ViewIt projects, because other projects did not have the same urgency of low defects and time to market.

In addition to the new process, the ViewIt project team created these product shipment criteria:

1. All unit tests executed and passed.
2. All code reviews complete, no outstanding issues.
3. All system tests executed, >98% pass.
4. Successful generation of executable images for all appropriate platforms.
5. Code is completely frozen.
6. Documentation review is complete.
7. There are zero showstopper bugs.
8. There are zero major bugs, and no more than 10 minor bugs.
9. The product is ready to ship on July 1.

These defect-focused criteria refer to low defects as a primary definition of quality. The bug numbers were chosen based on other products, and the assumption that the severity of each the existing defects would be assessed. The ship date is still very important, which is why it was part of the ship criteria. (There was no formal Beta period for this product.)

ViewIt's process was successful for this project. The project won a quality award from their largest customer. In addition to project success, some other project teams took on the review and two-stage build processes this team had found so successful.

Summary

Product quality is what the customers value [5]. The commercial success of a release depends on satisfying the requirements in the three quality areas discussed in this paper. Your job is to try to predict how marketplace (defined by existing customers, potential customers, magazine reviewers, etc.) will use these three criteria to evaluate your product, and then choose goals consistent with their expectations. All parts of your plan, including development processes, beta tests, documentation and training requirements, should be chosen to reflect your goals for these three criteria.

Those values should drive the project plan and the product development process. Additionally, creating and using Beta and Release criteria allow the project team to test their assumptions and progress as the project unfolds.

Here are the steps I use to determine the product development process for a given project:

1. Decide what aspect of quality is most valuable to the customers: time to market, features, or low defects for the entire project. Decide what else the customers value.
2. Create a balanced scorecard, indicating the trade-off levels of the three aspects of quality: time to market, feature set, and low defects.
3. Expand the scorecard into beta and ship criteria to test your assumptions all the way through the project.
4. Choose a product development process that meets what the marketplace values. Don't be afraid to use different processes on different subprojects.

One of this paper's reviewers offered an analogy about "typical" and "unique" aspects of the project. "Typical" means you've done a project like this before, you understand well how to do it, and you've succeeded at it. "Unique" means that you don't really understand it, and you may not know how to do it at all, never mind doing it well.

When you plan the project: is this mostly a typical or unique project? Are there subprojects that are mostly unique or mostly typical? Many projects are mostly typical with some unique aspects. Can you organize to separate the two, and then use different planning and development approaches for the sub-projects you create that way?

Choose a product development process to provide the leverage you need to the product your customers want to buy at this time.

Acknowledgements

The following reviewers made valuable contributions to this paper: Marsha Browne, James Bullock, Benson Margulies, Steve Smith, Dan Starr, Jerry Weinberg.

References

1. Crosby, Philip, *Quality is Free.* McGraw Hill, New York. 1980.
2. Grady, Robert. *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, NJ: Prentice Hall, 1992.
3. McConnell, Steve. *Rapid Development*. Microsoft Press. Redmond, WA. 1996.
4. Moore, Geoffrey. *Inside the Tornado*, New York: Harper Collins, 1995.
5. Rothman, Johanna. "Defining and Managing Project Focus", American Programmer (February 1998).
6. Weinberg, Gerald M. *Quality Software Management, vol. 1.* , New York: Dorset House Publishing, 1992.

Test Automation Architectures:
Planning for Test Automation

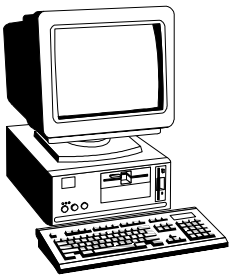
Quality Week '99

Douglas Hoffman
Software Quality Methods, LLC.
24646 Heather Heights Place
Saratoga, California 95070-9710
Phone 408-741-4830

Copyright © 1999, Software Quality Methods, LLC. No part of these graphic overhead slides may be reproduced, or used in any form by any electronic or mechanical duplication, or stored in a computer system, without written permission of the author.

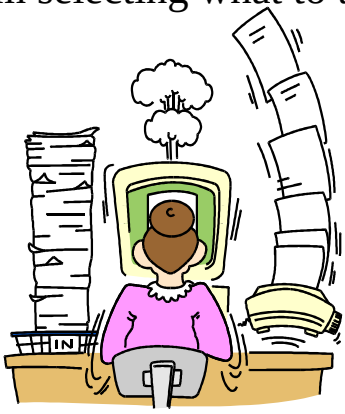
Test Automation

is not just machines running tests!



Test Automation Architecture

From selecting what to test



through reporting results

 Copyright © 1999, SQM, LLC.

Manual Software Tests

- Person initiates each test case

  Person must interact with the test, SUT, or the environment during the test case

  Person is required to act in order that analysis of test results takes place

  Person summarizes and reports results

 Copyright © 1999, SQM, LLC.

Automated Software Tests

- Able to run two or more specified test cases
  Able to run a subset of all the automated test cases
  No intervention needed after launching tests
  Automatically sets-up and/or records relevant test

  Captures relevant results
  Compares actual with expected results
  Reports analysis of pass/fail

 Copyright © 1999, SQM, LLC.

Levels of Automation

- Fully automated software testing

  Semi-automated software testing

  Manual software testing

 Copyright © 1999, SQM, LLC.

Key Automation Factors

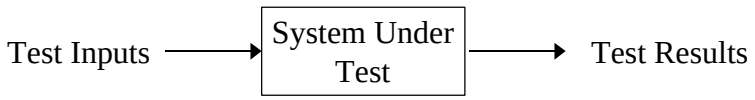
- Components of SUT
- Important features and capabilities

Access to inputs and results

Form of inputs and results

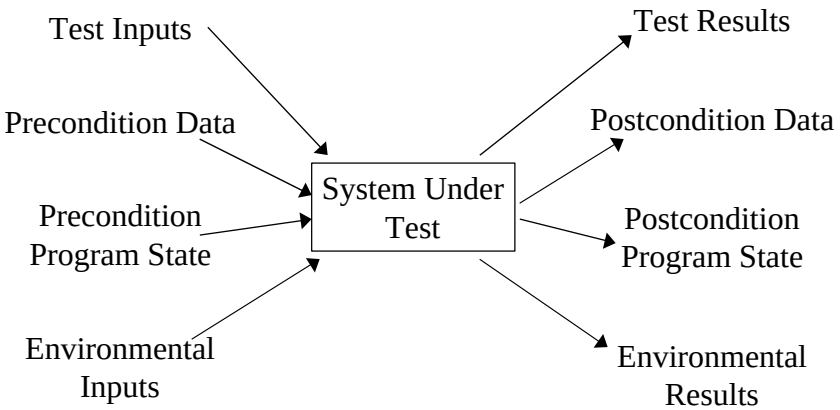
 Copyright © 1999, SQM, LLC.

Simple Testing Model (Black Box)



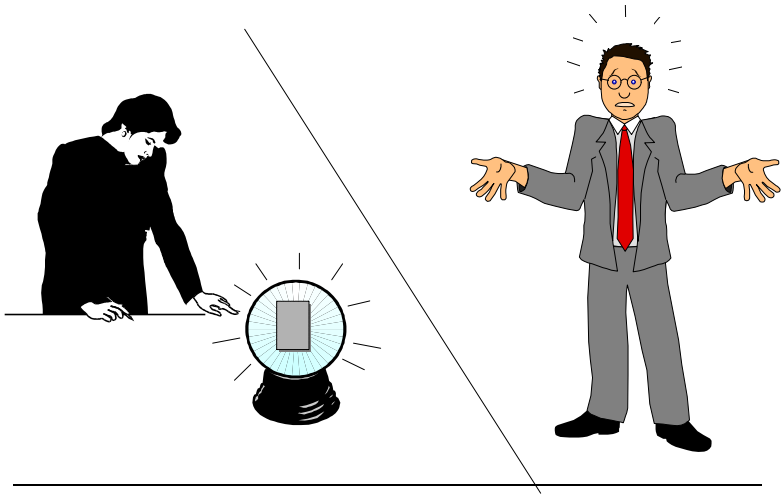
 Copyright © 1999, SQM, LLC.

Expanded Testing Model
(Black Box)



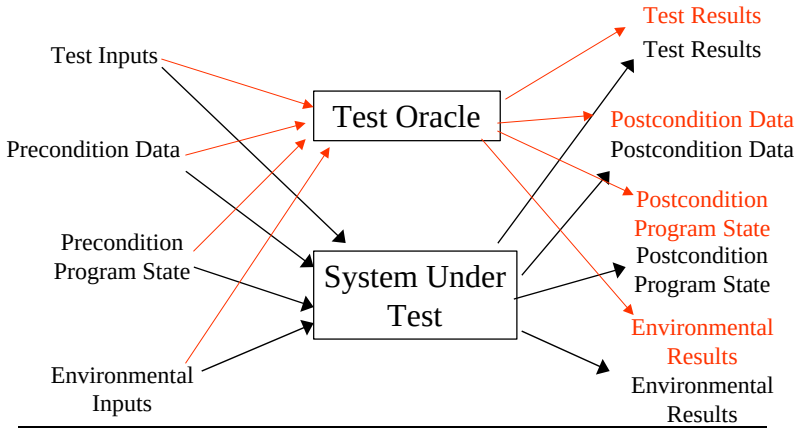
 Copyright © 1999, SQM, LLC.

Test Oracles



 Copyright © 1999, SQM, LLC.

Testing Model With Oracle



 Copyright © 1999, SQM, LLC.

Architecting Automation

- Model for SUT and environment
- Break down software testing problem

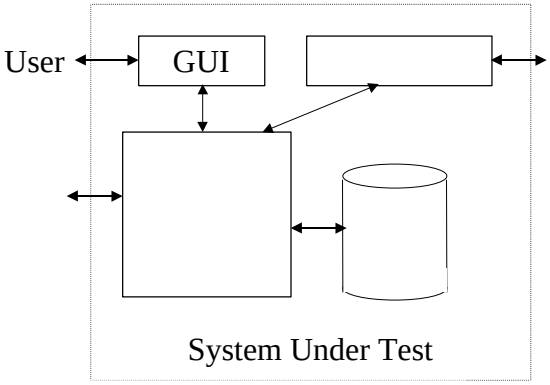
  Decide on location(s) of automation

  Decide on level(s) of automation

  Describe automation architecture

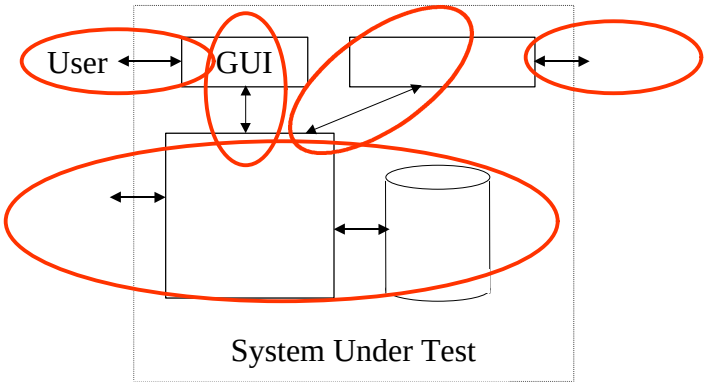
 Copyright © 1999, SQM, LLC.

A Model For SUT



 Copyright © 1999, SQM, LLC.

Break Down The Testing Problem



 Copyright © 1999, SQM, LLC.

Location and Level for Automated Testing

- Availability of inputs and results

  Ease of automation

  Practicality of Oracle creation and use

  Priorities for testing

 Copyright © 1999, SQM, LLC.

Automated Test Sequencing

- Testware version and configuration management

  Select the subset of test cases to run

  Set-up and/or record environment

  Run test exercises

  Monitor test activities

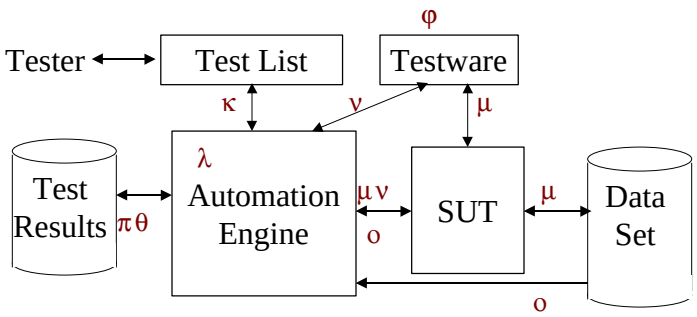
  Capture relevant results

  Compare actual with expected results

  Report analysis of pass/fail

 Copyright © 1999, SQM, LLC.

An Automated Software Testing Process Model



 Copyright © 1999, SQM, LLC.

Automation Process

- List the sequence of automated events

  Identify elements of each event

  Decide on location(s) of events

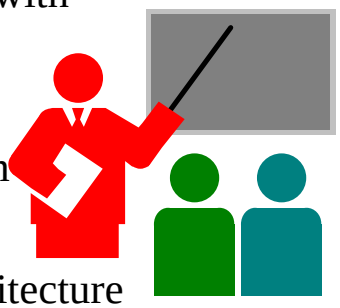
  Determine flow control mechanisms

  Design automation mechanisms

 Copyright © 1999, SQM, LLC.

Conclusions

- Architecture design begins with



Break down testing problem
Model the testing process
Design the automation architecture

Copyright © 1999, SQM, LLC.

Test Automation Architectures: Planning for Test Automation

Douglas Hoffman
Software Quality Methods, LLC.
24646 Heather Heights Place
Saratoga, California 95070-9710
Phone 408-741-4830
Fax 408-867-4550
doug.hoffman@acm.org

Keywords: Automated Testing, Test Oracles, Automation Architecture, Test Models

Abstract

Designing a practical test automation architecture provides a solid foundation for a successful automation effort. This paper describes key elements of automated testing that need to be considered, models for testing that can be used for designing a test automation architecture, and considerations for successfully combining the elements to form an automated test environment.

The paper first develops a general framework for discussion of software testing and test automation. This includes a definition of test automation, a model for software tests, and a discussion of test oracles. The remainder of the paper focuses on using the framework to plan for a test automation architecture that addresses the requirements for the specific software under test (SUT).

Introduction

Many managers, especially those outside of software quality, have a simplistic view of test automation. Test Automation is more than a set of tests run to generate apparent results. It includes designing testware, implementing automated test cases, and monitoring and interpreting a broad range of results. Automation by simply running test cases without human interaction doesn't provide interesting test exercises. We must know how the SUT reacts before the exercise can become a useful test. In fact, automating the running of tests generates data much more quickly than manual testing, and therefore there is more data to sift through before knowing how the SUT responded during the tests. This requires more time from testers and can result in less effective tests. Elements besides the SUT output can also be directly effected by the SUT, and these need to be evaluated in automated tests as well.

Prerequisites for effective test automation include several factors beyond having test cases and automated mechanisms for running them. Organization for the test cases is needed so we can select and understand what test run, When problems are encountered we need to be able to run selected subsets or start in the middle of the tests. Automated test execution is what most testers think of when thinking about test automation. But, test results need to be captured and expected

Copyright © 1999, Software Quality Methods, LLC.

outcomes compared. To automate capture and comparison the data has to be machine readable, which is somewhat difficult for many classes of results including GUI displays, screen navigation, and program states. Automated comparison of results is highly dependent on the form of information being compared. Filters often need to be made to compensate for expected differences.

Test automation can only be successful when we keep in mind that testware in general, and particularly automated testware, is easily made obsolete by changes in the SUT and environment. The architecture must take into consideration that changes will occur and be designed to minimize the impact of such changes.

A Definition of Automated Software Tests

Manual testing can be described as a situation where a person initiates each test, interacts with it, and interprets, analyzes, and reports the results. Software testing is automated when there is a mechanism for tester-free running of test cases. I generally call test cases automated when all of the following elements are present. If one or more elements are absent I consider the tests semi-automated. (Which is often the most cost-effective.)

- Ability to run two or more specified test cases
- Ability to run a subset of all the automated test cases
- No intervention is needed after launching the tests
- Automatically sets-up and/or records the relevant test environment parameters
- Runs the test cases
- Captures the relevant results
- Compares actual with expected results and flags differences
- Analyzes and reports pass/fail for each test case and for the test run

Key Factors in Automated Testing

The first step in planning for test automation is to identify and understand some key factors about the SUT: Identify what software is to be tested, its specific components and features we want to test, and the environment surrounding the SUT. These factors are critical to the automation architecture. Additionally, understand the existing and available testware elements and tools for testing and test automation in the SUT's environment.

Although sometimes obvious, it is often enlightening to formally describe what is it we want to test and distinguish it from all other elements in the system. It is also important to identify what things we think are outside the scope of our automation or we don't intend to test. Several related applications and utilities with different interfaces may comprise the SUT, possibly even running in different environments. Early decisions on which components to include, which to exclude, and which features are most important can put definitive boundaries on the automation tasks and substantially reduce their complexity.

After the SUT elements are identified the environment and interfaces must be considered. For large or complex SUTs there are often multiple environments to consider and the core programs

Copyright © 1999, Software Quality Methods, LLC.

to be tested may have several GUI and API interfaces. The immediate technical environment is important to automation because it provides the facilities and constraints on the most practical approaches. Multi-tasking, process communications, pipes, standard comparison utilities, etc., mold the mechanics of the test automation. Tools and utilities may be unique to one environment or they may be available across platforms. Many tools and utilities that are available in multiple environments have incompatible interfaces, which can make porting of automated test cases as difficult as using different tools in the different environments.

The form of the data for input and results capture is also important. Inputs and results may be keystrokes, data communication messages, pictures, sounds, digital device inputs or outputs, display information, etc. Comparison of binary data is different from characters, and sometimes the data has both syntactic and semantic components that must be dealt with. (e.g., In a data base search, the order of record retrieval may vary but the same records should match the search criteria and the contents of each record should be the same. Another program's output may be postscript listing, which can change substantially without changing the final printed form.) Utilities may run in different operating modes or systems from the core application. Creating a unified test automation architecture is difficult when the SUT needs to be tested in multiple incompatible systems environments. Depending on the requirements in a given situation, a unified architecture may be most effective for the automated testing system or it may be advantageous to employ multiple tools and mechanisms for the different environments.

It is not necessary to have one automation architecture that covers all the components. Testing can succeed when the tester and automation tools perform a useful test and draw proper conclusions based on the results. Automation is valuable to augment the tester by performing tasks that are tedious or impossible for a human or are more cost effective to automate. Different types of tests are run at different times and not all related tests have to be run at one time. There is little advantage in a unified automation architecture when there are substantially different products or environments and very little common testware across them. Two simple automation engines are often easier to create and support than one overly complex one.

The scope of the planned automation tasks also depends on the existing and available testware elements and tools. The testware elements include all of the software, documentation, test cases, data, programs, and associated procedures needed for all the test activities. The tools include operating system utilities, SCM, test selection and control programs, comparison routines, etc., that are employed to do the testing and automation. Although availability and current use of automated tools does not necessitate that automation be based on them, it is common for an automation architecture to have a requirement to include existing testware. (But, if the existing testware and tools were really effective we would not need to develop a new automation architecture.) Whether to constrain possible architectures by demanding inclusion of existing testware should be carefully considered, as it is always possible to continue to use existing automated tests as they are without compromising more effective automation architectures. Frequently the most cost-effective way of dealing with legacy tests is by gradually phasing them out as they require updating.

Two areas of test automation require special attention. Results capture can be a huge task when you realize the world of possible results from running software. The actual results of running a

Copyright © 1999, Software Quality Methods, LLC.

test are much more than viewing information on the screen. System environment variables, memory and file contents, program status, messages, etc., may all be impacted by the correct running of a program. Since we are looking for errors, we must include all the things that the program could possibly impact – a much larger set of things we need to check as 'results.'

Results interpretation is a second area that can spell trouble for an automation effort. We need to identify those things we are going to verify as 'the results' and we need to have an oracle to tell us what those results should be.[1,2] For manual testing we look at the displays, investigate internal variables and program states, and satisfy ourselves that the test passed or failed. For automated testing we must program the automation tools to perform the same task. Manually, the tester decides the sequence of tests and investigations, and it varies based upon the data they find. There may be one test, but there may be several 'correct' results from it, and there are a large number of ways an error would manifest itself. In an automated test environment we must plan the verification of results so that we know when the test passed and when it doesn't. That means identifying all important indications of errors and systematically checking them after every test.

A Model of Testing

Software testing involves more than feeding inputs to a program and observing results. Software today also has states and interacts with stored data and the computer environment. Figure 1 models the inputs and results for some software. Such a model is important in test automation because it provides categories to identify the inputs and results that must be monitored and manipulated during automated testing. For even a simple automated test that feeds inputs to the SUT, the automated test should verify the expected direct results and the postcondition program state to be sure the SUT did the right thing and ended in the correct program state. If the program isn't supposed to change the system environment or any data sets, then some verification should be performed to confirm the correct environment and data values after the test. (These are particularly difficult to do in many systems and extraordinarily important. Manual testers see these types of problems as "program hangs after completion," "user no longer has correct permissions," or "data corruption from unknown source" errors. When we automate tests we must ensure that such problems are detected of we risk incorrectly passing software that obviously doesn't work.

Copyright © 1999, Software Quality Methods, LLC.

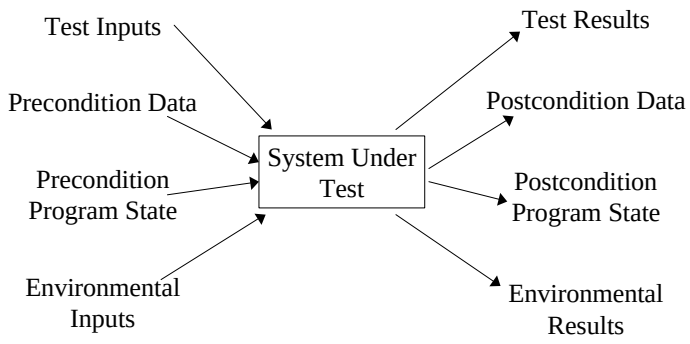


Figure 1: Expanded Testing Model

Software Test Oracles

Test oracles are the mechanisms for generation of expected results so we have something to compare with the SUT responses. For manual testing the oracle is most often the human running the tests. With automation of tests it becomes necessary to automatically generate the expected results in a computer compatible form, and then have the computer system compare actual with expected results[3].

A tester can interpret system behaviors such as GUI navigation and can also perform diverse functions like arithmetic computations, string concatenation, or data base interrogations. Humans know that many factors such as dates and times change and can factor that in when generating or comparing results. The human oracle is so capable that they are often unaware that they are continually modeling program behavior and comparing results.

However, human oracles are not perfect for test automation and some kind of computer based oracle has to be used to fully automate software testing. A human is sometimes slower than computers for generation of expected results. The computer can flash screens faster than a human can capture the images, and volumes of data can be generated much faster than a human can interpret and compare it. Humans are also limited in our view of program behavior. We can't observe system internal data or program states, sometimes lose concentration and therefore miss things, and are easily "trained" to overlook errors by seeing a repeated pattern even when it isn't there.

Architecting Test Automation

To this point in this paper the factors and models have been generic; they apply for any SUT. But, it is not true that any single automation mechanism today fits all situations for SUT. The SUT itself plays a major role in the architecture of the test engine. Assuming that good test cases exist, automation of SUT testing involves specific input values being fed and corresponding

Copyright © 1999, Software Quality Methods, LLC.

results checked through the SUT's interfaces. The process I use for deciding upon the architecture for test automation is derived from models and analysis from the key factors. Models for the specific SUT are used to identify interfaces with the environment and to split the testing into smaller components. Once the SUT and environments are understood, the best places and mechanisms for automation can be selected. From this information an architecture can be articulated that provides the best solution to the automation requirements.

Understanding the SUT for automation requires more than evaluating the inputs, processes, and outputs. Each of the eight input and output sections in the Testing Model must be identified in the SUT. Software can effect data values in memory, stored in databases, and program internal states. There are also side effects to program execution such as resource utilization (disk space, printers, system variables, system environment). Programs can interfere with one another, consume all of a resource, change the system state, etc. Manual testing factors in most of these effects because a human observes more than just the SUT. Test automation needs to account for these results from running the SUT. This is most easily done by modeling the SUT and identifying the various interface points. Figure 2 is an illustration of a simple model for some SUT. A model such as this for the specific SUT for which the automated test environment is intended is used for several things.

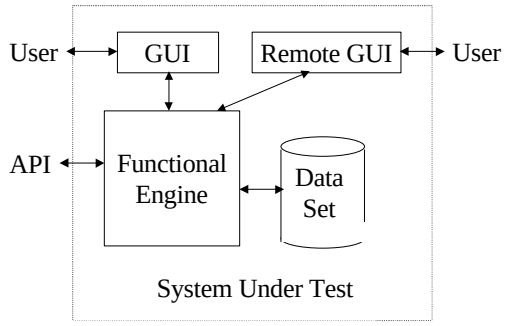


Figure 2: A Model For SUT

Given the model, we might split the testing problem into several smaller problems. The majority of functions performed by a program might be tested through a public or private API. Automating tests run through a programming interface is usually easier and more reliable than through a GUI. A GUI front-end can then be tested independently of the program functions. This means that functional testing is done with programs that can directly feed and retrieve values through the API. GUI testing becomes an exercise to see that values typed into fields are fed into the back-end correctly, that values passed from the back-end are displayed correctly, and display windows are traversed based on the rules for GUI navigation. Figure 3 illustrates a splitting of the testing problem for the SUT modeled in Figure 2.

Copyright © 1999, Software Quality Methods, LLC.

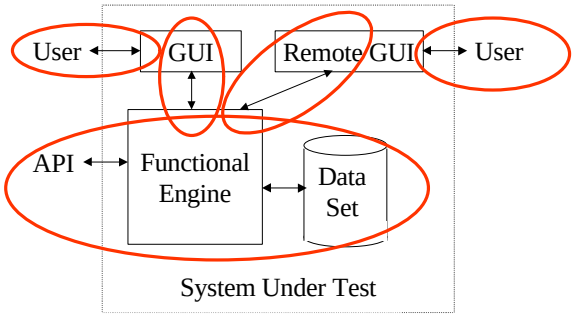


Figure 3: Splitting the Testing Problem

An automation architecture needs to include the SUT, test running, monitoring tools, program inputs, program outputs, stored data, program state, environment characteristics, results capture, results comparison, outcome analysis, and results reporting. Each factor is interconnected, but all are necessary for test automation to be effective.

Automated Test Architecture Descriptions

We have now collected the information needed to describe the architecture. This is the step in the process where "a miracle occurs." Alternatives are weighed based on the requirements and available information. Factors such as the availability and accessibility of inputs and results, the availability or ease of creation of tools, stability of the SUT, availability and practicality of oracles, testing priorities, and resource availability must be weighed with the technical requirements. The result should be a practical, cost effective automation architecture.

There are two parts to the description of the automation architecture. The first is a structural description that shows the elements and connections between them. Like a data flow diagram, information can be displayed and its control and movement depicted. An example of a structure diagram is shown in Figure 4. The second part of the description is the sequence or flow of events. What event starts what process, and when. This second part explains the process for the automated test sequence. A typical sequence is shown below. Not all steps need to be automated. There is always a cost tradeoff to be considered. Each step must be done, but the cost of automating them may be prohibitive for some of the activities.

Copyright © 1999, Software Quality Methods, LLC.

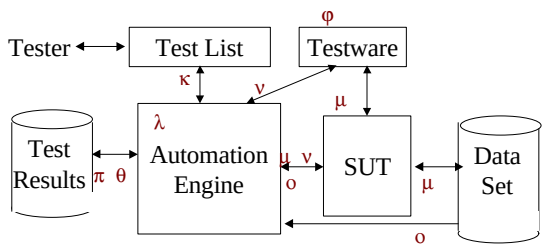


Figure 4: An Automated Process Model

Flow of events

1. Testware version control and configuration management
2. Selecting the subset of test cases to run
3. Set-up and/or record environmental variables
4. Run the test exercises
5. Monitor test activities
6. Capture relevant results
7. Compare actual with expected results
8. Report analysis of pass/fail

Note that the two descriptive elements may be combined into a powerful description of the automation environment. By labeling the structure diagram with the event numbers we can depict which elements perform what tasks in time sequence.

Conclusion

Test automation is much more than computers launching test programs. An automation architecture includes many factors that need to be understood and addressed before automating testing. It begins with understanding test requirements, the SUT, and the test environment. Using modeling we can understand and analyze the testing and automation problems. This information can then be applied to describe a test automation environment using structural diagrams and event sequences.

[1] Douglas Hoffman, "A Taxonomy for Test Oracles," Proceedings of 11th International Quality Week, May, 1998
[2] Douglas Hoffman, "Heuristic Test Oracles," Software Test & Quality Engineering, V1, I2, March/April 1999
[3] Cem Kaner and Douglas Hoffman, "Thoughts on Oracles and Software Test Automation," Proceedings of 12th International Quality Week, May, 1999

Copyright © 1999, Software Quality Methods, LLC.

Euro: An Automated Solution to Currency Conversion

Alan Ark and Sarah Ackroyd
Thomson Financial Services
Boston MA 02210

4/28/99

Euro Conversion: Participating (IN) EMU Currencies

- Austrian Schillings
- Belgian Francs
- Dutch Guilders
- Finnish Markka
- French Francs
- German Marks
- Irish Punt
- Italian Lira
- Luxembourg Francs
- Portugese Escudo
- Spanish Pesetas

4/28/99 2

Euro Conversion : Non Participating Currencies

- Non-EMU currencies:
  - British Pounds
  - Danish Kroner
  - Greek Drachmas
  - Sweedish Krona
- Non-EU currencies:
  - Icelandic Krona
  - Norwegian Kroner
  - Swiss Francs

4/28/99 3

Euro Conversion: Impact on First Call

- First Call is a leading source of real time broker research, earnings estimates and corporate news

4/28/99 4

Euro Conversion: Test Plan

- Scope
  - Specification
- Approach
  - Two types of data verifications
- Resources
  - Limited
- Schedule
  - Very limited

4/28/99 5

Euro Conversion: Test Execution

- Verifying currency descriptors
  - Select country, currency
  - From country_currency_table
  - Where country=‘Austria’

  - Before : |Austria | Shilling |
  - After: |Austria | Euro |

4/28/99 6

Euro Conversion: Test Execution

- Verifying quantitative data
  - Select e.company, e.broker e.year, e.cur_eps, e.prv_eps
  - From earnings_table e, country_currency_table cur, company_country_table com
  - Where com.company=e.company
    and cur.country=com.country
    and com.country='Austria'
  - Before |BMW| ML | 1998 | 2000 | 1500 |
  - After |BMW| ML | 1998 | 1000 | 750 |

4/28/99 7

Euro Conversion: An Automated Solution

- Quantity of data
- Limited time
- Variable data
- Variable participation

4/28/99 8

Identified Goals of Automation

- Test cases should be ***data driven***
- Results should be ***persistent***
- Conversion factors should ***not be hard coded***

4/28/99 9

Why Perl?

- Strong text processing capability
- Use system calls for SQL queries
- Nice looping constructs make this a natural choice
- Easy to pass command line switches to the script

4/28/99 10

Pseudocode of Script Work

Getopts – default to all euro ‘IN’ countries
Get currency conversion factors
Work on LOOKUP tables
Work on COMPLEX tables
Log info about the entire conversion

4/28/99 11

Pseudocode of Lookup Tables

```
Foreach LOOKUP TABLE
    Create and execute SQL query sending results to a file
    Use results file to extract table information
    Foreach FIELD
        Decide how to handle this field in the table
        Compare field with expected result
        Log error as needed
    END of FIELD loop
Log info about the conversion for this table
END of LOOKUP TABLE loop
```

4/28/99 12

Lookup Table Template

```
select g.country, s.currency
from currency_table s, country_table g
where g.c_code=s.c_code
and ( g.cnty_code='CAS'
or g.cnty_code='CBE’
...
or g.cnty_code='CPO'
or g.cnty_code='CSP')
```

4/28/99 13

Pseudocode of Complex Tables

```
Foreach COMPLEX TABLE
  Foreach COUNTRY
    Create and execute SQL queries sending results to files
    Use results file to get information about the tables
    Foreach FIELD
      Decide how to handle this field in the table
      Compare field with expected result
      Log error as needed
    END of FIELD loop
  Log info about the conversion for this table and country
END of COUNTRY and COMPLEX TABLE loop
```

4/28/99 14

Complex Table Template

```
select e.company, e.broker, e.year, e.cur_eps, e.cur_div,
    e.cur_pbt, e.cur_pe,e.cur_yld, e.prv_eps, e.prv_div,
    e.prv_pbt, e.prv_pe, e.prv_yld
from earnings_table e, country_table g, currency_table s,
where g.company =e.company
and s.cnty_code=g.cnty
and g.cnty='XXX'
order by e.company, e.broker, e.year
```

4/28/99 15

Post-conversion SQL Query

```
select e.company, e.broker, e.year, e.cur_eps, e.cur_div,
    e.cur_pbt, e.cur_pe,e.cur_yld, e.prv_eps, e.prv_div,
    e.prv_pbt, e.prv_pe, e.prv_yld
from earnings_table e, country_table g, currency_table s,
where g.company =e.company
and s.cnty_code=g.cnty
and g.cnty='CAS'
order by e.company, e.broker, e.year
```

4/28/99 16

Pre-conversion SQL Query

```
select e.company, e.broker, e.year, e.cur_eps, e.cur_div,
    e.cur_pbt, e.cur_pe,e.cur_yld, e.prv_eps, e.prv_div,
    e.prv_pbt, e.prv_pe, e.prv_yld
from ORIGINAL_earnings_table e, country_table g,
    currency_table s,
where g.company =e.company
and s.cnty_code=g.cnty
and g.cnty='CAS'
order by e.company, e.broker, e.year
```

4/28/99 17

Running the SQL Query

```
sql -number_format $db_info
        < $input_file
        > $output_file

system ("sql -f8f16.6 tools::euro_db
        < earnings_tbl_cas.in
        > earnings_tbl_cas.out ");
```

4/28/99 18

Results From Query

```
INGRES TERMINAL MONITOR -- Copyright (c) 1981, 1992 Ingres Corporation
INGRES SCO Unix SysV Version 6.4/04 (sco.us5/00) login Wed Jan  6 15:05:18 1999
Continue
* * * * * * * * * * * /* SQL startup file */
select e.company, e.broker, e.year, e.cur_eps, e.cur_div, e.cur_pbt, e.cur_pe,e.cur_yld,
     e.prv_eps, e.prv_div, e.prv_pbt, e.prv_pe, e.prv_yld
from earnings_table e, country_table g, currency_table s
where g.company =e.company
and s.cnty_code=g.cnty
and g.cnty='CAS'
order by e.company, e.broker, e.year
Executing

company     |broker|year  |cur_eps      |cur_div      |cur_pbt      |cur_pe       |cur_yld      |prv_eps
     |prv_div      |prv_pbt      |prv_pe |prv_yld      |
+-------------+------+------+------+-------------+-------------+-------------+-------------+-------------
     +-------------+-------------+-------------+-------------+-------------+
|     1621700|26    |  1995|          246|          240|  -5555555|       1681|          579|  -5555555|
     -5555555|  -5555555|          1684|          578|
```

4/28/99 19

Excerpt From a Log File

```
Running the queries now for earnings_tbl in cas at Thu Jan 7 10:39:50 1999
-----------------------------------------------
!!! e.prv_div for cas on row 702 . - company : 1661700   year:1996
Old value: 100    factor: 10   expected new: 10     new: 76
!!!  Extra row+++  e.prv_div for cas on row 9023 .
EXPECTED: company : 9573170   year:1996  GOT: company : 9554678   year:1998
+++ Number of rows in earnings_tbl.cas.out  Expected 9476, processed 9477
+++ Row mismatch
+++ Number of errors in earnings_tbl.cas.out: 0
+++ Number of zeros in earnings_tbl.cas.out: 4
+++ Number of elems checked in earnings_tbl.cas.out: 47380
...
===========================
Total number of errors in conversion: 35059
Total number of elems in conversion: 2818738
Total number of zeros in conversion: 14242
Off by two or less: 0          Off by more: 35059
+++ Time is now Thu Jan 7 13:47:22 1999
```

4/28/99 20

Implementation Results

- Design and implementation completed in less than 140 hours
- SQL query templates contain needed info about each table
- Results are in persistent files
- No data is hardcoded
- Optional command line switches for country choice

4/28/99 21

Empirical Results

- Able to verify the conversion process before the actual conversion went live
- Information in tables accessed by customers most often could be verified in less than 3 hours

4/28/99 22

What About Maintenance?

- Actual table names (both lookup and complex)
- Calls to database API
- Behavior of significant fields
- Behavior of logical NULLS
- Specific calculations that needed to be done for yields and price to earnings ratio

4/28/99 23

Runtime Results

- Two dry runs were successful - only rounding differences in less than 0.05% of the data points
- Successfully QA'ed the conversion process
- Successfully QA'ed the developer's conversion program via black box testing

4/28/99 24

Wins

- Identified rounding errors caused by C's handling of floating point numbers
- Identified an error in the equation being used by development
- Ran on autopilot while the QA engineers worked on other projects
- Identified and fixed any issues ***prior*** to the live conversion

4/28/99 25

Conclusions

- Value of teamwork and communication in action
- Automation saved the day
- Value of perl for QA tasks
- Now have a *robust, tested, portable, reusable currency conversion process and infrastructure* to QA future currency conversions

4/28/99 26

An Automated Testing Environment to Support Operational Profiles of Software Intensive Systems

Robert S. Oshana
Raytheon Systems Company
oshana@cyberramp.net

Agenda

- Introduction
- Conducting a statistical experiment
- Markov Models
- Test Environment
- Conclusions

An unbounded process

- Software testing is an unbounded

  - you can never know with total certainty whether the software:
    - » contains no faults
    - » will never fail
- <u>Any</u> testing process is a form of sampling. Most of this sampling today

Problems with coverage testing

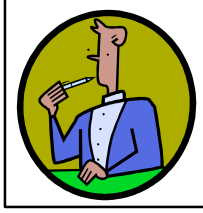
- Many errors occur in software despite complete coverage testing.
  - Many sources of failures are design errors
  - Failures found when executing s/w under the right usage scenarios
  - Failures defined by usage, not code
  - Executing a line of code does not mean that it is correct

Maximize the MTTF

- Some execution failures will occur frequently, others infrequently.
- Coverage testing likely to find the infrequent as well as frequent failure.
- If the goal of testing is to maximize the MTTF a strategy that concentrates on the failures that occur frequently is more effective than one that has equal probability of finding high/low failures.

Another focus on testing

- Test the S/W the way the users use it.
  - Allows development of tests without having to wait for the code to be complete.
  - Find failures that impact the user most.
  - Provide maximum quality for the testing
  - Focus on quality from the users point of



Find the high frequency faults

- Some failures occur more than others.
- Users determine which faults occur and

- Locate high frequency faults early (from a users point of view).
- Reliability increases when high frequency faults are eliminated early
  - ***Reliability is determined by the user !!***

Can we do this statistically?

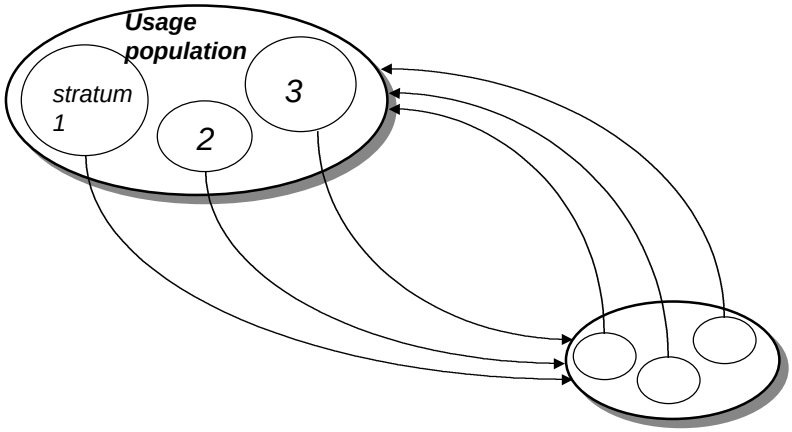
- When a population is too large for exhaustive study (software systems) a statistically correct sample must be drawn as a basis for inferences about

- Treat testing as an engineering problem to be solved by statistical methods

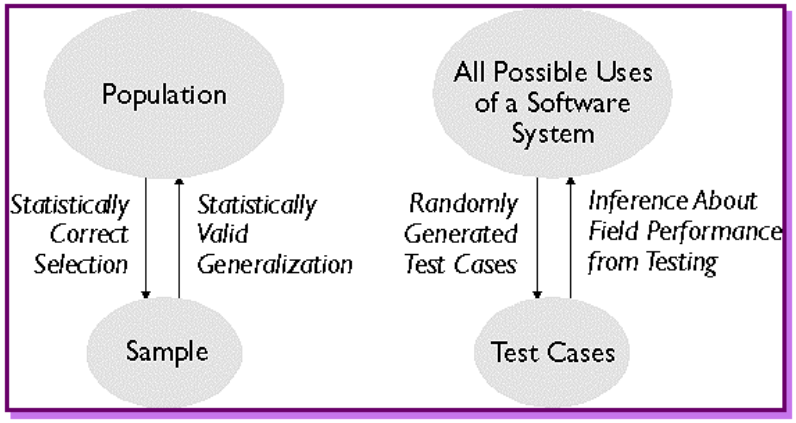
Components of a statistical experiment

- *Population*: the set of items in which we wish to make a statement.
- *Strata*: useful subsets of the population.
- *Sample*: subset of the population actually used in the experiment.
- *Inference*: process of estimating population statistics based on results of

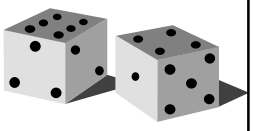
Sampling a population



Parallel between a classical statistical design and statistical software testing



Binomial Experiments (and how they relate to software testing)



- Two outcomes (true or false)
  - Software tests either pass or fail
- Identical trials
  - Outcomes of the trials must be consistent (by testers and evaluators)
- Common setting
  - Same version of software must be used in each test case (a new version marks the beginning of a new experiment)

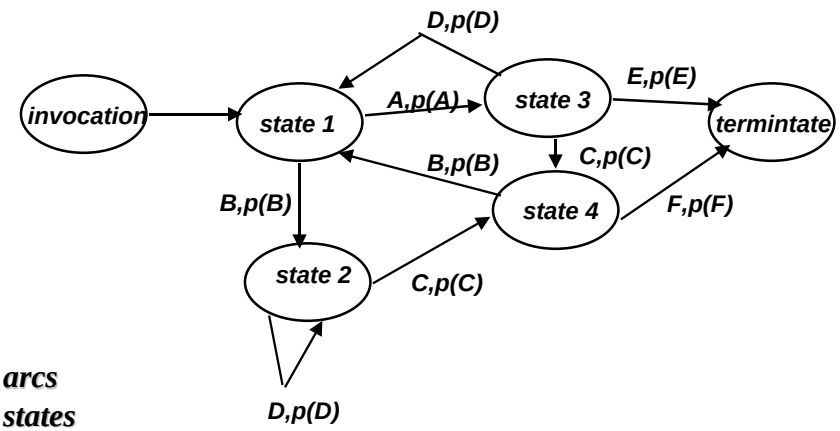
Binomial Experiments (and how they relate to software testing)

- Trials are independent
  - Each software test should be independent of
- Probability of success is the same in all
  - The pass/fail criteria does not change from test

Stochastic software

- Software use is stochastic. S/W has many different uses for different missions starting with different initial conditions and different input data.

Markov chain



1. arcs
2. states
3. probabilities

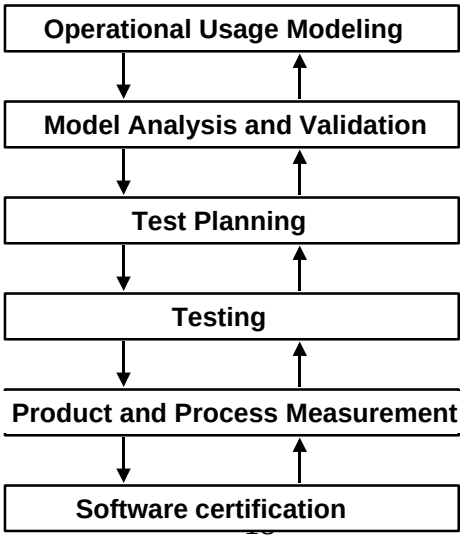
Properties of Markov Chains

- *Time homogeneous*: Probabilities on the arcs do not change with time
- *Finite state*: Finite number of states represented in the model
- *Discrete parameter*: The state transition probabilities are discrete

Advantages of Markov Chains

- State diagram is a common and familiar
- Graph theoretic modeling techniques
- Mathematical analysis available
- Sequences look more like typical use
- Generalizes well

The Statistical Testing Process



Developing Usage Profiles

- Software systems may have multiple users or classes of users that need to
- Different modes of operation are considered as well in different usage
- A finite state model of software usage is developed based on the operational states of the software system.

Special purpose models

- Usage models represent conditions under which S/W is used.
- In general, expected usage conditions
- Other usage conditions may be of

Special purpose models

- Models for special purposes
  - Hazardous usage conditions for safety critical software
  - Malicious usage conditions for special security requirements.
- Usage should be characterized in whatever terms are important in the

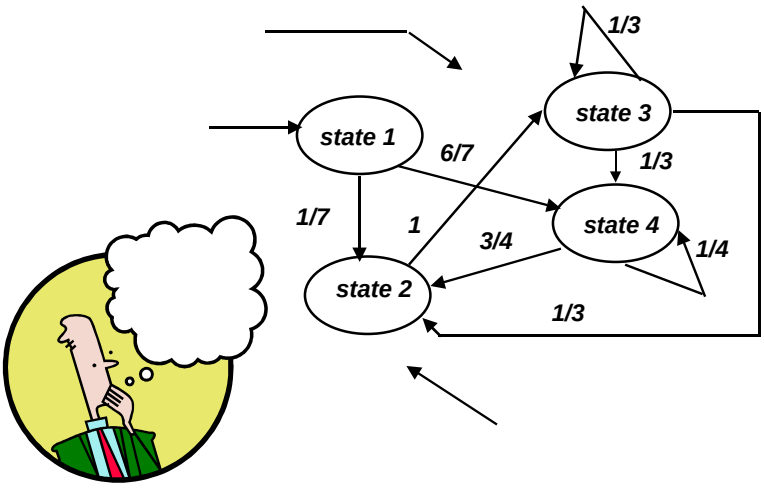
Define the usage probabilities

- Approach 1: <u>Uninformed approach</u>
  - assign uniform probability distribution across the exit arcs for each state.
- Approach 2: <u>Informed approach</u>
  - used when some actual user sequences are available (prototype, prior version, etc).
- Approach 3: <u>Intended approach</u>
  - sequences obtained by hypothesizing runs of the S/W by a careful and reasonable

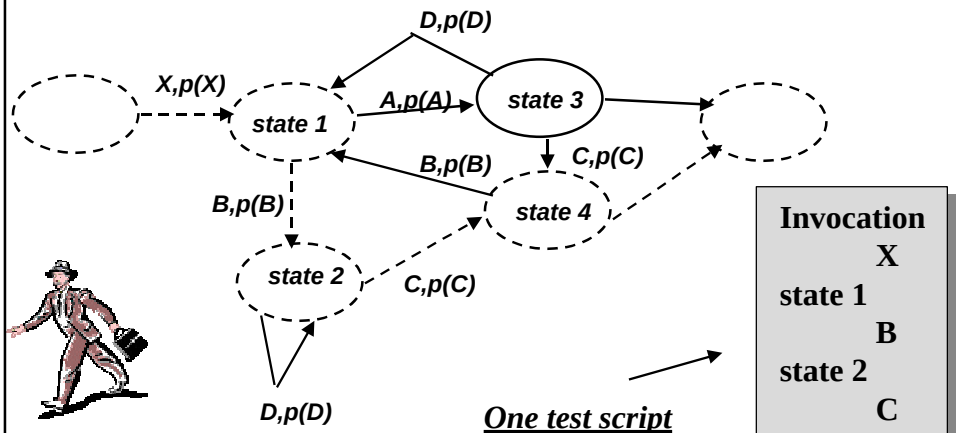
Conduct the statistical test

- Execute tests until required acceptance goals are reached or until enough failures are seen to warrant stopping.
- Determine what is an acceptable and unacceptable outcome for each test.
  - acceptable means the S/W is ready to use.
  - unacceptable requires determining the
  - should testing be stopped?

Using the model to drive testing



A test case is any traversal of

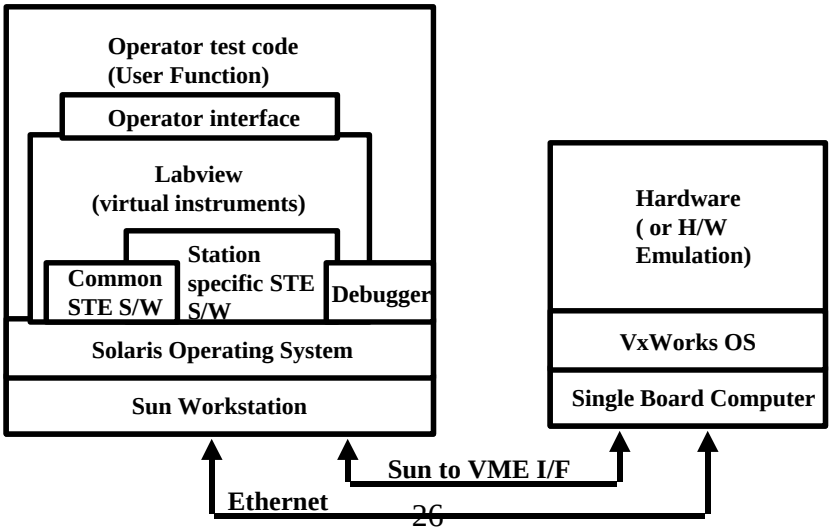


Random test cases based on state transitions that are randomly selected from the usage distribution

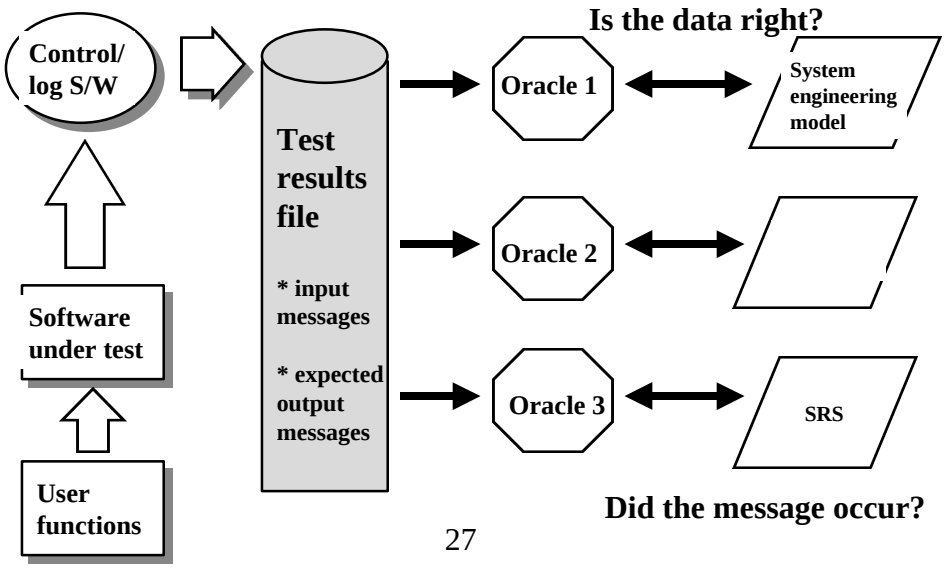
One test script
A combination of these forms the User Function

Invocation
X
state 1
B
state 2
C
state 4
F
terminate

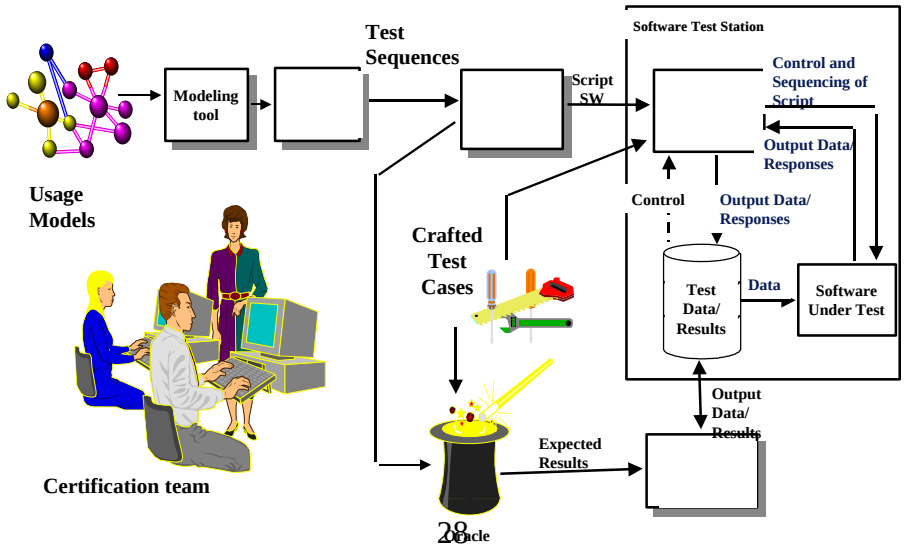
STE Software Hierarchy



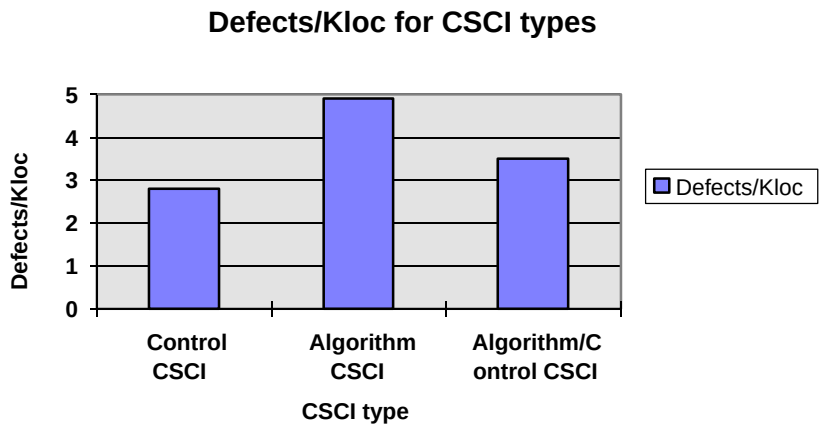
Oracle



Automating the Testing Process



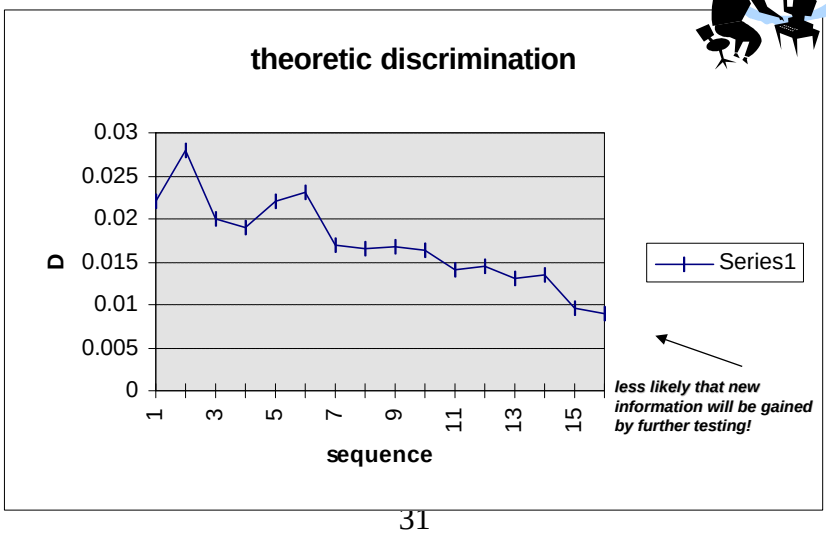
Data



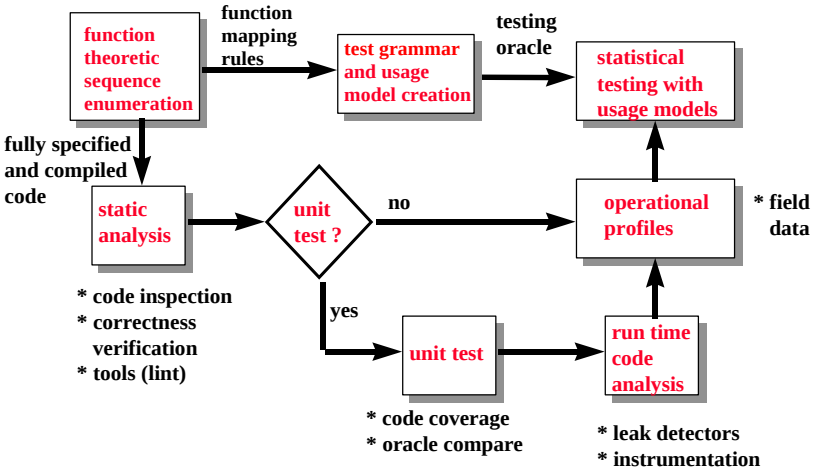
Coverage

- Minimal covering set of the model produced statement coverage approaching
- Full statistical test sequence (1000+) tests achieved coverage in the mid 90% range.
  - 8 - 12 hour test execution time
- Some error paths were tested separately

Stopping Criteria



Model for testing



Conclusion

- Statistical testing transforms the testing process from a craft to an engineering
- Automated testing environment supports the statistical testing approach
- Works well in conjunction with other

An Automated Testing Environment to support Operational Profiles of Software Intensive Systems

Robert S. Oshana
Raytheon Systems Company
oshana@raytheon.com
(972)344-7083

Abstract:

Raytheon Systems Company is a defense electronics company that has been actively engaged in a software process improvement effort at the organizational and program levels for over a decade. The company uses the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) as a basis for their improvement efforts and has been formally assessed at level 3 (defined).

The project is a real-time embedded software application using a heterogeneous computer architecture consisting of the two fundamental computing environments;

- PowerPC single board computer (SBC) using Ada for embedded command and control,
- Digital signal processors (DSP) using the C programming language for a primarily signal processing algorithm based application. There are significant real-time constraints within the signal processing application.

The program follows a tailored version of DoD-STD-2167A and MIL-STD-498 documentation standards. The project is built upon an Integrated Product Team (IPT) structure. There is a fully defined software development process as well as system engineering and hardware development processes.

Our testing approach is based on the concepts of software testing based on statistical principles. The statistical testing approach to software treats the software like a statistical experiment. A statistical subset of all possible software uses is first generated. Performance on this subset is used to form conclusions about operational performance based on the usage model developed. The expected operational use is represented in a usage model of the software. Test cases are then randomly generated from the usage model. These tests are executed in an operational environment. Failures are interpreted according to mathematical and statistical models.

This paper will focus on the successes and issues associated with developing a statistical testing environment for an industrial software project. The paper will also describe how both statistical testing based on software models and traditional testing based on unit and other functional tests can be combined into an effective approach to testing large software intensive systems.

A STATISTICAL EXPERIMENT

One approach to software testing is to treat the testing process like a statistical experiment. There are several components of a statistical experiment (Figure 1);

1. The *population* which is the set of items that we are attempting to make a statement about.
2. The *strata* which are the useful subsets of the population.
3. The *sample* which is the subset of the population actually used in the experiment.
4. The *inference* which is the process of estimating the population statistics based on the results from the sample.

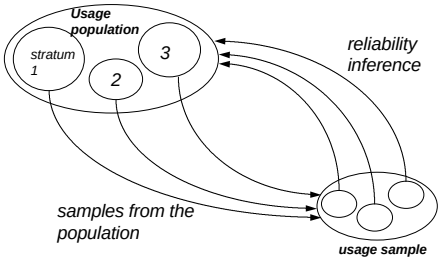


Figure 1. Sampling a population

Software use can be characterized as being stochastic. A *stochastic* process is a random process or experiment that takes place in stages. The outcome of any preceding experiment does not affect subsequent experiments. Stochastic processes can be used to model system state as a function of time. Short term or long term behavior can be studied using stochastic processes. The type of model we will use to model software behavior is the Markov chain. A *Markov chain* encodes the input domain as a set of states which represent usage history of the software from the users point of view. Arcs are used to connect the states and represent the transitions caused by the various stimuli to the system. These stimuli can be generated from hardware, human interface, other software, and so on. Finally, transition probabilities are assigned to the arcs and represent how a typical user is likely to apply stimuli to the system. A Markov chain of this type is a discrete time, finite state machine. A model of this sort can be represented as a directed graph (Figure 2) or a transition matrix (Figure 3).

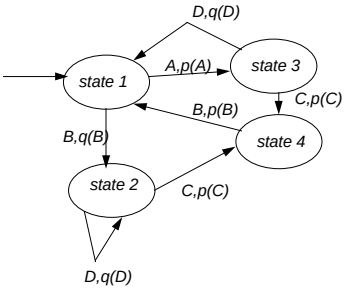
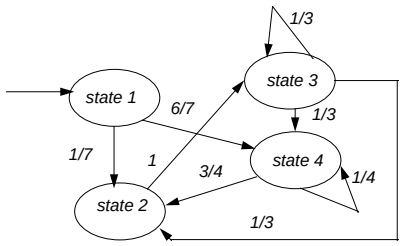


Figure 2. A Markov chain in directed graph format



A Markov chain in digraph format

$$T = \begin{array}{c} \\ \text{state 1} \\ \text{state 2} \\ \text{state 3} \\ \text{state 4} \end{array} \begin{array}{c} \begin{array}{cccc} \text{state 1} & \text{state 2} & \text{state 3} & \text{state 4} \end{array} \\ \begin{bmatrix} 0 & 1/7 & 0 & 6/7 \\ 0 & 0 & 1 & 0 \\ 0 & 1/3 & 1/3 & 1/3 \\ 0 & 3/4 & 0 & 1/4 \end{bmatrix} \end{array}$$

all rows sum to 1

A Markov chain in transition matrix format

Figure 3 Transition matrix and digraph form of a Markov chain

The common properties of Markov chains are;

1. *time homogeneous*; this means the probabilities on the arcs do not change with time.
2. *finite state*; this means the model has only a finite number of states.
3. *discrete parameter*; the state transition probabilities are discrete and not continuous.

Markov chains offer several advantages to modeling software systems:

1. The state diagram form is a common and familiar tool for modeling.
2. There are graph theoretic modeling techniques available.
3. Mathematical analysis is available for these models.
4. The models generalize well and sequences through the model can be made to look more like typical use of the software system.

There are several steps for developing Markov models for software systems;

1. Develop the usage profiles
2. Define the usage probabilities
3. Conduct the statistical test
4. Analyze the results and update the models as necessary

DEVELOPING THE USAGE PROFILES

Software systems can have multiple users or classes of users. Each of these classes of users can potentially use the system differently. The first step in developing usage models is to determine what requires testing. This is referred to as stratifying the input domain. There are two types of stratification; user level stratification and usage level stratification. User stratification refers to who or what can stimulate the system. Usage stratification refers to what the system can do under test. In other words, user level forces you to think about all the various types of users and how they can use the system (which may be different) and usage level refers to all the functionality that the system is capable of providing.

Different modes of operation are also considered when developing a stratification plan. A finite state machine of software usage is developed based on the operational states of the software system. The test developer must understand what the software is intended to do and how it is to be used. No knowledge about how the software is designed or constructed is required to do this. Each usage condition should have a model that represents conditions under which the software is used. In general, the expected usage is modeled but other usage conditions may also be of interest. Usage should be characterized in whatever terms are important in the testing context. For example, if it is very important to test for special purposes such as a hazardous condition or malicious use, a model should be constructed representing this scenario.

DEFINE THE USAGE PROBABILITIES

Once the user and usage models have been developed the arc probabilities are assigned. These probability estimates are based on;

1. user data collected from existing systems
2. talking to or observing the user or users
3. prototyping and/or trial analysis
4. domain experts

There are three approaches to defining the usage probabilities;

1. *Uninformed approach*. In the uninformed approach uniform probabilities are assigned across the exit arcs for each state. This approach maximizes the “entropy” which is a measure of statistical uncertainty. The higher the entropy, the less representative the test sequences are to the model itself. This approach is useful when no other information is available.

2. *Informed approach*. The informed approach is used when some actual user sequences are available (from prototypes, prior versions, etc.). These estimates are driven primarily from field data. This is the best approach to use if data is available.

3. *Intended approach*. The next best approach after the informed approach is the intended approach. In this approach, test sequences are obtained by hypothesizing

runs of the software by the various user and usage types. Data to support this approach comes from user data and domain experts.

CONDUCT THE STATISTICAL TEST

Once the usage models have been created and the probabilities have been assigned, the next step is to conduct the experiment. Tests are conducted using a form of Monte Carlo simulation. In this approach, random numbers are generated and used to traverse the model of the software. Using this type of simulation, statistical integrity is preserved. A random path through the software, driven by the usage model, arc probabilities, and the random numbers generated create a statistical experiment on the software. There are tools available to generate test scripts using these techniques. These tools also generate a wealth of statistical data and visual aids to help the testing organization establish reliability estimates and other stopping criteria.

Tests are executed until the required acceptance goals or stopping criteria are established (Figure 4). The testing organization needs to determine what is acceptable and unacceptable for each test executed. Acceptable means the software is ready for use. Unacceptable requires the software to be re-worked. Testing of this sort on software must be a controlled experiment. In order to use the statistical data available when running such an experiment, the same version of software must be used in each of the test cases. A new version of software marks the beginning of a new experiment. The outcomes of the trials must also be consistent. When analyzing the results of each test, pass/fail criteria must be used consistently by the testers and evaluators. Automated pass/fail evaluation (sometimes called oracles) will help make the pass/fail criteria more consistent. The testing organization must also be explicit to ensure experimental integrity.

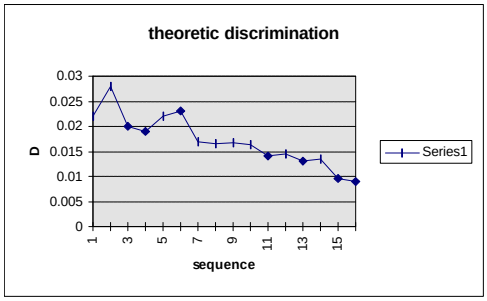


Figure 4. Theoretic discrimination used to determine stopping criteria

There are situations where pure statistical testing cannot be done. Other forms of non-statistical testing may be included because they are required by the customer, required by the contract for the product, or required by law. There are ways, using statistical testing tools, to generate tests scripts that produce the fastest coverage of a given usage model in order to achieve a level of coverage type testing. One effective technique is to perform both non-statistical tests (such as coverage tests) as well as statistical tests in the testing program. If the non-statistical tests are performed prior to the statistical tests, the statistics generated by the statistical testing process will still be

valid. Performing non-statistical tests after statistical tests may invalidate the reliability estimates of the software.

Our program has developed an automated testing environment (Figure 5) that uses statistically generated test scripts based on user and usage profiles [1] as well as other operational profiles [2] to drive the testing effort.

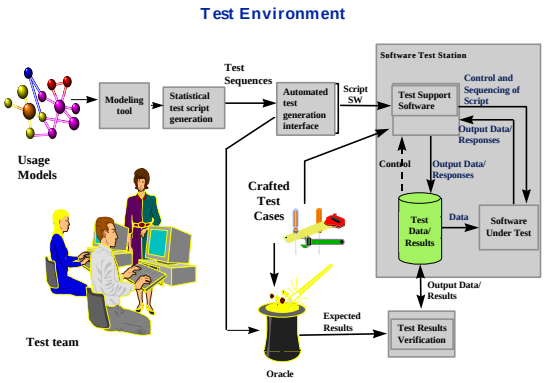


Figure 5. Testing environment automating the execution of statistically generated test scripts

The software components used in the test environment consisted of the following main components (Figure 6);

- Operator test software; also referred to as the "user function". There was a different user function for each of the usage models developed for the software. These programs were written in "C" and generated the message sequences that drove the software under test for each of the test scripts generated from the usage model,
- Labview interface; This software was composed of a number of different Labview "virtual instruments" used to provide an interface to the operator executing the tests,
- Station specific Special Test Equipment (STE) software; this software provides the low level functionality required by each of the different software test stations,
- Common STE software; this software was the Application Programming Interface (API) to the rest of the software. It provided capabilities to watch for certain events occurring in the software under test, log those results, and provide information to the user function software and Labview virtual instruments.

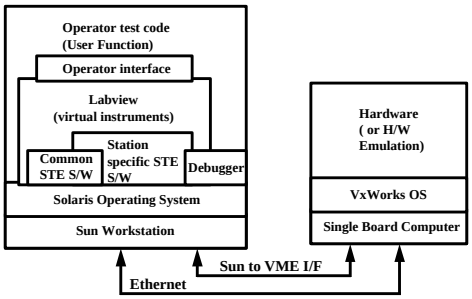


Figure 6. Software components for the testing environment

Once the test has been executed and the input and output events logged in an output file, the data is parsed into several different oracle files (Figure 7). Each of these oracle files was used for a different pass/fail criteria. "Generic" oracles were used to slice the data in such a way as to determine if the proper high level sequencing of the test was performed correctly. Other oracles were used to slice the data to determine other pass/fail criteria such as;

- Does the data reflect the expected outputs as determined by the system engineering model(s)?,
- Does the raw output data match the expected outputs defined in the algorithm document?,
- Does the output message sequence match what was expected in the Software Requirements Document?

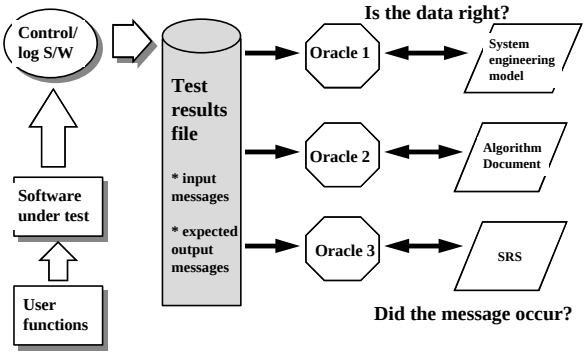


Figure 7. Testing Environment Oracle

TESTING AT THE PROGRAM LEVEL

Software testing at the program level was performed using various levels of statistical testing supplemented with unit and functional testing where required as well as operational profiles testing techniques. Unit and function testing was generally performed for the algorithmic portions of the software. This software involved the more mathematically based functions, with strict real time constraints. Various implementation and optimization strategies were performed to effectively map the algorithms to the processor for optimal real-time performance. We developed an effective model for certification of algorithmically intensive software that included a formal code inspection and correctness verification phase, an optional level of unit and function testing (based on

the nature of the algorithms) which included both static and run time analysis, an operational profile phase using real data collected from various user environments, and a final statistical testing phase using usage models developed for different user and usage stratifications (Figure 8).

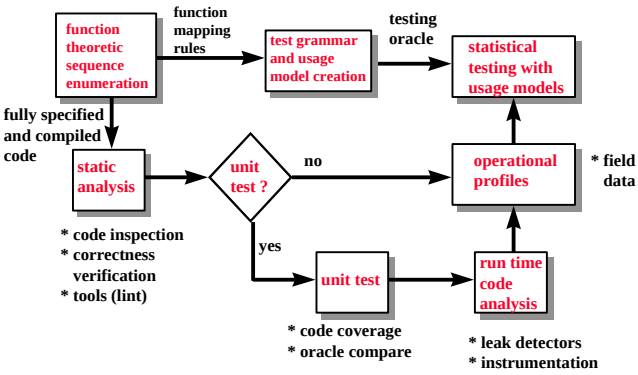


Figure 8. Testing model

LIMITATIONS TO MODELING

There are limitations to modeling software systems that must be understood in order to properly analyze the statistics generated when using statistical techniques such as the ones discussed in this paper. The quote "All models are wrong, but some are useful" has some truth to it. The main point to remember when modeling any type of system using any technique is that all modeling approaches lack, to some extent, the naturalness for representative power. The Markov approach has limitations in handling some of the common issues such as counting and concurrency. But there are ways of getting around some of these issues but it may result in state explosion (larger models). Also, the more abstract the model is, the less confident one should be in the predicted reliability generated by the tools. If one is careful not to blindly accept the statistics without proper analysis, then the statistical techniques discussed in this paper can be very useful in any testing organization and help lead to higher quality software products.

References

1. Musa, John D. "Operational Profiles in Software Reliability Engineering", IEEE Software, March 1993

2. Walton, G.H., J.H.Poore, and C.J.Trammel. "Statistical Testing of Software Based on a Usage Model", Software Practice and Experience, January 1993.

Telcordia Technologies

Performance from Experience

Event-Based Test Generation for Distributed Systems

David W. Carman
Principal Automation Architect
Software Quality Assurance
dcarman@telcordia.com

Copyright © 1999 Telcordia Technologies.
Used by permission only.

Motivating the Testing Challenge

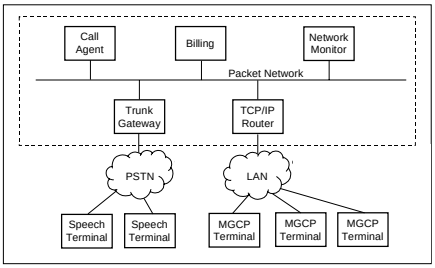
"Testing in the Internet Age"

- Distributed, reusable business objects
  - Connected objects and applications in heterogeneous and distributed computing environments
  - Open systems with component architectures
- Development paradigms evolving toward application frameworks
  - Increased development productivity from reuse
  - More and more commercial examples
- Pressures on testing integrated systems
  - Increased overall system complexity
  - Numerous deployable configurations
  - Time to market

Telcordia Technologies

Motivating the Testing Challenge

"Testing a New Product Line"



- New Product Line
  - Family of packet-switched multimedia conferencing systems
- Open Architecture
  - Multiple vendors with diverse computing platforms
  - Hardware and protocols still under development in the research laboratories
  - Multiple network topologies defined by different standards committees
- Multimedia Interfaces
  - Mixed media (audio, video, data) interfaces
  - Real-time performance requirements

Telcordia Technologies

Developing a Testing Strategy

"Beyond Limitations of Classic Approach"

- Limitations of classic manual design and scripting approach
  - Total testing effort is proportional to the number of test cases
  - Features divided among a group of people and documents
  - Functional and load tests executed independently with different tools

Telcordia Technologies

Developing a Testing Strategy

"Experimented with Model-Based Testing Approach"

- Initially, many issues were raised by model-based testing approach
  - Each commercial test generation tool has different modeling paradigms with advantages and disadvantages. Should more than one test generation tool be used together?
  - The scope of the model changed over time which required several complete re-designs.
  - It was difficult to reuse models. Starting from a "blank page" was best.

Telcordia Technologies

Developing a Testing Strategy

"*Refined Model-Based Testing into a Verification Framework"*

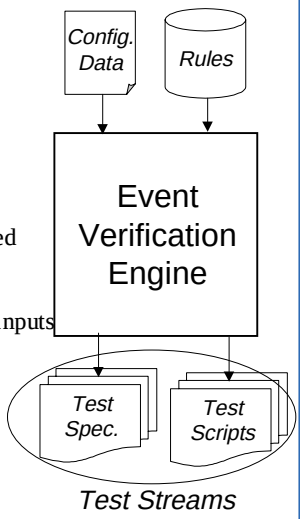
- A new testing strategy called *Verification Frameworks*
  - Inspired by application frameworks and model-based testing technologies
- A verification framework should be . . .
  - a component architecture built with object-oriented technologies.
  - an integrated approach for both test design automation and test execution automation.
  - a convenient system for building modular, reusable test designs and scripts.

Telcordia Technologies

Implementing the Test Strategy

"Event Verification Engine"

- **Event Verification Engine:** A verification framework for event-driven distributed systems
- Key features:
  - Rule-based description of device and interface behaviors
  - Flexible and scalable support for many networked devices
  - Tests designed for any number of simultaneous inputs and outputs
  - Tests reusable across diverse test environments

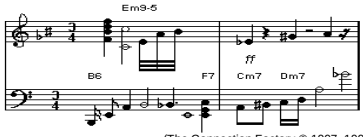


Event Verification Engine

"Automatically Generated Test Streams"

- A test stream is . . .
  - composed of steps with multiple actions (*events*) and results (*states*) per step.
  - both a specification and an implementation, like a single test case.
  - contains both explicit and implicit test cases.
  - is multi-threaded with synchronization capabilities.
  - has deterministic expected results.

"An orchestral score for testing"

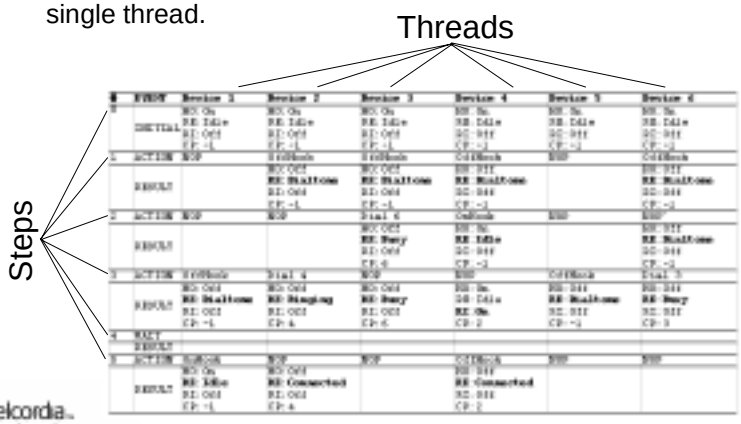


(The Connection Factory © 1997, 1998)

Event Verification Engine

"Automatically Generated Test Specifications"

- Each thread describes interactions for a single "device."
- Test cases are *explicitly* defined by a sequence of steps within a single thread.

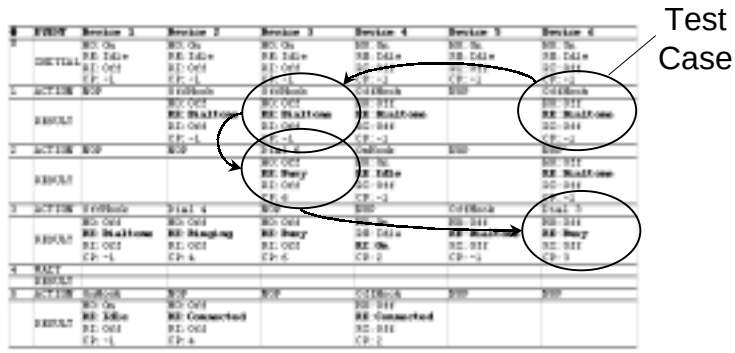


A Generated Test Stream Specification

Event Verification Engine

"Automatically Generated Test Specifications"

- Many more test cases are *implicitly* defined that span multiple threads and events.
- Defects were found by these implicit test cases
  - Called phone started ringing when off-hook with the busy signal.

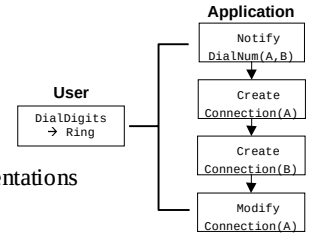


A Generated Test Stream Specification

Event Verification Engine

"Automatically Generated Test Scripts"

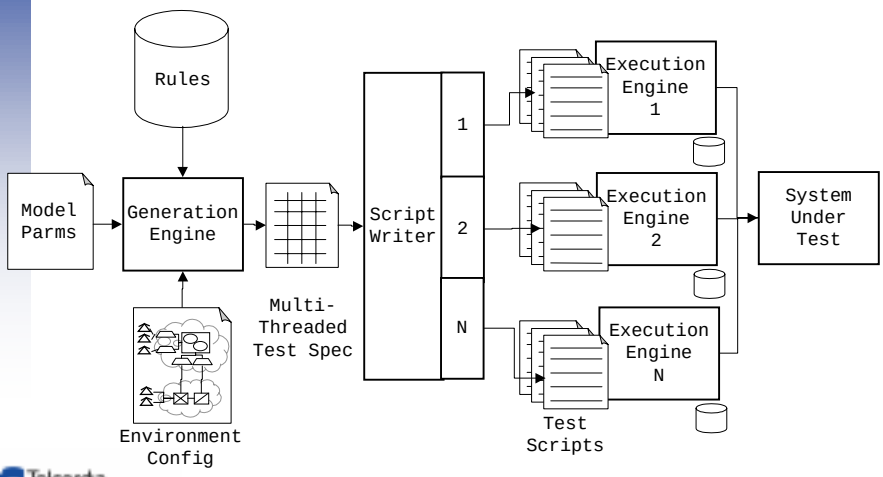
- Portability Objective
  - Independence from any one execution tool or scripting language
  - Test scripts quickly portable to different distributed test environment configurations and different systems under test
- Approach
  - Events are defined at multiple levels of detail (hierarchies of abstraction)
  - Event maps relate specifications and implementations
  - Event maps are execution tool specific



Telcordia Technologies

Event Verification Engine

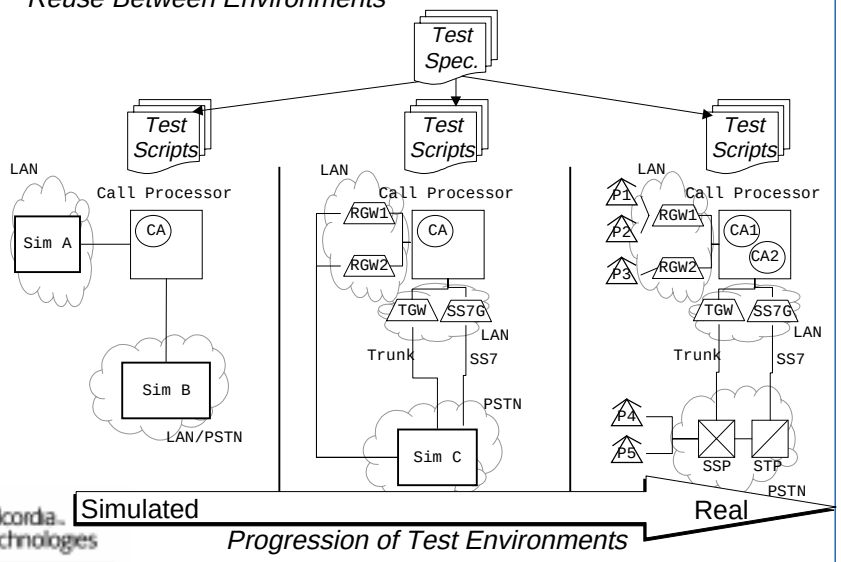
"Automatically Generated Test Scripts""



Telcordia Technologies

Portable Test Cases

"Reuse Between Environments"



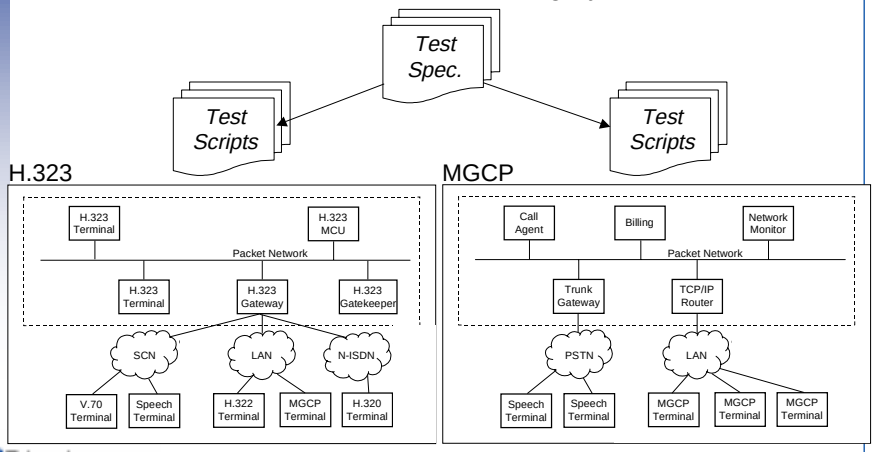
Telcordia Technologies

Event-Based Test Generation for Distributed Systems -- 13

Portable Test Cases

"Reuse Between Systems"

Two Multimedia Conferencing Systems



Telcordia Technologies

Event-Based Test Generation for Distributed Systems -- 14

Our Experiences
"Lessons Learned"

- Correctness and completeness emphasized
  - Extended definition of test coverage
  - Software Reliability Engineering (SRE) and "When to stop testing?"
- Integration into software development process
  - Requirements traceability
  - Test plan review
  - Tracking test cases and defects
- Maintenance and finding defects
  - Changes localized to new events and rules
  - Runtime failures more complex to debug than traditional approach
- Race conditions
  - Non-deterministic events were automatically detected during design and avoided during execution

Telcordia Technologies

Meeting the Testing Challenge
"Testing in the Internet Age"

- Integrated test automation strategy using a verification framework
  - Framework approach for testing systems with reusable component architectures
  - Test specifications and implementations generated simultaneously early in the development lifecycle
- Built a verification framework called the Event Verification Engine
  - Generated tests for a family of event-driven distributed systems
  - Combined functional and load testing objectives into the same tests
- Test streams and portable test cases
  - Reusable multi-threaded test specifications and implementations

Telcordia Technologies

Optimum Level of Test automation for Multi-Platform Client/Server Software

Heesun Park, Ph.D
SAS Institute, Inc
sashsp@sas.com

Test Automation for Client/Server

- Cost of Test Automation
- Cost of Maintenance
- Test Portability
- Level of Test Automation
- Defect Reproduction and Debugging

Client/Server Service in SAS

- SAS/SHARE
  - Data Base Management
- SAS/CONNECT
  - Remote Computing Service
  - File Transfer Service
  - Remote Library Service
- SAS/IntrNet

Version 6 Test Environment

- Multiple Platforms
- Multiple Access Methods
  - MVS (XMS, APPC, TCP)
  - CMS (IUCV, APPC, TCP)
  - VAX (DECNET, TCP)
  - UNIX (TCP, APPC)
  - PC (TCP, NETBIOS, APPC)

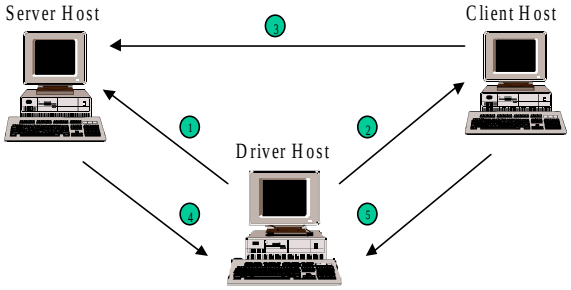
Version 6 Test Strategy

- Prioritization of test combinations
  - Installation Base
  - Marketing / Tech Support / Trend
- Selection of test cases for each combination
  - SRET Principle
- Full test automation
  - Use of SAS/CONNECT

Test Portability

- Should handle host and access method differences
- Extensive use of SAS macros
  * sas -sysparm mvs#h8x$tcp
  * &svrname=share1
  * libname mylib server=&svrname;

V6 SHARE Test Automation



1) Sign on to Server Host and Bring Up a SHARE Server.

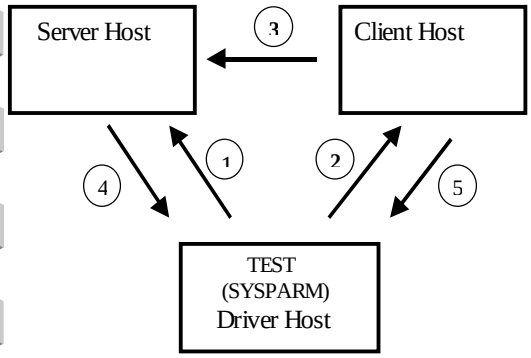
2) Sign on to Client Host and remote submit SHARE test code.

3) SHARE test runs from Client Host to Server Host.

4) Stop the Server and download the Server log.

5) Test completes on Client Host.

Version 6 Test Automation Scheme



1 Sign on to server host and bring up share server
2 Sign on to client host and remote submit share test code
3 Share test runs from client host to server host
4 Stop the server and download the server log
5 Test finishes on client host

Version 6 Test Process Review

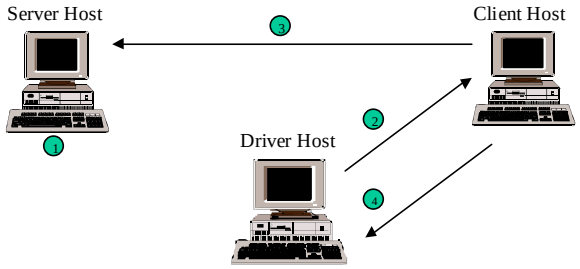
- Major achievement in client/server test automation
- Can run any client/server combination in fully automated manner
- Test setup overhead is fairly high
- Defect reproduction in isolation was not easy

Version 7 a New Challenge

- "..test automation always breaks down at some point."

  - very vulnerable to version change
- Minimize test setup overhead
- Need a tool to run tests easily in interactive mode so that developers can reproduce defects by themselves

V7 SHARE Test Automation



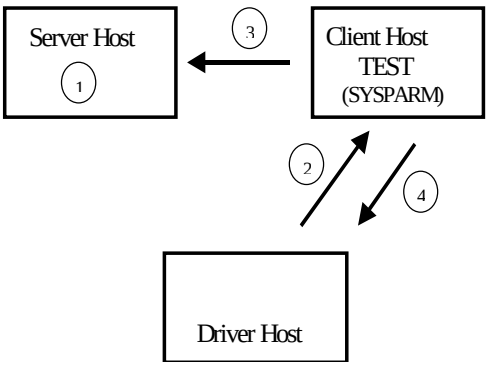
1) Bring up SHARE Server on Server Host (manually or with script).

2) Submit the test directly to the Client Host.

3) SHARE test runs from Client Host to Server Host.

4) Test completes on Client Host and results are moved to Driver Host.

V7 Test Automation Scheme



1 Bring up share server on server host (with script or manually)
2 Submit the test directly to client host.
3 Share test runs from client host to server host
4 Test finishes on client host and results are moved to driver host

Changes in V7 test process

- Long running server
- No longer collects server log
- Use of same tests with new macro definition
- No longer use SAS/Connect remote session on client side for single user test

Test Process Improvement

- Level of automation is lower
- More test throughput
- Maintenance of server is not expensive
- Server gets more stress test
- Handling of server log

Running automated tests interactively

- Defect reproduction becomes more difficult as the automation level grows
- Developers do not want to learn test automation scheme or logic
- Quick and easy way to reproduce defects
- %shrsvr; and %shruser ; macro
- Run any test interactively on any host

Conclusion

- Software testing is “craftsmanship”
- Understanding your test process is most important
- Full test automation may not produce maximum throughput
- Be ready to run automated tests interactively

Optimum Level of Test Automation for Multi-Platform Client/Server Software

Heesun Park, Ph.D
SAS Institute, Inc.
Cary, North Carolina
Sashsp@sas.com

Abstract

This paper is based on our testing experience of two major releases (version 6 and version 7) of the multi-platform client/server software, SAS/SHARE product which is the database management system for SAS users. It describes our test automation strategy for version 6 and identifies the achievements and weaknesses of our approach and explains how we improve our test process. The test process improvement effort for version 7 includes reduction of test automation level, use of macros that facilitate interactive execution of automated tests. It examines the different levels of test automation in client/server environment and presents how to achieve the optimum level that produces the maximum test throughput.

Introduction

Software test techniques and methodologies are developed and presented for applications in any software development process. But more often than not, these generalized techniques may not be the best fit for your specific software development and testing environment. Since software projects have different goals, designs and architectures, these software characteristics should be carefully considered in establishing test strategy. Client/server software is no exception and presents a unique challenge due to the number of combinations that we need to cover. Testing strategy for client/server software inevitably includes test automation. For successful test automation, one should consider many factors such as cost of test automation and maintenance, test portability, level of test automation, and the ability to run automated test interactively for defect reproduction and debugging. This paper shows our evolutionary path to test process improvement in client/server test automation at SAS Institute.

Background

The SAS system started out as statistical analysis software on MVS in mid 1970's. With major rewrite for portability in mid 1980's, it became an information delivery and application development software which runs on all major platforms including mainframes, UNIX machines and PCs. Currently, the SAS system is the leader in data warehousing, data mining and decision support server arenas. There are many factors that

contribute to the success and the expansion of the system. They include the portability of the application on multiple platforms, seamless client/server support on all platforms and capability of extracting data from many relational database management systems.

Application portability gives users freedom to choose hardware without losing their investment in application development. Also, the portable tests, which are SAS applications, allow testers to cover huge number of client and server combinations in a reasonable time frame.

As for the client and server support, we provide file transfer, remote computing, and remote library service capability with the SAS/CONNECT product and multi-user database management support with SAS/SHARE.

Another key ingredient to the success of the SAS system lies in its basic data storage format. From the beginning, SAS system used a “table” format for its data storage, which later became the de facto standard for all contemporary relational database systems.

In this paper, we will concentrate on test automation strategy for SAS/SHARE product, which is supported on all client/server platform combinations.

Version 6 Test Strategy

SAS Version 6 represents the first portable system targeted for multiple platforms. The last release of version 6 system was supported on 11 platforms including MVS, CMS, VAX, SUN, HP, IBM RS6000, OS2, WIN95 and WINNT. Client and server combinations for SAS/SHARE product were supported on all platform combinations and in many cases with multiple access methods on one platform. For example, on MVS, we support four different access methods: Cross Memory Services (XMS) for clients on the same CPU, Advanced Program to Program Communication (APPC) protocol for clients on IBM SNA networks, VTAM Logical Unit 0 (LU0) protocol for MVS mainframe network clients, and the TCP/IP access method for clients on UNIX and PCs. Roughly, we have more than 250 client and server combinations to cover and it is impractical to cover each and every combination with full test suites.

Our test strategy focuses on prioritization of test combinations, selection of test cases to run on each combination and full test automation, to provide maximum coverage with minimum risk in a reasonable time frame.

As for the prioritization of combinations, we consider inputs from marketing, technical support and development groups. In most cases, it is not difficult to identify major and important combinations since we clearly experience the proliferation of UNIX boxes and PCs.

Selection of test cases to run on each combination becomes very important since we may have to resort to a small subset due to our test resource constraints. I used principles introduced in SRE (Software Reliability Engineering) test method by John Musa [1]. In short, we came up with most likely “operational profile” or the scenario that users may use most often and selected test cases that covered this scenario. Later, this subset was used as a maintenance test suite.

Test automation for client/server environment is a unique challenge by itself. SAS has very sophisticated homegrown test tool that you can use to submit test tables in a fully automated manner to a single host. Our test automation effort focused on the test programs themselves. Our objective was to run the test tables for any client and server combination from the driver host. A test program should be able to determine client and server hosts dynamically from the command line input, run the test case accordingly, collect server log and client log for comparison with a benchmark. This client and server test automation logic will be explained in detail shortly.

Test Portability

It is worthwhile to mention our portable test structures. It is not uncommon that we have host specific environments and access method specific options in any client and server combination. For instance, server name may vary by access method or security options may differ by operating system. It is very clear that we cannot afford multiple copies of the same test solely because of host differences. We made our tests fully portable by utilizing the SAS macro facility extensively. Like other macro languages, SAS macro facility is basically a text substitution tool with some conditional statement capability. The SAS system also provides a sysparm option (it reminds me of good old MVS JCL) that you can use to pass in any character string that can be accessed within the SAS system. We exploit this option and pass in client, server, access method and other host specific information as a sysparm value and process this information, converting the single sysparm value into multiple global macro variables which are available throughout the SAS session. The portable tests are written with macro variables whenever necessary. These macro variables are resolved dynamically before test execution. Here is an example:

1) sysparm is specified with SAS invocation

```
sas –sysparm  mvs#h8x$tcp –autoexec shrauto
```

2) specified macro, shrauto, is called automatically and generates macro variables by parsing the sysparm value:

```
&server_node = sdcmvs
&client_node= vivaldi.unx.sas.com  (for h8x, or HP, machine)
&server_name=share1
&access_method=tcp
```

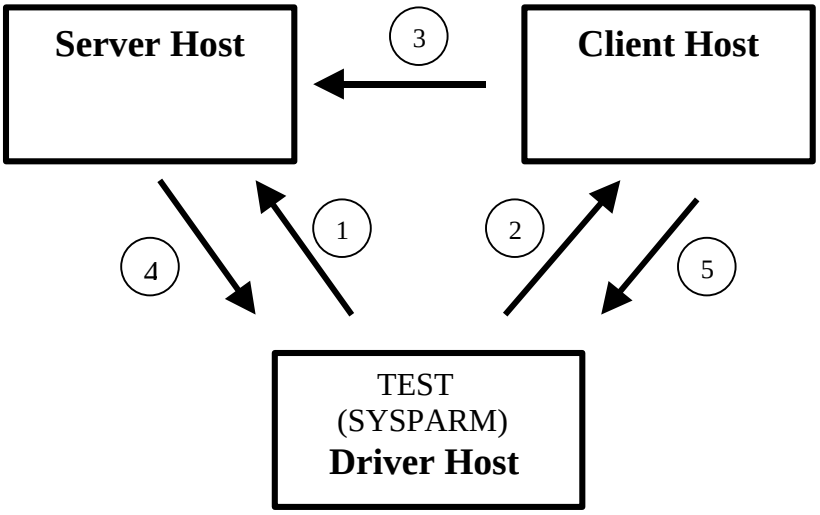
3) portable tests use these macro variables

```
options comamid=&access_method;
libname  mylib server= &server_node..&server_name;
```

All our portable tests are developed on our base development machine, which is HP UNIX, and get ported to all platforms on a weekly basis.

Version 6 Test Automation Scheme

As mentioned earlier, our objective was to run our test tables on any client/server combination from a driver host in a fully automated manner. We achieved this goal by using our own SAS/CONNECT product as a vehicle to connect (signon) to client and server hosts as depicted in Figure 1.



1 Sign on to server host and bring up share server
2 Sign on to client host and remote submit share test code
3 Share test runs from client host to server host
4 Stop the server and download the server log
5 Test finishes on client host

Figure 1. Version 6 test automation scheme

As the test begins execution on a driver host, it connects to the server host, brings up a SHARE server, connects to the client host and brings up a remote SAS session on the host. SHARE tests are remote submitted from the driver host to the client host and test results are returned to the driver host. At the end of the test, the server is stopped and its log is downloaded to the driver host for comparison.

This process was a major achievement in test automation for the client/server environment covering multiple platforms and their combinations.

Towards the end of the version 6 test cycle, we were able to run our test suites nightly in a fully automated fashion. When everything went well it produced fairly high level of test output. Version 6 was shipped slightly behind the schedule.

Version 7 a New Challenge

In his paper on test automation, Pettichord [2] appropriately pointed out that***, "test automation always breaks down at some point"***. I totally agree. Test automation scheme

is known to be vulnerable to product or version changes but ironically it is not that easy to predict the problems until you face them.

During version 7 development, we have experienced a great deal of frustration with our test automation process since our SAS/CONNECT product itself is under development for version 7 and fails time to time. Another drawback of this process is extremely high overhead for test set up such as signon, signoff, uploads of test information and downloads of server logs, etc. It is really handy for nightly cron jobs but rather painful to use during the day. We also notice that it requires some amount of rework when we try to recreate a problem in isolation without the SAS/CONNECT environment in the picture for developers.

Version 7 Test Automation Scheme

As an alternative, I have proposed and implemented a new SHARE server testing strategy using "Long Running Server". This strategy includes the following changes and advantages:

- A SHARE server needs to be brought up by hand or by a separate step on target host machine. To make this chore simple, I wrote a macro, %shrsvr, which will bring up server on any host. This SHARE server does not get stopped on every test. It stays up (long running) for at least the whole table. It eliminates the server manipulation overhead (signon to host, bring server up, bring down server, get server log for each test).
- For single user tests, which account for more than 90% of the entire SHARE test suite, we do not need to use a SAS/CONNECT remote session any more and all tests run directly to the long running SHARE server. We still use the same test suites without any changes in test source or benchmarks by blanking connect related macros in the test with a new autoexec macro.
- Another change is that we no longer collect and compare server logs for each test any more. A rationale behind this decision is that if anything goes wrong on server, it should surface on the client side in one way or another. When that happens, we check out the server log manually. We also save a lot of overhead involving server log manipulation.
- One may point out that we lose a level of test automation that we enjoy since we have to bring up a SHARE server separately (not in the test program itself). To some extent, this is true. But it is rather easy to provide one more activity in the script that runs the regularly scheduled batch jobs. For nightly cron jobs, we bring up a long running server from a separate test table that houses the server program before we run the test table that contains single user tests. For multi-user SHARE tests, we still take advantage of the long running server and use SAS/CONNECT remote sessions for multiple clients.

Figure 2 below shows our new version 7 test automation scheme:

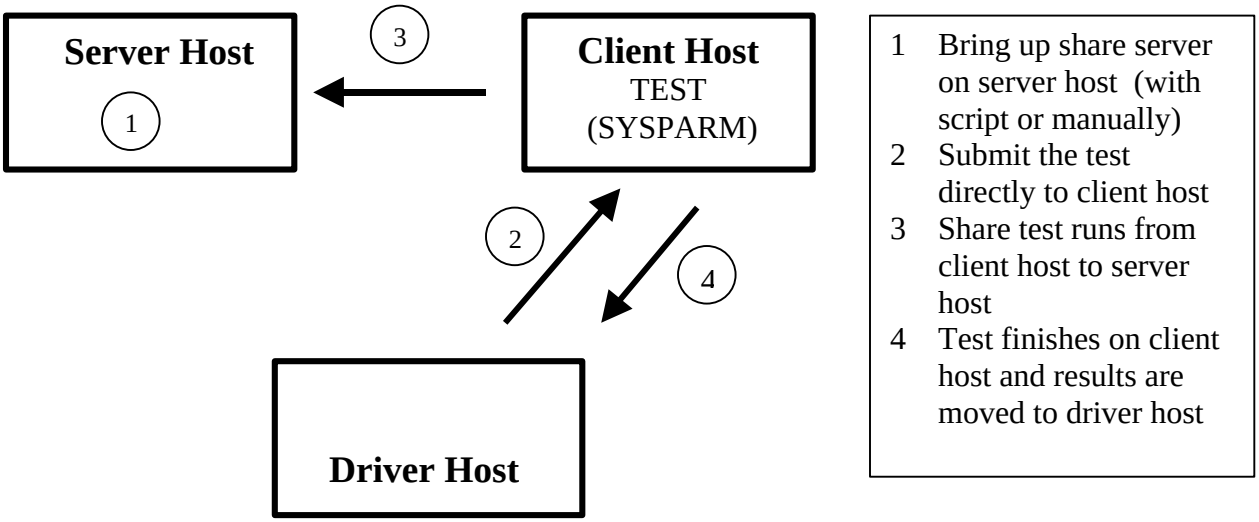


Figure 2. Version 7 test automation scheme

Notice that after the long running SHARE server comes up on a server host, test cases are directly sent to the client host for execution. This change eliminates the lengthy signon and signoff process that we used for version 6 and reduces the test execution time very significantly. Also, server logs are no longer collected for comparison.

Test Process Improvement by De-automation

We have noticed drastic improvement in test efficiency. For example, our level 2 test table that houses about 100 tests used to take 3 hours to complete with version 6 process, but with our new version 7 test automation scheme it took less than 25 minutes. This is one good example of increasing test output by reducing the level of test automation.

Our version 6 defect analysis [3] indicates that we got more than the expected number of defects from our tech support group and beta sites where they stress tested the version 6 SHARE product in a user environment. I have to acknowledge that version 6 SHARE server did not get much stress test since server's life cycle was very short. With the "long running server" strategy on version 7, the SHARE server runs longer and definitely runs under a stressed environment, which gives us a better idea on server survivability and "failure intensity"[1].

One big hidden cost of test automation lies in the fact that as the level of test automation grows, it becomes increasingly difficult to reproduce the defects in isolation. Defect description by testers gets lengthy and time consuming, and sometimes it is not well received by developers because it looks more complicated than it should. According to my experience, most developers are very reluctant to learn a test automation scheme or

anything related to that subject. From the developers' point of view, recreating the defect in a simple and reliable manner is most important in debugging and fixing the defects quickly. To save both the testers' and developers' time in defect description and defect reproduction, it is very important to have a tool that allows one to run automated test cases interactively in isolation.

Running Automated Tests Interactively

As mentioned above, one of the shortfalls of test automation is the difficulty of defect reproduction in isolation. I believe good test automation strategy in general and especially for client/server software should include tools that facilitate interactive running of automated tests. During the version 7 cycle, I proposed and implemented two macros that make testers' and developers' job much easier.

The first macro is %shrsvr macro, which brings up a SHARE server on any host. By default, it brings up the same server that is used by automated tests. The second macro, %shruser, is for client side. On any client host, it provides all necessary test set up based on current environment and macro parameters and includes the test case that we want to run interactively. For example, if our automated test***, sqltest***, fails from UNIX client to MVS SHARE server during the batch test, we can easily recreate the same problem interactively as follows:

- Bring up sas session on MVS and issue : %shrsvr;
- Bring up sas session on unix and issue : %shruser(snode=mvs, test=sqltest);

With version 7, we have more test combinations due to our backward compatibility support. In other words, we have to touch combinations such as server on version 6 and client on version 7 and vice versa. It is very obvious that full test automation on these combinations is not a good investment at all [2] considering the maintenance costs involved. As an alternative, we developed very comprehensive mega tests that contain coverage for all major functions and features and ran them interactively with the macros mentioned above on cross host and cross version combinations. This process gave us comfortable level of coverage on a vast number of combinations.

Icing on the Testing Cake

These two macros were very well received by both developers and testers and contributed heavily in strengthening developer / tester relations. At the end of the version 7 cycle, one of our developers sent us following comments:

"… I want to give kudos to SHARE testers. … those testing macros made testing, and more importantly debugging, almost a breeze. "

Our successful test organization and teamwork was very well depicted by Langston [4], in her award winning paper "How to have a perfect 'T' party".

Conclusion

Each and every software development and testing environment is different and has unique characteristics. In most cases, general test guidelines may not be directly applicable since software testing is more of a craftsmanship [5] rather than an industrial production. The same principle applies to test automation of multi-platform client/server software. Consequently, it is very important to understand the effects and implications of different levels of test automation for a given environment. My experience described in this paper is one good example of increasing the test throughput by reducing the level of test automation and the use of a macro facility that supplements the shortcomings of test automation in a client/server environment.

References

[1] J. Musa, "More Reliable, Faster, Cheaper Testing through Software Reliability Engineering – Overview ", Testing Computer Software conference, Washington, D.C., June 1998

[2] B. Pettichord, "Success with Test Automation", Quality Week'1996 conference, San Francisco, CA., May 1996

[3] H. Park, "611 IDB Defect Analysis", SAS Internal Testing News Letter, Cary, NC., 4th Quarter, 1995

[4] B. Langston, "How to have a perfect 'T' Party", STAR'98 East conference, Orlando, FL., May 1998

[5] P. Jorgensen, "Software Testing : A Craftsman's Approach", CRC Press, 1995

Automating Client/Server Testing in the Real World

Speaker: Huey-Der Joseph Chu
Institute of National Defense
National Defense Management College
Email: jchu@casq.org
URL:http://www.casq.org/jchu/

Automating Client/Server Testing in the Real World

- Outline
  - Introduction
  - Mobile Agents
  - The Architecture of VISITOR
  - A General Structure of Agents
  - Automated Test Execution Through VISITOR
  - Conclusion

Introduction

- Current testing tools
  - Capture/Playback paradigm
    - emulate a multi-user environment
    - testing on client sites
  - two-tier client/server applications
    - in the real world
      - multi-tier client/server applications
      - testing for middleware or on server sites

Introduction

- VISITOR
  - a flexible infrastructure for mobile agent computing
  - likes visitors
    - move around one site to another for their particular goals
- A simple 3-tier banking application
  - with an integrated test environment

Mobile Agents

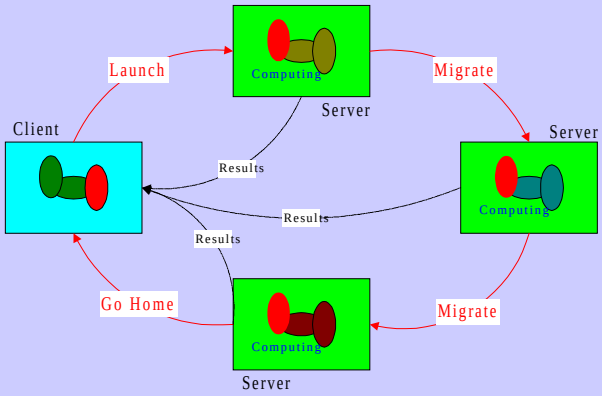
- An Agent
  - an object that is autonomous enough to act independently
  - receive requests from external sources
  - help users perform some tasks

Mobile Agents

- Mobile Agents
  - a computer object that can move through a computer network under its own control,
    - migrating from host to host
    - interacting with other agents and resources
      - to satisfy requests made by its clients.
  - Move around on behalf of their users seeking out, filtering and forwarding information or even doing business in their own name

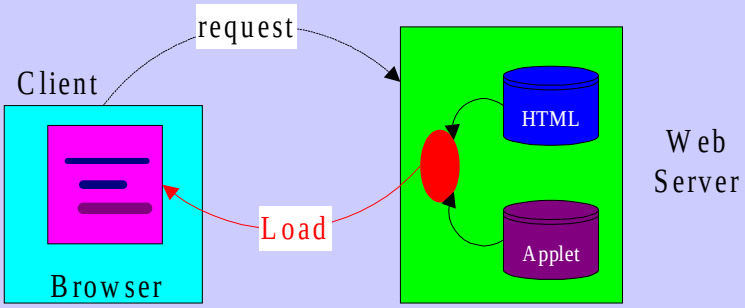
Mobile Agents

- Mobile Agents



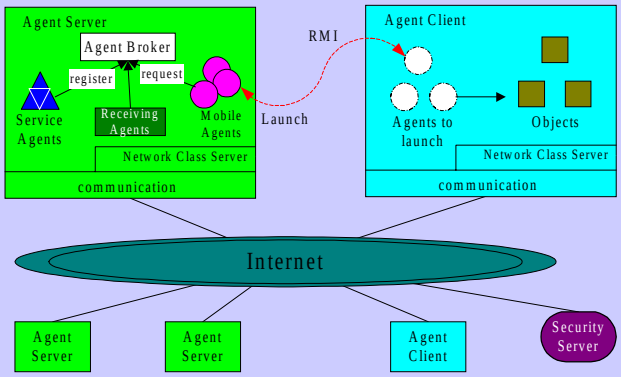
Mobile Agents

- Java Applet



VISITOR

- The architecture of VISITOR

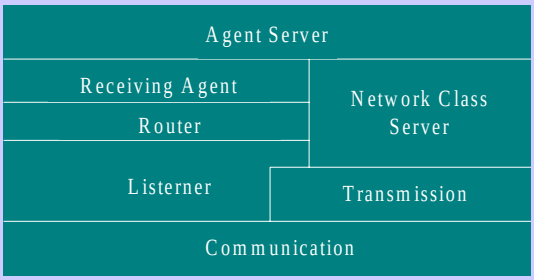


VISITOR

- VISITOR
  - supports flexible communication and co-operation between agents and local agents
  - provides some services through agent broker
  - combines with the Java RMI
    - launches tasks out
    - sends the results back

General Structure

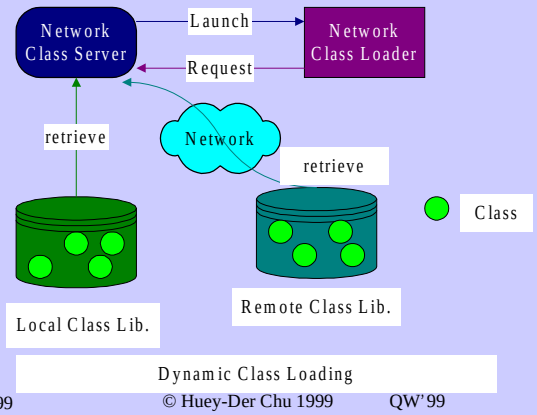
- Receiving Agents



Layers of a Receiving Agent

General Structure

- Network Class Server

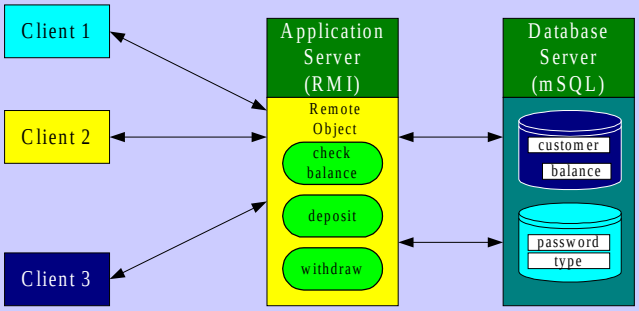


General Structure

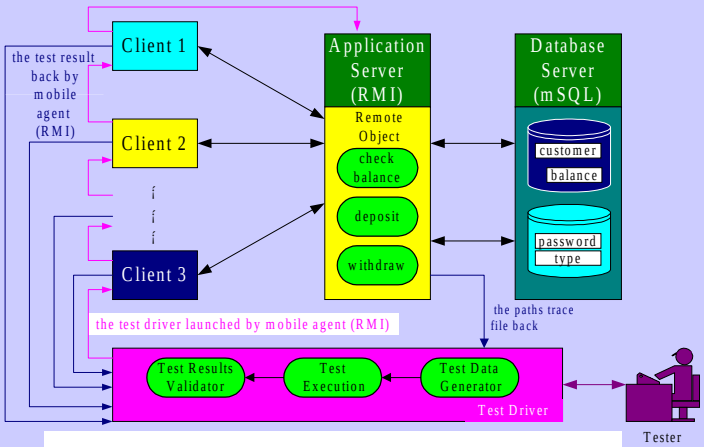
- Communication between Agents
  - KQML
  - example:
- (evaluate :sender kbase :receiver agent
- :language KQML :ontology agent
- :content (tell-resource :type address
- :name AB :value mis.ndmc.edu.tw:5001))

Automated Test Execution

- A simple banking application



Automated Test Execution



Automated Test Execution Through a Mobile Agent

Automated Test Execution

```
shelltool – /bin/sh
IOException: java.io.FileNotFoundException: /home/njrc1/MobileAgent/Agent/Statio
nary/scripts/init_file2
come here61
*** ReceiveAgent(beta): Running MTA !
*** RemoteClassLoader: Attempting to load java.rmi.Naming
*** RemoteClassLoader: Attempting to load Example.agents.implementation.Service_
Stub
*** RemoteClassLoader: Attempting to load java.rmi.server.RemoteStub
*** RemoteClassLoader: Attempting to load java.rmi.server.RemoteObject
*** RemoteClassLoader: Attempting to load java.rmi.dgc.Lease
*** RemoteClassLoader: Attempting to load java.rmi.dgc.VMID
*** RemoteClassLoader: Attempting to load java.rmi.server.UID
*** RemoteClassLoader: Attempting to load Example.agents.implementation.RMIInter
face
*** RemoteClassLoader: Attempting to load System.context.ComingContext
*** RemoteClassLoader: Attempting to load java.lang.String
*** RemoteClassLoader: Attempting to load java.lang.StringBuffer
*** RemoteClassLoader: Attempting to load java.io.PrintStream
I'm here***********
*** RemoteClassLoader: Attempting to load java.lang.Class
*** RemoteClassLoader: Attempting to load COM.imaginary.sql.msql.MsqlDriver
*** RemoteClassLoader: Attempting to load java.sql.DriverManager
*** RemoteClassLoader: Attempting to load java.sql.Connection
*** RemoteClassLoader: Attempting to load java.sql.ResultSet
Sampling No: 1 Sampling Size = 10
Generate Test Data 1 ......
*** RemoteClassLoader: Attempting to load java.io.File
*** RemoteClassLoader: Attempting to load java.lang.Math
Test Execution 1 .....
*** RemoteClassLoader: Attempting to load java.lang.Runtime
*** RemoteClassLoader: Attempting to load java.lang.Process
```

Automated Test Execution

- The Test Driver is launched to remote client sites
- Testing tools are dynamically loaded by Network Class Server
- Run the test on the client site
- The test result will be sent back

Conclusion

- The real world operating environment
  - across multiple platforms
  - middleware and server site testing
- The application with mobile agents
  - the interaction can be inspected
  - the test can be run on multiple clients across different platforms
- MOST site: *http://www.casq.org/most/*

Automating Client/Server Testing in the Real World

Huey-Der Chu

Department of Information Management
National Defense Management College
ChungHo, Taipei 23500, TAIWAN
jchu@mis.ndmc.edu.tw

Abstract

This paper applies mobile agents to automated test execution across multiple platforms. A test driver can be launched by a mobile agent to remote client sites to run the tests. During the testing, the mobile agent will use the Network Class Server to dynamically load the classes relevant to this test. The test result on each client site will be sent back by the mobile agent. The mobile agent roams across different platforms and finally it arrives at the application server site to bring back the trace file for inspecting the interaction behaviour among clients. This work can really let the tests run on multiple client sites across different platforms.

Keywords: Mobile Agents, Automated Test Execution, 3-Tier Client/Server Applications

1 Introduction

Current testing tools with capture/playback paradigm have some limitations for client/server applications [8,9]. The common one is a lack of consideration of the real world operating environment across multiple platforms. These testing tools emulate a multi-user environment and ends at the client site but are not designed to test the server, therefore, their products may not provide a way to test the effect of multiple users of the software. Moreover, these testing tools focus on two-tier client/server applications. However, the Gartner Group found 80% were planning for multi-tier (at least three-tier) client/server applications [8]. For client/server applications, some test scripts for middleware can be very hard to capture/playback automatically [9], for example, the communication mechanism between clients and servers uses technology like an RPC protocol that current capture/playback tools cannot effectively capture.

This paper introduces a flexible infrastructure for mobile agent computing: VISITOR [2], which can support flexible communication and co-operation between mobile agents and local agents which may provide some services through the agent broker. Furthermore, combining with the Java Remote Method Invocation (RMI), mobile agents can make use of distributed objects to accomplish such tasks as sending the results back to the home machine. These local agents are like offices which receive visitors and provide some services to them, while the mobile agents are like visitors which move around one office to another for their particular goals.

Based on VISITOR, a simple three-tier banking application with an integrated test environment has been implemented, which is big enough to address middleware testing issues such as Java RMI and JDBC. The test driver is launched by a mobile agent to remote client sites to run the tests. During the testing, the mobile agent will use the Network Class Server to dynamically load the classes relevant to this test such as the Test Data Generator and Test Results Validator. The test result on each client site will be sent back by the mobile agent. The mobile agent roams across different platforms and finally it arrives at the application server site to bring back the trace file for

inspecting the interaction behavior among clients. The application of VISITOR to automated test execution can let the tests really run on multiple client sites across different platforms.

Firstly, in this paper, the concept of mobile agents is introduced; secondly, the framework of VISITOR is proposed; thirdly, the application of VISITOR to the automated test execution is described; fourthly, the concept of automated test execution through mobile agents across multiple platforms is illustrated on a three-tier client/server application with Java RMI and JDBC and finally, summarizes the work and offers suggestions for further study.

2 Mobile Agents

An agent is an object that is autonomous enough to act independently even when the user or application that created it is not available to provide guidance and handle error. Agents can receive requests from external sources, such as other agents, but each individual agent decides whether or not to comply with external requests. In the computer world, an agent is a computer program whose purpose is to help a user perform some tasks (or set of tasks) [6]. To achieve this aim, it maintains a persistent state and can communicate with its owner, other agents and the environment in general. Agents can do routine work for users or assist them with complicated tasks. In addition, they can mediate between incompatible programs and thus generate new, modular and problem-oriented solutions thus saving work.

Mobile agents [1,3] provide a new alternative paradigm for distributed object computing on the WWW. A mobile agent is a computer object that can move through a computer network under its own control, migrating from host to host and interacting with other agents and resources in order to satisfy requests made by its clients. They may move around on behalf of their users seeking out, filtering and forwarding information or even doing business in their own name. Possible applications for mobile agents include information retrieval, data-mining, network management, electronic commerce, mobile computing, remote control and monitor, etc. Therefore, mobile agents show a way to think about solving software problems in a networked environment that fits more naturally with the real world.

The concept of mobile agents is in contrast to the concept of Java Applets. In the latter case, a program is downloaded from remote computers to execute locally, while in the former, a program is sent to remote machines to execute remotely, When mobile agents execute remotely, there may not be any transactions in the home machine. The advantages of mobile agents are [3]: firstly, they offer an effective paradigm for distributed applications, particularly in partially connected computing; secondly, they can provide a pervasive, open, generalized framework for the development and personalization of network services; thirdly, they move the programmer away from the rigid client/server model to the more flexible peer-to-peer model in which programs communicate as peers and act as either clients or servers depending on their current needs and fourthly, they allow ad-hoc, on-the-fly applications that represent what would be an unreasonable investment of time if a code had to be installed on each network site rather than dynamically dispatched.

Nowadays, there are already some frameworks for mobile agents, such as the Aglet and the Java-to-go. They all support dispatching a segment of code to remote machines to execute, however, they do not give proper support to the co-operation between mobile agents and services in remote machines. This next section introduces a flexible infrastructure for mobile agent computing: VISITOR [2], which can support flexible communication and co-operation between mobile agents and local agents which may provide some services through the agent broker. Furthermore,

combining with the Java Remote Method Invocation (RMI), mobile agents can make use of distributed objects to accomplish such tasks as sending the results back to the home machine. VISITOR shows a paradigm for service-providers to provide services and for service-clients to get services in a networked environment that fits more naturally with the real world.

The application of VISITOR to software testing, Mobile Testing Agent [5], has been implemented and can be downloaded at the MObile Software Testing (MOST) wesite (http://www.casq.org/most/) constructed and maintained by Huey-Der Chu 1998.

3 The Architecture of VISITOR

The architecture of the VISITOR is shown as in Figure 1 which consists of a network of agent servers, agent clients and a security server.

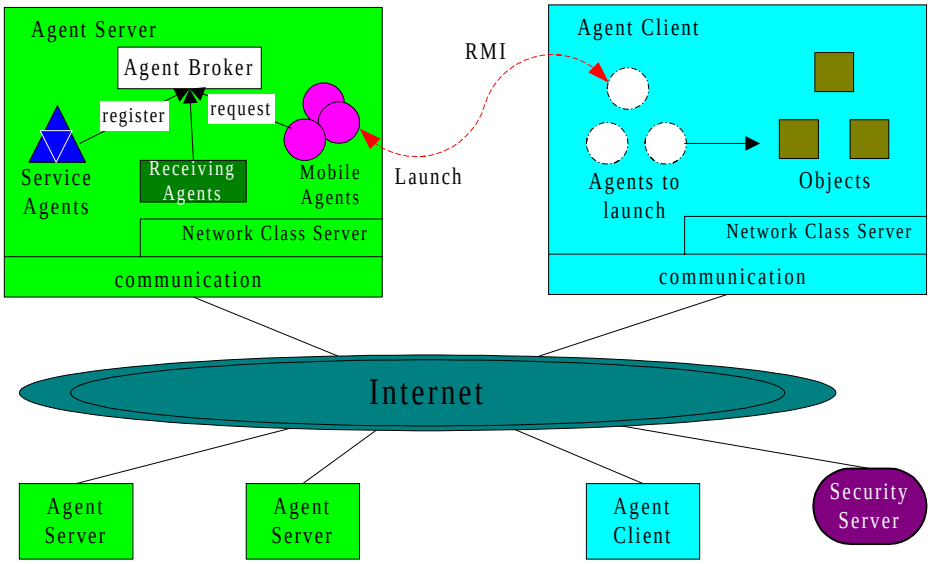


Figure 1: The architecture of VISITOR

These components communicate with one another based on Java sockets. Agent servers are destinations which mobile agents want to visit. Agent Servers are also the hosts which accommodate mobile agents and provide services to them. Agent Clients are applications which launch mobile agents to the agent servers for accomplishing their particular tasks. In this framework, the agent servers are like offices which receive visitors and provide some services to them, while the mobile agents are like visitors which move around one office to another for their particular goals.

3.1 Agent servers

In each agent-server, there are five types of components: The Agent Broker (AB), service agents, the receiving agent, mobile agents and the network class server.

An AB is a stockbroker among agents. All other agents have to be registered with the AB. The AB keeps them as resources. When an agent is created, it sends a message to the AB to register its existence and address. When an agent A wants to communicate with another agent B, A first transmits a message to the AB to ask B's address. The AB would acknowledge with B's address if B exists. When B first receives A's message, it also need to ask the AB for A's address. Afterwards, A

and B would communicate with each other directly.

Furthermore, if an agent, for example a service agent, could provide some service, it would send the AB a message to register that service. When some agent, for example a new coming mobile agent wants the service, it would request the AB. If the service has been registered, the AB would return the agent's address that can provide that service. Then, they would dialogue directly as normal.

Service agents provide services for other agents. When they are created, they would register the service with the AB which they can provide. The services they can provide are various, from general information services (e.g. databases) to particular commercial services (e.g. purchasing some CD at the lowest price).

It is the receiving agent that is responsible for receiving and instantiating coming mobile agents. It also creates execution environments and forks a thread for the agent run. There is only one receiving agent in each agent-server. For the structure of the receiving agent, see section 4.1.

Mobile agents come from remote agent clients. When they arrive, the receiving agent creates the execution environment for them and they would register with the AB. Together with main classes, a knowledge base which include initial information is sent. The receiving agent will save this knowledge as a specific file. A mobile agent will run in a separate thread to accomplish its tasks. It can also make use of services which are provided by execution environments or service agents. For the structure of mobile agents, see section 4.2.

The Network Class Server listens to the network. If there is a request for loading a class from this machine, it is responsible for finding, loading and sending the class to the destination. When a mobile agent is launched, only the main class is sent. the auxiliary classes are loaded on demand from the home machine or the previous machine, where a network class server is set up.

3.2 Agent clients

Agent clients design and launch mobile agents for accomplishing their particular tasks. The clients may be located in an agent-server or in a separate machine. For the latter, a network class server has to be set up for remote class loading. In the case where there is no network class server set up, the agent launcher has to send all class of the agent, or the class loader would fail.

Arriving at remote agent servers, mobile agents can execute home transactions by the Java RMI. For example, when a mobile agent retrieves information in remote agent servers, it can make use of the RMI to display the result on the home machine simultaneously.

The picture above characterises a flexible agent-oriented method of constructing client applications, producing a new paradigm for distributing computing.

3.3 Security server

The security Server is listening to the network. When clients want to launch a mobile agent for accomplishing their particular tasks, they have to register with the security server to gain a key, which the mobile agent will bring with it. The agent servers will check the key to see whether or not it is valid. If the key is valid, the process will continue, if not the server will send back an error message to the client.

4 A General Structure of Agents

The static structure of an agent is designed following the Java Agent Template (JAT). An agent consists of three parts: a message handler, a resource manager and a knowledge base. The message handler sends and receives Knowledge Query and Manipulation Language (KQML) messages by the communication interface *Comminterface*. The message handler is also responsible for message processing.

The resource manager is responsible for managing resources which the agent possesses. There are five types of resources: Languages, interpreters, classes, files and addresses in the JAT.

The knowledge base includes the initial information of agents and the information about the services which it can provide. When the agent moves from one machine to another, the information in the knowledge base will move along.

An agent executes within a *AgentContext* which is the execution environment of the agent. Agents could make use of services in the agent-server by the *ContextInterface* which is implemented by the *AgentContext*. When an agent arrives at a new agent-server, the receiving agent will initiate the agent with the knowledge base, which is sent with the underlying agent. When an agent leaves the machine, it will clean up the environment. The *initiate* and *cleanup* methods are provided by the *AgentInterface*.

Dynamically, an agent is a thread. When an agent moves to a new agent-server, a new thread is created, on which the agent is running.

4.1 Structure of the Receiving Agents

The receiving agent inherits from a general agent but the receiving agent has its specific functions in VISITOR. The layers of a receiving agent are as shown in Figure 2.

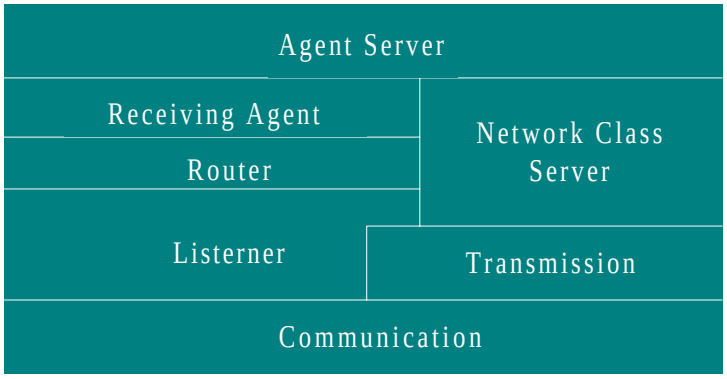


Figure 2: Layers of a Receiving Agent

When the *receiving agent* starts up, it forks a thread to execute the *Router*, which in turn forks a thread to execute the *Listener*. Based on Java Sockets and severSockets, the *Transmission* layer provides semantics of the agent-packet transmission. The *Listener* is monitoring the network to see if a new packet is coming. If so, the *Listener* makes use of the methods provided by the *Transmission* layer to receive the packet and pass it to the *Router*. The *Router* unpacks the packet

and instantiates the coming agent first, then checks if the underlying machine is the destination of the agent. If not, the *Router* would rout the agent to the correct machine. If it is true, the *Router* would initiate the agent, create its execution environment and pass it to the *receiving agent*. The *receiving agent* forks a new thread to execute the new coming agent.

It is the Network Class Server (NCS) that implements the dynamic class loading. The principle of dynamic class loading is shown as in Figure 3.

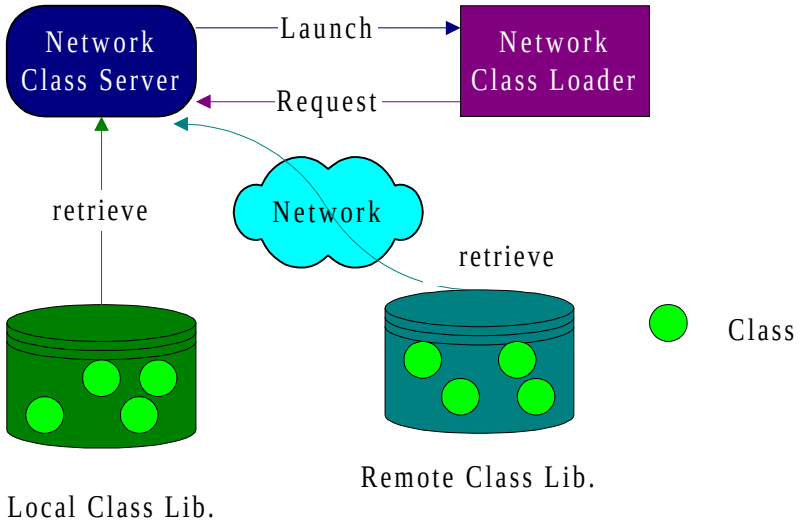


Figure 3: Dynamic Class Loading

The NCS is listening on the network, when a Network Class Loader (NCL) asks for classes it will find, load and transport the classes. The NCS not only can load classes from the local class library but can also load classes from a remote class
library.

The layer structure of a NCS is as shown in Figure 4. Like the agent-server, it is also based on Java *Sockets* and *serverSockets*.

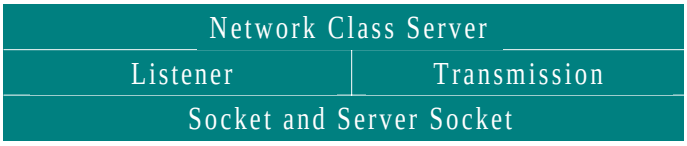


Figure 4: The Layers of Network Class Server

In the context of VISITOR, when the *Router* in the agent server *instantiates* a coming mobile agent, it will load the classes relevant to the agent dynamically.

4.2 Communication between Agents

Agents communicate with each other using the KQML [7], which is a high-level language intended for the run-time exchange of knowledge between intelligent systems.

Logically the KQML message consists of three layers: the content layer, the message layer, and the communication layer. The content layer includes the actual content of the message in the

programs' own knowledge representation of the message. KQML can carry expressions encoded in any representation language such as the Knowledge Interchange Format (KIF), the KQML or even ASCII strings.

The communication layer encodes a set of message features which describe the lower level communication parameters, such as the identity of the sender and recipient, and a unique identity associated with the communication.

It is the message layer that is used to encode a message that one application would like to transmit to another. The message layer forms the core of the KQML and determines the kinds of interaction one can have with a KQML-speaking agent. A primary function of the message is to identify the protocol to deliver the message and to supply a speech act or *performative* which the sender attaches to the content (such as an assertion, a query, a command or any of a set of known *performatives*). In addition, since the content may be opaque to the KQML-peaking agent, this layer also includes optical features which describe the content language, the ontology it assumes and some type of description of the content such as a descriptor naming a topic with the ontology.

Syntactically, a KQML message is a ASCII string called a *performative*, which consists of a performative's name and a list of its parameters. A parameter is represented as keyword/value pair. The keyword, that is the parameter name must begin with a colon and must precede the corresponding parameter value.

Here is an example of a KQML message, which is used as an initial message in our framework:

```
(evaluate :sender kbase :receiver agent
        :language KQML :ontology agent
        :content ( tell-resource :type address
                :name AB  :value mis.ndmc.edu.tw:5001))
```

In this message, the KQML *performative* is the *evaluate*, the content is (tell-resource :type address :name AB :value mis.ndmc.edu.tw:5001), another KQML message which tells the agent that the AB's address is *mis.ndmc.edu.tw:5001*, the ontology assumed is *agent*, the receiver and sender of the message are agent and kbase respectively, and the content is written in the language KQML.

The value of the content keyword is content level, the values of :sender and :receiver belong to communication level, and the *performative's* name (evaluate) with :language and :ontology form message layer.

When an agent ClientA moves to an agent-server, it would transmit a message like the following to the AB for telling its existence:

```
(evaluate :sender ClientA
        content (tell-resource :type address :name ClientA
                :value mis.ndmc.edu.tw:54100)
        ontology agent :receiver AB :language KQML)
```

Suppose that there was already another agent ClientB which sent the following message to the AB when it started up.

```
(evaluate :sender ClientB
        :content (tell-resource :type address :name ClientB
                :value mis.ndmc.edu.tw:54103)
        :ontology agent :receiver AB :language KQML)
```

When the agent ClientA wants to communicate with the ClientB, it would first send the following message to AB:

```
(evaluate :sender ClientA
        :content (ask-resource :type address :name ClientB)
        :ontology agent:receiver AB :language KQML)
```

The AB would answer with the message below:

```
(evaluate :sender AB
        :content (tell-resource :type address
                :value mis.ndmc.edu.tw:54103 :name ClientB)
        :ontology agent :receiver AB :language KQML)
```

After that, the ClientA would dialogue with ClientB directly.

5 Automated Test Execution Through VISITOR

5.1 A Simple Banking Application

A banking application is an embedded software system which is commonly seen inside or outside banks to drive the machine hardware and to communicate with the bank's central banking database. This application accepts customers requests and produces cash, account information, database updates and so on. In this Section, a Simple Banking Application (SBA) will be designed as a 3-tier client/server application as shown in Figure 5.

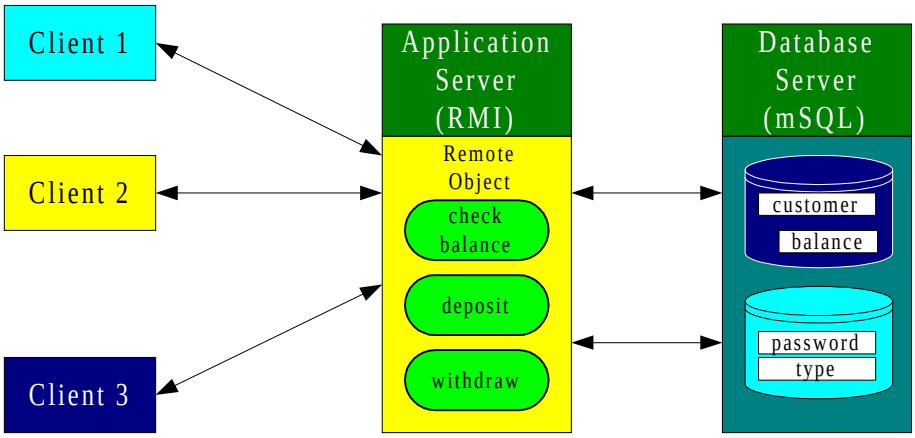


Figure 5: Three-tier System Structure

Within a banking enterprise, more specifically a corporate and distributed database collection for the personal data of customers, the balance status of customers, the password data and account type

data. The corporation seeks to assimilate their data sources into one virtual data store and access it through a common interface.

There are four business activities at this application: check balance, deposit money, withdraw money and print the statement. This standard transaction will accept customer requests (checking, depositing, withdrawing and printing) after the customer has input the account id, the account type and the correct password on the Client site. SBA will retrieve the balance from the database on the Database Server site, process the request on the Application Server site and save the balance back to the database. It also will produce the balance or print a banking statement to the customer. It has been implemented with an integrated test environment using Java RMI and JDBC [4].

5.2 The Application with VISITOR

The application of VISITOR to the automated test execution on the banking application is as shown in Figure 6.

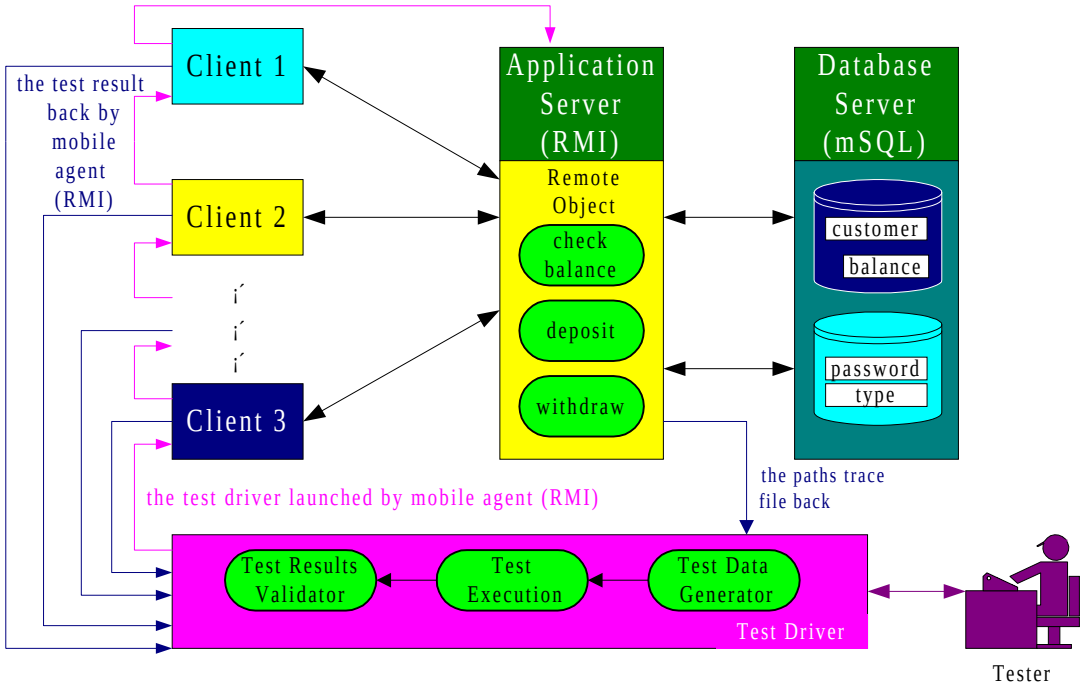


Figure 6: Automated Test Execution Through a Mobile Agent

The test driver sets up the test execution environment for the banking application, initiating the Test Data Generator to generate an input unit, sending it to the Test Execution to execute the application and getting the product unit and delivering it to the Test Results Validator. The test driver is launched by a mobile agent to remote client sites to run the tests. During the testing, the mobile agent will use the network class server to dynamically load the classes relevant to this test such as the Test Data Generator and the Test Results Validator. The test result (pass/fail) on each client site will be sent back by the mobile agent with Java RMI. The mobile agent roams across different platforms and finally it arrives at the application server site to bring back the paths trace file for inspecting the testing order.

This framework has been implemented with the Java Agent Template (JAT) and the Java Remote Method Invocation (RMI). The JAT provides a fully functional template for constructing agents

which communicate peer-to-peer with a community of other agents distributed over the Internet. However, JAT agents are not migratory but rather have a static existence on a single host. As an improvement, the Java RMI is used to let JAT agents dynamically migrate to foreign hosts in this implementation. As a result of the Java RMI not currently working effectively well on the Netscape Browser currently, the implementation of MTA (Mobile Testing Agent), the name of a mobile agent for the automated testing in this implementation, is not available with Java Applet, but with stand-alone style. It can be downloaded at the MObile Software Testing (MOST) web site (http://www.casq.org/most/) which is under the web site for Chinese Association for Software Quality (CASQ) constructed and maintained by Huey-Der Chu 1998.

6. Conclusion

Current testing tools with a capture/playback paradigm have some limitations for client/server applications. These tools emulate a multi-user environment and ends at the client site but are not designed to test the server. A mobile agent is a computer object that can roam over the Internet under its own control, migrating from host to host and interacting with other agents and resources in order to satisfy requests made by its clients. Based on the concept of mobile agents, the test driver can be launched by a mobile agent to remote client sites to run the tests and the paths trace file on the server side can also be sent back to the user for inspecting the test ordering. This concept has been implemented on a 3-tier banking client/server application with Java RMI and JDBC. It is completely different from current automated testing tools. The major advantages of this approach are the interaction behaviours between clients and server can be recorded in a paths tracer file which can be inspected and the tests can be really run on multiple clients across different platforms.

References

1. Chen, J., “A flexible framework for mobile agent systems,”. Available at http://www.casq.org/most/chen.ps.
2. Chen, J., Greenwood, S. and Chu, H., “VISITOR: A Java-based Infrastructure for Mobile Agent Computing,” *Proc. in 10th International Conference on Software Engineering and Knowledge Engineering (SEKE’98)*, June 18-20, 1998, San Francisco, USA.
3. Chess, D., Harrison, C. and Kershenbaum, A., “Mobile agents: Are they a good idea?” *Lecture Notes in Computer Science 1222*, 25-45.
4. Chu, H., *Distributed Testing: Towards Quality Programming in the Automated Testing of Distributed Applications*, Europe Arts, Science & Culture Publishing C. Ltd., Feb. 1999, ISBN 1-902409-07-8.
5. Chu, H., Dobson, J.E., Chen, J. and Greenwood, S., “The Application of Mobile Agents to Software Testing,” *Proc. in 15th International Conference and Exposition on Testing Computer Software (TCS’98)*, June 8-12, 1998, Washington, D.C., USA.
6. Lingnau, A. and Drobnik, O., “An HTTP-based infrastructure for mobile agents,” Available at http://www.w3.org/pub/Conferences/WWW4/.
7. Mayfield, J., Labrou, Y. and Finin, T., “Evaluation of KQML as an agent communication language,” Available at http://www.cs.umbc.edu/lait/papers/kqml-eval.ps.
8. Mooney, K. and Chadwick, D., “Overcoming the Challenges of Testing Client/Server Applications,” Available at http://www.rational.com/support/techpapers/challenges/.
9. Quinn, S.R. and Sitaram, M., “Shrink-wrapped and custom tools ease the testing of client/server applications,” *Byte*, Sept. 1996, 97-102.

ATRs (Atomic Requirements) Used Throughout Development Lifecycle

Hanania Salzer
RTS Software Ltd., Israel

Definition of ATRs

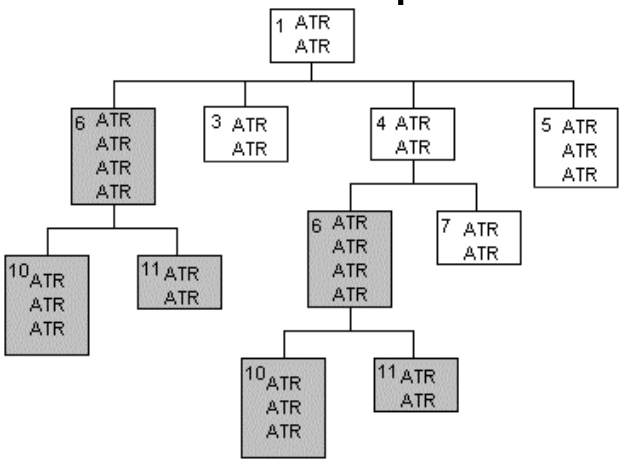
- Well formed requirements
  - Abstract, unambiguous, traceable, validatable (testable)
- Indivisible, elementary
  - Test result: clear pass or fail
- Not only client requirements
  - Also (and mostly) technical design

ATR Example

- *Non-atomic Requirement:*
  - After the customer has removed the magnetic card, a slip is printed with the transaction details.
- *ATRs:*
  - After the customer has removed the magnetic card, a slip is printed.
  - The transaction details are printed on the slip.

May 1999 Quality Week '99, San Jose, CA 3

ATRs Associated with Software Components



May 1999 Quality Week '99, San Jose, CA 4

Properties of ATRs

- Most ATRs are at the level of detailed design; originate from designers
- Easier to achieve unambiguity
- Less vulnerable to subjective phrasing and to errors in grammar and vocabulary
- Easy communicating with another person
- Explicitly associated with a software component

May 1999 Quality Week '99, San Jose, CA 5

Documenting Software Design, Tests

- The primary specification of a software component *(such as a program, an event, a window, a stored procedure)* is a <u>list</u> of ATRs
- List of ATRs provides permanent documentation
- Scope of test implies list of ATRs
- Tester must document new ATRs

May 1999 Quality Week '99, San Jose, CA 6

Documenting Standards

- Standard is expressed as a list of ATRs
- Only ATRs where a programmer may err
- Concatenated to software component's specific ATRs
- Single point of maintenance

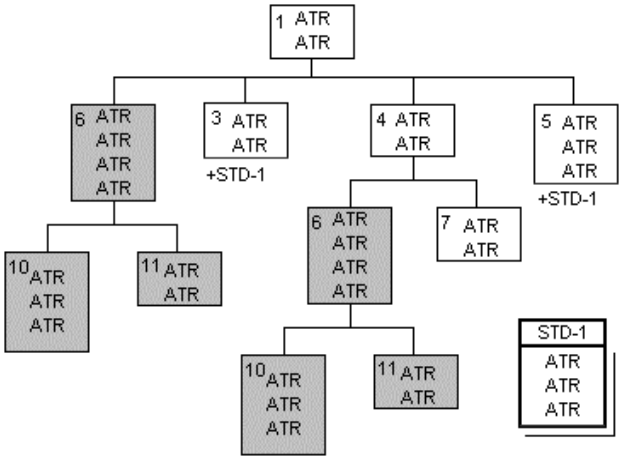
May 1999 Quality Week '99, San Jose, CA 7

Standard (Example)

- Dates and time of the day are displayed in the user's time zone
- Dates and time of the day are recorded in the database as GMT

May 1999 Quality Week '99, San Jose, CA 8

Standard



The Programming Phase

- ATRs make meaningful internal documentation
- ATRs can be copied-and-pasted from design into source code comment
- Programmers can add new ATRs
- ATRs make testing by a programmer -- unbiased

Controlling Work

- Determine ATRs' priority, *a priori*:
  - For programming, for testing
  - Controversy is avoided
- Express progress in terms of software functionality:
  - During programming, during testing

May 1999 Quality Week '99, San Jose, CA 11

Functional Coverage vs. Depth of Test

- Scope of test implies list of ATRs
- Modify implied list of ATRs
- Test coverage:
  - *Functional*: list of ATRs
  - *Semi-quantitative*: number of ATRs
- Each ATR has its own depth of test
- Two dimensions of control

May 1999 Quality Week '99, San Jose, CA 12

The Maintenance Phase

- Translate bug reports to ATRs
- *Bug* = Behavior deviation from ATRs
- *Enhancement* = Expectation for new ATRs
- Discuss one ATR at a time

May 1999 Quality Week '99, San Jose, CA 13

Implementation in a Hostile Atmosphere

- Communicating does not require the other party to acknowledge ATRs
- ATRs fit into most document formats (formal and informal)
- Useful for fragmentary documentation
- Testers can document their understanding of the software

May 1999 Quality Week '99, San Jose, CA 14

Education

- Some schools teach programming as a series of puzzles, emphasizing the “how” over the “what”
- Young students can easily build an attitude of segregation between requirements and coding

May 1999 Quality Week '99, San Jose, CA 15

CASE and Test Automation Tools

- A prototype tool implements much of what has been described
  - Integrates design, test management and maintenance
- Test automation may find in ATRs a relatively stable base to solve the moving target problem

May 1999 Quality Week '99, San Jose, CA 16

Conclusion: New Concepts

- ATRs provide the means to test exactly what the programmer must implement
- Standards are lists of ATRs inherited by specific designs
- Functional coverage of tests is segregated from the "depth" of the test

May 1999 Quality Week '99, San Jose, CA 17

ATRs (Atomic Requirements) Used Throughout Development Lifecycle

Hanania Salzer, RTS Software Ltd., Israel

Abstract

Atomic requirements (ATRs) are indivisible "well-formed requirements" (ANSI/IEEE-Std-1233-1996) that enable control over software design, test planning, and work management with an ease and accuracy not previously attainable. The use of ATRs is described in various phases of the software development lifecycle. New concepts are introduced:

- ATRs provide the means to test exactly what the programmer must implement.
- Standards are lists of ATRs inherited by specific designs.
- Functional coverage of tests is segregated from the "depth" of the test.

The presentation's nucleus is the notion of the atomic requirement (ATR), which, by itself, may not be new nor seem promising. During its use in over a decade, many benefits have become visible.

- A reduction has been noted in the number of potential misunderstandings between client and analyst, designer and programmer, designer and tester, etc.
- A list of ATRs can be used to communicate exactly the same detailed design to programmers and to testers.
- Development and testing work can be controlled on the basis of the ATRs' significance.
- Implementation of ATRs is very simple. In fact, a person can enjoy the benefits even without the cooperation of the organization where he or she works.
- ATRs can be applied to any format of design or test documents.
- ATRs are less vulnerable to language difficulties experienced by developers whose mother tongue is different from the one they have to use for software documentation.
- Organizations (small and large) that maintain legacy software, which is now too large to be comprehended, can incrementally document fragments of knowledge using ATRs. Similarly, testing teams that are provided with insufficient documentation of the software functionality can fill in the gaps with ATRs.
- Providers of CASE and test automation tools should consider using ATRs as the combining link between functional design and testing.

Copyright © 1999 Hanania Salzer

The presentation has two parts: a definition of ATRs (atomic requirements), and a description of their usefulness for achieving some of the developers' and testers' basic goals. ATRs seem to have properties that make them considerably more effective than non-atomic requirements in a large span of software development activities.

Some of the people who have been exposed to the notion of "atomic" in the context of requirements have found it hard to comprehend this concept. This resistance can be contributed to the (justified) desire to cover all functionality with requirements: a single atomic requirement seems to cover very little. But later in this presentation it will become clear that the collection of ATRs for a software component does cover all functionality, and probably does it better than a bunch of its non-atomic cousins do.

The presentation extrapolates personal experience into suggested practices and into further potential benefits.

Definition of ATRs

Atomic requirements are, primarily, "well-formed requirements" *à la* ANSI/IEEE Standard 1233-1996. "Well-formed requirements" are abstract, unambiguous, traceable and validatable (testable). In addition to all these, atomic requirements (ATRs) are also the result of splitting higher level requirements into elementary, or indivisible, requirements.

Consider, for example, this non-atomic requirement for a teller machine:

> **R11**: After the customer has removed the magnetic card, a slip is printed with the transaction details.

The requirement should be split into two ATRs:

> **R11-a**: After the customer has removed the magnetic card, a slip is printed.
> **R11-b**: The transaction details are printed on the slip.

Typically, an ATR is implemented by a few lines of code in a single software unit. Very few ATRs target much more code in several software units. Hence, an ATR is always associated with a software component.

Properties of ATRs

ATRs use non-formal language. We have developed guidelines for phrasing them. Programmers, designers, testers and technical writers get formal training in correctly phrasing ATRs.

Every ATR deals with only a single functionality. Therefore, it is considerably easier to achieve unambiguity with an ATR than with a non-atomic requirement. For the same reason, moreover, one can test each ATR independently.

When software is tested for compliance with a well-formed ATR, the result is always a clear pass or fail. In contrast, the result of testing for a non-atomic requirement may be partial success (or partial failure).

ATRs are less vulnerable to subjective phrasing (something that only the writer can understand.)

When ATRs are written in a foreign language, their intelligibility is less affected by errors in grammar and vocabulary.

A person can write ATRs when communicating with another person, without the need for the other party to understand ATRs, or even to know that what they are reading is an ATR. In fact, the two parties enjoy a better level of understanding than when documenting non-atomic requirements.

ATRs are explicitly associated with a software component, be it a software unit, or an integration of several of them. Hence, the location of any bugs targeted by an ATR becomes self-evident.

A list of ATRs can serve as the "work orders" both for programming and testing.

Client vs. Non-client Requirements

Many use the term "requirement" to denote only the requirements that originate from a client or user. This presentation uses the term in a much broader sense. Our requirements cover everything that the software must accomplish, regardless of "whose idea it was."

Think what is required of a subroutine; something the client is neither aware of, nor is likely to understand. What values the subroutine should return in different conditions, what data should be updated in the database, and what should be displayed in a window field are three examples. Designers create these kinds of requirements. In fact, the requirements that designers add largely outnumber the clients' requirements. Therefore, it is immaterial which ATRs have been requested by the client, and which the designers have added.

ATRs' Target Bugs

At the time of coding, the programmer views an ATR as an instruction for what the software must do. ("Purchase date must not be in the future.") At the time of testing, the same programmer views the ATR as a question. ("Can the purchase date be in the future?") A good question is necessary if one is to expect a useful answer.

A non-atomic requirement asks several questions simultaneously. When we look at a non-atomic requirement, we may see only some of the "questions" in it, and overlook the rest. This can happen to programmers as they write code according to a non-atomic requirement, or as they test according to the same non-atomic requirement. The first results in a bug. The second results in the test not looking for the bug.

Documenting Software Design and Tests

The main part of a software component's design is the list of ATRs that you, the programmer, must implement. After programming is completed, keep the ATRs as permanent documentation of the software's detailed functionality.

The functionality of any software component, no matter how small and insignificant it is, can be defined as a list of ATRs. The result is that designers and programmers are educated to understand the difference between functional and technical (e.g., structured English) thinking. They start understanding that "requirements" refer to the complete functionality of any software component, not only to user defined requirements.

You define the scope of a test by the software components included. Since every software component has a set of ATRs associated with it, the scope of the test implies the list of ATRs to be tested. You can refine the scope by removing ATRs from the test.

If you are an independent tester, and you want to test for an ATR that is not part of the documented design, you are looking for trouble. The programmer was probably not aware of it. You may want to document the new ATR in the design document, and communicate it to the developers.

Documenting Standards

Standards are, in fact, requirements that must be implemented in many software components. They are documented in the form of ATR lists that provide a single point of maintenance.

What ATRs to Include in a Standard

Assume a project that is developing an application that prints several reports. As a designer in the project, you could include the following ATR in your project's standard:

> *Standard for Reports*
> **R21:** When there is no data for a report – print a message on the first page.
> **R22:** At the bottom of the last page there is a message indicating the end of the report.

Other projects may have different standards.

The list of two ATRs in this standard has several purposes. The list communicates to designers *not* to repeat the two ATRs in the individual reports' designs. It also communicates to developers what functionality to develop. To testers it communicates what to test for.

However, any requirements already covered by tools used by your developers do not need standards. For example, if the application is to run under an operating system such as MS Windows, where the user can choose the date's display format, you would not include the following requirement in your standard: "Dates are printed in the format of dd-Mmm-yyyy." You do not want to write standards for the developers of a shrink-wrapped infrastructure tool, which you or your clients have purchased, even though the infrastructure may have bugs. Restrict the list only to requirements that the programmers have control over, and therefore, in which they have the chance to err.

How to Use the ATRs in Standards

The design of any software component, therefore, is a list of specific ATRs as well as pointers to standards.

In the above example ("Standard for Reports"), you need to specify all ATRs that are specific for a particular report, and state that it must follow the standards for reports. That being said, you have implied that **R21** and **R22** apply to the report. The programmer writes code for the report according to its specific ATRs, and according to the above two ATRs that originate from the standard.

The programmer applies both the specific ATRs and the ATRs that originate from standards. Correspondingly, the tester uses the same two sources of ATRs (specific and standard) as the developer.

What if the Standard Changes?

Standards may change: ATRs can be added, removed or replaced. For example, the project could decide to replace **R22** with a different ATR:

```
R22: The number of lines printed in the report is indicated
at the bottom of its last page.
```

All these changes are easily implemented at a single point, namely in the standard. By knowing where the standard is referenced, and by knowing the configuration of the software, it can be determined what part of the software needs modification in order to comply with the new standard, and what part needs to be re-tested according to the new ATRs.

The Programming Phase

Internal Documentation

Programmers need to document programs, usually by comment lines among the lines of the source code. If this documentation results in the same information that an average programmer would anyway deduce from the code, then the documentation is of no real value. Instead, it should provide information that cannot be coded, that is, the functional meaning of what the code lines accomplish.

For example, in a service call dispatching application, the following comment line is really redundant:

```
   emplyee_type_manager := “M”
   . . .
/*-----------------------------------------------------*/
/* When the employee type code is “M,” display an
/* error message.
/*-----------------------------------------------------*/
   if   employee_type = emplyee_type_manager
   then
      call error_message (a, b)
   endif.
```

However, the following comment tells a future reader of the code what business functionality it implements:

```
/*-----------------------------------------------------*/
/* A manager cannot be dispatched to a call.
/*-----------------------------------------------------*/
```

This comment line is, in fact, an ATR.

You may be writing program specifications for programmers. The most important part of a unit’s specifications is the list of ATRs. The programmer simply copies and pastes the ATRs into the source code, which then serve as internal documentation.

But that is not all. Programmers usually continue to elaborate the design with functionality that was not in the specifications they received. This new functionality must be documented in the very same way. Therefore, the programmer too must have the skills to phrase good ATRs. The skill to phrase good ATRs results in more meaningful (functional) comments among the lines of code.

Objectivity of Testing

There is a belief that programmers are usually biased when they test programs written by themselves. When the programmer writes code according to ATRs, they have to test the program against the same ATRs.

You do not want to fool yourself, therefore you write down all ATRs. Since the list of ATRs provides a 100% functional coverage of the program, you only need to follow the list, and test the program against each and every ATR. The result is a complete functional coverage of the test. The test is not carried out against a subjective interpretation of a complex narrative, but against ATRs that cannot be interpreted much further, hence they may be considered objective for all practical purposes.

Ensuring that all ATRs have been tested is a process that is not biased by the familiarity of the programmer with the code. The efficiency of the test for each ATR is covered elsewhere in this presentation.

Controlling Work

The task of developing a software component includes drawing up a list of ATRs. The list form makes it convenient for the involved parties to agree *a priori* on the relative significance of each ATR, and even on the exclusion of some of them. When time is limited, the list can be prioritized, and the risk can be taken that the ATRs at the end of the list will never be implemented.

It is the very nature of ATRs' that makes them so convenient to prioritize. First, they are maintained in the form of lists. Second, the lists are associated with software components thus handling them is streamlined with handling the natural tasks of writing programs one by one. Third, and probably most important, whereas a non-atomic requirement may include components with different levels of significance, there is no ambiguity in associating a single level of significance (or priority, if you wish) with an indivisible requirement.

Furthermore, frequently, the functionality that has the lowest significance to the application is a complex one, and has a high appeal to the programmer because of the challenge that it offers. By drawing up a list of ATRs, you are able to explicitly state that a certain ATR should be implemented only when all other ATRs are already in.

When significance is defined (or at least approved) by management, not implementing nice-to-have ATRs is acceptable. *A priori* agreement reduces the chances of an after-the-fact conflict. Note that instead of defining priorities as lists of software components, ATRs help to define them in terms of functionality.

Similarly, the tester can get an *a priori* agreement with management on what functionality will be placed at the bottom of the testing list, or what functionality will be knowingly left untested.

The progress of the development stage can be measured in the same way as progress of testing: How many ATRs have been implemented? How many ATRs have passed the test and how many have failed? For how many ATRs do we not know the results because they have not been included in the test? This counting of ATRs is only a semi-quantitative measure, because the cost to implement different ATRs is largely variable. The same information can be provided also as a list of ATRs. Thus we can produce reports such as "functionality that the software successfully implements," "functionality that we know the software has failed to implement" and "functionality that we have no information about." Both the quantitative and the descriptive reports describe the software's quality.

Avoiding Controversy

When a team leader agrees with a programmer on the list of ATRs, it is easy to tell whether a flaw in the program is the result of non-compliance with a certain ATR or a flaw in the ATRs themselves (frequently, it is a missing ATR.)

The result is clarity of responsibility and the avoidance of misunderstandings.

Functional Coverage vs. Depth of Test

Testers use the list of ATRs to control one dimension of a test – its functional coverage. Testers also have the freedom to control another dimension – the "depth" to which each ATR is being tested.

Scope of Test Implies List of ATRs

You can define the scope of a test by listing the software components included in it, be it a single software unit or a chunk of an application. Knowing that each software component is associated with a list of ATRs, the list of software components implies a list of ATRs. The software must be tested for compliance with all of the implied ATRs in order for the test to achieve 100% functional coverage.

Functional Coverage

What ATRs to include in a test, and what not, may depend on a number of reasons. For example, you could include in one test only performance related ATRs. In another test you may want to verify only ATRs related to a bug that has been fixed, and a few more that are most likely to be affected by the fix. Yet in another test, you simply pick ATRs of the highest significance to the upcoming release, and (what a shame!) not test at all for the rest of the ATRs.

In order to get a full functional coverage, you must test for all ATRs, even if it means splitting the ATRs among several tests.

You can express the dimension of functional coverage in a number of ways, depending on what you are looking for. It can be the number of ATRs included in a test, as compared to the total number of ATRs in the application under test. Alternatively, it can be the number of ATRs already tested at a certain point of time, as compared to the same total number of ATRs in the application or as compared to the number of ATRs included in the test. This way or that, the number and the proportion are objective parameters of the functionality covered by the test. (Admittedly, the measures are, at the most, semi-quantitative.)

Depth of Test

You can chose to make just the right amount of test, less than that, or overkill the test. Testing too little or too much could be the result of lack of knowledge in testing theory. Testing too little could also be the result of a decision to save time, and pay with the lack of knowledge about the software's quality. I call the amount of test per a single ATR the *depth* of its test. An ATR can be tested to the correct depth, to not deep enough, or to an excessive depth. By the way, the shallowest test for an ATR is not to test for it at all.

The testing for each ATR is independent of the testing for all other ATRs. Hence, testers can vary the testing "depth" of each individual ATR.

The method for testing for an ATR is out of this presentation's scope.

Two Dimensions of Control

This approach provides you with control over two independent dimensions of your tests. One dimension is the functional coverage, which is controlled by the number of ATRs selected for a test, or the total number of ATRs tested for, in a series of tests. The second dimension is the depth of test, which is controlled by the effort put into testing a certain ATR.

The segregation between these two dimensions provides you with additional control over the use of your resources. A frequent source of frustration results from the conflict between what should be tested and the resources available for the test. You know that you must cut off corners (or even more than that) and the question is what to cut. There is a wealth of good literature to help answer the question. In addition, you want to control what functionality to test more, and what to test less, on an ATR by ATR basis. Thus, understanding the segregation between the two dimensions – coverage and depth – endows the decision-making process with a very fine granularity.

The Maintenance Phase

During the maintenance phase of software development reports are provided of bugs, or what users believe are bugs. This is the phase in the software's lifecycle when communication with the client could be more problematic. ATRs might help.

Analyzing Bug Reports

An undesired symptom implies a requirement with which the software does not comply. This requirement can be expressed as one or more ATRs.

As a minimum, users report undesired symptoms by describing the software's actual behavior, and remarking that this behavior is unacceptable. A more useful bug report also includes what the user expects the software should have done. Before you rush to fix the bug, write down the ATRs that you think the user has assumed. If you can, communicate the ATRs to the user or to a business matter expert, and ask for a confirmation. This simple process may prevent you from having to fix a non-existent bug, from partially fixing the bug, or from having to fix the bug only to implement another bug. The user may be thinking of one specific situation, but by communicating your understanding in the form of ATRs, both of you may realize the more general case, and come up with more comprehensive ATRs.

If the bug report indicates that the user has an incorrect understanding of how the software should behave, send them the relevant ATRs. Although an example could be useful to illustrate the abstract ATR, it should not replace it in the explanation.

Bugs vs. Enhancements

The difference between a reported bug and a request for enhancement may be the subject of dispute with the customer. ATRs can provide insight to help solve this kind of dispute.

A customer complains. There is no question that the software does not behave as the customer expects it to behave. As a developer, you are interested in segregating between two extreme cases: one when functionality is missing, and the other when the customer assumes the existence of functionality. In the first case, the ATR can be found in the software design. This means that the customer expects a functionality that the software should have implemented. Because the software does not behave according to the documented requirements, the symptom is attributed to a software bug. In the second case, the ATR is a new one. This means that the customer expects new functionality that the software was not expected to have implemented. Hence it is assumed the customer is requesting an enhancement. True, the lack of the ATR may be the result of a bug in the original requirements.

Discussing Requirements

Errors in requirements are considerably easier to spot in ATRs than in non-atomic requirements. That is why ATRs are found to be a useful format in communicating among parties that cannot meet directly, for example, when corresponding with a remote customer who requests an enhancement.

We have often experienced virtually endless transatlantic telephone conversations that yielded definitions that still did not seem to satisfy everybody. But when prior to making the telephone call we sent over ATRs, communication was ablaze, and both parties expressed confidence that the other party had understood them.

Implementation in a Hostile Atmosphere

Individuals and teams who do not get management support for process improvement can enjoy the benefits of using ATRs without inducing conflicts with other parts of their organization. This is because they can write ATRs into any document type mandated by their organization without damaging its structure. They can humbly write useful ATRs even in non-formal documents.

ATRs in any Document Type

ATRs can be used in many different document types, including, but not restricted to, requirements specifications, functional specifications, technical specifications and even physical data models. You can write ATRs into most formats of design document without disturbing the document's structure, and still enjoy the above-described benefits.

Indeed, you can also enjoy the benefits of ATRs in informal documents. End users, business experts and IS professionals alike appreciate the clarity in which ATRs express the intentions of the writer. For that reason, handwritten instructions given to programmers, minutes of meetings and plain letters are among my favorite places to use ATRs.

Testers with Insufficient Documentation

Even when you need to test an application with insufficient documentation, ATRs may come to the rescue. Document your understanding of how the software should work, by writing ATRs. Write bullet lists of ATRs, grouping them in a way most convenient for you (by windows, by the order you plan to test them, or whatever.)

Request the developers to approve the ATRs before the tests start. Every ATR they will correct is likely to save you one more false alarm.

Finally, test against these ATRs, and use them to report anomalies.

Education

My personal wish is that schools will oblige students not only to learn how to "write programs", but also to learn how to write requirements in the form of ATRs (which are not to be confused with structured English.) This would contribute to the education of future generations of the software development work force and teach them how to comprehend the difference between the "what" and "how" notions.

CASE and Test Automation Tools

A prototype tool has been developed and is in use by several companies. The tool implements the use of ATRs, as described above. It captures the software's design as a hierarchy of software components and the associated ATRs. It depicts standards as a list of ATRs, with the lists being linked to software components.

Testers use the tool to design tests for a desired functional coverage. The tool also supports multiple, concurrent versions of the software's design. This is necessary to simultaneously support the versions in production, under test and in development. It supports multiple, concurrent, different versions of the software's testing and maintenance. Maintenance means, in this context, capture and resolution of user complaints (bugs) and user requested enhancements.

The approach is so simple and robust that an application's ATRs, which have been recorded by a word processor, could be conveniently migrated into the tool.

Toolmakers may want to use the concepts described here to develop interface between non-formal-language, functional design, CASE tools and automated testing tools.

Test suite maintenance is reported to be one of the largest ongoing expenses of test automation. In addition to the already described benefits, a testing tool that supports ATRs may help streamline test suite maintenance in view of software changes from version to version. Test tools' scripts are at the same abstraction level as the code they test. Both stem from the requirements. ATRs, in contrast, are themselves the requirements. Consequently they have the potential of providing a relatively stable "Archimedes Fulcrum." In addition, ATRs also have the necessary fine granularity to address every single feature of the software under test.

Use-Cases Are Not Requirements

Steve Meyer

AT&T
sameyer@att.com
513-629-7321

Larry Apfelbaum

Teradyne SST
larry@sst.teradyne.com
603 879-3555

©Teradyne Inc. All rights reserved, 1999

What is a Requirement?

- Webster**:**
  - **"something required: something wanted or needed"**
- IEEE*:
  - **1) a software capability needed by a user to solve a problem or achieve an objective;**
  - **2) a condition or capability that must be met or possessed by a system or a component in order to satisfy a contract, specification standard or other formally imposed documentation."**

*Source: Dorfman M, and Thayer R. Standards, Guidelines and Examples of System and Software Requirements Engineering. Washington D.C. IEEE 1990

©Teradyne Inc. All rights reserved, 1999

What is a Use Case

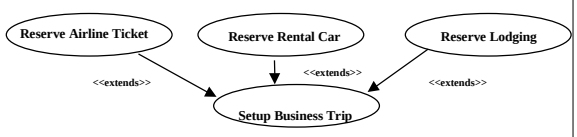


Figure 1 Use Case Diagram for Business Trip

©Teradyne Inc. All rights reserved, 1999

Understanding the Cost of Uncovered Requirements

- **Requirement volatility can cause significant overruns and/or delays in test/project completion**
- **Requirements that have not been verified can result in missed market windows, or lower then expected profit margins**
- **Inaccurate or incomplete tracking of requirements leads to uncertain test quality**
  - **Complexity of project and limited time may limit test to nominal conditions only**
  - **No explicit way of measuring test completeness**

©Teradyne Inc. All rights reserved, 1999

Relative Cost to Fix an Error

Phase in Which Found	Cost Ratio
Requirements	1
Design	3-6
Coding	10
Development Testing	15-40
Acceptance Testing	30-70
Operation	40-1000

Source: Gause & Weinberg [references data from Boehm]

©Teradyne Inc. All rights reserved, 1999

The Compounding Effect of Defects

Requirements	Functional	Design	Code
Correctly defined requirements	Correctly defined functional description	Correctly designed specifications	Correctly coded software
Incorrectly defined requirements: missing, misunderstood, ignored, outdated, unneeded, unspoken (assumed)	Erroneous functional specs based on incorrectly defined requirements	Design errors based on incorrectly defined requirements	Coding errors based on incorrectly defined requirements
	Incorrectly defined functional specs	Design errors based on incorrectly defined functional specs	Coding errors based on incorrectly defined functional specs
		Design errors	Coding errors based on design errors
			Coding errors

Even good work on a faulty base will result in a defective product

Source: A. Davis. Software Requirements; J. Taft, Nationsbank

©Teradyne Inc. All rights reserved, 1999

Cumulative Effects on a Multi-Stage Process

Scenario	Stage->	Requirements	Functional	Design	Code
All Stages at 90% Correct	% Work Done Correctly	90%	90%	90%	90%
	Cumulative Defect %	90%	81%	73%	66%
All Stages at 85% Correct	% Work Done Correctly	85%	85%	85%	85%
	Cumulative Defect %	85%	72%	61%	52%

At an 85% quality level almost half of the code delivered will be defective!

©Teradyne Inc. All rights reserved, 1999

What Use Cases Lack

Sequence and flow of operations

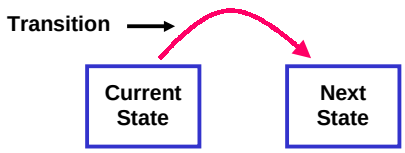
Frequency and arrival rate information of individual Use Cases

Use cases often describe ***only best case*** and limited exception information

How the system(s) are used, or could be used

©Teradyne Inc. All rights reserved, 1999

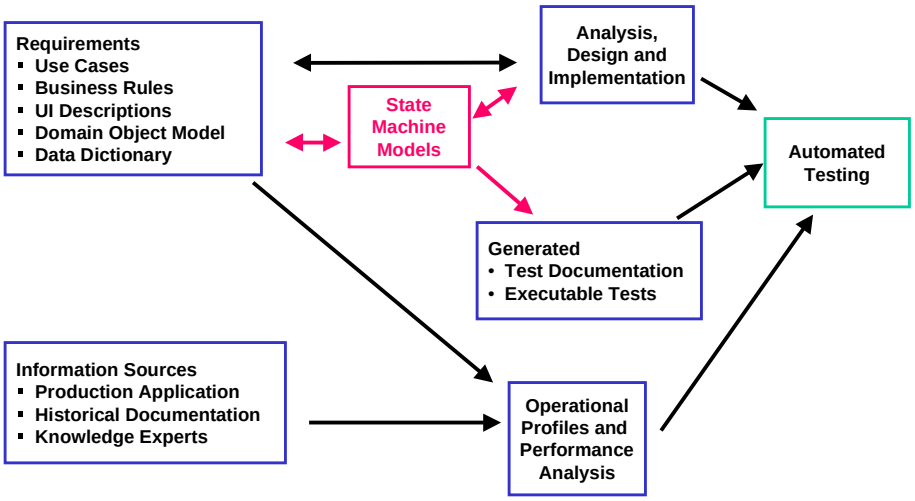
A Finite State Machine is Composed of States and Transitions



- Transitions define actions that move the system from the current state to a new state
- Transitions can be based on specific stimuli and/or previous actions
- Text descriptions of user actions can also be associated with transitions
- Use cases are built by concatenating the descriptions from a sequence of transitions through a diagram

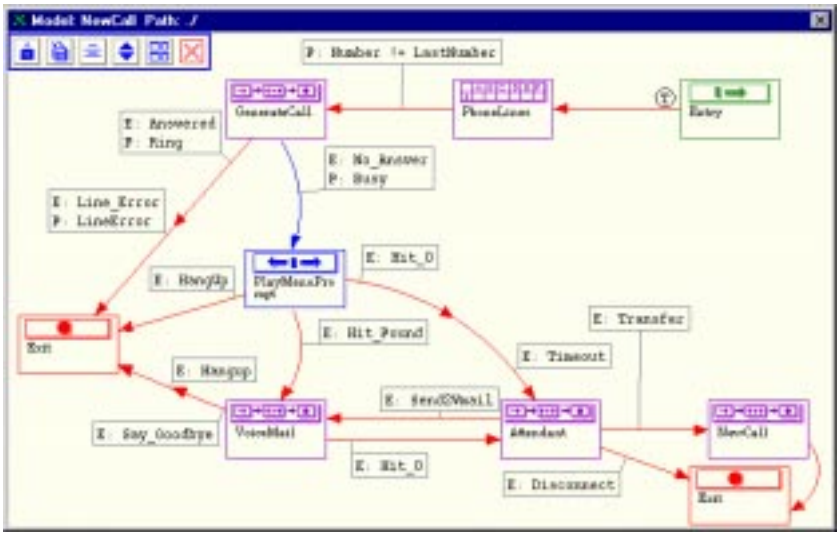
©Teradyne Inc. All rights reserved, 1999

System Development Process



©Teradyne Inc. All rights reserved, 1999

VoiceMail Model Example



©Teradyne Inc. All rights reserved, 1999

Each Requirement can be Defined in Terms of Model Objects

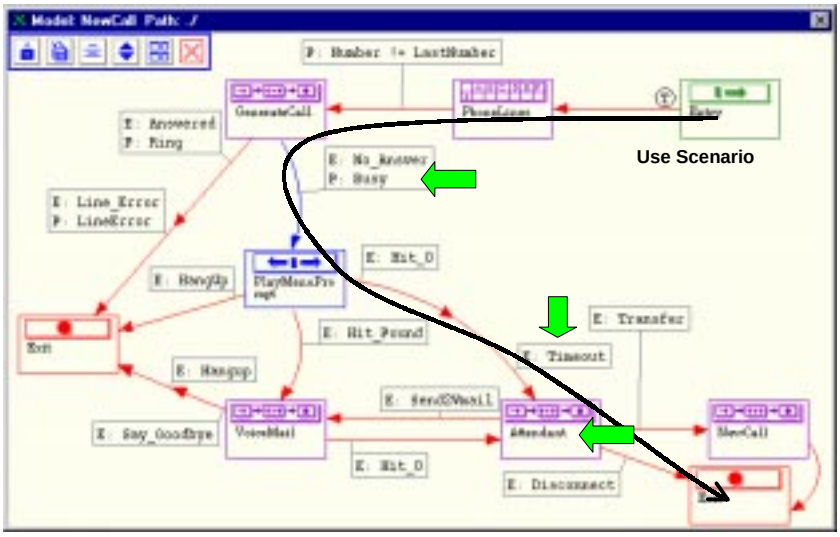
Requirement *timeout_transfer* :
On a busy line, the system shall transfer a caller to the attendant after the specified timeout period

Becomes:
***timeout_transfer* = Busy && (Timeout -> Attendant)**

-> is an 'is followed by' operator

©Teradyne Inc. All rights reserved, 1999

Each Path Defines a Test / Use Scenario



©Teradyne Inc. All rights reserved, 1999

Each Path Automatically Creates Test Documentation & Scripts

Description for Path: 6

1. **Setup protocol.**
2. **Initiates a call.**
3. **Generate busy signal on line 12.**
4. **Verify pre-recorded answering prompt.**
5. **Initiate a delay of 10 seconds.**
6. **Verify recorded message for attendant**
7. **Caller Disconnect. Verify dialtone.**

Path Probability : 0.0002075487

```
/*                    START TEST*/
/*Define Line protocol*/
/*Start App*/
'                    Initialize Line
'$Include "subs.sbl"
Sub Main
if declareProtocol () then goto TelError
/*Generate incoming call*/
digits = get_first_extension ()
if seizeLine () then goto TelError
' got dial tone, now dial in
if dial (digits) then goto TelError
/*Normal call w/attendant and Voicemail*/
/*Generate Call to this line*/
/*Make line Busy*/
if makeBusy (12) then goto TelError
if expectBusy () then goto TelError
/*No Answer..*/
/*Verify Menu Prompt*/
if expectVoice ("Thank you for calling") then goto TelError
/*Delay for timeout transfer to attendant*/
Pause timeout_delay
/*Cannot have same number as line,*/
/*Attendant == 0 means none specified*/
/*Verify Attendant Message*/
if expectVoice ("Hello can I help you") then goto TelError
/*Caller disconnects*/
if disconnect () then goto TelError
End Sub
'  END TEST
```

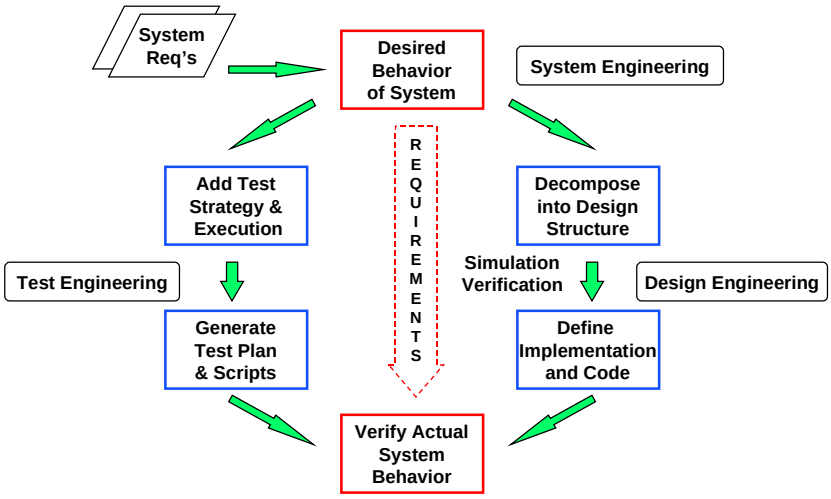

©Teradyne Inc. All rights reserved, 1999

Requirements vs. Test Matrix Report

		Requirement Numbers						
Total Test per Req.		2	3	2	3	3	0	1
Total Req per Test	Test Number	1.0	1.0.1	1.0.2	1.1	1.1.1	1.2	1.2.1
3	1		X			X		X
1	2	X						
2	3		X		X			
1	4				X			
0	5							
2	6		X		X			
3	7	X		X		X		
0	8							
1	9					X		
1	10			X				

©Teradyne Inc. All rights reserved, 1999

A Behavioral Model Facilitates an Integrated Design and Test Process



©Teradyne Inc. All rights reserved, 1999

Benefits of Model Based Requirements

- Provides early detection of errors in requirements
- Significantly improves test coverage of requirements
- Provides an easy method of documenting covered requirements by tests
- Manual testing can be error prone and not reproducible
- Given a change to the requirements, it is easier to adapt model and then automatically re-generate the test scripts than to manually change all the test scripts.
- Facilitates test suite management as the model can be base-lined for a particular release

©Teradyne Inc. All rights reserved, 1999

AT&T and Teradyne Software & System Test

Use-Cases Are Not Requirements

Date: March 19, 1999

From: Steve Meyer
AT&T
E-mail sameyer@att.com

Larry Apfelbaum
Teradyne Software & System Test
E-mail larry@sst.teradyne.com

ABSTRACT: Use Cases are Not Requirements
Much work has been done in recent years to improve the process used to develop software. Increasingly, use of object oriented methodologies have become standard. One aspect common to many of them is the increased emphasis on Requirements Modeling. Capturing requirements accurately is essential to developing correct software. A currently popular Object Oriented method for requirements capture has been use cases, use scenarios or Use Case Requirements Modeling. A use case is a mechanism where an engineer can describe a specific scenario for the system, illustrating one or more key characteristics of its business functionality and processes. Use scenarios provide a valuable means for a team to review the proposed business solutions under a limited number of *specific conditions*. It is a communications tool valuable in the specification, analysis, development and testing of a system. Although a Use case approach is very effective at capturing business functionality, they are somewhat lacking in capturing usage and behavior characteristics of the system.

Outline

1. Introduction **2**
2. What are use cases? **2**
3. What are requirements **3**
4. What use cases lack **6**
5. Fitting behavioral modeling into the development process **7**
6. Use of models **7**
7. Conclusions **11**

1. Introduction

Most systems can have very large numbers of potential usage scenarios. It is not practical for designers or system engineers to explicitly describe all usage scenarios for all Use Cases. These scenarios often only implicitly define the actual *behavioral* requirements of the system. To completely define the requirements illustrated by a usage scenario one must explicitly enumerate each possible sequence of actions. As the number of requirements becomes large, manual specification of all possible usage scenarios becomes increasingly difficult and the possibility of missing, incomplete or ambiguous behavior increases.

At a minimum Uses Case Requirements Modeling provides use scenarios which describe the business functionality to be captured as a set of enumerated steps. This may include some pre/post conditions and some exception processing alternatives. Ivar Jacobson's Requirements Model consists of: Use Case Model, Domain Object Model and Interface Descriptions. A thorough approach to Use Case Requirements Modeling can include:

- Uses Cases described via a well defined template
- Use Case diagrams to illustrate high-level use case relationships
- Domain Object Model depicting objects in the business domain and their attributes
- Interface diagrams, Business Rules, User Interface Descriptions

At best, Use cases fail to provide information that could greatly enhance subsequent steps in the development process. An approach of using behavioral modeling is an opportunity to further "nail-down" requirements in a way that can provide continuity from System Engineering through test.

This paper presents a methodology where a behavioral model describing the systems actions can augment use cases as a compact means of describing them. In addition a process of overlaying the systems requirements onto this model will provide a very efficient mechanism for determining when any arbitrary use scenario generated by the behavioral model has covered any of the specified requirements. Benefits of this approach include easy analysis of behavior, rapid response to changes in specifications of requirements; tests generation correlated to requirements as well as automated generation of requirements based tests.

2. What Are Use Cases?

Use Case Requirements Modeling is one of the first steps in an Object Oriented approach to systems development. Independent of the subsequent analysis and design methodology used, Use Cases provide a superior method for communicating the business functionality to be developed. Traditional thinking maintains requirements describe the "what" is required, whereas subsequent development steps translate from the "what" to the "how". In spite of improvements in requirement specifications this thinking has led to a gap between requirements and behavior. Use Cases describe requirements for a developer’s translation; from a function but not usage point of view. They lack the ability to portray the business needs from a behavioral or user perspective. Behavioral modeling techniques and tools are now available and sufficiently mature that can bridge this gap between function and behavior. One such behavioral approach models requirements in terms of states and transitions between states. Use Case requirements can now be extended with behavioral modeling to add context and organization. A sort of middleware between the **What** and the **How**.

2.1 Use Case Descriptions

A Use Case is a specific way an actor [a person, or process] uses the solution/system by performing some part of the functionality. A use case specifies the interaction between the actor(s) and the solution/system and describes the functionality to be performed. Typically these descriptions consist of an enumerated set of steps that are needed to be performed for the functionality to reach a successful conclusion. Pre/post conditions and limited exception conditions are often included. Frequency and arrival rate information is almost never included.

2.2 Use Case Diagrams

Use case diagrams are often used to capture relationships between use cases. Certain modeling conventions can be used to further define use case relationships. A well-structured diagram with well-focused use cases is valuable for adding clarity to large models. Following are a couple of trivial examples:

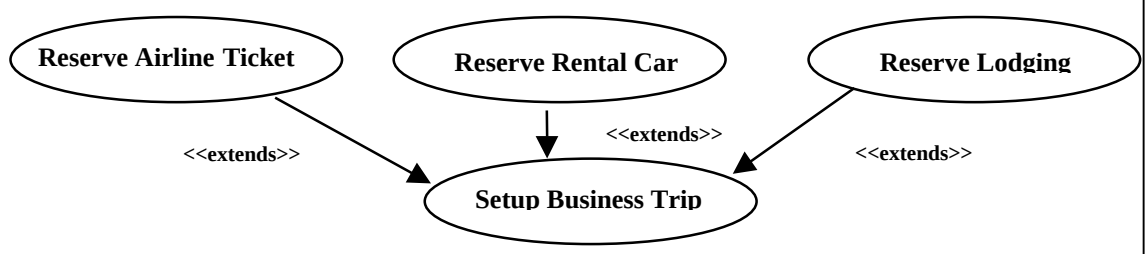


Figure 1 Use Case Diagram for Business Trip

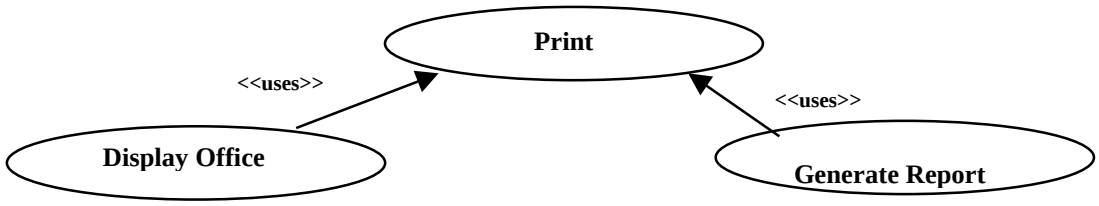


Figure 2 Use Case Diagram for Print System

2.3 Additional Supporting Documentation

Other documentation commonly employed to support Use Cases include: the domain Object model, interaction diagrams, business rules and user interface specifications. Each adds a level of detail to support development efforts by providing information relevant to the problem domain.

3. What are Requirements

I think it would be safe to say that, "The customer wants a solution (system) that meets functional requirements." There may often be constraints in terms of response time or other concerns but it is the requirements that describe the intent. Webster's defines a requirement as "something required: something wanted or needed". Within the engineering discipline the definition [IEEE 729, Dorfman & Thayer[1]] has focused more on the reasons why we have them: "1) a software

capability needed by a user to solve a problem or achieve an objective; 2) a condition or capability that must be met or possessed by a system or a component in order to satisfy a contract, specification standard or other formally imposed documentation." The objective of the requirements phase in a development project is to be able to communicate the needs of the users of a system to the entire team. To accomplish this we need to first verify that the requirements are correct then as the various part of the development team work on the project we need a means to correlate the results back to the requirements themselves.

A key aspect of a software requirement specification is to describe what the software is to do without describing how it is to do it. Davis[2] has defined it as a two phased process, problem analysis and product description. In problem analysis the team will interview people who understand the problem and it's constraints until they have a thorough understanding of the problem. Next in the product description phase the team uses this understanding to define the external behavior of a product that will solve the problem. The solution may not be complete but the issues around the tradeoffs made should all be defined and understood. In addition to describing what is desired it is often just as important to define what is not. Defining the behavior a system should not exhibit is also important in communicating clearly the problem to be addressed.

Many engineers would rather define how they think something should be built rather than focus on what the customer cares about. The answer to the how question is best covered in separate phases known as a functional and design specifications.

3.1 Why are Requirements Important

It is very important to define the requirements accurately, and unambiguously. There have been many studies documenting the cost of repairing a defect, all define a clear escalation of the cost as the development process progresses. An example of this is shown in Table 1, it is from Gause & Weinberg[3] and references data from a Boehm[4] study of 63 software projects from leading corporations (IBM, GTE, TRW).

Phase in Which Found	Cost Ratio
Requirements	1
Design	3-6
Coding	10
Development Testing	15-40
Acceptance Testing	30-70
Operation	40-1000

Table 1 Relative Cost to Fix an Error

It should be obvious to anyone that the business impact of detecting and repairing defects at the requirement stage is immense. The escalation in cost is due to two primary factors, 1) the delay from when the defect was introduced until it was detected and, 2) the cost of the rework involved to repair the defect. The delay is introduced because the detection process, also known as testing, verification or review, is based on determining a difference from the article under test and an oracle, a source of truth. The problem feeds on itself if the basis for the validation is also flawed. If the information used to drive a stage of the process is flawed then even if the process works perfectly, the output will also be flawed. Given that each stage of the development process is not perfect what we have is a larger and larger proportion of defects. Table 2[5] describes the compounding affects of defects.

Requirements	Functional	Design	Code
Correctly defined requirements	Correctly defined functional description	Correctly designed specifications	Correctly coded software
Incorrectly defined requirements: missing, misunderstood, ignored, outdated, unneeded, unspoken (assumed)	Erroneous functional specs based on incorrectly defined requirements	Design errors based on incorrectly defined requirements	Coding errors based on incorrectly defined requirements
	Incorrectly defined functional specs	Design errors based on incorrectly defined functional specs	Coding errors based on incorrectly defined functional specs
		Design errors	Coding errors based on design errors
			Coding errors

Table 2. Compounding Effect of Defects, even good work on a faulty base will result in a defective product

This is analogous to the problems encountered in determining the yield of a multi-stage manufacturing process. Using a formula of:

$$\text{Quality}_{\text{of this stage}} = \text{Quality}_{\text{of previous stage}} * \%\ \text{DefectLevel}_{\text{of this stage}}$$

a simple analysis of the problem as shown in Table 3, results in some scary conclusions. Even scarier than the mathematical analysis is the consistency of the conclusion to many real life situations.

Scenario	Stage->	Requirements	Functional	Design	Code
All Stages at 90% Correct	% Work Done Correctly	90%	90%	90%	90%
	Cumulative Defect %	90%	81%	73%	66%
All Stages at 85% Correct	% Work Done Correctly	85%	85%	85%	85%
	Cumulative Defect %	85%	72%	61%	52%

Table 3. Cumulative Effects on a Multi-stage Process, at an 85% quality level almost half of the code delivered will be defective

3.2 Characteristics of Good Requirements

The primary goal of good requirements is an effective communications mechanism. Customers will have their expectations met and the entire development team can discuss all aspects of the product in terms that relate directly to the customer goals. In particular the work done in the system-testing phase can be directly tied to the needs of the customer. An additional long-term value of the requirements spec is the ability to quickly understand the affects of changes made to the system as it evolves.

This paper is focused on some of the issues related to creating a robust description of a system's behavior. For most modern systems there is an infinite number of potential scenarios that can be defined for a system. Clearly an engineer/analyst cannot define this many scenarios, a common substitute is a long text description of the system combined with some use cases. These tomes are often too large for effectively communicating the desired behavior of a system and lack an effective means of helping the engineers analyze the specification. The use cases do aid the engineer by providing a clear description of a use scenario. A problem with this approach is the lack of a standard mechanism to generalize the example used in the use scenario to the robust model needed to communicate the system to engineers.

4. What Use Cases Lack

Some things Use Cases lack in describing the requirements are described below. An example of a good use case diagram and the problems described is shown in Figure 3.

1. ***Sequence and flow*** of operations. Although Use Case diagrams can illustrate some of the relationships between individual Use Cases, they do not convey sequence and flow of related operational usage. In a large model detailing to this level of behavior becomes increasingly difficult. Besides requirements traceability this is one of the biggest benefits of behavioral modeling. It gives the engineer an opportunity to describe requirements from a usage standpoint.

2. ***Frequency and arrival rate*** information of individual Use Cases. Such information can be used for performance engineering of a system and help define load test scenarios. This type of information is easily portrayed in a state transition model, not so easily captured with Use Cases. Once sequence and flow is effectively modeled annotating frequency information can enhance the graphical representation. This information can be valuable for performance and load testing

3. Use cases often describe ***only best case*** (successful completion of operation) and limited exception information. Visualizing and modeling many exception conditions can be a large task. This is more easily undertaken using state machine techniques.

4. ***How the system(s) are used***, or could be used. An improved process can be realized if the actual requirements are separated from the discrete descriptions of individual use cases. Following paths through a behavioral model has proven to be a very effective approach to understanding how requirements might be used with in a "System" framework. This modeling gives the engineers a chance to further nail-down requirements and how they behave.

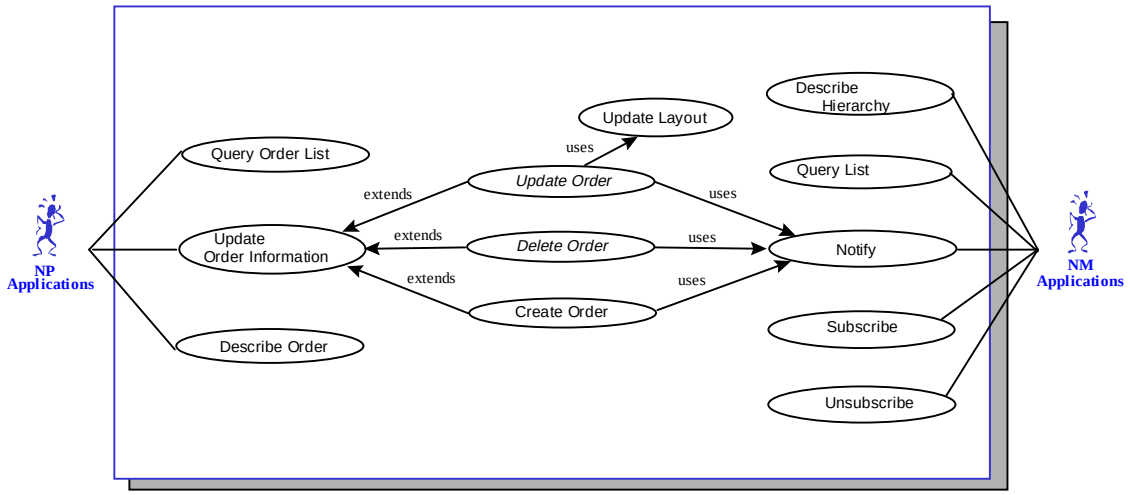


Figure 3 Simplified - High Level Use Case Model **[Ehrlich 1998[6]].** *Where is -- 1. Sequence and Flow, 2. Frequency and Arrival Rate, 3. Only Best Case, 4. How the system(s) are used?*

It is not always evident how various Use Cases interact within a developed solution. Systems engineers can use models to further refine the relationships between Use Cases at a ***high*** level.

This is useful in confirming the Use Case framework used to describe the feature functionality of the solution. [Meyer May 1998[7]]

5. Fitting Behavioral Modeling into the Development Process

This discussion will focus on the "State machine models" box as the piece inserted into the development process (see Figure 4). The premise here is modeling work is appropriate for both system engineers and test engineers. There are also implications the system engineers models could be useful for the analysis, design and implementation teams. As the application progresses to integration and system test, executable test scripts will have been produced to exercise the applications behavior.[Meyer Jan 1999[8]]

5.1 System Engineers Model

The focus of the system engineers modeling work is to further nail-down requirements by describing them with state/transition methodology. This description should provide organization and sequence information and requirements traceability.

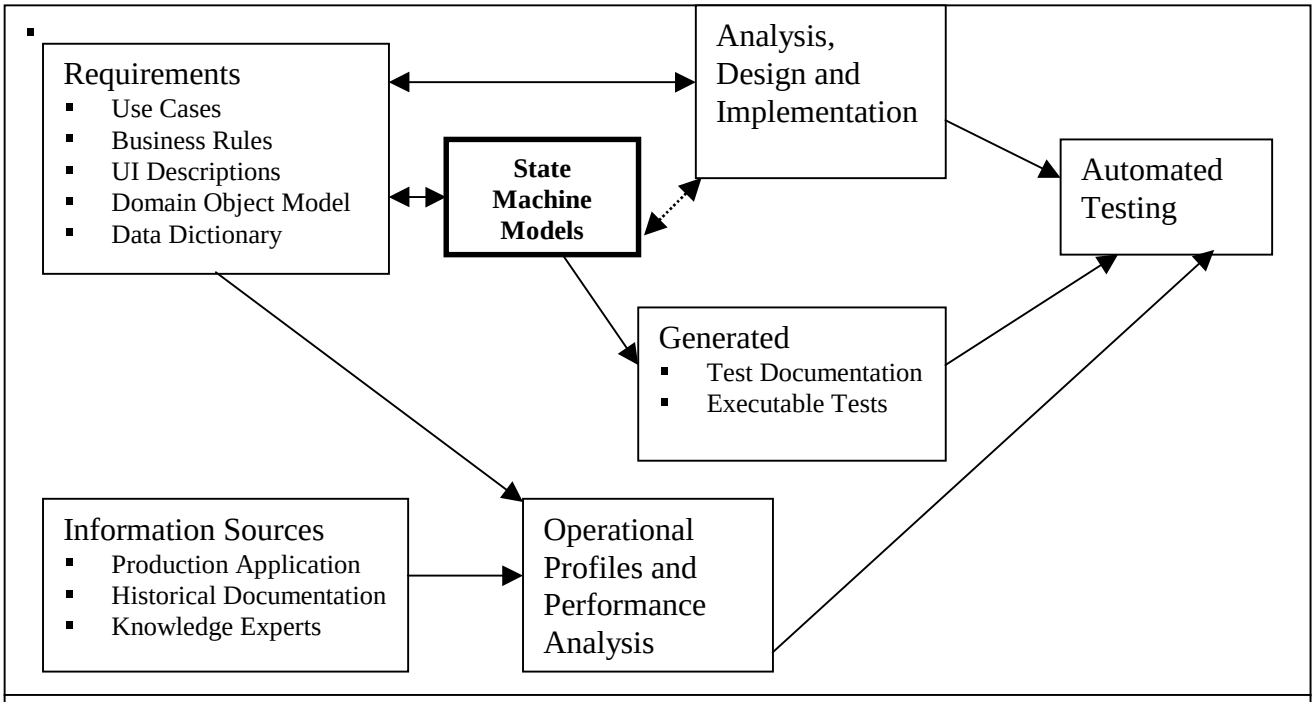


Figure 4 System Development Process, a step has been added to develop state models

5.2 Test Engineers Model

Test engineers can take the system engineers state-machine behavioral models and extend them. Calculating paths through these extended models exposes a great amount of complex behavior that needs testing. Taken a step further these paths become tests.

6. Use of Models

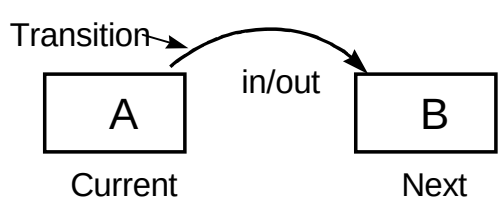
The use of models to define behavior is not new, it has been used in both development and test processes.[9,10] The recent popularity of Object Oriented Analysis and Design techniques have increased the deployment of model based techniques across the software development community.

We have developed a technique where a behavioral model of a system can be used to replace the sets of use cases used to describe a system's requirements. The model provides a means of generating use scenarios when needed. Each path through the model is equivalent to a unique use scenario. By exploring different paths, including potential loops, error conditions etc. a more robust understanding of the system can be developed. Furthermore, as the system is changed the graphical view of the model makes communicating the impact of a change much easier.

6.1 Building a Model

Modeling is not new, engineers always build and use models. The 'model' may only exist for a short time and live on a napkin or remain in the mind of the engineer; it is not always preserved in a reusable form. This model of behavior is analyzed when one creates requirements, writes a use case, designs code, or develops a test plan; an engineer must understand the basic operations and actions of the system. The means used to implement the behavior is not required until design and/or implementation begins. Modeling at the behavioral level is straightforward and can contain information from specifications, use case diagrams, sequence diagrams and flowcharts.

This approach is based on the concept of a state machine (see Figure 5) where the transactions, represented by arrows in the diagram, correspond to the actions (transitions in the state machine), while icons in a variety of shapes represent the states. State machines are an established modeling approach and have been extended[11] to provide a more powerful means of describing complex systems. Actions that can occur during the use of the system are defined by adding arrows and or states to the basic diagram. The actions that "could" occur also imply that that there may be more than one possible action at a specific point in the process. Most modeling techniques support the idea that there are multiple potential "next" actions. Some of these actions can only occur if certain conditions exist in the system, this information can be added to the basic model by associating a rule or condition with each action. In our approach these are included as predicate expressions on the transitions. An effective modeling technique allows an engineer [analyst or other specifier] to unambiguously define these alternative sequences and any rules associated with them. Another modeling technique we utilize is hierarchical models, where a state can be replaced by a 'call' to another model that defines the behavior within the state, this approach is also referred to as 'super-states'. Hierarchical models allow complex behavior to be decomposed into simpler lower level models.



- Transitions define actions that move the system from the current state to a new state
- Transitions can be based on specific stimuli and/or previous actions
- Text descriptions of user actions can also be associated with transitions
- Use cases are built by concatenating the descriptions from a sequence of transitions through a diagram

Figure 5. A Finite State Machine is Composed of States and Transitions

Developing a specification in the form of a model is a very effective means of:

1) discovering defects in the system (many are made visible by the modeling effort alone),
2) rapidly defining the basis for use scenarios of the system, and
3) preserving this investment for future releases or other similar systems.

Furthermore, the process of developing a model is best done in a measured series of small incremental steps. This incremental building approach allows the core functionality to be defined, evaluated and understood before all of the secondary features are added. This also allows a larger team to be applied to the problem. They can divide up the more detailed specifications (typically in lower level models), and integrate the work into the larger model.

Once a model has been developed, even if it is a partial model, use scenarios can be developed by finding paths. A path is a sequence of events or actions that traverse through the model defining an actual use scenario of the system. Each element in a path, a transition or state, can have some text associated with it. Concatenating all of the text found on the path elements will provide us with a textual description of the entire use scenario. This process can be repeated for another path, which defines another use scenario, and validates another sequence of actions. Many methods can be used to select paths, each with its own distinct objectives and advantages. Operational profiles, reliability/criticality data, and coverage all provide different tradeoffs to the type of scenarios and the resulting coverage. A primary advantage of this type of structured definition of the behavior is that automated tools can be used to help in the analysis. This approach also allows an engineer or analyst to see potential secondary paths that may exist. The benefits of having this complete view of all of the potential behaviors at any point in the flow will allow a more complete analysis as well as make updates and changes easier to insert.

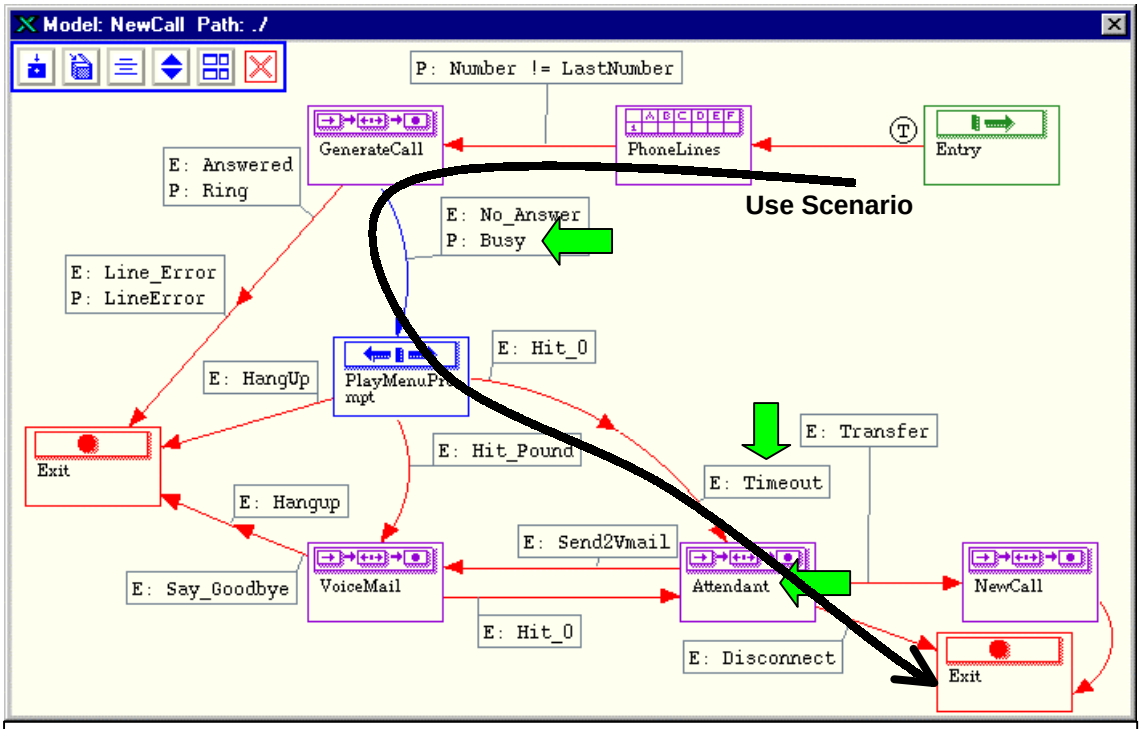


Figure 6 Voicemail Model Example. A model can show a use case in the context of the overall system functionality. This example illustrates a use scenario where after the greeting is played on a busy line a caller will be transferred to the attendant after a timeout period.

6.2 Overlaying Requirements on the Model

At this point we can define a behavioral requirement in terms of model objects. If a system must provide a specific behavior as one of its requirements it can be expressed in terms of the objects in the model. For example in Figure 6 we define a voicemail system. If a requirement specification included "Requirement *timeout_transfer* : On a busy line, the system shall transfer a caller to the attendant after the specified timeout period" we could define this by linking the requirements to the elements of the model [block arrows in Figure 6] with an expression like:

"*timeout_transfer* = **Busy** && (**Timeout** -> **Attendant**)"

The -> operator is defined as "is immediately followed in a sequence by", the operands are the objects in the model diagram [**Busy** and **Timeout** are events, **Attendant** is a state (actually a superstate)]. This expression now provides us with a versatile mechanism for understanding when any path is verifying the requirement. Any path where *timeout_transfer* evaluates to true is one that verifies the requirement. This model and expression based approach now allows other portions of the model to change without affecting this requirement, and as new paths are generated they can all be evaluated to see which of them verify a requirement.

The graphical model can also be processed by an automated test generation tool to produce not only test plans and scripts but a detailed document describing how each requirement was covered in the resulting set of tests. As the tests are created and later executed a team can track the project's completeness and the product's consistency with the requirements. A sample of a report from an automated test process with the requirements tracking integrated into it is shown in Figure 7.

				Requirement Numbers				
Total Test per Req.		2	3	2	3	3	0	1
Total Req per Test	Test Number	1.0	1.0.1	1.0.2	1.1	1.1.1	1.2	1.2.1
3	1		X			X		X
1	2	X						
2	3		X		X			
1	4				X			
0	5							
2	6		X		X			
3	7	X		X		X		
0	8							
1	9					X		
1	10			X				

Figure 7 Tests vs. Requirements Report – This allows the engineer to determine which tests are redundant as well as which requirements are not yet covered. In addition, tests that are not adding new requirements can be dropped to optimize test execution time.

Another benefit occurs when changes are introduced, if a new feature is added, an incremental edit is made to the model and then paths can be regenerated to confirm that the old requirements

are still covered. If the changes originate with the requirements, they can be added as new requirement expressions and then the model can be analyzed to see if the requirement is already covered by the existing functionality or new behaviors must be added.
A second type of analysis, similar to the requirement can also be performed on a model. If there are requirements that the system not allow certain behaviors to occur, they too can be expressed as expressions and verified. For example, in a system there may be relationships that should not be violated, sometimes referred to as invariants, these can also be verified using an expression. The expression, similar to the ones used for specifying requirements, is evaluated during the path generation process; if at any time it becomes false, we have uncovered a design flaw in the system.

7. Conclusions

An approach where a system's behavioral requirements are used to incrementally develop a behavioral model can provide a team with significant advantages. It can be used to thoroughly describe application behavior from specifications or models and automate the creation of use cases as well as test programs. A path generation tool can be used to extract ***directly executable*** tests.[12]

Benefits include:

- Provides early detection of incomplete or inconsistent requirements.
- Significantly improves test coverage of requirements.
- Provides an easy method of documenting covered requirements by tests
- Manual testing can be error prone and not always reproducible
- Given a change to the requirements, it is easier to adapt model and then automatically re-generate the test scripts than to manually change all the test scripts.
- Facilitates test suite management as the model can be base-lined for a particular release [Meyer Jan 99]

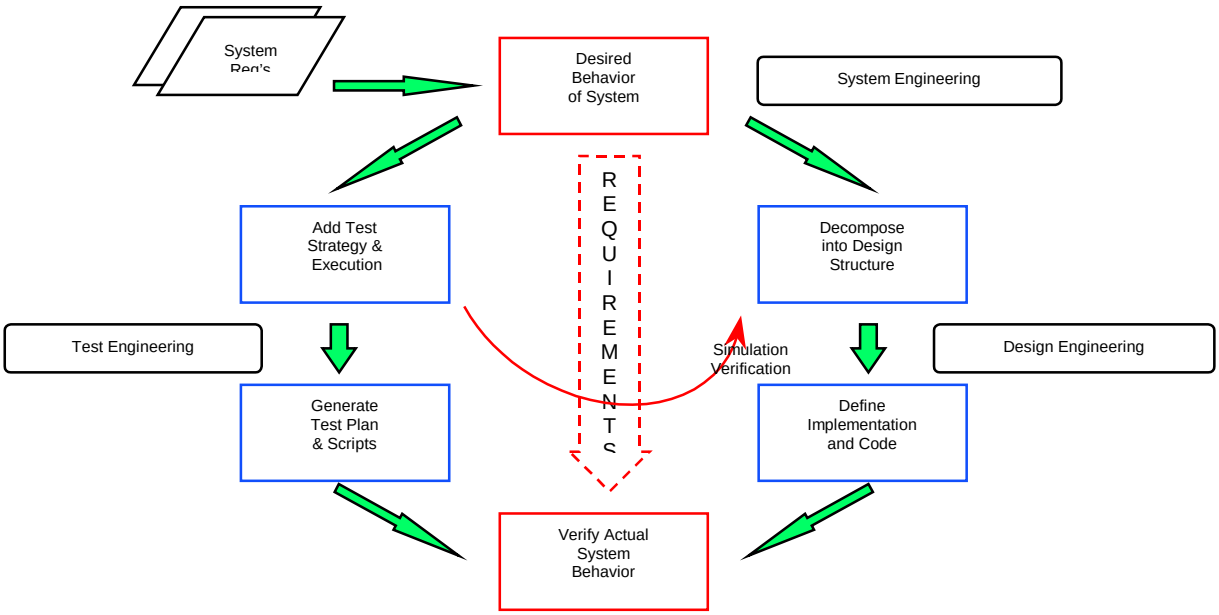


Figure 8 The System Behavior is the reference for all later stages. Requirements defined based on the system level model will carry forward into the test.

Furthermore if the requirements of the system are integrated as expressions, the system can also automatically track the progress and completeness of the team. This system level behavioral model can serve as a reference for the systems engineering as well as the development and test phases of product development, see Figure 8.

Benefits of this approach include easy analysis of behavior, rapid response to changes in specifications of requirements, tests generation correlated to requirements as well as automated generation of requirements based tests. This system provides an organization with an efficient, reusable mechanism to both maintain a suite of tests and respond to feature or requirements changes during the product development cycle.

References

[1] Dorfman M, and Thayer R. Standards, Guidelines and Examples of System and Software Requirements Engineering. Washington D.C. IEEE 1990

[2] A. Davis, Software Requirements Objects Functions and States, New Jersey, Prentice Hall, 1993

[3] D. Gause & G. Weinberg, Exploring Requirements – Quality Before Design, New York, Dorset House Publishing, 1989

[4] B. Boehm, Software Engineering Economics, Prentice Hall, New Jersey, 1981

5 J. Taft, "Implementing the 'V-Model' Quality Framework, Software Testing Analysis & Review West, 1997

6 W. Ehrlich, "Facility Component System", Use Case Diagram, 1998

[7] S. Meyer, " Foundation Architecture Evaluation Results - Release 7.0 , Test Case Generation Tools ", AT&T Internal Memorandum, 1997

[8] S. Meyer, "TestMaster Pilot Update", AT&T Internal Memorandum, 1999

[9] Beizer, B., *Black Box Testing*, New York, John Wiley & Sons, 1995. ISBN 0-471-12094-4

[10] L. Apfelbaum. Doyle, J., "Model Based Testing", Proceedings of the Software Quality Week 1997 Conference, 1997.

[11] D. Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming 8, 1987

[12] J. Clarke, "Automated Test Generation from a Behavioral Model", Proceedings of the Software Quality Week 1998 Conference, 1998

12th International Software Quality Week

Testing for Y2K Readiness: A Case Study

Jim Williams, CQA
CableData, Inc.
jim_williams@uscs.com

May 27th, 1999

Y2K Readiness Disclosure Act :S 2392

- This document is a Year 2000 Readiness Disclosure as defined under the Year 2000 Information and Readiness Disclosure Act of 1998. All obligations of CableData with respect to its systems and services are described solely in written agreements between CableData and its customers. This document does not constitute any express or implied representation or warranty by CableData or any amendment, interpretation or other modification of any agreement between CableData and any party.

Introduction

- International Company
  - Produce Software for Ordering, Tracking, Supporting, Billing, and otherwise processing Cable TV & Telephony.
- For these kinds of Applications, the FCC has mandated that Y2K readiness is critical - interruptions of services would have a very high impact.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 3

Introduction…

- Applications Under Test (AUT)
  - Two Products on separate platforms.
    - » 150 applications on a UNIX Server
    - » 110 applications on a Tandem Computer
- Both sets of Applications are accessible from PC's -- Work-stations that connect via on-board Terminal Emulator.
- Each product has its own SQL Database.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 4

Introduction….

- The total number of lines of code is in excess of several million.
- Almost all of the applications produce results that are seen as responses on the PC's
- Automated Testing is used for every Release during the Regression Testing Phase.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 5

The Y2K Challenge

- An Early and Absolute Deadline to meet...
  - Our Software allows Customers to do Annual Billing.
  - We needed to be Y2K Ready by mid 1998, so our software could be implemented throughout our Customer base by year's end.
  - Enhance current Automated Scripts to cover Y2K Releases, with an acceptable level of Path Coverage.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 6

Data-Aging Method Used

- Critical testing dates were determined
  - 09/09/1999 - Four nines (sometimes 9/9/99)
  - 12/31/1999 - The Day in Question
  - 01/01/2000 - The first Day after
  - 02/29/2000 - Yes, a Leap Year!
  - 03/01/2000 - The Day after the Leap Year
  - 12/31/2000 - 366th Day of the Year
  - 01/01/2001 - The first Day of a Normal Year

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 7

Data-Aging Method Used...

- We tested two additional dates - just to see what would happen with our AUT…
  - 01/01/2037 - UNIX OS Dates OK.
  - 01/19/2038.03:14:07 to be exact... UNIX OS Dates are not OK.
- This test is for the time when the 64-bit UNIX long-integer date, the number of seconds since 01/01/1970, "turns over".

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 8

Y2K Test Methodology...

- Aging the Data
  - Because of the nature of our data, we have dates stored in the past, present and future.
  - We needed a tool that could age all of the dates the same number of units, therefore keeping the database in sync.
  - Platform specific tools were developed that allowed us to age our data to the desired date under test.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 9

Y2K Test Methodology...

- Our Tests scripts were developed to test a range of Dates for the Date in question.
  - For most features of the AUT, we tested current date, one day, one week, one month and six months in the past and same units of time in the future.
  - This method systematically allowed us to test both past and future dates from our base date.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 10

Y2K Test Methodology...

- Our AUT uses Dates in two ways..
  - Dates used in Calculations - Ages, Amounts, Periods of Time, Expiration, Interest, Taxes and Prorates.
  - Dates used Operationally - Sorting, Searching, Qualifying, Ranges, Reporting and Time stamps for License Validation.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 11

The Test Scripts

- The test scripts are many in number and very complex.
  - UNIX used over 22,000 test scripts.
    - » 280,000 screen captures.
  - Tandem used over 18,000 test scripts
    - » 124,000 screen captures
- The scripts varied in length from 50 lines, up to 12,000 lines for more complex tests.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 12

The Test Scripts...

- Most of the test scripts were "constructed" from pieces that were engineered to fit together, using the tree-oriented method of test script generation.
- These scripts were verified and validated by using a "Path Coverage" tool, obtaining an average Path Coverage of 80% for the AUT.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 13

The Test Scripts...

- In some cases "masking" was used to allow the differing engine, to skip over unimportant differences and not report them as a difference.
- Even though this was a lot of work, it saved hundreds of hours of unnecessary work as we repeated the same tests for each Y2K date.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 14

Running the Tests

- The Tests were run from a cluster of twelve PC's, with an average of eight being used 12 hours a day 6 days a week.
- UNIX suite of Tests took about 7 Days to run to completion.
- Tandem suite of Tests took about 10 Days to execute.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 15

Running the Tests…

- Each Test suite was run a minimum of 14 times per platform.
  - Once for each Date before remediation and at least once after the AUT had gone through remediation.
- Remember, however, there is significant rework time when a test uncovers one or more differences.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 16

Running the Tests…

- Our estimate of re-work was an additional 20% overall.
- We averaged seven resources on the Y2K Automated Test Team for approximately two years.
  - One year to enhance and debug Test scripts.
  - One year to run and re-run the Test scripts.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 17

Test Results

- We checked out our Test results as a suite of Tests would finish.
- Results were verified at three levels…
  - PC Screen Captures
  - SQL Table Level
  - AUT Performance level
- All three were compared against a known baseline for that product and release.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 18

Test Results...

- The results of our Y2K remediation efforts were very, very good.
- We had no major problems and only a few minor problems were detected that needed additional remediation.
- Some collateral defects were found, in some cases related to Dates and Times, but not critical for Y2K reasons.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 19

Test Results...

- The vast majority of Y2K changes were confirmed without difficulty.
- All Y2K defects were very obvious and easy to detect.
- Overall, we found close to 100 defects that were Y2K related.
- All of our Data, Tests Scripts and Test Results are Documented and Archived.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 20

The Benefits of this Effort

- Allowed QC more time for testing per Release and Product -- seven passes for Y2K versus two or three during a normal Test Cycle.
- Improved our Level of Test Automation.
- Because of the Remediation Efforts...
  - Number of lines of source code went down.
  - Level of Cyclomatic Complexity decreased.

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 21

In Conclusion

- We have a very high confidence that our systems will sail through the Y2K transition without difficulty, making it very easy for our Customers to start the next Millennium with the utmost confidence in our Software.
- Without Test Automation, we would still be testing our first passes for Y2K...

The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQW 99
Jim Williams # 22

Testing for Y2K Readiness: A Case Study

by

Jim Williams, CQA
CableData, Inc
March 24, 1999

ABSTRACT: Automated testing methods are used to confirm Y2K Readiness for a complex, multi-module, multi-platform application under test (AUT). The 2-year effort resulted in some 40,000 test scripts and slightly over 400,000 saved comparison images. Excellent payback from test mechanization and highly enhanced confidence in the AUT are the results.

***Note*: This document is a Year 2000 Readiness Disclosure as defined under the Year 2000 Information and Readiness Disclosure Act of 1998. All obligations of CableData with respect to its systems and services are described solely in written agreements between CableData and its customers. This document does not constitute any express or implied representation or warranty by CableData or any amendment, interpretation or other modification of any agreement between CableData and any party**

INTRODUCTION

The application under test (AUT) is operated by CableData, Inc., an international corporation and primarily includes software for ordering, tracking, billing, and otherwise processing cable TV and telephony technical and financial information. The systems under test include a wide range of applications, from customer billing and associated business management to controllers and line-feed electronics, plus some cell-phone support systems. Obviously, for these kinds of applications the FCC has mandated that Y2K readiness is critical -- interruptions of service and other potential problems have very high economic impact to CableData's customers' business.

There are about 150 applications that run under UNIX on IBM RS/6000 machines, and about 110 applications that run on a Tandem computer. The total number of lines of code is in excess of several million -- i.e., something over 2500 KLOC. Both sets of applications are accessible from PCs -- work-stations that connect to the RS/6000 or Tandem computer clusters via on-board terminal emulators. Almost all of the applications produce results that are seen as responses on the PCs.

Y2K TEST METHODOLOGY

The main method used for Y2K readiness testing is the conventional date-aging method: the application is tried with time-intervals that vary in length over a series of dates before, near, just after, and much later than 31 December 1999.

To get results in this way, it is necessary to "age the database". We developed special software just for this purpose that helps maintain a coherent database of information, aged to the desired date that the Y2K tests are actually run against.

NOTE: The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

SQL relational databases are used in both products, so a special tool was used to verify and validate all SQL activity to a known baseline after each suite of tests had been run.

Some of the software that is used in the system has to be Y2K readiness checked first. This is done to make sure that any defects or anomalies that are found in the specific AUT tests are caused by fixable errors, rather than by errors in vendor supplied code such as databases, operating systems, system utilities, etc.

The date changes that were used include the following nine dates:

09/09/1999	Four nines (sometimes 9/9/99)!
12/31/1999	THE date in question!
01/01/2000	The day after THE date in question.
02/29/2000	Yes, a leap year!
03/01/2000	The day after leap year.
12/31/2000	The 366th day of the year.
01/01/2001	The first day of next year.

We tested two additional dates – just to see what would happen:

01/01/2037	UNIX Dates OK.
01/19/2038.03:14:07 to be exact…UNIX Dates not OK.	

Note that the UNIX dates are included in the testing to attempt toconclude that the systems are OK, for the time when the 64-bit UNIX long-integer date, the number of seconds since 1 January 1970 "turns over".

Our AUT uses dates in two ways: firstly, dates that are used in calculations such as ages, amount, periods of time, expiration, interest, taxes and prorates; secondly, dates used operationally such as sorting, searching, qualifying ranges, reporting and time stamps for license validation.

Our test scripts were developed to test a range of dates for the date in question. For most features in our AUT, we created a test function that would test current date, one day, one week, one month and six months in the past. The same units of time were tested into the future. This method allowed us to systematically test both past and future dates from our base date.

THE TEST SCRIPTS

The test scripts are many in number and very complex. They were only partially generated manually. Most of them were "constructed" from pieces that were engineered to be put together. This is a kind of

NOTE: The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

tree-oriented test script generation that we were able to use with great success.

All automated testing scripts were validated using a “Path Coverage” tool, obtaining an average Path Coverage of 80%.

For the RS/6000 AIX testing, we used a total of 22,000 test scripts that resulted in a set of 280,000 different "screen shot" images that were used as the basis for comparison. In some cases, we had to program the differencing engine to ignore unimportant differences. Even though this was a lot of work at the outset, it saved hundreds of hours of unnecessary work as we ran the tests.

On the Tandem side we have a total of approximately 18,000 scripts and about 124,000 screen images. As with the AIX testing we used some special processing to make the job of comparison easier.

The scripts varied in length from 50-60 lines (some of the simplest tests) to over 12,000 lines (the most complex tests).

About 40% of the tests -- mostly the smaller ones -- were generated by hand, but the 60% that were mechanically generated probably account for 80%+ of the total volume of tests.

RUNNING THE TESTS

The tests were run from a cluster of 12 PCs. Typically, we would run tests for 12 hours a day, six days a week. Even though we had 12 machines to drive the tests, on average we used as few as four and only infrequently had to use all 12 machines. A good estimate is that we used about eight of the 12 PCs for each set of test runs.

The UNIX side suite of tests took about seven days to run to completion, and the Tandem side tests took just under ten days to execute.

Each test suite has been run a total of fourteen (14) times for each platform. This is based on seven executions of the suite: one each for the above-defined dates before Y2K modifications, and the corresponding seven executions for the AUTs after they had been modified.

Remember, however, that there is significant rework time when a test uncovers one or more Y2K deficiencies. We estimate that this accounts for an additional 20% overall.

We have had as many as 13 people on the test team, but the average has been about seven for the approximately two years it has taken to enhance, debug, run and re-run the tests.

RESULTS AND RECOMMENDATIONS

As a pattern of practice, we check our test logs and reports as a suite of tests finishes. The output that is verified is broken out into three categories: PC Screen Captures or Bit Maps, SQL Table Level differing and AUT Performance Levels. All three were compared to a known baseline for that product and release.

NOTE: The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

Our results have been very, very good so far. We have had no major problems and only a few minor problems detected due to the Y2K remediation efforts.

We did find some collateral defects -- in some cases, related to dates and times – but these were not critical for Y2K reasons. The vast majority of the Y2K changes we made were confirmed without difficulty.

Overall, we found and fixed approximately 100 of these Y2K "discoveries".

All of our databases, test scripts and test results are documented and archived.

There were a number of unexpected benefits from the remediation effort. Our suite of test scripts improved from the knowledge learned. More time was given for QC to test each release and product. Normally, we would have between two and three passes during the regression test cycle, but with the added Y2K passes, we had a minimum of seven passes. Another side benefit from the remediation was that the number of lines of source code went down and the level of cyclomatic complexity decreased, which should make the source code easier to maintain and test.

In conclusion, we have very high confidence that our systems will sail through the Y2K transitions without difficulty, making it very easy for our customers to start the next millennium with the utmost confidence in our software.

I know that without Automated Testing, we would not be done, but would still be testing our first pass for Y2K.

NOTE: The statements contained herein are being designated as year 2000 readiness disclosures under the Year 2000 Information and Readiness Disclosure Act.

Test Architectures for Testing Distributed Systems

Andreas Ulrich, Peter Zimmerer, Gunther Chrobok-Diening
Siemens AG, ZT SE 1, 81739 München, Germany
andreas.ulrich@mchp.siemens.de

12th International Software Quality Week 1999
San Jose, 27 May 1999

Presentation Outline

1. **Introduction**
2. Specification-based **test method**
3. **Challenges** in testing distributed systems
4. A **global tester** for testing distributed systems
5. Two types of **distributed testers**
6. A **tool for testing and monitoring distributed systems**
7. **Conclusions**

Introduction

Assumptions

Distributed system DS = collection of communicating components

$$\Im = M_1 \parallel \ldots \parallel M_k$$

E.g. implemented on a middleware platform Corba, DCOM or Java RMI or proprietary implementations

Testing goal

to test a DS against its functional specification (conformance test)

Focus: **integration testing** of DS; requires a grey-box testing approach

Test object in practice

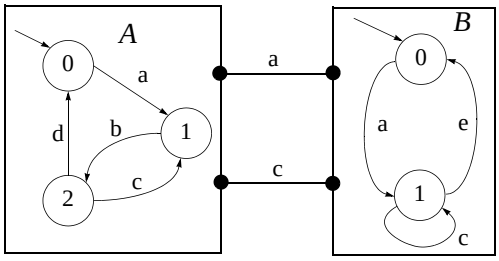
telecommunication switching software developed at Siemens ICN

Specification-based test method

The behavior of the distributed system DS is given in a specification

e.g. (typically) communicating state machines (SDL, UML state charts)

Example: $\Im$ = *A* |[*a*, *c*]| *B*



A possible global action sequence the distributed system may perform:

σ = *a.b.c.e.b.d.a.b.e.d.a.b.d.e*

Note: any interleaving sequence {*e.b.d*, *b.e.d*, *b.d.e*} may occur!

Specification-based test method (cont.)

Test architecture

the concurrent system: *Tester* || *System Under Test* (*SUT*)

In case of **integration testing**:

The tester observes/controls all external and internal interactions of the SUT.

Test run

The tester implements a test case to test the SUT

Test run = a (concurrent) execution path through *Tester* || *SUT*

Test oracle

A test was successful (*pass*) if the SUT performs correctly all execution sequences described in the specification (acceptance test).

Challenges in testing distributed systems

The main challenges are

- ☐ true concurrency between components of the DS
- ☐ other types of non-determinism in the DS
- ☐ absence of a global clock
- ☐ unobservable messages exchanged between components

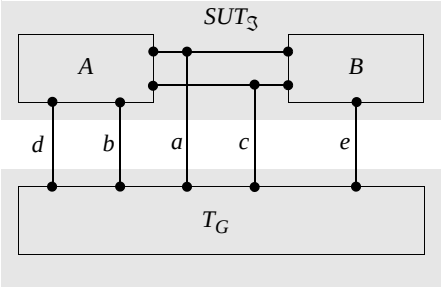
Non-determinism as the main challenge

- ☐ due to interleaving of concurrent actions of the SUT
- ☐ within a component of the SUT
- ☐ due to race conditions within the SUT

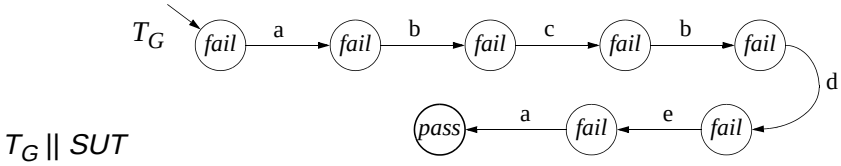
➔ An appropriate test architecture needs to address non-determinism in a DS to obtain a **deterministic test run**!

A global tester for testing distributed systems

Test architecture $T_G \parallel SUT$



The global tester implements an acyclic machine of a global test case, e.g.



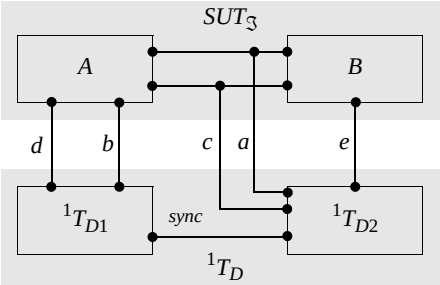
$T_G \parallel SUT$

➔ **Concurrency is not respected!**

Two types of distributed testers

Distributed tester with synchronization events

Tester itself is distributed; test coordination procedure required (TCP)



$({}^1T_{D1} \,|[sync]|\, {}^1T_{D2}) \parallel SUT$

Each tester component ${}^1T_{D1}$, ${}^1T_{D2}$ implements a partial test case derived from the global test case and augmented with synchronization events:

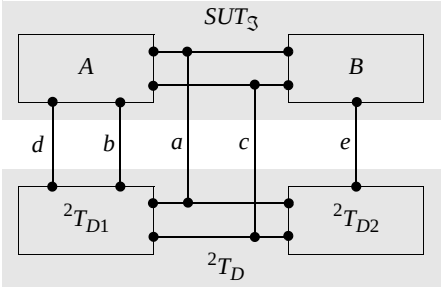
${}^1T_{D2}$ implements ${}^1\sigma_{11}$ = *sync.b.sync.sync.b.d.sync*

${}^1T_{D2}$ implements ${}^1\sigma_{12}$ = *a.sync.sync.c.sync.e.sync.a*

➔ **Concurrency is respected, but complicated TCP needed!**

Two types of distributed testers (cont.)

Distributed tester with redundant observation of internal actions



$(^2T_{D1} \parallel {}^2T_{D2}) \parallel SUT$

➔ Tester represents an inverted image of the architecture of the SUT.

Test cases for tester components:

$^2T_{D2}$ implements $^2\sigma_{11} = a.b.c.b.d.a$

$^2T_{D2}$ implements $^2\sigma_{12} = a.c.e.a$

➔ **Most simplest solution; concurrency is respected!**

A tool for testing and monitoring DS

Test and Monitoring Tool (TMT)

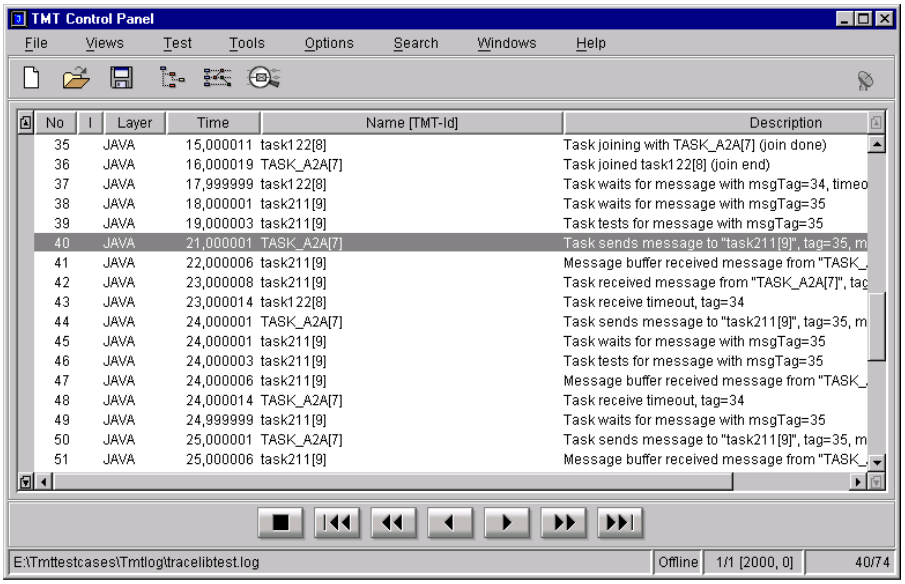
- a first prototype tool to support in-house integration testing
- implemented in Java, JDK 1.2
- more than 210 class files and 60,000 lines of source code
- developed in cooperation with the Information and Communication Networks Division and the R&D Software and Engineering Department

Features of TMT

- resembles a distributed tester with redundant observation of actions
- provides different views on the distributed system under test
- allows user interactions to control a test run
- provides filter and other mechanisms to reduce the amount of data
- supports DS implemented in C++ or Java

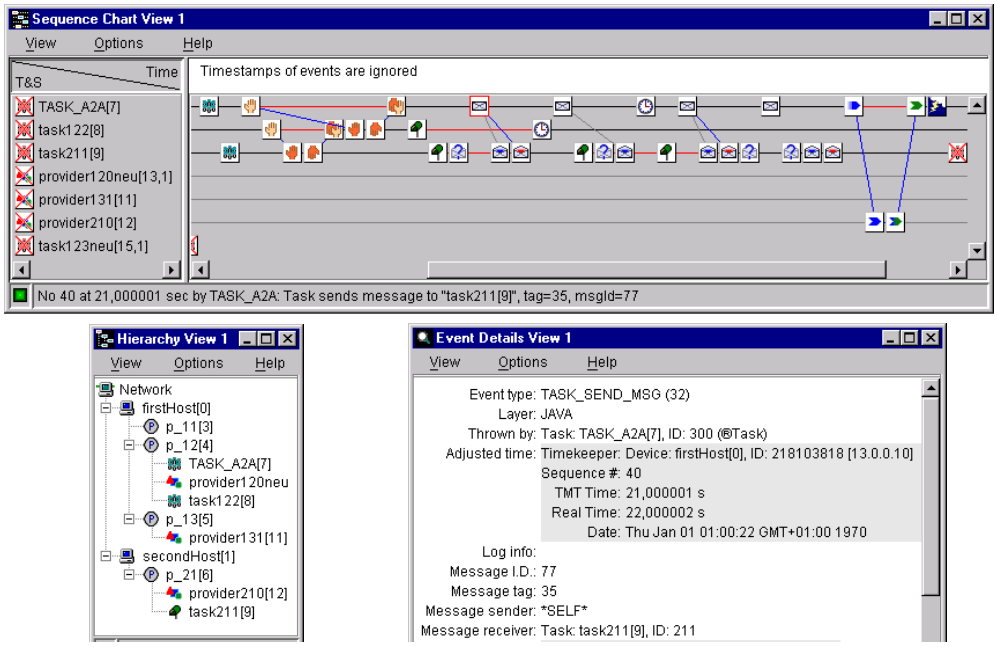
A tool for testing and monitoring DS (cont.)

The main window of TMT



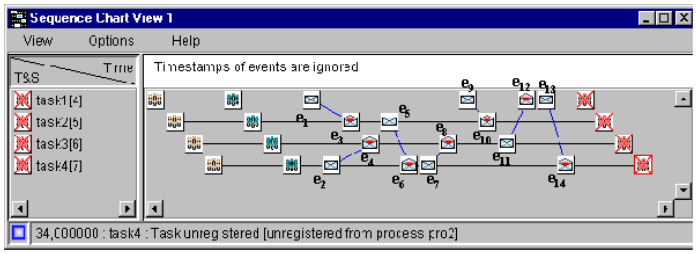
A tool for testing and monitoring DS (cont.)

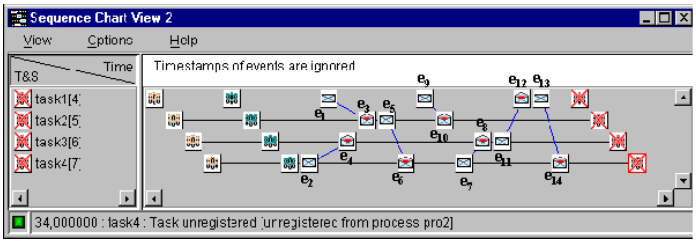
Different views on the SUT provided with TMT



A tool for testing and monitoring DS (cont.)

Assessment of test runs with TMT





➔ Assessment is done in the TMT main component.

➔ It is based on dependencies between test events.

Conclusions

- ☐ Testing distributed systems must be conducted such that a **deterministic test run** is achieved.
- ☐ **Concurrency** between test events must be respected during test execution and for the assessment of the test run.
- ☐ **Race conditions** and **non-determinism** in a SUT component may still exist.

➔ Testing **distributed systems** needs to be assisted by a **distributed tester**.

➔ A distributed tester with redundant observation of internal interactions of the SUT seems to be most appropriate.

Test Architectures for Testing Distributed Systems

Andreas W. Ulrich, Peter Zimmerer, Gunther Chrobok-Diening
Siemens AG, ZT SE 1, 81739 München, Germany
{andreas.ulrich, peter.zimmerer, gunther.chrobok-diening}@mchp.siemens.de

Abstract
This paper suggests two test architectures for testing distributed, concurrent systems: a global tester that has total control over the distributed system under test (SUT) and, more interestingly, a distributed tester comprising several concurrent tester components. The test architectures rely on a grey-box testing approach that allows to observe internal interactions of the SUT by the tester. In order to realize distributed testers, the correct global view on the behavior of the SUT must be maintained by the tester. This can be achieved if coordination procedures are established between the concurrent tester components. Coordination between tester components can be implemented by means of redundant observation of internal interactions within the SUT or by means of using synchronization events between tester components. First concepts of a distributed tester are realized in a prototype of a test and monitoring tool used for in-house testing, monitoring, and analyzing distributed systems within Siemens. The project is conducted in Siemens' R&D Software and Engineering Department in charge of its Information and Communication Networks Division.

Keywords
Test architectures, distributed systems, concurrency, specification-based testing, monitoring.

1 INTRODUCTION

Distributed software systems have gained increasing importance in recent years. Such applications are not just client/server systems, but more and more complex systems consisting of a number of independent components running concurrently on different machines on top of a communication network. Today distributed systems are implemented using middleware technologies, such as Microsoft/COM, OMG/CORBA, or Java RMI. But there exists also a number of proprietary solutions to distributed systems, e.g. implementations of a H.323 video conferencing protocol and others.

Designing, implementing, and validating such complex distributed systems requires a deep understanding of the communication taking place and the order of communication messages

exchanged between the individual components to avoid deadlocks or livelocks during runtime. This applies in particular to the testing of these systems.

We assume in our discussion that a distributed system consists of a collection of components, each of them realizing a sequential flow of execution. Each component has an interface that defines its incoming and outgoing messages. Testing distributed systems usually follows the steps of single component tests, integration tests of components, and finally the system test. Especially, the integration tests is mainly aggravated by the following properties of distributed applications:

- true concurrency between components,
- race conditions and other types of non-determinism
- absence of a global clock, and
- message exchanges between components that are unobservable by a tester.

We assume that a test architecture for distributed systems consists of the following basic elements:

- the system under test (SUT), i.e. the executable implementation of the distributed software system to be tested,
- the tester that implements a test case and also assesses the results of a test run,
- points of control and observation (PCOs) and pure observation points (POs) between tester and SUT, and
- possibly test coordination procedures which coordinate the activities between different tester components.

The paper defines requirements to the test architecture that need to be matched to support the execution of test runs and their correct assessment afterwards. These requirements address the SUT, which must be prepared in a suitable way to perform a test run. Requirements in this category are referred commonly to rules of “design for testability”. Further requirements are suited to design testers that correctly assess the results of a test run. Note that our discussion is confined for the testing of functional aspects. We do not consider quality-of-service and performance aspects which are also relevant for distributed systems as discussed in [8].

As a result, two principal test architectures are presented: a *global tester* that observes and controls all distributed components of a SUT in a central manner, and a *distributed tester* that consists of a number of distributed, concurrent tester components, each of them collecting only partial information about the execution progress in components of the SUT. Especially, a distributed tester is of interest since it makes better use of system resources than the global tester that can be a performance bottleneck during a test run.

Assuming a grey-box testing approach and the right choice of PCOs and POs between tester and SUT, we show that also a distributed tester works correctly and brings up the expected result. The PCOs and POs must be provided in the implementation of the SUT to allow the tester to observe necessary information for assessing a test run. A common means for this purpose is the insertion of software probes into the SUT.

The considerations to the design of test architectures helped implement a prototype of a *Test and Monitoring Tool* (TMT) for testing distributed systems. This tool makes the concurrent flow of messages in a SUT visible and allows to compare the results of two test runs in order to detect differences in the execution order of messages in the SUT. The tool supports an on-line mode that displays message exchange in the SUT at run time and an off-line mode for post-mortem analysis. TMT provides different views on the concurrent system that focus on the hierarchical structure between components of the SUT, on the temporal order of messages exchanged between components, or on a detailed view of the contents of transmitted messages. Moreover, a TMT user is able to control the execution of the SUT interactively.

The paper is organized as follows. In Section 2 we introduce some basic notions as well as an example which are used in the following discussion. Section 3 describes the proposed test architectures. Section 4 presents the prototype of TMT that implements a distributed test architecture. Finally, Section 5 summarizes the presentation.

2 Preliminaries

2.1 Basic Notations

We suppose a specification of a distributed system as a parallel composition $\Im = M_1 \parallel \ldots \parallel M_k$ of interacting components. Each component realizes a certain function of the distributed system, e.g. in the form of a client and/or server. It is described by a sequential automaton (*labeled transition system*, LTS). Components communicate with each other solely via interaction points. The communication pattern used is synchronous communication and non-blocking send based on interleaving semantics. Transmitting messages and their receipt through interaction points are referred to *actions*.

Definition 1. A *labeled transition system* (LTS or *machine* for short) M is defined by a quadruple $(S, A, \rightarrow, s_0)$, where S is a finite set of states; A is a finite set of actions (the alphabet); $\rightarrow \subseteq S \times A \times S$ is a transition relation; and $s_0 \in S$ is the initial state.

To distinguish the different kinds of communication, we denote all inputs and outputs of the distributed system implementation from/to the environment as *external* (reactive system), analogously all inputs and outputs belonging to the inter-component communication as *internal*. Events appearing only inside a module are not considered.

Consider the simple system $\Im = A \parallel B$ whose LTSs are given in Figure 1. Under the assumption that actions a and c in each component synchronize, removal of parallel operator $\parallel$ by applying interleaved-based semantics rules yields the composite machine $C_\Im$ representing the global behavior of the distributed system $\Im$.

Figure 1. Example system $\Im = A \parallel B$ and its composite machine $C_{\Im}$.

The composite machine $C_{\Im}$ describes all possible sequences of actions the distributed system is allowed to perform, e.g. the system $\Im$ may perform the following sequences:

$\alpha = a.b.c.e.b.d$, $\beta = a.e.b.d$, $\gamma = a.b.d.e.a.b.c$ and so on.

During testing the system we have to show that the implemented system (SUT) follows the behavior described in its specification (specification-based testing approach). This testing concept is more formally defined in the following Section.

2.2 The Testing Goal

The testing goal within this specification-based testing approach is to show conformance between the specification of the distributed system and its specification, i.e. the SUT of the implementation obeys the requirements given in the (functional) specification, it implements the specification. To show conformance a formal model of the specification must be provided and the same formal model must be assumed for the SUT. We assume the formal model of a system of communicating machines as presented in the previous Section is used as a specification model as well it is imposed on the behavior of the SUT.

Conformance between SUT and specification can now be established formally.

Definition 2. Given the specification of a distributed system $\Im$ as a composite machine $C_{\Im}$. An implementation $SUT_{\Im}$ of $\Im$ conforms to its specification iff $SUT_{\Im}$ exposes the same set of action sequences $\boldsymbol{Seq}(SUT_{\Im})$ as are described in $C_{\Im}$, i.e. $\boldsymbol{Seq}(SUT_{\Im}) \supseteq \boldsymbol{Seq}(C_{\Im})$.

This definition states that the implementation at least needs to implement all possible action sequences of the composite machine defining the global behavior of the distributed system $\Im$. But it may implement more functionality.

To test a SUT for Definition 2, an acceptance test is appropriate. This test defines test cases that cover all transitions of the composite machine at least once. Such test cases are referred to a transition tour through a machine. Note that the definition includes also the internal events of a distributed system in order to assure that the communication between the components of the SUT works. This test is essential in the integration test of a distributed system, and a test architecture supporting this grey-box testing approach is required.

A composite machine is however not the best model to derive acceptance tests since it is usually too huge to be computed due to the state explosion phenomenon caused by interleaving. In [1] it was shown that a composite machine is not needed as a basis model for an accep-

tance test even if the same degree of test (fault) coverage should be maintained. Instead it is sufficient to derive acceptance tests from a reduced partial-order based model, called *Mazurkiewicz Trace Machine* (MTM), a model commonly used in partial-order verification techniques.

Possible test cases of our example are the following sequences:

$\sigma_1 = a.b.c.b.d.e.a$, $\sigma_2 = a.b.c.b.e.d.a$, $\sigma_3 = a.b.c.e.b.d.a$.

All three sequences can be equally employed in a test run. They possess the same fault detection capability [1]. They differ only in different orders of the interleaving actions *b*, *d* and *e*.

2.3 Challenges in Testing Distributed Systems

There are many challenges in testing distributed systems. One problem is the generation of appropriate test cases to test distributed systems sufficiently with a certain degree of test or fault coverage. Test case generation is in particular aggravated by the state explosion problem that occurs when the global system behavior is being analyzed. However, this problem in not discussed further in this paper.

Another challenge of testing distributed systems one is faced in the test execution phase is the intrinsic non-determinism of a distributed system under test. There exist three types of non-determinism in distributed systems:

1. non-determinism due to interleaving of concurrent actions of the SUT,
2. non-determinism within a component of the SUT, and
3. non-determinism due to race conditions within the SUT.

The first type of non-determinism refers to the fact that concurrent actions of a distributed system are independent from each other. Consequently, their occurrence within a global time of the system cannot be predicted. The action *b*, *d* and *e* from the example system in Figure 1 are such concurrent actions.

The second type refers to the fact that a tester may be not able to observe all actions going on within a component of a distributed system. Due to this limited observation the behavior of the component appears to be non-deterministic to the tester.

Finally, the third type of non-determinism occurs in the global behavior of the SUT. A typical example of a race condition is the receipt of two concurrent messages by a component of the distributed system. The further behavior of the system depends now on the order, in which the concurrent messages are received (e.g. concurrent write access to a shared variable, followed by a read of this variable).

The complete observation of the internal communication (grey-box test method) is required to observed the correct execution of internal actions within the SUT, but also to enforce a deterministic test run. By observing the internal communication of the SUT by means of a *controller* according to [4], the SUT is forced to follow a certain execution order defined by a test case that is implemented in the tester. To (partially) omit the control of internal observation, it must be proved whether this does not lead to a non-deterministic behavior of the SUT.

The first type of non-determinism can be avoided if the tester itself is implemented as a distributed system. See Section 3.2 for this discussion. Furthermore, if the analysis of the system specification shows that the 2nd and 3rd type of non-determinism cannot appear, then instant replay techniques to assure a deterministic behavior can be safely omitted and the tester only needs to observe the internal actions of the SUT via POs.

3 Test Architectures for Testing Distributed Systems

3.1 The Global Tester

In this Section, we describe possible test architectures which can be employed for the test of distributed systems. We first start with the simplest model, the *global tester*, which entirely simulates the environment of the STU during test run. The global tester centrally collects information of the distributed SUT, and derives the test verdict. Thereafter we extend the simple model such that the tester itself represents a distributed system.

A global tester is always implemented as a sequential machine T_G. The tester runs in parallel with the distributed system observing and controlling if necessary all external and internal actions of the SUT (grey-box test):

$$T_G \parallel SUT.$$

After starting the test run the tester executes the events of the test sequence σ step by step. This leads to a rendezvous or multi-rendezvous communication between the tester and the SUT. The tester participates in the execution of a test event and records it together with the components of the SUT participating in this test event.

The global tester performs an action sequence derived from the composite machine of the distributed system as a test case. This action sequence contains a certain, assumed interleaving order of concurrent events. This assumed interleaving order must be validated to exist in the SUT during the test run. However since the interleaving sequence consists of concurrent test events with no causal dependency between them, the action sequence assumed in the global tester is only observed by chance during a test run.

For example, possible interleaving action sequences of our system $\Im = A \parallel B$ of Section 2 that can be used to describe a global tester are the following three sequences:

$$\sigma_1 = a.b.c.b.d.e.a,\ \sigma_2 = a.b.c.b.e.d.a,\ \sigma_3 = a.b.c.e.b.d.a.$$

In these sequences the actions *b*, *d* and *e* are interleaved meaning that they may occur in any of the three possible orders during a test run. Thus, in practice it may be possible that a global tester implements σ_1 as the test case, but observes only sequences of σ_2 or σ_3 in a test run. The test verdict *fail* might be assigned to such a test run, although the SUT obeys the specification.

To prevent this situation from happening, the global tester needs to control the execution of internal events of the SUT (PCOs also for internal actions of the SUT). This behavior can be achieved if *instant replay techniques* are applied to guarantee a *deterministic test run* [3]. The instant replay technique defines measures to include software probes in the SUT that allow the tester to control the subsequent behavior of the SUT.

The global tester itself is modeled as an acyclic machine $T_G = (S_T, A_{CCT}, \rightarrow_T, S_{T0})$ with $|\sigma|+1$ states. The transitions in T_G are determined by the sequence of actions $\sigma = a_1.a_2. \ldots .a_n$ used as the test case:

$$S_{T0} \text{–}a_1\rightarrow S_{T1}, S_{T1} \text{–}a_2\rightarrow S_{T2}, \ldots, S_{Tn\text{-}1} \text{–}a_n\rightarrow S_{Tn}$$

For assigning the test verdict, a verdict label is attached to each state: *pass* to the final state and *fail* to all other states. If the tester reaches the final state after correctly executing σ, the test verdict *pass* is assigned to the test run. In all other cases the test run results in a *fail* verdict. The tester may not reach the final state in cases that a desired component is not able to participate in a certain test event; the test architecture including the tester and the SUT deadlocks in this case.

Figure 2 shows the global tester to test the system $SUT_{\mathfrak{I}} = A \parallel B$. Note again that the tester has access to the internal actions of the SUT. The model of the global tester implementing σ_1 as the test case is given in Figure 3.

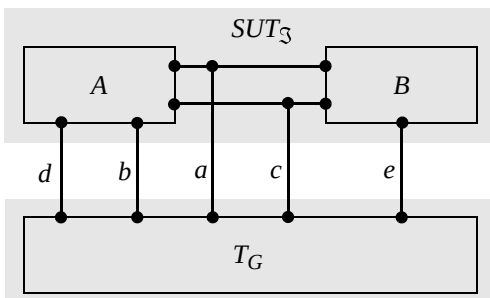


Figure 2. The global tester T_G for system $SUT_{\mathfrak{I}} = A \parallel B$.

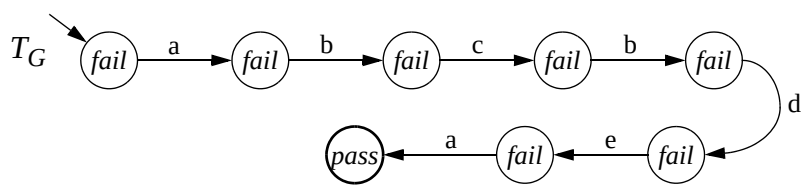


Figure 3. Global Tester of the system $SUT_{\mathfrak{I}} = A \parallel B$.

The advantage of the global tester approach is its simplicity. Due to its global view on the distributed system under test it can register the processed test events directly, with no further computations, as a unique sequence which preserves the correct causal dependencies between the actions of the SUT. A severe drawback of the global tester is however that it requires strict control over the execution of concurrent test events. This control might heavily intervene in the original behavior of the SUT. The question is whether there are other options to realize a test architecture that takes attention to the concurrency of test events.

3.2 Distributed Testers

3.2.1 General Characteristics

A distributed tester is characterized by the following properties:

- It consists of several concurrently operating tester components which process together, but independently a *global test case* (TC). The tester can be described by the same model as the SUT, e. g. as a set of communicating machines.
- Each tester component executes a *partial test case* (PTC). A PTC is projection of the global test case TC which comprises only those events which can observed at the PCOs and POs assigned to the particular tester component. The causal dependencies between the test events of a PTC are determined by the global TC.
- Each tester component observes a subset of the set of all PCOs and POs. PCOs exist for the external actions of the SUT, whereas POs are imposed on internal actions of the SUT. Their selection is a decision taken by the designer of the test architecture.
- The behavior of the tester components is controlled by a *Test Coordination Procedure* (TCP).

The general issue in distributed tester design is that the tester may assign a successful verdict to a test run although the SUT contains faults. This is possible because there is no global view on the SUT if a distributed tester is used. This problem can be tackled by a TCP between the distributed tester components, either by introducing synchronization events into the partial test cases of tester components or by the use of redundant PCOs/POs to observe internal events of the SUT simultaneously by several tester components. For example, if a component of the SUT sends a message to another component, one tester component observes the send event of this communication, whereas another tester component observes the resulting receive event.

We can describe a distributed tester T_D by means of a set of concurrent tester components

$$T_D = T_{D1} \parallel T_{D2} \parallel \ldots \parallel T_{Dn}$$

which are controlled by a TCP. Each tester component processes a sequential PTC via an acyclic machine that may be supplemented with synchronization events. We further assume the same markings of *fail* and *pass* to assign a test verdict as discussed for the global tester.

In a test run the distributed system $(T_{D1} \parallel T_{D2} \parallel \ldots \parallel T_{Dn}) \parallel SUT$ is executed. The tester components and the SUT participate in the respective test events and transfer from one local state to the next. If a test event or synchronization event of a TCP cannot be executed, the whole test architecture deadlocks. Only if all tester components reach their final state, the verdict *pass* is assigned to the test run.

3.2.2 Distributed Testers with Synchronization Events

A common distributed test architecture that is often used in testing distributed telecommunication systems uses synchronization events to implement a TCP. We discuss this type of distributed testers first.

We develop a distributed tester ${}^1T_D$ for the example system in Figure 1 and assume two tester components: ${}^1T_{D1}$ observes the actions *b* and *d* of the SUT, while ${}^1T_{D2}$ tests the actions *a*, *c* and *e*. Additionally, the tester components exchange the synchronization event *sync*. Figure 4 shows this test architecture.

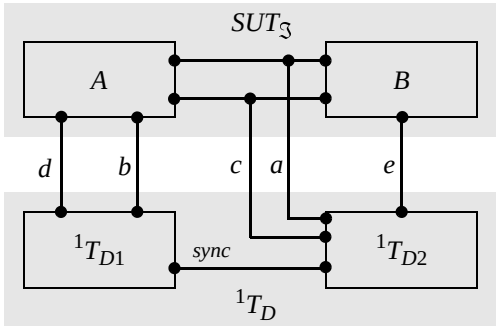


Figure 4. Test architecture of the distributed tester ^1T_D with synchronization event *sync*.

The derivation of test cases for the tester components is done as a projection of test events observable by the particular tester component from the global test case. Let us assume the global test case σ_1 = *a.b.c.b.d.e.a* is used. The projected partial test cases for the two tester components are the following:

$^1\sigma_{11}$ = *b.b.d* for tester component $^1T_{D1}$ and
$^1\sigma_{12}$ = *a.c.e.a*. for tester component $^1T_{D2}$.

These sequences must be supplemented with synchronization events. Synchronization events are always included when the control goes over from one tester component to the other. Consequently, the tester components run the following partial test cases:

$^1\sigma_{11}$' = *sync.b.sync.sync.b.d.sync*
$^1\sigma_{12}$' = *a.sync.sync.c.sync.e.sync.a*

Note that the synchronization events are indispensable for assuring the correct global view on the SUT. If, for instance, the first appearance of action *b* in component *A* of the SUT follows action *c*, a distributed tester without a test coordination procedure would not detect this fault.

Assuming that a tester component is assigned to each component of the SUT, the distributed tester can fully exploit the concurrency among test events. For example, total order of the interleaving actions *b*, *d* and *e* is not important for assessing a test run. Any test run is correct if only the causal dependency *b* before *d* is assured. Action *e* can interleave at any time instant (within the constraints of the *sync* events).

Thus, the distributed tester does not need to control the execution of the test case entirely. Each tester component can run independently its partial test case. Only coordination between the tester components is necessary to guarantee the correct global view on the SUT. As a consequence, it is possible to replace the PCOs for the internal actions of the SUT to mere POs, thus controlling the behavior of the SUT only via the external actions of the SUT. Control over internal actions might be still required however in the case of the existence of race conditions or internal non-determinism. See Section 2.3 for this discussion.

3.2.3 Distributed Testers with Redundant Observation of Internal Actions

The insertion of synchronization events is not the only possibility to realize coordination among tester components. Another and a more elegant option is the redundant observation of internal actions of the SUT by several tester components. This type of distributed test architecture is again discussed using the example from Section 2. It consists of two tester components

$^2T_D = {}^2T_{D1} \parallel {}^2T_{D2}$.

Each tester component controls and/or observes all interactions of a single component of the SUT. Thus, the tester represents an inverted image of the SUT. Figure 5 shows the test architecture of tester ^2T_D.

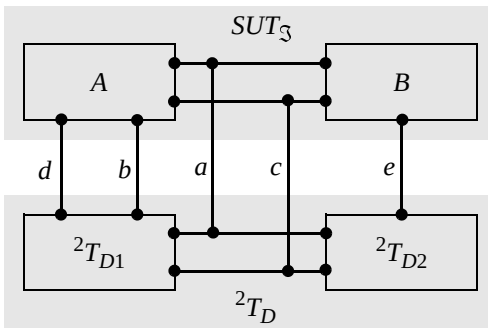


Figure 5. Test architecture of the distributed tester ^2T_D.

Partial test cases for the tester components are derived in the same manner as already described for the first type of distributed testers in Section 3.2.2. Using the global test case σ_1 = *a.b.c.b.d.e.a* again as the basis for the derivation of partial test cases, the tester components $^2T_{D1}$ and $^2T_{D2}$ need to implement the following partial test cases:

$^2\sigma_{11}$ = *a.b.c.b.d.a* for tester component $^2T_{D1}$ and
$^2\sigma_{12}$ = *a.c.e.a* for tester component $^2T_{D2}$.

Since both tester components observe the internal actions of the SUT, they coordinate each other to maintain the correct global view on the system. A practical solution to implement the redundant observation of internal communication is that in case of message passing, a tester component observes the send event of the message, whereas another tester component observes the receive event of this message. The causal dependency between the send and the receive event for the same communication action (message passing) can be assumed. This is the way how an in-house prototype of a test and monitoring tool works (see Section 4).

Note that also here concurrency within the SUT is greatly supported if a tester component is provided for each component of the SUT. The advantage of the second distributed test architecture is that no additional synchronization events are needed to coordinate the tester components. The renunciation of PCOs for internal actions of the SUT is possible if no other types of non-determinism exist in the SUT. See Section 2.3 for this discussion.

3.3 Test Cases for Distributed Test Architectures

This Section finally emphasizes the need of an appropriate description of test cases for distributed systems tested by means of a distributed tester. As discussed above, test cases described as sequences of test events are not very suitable since they do not respect the concurrency that exists among interleaving events. Hence, a notation of test cases is required that respects concurrency of test events. Such a notation should use a partial-order approach to make concurrent test event explicit. A standard notation for this purpose that has been in use for several years to describe (concurrent) system requirements of distributed telecommunication systems is the *Message Sequence Chart* (MSC) representation [9].

Using a representation similar to a MSC, the global test case for the example system in Figure 1 can be represented as depicted in Figure 6.

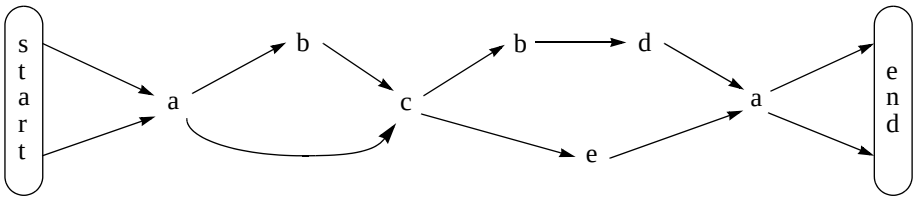


Figure 6. A concurrent global test case of system $\Im = A \parallel B$.

The concurrent global test case depicts the direct causal dependencies between test events. It does not assume a certain global order of all events, instead concurrent events are represented independently from each other. It can be easily seen that all three global action sequences σ_1, σ_2 and σ_3 that are contained in the composite machine of the example system in Figure 1 are still preserved in the concurrent test case [6]. In the diagram of Figure 6 the time axis is assumed to run from left to right.

4 A Tool for Testing and Monitoring Distributed Systems

4.1 Overview

In order to support clients within the Siemens Corporation, we are currently developing a *test and monitoring tool* (TMT) to support testing, monitoring and analyzing distributed applications. A first prototype of TMT was implemented in Java JDK 1.2 in more than 210 classes and 60,000 lines of code. The prototype can be applied to distributed systems implemented in C++ on UNIX or Windows NT platforms or to distributed systems in Java. The prototype was developed within the R&D Software and Engineering Department of Siemens as a contract work of its Information and Communication Networks Division. Figure 7 shows the main window of TMT prototype.

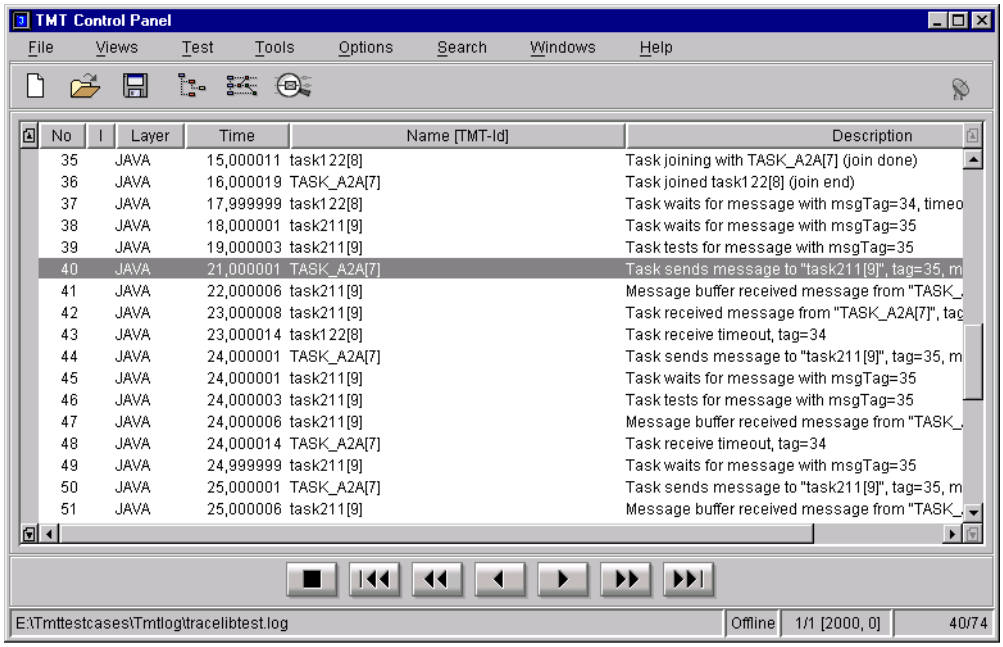


Figure 7. Main Panel of TMT.

The main panel shows the list of recorded test events observed from the SUT. The events are listed in an ascending temporal order of their receipt by the tester. Since the time stamps are generated locally, i.e., on the local machines where components of the distributed system are running during the test, the received sequential order of test events might not reflect the correct causal dependencies between the events if the accuracy of local clock synchronization is insufficient. This is however not a drawback since TMT considers only the causal dependency of test events for functional testing. The values of the time stamps matter only in case of performance tests that is not focus of this paper.

Further functions of the main panel of TMT include buttons to open different view charts of the SUT in the upper part of the window and VCR-like buttons to scroll through the event list.

4.2 Operation Modes of TMT

The TMT prototype supports two operation modes: on-line and off-line mode. In the on-line mode, TMT collects and displays all test events during the test run. Test events are produced from an instrumented source code of the SUT and refer to interaction events within components of the SUT or between SUT and environment. Thus, TMT realizes a global tester.

The off-line mode, on the other hand, allows to record partial information of a test run in distributed tester components. The tester components produce local files containing the observed test events during the test run. These local files are read by TMT after the test run has finished. The local information of test event sequences are merged within TMT according to the time stamp values of the test events. In this scenario, TMT realizes a distributed tester. The distributed tester components allow currently only to write observed test events to a file, but do not perform an assessment of the (partially observed) test run. This functionality is still centralized within the main tool of TMT.

Since the tester components do not use test coordination procedures between them, the observed test events must be issued redundantly by the SUT as discussed in Section 3.2.3. This is entirely in the responsibility of the developer of the distributed system to provide a SUT that is sufficiently instrumented to produce the desired test events.

4.3 The Test Library

TMT realizes a monitor tool and a distributed tester based on a redundant observation of internal interactions of the SUT. To generate the distributed tester components for a specific SUT with a set of communicating system components, a test library is provided that contains function calls to record test events by the tester. This test library resides on the machine where a component of the SUT is running. Connection between the tester and the SUT is established in such a way that the source code of the SUT is augmented with additional function calls of the test library in cases where interactions between components of the SUT will occur.

The test library does not allow in its current version to perform also the assessment of a test run. This functionality is entirely moved to the central main part of TMT (see Figure 7). The tester components implemented in the test library only allow to record the observed test events in a file or to send them directly to the TMT main component. The advantage of this approach is that the test library is generic for a large variety of implementations of distributed systems. Since the augmentation of the SUT is done in its source code, implementations in C++ and Java are currently supported by the test library only.

4.4 The Sequence Chart View of TMT

This view shows the execution process of interactions within a distributed system under test in a *Sequence Chart View* that is similar to a MSC graph. The test events received from the SUT are ordered according to their causal dependencies. Each sequential component is represented as a separate line within this view. Different icons in this view denote different interaction events (send, receipt of messages, creation of a new component and so on). See next Figure.

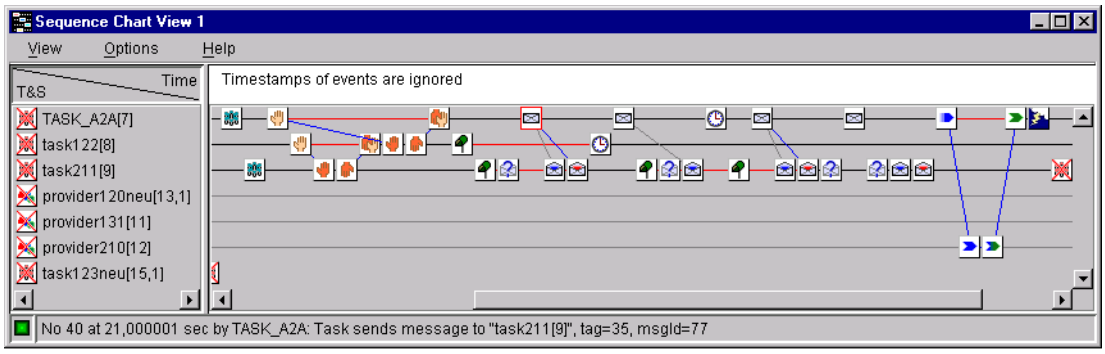


Figure 8. Sequence chart view of TMT.

Other views provided with TMT are

- a *Hierarchical View* on the structure of the SUT that depicts the composition of the SUT in components and sub-components, and

- a *Detailed View* to show details on received test events, e.g. certain parameter values.

4.5 Interactive Mode of TMT

TMT allows the generation of special interactive test events that help the tester control the subsequent execution of interactions in the SUT. At the moment this test event requires a manual interaction with the user of TMT. For this purpose a windows pops up that requires user inputs (see Figure 9). The further execution of the test architecture waits until this desired user interaction was performed. In this way, control over the behavior of the SUT can be achieved by TMT. This control is necessary to avoid non-determinism and to assure a deterministic test run (see Section 2.3).

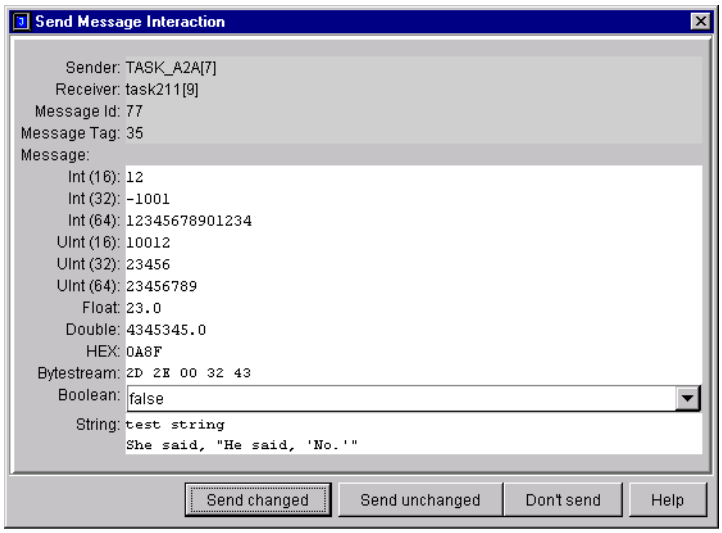


Figure 9. User interaction pop-up window of TMT.

4.6 Assessing Test Runs

An important feature of TMT is its ability to compare test runs with test cases or other test cases obtained in previous test runs. This assessment is, unlike the discussion in Section 3.2, is performed centralized in the main component of TMT to allow more flexibility of TMT for different test runs.

The implemented feature of assessing test runs takes into consideration the possible concurrency among the observed test events in order to correctly assign the test verdict. For example, although the two sequence chart views in Figure 10 look different, the test events in these views still obey the same causal dependencies. Thus, both test event sequences are considered to be equivalent by TMT, and the test verdict assigned to this test run is *pass*.

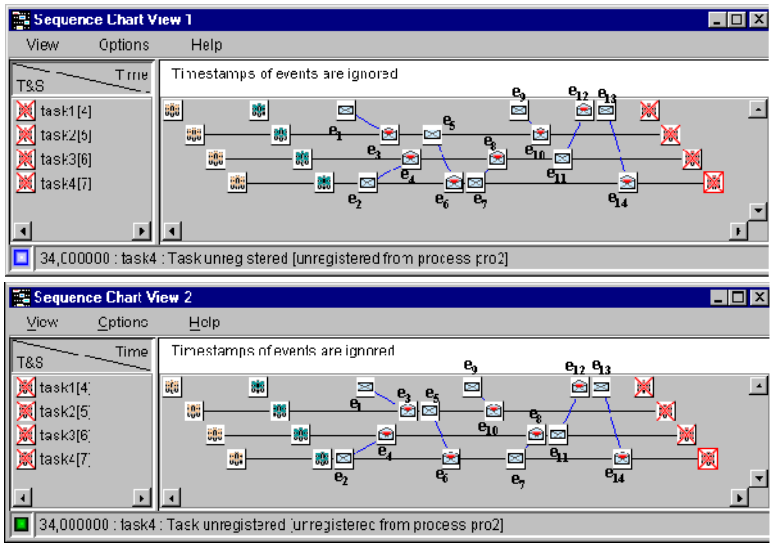


Figure 10. Two similar test runs.

5 Conclusions

In this paper we have suggested different architectures for the test of distributed systems. Two principle test architectures have been considered: a global and a distributed tester. The global tester is a sequential module which observes and controls all actions of the SUTs at its PCOs, whereas the distributed tester consists of several concurrent tester components which observe a partial behavior of the SUT only, but supports concurrency within the SUT. The latter must provide test coordination procedures to assure a correct and consistent global view on the SUT.

A prototype implementation TMT of a distributed tester was also presented. The current version of this prototype allows to monitor distributed applications and to assess a recorded test run according to a test case derived from a specification or recorded from a reference test run done before. This test assessment does not compare the test events of a recorded global test sequence in a sequential way, one after the other, but exploits the causal dependencies between test events to avoid misinterpretation caused by interleaving concurrent events.

6 Acknowledgment

The authors wish to thank all colleagues within Siemens who have participated in the TMT project. Special thanks in particular to Kai Tödter, ZT SE 2, and Klaus Berg, ZT SE 1, who designed the TMT architecture. Many thanks also to Michael Kälbling, Anja Hentschel, ZT SE 5, and Jürgen Schmitz-Foster, ZT SE 2, who implemented most of the TMT features, as well as to Sylvia Jell, ZT SE 2, who was responsible for the test library implementation of TMT. Last but not least, we thank Dietmar Lehmann, ICN WN ES D 42, and Erwin Reyzl, ZT SE 2, who helped funding and organizing the project.

7 REFERENCES

[1] A. Petrenko, A. Ulrich, V. Chapenko: *Using partial-orders for detecting faults in concurrent systems*; Proceedings of the IFIP 11th International Workshop on Testing of Communicating Systems (IWTCS'98), Russia, 1998.

[2] Sidhu, D. P.; Leung, T. K.: *Formal methods for protocol testing: a detailed study;* IEEE Trans. on Software Eng. 15 (1989) 4, 413–426.

[3] K. C. Tai, R. H. Carver: *Testing of distributed programs*; in A. Zomaya (ed.): *Handbook of Parallel and Distributed Computing*; McGraw Hill; 1995; pp. 956-979.

[4] K. C. Tai, R. H. Carver, E. E. Obaid: *Debugging concurrent Ada programs by deterministic execution*; IEEE Transactions on Software Engineering, vol. 17, no. 1 (Jan. 1991); pp. 45-63.

[5] A. Ulrich, S. T. Chanson: *An approach to testing distributed software systems*; 15th PSTV 1995; Warsaw, Poland; pp. 107-122; 1995.

[6] A. Ulrich, H. König: *Specification-based testing of concurrent systems*; IFIP Joint Int'l Conference on Formal Description Techniques, and Protocol Specification, Testing, and Verification (FORTE/PSTV'97); Osaka, Japan; Nov. 18-21, 1997.

[7] A. Ulrich: *Testfallableitung und Testrealisierung in verteilten Systmen*; Dissertation (in German), University of Magdeburg; Shaker Verlag, 1998.

[8] T. Walter, I. Schieferdecker, J. Grabowski: *Test architectures for distributed systems: state of the art and beyond*; Proceedings of the IFIP 11th International Workshop on Testing of Communicating Systems (IWTCS'98), Russia, 1998.

[9] Z.120: *Message Sequence Chart (MSC)*; ITU-T, Geneva, April 1996.

8 BIOGRAPHIES

Andreas received the M.Sc. degree (Diplomingenieur) in Computer Sciences from the University of Magdeburg, Germany, in 1992 and his Ph.D. degree (Doktoringenieur) from the same university in 1998. In 1998 he joined the Corporate Technology Division, the research and development center at Siemens AG, in Munich, Germany. His research interests include test architectures for distributed and concurrent systems, test case generation and specification-based testing, and formal specification techniques. He is an active participant at international conferences such as PSTV, FORTE, IWTCS and others.

Peter studied Computer Science at the University of Stuttgart, Germany and received his M.Sc. degree (Diplominformatiker) in 1991. He then joined the Siemens AG, Corporate Technology, and has been working in the field of software testing for object-oriented (C++, Java), distributed, and component-based software. He is co-author of several international conference publications, e.g. at EuroStar, Conference on Testing Computer Software and Software Quality Week.

Gunther is with Siemens AG, Corporate Technology in Munich, Germany. His current interests are testing distributed systems, component-based and object-oriented software. Gunther started his career in 1992 at DLR, the German Aerospace Center. He holds a M.Sc. degree in Applied Mathematics from the University of Kaiserslautern and a M.Sc. degree in Meteorology from the University of Hannover.

Genetic Algorithm with Cluster Analysis for Software Testing

Methods and Examples from Testing Real-Time Systems

Michael O'Sullivan, Siegfried Vössner and Joachim Wegener

Stanford University, USA DaimlerChrysler, Germany

Evolutionary Algorithms

Principles

- **Population of candidate solutions**
- **Selection**
  - "Survival of the Fittest" solutions
- **Recombination of solutions**
  - Reproduction, Mutation, Crossover ...

A GA @ Work

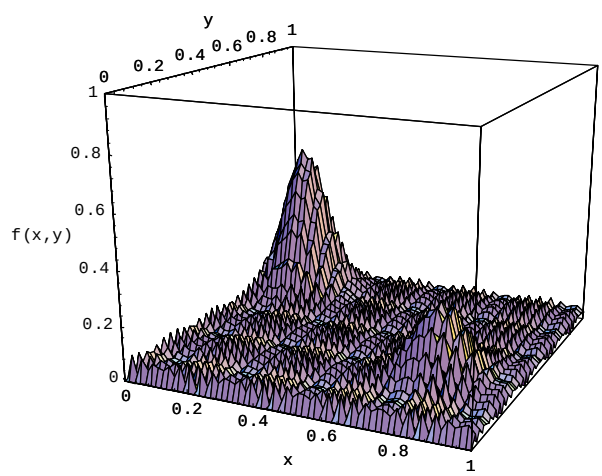
To allow a graphical representation of the distribution of the individuals in parameter space and to show clustering, a three dimensional fitness function $f(x,y,z)$ is chosen :

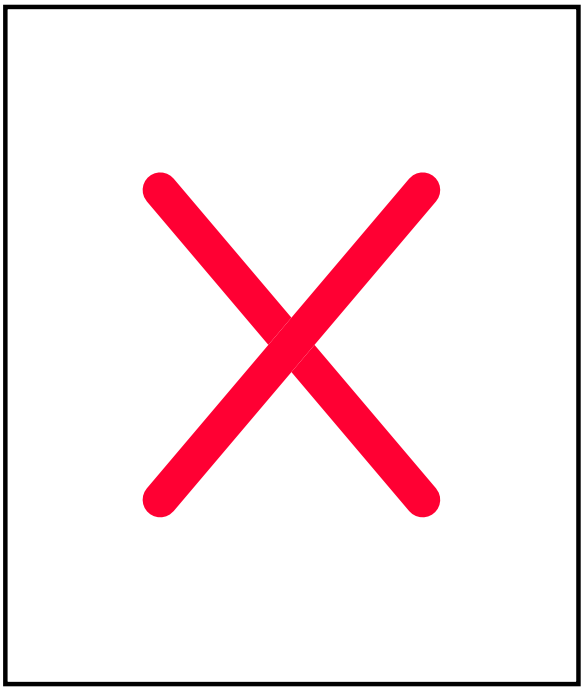
$$
\begin{aligned}
f(x,y,z) = 0.1|\sin(100x)||\sin(100y)||\sin(100z)| + \\
0.7e^{-100\left((x-0.8)^2+(y-0.3)^2+(z-0.3)^2\right)} + \\
0.8e^{-100\left((x-0.141)^2+(y-0.8)^2+(z-0.141)^2\right)} + \\
0.9e^{-100\left((x-0.8)^2+(y-0.8)^2+(z-0.8)^2\right)}
\end{aligned}
$$

The objective is to optimize this function in the range x, y, z [0,1]. Among a lot of local optima there are three "main" optima, which we calculate in order to make sure our analysis technique works :

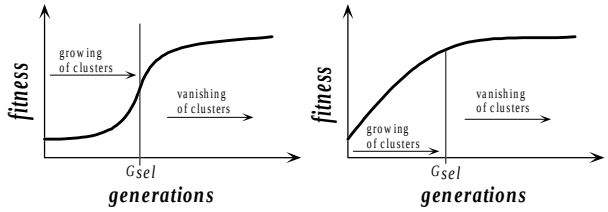
optimum	***x***	***y***	***z***	***f (x,y,z)***
1	0.801	0.299	0.299	0.797
2	0.141	0.801	0.141	0.897
3	0.801	0.801	0.801	0.998

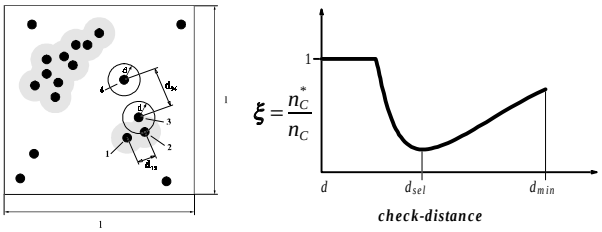
Fitness Function (for *z* = 0.2).



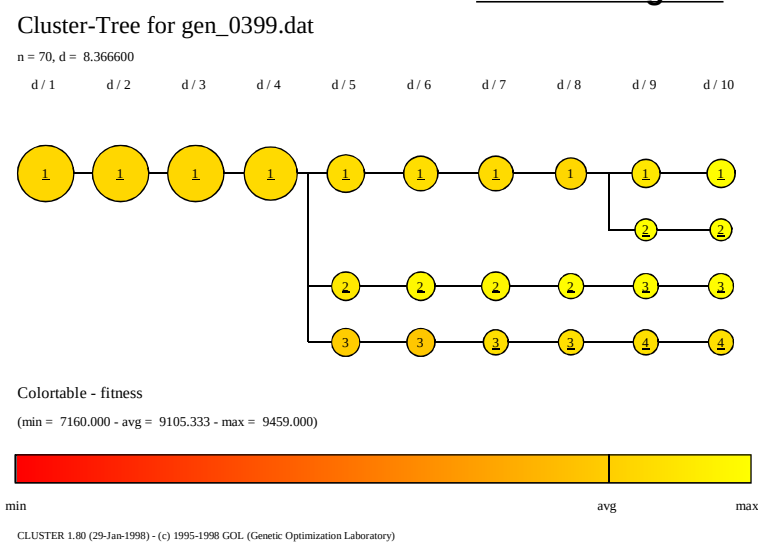


CLUSTER ANALYSIS

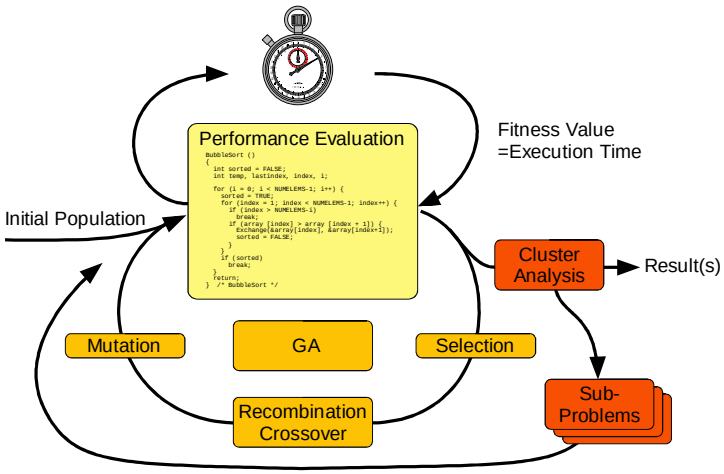




Cluster Diagram

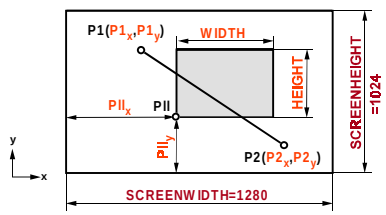


Optimization of Execution Times of Algorithms

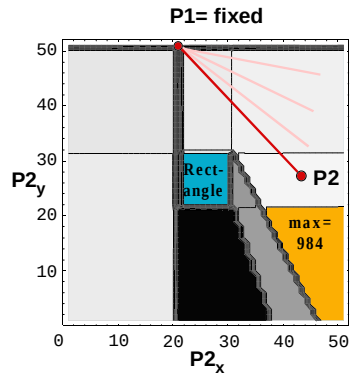


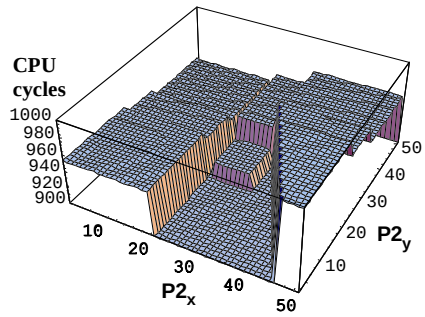
ISLINE.C

```c
/*****************************************************************/
/* Testobject: is line covered by rectangle                      */
/* t_answer isline( rectangle, line ):                           */
/*****************************************************************/
t_answer isline (struct Rectangle rectangle, struct Line line)
{   ....

    /* Compute corner points of the rectangle */
    rec.p1.x = rectangle.rect_p.x;
    rec.p1.y = rectangle.rect_p.y;

    rec.p2.x = rec.p1.x + rectangle.width;
    rec.p2.y = rec.p1.y;

    rec.p3.x = rec.p1.x;
    rec.p3.y = rec.p1.y + rectangle.heigth;

    rec.p4.x = rec.p1.x + rectangle.width;
    rec.p4.y = rec.p1.y + rectangle.heigth;

    /* Compute determinant for the line and the 4 sides: */
    for (i=0; i <= 3; i++) {
        if (i == 0) {
            p1x = rec.p1.x;
            p1y = rec.p1.y;
            p2x = rec.p2.x;
            p2y = rec.p2.y;
        }
        else if (i == 1) {
            p1x = rec.p1.x;
            p1y = rec.p1.y;
            p2x = rec.p3.x;
            p2y = rec.p3.y;
        }
        else if (i == 2) {
            p1x = rec.p2.x;
            p1y = rec.p2.y;
            p2x = rec.p4.x;
            p2y = rec.p4.y;
        }
        else /* (i == 3) */ {
            p1x = rec.p3.x;
            p1y = rec.p3.y;
            p2x = rec.p4.x;
            p2y = rec.p4.y;
        }

        /* Compute the determinant of both lines  */
        det = (((line.p2.x - line.p1.x) * (-p2y + p1y))
             -((-p2x + p1x)  * (line.p2.y - line.p1.y)));

        if (det != 0) {
            /* lines cross in one point */
            m = (((p1x - line.p1.x) * (-p2y + p1y))
               -((-p2x + p1x) * (p1y - line.p1.y))) / det;

            n = (((line.p2.x - line.p1.x) * (p1y - line.p1.y))
               -((p1x - line.p1.x) * (line.p2.y - line.p1.y))) / det;

            /* is point on the rectangle? */
            if ((m >= 0) && (m <= 1) && (n >= 0) && (n <= 1))
                return yes; /* line is covered by rectangle */
        }
    }

    /* there is no intersection between the given line and all sides
       of the rectangles! Is line on the rectangles border or
       entirely within?                                              */
    if ((rec.p1.x <= line.p1.x) && (line.p1.x <= rec.p2.x)
      &&(rec.p1.y <= line.p1.y) && (line.p1.y <= rec.p3.y))
        return yes; /* line is on or within rectangle */
    else
        return no;  /* line is not covered by rectangle */
} /* isline */
```

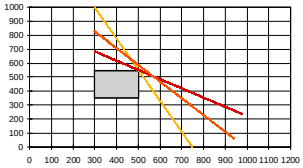


Fitness Landscape of *ISLINE*

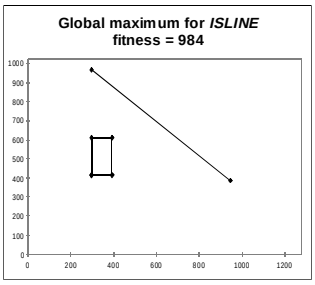


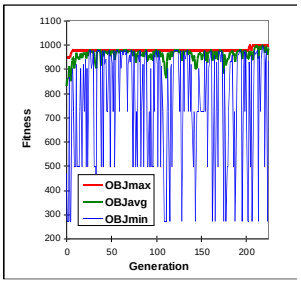


Screen dimensions:	**[0-50, 0-50] pixel**	**Line - P1x:**	**20**
Rectangle ll corner:	**[20, 20]**	**Line - P1y:**	**50**
Rectangle widh:	**10**		
Rectangle height:	**10**		



Finding the Global Optimum





Rectangle:

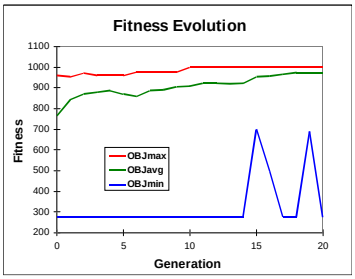
bottom-left corner = (297, 415),
width = 95, height = 197

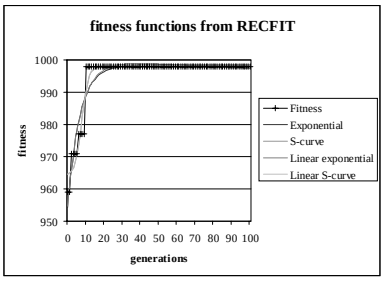
Line:

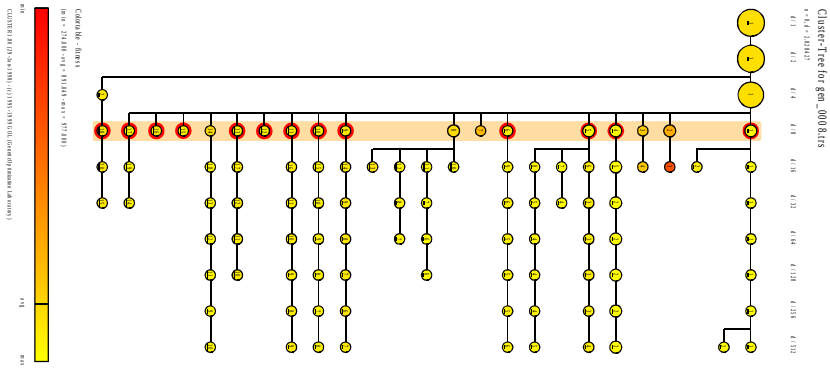
start = (297, 967),
end = 945, 388)

Fitness Evolution from GOAL
(20 individuals, 225 generations)

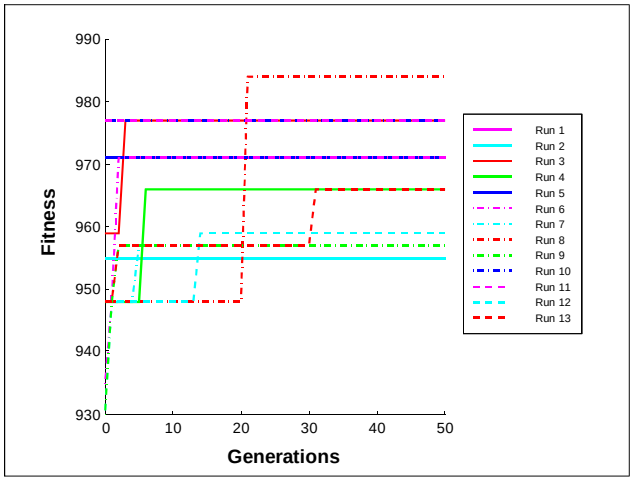
Finding the Local Optima



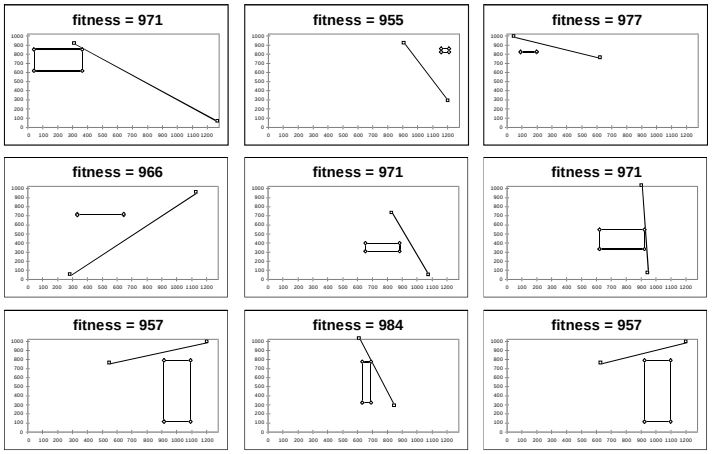




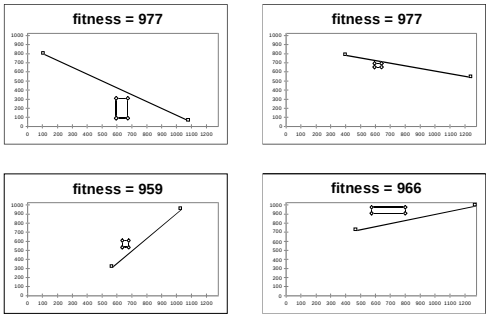
Finding Local Optima of *ISLINE*



13 Local Optima of *ISLINE* (1/2)



13 Local Optima of *ISLINE* (2/2)



Global versus Local

	"Global" Optimum	Local Optima
# Individuals:	20	100
# Generations:	225	20
Fitness Evaluations:	4500	15000
# Optima:	1	13
Fitness:	984	971, 955, 977, 966, 971, 971, 957, 984, 957, 977, 977, 959, 966.

Conclusion and Future Work

- **Experiments show:**
  - Evolutionary Testing is important
  - Cluster Analysis Adds valuable Info
    - Local Optima
    - Search Space Structure
    - Convergence Measure / Termination
- **To be done:**
  - GA Parameters
  - Search Space Coverage
  - Other Usage of Cluster Analysis

Genetic Algorithm with Cluster Analysis for Software Testing

Methods and Examples from Testing Real-Time Systems

Michael O'Sullivan, Siegfried Vössner and Joachim Wegener[*]

Stanford University, USA - DaimlerChrysler, Germany[*]

Abstract

Testing is one of the most complex and time-consuming activities within the development of embedded systems. Usually, embedded systems have to fulfil real-time requirements, and correct system functionality depends on both their logical correctness and their temporal correctness. An investigation of existing testing approaches shows a lack of support for testing temporal system behavior. With genetic algorithms, input situations that result in extreme temporal behavior may be identified automatically. This procedure is called evolutionary testing.

Evolutionary testing is an iterative testing procedure based on the use of genetic algorithms, which approximates the best-case and worst-case execution times of the system under test within several generations. The test is regarded as a discontinuous, nonlinear, optimization problem, with the input domain of the test object as search space, sets of test data as decision variables, and execution times as objective values.

Standard evolutionary testing finds input situations with the best-case and worst-case execution times. However the development and analysis of complex systems benefits from information about the entire input domain. We have therefore extended evolutionary testing with a technique based on cluster analysis. This technique examines the distribution of the individuals of a generation in the search space. Individuals usually form clusters in the search space after several iterations. These clusters provide information about the structure of the entire search space — something that might be unknown if the source code of the test object is too complicated to be analyzed or not available (e.g. in case of black-box testing). Cluster analysis provides additional information about local optima in order to detect several performance leaks within one test run. This is a good basis for optimizing the program code. Furthermore, cluster analysis gives information about the convergence state of the optimization run. This aids in forming a reliable termination criterion. The use of cluster analysis will be illustrated with a case study.

1 Introduction

The development of real-time systems is an essential industrial activity. The correct system functionality of real-time systems depends on their logical correctness as well as on their temporal correctness. In practice,

Corresponding Author: Siegfried Voessner, Dept. of Engineering Economic Systems and Operations Research, Stanford University, Stanford CA-94305-4023, voessner@leland.stanford.edu.

dynamic testing is the most important analytical method for assuring the quality of real-time systems. Dynamic testing is the only method that examines the run-time behavior of systems, based on an execution in the application environment. Testing is aimed at finding errors in the systems and giving confidence in their correct behavior by executing the test object with selected inputs. Often more than 50 % of the overall development budget for embedded software is spent on testing.

Most existing test methods focus on testing for logical correctness as they are not specialized in the examination of temporal correctness, which is also essential to real-time systems. For this reason, existing test procedures must be supplemented by new methods, which concentrate on determining whether the system violates its specified timing constraints. This work tries to fill the gap by giving support to testing temporal behavior. A temporal error is normally caused because outputs are produced too early, or the computation of the outputs takes too long.

The tester must therefore find the input situations with the longest or shortest execution times, in order to check whether they produce a temporal error. It is virtually impossible to find such inputs by analyzing and testing the temporal behavior of complex systems manually. However, if the search for such inputs is interpreted as a problem of optimization, genetic algorithms can be used to automatically find the inputs with the longest or shortest execution times. This automatic search for accurate test data by means of genetic algorithms is called evolutionary testing.

Evolutionary testing examines the temporal behavior of a system as a discontinuous, nonlinear, optimization problem, with the input domain of the system under test as search space, sets of test data as decision variables, and execution times as objective values. Evolutionary testing is an iterative testing procedure based on genetic algorithms. Unlike classical optimization techniques, genetic algorithms use the principles of an apparently very powerful optimization tool — biological evolution. They imitate fundamental principles like selection, recombination and mutation. The concept is to evolve successive generations of increasingly better combinations of those parameters which significantly effect the overall performance of a design. Starting with a selection of good samples, the genetic algorithm achieves the optimum solution by the random exchange of information between these increasingly fit samples (recombination) and the introduction of a probability of independent random change (mutation). The adaptation of the genetic algorithm is achieved by selection and reinsertion procedures since these are based on fitness values that permit an assessment of the performance of the samples.

The main objective of evolutionary testing is finding test data with execution times that violate the timing constraints specified. However with cluster analysis, evolutionary testing becomes an important analytical tool as well. By locating regions within the input domain that produce extreme execution times, the tester uncovers the structure of the input domain — something entirely unknown if the source code of the test object is too complicated to be analyzed or not available. Examining this structure can lead to improvements in the temporal behavior of the system. Local optima also exist within the clusters, these are important as test data when improving the system and may be found using short test runs. Further, cluster analysis adds information about the convergence state of the optimization run. This aids in forming a reliable termination criterion for evolutionary testing. When no timing violations are detected, deciding when evolutionary testing is complete is an important step.

The first section of our paper gives an overview of the current state of testing real-time systems in practice. The second section deals with evolutionary testing and describes how it is applied to examine the temporal behavior of real-time systems. The third section introduces cluster analysis and a strategy for detecting local optima. The fourth section examines evolutionary testing with cluster analysis applied to a simple example. After some concluding remarks the paper closes with a short outlook on future work.

2 Testing Real-Time Systems

Testing is one of the most complex and time-consuming activities within the development of real-time systems [Heath, 1991]. It typically consumes 50 % of the overall development effort and budget since embedded systems are much more difficult to test than conventional software systems. The examination of additional requirements like timeliness, simultaneity, and predictability make the test costly. In addition, testing is complicated by technical characteristics like the development in host-target environments, the strong connection with the system environment, the frequent use of parallelism, distribution and fault-tolerance mechanisms as well as the utilization of simulators.

Nevertheless, systematic testing is an inevitable part of the verification and validation process for software-based systems. Testing is the only method that allows a thorough examination of the test object's run-time behavior in the actual application environment. Dynamic aspects like the duration of computations, the memory actually needed during program execution, or the synchronization of parallel processes are especially important for the correct functioning of real-time systems.

Real-time systems must be tested for compliance with their functional specification and their timing constraints. An investigation of existing software test methods shows that a number of proven functional and structural test methods are available for examining logical correctness. For functional tests, test cases are derived from the requirements specification. A functional test procedure, for example, is the classification-tree method [Grochtmann and Grimm, 1993], which has already been used with success for the functional test of different real-time systems from various application fields [Grochtmann and Wegener, 1995]. When using structure-oriented test methods, test cases are determined on the basis of the program code. The aim of the test is to obtain a high coverage according to the selected test criterion, e.g. statement or branch coverage. An important principal disadvantage of structural testing is that the tester cannot check whether all specified requirements have been implemented into the system. Additionally, the tester must take into account that an instrumentation of the test object to monitor program execution may change the run-time characteristics of the program. Probe effects, i.e. deviations from the real system behavior are possible. Furthermore, the program under test may no longer fit into the target machine if the code is expanded to monitor execution [Hennell et al., 1987]. Schütz [1993] discusses suggestions how probe effects can be avoided.

For examining temporal correctness there are no specialized test methods available which are suited for industrial use [Wegener and Grochtmann, 1998]. For this reason, we have developed and examined a new approach for testing temporal behavior which is based on the use of genetic algorithms, namely evolutionary testing.

3 Evolutionary Testing

The major objective of testing is to find errors. Real-time systems are tested for logical correctness by standard testing techniques such as the classification-tree method. A common definition of a real-time system is that it must deliver the result within a specified time interval and this adds an extra dimension to the validation of such systems, namely that their temporal correctness must be checked.

The temporal behavior of real-time systems is defective when input situations exist in such a manner that their computation violates the specified timing constraints. In general, this means that outputs are produced too early or their computation takes too long. The task of the tester therefore is to find the input situations with the shortest or longest execution times to check whether they produce a temporal error. This search for

the shortest and longest execution times can be regarded as an optimization problem to which genetic algorithms seem an appropriate solution.

3.1 A Short Introduction to Genetic Algorithms

Genetic algorithms [Goldberg, 1989] represent a class of adaptive search techniques and procedures based on the processes of natural genetics and Darwin's theory of biological evolution. They are characterized by an iterative procedure and work in parallel on a number of potential solutions, the population of individuals. In every individual, permissible solution values for the variables of the optimization problem are coded.

The fundamental concept of genetic algorithms is to evolve successive generations of increasingly better combinations of those parameters which significantly effect the overall performance of a design. Starting with a selection of good individuals, the evolutionary algorithm achieves the optimum solution by the random exchange of information between these increasingly fit samples (recombination) and the introduction of a probability of independent random change (mutation). The adaptation of the genetic algorithm is achieved by the selection and reinsertion procedures used because these are based on fitness. The selection procedures control which individuals are selected for reproduction depending on the individuals' fitness values. The reinsertion strategy determines how many and which individuals are taken from the parent and the offspring population to form the next generation. The fitness-value is a numerical value that expresses the performance of an individual with regard to the current optimum so that different designs may be compared. The notion of fitness is fundamental to the application of genetic algorithms; the degree of success in using them may depend critically on the definition of a fitness that changes neither too rapidly nor too slowly with the design parameters.

Figure 1 shows the principle of a genetic algorithm. First, a set population of candidate solutions is generated, usually at random. Each individual in the population is evaluated by calculating its fitness. Usually, this will result in a spread of solutions ranging in fitness from "very poor" to "mediocre". The remainder of the algorithm is iterated until a stopping condition is fulfilled. Stopping conditions can be based on the number of iterations, the fitness function value or convergence measures discussed later. Iteration starts with selecting individuals from the current ("parent") population n according to a pre-defined selection strategy - giving "better" individuals a higher chance to be selected. These individuals are recombined to produce new solutions in an analogous way to biological reproduction. Recombination algorithms are many and varied. In addition, mutation is applied to introduce the variety required to cover a larger search space. The new individuals form the "offspring" generation n+1 and are evaluated for their fitness also. Before the loop starts from the beginning, generation n+1 becomes the "parent" generation from which members are being selected for the next generation n+2.

Genetic algorithms are particularly suited to problems involving large numbers of variables and complex input domains. Even for nonlinear and large search spaces of unknown structure, genetic algorithms have been used successfully. Since genetic algorithms search from a population of points rather than from a single point, the probability of getting stuck at local optima is significantly reduced compared with more traditional optimization techniques, like gradient based techniques. The use of mutation and subpopulations can further reduce the chance of getting stuck in local optima [Sthamer, 1996].

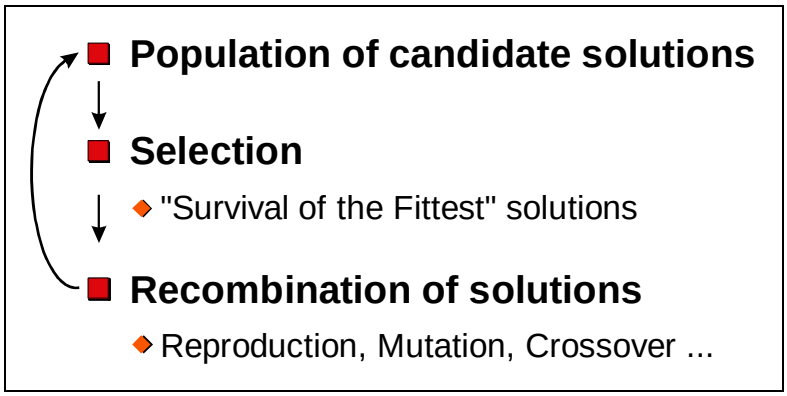


Figure 1. Principle of Genetic Optimization

Genetic algorithms have been successfully applied to various testing problems. Several papers deal with structural testing (e.g. [Jones et al., 1998] and [Roper, 1997]), others concentrate on test case generation based on formal specifications [Jones et al., 1995], the testing of APIs [Boden and Martino, 1996], and testing for robustness [Schultz et al., 1993]. Wegener et al. [1997] offer a brief survey on applications of evolutionary testing.

3.2 Application of Genetic Algorithms to Testing Temporal Behavior

In this work which is based on [O'Sullivan, Voessner, Wegener 1998], a genetic algorithm with cluster analysis is used for testing the temporal correctness of real-time systems by finding the set of input parameters that cause the worst execution times of an algorithm. The problem can be stated as an optimization problem. Where the time it takes to execute the algorithm is the objective function value. In practical applications there are many local optima in these problems. Since the computational effort is relatively high, we would like to find as many of them as possible.

Figure 2 illustrates how our framework proceeds: The initial population is generated at random. Each individual of the population represents a test datum with which the test object is executed. For every test datum the execution time is measured. The execution time determines the fitness of the respective individual or test datum. If one searches for the worst-case execution time, test data with long execution times obtain high fitness values. If one searches for the best-case execution time, individuals with short execution times obtain high fitness values. Afterwards, members of the population are selected with regard to its individual's fitness and subjected to combination and mutation to generate a new population. First, it is checked that the generated test data are in the input domain of the test object. Then the individuals of the new generation are also evaluated and combined with the previous generation to form a new population according to the survival procedures laid down. From here, this process repeats itself, starting with selection, until a given stopping condition is reached or a temporal error is detected.

Cluster Analysis, in the outer loop drawn in Figure 2, is used to analyze the distribution of individuals in the current population providing information about local optima, the structure of the search space, and the state of convergence. This information can be used to break the problem down in smaller, more easily tractable sub-problems (decomposition) and/or as a reliable termination criterion.

3.3 Experimental Setup

The experiments presented in this paper were done on SUN Ultra I Sparc workstations. We used Quantify 3.1 [Rational Software 1998], a commercial package to determine the execution time of the algorithm and GOAL [Voessner and Braunstingl 1998] for the genetic algorithm part.

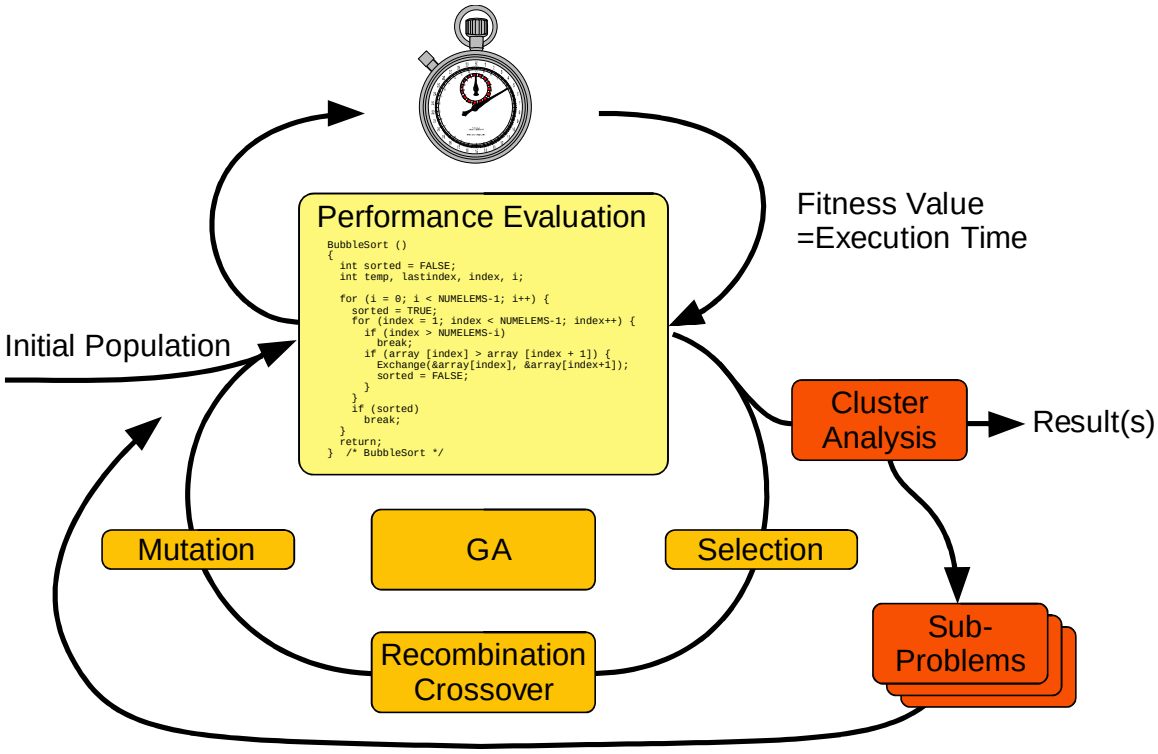


Figure 2. Evolutionary Testing of Temporal Behavior

4 Cluster Analysis

Cluster analysis for genetic algorithm output is a technique developed [Vössner and Braunstingl 1996a] to provide more information about the population of individuals at a given generation of a genetic algorithm (GA). Selecting an appropriate generation from a GA run and applying cluster analysis provides considerable information about the behavior of the fitness values over the search space. Additionally, using cluster analysis on the population from the most recent generation produced by a GA helps determine if the GA should terminate.

4.1 Analyzing a Generation

To simplify cluster analysis for a generation, all parameter intervals are normalized to the range $0 \leq$ *parameter* ≤ 1, so that the parameter space, formerly a hyper rectangle, is now a hyper cube (with unit edge length and volume). In this normalized parameter space, the distance d_{ij} between two individuals *i* and *j* is taken with the Euclidean metric and calculated for all individuals. Doing this yields a symmetric distance matrix ($n \times n$) where $d_{ij} = d_{ji}$ for $i, j = 1, \ldots, n$ (see Figure 3).

Point	**1**	**2**	**3**	**4**	**...**	***n*-1**	***N***
1	0	d_{12}	d_{13}	d_{14}	...	$d_{1\ n-1}$	d_{1n}
2		0	d_{23}	d_{24}	...	$d_{2\ n-1}$	d_{2n}
3			0	d_{34}	...	$d_{3\ n-1}$	d_{3n}
4				0	...	$d_{4\ n-1}$	d_{4n}
...					0	$d_{...n-1}$	$d_{...n}$
***n*-1**						0	$d_{n-1\ n}$
N							0

Figure 3. Symmetric Distance Matrix

Clusters of individuals are defined using the distance matrix. The distances of all individuals are compared with a particular threshold value d_{check} (also called the *check distance*). If the distance between two individuals is smaller than d_{check}, they lie in the same cluster.

Clusters identified by this algorithm may have arbitrary shapes, which is especially useful if the fitness function has an asymmetric gradient around a local optimum. Suppose the optimum is the highest point of a narrow ridge. Then the individuals will accumulate along this ridge, forming a longish, thin cluster.

The clustering algorithm is summarized in the following pseudo code and depicted in Figure 4, where we define a cluster to consist of at least 2 individuals (solution points). Given any two individuals *i* and *j*, if the distance between them $d_{ij} < d_{check}$, then they should be members of the same cluster.

There are three cases:

1. If they do not belong to a cluster yet, they form a new one.
2. If only one of the two individuals already belongs to a cluster, the other one joins this cluster.
3. If both individuals are members of different clusters, these clusters are combined thus forming a single one.

4.2 Forming a Cluster Diagram

The clusters for a specific check distance are very dependent on that threshold value. Observing the clusters as the check distance changes provides complete information about the clustering of individuals in a generation. Decreasing the check distance by discrete factors, starting from the hypercube's space diagonal d_0 (where exactly one cluster is found) down to a distance where no clusters exist, and performing the cluster analysis described above for each check distance forms a cluster diagram (see Figure 5).

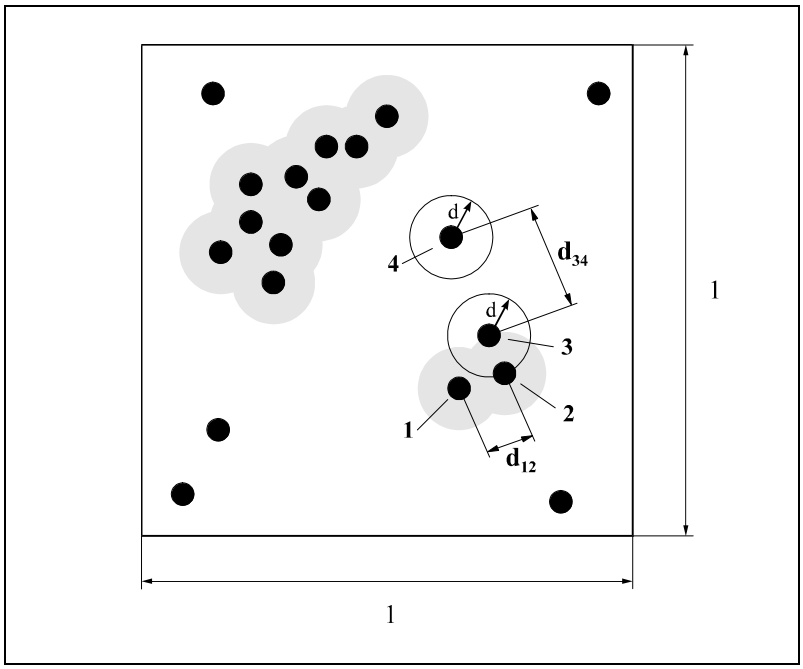


Figure 4. Detection of clusters in the unit square

The results of the different cluster analyses, ordered by decreasing check distance (d/1, d/2, d/4, …), are arranged in the cluster diagram, where each circle represents a cluster, and the connection from one cluster to the next outward cluster(s) shows how clusters split with decreasing check distance.

The diagram (Figure 5), generated by CLUSTER [Vössner and O'Sullivan 1998a] shows the number of clusters, their sizes, history of origin and the average fitness of the corresponding individuals. The numbers within the circles identify them, while the size of the circles is proportional to the number of individuals forming that cluster. The shading of a cluster indicates the average fitness of the individuals within it. One of the shades is labeled with "avg" in the fitness scale at the bottom of the diagram. If a cluster number is underlined, the average fitness of its elements is greater or equal to the average fitness of all individuals. We call such a cluster "outstanding."

4.3 Detecting local optima

Using cluster analysis, finding local optima in a GA search space is a relatively simple procedure, one that should take a less CPU time on average than finding the global optimum. Initially the appropriate generation must be selected from a previous GA run. The clusters found by analyzing this generation are regions that are likely to contain local optima. By taking each of these clusters in turn, sampling an initial population from the cluster and running a new GA, the local optima may be refined from the cluster.

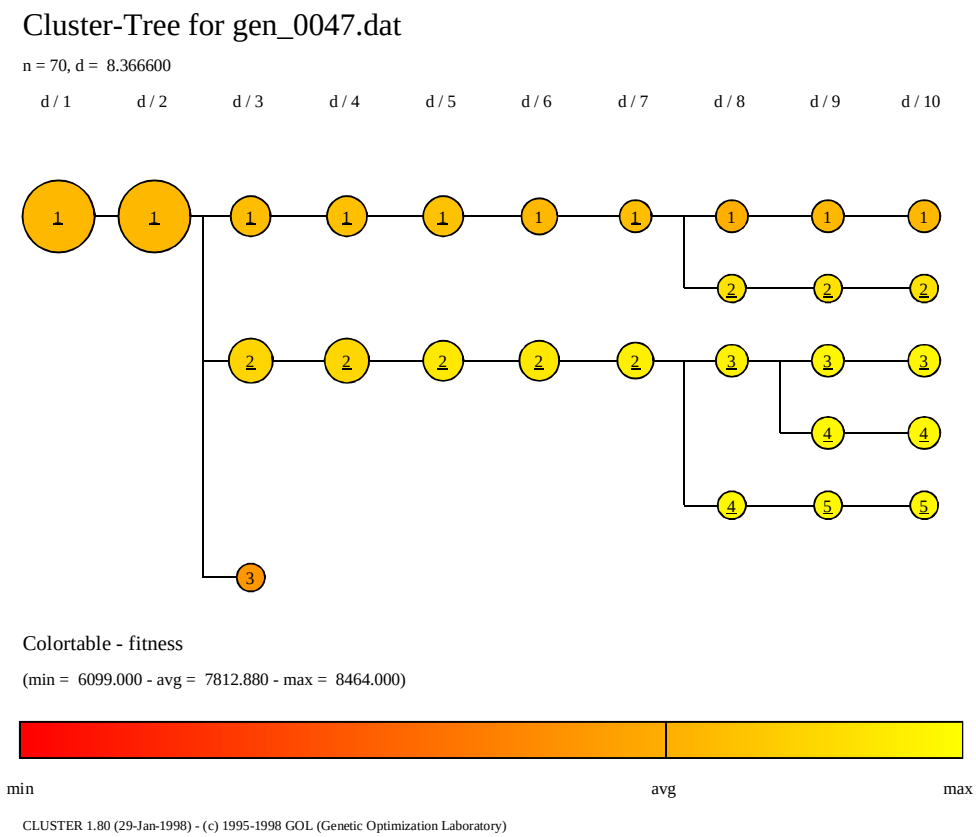
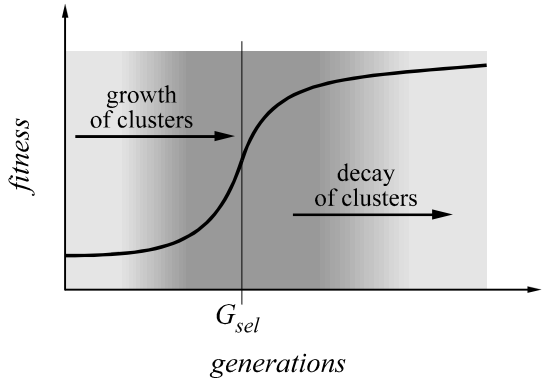


Figure 5. Example Cluster Diagram

Selecting a generation. By taking note of the average fitness of the population throughout the initial GA run, an appropriate generation for cluster analysis is selected. The average fitness of all generations of a GA run yields function shapes similar to the ones shown in Figure 6.



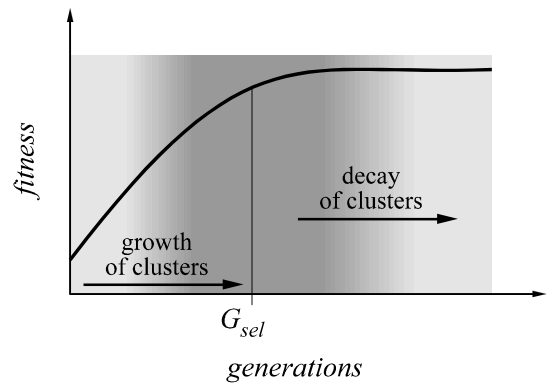


Figure 6. Selection of a Generation for Cluster Analysis

When the individuals are distributed randomly, as in the initial population, no definitive clusters have formed. As the GA proceeds, individuals start moving towards areas of higher fitness, more clusters appear, and the average fitness increases rapidly. Then the individuals begin congregating into one or more optima. As the initial clusters start to disappear or split, the average fitness grows more slowly.

The following heuristic has turned out to be very useful for determining an appropriate generation to begin cluster analysis - at least for all our optimization problems. We recommend selecting the population G_{sel}, which comes closest to the point of inflection respectively, to the point of maximum curvature of the curves shown in Figure 6.

For further analysis it is necessary to determine a proper check distance. On one hand as much local optima as possible should be detected - on the other hand the number of clusters to be examined should be as small as possible. That means if the check distance is too big a cluster may not even mark a local optimum, and if the distance is too small a cluster indicating a local optimum could disintegrate into a lot of smaller ones. We found that the quotient x of the number of "outstanding" clusters n_C^* and the number of all clusters n_C has in general a shape similar to Figure 7 ($x = n_C^* / n_C$).

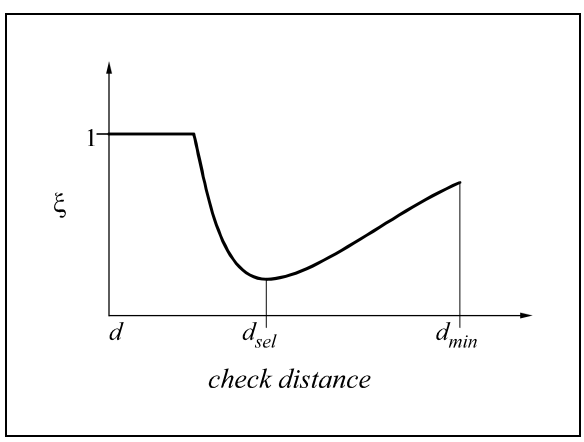


Figure 7. Normalized Number of Outstanding Clusters versus Check Distance

This figure shows the general shape of x versus check distance. The diagram starts at distance d which is the diagonal of the hyper cube and decreases until the smallest possible check distance d_{min} is reached, where no more clusters can be found.

The shape of x can be explained by the following consideration: During a GA run the density of individuals increases in regions where the fitness function has high values - which is generally known. If we are looking at the shape of x of a generation we first find exactly one outstanding cluster, containing all individuals ($x = 1$). Then the total number of clusters n_C grows faster than the number of outstanding ones (n_C^*). The ratio x goes down. Reducing the check distance further, clusters of low density and fitness start to disappear first. Therefore x starts growing again. From that point on outstanding clusters, marking areas of possible local optima, start to disappear too. That makes it reasonable to choose the check distance d_{sel} where x has a minimum:

$$d_{sel} = \min[x]$$

Searching the clusters. If the original GA run has a small population size, the clusters discovered during the analysis may not be informative. However this run may terminate after few generations, only long enough for clusters to begin to form. The GA run starting from each cluster may have a small to medium population size as the search is only in the neighborhood of the cluster. This run should also converge quickly as the population moves toward the local optimum. By keeping the GA run with a large population short and using multiple GA runs with small populations and quick convergence, experiments have found local optima using considerably less fitness evaluations per optimum than required to find the global optimum.

4.4 Checking for convergence

When a GA has converged to a global maximum, then the individuals in the population have clustered on one or more points with the same fitness value. By looking at the cluster analysis of the most recent generation, one can determine if this has taken place. If the clusters split suddenly from one cluster into several clusters that even exist at small check distances then the population has clustered around several points. If the fitness of these clusters is almost identical, then all these points have the same fitness value. Both of these conditions may be ascertained by examining the cluster diagram (see Figure 8) of the generation, or automatically by checking the size and fitness of the clusters.

This criterion may be less useful with small populations as there are not enough individuals to provide informative clusters. The larger the population the better the information provided by cluster analysis.

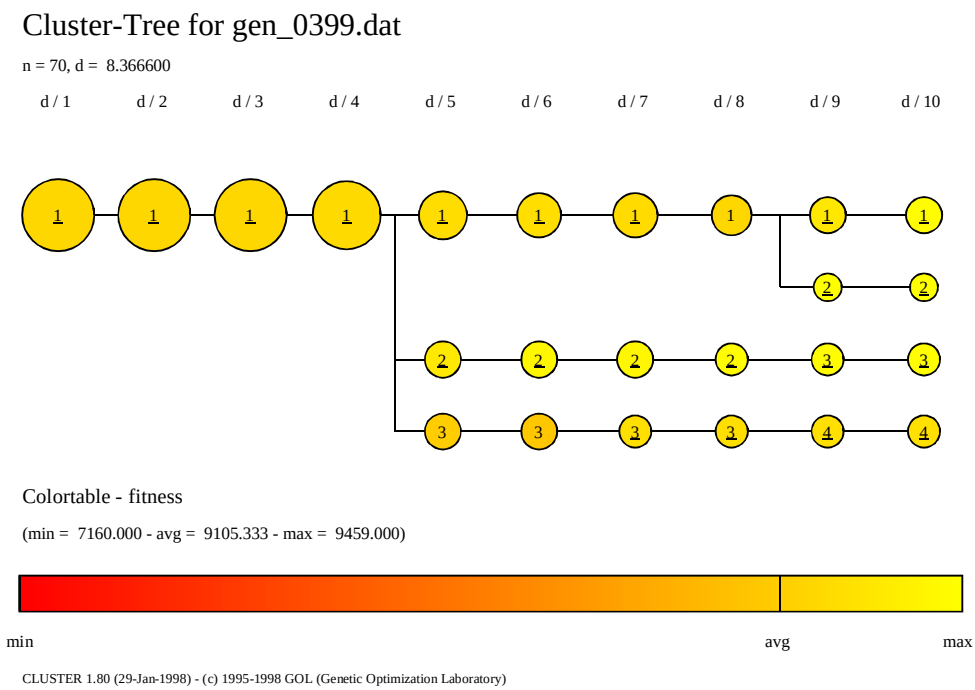


Figure 8. Cluster Diagram for a Convergent Generation

5 The *isline* Experiment

While incorporating cluster analysis into the existing evolutionary testing procedures, experiments were performed on several algorithms. We examine the findings for the *isline* algorithm, a simple graphical algorithm with 8 input parameters.

The *isline* algorithm checks if a line is covered by a rectangle. The rectangle consists of one point, the bottom-left corner (PII), and the two lengths, the rectangle width (WIDTH) and height (HEIGHT). A start (P1) and an end point (P2) define the line (see Figure 9). With these inputs, *isline* checks to see if any part of the line is covered by the rectangle. Both the rectangle and the line are contained within a bounding rectangle, i.e., the coordinates of the rectangle corners and the start and end of the line lie within the bounding rectangle.

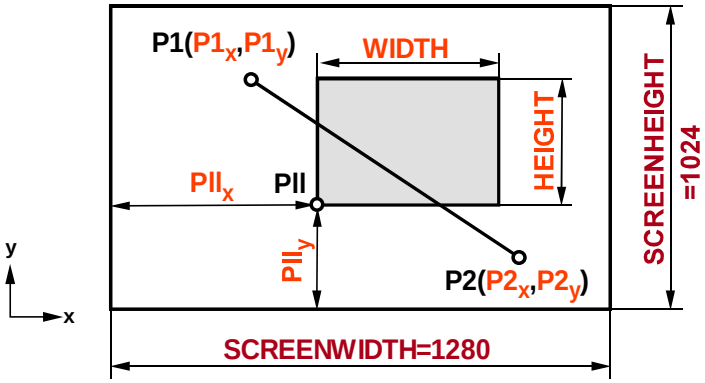
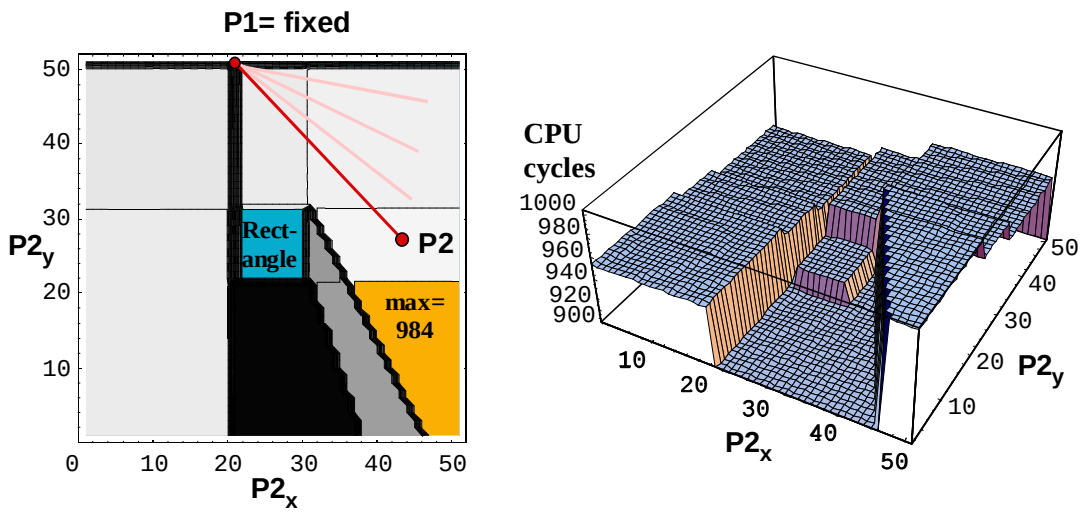


Figure 9. The *isline* Algorithm

5.1 Analysis of *isline*

By carefully examining the *isline* code one sees that the worst case performance occurs when the start of the line has the same *x*-coordinate as the bottom-left corner of the rectangle. Using a bounding rectangle with the bottom-left corner at (0, 0) and width and height both 51 (i.e., the *x*- and *y*-coordinates are in the range [0, 50]) a rectangle was placed with a bottom-left corner at (20, 20) and width and height both 10. Setting the start of line at (20, 50) the end of the line was allowed to vary throughout the bounding rectangle. Using the rectangle and the various lines as input to *isline*, Figure 10 shows the variation in fitness values (i.e., the number of CPU cycles used by *isline*) as the end point ranges throughout the bounding rectangle.



Screen dimensions:	**[0-50, 0-50] pixel**	**Line - P1x:**	**20**
Rectangle ll corner:	**[20, 20]**	**Line - P1y:**	**50**
Rectangle widh:	**10**		
Rectangle height:	**10**		

Figure 10. CPU Cycles Used by *isline*

Figure 10 shows that even when the only difference between solutions is the end point of the line there are large areas in the search space with the same fitness value (called "fitness plateaus"). By varying the placement and size of the rectangle and/or the placement of the start of the line similar fitness function surfaces may be generated. The configuration of the rectangle and the line (with respect to each other) are the important factors that determine the execution time of *isline*. The same configuration may be repeated multiple times with different inputs, so the search region contains a multitude of local and global optima (see Figure 11), dependent on the size and position of the bounding rectangle.

5.2 Finding the optima for isline

As the maximum fitness value for *isline* is known and attained, there are no problems with termination criteria for evolutionary testing with *isline*. However, cluster analysis may be used to isolate local optima. The global optimum of *isline* was found running GOAL [Vössner and Braunstingl 1998] with 20 individuals for 225 generations (i.e.: 4500 fitness evaluations). The global optimum and the fitness evolution curve for this run are shown in Figure 12.

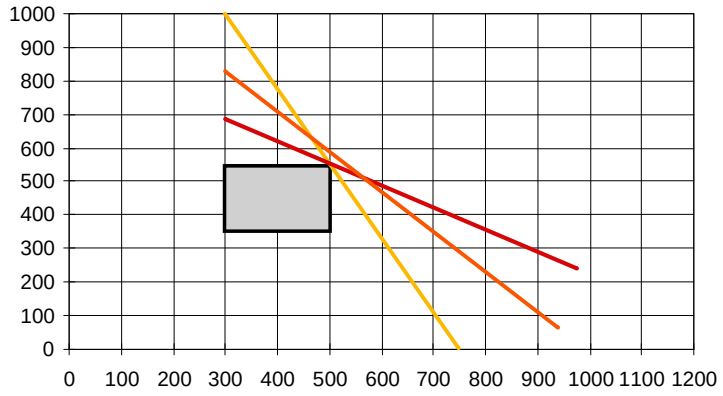
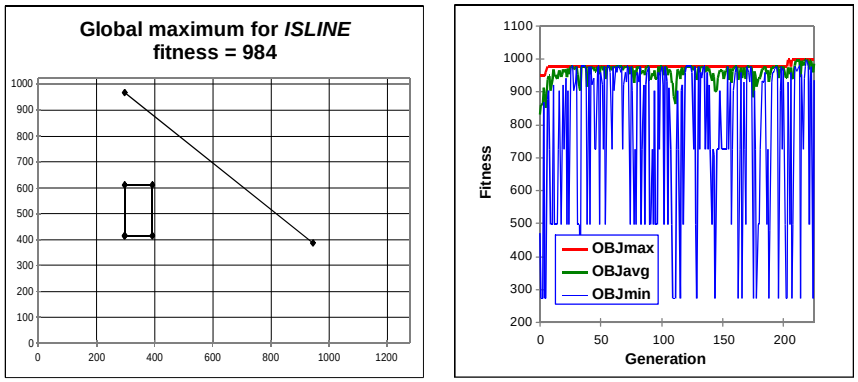


Figure 11. Set of Global Optima for *isline*

The GA run to find the global optimum used a population size too small to provide informative clusters. By running GOAL with 100 individuals for 20 generations, clusters are formed and CPU time is conserved (2000 fitness evaluations). This gave the fitness evolution curve in Figure 13.



Rectangle:

bottom-left corner = (297, 415),
width = 95, height = 197

Line:

start = (297, 967),
end = 945, 388)

Fitness Evolution from GOAL
(20 individuals, 225 generations)

Figure 12. Finding the Global Optimum for *isline* (20 individuals, 225 generations)

The appropriate generation was selected by using RECFIT [Vössner and O'Sullivan 1998b] to find a function that approximated the fitness evolution curve, Figure 14 shows the results of the four different functions (exponential, S-curve, linear exponential and linear S-curve) compared with the fitness evolution curve.

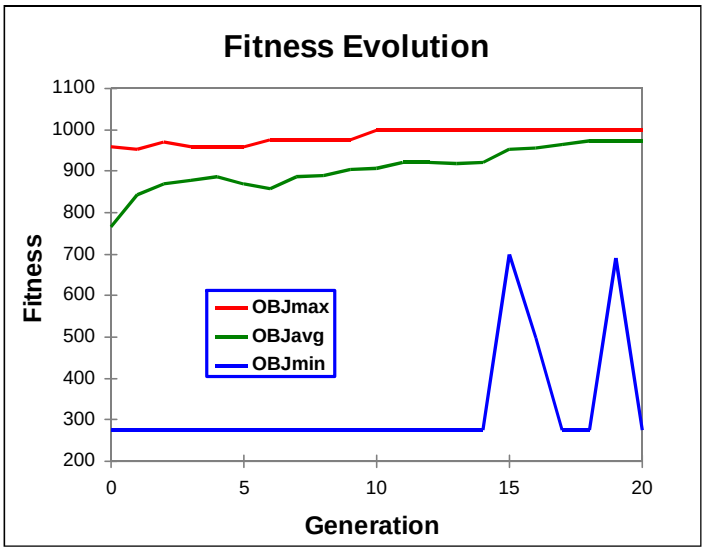


Figure 13. Fitness Evolution from GOAL (100 individuals, 20 generations)

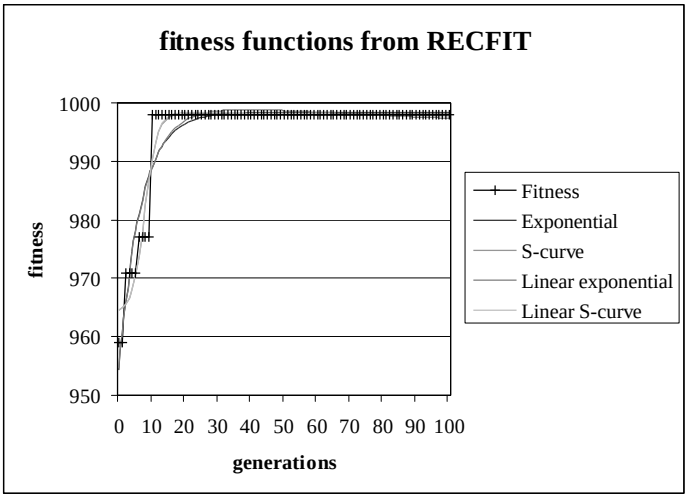


Figure 14. Comparison of Functions from RECFIT with the Fitness Evolution Curve

The best fit was provided by the linear S-curve. Using this function approximation the best generation for cluster analysis was shown to be generation 8. The cluster analysis of this generation gave the cluster tree in Figure 15.

Looking at this cluster tree shows that there are several areas that contain local optima. The 4th level of the cluster tree (highlighted) has a large number of clusters, but the check-distance (the distance defining if two points form a cluster) is large enough so the clusters are not too exclusive. There are 13 outstanding clusters (circled), each of which may be an area where a local optimum may reside. Looking at each of the 13 clusters in turn, the individuals from the cluster form the initial population, i.e., initially the GA searches only in the neighborhood of the cluster, and GOAL was used with 20 individuals for 50 generations (1000 fitness evaluations). The fitness evolution curves are shown in Figure 16.

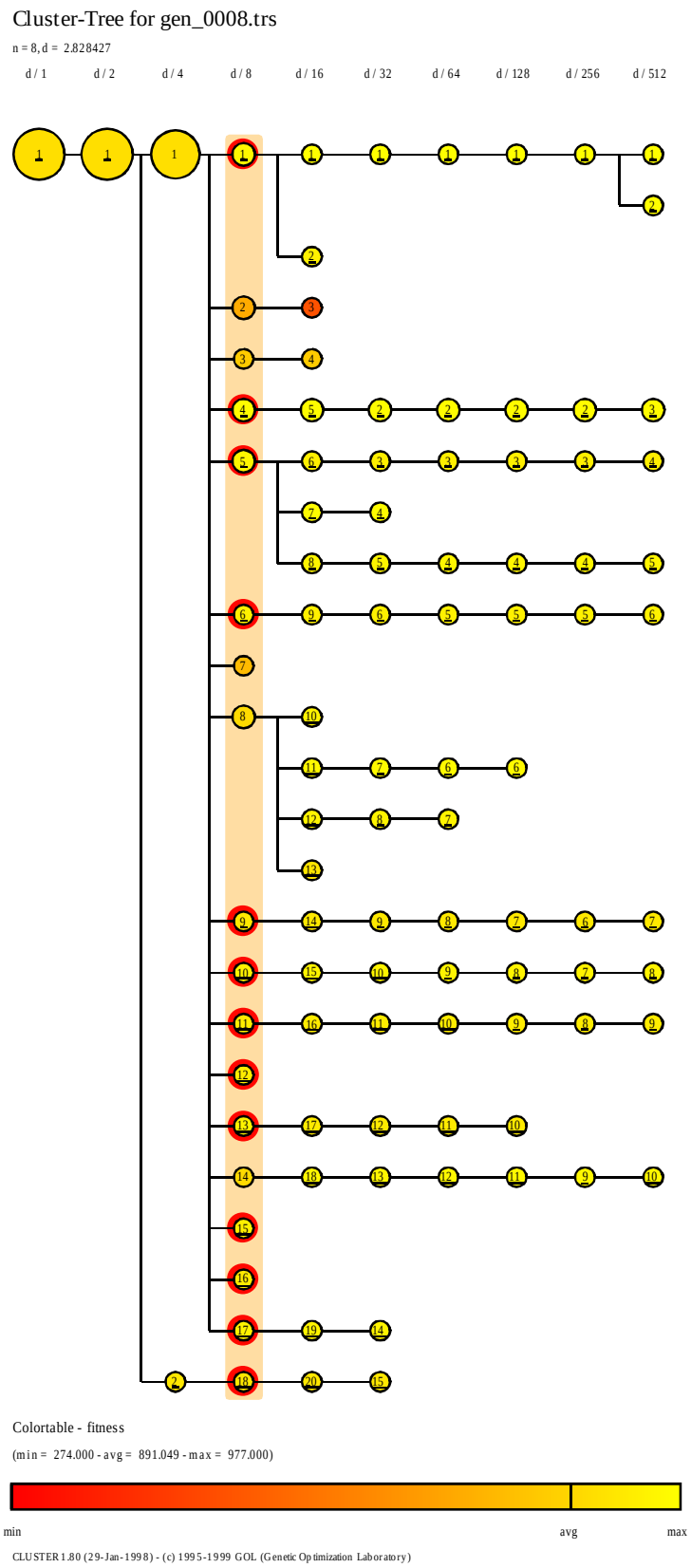


Figure 15. Cluster Tree for Generation 8

Local optima for *isline* were found by each of these runs and these optima are shown in Figure 17.

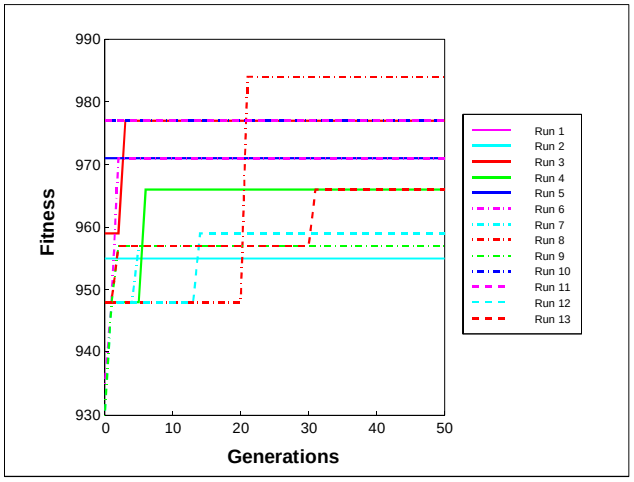
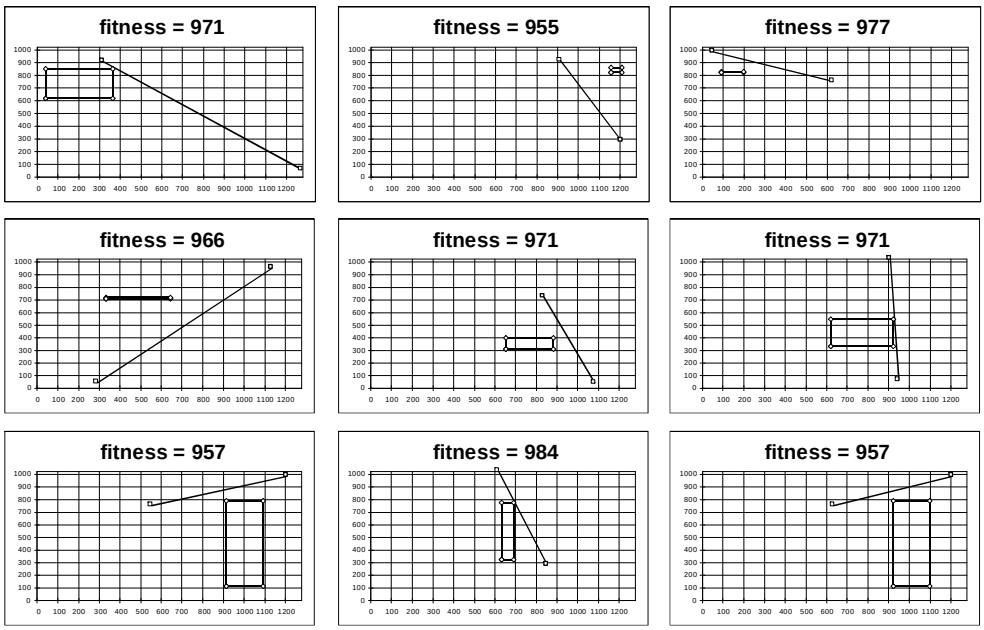


Figure 16. Fitness Evolution Curves for the Clusters

5.3 Conclusions

The experiments with *isline* show the value of cluster analysis in temporal testing. Finding the initial global optimum required 4500 fitness evaluations. However, finding 13 local optima required only (2000 + 13 × 1000 =) 15000 fitness evaluations (see Figure 18), an average of less than 1200 fitness evaluations per local optimum. The local optima provide important information about the behavior of the fitness function over the search space. Several of the optima have the same fitness value, despite very different input parameters. However, by graphically examining the input it can be seen that the configuration of the rectangle and the line are the same for these local optima. This indicates that the configuration of the input parameters, not the actual values, determine the performance of *isline*, a conclusion also suggested by the analysis of the *isline* algorithm alone.



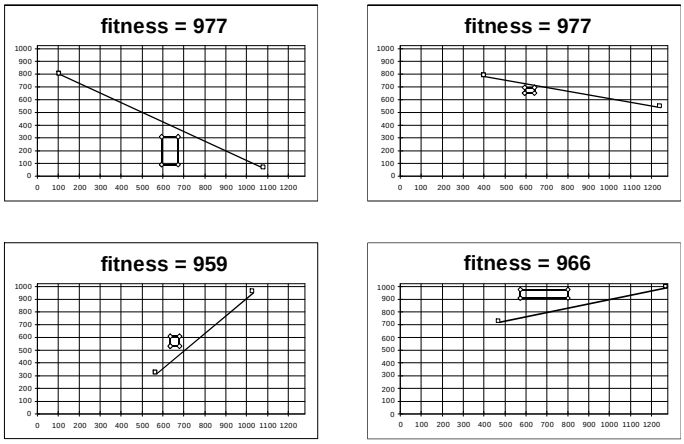


Figure 17. Local Optima (Configurations) for *isline*

	"Global" Optimum	Local Optima
# Individuals:	20	100
# Generations:	225	20
Fitness Evaluations:	4500	15000
# Optima:	1	13
Fitness:	984	971, 955, 977, 966, 971, 971, 957, 984, 957, 977, 977, 959, 966.

Figure 18. Global versus Local Optima for *isline*

6 Conclusion and Future Work

The correct functioning of real-time systems depends critically on their temporal correctness. Testing is the most important analytical method for the quality assurance of such systems. An investigation of existing testing approaches showed a lack of support for testing the temporal behavior. Therefore, existing test procedures must be supplemented by new methods and tools. In various experiments evolutionary testing has been successfully applied to search the worst-case and best-case execution times of real-time programs in order to check whether they violate their specified timing constraints.

Further improvements are possible through the combination of evolutionary testing with cluster analysis. First of all, cluster analysis provides additional information about local optima in order to detect several performance leaks within one test run. It gives information about the temporal structure of the entire search space. This is a good basis for optimizing the program code of real-time systems. Cluster analysis also aids in forming a reliable termination criterion. The clusters provide detailed information about the convergence

state of the evolutionary test. A test that has converged to one or more optima should be terminated because the probability of finding even better solutions is small.

Evolutionary testing shows considerable promise in testing and validating the temporal correctness of real-time systems and further research work in this area should prove fruitful. More work is needed to find the most appropriate parameters for the underlying genetic algorithms. Complementary to cluster analysis the degree of coverage (e.g. branch coverage) achieved during evolutionary testing and the observation of the program paths executed could be an interesting aspect for deciding when to stop the test. Additionally, further studies should focus on the question how cluster analysis information can be used to react to stagnations with appropriate changes of the search strategy.

References

Boden, E.B., and Martino, G.F. (1996). *Testing Software Using Order-Based Genetic Algorithms*. Proceedings of Genetic Programming '96, pp. 461 - 466, July 1996, Stanford University, USA.

Goldberg, D.E. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, USA.

Jones, B.F., Eyres, D.E., and Sthamer, H.-H. (1998). *A Strategy for using Genetic Algorithms to Automate Branch and Fault-based Testing.* The Computer Journal. Vol. 41, No. 2, pp. 98 - 107.

Jones, B.F., Sthamer, H.-H., Yang, X., Eyres, D.E. (1995). *The Automatic Generation of Software Test Data Sets using Adaptive Search Techniques.* Proceedings of Software Quality Management '95, pp. 435 - 444 Seville, Spain.

O'Sullivan, M., Vössner, S., and Wegener, J. (1998). *Testing Temporal Correctness of Real-Time Systems - A New Approach Using Genetic Algorithms and Cluster Analysis*. in 6th. European International Conference on Software Testing, Analysis & Review - EuroSTAR'98, Munich, Germany.

Rational Software. (1998). *Quantify 3.1*. Rational Software, Lexington MA, USA.

Roper, M. (1997). *Computer Aided Software Testing using Genetic Algorithms*. Proceedings of Quality Week '97, 27 - 30 May 1997, San Francisco, USA.

Schultz, A.C., Grefenstette, J.J., and De Jong, K.A. (1993). *Test and Evaluation by Genetic Algorithms.* IEEE Expert. Vol. 8, No. 5, pp. 9 - 14.

Sthamer, H.-H. (1996). *The Automatic Generation of Software Test Data Using Genetic Algorithms*. PhD Thesis, Department of Electronics and Information Technology, University of Glamorgan, Wales, UK.

Vössner, S. and Braunstingl, R. (1998). *G.O.A.L (Genetic Optimization Algorithm).* Genetic Optimization Lab, Stanford University, USA.

Vössner, S. and O'Sullivan, M. (1998). *CLUSTER (Cluster Analysis Package for Genetic Algorithms).* Genetic Optimization Lab, Stanford University, CA, USA.

Wegener, J., Grochtmann, M., and Jones, B. (1997). *Testing Temporal Correctness of Real-Time Systems by Means of Genetic Algorithms.* Proceedings of Quality Week '97, 27 - 30 May 1997, San Francisco, USA.

In-Process Validation and Improvement Of Test-Case Effectiveness

Yuri Chernak, consultant

e-mail: ychernak @idt.net

© Valley Forge Consulting, Inc., 1999

12th INTERNATIONAL SOFTWARE QUALITY WEEK (QW99)

24-28 May 1999, San Jose, Silicon Valley, California USA

Presentation Outline

- Why evaluate test cases?
- Test suite evaluation tasks (Verification &Validation)
- Overview of related sources
- A new metric – Test-Case Effectiveness (TCE)
- An approach to Test-Case Effectiveness improvement
- The steps of the improvement framework
- Case study
- Conclusion

© Valley Forge Consulting, Inc., 1999

Why Evaluate Test Cases?

- Test-case specifications are the main deliverable of the formal test process
- Written test cases benefit both – project management and testers
- The quality of test cases determines testers'productivity in finding defects
- Design of test cases requires project resources and testers'skills
- Test suite evaluation can help us ensure the quality of this deliverable before it is used in testing
- Evaluation = Verification & Validation (IEEE Std.1012)

A project team needs to ensure that the test suite used in testing was sufficiently effective

©Valley Forge Consulting, Inc., 1999

A Test Suite Verification Task

- The objective of verification is to ensure that test-case specifications satisfy the imposed requirements
- Verification of test cases should be performed at the end of the test design phase
- Common verification techniques:
  - Reviews or inspections of test-case specifications
  - Traceability Matrix – can be used to ensure test suite completeness, i.e., that all business rules are covered by test cases

Verification alone cannot guarantee that test cases will be sufficiently effective

©Valley Forge Consulting, Inc., 1999

A Test Suite Validation Task

- The objective of validation is to conclude whether a test suite was sufficiently effective in finding defects
- Validation of test cases should be performed at the end of the test execution cycle (in-process validation)
- If TCE is not sufficient, a test suite should be revised and improved and then executed again. As a result, testers should find more defects

In-process validation provides feedback on the test process before a system is released into production

© Valley Forge Consulting, Inc., 1999

The ODC Concept

- Orthogonal Defect Classification (ODC), developed at IBM, was the first in- process validation technique
- ODC uses a defect model as a process signature (baseline vs. actual). The major components of the defect model are ***defect type*** and ***defect trigger***
- The defect type signature is used to analyze the development process. The defect trigger signature is used to validate the effectiveness of testing
- ODC requires a stable and repeatable project profile (development process, project team, etc.) in order to build and re-use a baseline process signature
- ODC may not be reliable when the defect injection factor varies from project to project

Other validation techniques should be developed in order to provide testers with choices which best apply to their projects

© Valley Forge Consulting, Inc., 1999

The Test-Case Effectiveness (TCE) Metric

- In order to perform validation objectively, we need a metric to measure test-case effectiveness
- Our assumption is – ***the more defects test cases find, the more effective they are***. However, when we are testing, a number of defects are always missed by test cases and found as a side effect
- The proposed metric – ***TCE is the ratio of defects found by test cases to the total number of defects reported by testers during the test cycle***
- In our experience, a TCE metric of 75% is an acceptable level for business applications

Test-Case Effectiveness Metric:
TCE = Ntc / Ntot * 100%

©Valley Forge Consulting, Inc., 1999

TCE Metric vs. DRE Metric

- What they have in common –
  - Both metrics are used for validation
  - Both metrics can have various baseline values depending on the application type
- What is different –
  - TCE validates a test suite from the current test cycle perspective and can benefit the same project
  - DRE validates an entire test process from the production perspective and can benefit future projects

TCE – is used for in-process validation
DRE – is used for post-release validation

©Valley Forge Consulting, Inc., 1999

IBM's Defect Prevention Process

- Is a technique based on causal analysis of defects
- Can be used to improve both the development and test processes
- Allows test process improvement based on the causal analysis of test escapes

Definition
"*Test Escapes* – are software product defects that a particular test failed to find but which were found in a later test or by a customer" [R.G. Mays, et.al.]

© Valley Forge Consulting, Inc., 1999

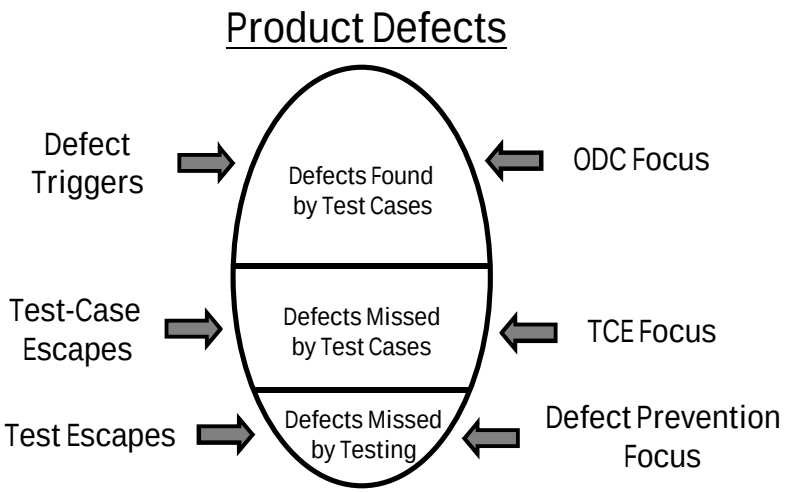
Our Approach to TCE Improvement

- Defects missed by test cases are a manifestation of a deficiency in the test process
- Such defects are an important source of information that can be used to improve the effectiveness of test cases

Definition
"*Test-Case Escapes* – are software defects that a given suite of test cases failed to find but which were found as a side effect in the same test cycle" [Y. Chernak]

© Valley Forge Consulting, Inc., 1999

Test-Case Escapes vs. Test Escapes



©Valley Forge Consulting, Inc., 1999

The Steps of the TCE Improvement Framework

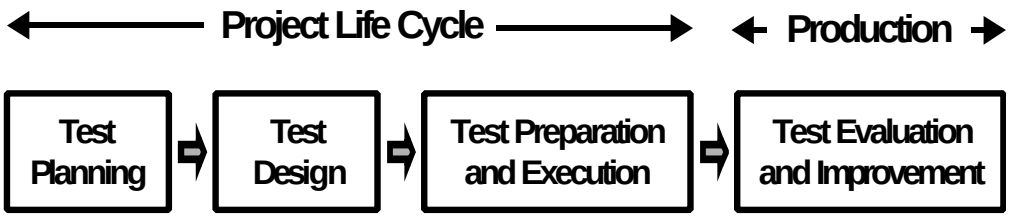
Step 1. Understand and document the test process
Step 2. Make assumptions about the factors affecting TCE
Step 3. Gather defect data and perform causal analysis of test-case escapes
Step 4. Identify the main factors
Step 5. Implement corrective actions

Causal analysis of test-case escapes is a key activity in the improvement framework

©Valley Forge Consulting, Inc., 1999

Step 1: Understand and Document the Test Process

A common test process definition:

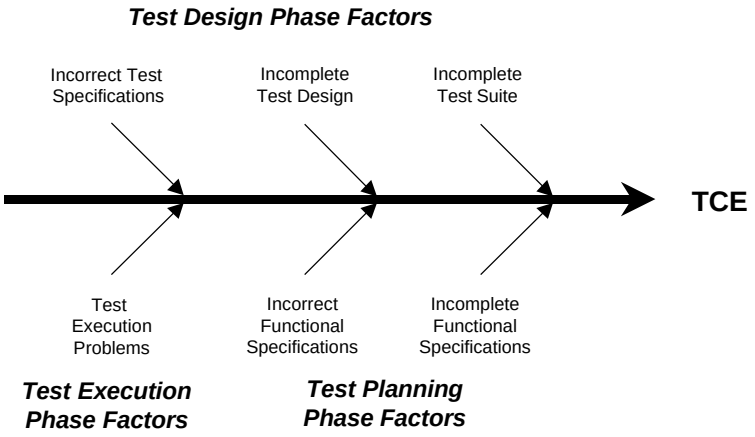


A number of factors in all but the last of these phases could determine the TCE in a test cycle

© Valley Forge Consulting, Inc., 1999

Step 2: Make Assumptions

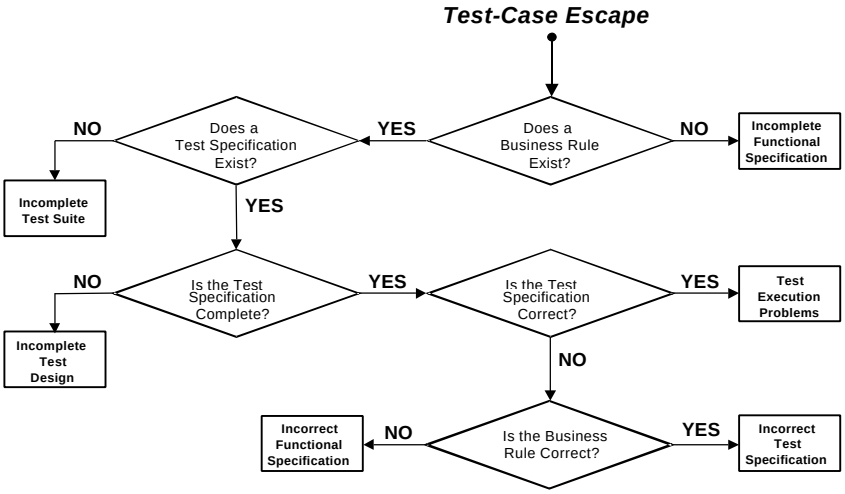
The factors are grouped by test process phases where they originate



The factors of the Test Planning and Test Design phases are common causes of insufficient test-case effectiveness

© Valley Forge Consulting, Inc., 1999

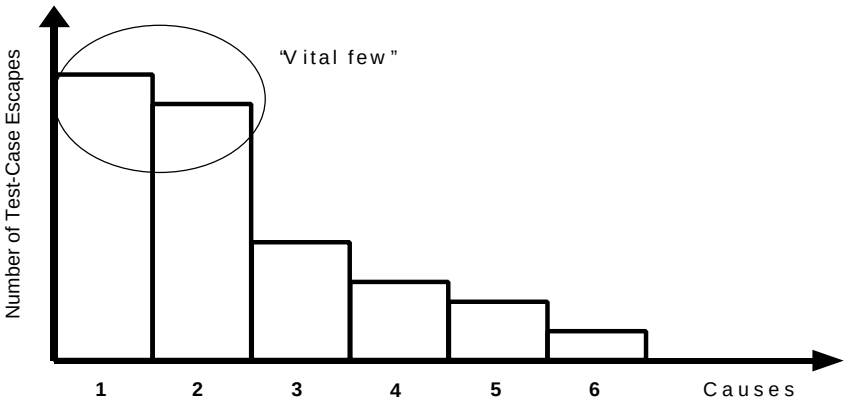
Step 3: Gather Defect Data and Perform Causal Analysis



The purpose of causal analysis of defects is to understand why a test suite missed each test-case escape

©Valley Forge Consulting, Inc., 1999

Step 4: Identify the Main Factors



The purpose of this step is to identify those "vital few" factors that are responsible for the majority of defects missed by test cases

©Valley Forge Consulting, Inc., 1999

Step 5: Implement Corrective Actions

Factors Affecting TCE	Examples of Corrective Actions
Incomplete or incorrect functional specifications	Inspections and re-work of functional specifications
Incomplete test suite	Use a traceability matrix to ensure test suite completeness
Incomplete test design	Training of testers, use of checklists or templates, etc. Revision of specifications
Incorrect test-case specifications	Inspections and re-work of test-case specifications
Test execution problems	Training of testers, use of a test execution procedure, etc.

After a correction, testers should re-test a system and find more defects

©Valley Forge Consulting, Inc., 1999

Case Study: The Project Description

- A banking application for external clients
- A Client/Server architecture
- Front-end – Windows NT, VB, VC++
- Back-end – Unix/Sybase, Mainframe/DB2
- A project team – 10 developers, 3 testers
- PVCS-based defect tracking system

The project team implemented a formal test process with its focus on functional testing

©Valley Forge Consulting, Inc., 1999

Case Study: Evaluation of Test-Case Effectiveness

At the end of the test execution cycle:

- 112 defects were found by test cases
- 71 defects were found as a side effect (test-case escapes)
- The actual TCE was 61% vs. the acceptable level – 75%

The project team decided to perform the TCE improvement task

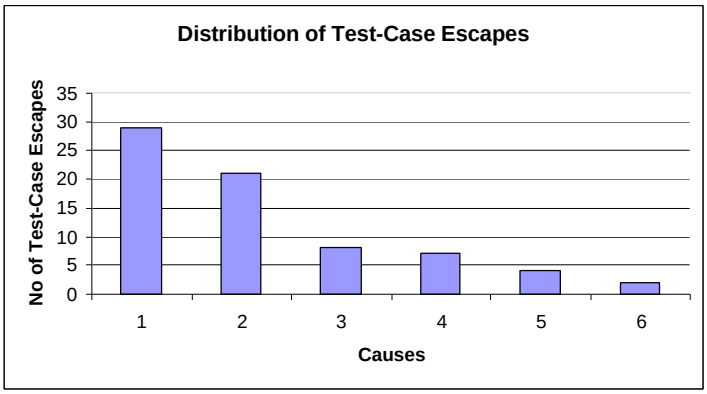
©Valley Forge Consulting, Inc., 1999

12th INTERNATIONAL SOFTWARE QUALITY WEEK (QW99)
24-28 May 1999, San Jose, Silicon Valley, California USA

Case Study: Causal Analysis of Test-Case Escapes

The project team analyzed the causes of the test -case escapes

A Pareto chart was used to identify the main factors



The main two factors were:

- **Incomplete test design**
- **Incomplete functional specifications**

©Valley Forge Consulting, Inc., 1999

Case Study: Process Correction and System Re-testing

- The project team reviewed, corrected, and completed functional specifications
- Testers created additional test-case specifications to completely cover business rules
- Testers revised existing specifications
- The system was re-tested using the revised test suite
- Testers found and reported 48 additional defects
- During the first two months in production users reported 23 defects, none of them critical

After two months in production:
DRE = 91%

©Valley Forge Consulting, Inc., 1999

Conclusion

- For complex systems, test cases can be a critical success factor for effective testing
- If test cases are not sufficiently effective, there is a high risk that testing will not be effective either, hence, in-process validation of TCE is an important task
- In the presentation we proposed a new metric for in-process TCE validation and a framework for TCE improvement
- The framework consists of five steps and is based on causal analysis of test-case escapes
- A case study illustrated how the proposed approach can be implemented in software projects

©Valley Forge Consulting, Inc., 1999

In-Process Validation and Improvement of Test-Case Effectiveness

Yuri Chernak, consultant
Valley Forge Consulting, Inc.
e-mail: ychernak@idt.net

1. INTRODUCTION

Why evaluate test cases?

In critical software projects, management must focus on reducing the risk of releasing a system with quality not acceptable to users. This risk can be managed through software testing, which is used to find and remove software defects prior to production. Formal test methodology defines a number of test types, including the function test that is most responsible for the ultimate system quality. The function test, as defined by GMyers [1], focuses on a system's functionality and is performed by testers with the objective of finding defects. Implementation of the function test as a formal test process allows us to better cope with the functional complexity of the software applications being tested. In a formal test process, test plan documents and test-case specifications are important deliverables [2]. The use of test cases benefits both project management and testers:

- *Management benefits*. Test-case specifications allow better visibility into the test process by providing a precise picture of what will be tested and how. In addition, test-case specifications allow management to monitor test execution progress and to measure testing completeness.

- *Testers' benefits* . Design and execution of test cases result in higher testers' productivity in finding defects.

For complex systems, test cases become a critical success factor for accomplishing effective testing. Commonly, test-case specification design is the major effort of testers on projects that implement formal testing. In our experience, the design of test cases takes considerably more effort than their execution. In order to justify the design effort and to ensure the quality of this deliverable, test-case specifications should be the object of in-process evaluation and improvement, if needed. Besides the immediate benefits for the current project, such evaluation can help testers improve their performance in future projects as well. Indeed, testers reuse their test-case design skills from project to project. Hence, the evaluation of this deliverable can help them identify the weaker areas of their skills that, being improved, will result in better test design in future projects.

On the other hand, the fact that testers used test-case specifications in testing does not necessarily guarantee that the system was sufficiently tested. A number of other factors also determine whether testers performed well and testing was effective. In general, actual testing effectiveness can be evaluated only when a system is in production. Such evaluation can certainly benefit future projects. However, if it shows that a system was

not sufficiently tested, this result is too late to benefit the current project. To reduce such a risk, the project team can assess testing effectiveness in the current project by performing in-process evaluation of test-case effectiveness. If the test cases were not sufficiently effective, a high risk exists that the system was not sufficiently tested either. At this point the system has not yet been released into production; therefore, it is not too late for the project team to identify the reasons and to correct the test process.

Evaluation = Verification + Validation

In general, the purpose of evaluation is to ascertain that the work product fits the use. In the case of software project deliverables and test cases in particular, evaluation can be performed as a combination of verification and validation tasks [3]. According to the IEEE dictionary of software terms [4], these terms have the following definitions:

▪ *Verification* is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase	▪ *Validation* is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements

Verification Task. Verification of test-case specifications should be done at the end of the test-design phase. The objective here is to ensure that this deliverable satisfies the imposed requirements before testers begin to use these documents in the next, test execution, phase. Reviews or inspections are common verification activities that can be used to evaluate test-case specifications for their correctness and completeness, compliance with conventions, templates, or standards, etc. Traceability analysis is also a common way to verify the whole test suite for completeness. By building atraceability matrix or tree we can trace from the functional specifications to the corresponding test-case specifications and ensure that all functional requirements are covered by test cases. Verification of test cases before test execution is an important task that allows the project team to assess the conformance of test-case specifications to their respective requirements. However, such conformance does not mean that the test cases will automatically be effective in finding defects. Other factors also determine whether test cases will be effective in the test cycle. For example, functional specifications, used to design test cases, can be incomplete or outdated; test cases can be written based on poor test design logic; testing personnel can misunderstand test specifications, etc. Hence, even test cases that have passed verification could have weak failure-detecting ability and should be required to pass validation as well.

Validation Task. As we discussed, test cases are a critical success factor for achieving testing effectiveness in critical software projects. The main requirement for test cases is that they should be effective in finding defects. We assume if a test suite was productive in finding defects, the system under test will likely be of high quality when released into production and that it will satisfy the users. On the other hand, if a test suite was not productive, there is a high risk that the system was not sufficiently tested and the users will not be satisfied with the system quality. Hence, the main objective of test suite validation should be to conclude whether test cases were sufficiently effective in finding

defects. Validation can be performed as soon as all test cases have been executed, which is the end of the test-execution phase of the test process. At this point the system has not yet been released into production, and if test-case effectiveness has not proved satisfactory, it is not too late to analyze the reasons and to correct the test process. As a result, a test suite can be revised and improved, and then executed again. This, in turn, can help testers find additional defects and release a better software product. As it will be illustrated by the case study in Section 5, such in-process test-case validation can be very effective in providing management and testers with feedback on the test process during the project life cycle.

This article describes how functional test cases can be in process validated by assessing their effectiveness in finding defects. It also presents a framework, which can be used to identify weak areas in testing and improve test-case effectiveness. In Section 2 we review some related sources that discuss issues pertinent to the evaluation and improvement of test-case effectiveness. In Section 3 we propose a new metric – Test-Case Effectiveness – that can be used for in-process test suite validation. In Section 4 we describe a framework for improving test-case effectiveness based on defect causal analysis. Section 5 provides the results of the case study and illustrates how the new metric and improvement framework can be applied in software projects. The paper concludes with Section 6.

2. RELATED WORK

Orthogonal Defect Classification

Our research was inspired by the publications of RamChillarege and his colleagues from the IBM T.J.Watson Research Center [5,6]. They discussed the need for in-process evaluation and improvement of testing, especially important for critical projects as a means of avoiding the risk of delivering a software product of poor quality. In particular, their focus was on validating test suite effectiveness at the end of the test execution cycle and deciding whether a product was sufficiently tested before releasing it into production. They proposed Orthogonal Defect Classification (ODC) – a concept for in-process measurement and feedback to a project team on the development process. The underlying ideas of this concept can be summarized as follows:

- Capturing and analyzing software defect semantics can help us understand and improve the weak areas of both a product and a process;

- Building a defect model as a process signature and using it as a baseline for future projects can help us identify deviations from the norm where improvement is needed;

- The major components of a defect model are *defect type* and *defect trigger*. The defect type specifies the actual fix of a given defect, and the defect trigger identifies the event or condition that detected such a defect. The defect type signature indicates the product's stability relative to its progress through the development process. The defect trigger signature helps us validate the effectiveness of the testing activities in uncovering defects;

- A system is qualified not sufficiently tested, if some triggers in the actual trigger distribution of a given project have lower values than in the baseline. In this case the test suite should be improved by creating additional test cases that target defects with the triggers that caused a concern. Then testers should re-test the system and try to find more defects.

ODC has evolved significantly since its inception and has been implemented by various teams at IBM. The key to success with ODC is in a stable software process that has a baseline signature repeatable from project to project. Under these conditions, ODC has proven to be a valuable technique that allows project teams to reduce development cost and to increase customer satisfaction [7].

According to ODC, test suite effectiveness can be evaluated based on the analysis of defects found in a test cycle. By building an actual defect trigger distribution and comparing it with the baseline, i.e. intended signature, the project team can identify the weakness of a test suite and determine whether it needs improvement. In our opinion, this approach may not be reliable under the following conditions. Trigger values in both signatures, intended and actual, are determined by at least two factors – defect injection and defect detection. If defects with some triggers were not injected in a given project, these triggers will not be present in the actual trigger distribution, unlike the intended signature, even if the test suite is perfect. In this example, the intended signature could possibly mislead a project team. As a result, testers would waste resources performing test suite improvement, re-testing a system, and trying to find defects that are simply not present in the product. This illustrates our point that defect triggers may not directly point to the test suite deficiencies. Apparently, the test process can be effectively evaluated by comparing intended versus actual trigger distributions only if the defect injection factor remains the same, which is difficult to assess in a real life situation. In general, the development process and developers' skills mainly determine the defect injection characteristics. In other words, even if the software process is stable, some "human" factors can significantly impact and invalidate the "historical" process signature:

- The first factor that can invalidate a signature is the fact that developers always tend to learn from their mistakes and reduce frequent defects. This could result from the individual effort of developers or from a planned defect prevention and process improvement task performed by the project team;

- Another factor is personnel turnover. Hiring consultants for software projects has become a common practice in various fields such as financial, brokerage, insurance, etc. This frequently makes project teams very different from one project to the next, thus, affecting the process signature.

Based on this discussion we conclude that a defect trigger distribution may not always provide a reliable basis for test-case effectiveness validation. Therefore, other techniques should be developed and further evaluated in order to provide testers with choices best applicable to their projects.

Defect Prevention Concept

The defect prevention concept [8], also originated at IBM, suggests improving the test process based on a causal analysis of defects, so-called test escapes, which were missed in testing. We can further evolve this idea and state that the analysis of defects missed by test cases can help us improve their effectiveness. This concept is essentially different from ODC, which, in contrast, focuses on defects found by test cases. Based on our experience with client/server projects, even if a project team implements an effective test process and testers have a good test suite, a number of defects are typically found as a side effect, i.e., by exercising conditions not specified by test cases. These defects can be considered a manifestation of deficiencies in the *formal* test process. Therefore, their causal analysis can help us identify the areas for test process improvement that can be implemented before the system is released into production. However, this approach is entirely based on the analysis of defects missed by test cases and requires that a sufficient number of such defects be available. This fact can limit the approach applicability for some projects. For example, testing mainframe batch systems, testers usually exercise only those conditions that are planned in advance, and the number of defects found as a side effect is usually very low in the test cycle. On the other hand, in the case of client/server projects that implement formal testing, the share of such defects could be from 20 to 50 percent, providing a valuable source of information for test suite validation and improvement. In the following sections we discuss this approach in detail. We propose a new metric that can be used for validation of test cases and we describe a framework for in-process improvement of test suite effectiveness.

3. THE TEST-CASE EFFECTIVENESS METRIC

The function test is destructive; its purpose is to break systems and find defects. When a test process is formal, testing is executed based on written test-case specifications. Hence, an important requirement for test cases is that they should be effective in finding defects when executed by testers. In Section 1 we discussed the benefits of in-process validation of test-case effectiveness. In order to perform validation objectively, we need a metric to measure the test-case effectiveness that is the key point in the validation task. When the result of the effectiveness assessment does not meet expectations, the reasons should be analyzed and corrected and test cases should be executed again.

When testing on-line mainframe systems and especially client/server systems, a certain number of defects are always found as a side effect. By "side effect" we mean the situation where a tester finds a defect by executing some steps or conditions that are not written in a test-case specification. This can happen either accidentally or, more frequently, when the tester gets the idea "on the fly". Defining a metric for test-case effectiveness, we can state that the more test cases find defects, the more effective they are. On the other hand, if functional test cases result in finding a small number of defects, their value comes into question. Based on this logic, we propose the metric – **Test-Case Effectiveness** (TCE) – that is defined as the ratio of defects found by test cases (Ntc) to the total number of defects (Ntot) reported by testers during the function test cycle:

$$TCE = Ntc / Ntot * 100\%$$

In this definition, the total number of defects (Ntot) reported by testers is the sum of defects found by executing test cases and defects found as a side effect. The TCE metric may look similar to the **Defect Removal Efficiency** metric (DRE) [9], which is defined as the ratio of defects found prior to production to the total number of reported defects. However, the important difference is that the DRE metric has the purpose of evaluating user satisfaction with the entire test process. It measures the test process effectiveness and reflects a production or users' perspective on the test process. In contrast, the proposed TCE metric is used specifically to validate effectiveness of functional test cases, which are just a component of the test process. The TCE metric evaluates test cases from the test cycle perspective and provides in-process feedback to the project team on how well a test suite worked for testers. If a project goes through a number of incremental builds, as is common in large-scale system development, each build requires its own functional test cases and has its own test cycle. In this situation, it is beneficial to perform test suite validation based on the TCE metric for each build. If the result of the validation is not satisfactory, we can identify the areas for improvement and correct the test process in the current or next builds.

As we discussed above, the main purpose of validation is to determine whether a test suite was sufficiently effective in the test cycle. Such a conclusion can be made by comparing the actual TCE value, calculated for the given test cycle, with the baseline value, selected in advance by the project team. When the actual TCE value is at the baseline level or above, we can conclude that the test cases have been sufficiently effective in a test cycle. In this case the project team can anticipate user satisfaction with the system quality in production. On the other hand, the more that TCE value is below the acceptable level, the higher the risk that the users will not be satisfied with the software product quality. In this case the project team can implement a correction of the test process based on the framework described in the next section. The baseline value can be obtained by analyzing the experience of the previous successful projects that are qualified as models for current and future projects. Apparently, the goal for test-case effectiveness can be different for various application categories, such as commercial, military, or business applications. Our experience with successful client/server projects delivering business applications shows 75% to be an acceptable level of the TCE metric.

4. A FRAMEWORK FOR TEST-CASE EFFECTIVENESS IMPROVEMENT

If the result of validation of test-case effectiveness in the test cycle does not meet the requirement, the project team should analyze the causes and identify the areas for test process improvement. The improvement framework proposed in this paper and described below stems from the work [8] that presents the approach to defect prevention developed at IBM. In this work the authors reported that the defect prevention concept could be successfully applied to software development and testing as well. The purpose of testing is to find defects during the test cycle and to deliver a quality system acceptable for the users. However, a number of defects are always missed in testing and reported later by

users during production. The causal analysis of these defects, so-called *test escapes* [8], is a key activity in test process improvement. In [8] test escapes are defined as "product defects that a particular test failed to find but which were found in a later test or by a customer [in production]".

In this article we focus on the evaluation of one very important component of the test process – test cases – and apply the concept of test escapes to in-process test-case effectiveness validation. The main purpose of documented test cases is to assist testers in finding defects. Hence, software defects missed by test cases and found by testers as a side effect can be considered a manifestation of deficiencies of the test process and a given test suite in particular. Such defects are an important source of information, and their causal analysis can help us understand and identify the main factors that reduce the effectiveness of test cases. Based on this discussion, we can give the following definition of "test-case escapes" that reflects the test-case perspective:

> ***Test-Case Escapes*** are software defects that a given suite of test cases failed to find but which were found as a side effect in the same test cycle.

Analysis of test-case escapes can help us improve effectiveness of white box test cases during unit testing and black box test cases during functional testing. In addition, the same concept can be used to improve the formal inspection process as well. However, in this article we are focusing on black box testing only. The in-process improvement framework that we propose below is based on the causal analysis of test-case escapes and consists of the following steps –

1. Understand and document the test process used by the project team,
2. Make assumptions about the factors affecting test-case effectiveness,
3. Gather defect data and perform causal analysis of test-case escapes,
4. Identify the main factors, and
5. Implement corrective actions

After these steps, the test suite should be executed again. As a result, testers should find additional defects that justify the improvement effort. Below we discuss each of these steps in detail.

<u>Step 1. Understand and Document the Test Process</u>

When a project team uses written test-case specifications and focuses on their evaluation and improvement, this already indicates that a certain test process has been established and followed in the project. The test process should be planned at the beginning of the software project and documented in a test plan. Commonly, the test process is defined in terms of the following phases – 1. Test Planning, 2. Test Design, 3. Test Preparation and Execution, and 4. Test Evaluation and Improvement [10,11]. Each of these phases should be planned and defined in terms of tasks and deliverables. For example, the test process can be defined as follows:

- *Test Planning*. In this phase, definition of the scope, objectives, and approach to testing is the main task and a test plan document is the main deliverable;

- *Test Design*. In this phase, design of test cases is the main task and test-case specifications are the main deliverable;

- *Test Preparation and Execution*. In this phase, preparation of the test environment, executing test cases, and finding defects are necessary tasks and defect reports are the main deliverable;

- *Test Evaluation and Improvement*. In this phase, analyzing results of testing is the main task and the test summary report is the main deliverable.

In all these phases, except for the last one, there are a number of factors that determine the effectiveness of functional test cases in a given project. Hence, the purpose of the following steps of the framework is identification and evaluation of these factors.

Step 2. Make Assumptions

When the test process is understood and documented, we need to analyze each phase and identify factors that can affect test-case effectiveness.

Test Planning. The main deliverable of the Test Planning phase is a test plan document that, among other things, defines the scope and objectives of testing. The objectives of testing can be defined as "features to be tested" [2] that, in turn, should be identified in functional specifications. If the functional specifications do not completely define functional features, the test plan document will not be complete either. Hence, the test cases will not cover completely a system's functionality, thereby reducing their effectiveness.

Test Design. The Test Design phase, in turn, can have other factors that can affect test-case effectiveness. First, the test suite could be incomplete and some of the business rules in the functional specifications could be left not covered by test cases. Secondly, test design logic could be incomplete and some of the necessary test conditions could be missing in test-case specifications. Writing a test-case specification, we usually begin with understanding and analyzing the corresponding business rule that is the object of this test. Then we think about the test logic required for testing this functional feature. In order to identify necessary test cases, we can use test design techniques such as decision tables, equivalence partitioning, boundary analysis, etc. [10]. If the test design logic is not complete, some of the necessary test cases could be missing in the test-case specifications. A common example of this situation is the lack of negative test cases in the test specifications. We qualify a test case negative if it exercises abnormal conditions by using either invalid data input or the wrong user action. In addition, test-case specifications could be incorrect. This can happen either if a source document, i.e., a corresponding functional specification, is not correct or clear, or an error is made in the test-case specification itself.

All these deficiencies identified in the Test Planning and Test Design phases of the test process will ultimately require revision of test-case specifications and their re-testing.

Test Preparation and Execution. In addition, the Test Execution phase could be a source of other factors reducing test-case effectiveness. For example, some of the test cases can be left not executed or can be executed incorrectly. Also, a tester could overlook defects especially when the verification of expected results is not straightforward. However, another tester could later find such defects as a side effect. Based on our experience, only a small fraction of test-case escapes stems from the test execution factors. Therefore, these factors will not likely be in the focus of the test-case effectiveness improvement effort. We can initially qualify the causes of such defects for one common category "test execution problems". However, if the further analysis, described below at Step 4, shows that the fraction of defects in this category is significant, a detailed evaluation of test execution factors should be performed.

In general, the fact that testers find test-case escapes in addition to defects found by executing test cases should be considered a plus, since it results in more defects being removed from the software product before production. On the other hand, test-case escapes, being found "by chance", manifest the weakness of the formal test process. If the share of test-case escapes is higher than expected, it should raise a management concern about how many other defects are still hidden in the product. In this case a correction of the test process and system re-testing should be performed. If additional defects are found, this can increase management's confidence in the software product quality.

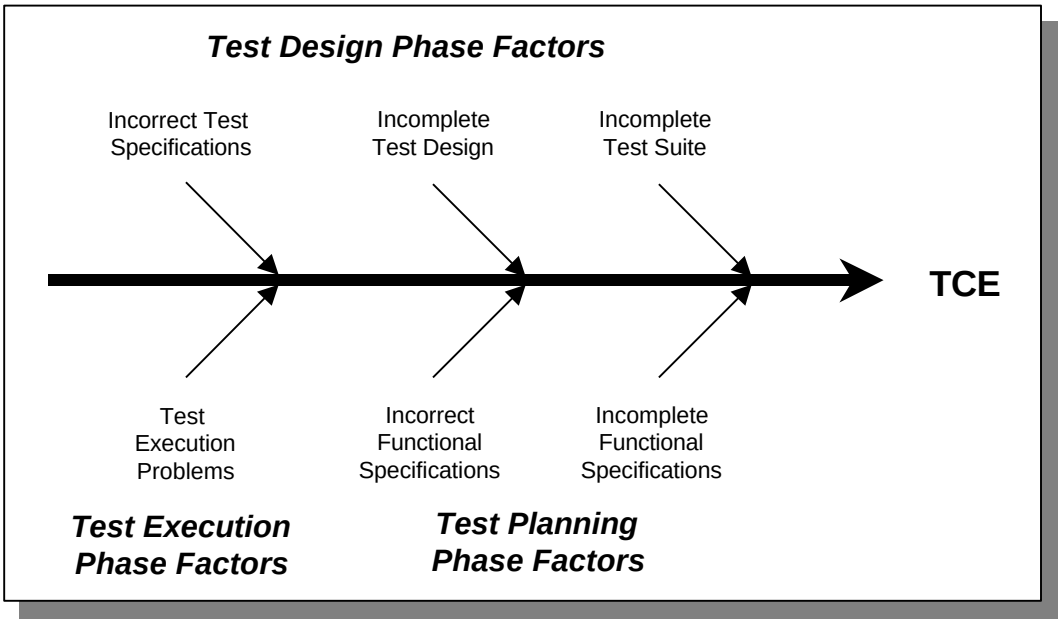


Figure 1. Factors Affecting Test-Case Effectiveness

The factors identified above are shown in Figure 1 in the form of the cause-effect diagram. We grouped them by the test process phases in which they originate. However,

at this point they are just our assumptions that should be evaluated in the following steps with the objective of identifying those factors that are mostly responsible for insufficient test-case effectiveness.

Step 3. Gather Data and Perform Causal Analysis

To perform causal analysis of test-case escapes at the end of the test execution phase, we need to select the defects missed by test cases. This requires that the project team use a defect tracking system and that the testers, reporting defects, identify which defects were found as a result of test-case execution and which defects were found as a side effect. By definition, the latter defects should be qualified as test-case escapes. Once the test-case escapes are identified and selected, they should be qualified by one of the factors based on the causal analysis logic presented in Figure 2. The purpose of this analysis is to evaluate each test-case escape and understand why the test suite missed a given defect during the function test execution.

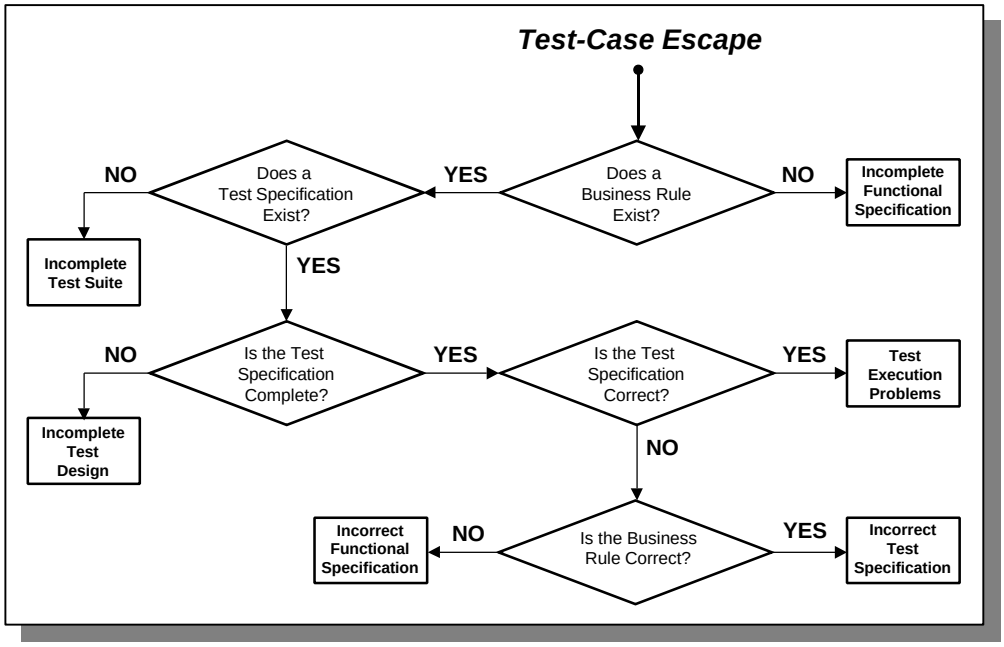


Figure 2. Test-Case Escape Classification Logic

We can begin the causal analysis with verification that a functional specification has a business rule related to the given defect. If the answer is "no", we have determined that the cause of this test-case escape is an incomplete function specification. If the answer is "yes", we need to check whether the test suite has a test specification that should have found this test-case escape. If a test-case specification does not exist, this means that the test suite does not cover all business rules. Therefore, an incomplete test suite is the reason why this defect was missed by test cases. If the test specification does exist, then we need to check the defect against test cases in the specification. If none of them were designed to catch such a defect, this indicates that the test specification is incomplete. Indeed, all test inputs and expected results in the test-case specification can be correct.

However, it can specify, for example, only positive test cases. The lack of negative test cases in test specifications is a common reason for missing defects. This is a case of deficiency in test design that was used to derive test cases. Hence, we can qualify that the cause of such test-case escapes is incomplete test design. On the other hand, if the test specification has conditions related to a given defect, we need to verify that these conditions and a corresponding expected result are correct. If the answer is "yes", we should qualify that test execution problems are the likely reason that this defect was missed when the test specification was executed. If the answer is "no", we need to understand why the test specification is not correct. First, we should check the source document and see if the business rule in not correct as well. If this is the case, we should qualify that the cause of this test-case escape is incorrect functional specification. Otherwise, the cause is incorrect test specification itself. As a result of defect causal analysis, all test-case escapes should be classified by one of the possible causes presented in Figure 1.

Step 4. Identify the Main Factors

When all test-case escapes are qualified by their respective causes, we need to identify those "vital few" factors that are responsible for the majority of the defects being missed. This can be done by building a Pareto chart (see Figure 3) and analyzing the distribution of defects. Once identified, these factors will be in the focus of the next step – implementation of corrective actions.

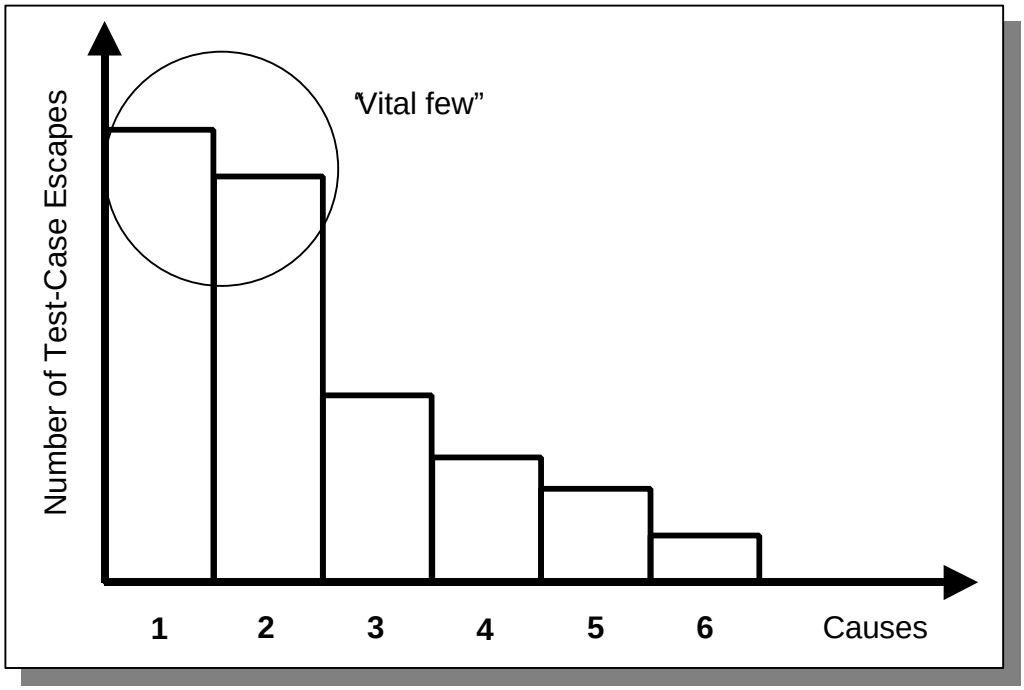


Figure 3. A Pareto Chart of Causes

Step 5. Implement Corrective Actions

After the main causes of test-case escapes have been identified, the project team should implement corrective actions and repeat the test execution cycle. For the factors, shown in Figure 1, corrective actions could be the following:

- *Incomplete or incorrect functional specifications* – inspection and rework of specifications
- *Incomplete test suite* – use of traceability matrix to ensure complete coverage of business rules by test cases
- *Incomplete test design* – training of testers on test design; use of checklists or templates to design test-case specifications
- *Incorrect test-case specifications* – inspection and rework of specifications
- *Test execution problems* – training of testers on test execution, development and use of procedures for test execution and verification of test results, etc.

In a case when correction was required for functional specifications or test cases, the project team should revise the test suite and execute it again. However, if correction was required only due to the test execution problems, the same test suite can be used for re-testing. The main objective of re-testing is finding additional defects. Finally, if additional defects are found, this fact can justify the whole improvement effort.

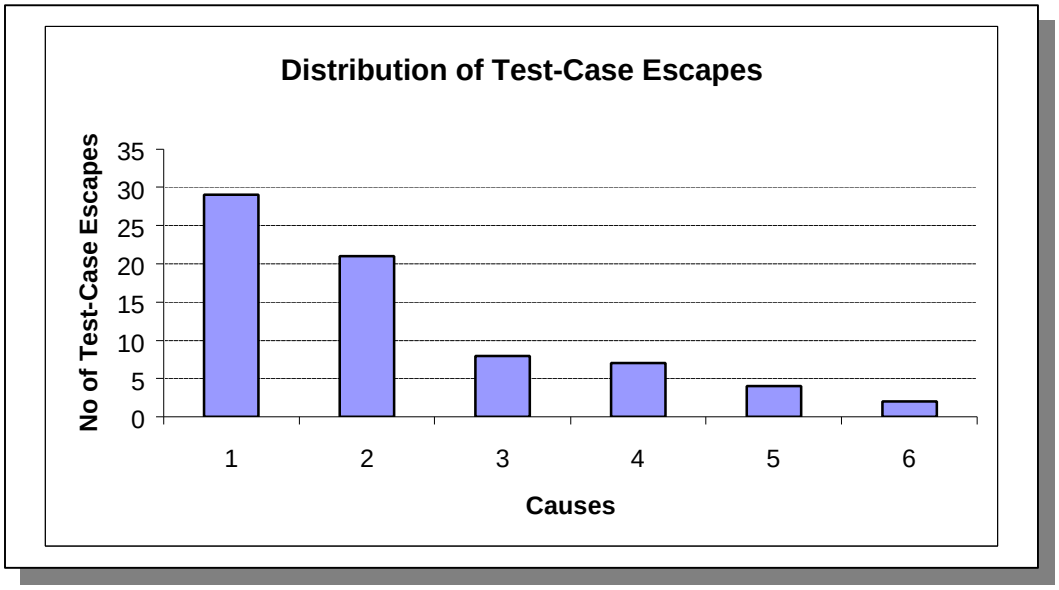
5. CASE STUDY

In this section we illustrate how the proposed approach to test-case effectiveness validation and in-process improvement was implemented in a client/server project. This project was a banking application intended for external clients – financial institutions. The system had a three-tier client/server architecture with a Windows NT front-end developed with Visual Basic and Visual C++. The second tier was implemented in a Unix environment with Oracle 7 as a database engine. The third tier was a data feed from a mainframe COBOL/DB2 system. The project team consisted of ten developers and three testers. Since the application was intended for external clients, software quality was of great importance to project management. To ensure high quality for the system, the project team implemented a formal test process with a focus on functional testing. The development team was responsible for functional specifications and the test team was responsible for the test plan document and test-case specifications.

Testing was performed in the designated test environment. By the end of the Test Execution phase, testers had executed all test specifications and reported 183 defects. Defects were managed in the PVCS-based defect tracking system. Reporting defects, testers classified them as test-case escapes or as being found by conditions in test-case specifications. In this test cycle testers reported 71 test-case escapes and 112 defects

found by test cases. Based on these numbers, the calculated TCE metric value was 61%, which was considerably lower than the acceptable level – 75%. As a result, the project team concluded that the system was not sufficiently tested and that a test process correction was needed.

The project team performed the test process improvement according to the framework described in Section 4. First, all test-case escapes were analyzed and assigned to the causes they were missed by the test cases. After this step, the distribution of causes was built (see Figure 4). The analysis of the distribution showed that incomplete test design and incomplete functional specifications were the main causes why test cases missed defects. To correct the test process, the project team began by correcting and completing the functional specifications and reviewing them with the users. Then the existing test-case specifications were reviewed and completed with negative test cases. In particular, the review of test-case specifications showed that the main deficiency of the test design was its lack of negative test cases. By definition, negative test cases focus on abnormal workflow and are intended to break a system. However, the test suite initially used by the testers was not sufficiently "destructive", and a significant number of defects were found as a side effect as opposed to being found by conditions in test cases. In addition, a number of new test-case specifications were created to completely cover the business rules in the revised function specifications. To verify test suite completeness, this time the project team used a treceability matrix. This was not done in the first test cycle, and test suite incompleteness was one of the factors that reduced test-case effectiveness (see Figure 4).



Description of Causes:
1. Incomplete Test Design
2. Incomplete Functional Specifications
3. Incomplete Test Suite
4. Incorrect Functional Specifications
5. Incorrect Test-Case Specifications
6. Test Execution Problems

Figure 4. A Pareto Chart

After these corrections, the revised test suite was executed again. As a result, the testers found 48 additional defects that otherwise would have been released into production. At this point the number of defects found during the test cycles had grown to 231. After two months in production, the rate of defects, reported by users, had noticeably declined. By the end of the second month the number of production defects was 23. The DRE metric calculated at this time was 91%, which is 231/(231+23) = 0.91. None of the defects reported by the users were of critical severity, and the users were fairly satisfied with the system quality.

6. CONCLUSION

When a project team implements a formal test process, test cases are the main factor that determines overall testing effectiveness and system quality. However, the actual quality of a system can only be evaluated when a system is in production. Such evaluation can definitely benefit future projects, although, if a system was not sufficiently tested, the evaluation results are too late to benefit the current project. The project team can reduce such a risk by performing in-process evaluation of test-case effectiveness for a given test cycle. If the test cases were not sufficiently effective, a high risk exists that the system was not sufficiently tested either. At this point the system has not yet been released into production; therefore, it is not too late for the project team to identify the reasons and to correct the test process. After a correction, testers should re-test the system and find more defects, thereby further improving the product and increasing confidence that the users will be satisfied with the system quality in production.

In this article we described an approach to in-process validation and improvement of test-case effectiveness. We proposed a new metric – Test Case Effectiveness – that can be used to validate a test suite by the end of a test cycle. In a case where the result of the validation was not as good as expected, the project team should identify and correct factors that reduced test-case effectiveness. The framework that can be used for this purpose is described in the article. It is based on the ideas of the defect prevention process [8]. We applied the concept of test escapes specifically to test-case evaluation and defined "test-case escapes" as defects missed by test cases in testing. The framework consists of five steps described in detail in the article. The key activity of this framework is casual analysis of test-case escapes and identification of those "vital few" factors that are responsible for the majority of the defects missed by test cases. Using a case study, we illustrated how the proposed approach can be applied in software projects.

REFERENCES:

1. Glenford Myers "The Art of Software Testing", John Wiley & Sons, 1979
2. IEEE Std.829 – 1983 "Software Test Documentation"
3. IEEE Std 1012-1986 "IEEE Standard for Software Verification and Validation Plans"
4. IEEE Standard 610.12-1990 "Dictionary of Software Terms"

5. R. Chillarege, I.Bhandari, J.Chaar, M. Halliday, D.Moebus, B.Ray, and M. Wong "Orthogonal Defect Classification – A Concept for In-Process Measurements", TSE, November 1992, pp.943 - 955
6. J. Chaar, M. Halliday, I.Bhandari, and R. Chillarege "In-Process Evaluation for Software Inspection and Test", TSE, November 1993, pp.1055 - 1070
7. K. Bassin, T. Kratschmer, and P. Santhanam "Evaluating Software Development Objectively", IEEE Software, November 1998, pp.66 - 74
8. R. Mays, C. Jones, G. Holloway, D. Studinski "Experiences with Defect Prevention", IBM Systems Journal, Vol.29, No.1, 1990, pp. 4 - 32
9. C. Jones "Applied Software Measurement", McGraw-Hill, Inc., 1991
10. E. Kit "Software Testing in the Real World", Addison-Wesley Publishing Co., 1995
11. P. Goglia "Testing Client/Server Applications", QED Publishing Group, 1993

Automated test generation from SDL/UML specification

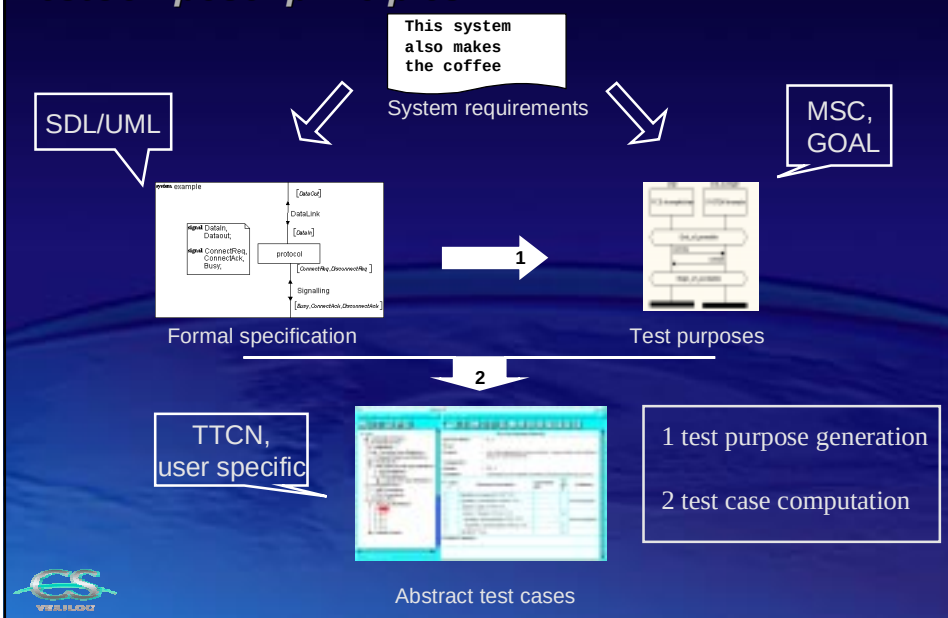
Alain Kerbrat, Iulian Ober

VERILOG

Automated test generation

- **Benefits**
  - . **correct, reproducible tests**
  - . **cost effective**
- **Requirements**
  - . **unambiguous, tractable description**
  - . **clear definition of the test semantics**

TestComposer principles



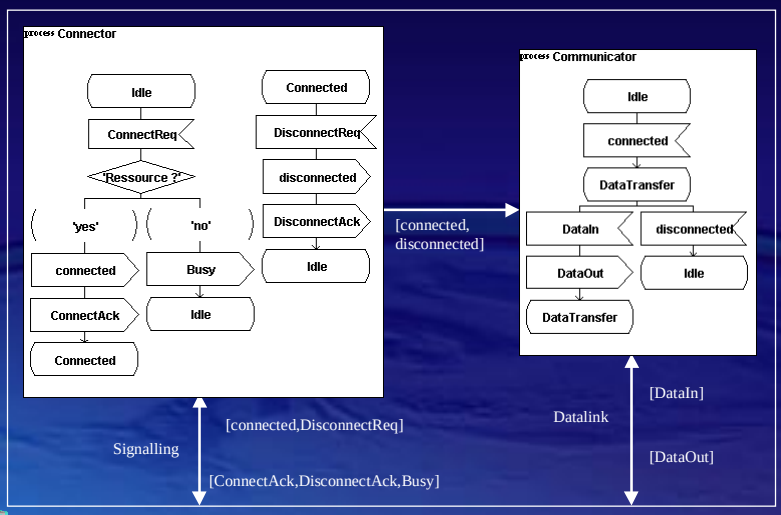
TestComposer basics

- **ISO/IEC 9646: « conformance testing »**
- **based on two research prototypes :**
  - **. Tveda from France -Telecom/CNET**
    pragmatic test purposes generation
  - **. TGV from Irisa/Verimag**
    efficient test cases generation

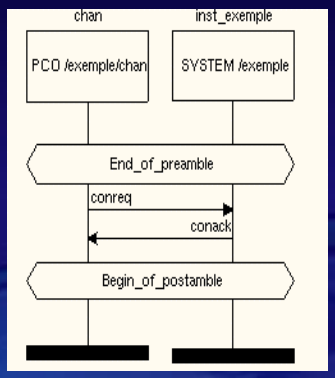
TestComposer objects

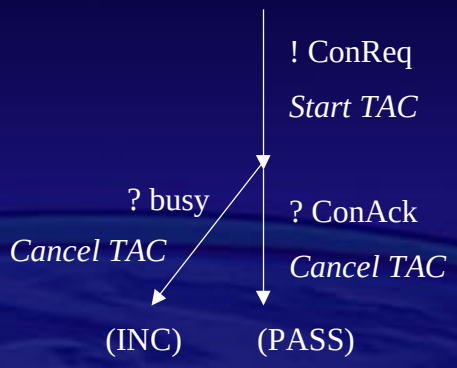
- **SDL specification :** ***expected system's behaviour***
  - ***must be validated***
- **Test environment specification**
- ***Test purpose :*** ***functional view of the test***
  - ***"What should I test ?"***
- ***Test case :*** ***operational view of the test***
  - ***"How should I test ?"***

SDL specification



From test purpose to test case





Main test sequence

A tree with all possible outcomes

Test suite production

- **Test cases database**
  - **build in a multi-session mode**
  - **Application Programming Interface**
- **TTCN production module**
  - **ISO/IEC 9646-3 compliant**
  - **SDL type definitions translated in ASN.1**

TestComposer prototype usages

- **France Telecom/CNET :** ***relevancy for the end user***

  Experimentation on several low level protocols

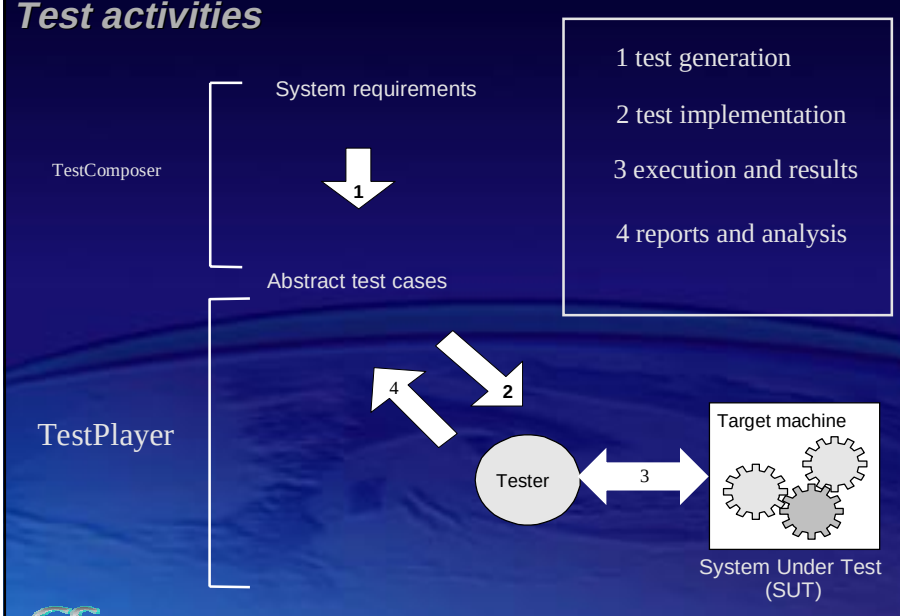
- **CS-Telecom :** ***down to the test execution***

  Two ISDN complementary services (MCID, Call Waiting)

- **Danet, Sema Group :** ***integrated TTCN environment***

  Edition, generation, cosimulation and execution

Test activities



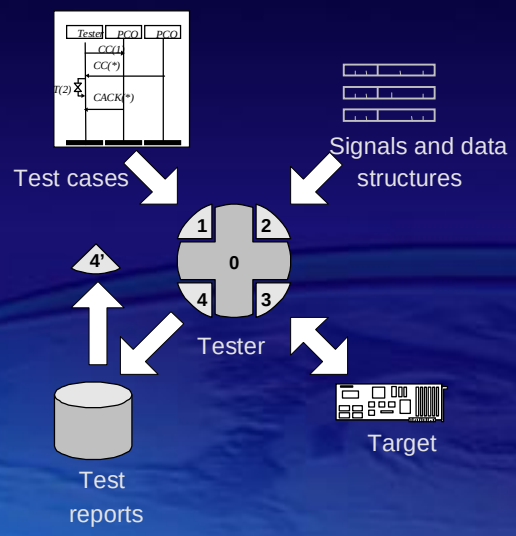
1 test generation

2 test implementation

3 execution and results

4 reports and analysis

TestPlayer principles



1 test cases interpreter

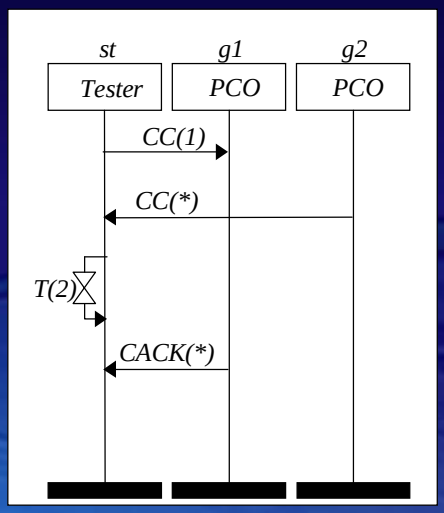
2 declarations handler

3 communication, encoding/decoding

4 results storage

4' results retrieval

MSC and Test Description Language example




```
Output CC with 1 via g1
TimerSet U 5
Expect CC  with star via g2
TimerSet T 2
TimerWait T
Expect CACK  with star via g1
Verdict PASS
```

TestComposer and TestPlayer summary

- **standardized formal notations (SDL, UML, MSC)**
  - *unambiguous and tractable descriptions*
- **interactive or automated test purposes generation**
  - *covers both functional and structural coverage concerns*
- ***sound* and *efficient* test cases generation**
  - *no ambiguous interpretation of the test results*
- **APIs for user specific test language and environment**
  - *seamless integration within customer testbed*

Automated test generation from SDL/UML specifications

Alain Kerbrat[a], Iulian Ober[b*]

[a]Verilog, 150 rue Nicolas Vauquelin, BP1310, 31106 Toulouse cedex, France
Tel: (33) 5 61 19 29 39, Fax: (33) 5 61 40 84 52
e-mail: Alain.Kerbrat@verilog.fr www: http://www.verilogusa.com

[b]INPT-ENSEEIHT, 2 rue Camichel, 31000 Toulouse cedex, France
e-mail: Iulian.Ober@enseeiht.fr

Automated test generation from formal specifications presents a lot of promises, either in cost control or test suite correctness. The telecommunications domain is a field where some interesting tools begin to emerge, either prototypes or industrial strength tools. We present TESTCOMPOSER, a new industrial test generation tool. [2] TESTCOMPOSER allows the generation of conformance tests, from SDL/UML specifications. It comes with TESTPLAYER, which allows to execute test cases written with the MSC language. Both tools are part of *Object*GEODE, an industrial toolset for real time systems development.

1. Introduction

Test generation is an area on which many companies are focusing in search of productivity and quality gains. Current industrial practices include at best semi automated test generation techniques, with all the cost and maintenance problems of an error prone process. This process should be fully and automatically re-executable, with a complete understanding of the relationships between specification and tests, their meaning and their execution results.

Automating the test generation requires a precise and tractable description of the system to test. This is where using Formal Description Techniques (FDT) such as SDL brings most of their benefits. SDL (Specification and Description Language) is an international standard [16], defined over the years by the International Telecommunications Union as a common language for the specification of protocol norms. From 1988, a formal definition of the language semantics was added. These formally defined semantics allow to design reliable tools for the simulation, code generation and test generation.

In the telecommunications field, the FDT based test generation has been developed first by giving a concrete framework to conformance testing [1]. Conformance testing is a necessary step to ensure the compliance of various implementations of a given protocol to its standards. Then this approach has been formalized further in order to facilitate the emergence of tools [18]. However, the interest into FDT based test generation tools has been mostly academic up to the middle of the 90s.

*Iulian Ober's work is financed by a research grant provided by the CS (Communications and Systems) group

[2]This work has been initiated and funded by France-Telecom

TestComposer: automated test generation for real time critical systems

The main focus of this paper is TestComposer. TestComposer [11] is a test generation tool, which allows to generate automatically test suites from Sdl/Uml specification. While designed primarily for conformance black-box tests generation, its design principles make it applicable for grey or white box tests.

TestPlayer: a flexible approach for rapid test prototyping

TestPlayer [12] is a flexible toolbox for specifying, executing and getting reports for tests written using the graphical language Msc. The actual version of TestPlayer works for implementation coming from Sdl/Uml specification. It is however designed in a modular and open way, allowing the user to integrate its own extensions to perform tests in his own environment. Although built as a separate tool, TestPlayer can use TestComposer outputs, the two tools forming a complete test environment, from the test specification down to the test execution and reporting.

Structure of the paper

In section 2, we present the main features of the *Object*Geode [19] toolset.

Section 3 is devoted to the architecture and working principles of the tool TestComposer. We describe the different phases of the test generation process (test purposes edition and generation, test cases generation and test suite structuring). Then we give indications on the added value of this tool.

In section 4, we give an overview of TestPlayer. We show how it integrates with *Object*Geode and TestComposer.

2. *Object*Geode

*Object*Geode is a development toolset for real time and safety critical systems. It supports the joint use of Sdl, Msc and Uml.

Sdl is the Specification and Description Language [16]. Sdl is a formally defined language which allows to describe the architecture and the behavior of a real time distributed system. Originally defined in 1984 it was given formal semantics in 1988. In 1992, object oriented concepts were introduced, especially at the architecture and behavior level. 1996 was mainly for clarifications and strengthening of concepts.

Msc is the Message Sequence Charts language [17]. It is used to describe runs of the system, with a more or less abstract view of its architecture. Msc is especially useful for the description of nominal and error handling behaviors

Uml is the Unified Modeling Language. In the *Object*Geode's process, Uml is used for the analysis phase and for the data description, through the Class Diagram, Use Cases and Statecharts.

The combination of Sdl and Msc can be viewed as a formal extension of Uml, dedicated to hard real-time and safety-critical systems [7].

*Object*GEODE provides tools for the edition of these three languages, simulation, formal verification and C code generation. TESTCOMPOSER relies heavily on *Object*GEODE's simulator, so we give some more details on this tool.

The simulator tool

Validation and verification with *Object*GEODE is done with the Simulator tool. The Simulator allows to simulate the execution of the program, without having to actually execute it in a real environment. It can be seen as a sort of abstract debugger, as it allows to simulate the description step by step, to undo execution steps, to set break points and to watch the contents of variables and queues. Finally, it also allows to record, visualize as MSCs or replay some simulation sequences.

It is also more than a debugger, as it allows to perform automatic simulation, either randomly or exhaustively, with systematic comparison of the behavior with special state machines called observers. The simulator working principle is based on the model checking principle. Model checking [5, 15] consists in building a model of the system under analysis and to check the desired requirements on this model. For the simulator, the model consists in representing the state graph of the specification. Each state of this graph correspond to a different valuation of the system's variables. Each transition correspond to actions such as inputs, outputs, assignments, which can change the content of variables. The check itself amounts to a partial or complete exploration of the model.

The main advantages of this technique are that it can be automated and it is fast. Furthermore, model checking allows the easy production of counterexamples, when a property happens to be false.

The bottleneck of these techniques is the size of the state space to explore, which is exponential in the number of parallel processes of the system. In fact, for SDL-UML specifications, the size of the state space is usually infinite, as SDL allows unbounded process creation, and communication channels are based on unbounded queues. However,the simulator exploration engine works "on the fly", meaning that it builds this state graph on demand, according to the transitions it needs to explore. This "lazy" technique allows to check properties even on infinite state graphs. It will not allow to prove these properties, as proofs need exhaustiveness, but it is very useful for finding errors.

3. Production of test suites with TESTCOMPOSER

3.1. TESTCOMPOSER basics

TESTCOMPOSER working principles come from the conformance testing framework, as defined in [2] and formalized in [18]. Conformance testing can be viewed as a form of black box testing. The interactions between the tester and the System Under Test (SUT) are messages, exchanged through Points of Control and Observation (PCO).

TESTCOMPOSER is the result of the industrial transfer of two complementary research tools, TVEDA [14, 6] and TGV [9, 4]. TVEDA comes from the research laboratory France-Telecom-CNET, it provides for test purposes and test case generation, based on state space exploration combined with heuristics. TGV has been designed in the French research laboratories IRISA and VERIMAG, it provides test case generation based on efficient, on-the-fly state space exploration techniques.

The two prototype tools TVEDA and TGV offer complementary aspects of the test generation process. TVEDA's strengths are the analysis techniques of the formal specification. These techniques allow to derive from the specification meaningful behaviors for the test, which are referred below as test purposes. On the other side, TGV applies powerful formal verification techniques to generate from these test purposes and the specification, the test cases which will constitute the final test suite.

The combination of these two tools, integrated on top of the efficient *Object*GEODE's simulator, forms the new TESTCOMPOSER tool. It comes as a replacement to the current *Object*GEODE's test generator TTCGEN.

3.2. TESTCOMPOSER principles

TESTCOMPOSER principles are depicted in figure 1. TESTCOMPOSER takes as inputs a SDL/UML specification, a specification of the test environment and a possibly empty set of user defined test purposes. These user defined test purposes can be built interactively using the *Object*GEODE's simulator, or written using MSC observers. The tool can complete the set of test purposes by computation of new test purposes according to structural coverage (arrow 1). Then the generation engine (arrow 2) is fed with each test purpose, and produces test cases which are stored in the test case database. Finally, through the test case database API, the test suite is build and written in a user defined format.

3.2.1. SDL/UML reference specification

The SDL/UML specification is the central item for efficient test generation. This specification is generally not used for the generation of the code of the Implementation Under Test (IUT), but rather for describing what is the correct behavior of the IUT, from an external view point. In that sense, the specification may not even reflect the internal structure of the IUT.

This specification must be correct with respect to the user requirements, so that it can be used as a reference later during the tests execution. One way to ensure this correctness is of course the use of validation and verification tools.

Test architecture

The specification is also used to describe the test architecture : an IUT is generally not directly reachable, one must interpose communication layers and Service Access Points. Furthermore, more than one tester are sometimes needed for testing distributed systems. In order to model various test architectures, we choose to integrate its description into the SDL-UML specification. A complete specification for test generation thus corresponds to the description of the System Under test : the IUT embedded into its test architecture.

In example 2, the Implementation Under Test *procotol* is described as a SDL `block`. This `block` is connected to the system's environment via two channels, which bear respectively the signals `ConnectReq, DisconnectReq, ConnectAck, DisconnectAck, Busy` and `DataIn and DataOut`. These two channels will correspond to Points of Control and Observation, when we will consider this specifications for test generation purposes.

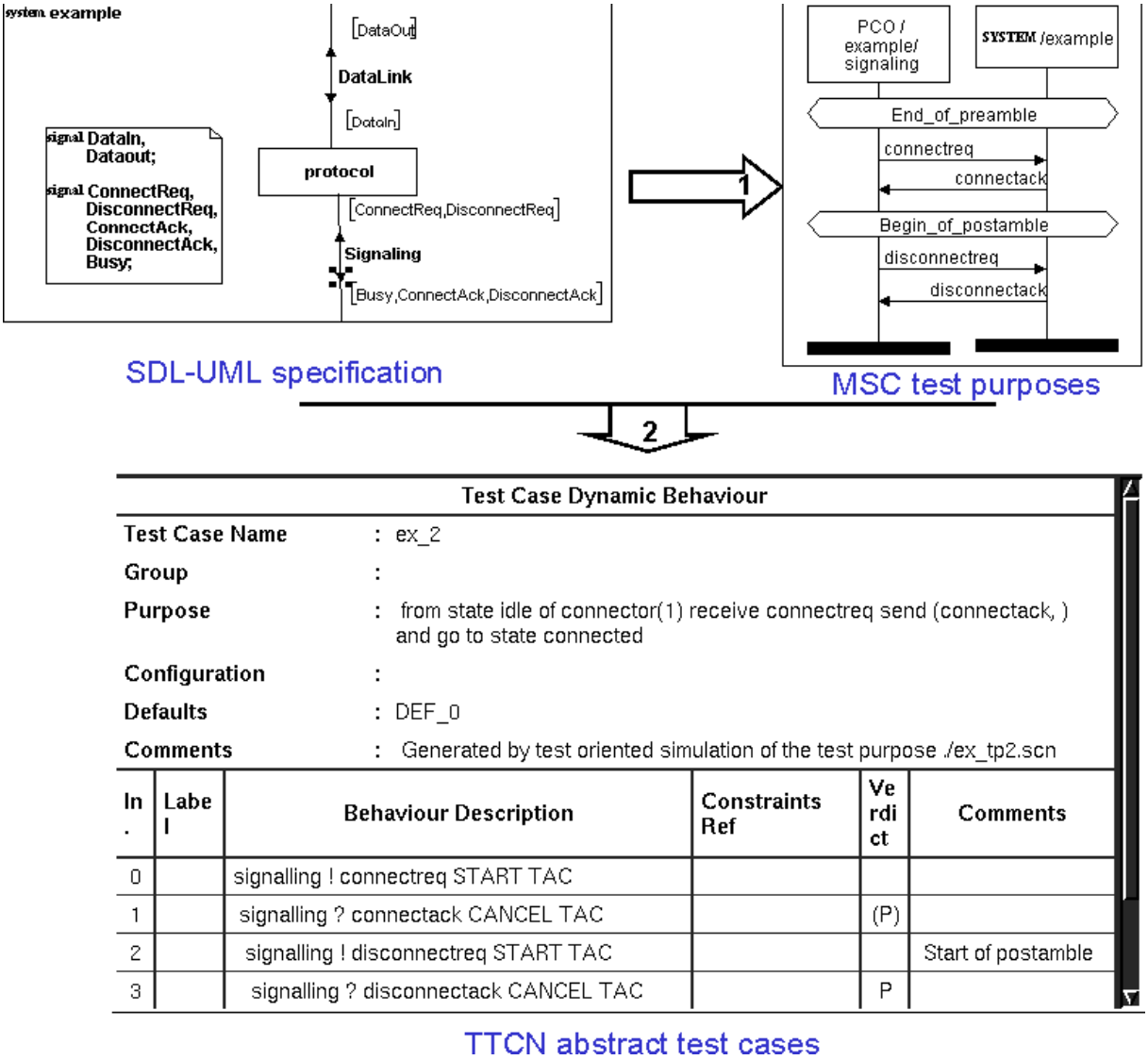


Figure 1. Test generation process with TestComposer

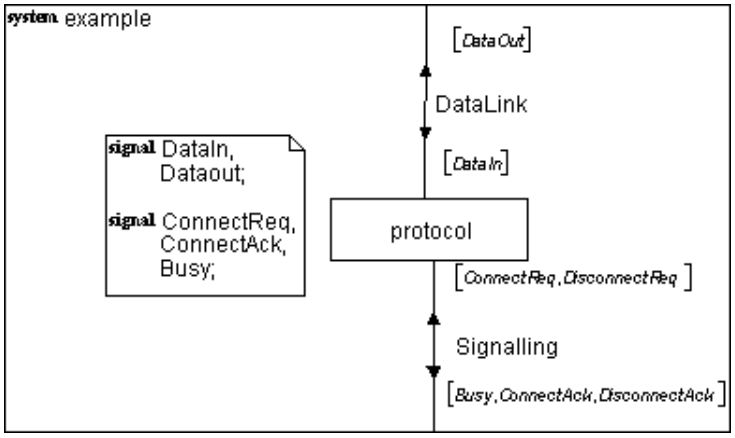


Figure 2. Example of test architecture

Behavior description

At this level, the internal structure of the block *protocol* is hidden. In figure 3, the internal structure, as well as the behavior description of the protocol block is given. This block is composed of two processes : *Connector* which handles the control of the connection, and *Communicator* which handles the data transfer. An internal communication route allows *Connector* to control *Communicator* work.

The behavior of *Connector* is simple : it starts in the `Idle` state; when it receives the `ConnectReq` signal, a check is made to see if resources are available. This check is not formalized, i.e. at this description level, one does not need to know how the check is performed, only that an internal choice is performed. If resources are not available, *Connector* answers `Busy` and returns to the `Idle` state. If resources are available, when *Connector* send the signal `connected` to the *Communicator*, which can begin its work. *Connector* also acknowledges the establishment of the communication by sending `ConnectAck` to its environment.

3.3. Test environment specification

SDL allows to design open systems, i.e. systems whose description is in some sense incomplete, as part of the behavior depends on its interactions with its environment. In the example given in figure 2, the environment and the system interact through the channels SIGNALLING and DATALINK . However, this specification does not say anything more on what the environment does. Will it in fact issue theses messages ? How many times ? In which order ?

To give answers to these questions, TESTCOMPOSER accepts a description of the test environment using the following means :

The `output` command is used to issue one signal just once from the environment to the system.

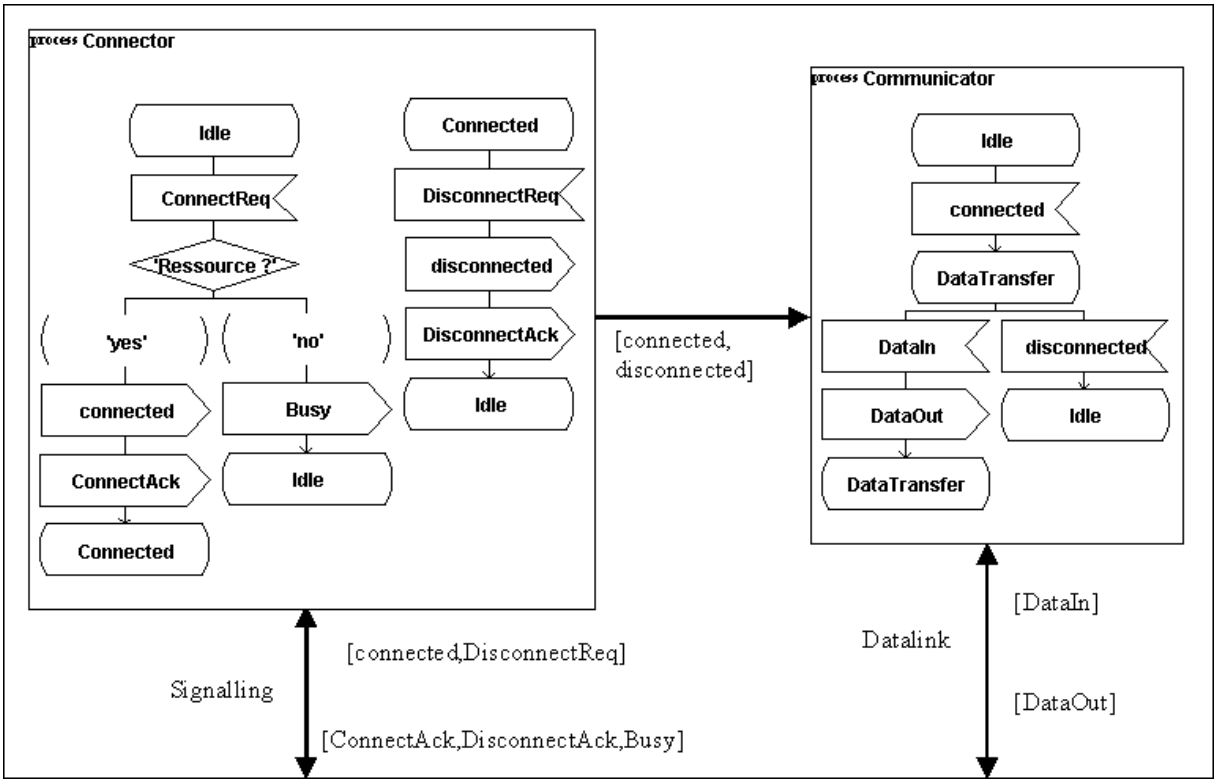


Figure 3. Example of behaviour description

the `feed` command is used to describe what kind of signals the environment is ready to send to the system at any time.

the MSC and GOAL observers come generally in complement to `feeds`, as they are usually used to filter signals so they occur only under specific conditions. For example, a MSC can be used to model the signals interchange of a connection protocol, and then a GOAL [3] observer to model the data transfer. By using appropriate MSC and GOAL observers combination, one can effectively control the simulation.

All of these are parts of the test environment specification. This means in particular that MSC and GOAL observers can be combined with test MSC and GOAL test observers, for a more efficient or more directed test case generation.

For our little example, a possible test environment specification would be :

```
feed Signaling ConnectReq
feed DataLink DataIn
```

which will send these two signals from the environment to the SUT, at any time.

3.4. Test purpose and test case

A test purpose gives a functional view of the test. It is used by TESTCOMPOSER to select in the SDL-UML specification the corresponding behaviors, which will be combined to build the test case. In this respect, the test case can be viewed as the operational view of the test. In short, the test purpose stands for "What should I test ?", where the test case answers to the "How will I test ?" question.

Test purpose description

A test purpose usually is extracted from the system's requirements. It is then formalized, so it can be processed by a tool. A very frequent candidate for this is the Message Sequence Charts (MSC) language [10]: a test purpose MSC contains the main interaction sequence, which correspond to the functionality to test.

An informal test purpose could be "Open the connection, do one data transfer and close the connection". A corresponding abstract, but formal test purpose would be `send ConnectReq, send DataIn, receive DataOut, send DisconnectRequest`, where `ConnectReq, DataIn, ...` are messages send and received by the SUT. A complete test purpose would integrate all the intermediate events needed to fulfill the sequence. Figure 4 shows this kind of test purpose.

TESTCOMPOSER also accepts test purposes described with the GOAL language. GOAL [3] is used to describe automata which will play the role of observer during the system simulation. GOAL observers are used to filter out messages and behaviors, impose a predefined order on messages exchanges, evaluate stop or continue conditions depending on expressions of the system's variables. In addition, GOAL observers have access to all internal elements of the specification, and can also change the values of variables or the contents of queues. So GOAL observers can be used for more elaborate simulations, involving fault injections for example.

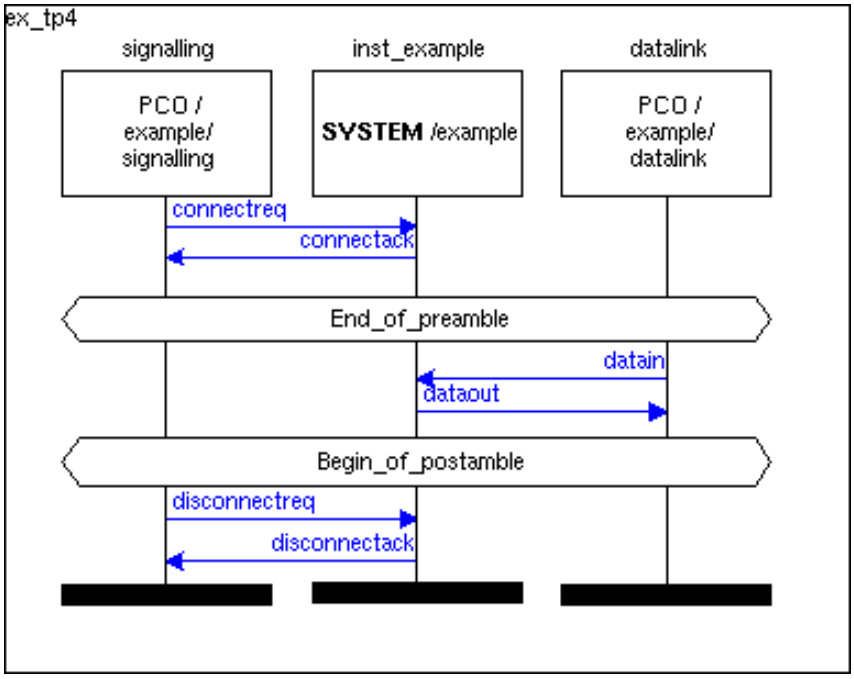


Figure 4. Example of MSC test purpose

"Manual" test purpose description

The test purposes can obviously be written using the MSC and GOAL editor which is part of the *Object*GEODE toolbox. When an SDL-UML specification exists, the simulator allows to produce the MSC test purposes more efficiently, as it will be shown in the next paragraphs. However, protocol norms generally use the MSC language to describe nominal and error handling use cases. These use cases can then be straight used as test purposes.

Interactive test purpose description

When a SDL specification exists, a more efficient way is to use the *Object*GEODE simulator (see section 2). The user can use the simulator to "compute" what is in his sense, an interesting test purpose. Then the simulator can complete this test purpose by computing all the reactions resulting from the last inputs to the system, so that the test purpose ends in a stable testing state. Finally, the simulator saves this test purpose in the test purposes database.

Automatic test purpose generation

In addition to user defined test purposes, TESTCOMPOSER is able to compute test purposes straight from the system's specification. This feature allows to take the maximum benefits of the specification, especially if its internal architecture maps the IUT architecture, as this is often the case for low level protocols.

The principles of this feature come from a practical viewpoint, which was exemplified by the tool TVEDA : for telecommunication protocols, the SDL specification architecture generally maps to the code architecture. A good test coverage at specification level translates into a almost-as-good coverage at implementation level. So the idea is to generate a set of test purposes such that every transition of the specification is covered by at least one test purpose.

Technically, the generation of test purposes with TESTCOMPOSER is done at a finer level, as the coverage unit chosen is the *basic block*, i.e. roughly the biggest sequence of instructions without branching conditions (no decision, no join, no state change). The computation itself is again based on the *Object*GEODE simulator. We extended its state space exploration techniques so that for each uncovered basic block during exploration, a new test purpose is computed and saved.

In practice, the user first chooses in the specification what are the parts to cover by the test purpose generation, then he gives an upper percentage of coverage to reach (one hundred per cent is usually unreachable for real specifications). Once the computation finishes (by reaching the desired percentage or by resources exhaustion), the user can analyze the coverage results through a graphical browser, which allows to visualize the uncovered parts. Then he can alternatively change some generation parameters, such as the exploration mode, or use his system's knowledge to build interactively a test purpose for the coverage of these parts.

Postamble computation

A test purpose is structured into three parts :

A test body which corresponds to the execution of the events covered by this test purpose,

a preamble which leads the SUT from its starting state to the beginning of the test body,

a postamble which gets the system back in a state where it can execute the next test case. This kind of state can be of course the initial state of the system, but these states usually are no more reachable after the system start. Other states of interest are the *initial stable states*, i.e. the states where the system ends in, when started with no external stimulation.

With TESTCOMPOSER the postamble goal can be user defined in the sense that the user can define the characteristics of the state the postamble should go to. Classical examples can be getting a given process back to its initial state, or forcing the disconnection phase of the protocol. A suitable postamble goal for our example would be the `Idle` state of the *Connector* process. During the test purpose generation process, TESTCOMPOSER automatically computes a path leading to this goal. The postamble of the figure 4 corresponds to a sequence going back to this state.

Test purposes database

The combination of interactive and automatic test purpose generation allows the user to build a test purposes database. This database is saved in session independent files, so

the user can possibly go on with the subsequent test generation phases, and come back to add some test purposes in the database.

3.5. Test cases computation

A test purpose describes what is the aim of the test, by giving the sequence of events leading to test success. However, to turn a test purpose into a test case, one must add some more information, especially when dealing with real-time distributed systems. The test case must provide all the information needed by the tester for the test correct execution, but also additional information to provide the user with accurate data in case of test failure. The failure of the test can come from several causes, among them : an implementation fault, a faulty test case, a non deterministic implementation.

Let us take care of the first two cases.

- A implementation fault is what we are looking for. In a real time system, it corresponds essentially to the following situation: a given message was expected during a given interval of time; a wrong message was received, or the right message, but at the wrong time. It is easy to take care of wrong messages, by considering that any unexpected message leads to the test failure. For dealing with out-of-sequence messages, the test case should contain timers to bound the wait for messages.

- We claim that with TestComposer test cases can not be faulty, as long as the Sdl-Uml specification reflects the correct behavior of the specification. This can only be true if this specification has been thoroughly simulated, best verified, so that it can be used as a trustful reference.

The last point is the most tricky one, and in fact it is what makes test generation and writing difficult for distributed systems.

Dealing with non determinism

Distributed and real time systems are generally non deterministic. Moreover, even a deterministic IUT can be viewed as non deterministic when embedded in its test architecture. Many test tools do not allow to handle well non determinism. They usually allow only two test execution results : either the test succeeds, or it fails. However, several executions of the same test can hold both results, when applied on a non deterministic system. This makes the interpretation of test results difficult and more time consuming, as every test failure must first be analyzed to reject failures due to non determinism.

The handling of non determinism in the description of test case thus helps to reduce greatly the test execution costs. To allow this, a test language such as Ttcn, designed for the test of telecommunication protocols, introduces an alternate test verdict, the *Inconclusive verdict*. This verdict allows to distinguish between a test failure due to an internal choice of the SUT, and a real implementation error.

TestComposer is designed to handle correctly non determinism, both during the test case computation and in the test case description. This is explained in the following section.

3.5.1. From a test purpose to a test case

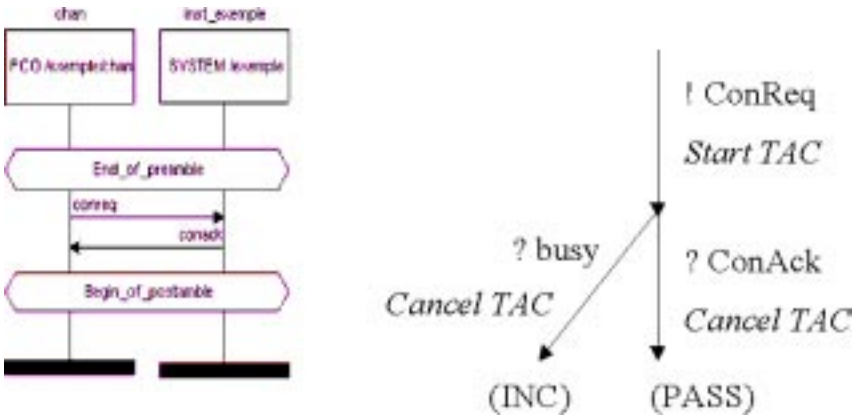


Figure 5. From a test purpose to a test case

The computation of the test case from a test purpose is done in several steps, described in the following paragraphs.

Computation of the main path

The test purpose describes the sequence of events leading to the test success. This sequence can be more or less abstract, i.e. there may be some events missing. In any case, many behaviors described at the specification level can match the test purpose. The first task of the tool is to find such a matching behavior in the specification. This is done by exploration of the state graph which corresponds to all the behaviors described by the specification. The first one found will be the success path of the test case, i.e. the path leading to the Pass verdict. In example 5, the main path is indicated in the test purpose, and transposed as is in the test case, from the tester point of view (i.e. inputs to the system besome outputs from the tester, and vice versa).

Computation of alternatives

Once a success path is found, the tool begins to compute alternatives to this paths. These alternatives are still behaviors described in the specification. They are of two natures :

Pass alternatives When testing distributed systems, one often has to wait for two or more messages, without any idea of the outgoing order of these messages. Even if these messages are issued in a given order by the IUT, the use of several PCOs and communication delays between the IUT and the tester can change the order of messages on reception. So even if the test success path specifies that message `a` should be followed by message `b`, the other order can lead to the same test success.

In order to detect these situations, TestComposer explores alternatives paths from the success path, to see if they also lead to success. These paths are explored as long as they consist only of SUT's outputs. Loops are also computed, i.e. paths going back to some previous step of the main path.

Inconclusive alternatives If an alternative to the main path does not lead to the success, it still remains a valid sequence, as it is described in the specification. So TestComposer keeps them in the computed test case, truncated to their first event, and annotated with the *Inconclusive* verdict.

In example 5, an *Inconclusive* alternative is computed, for the reception of the valid, but not desired message `Busy`.

Timers setting

Several timers are introduced to handle the different situations the tester must face :

"Wait for answer" timer TAC This timer is the most classical one. The tester sends a message, then waits a bounded time for reactions. Its timeout leads to the test failure.

"Wait for user input" timer TWAIT Protocol testing sometimes requires user interaction, even in the course of an automatic test execution, as The tester does not always have access to all the interfaces of the System Under Test. In that case, a long range special timer is set, leaving enough time for a user to provide the necessary inputs. Its timeout leads to the test failure.

"Wait for no answer" timer TNOAC A test sequence can lead the system in a state where it just waits for another input, without emitting any message to the tester. In order to detect this situation, the tester sets a specific timer, which timeout allows to continue the test.

"Wait for timer expiration" timer TEMPTY The System Under Test can have its own timers running. The timer TEMPTY is meant for the detection of the expiration of these timers.

3.5.2. On the fly generation

The generation engine is base on state space exploration techniques. Many test generation tools need to build the complete state space before being able to compute a test case. The ability to work "on the fly", i.e. generation of the test case during the state space exploration is a key to efficiency, as it allows to stop the exploration as soon as the test case is complete

3.6. From test cases to the test suite

The test cases produced are stored into a database. This database is accessible through an Application Programming Interface which provides access to data types and signals definition, messages values, test architecture, test cases and test suite definitions. Then a test suite production module can be interfaced to this API, to produce the user desired format for its test suite.

The first test suite production module we implemented with this API generates a Ttcn-mp file. Ttcn is an international standard for the description of protocols tests, defined in [1]. Existing Ttcn environments allow the definition and edition of test suites, as well as the compilation and execution of the tests on real targets.

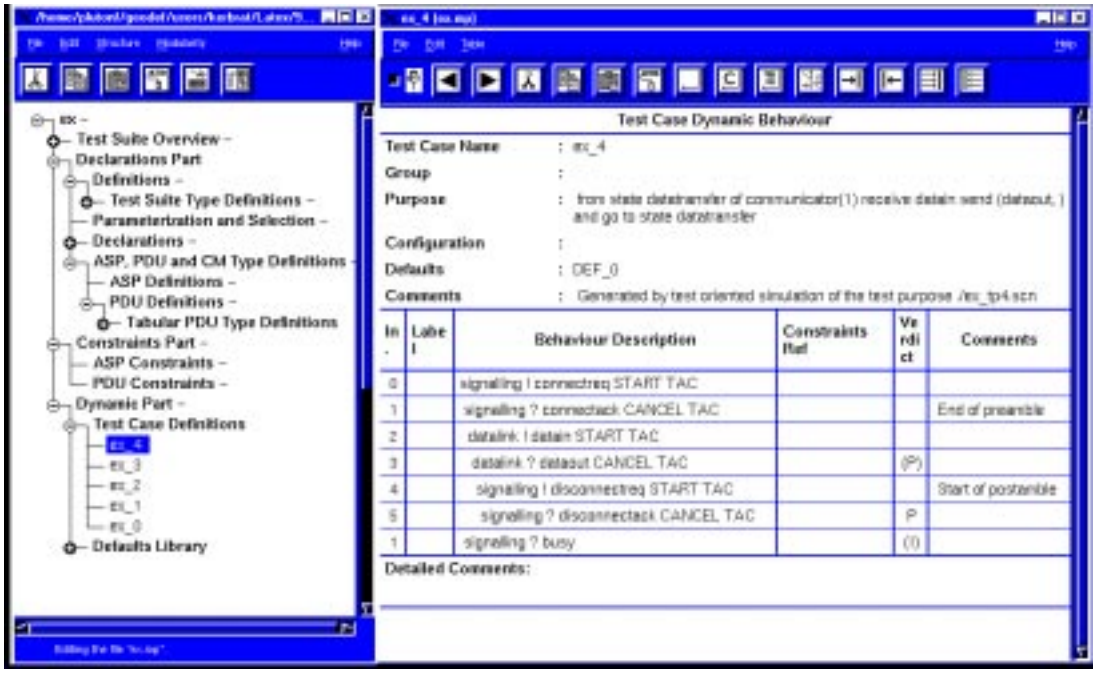


Figure 6. Excerpt of the TTCN suite, with Sema Group TED editor

On the TTCN example of figure 6, we show the final test case, generated from the figure 4. TTCN is a tabular notation, the indentation (denoted in the first column) indicates the sequence number of the corresponding event. Each event is either a sending (!) or reception (?) of a message, on a given PCO. Timer commands (`START TAC`, `CANCEL TAC`) are associated to each event.

The main sequence is annotated with the temporary and final Pass verdicts, and information about its structure. The `busy` reception is indicated as an alternative to the `ConnectAck` reception, and leads to the temporary Inconclusive verdict.

Finally, an informal test purpose is synthesized from the test body, allowing a human operator to manually perform the test if necessary.

3.7. TESTCOMPOSER added-value

The added value of TESTCOMPOSER depends first on the availability of a formal specification of the SUT. Developing a SDL-UML specification is a fruitful task, as it allows to track many analysis and design errors at a early stage, just by having to formalize the requirements, and later by using a tool such as the simulator to simulate and validate the specification behaviors. Writing and maintaining such a specification is very cost effective, especially when dealing with new systems. In this context, test generation as well as code generation brings its best value.

For existing systems, where no formal specification exists, the cost of setting up automated test generation must be compared to the cost of writing a test suite from scratch.

The specification must be written, validated, the test environment must be described and all generation parameters must be set and tuned. The complete task can take up to twice the time needed to write the test suite, especially if we consider someone expert in test languages and novice in specification languages, as shown by the ongoing study [8]. However, this overhead is paid for only initially,furthermore, one must consider the other benefits of test generation :

Link between requirements and tests The relation between tests and high-level requirements is better maintained, as the SDL-UML specification comes straight from the high-level requirements. Then TESTCOMPOSER test generation process ensures the correctness of the test produced, and allows to trace each test back to the specification.

Easy test suites maintenance Most requirement or design changes usually imply major rewriting of the test suites. When using automated test generation, any requirement or design change can be reflected directly in the SDL-UML specification, and the new test suites are then re-generated at no cost.

Faster implementation debugging the SDL-UML specification, once validated, can be considered as a reference. Then TESTCOMPOSER ensures that the tests generated are correct, as long as the specification is correct. It is a major improvement on current test practices; when a test fails, the usual question is "What is wrong, the implementation, or the test ?" As up to 15 percent of hand written test cases can be false, this is a valid, very time consuming question. With TESTCOMPOSER, the specification is the reference, and the test failure is always attributed to the implementation.

Identifying `Inconclusive` verdicts from `Fail` verdicts is also a gain when debugging, as an `Inconclusive` does not require an investigation.

4. Executing tests with TESTPLAYER

TESTCOMPOSER requires a full SDL- UMLspecification in order to generate tests. However, it is often convenient to use directly the MSC language to describe and execute tests, without having to bother with the full system's description.

This is where a tool such as TESTPLAYER brings its value. TESTPLAYER defines an open architecture for executing conformance tests, and consists of a set of tools for interpreting test MSCs, translating MSCs into some internal test scripting language, executing test scripts and examining test results. TESTPLAYER can take as inputs the test MSCs generated by TESTCOMPOSER, or independently, test MSCs provided by some other means.

The architecture of TESTPLAYER described below, relies on two fundamentals: the MSC language and an internal scripting language called TDL (Test Description Language). The TDL language exists in order to provide the parts that are missing in MSC but are still necessary to express certain test cases: variables, control structures, explicit verdicts and an imperative language. The TDL language is based on a set of test primitives (see [20] for a complete description):

Output - sends a message to the system

Expect - generically specifies a signal that is expected to come from the system

TimerSet, TimerReset, TimerWait - for manipulating timers

Alternative - expects one in a set of possible (generically specified) responses from the system

Verdict - explicitly gives the verdict of a test script

These primitives can be implemented in almost any language (C/C++, TCL, Java) at little expense, and the host language brings in variables and control structures. Within our architecture the main host language is TCL [13] and TDL denotes the extended TCL.

TESTPLAYER provides an open architecture for testing, based on the lightweight formalisms of tMSC and TDL. It offers customization and power of expression sufficient to be applicable to a large number of real systems and test cases.

Figure 7 depicts the overall architecture of TESTPLAYER. We distinguish between three phases: the test preparation phase, the test execution phase and results use phase.

Test campaign preparation

When preparing a test campaign, the user must provide the test cases. A test case may either be expressed by an MSC hand-written or generated by TESTCOMPOSER or it may be expressed directly in a TDL script. If the former is the case, then the MSC is translated automatically in a TDL script, which can be modified further. The TDL Gen tool that does this translation is provided as a part of TESTPLAYER. The rules for translating an MSC into a TDL script are quite straightforward, as MSC constructs map almost one-to-one to TDL primitives.

The engine that executes the TDL scripts needs to know the data types and the signals that will be exchanged between the tester and the system under test (SUT), in order to be able to encode/decode and send them out to the system. Consequently, a part of the preparation phase is dedicated to the description of the data and signal types of the system, and to the generation of the encoding/decoding component (1) of the TDL execution engine. The data and signal types may be described in SDL [16] and ASN.1 /citeZ.105, and then the encoding/decoding component (1) is generated automatically using StubGen, a tool provided with TESTPLAYER.

Test campaign execution

At execution time, the core of TESTPLAYER is the TDL execution engine. The library containing the implementation of the test primitives is the fixed part of execution engine, i.e. it cannot be modified or replaced. The entire conception and architecture of TESTPLAYER relies on it.

The test primitives library depends on a set of customizable components. First there is the TCL wrapper that makes the test primitives (Output, Expect, Verdict, etc.) available as TCL commands in an extended TCL interpreter. This is essential, since the TDL scripts generated from MSCs are TCL programs (with calls to our primitives). As mentioned above, the implementation of the test primitives is not depending on any host language,

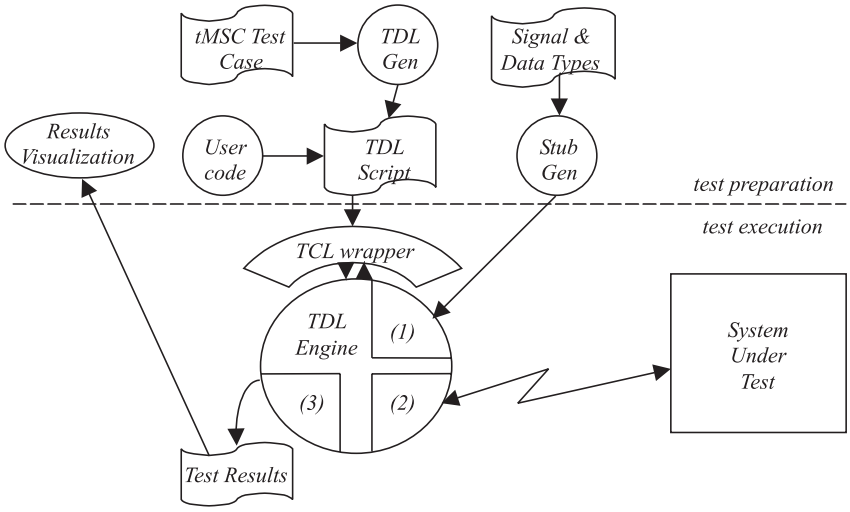


Figure 7. TestPlayer Open Architecture

so they may be used as well from C. However, by doing so, the user misses the advantage of having an automated translator that converts test MSCs into TDL scripts.

Another component used by the test primitives implementation, and part of the TDL engine, is the Signal Encoding/Decoding component denoted by (1) on Figure 7. This component contains call back functions for signal and data encoding/decoding/matching. The tester communicates with the SUT using a certain protocol, and the functions contained in component (1) are called to pack/unpack the signals and data to be transmitted/received. Component (1) also contains the functions that match an incoming message towards a generic message specification given in an Expect or Alternative command. As mentioned above, the Signal Encoding/Decoding component may be generated from the SDL or ASN.1 description of the system signals and data types.

The Protocol component denoted by (2) on Figure 7 implements the protocol used by the tester to communicate with the system under test. TestPlayer may be used to apply tests in different configurations, in which the tester and the SUT may be on the same machine or on two different machines connected by a network, a serial line, etc. To achieve such a generic architecture, the communication functionality must be kept in a separate component in order to allow for changes in the underlying communication means. The Protocol component (2) must provide a simple interface containing functions for opening and closing connections, sending binary data to a connection and doing blocking reads on a connection. Beyond this, the user is free to use whatever suitable means to implement the communication between the tester task and the SUT (TCP/IP, some proprietary protocol, OS IPC, etc.)

The Test Reports component (3) formats and stores the partial or final results of a test in a log. We kept this functionality in a separate component because a user may require that the results of the tests be stored in a specific format: a database, a log file

with a proprietary format, etc. Any replacement for this component must implement a simple interface containing the following functionality: opening/closing a log, writing a test event to a log, writing a test verdict to a log.

Using test results

After executing a test campaign, the results of the test cases and the events that occurred all along the campaign can be visualized. Using a specialized tool, the Results Visualization tool, one may navigate from any event occurrence to the TDL source line or to the MSC construct that corresponds to it. This tool accesses the information contained by the test log, using a replaceable component for reading the log, similar to (3).

Thus our test execution architecture is based on the interpretation of test MSCs and on the Test Description Language, and the parts describing system signals and data types, the communication protocol between the tester and the system, and the formatting of test results are left open and customizable. We instantiated the open components to obtain a concrete test execution framework for testing *Object*GEODE systems.

4.1. TESTPLAYER for *Object*GEODE

TESTPLAYER for *Object*GEODE is the complete TESTPLAYER framework for systems modeled in SDL [16] with *Object*GEODE [19] and for which code is automatically generated with the *Object*GEODE SDL-C Code Generator. It is an instantiation of all the generic components described in the previous section. The Signals Encoding/Decoding component (1) is generated by the StubGen tool, which is able to take in input the SDL description of the system and extract all the necessary information. The Protocol component (2) uses TCP/IP to communicate with the system side. The tested system being generated automatically using the *Object*GEODE SDLC Code Generator, a proxy task is automatically added to the system. This task ensures the communication with the tester using the same protocol as the TDL engine component (2), and has a minimal interference with the implementation under test. Finally, the Test Reports component (3) was implemented so that it keeps the test results in an ASCII format test log. These results can be visualized and used with the Results Visualization tool.

5. Conclusion

We have presented TESTCOMPOSER, an automated test generation tool for conformance testing. It is completed by TESTPLAYER, an open and flexible environment for test description and execution using MSCs. Both modules will be part of the *Object*GEODE toolset during the coming year.

TESTCOMPOSER allows the test designer to work on a high level SDL/UML specification, rather than working at a low level test language. This specification can be validated, using a tool such as *Object*GEODE's simulator. This allows to consider this specification as a reference. TESTCOMPOSER then provides for assisted or automated test generation, dealing completely and correctly with the non deterministic nature of distributed real-time applications.

The ability to generate test purposes based on the structural coverage of the specification makes of TESTCOMPOSER an important tool for certification purposes. If we consider a critical application, which code is generated from an SDL-UML specification,

then its certification requires either the certification of the code generator, or a complete, one hundred per cent functional and structural coverage of the code by the tests. As the code generated from SDL is structurally very close to the specification itself, test purposes generated by TESTCOMPOSER are very useful to ensure coverage of the code. Furthermore, abstract test purposes can be defined by the user, to provide for functional coverage.

Last but not least, the performances of the test case generation engine allow the computation of long, tricky test cases which would be hard to design by hand. Failures of critical systems usually don't come from single errors, but mostly from the conjunction of different errors. Writing tests for this kind of "errors interaction" is hard and very time consuming; the ability to work on-the-fly brings here the possibility to explore very long behaviours, and to generate test cases corresponding to these behaviours.

TESTCOMPOSER is actually tuned for working in a TTCN environment, together with the TED editor of the UK-French company SEMA GROUP, and the TTCN compilation execution toolbox from the German company DANET GMBH. However, its reliance on standards for test purposes description, and its API to the test suite make it open to user specific languages and testbeds.

The main bottleneck for the use of TESTCOMPOSER is writing and validating the SDL-UML specification. Development processes which do not already integrate such practices can find this is too heavy an investment to make in the short term. Using a light weight approach such as TESTPLAYER, which in its most generic form does not require a full SDL-UML specification, can fit these development processes.

REFERENCES

1. ISO/IEC International Standard 9646-1/2/3. OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework - Part 1 : General Concept - Part 2 : Abstract Test Suite Specification - Part 3 : The Tree and Tabular Combined Notation (TTCN), 1992.
2. ISO/IEC International Standard 9646-1/2/3. OSI-Open Systems Interconnection, Information Technology - Open Systems Interconnection Conformance Testing Methodology and Framework, 1992.
3. B. Algayres, Y. Lejeune, and F. Hugonnet. GOAL: Observing SDL Behaviors with GEODE. In *SDL forum'95*. Elsevier Science (North Holland), 1995.
4. Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Claude Jard, Thierry Jéron, Alain Kerbrat, Pierre Morel, and Laurent Mounier. Verification and test generation for the SSCOP protocol. *Journal of Science of Computer Programming, Special Issue on The Application of Formal Methods in Industry Critical Systems*, To appear, 1999.
5. E. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic. In *10th Annual Symposium on Principles of Programming Languages*. ACM, 1983.
6. M. Clatin, R. Groz, M. Phalippou, and Richard Thummel. Two approaches linking a test generation tool with verification techniques. In A. Cavalli and S. Budkowski, editors, *Proceedings of IWPTS'95 (8th Int. Workshop on Protocol Test Systems, Evry, France)*. INT, sep 1995.
7. E. Cohen. UML, SDL, and MSC: Object-oriented and real-time combined. *Distributed Computing*, 1(2):146–160, June 1998.

8. CS-Telecom. Experimentation of an automated test generation prototype. Technical report, CS-Telecom, 1999.
9. L. Doldi, V. Encontre, J.-C. Fernandez, S. Jéron, T. and¡ Le Bricquir, N. Texier, and M. Phalippou. Assessment of automatic generation methods of conformance test suites in an industrial context. In B. Baumgarten, H.-J. Burkhardt, and A. Giessler, editors, *IFIP TC6 9th International Workshop on Testing of Communicating Systems*. Chapman & Hall, September 1996.
10. Jens Grabowski. *Test Case Generation and Test Case Specification with Message Sequence Charts, Ph.D. Thesis*. Universitat Bern, 1994.
11. A. Kerbrat, T.Jéron, and R. Groz. Automated test generation from SDL specifications. In *Proceedings of SDL forum'99*. Elsevier Science (North Holland), 1999.
12. I. Ober and A. Kerbrat. Test specification and execution using tmscs. In *Submitted to Forte/PSTV'99*, 1999.
13. John K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Pub Co, 1994.
14. M. Phalippou and R. Groz. Evaluation of an empirical approach for computer-aided test cases generation. In I. Davidson, editor, *Proceedings of IWPTS'90 (3rd Int. Workshop on Protocol Test Systems, McLean VA, USA)*. Corporation for Open Systems, oct 1990.
15. Jean-Pierre Queille and Joseph Sifakis. Fairness and related properties in transition systems — a temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983.
16. ITU-T Recommendation Z-100. Specification and Description Language, 1996.
17. ITU-T Recommendation Z-120. Message Sequence Charts, 1996.
18. ITU-T Recommendation Z.500. Formal methods for conformance testing, 1997.
19. VERILOG. ObjectGeode reference manuals. Technical report, VERILOG, 1996.
20. VERILOG. TestPlayer Reference Manual. Technical report, VERILOG, 1999.

Test Factory (TSite®):
a next step in structured testing

Ingrid B. Ottevanger

Agenda

- Introduction
- Structured testing
- Rationale of TSite
- Characteristics
- Examples
- Conclusions
- Challenges

Introduction

IQUIP Informatica B.V.

- Large test department (over 275 testers)

- Customers are most Dutch and Belgian top-200 companies
- Structured testing according to the TMap® product set
- R&D

IQUIP
98 425 C&T 3

Testing: what is it?

- Find errors

- Inform about quality and risks
- Prevent errors
- Produce reusable testware

- Improve test process

IQUIP
98 425 C&T 4

Testing of *information systems*

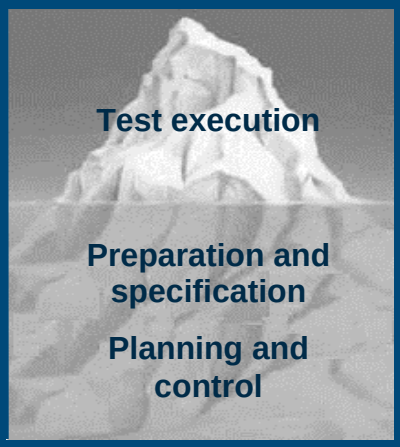
- Application software
- Hardware, system software
- System management
- Procedures, tools
- Docuware
- Training, support
- Implementation
- ...

Quality characteristics

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

IQUIP
98 425 C&T 5

Time needed

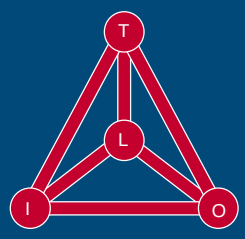


- 40 % find defects, analyse and evaluate them
- 40 % prepare tests and specify test cases
- 20 % plan and control tests

IQUIP
98 425 C&T 6

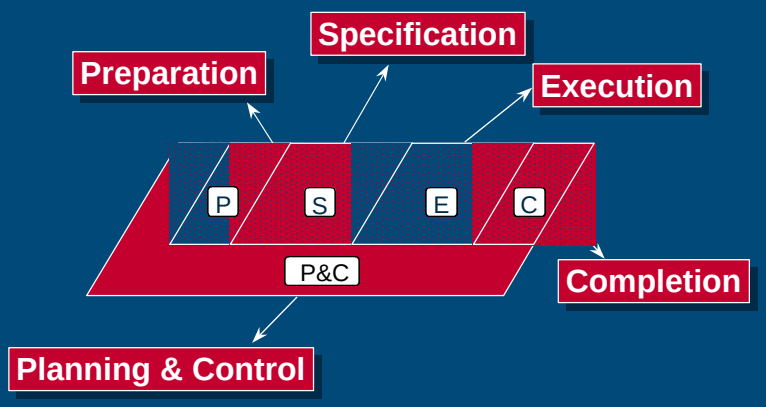
Need for structure

- What, when? — Life cycle of activities
- How? — Techniques
- Where, using what? — Infrastructure and tools
- Who? — Organisation



IQUIP
98 425 C&T 7

Phases (Life cycle)



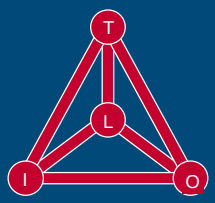
IQUIP
98 425 C&T 8

Step 1

Test as good as possible under:

- Complexity
- Risks and priorities
- Time pressure

Solution: structured testing



IQUIP
98 425 C&T 9

Step 2

- Higher and higher demands
- Too much time, too much money
- Unknown test quality

- Loss of investment
- Test personnel elsewhere

- 'Management' of test team
- Improve test process

Solution: Test Factory

IQUIP
98 425 C&T 10

TSite®

- Dedicated test organisation
- For more than one project, system or customer
- Controllable costs and time
- Continuous improvement of process quality

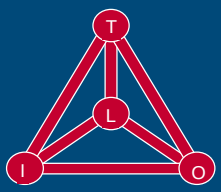
Synonyms:

- Test Factory
- Test Service Centre
- Test Competence Centre
- ...

IQUIP
98 425 C&T 11

What's the difference?

- Structured test process with extra demands
  - reusable
  - flexible, adaptable
  - ready-to-use



- And special use of
  - automated testing
  - infrastructure
  - human resource management

IQUIP
98 425 C&T 12

Test automation

- Tools for all phases, e.g.
  - project management
  - defect repository
  - coverage analysis
  - reporting
- Test suite is automated system
  - maintainable
  - data driven
  - business specialist for test cases
  - tool specialist for test programs

IQUIP
98 425 C&T 13

Reusable infrastructure

- Implementation for parallel test projects
- Configuration for more than one business system
- Standards for test products
- Procedures and tools for several phases

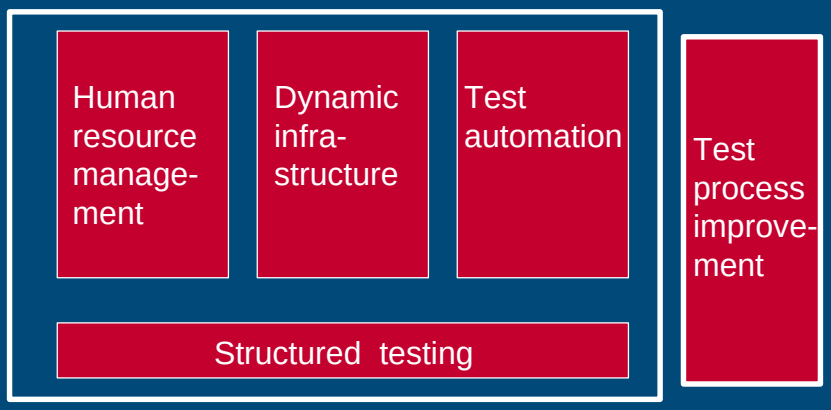
IQUIP
98 425 C&T 14

Human resource management

- Long term
- A career for personnel
  - personal development within TSite
  - knowledge and skills
  - testing is a **profession**
  - atmosphere at work
- Specialisations for different tools
- Training in standards and procedures

IQUIP
98 425 C&T 15

Implementation



IQUIP
98 425 C&T 16

Prerequisites

- Structured testing
- Test maturity
- (Semi) permanent organisation
- 1 : n
- Scope
- Reusable environment, procedures, expertise
- All (most) phases of life cycle model
- Process improvement

IQUIP
98 425 C&T 17

Examples

	TSite?
Usability labs (various)	✓
Test Centre (Bank and Insurance company)	✗ → ✓
Test Service Centre (Telecom industry)	✓
Competence Centre Testing (Social insurance)	✓ → ✗
Y2K Test Centre (Railways)	✗ → ✓ ?
Test Centre (Government)	✗ → ✓ / ✗

IQUIP
98 425 C&T 18

Conclusions

- Structured testing
  - yes or on the way
  - TSite sometimes misused (internally as well as externally)
- Test maturity
  - in varying degrees
- (Semi) permanent organisation
  - depending on management support
- 1 : n
  - e.g. complete organisation, one BU, one mega project, all Y2K projects, one supplier

IQUIP
98 425 C&T 19

Conclusions (continued)

- Scope
  - deliberate choice
- Reusable
  - expertise sometimes not
- All (most) phases of life cycle model
  - specification and execution sometimes not
  - completion 'in the crumple zone'
- Process improvement
  - going concern

IQUIP
98 425 C&T 20

Conclusions (general)

- All criteria work in varying combinations
- Success as a result of availability of
  - expertise
  - infrastructure
  - resources

- Shorter total project time?
- Cheaper?
- Higher and predictable quality

IQUIP
98 425 C&T 21

Challenges

- Acquisition
- Personnel
- Bureaucracy
- Planning
- Services
- Communication and PR

IQUIP
98 425 C&T 22

IQUIP
Test Factory (TSite®):
a next step in structured testing
IQUIP
98 425 C&T 23

Test factory (TSite®): A next step in structured testing

Dr. Ingrid B. Ottevanger

IQUIP Informatica B.V.
PO Box 263, 1110 AG Diemen, The Netherlands
Tel: +31 20 660 6600
Fax: +31 20 695 3298
E-mail: i.b.ottevanger@iquip.nl

0 Abstract

Both the increasing importance of software in today's society and the costs that are involved in testing, confirm the need for structuring the testing process. Structured test methods have become widely used for this purpose, e.g. TMap® (**T**est **M**anagement **ap**proach to structured testing of software products). Given a structured way of testing, organizations may still face challenges, the next level of them: how to make testing faster, cheaper, better still.

TSite is a dedicated test organization that performs tests for more than one project, system or customer. Emphasis is put on the process of testing (efficiency, quality) rather than project aspects. Much attention is therefore paid to reusability, conservation of testware, tooling, human resource management; in short, to long term aspects.

1 Introduction - When Structured Testing does not Suffice

Testers have been mainly occupied with facing the major challenge: how can we organize the best possible test, for a large and complex information system, within the narrow limits of time and budget.

The approach of 'structured testing' provided an answer to this problem. Key issues in this approach are thorough planning and preparation, building a suitable test infrastructure, developing standard test techniques and embedding the test process in the organization. For most organizations today, this is still a goal worthwhile to achieve.

However, organizations that have already implemented structured testing successfully, are facing a next level of challenges:

High cost
Testing claims an increasing amount of the IT budget. The productivity of programmers increases owing to the use of advanced languages and tools, while the productivity of testers lags behind, one of the reasons being that each test project has to be organized from scratch.

Long lead times
Many customers demand ever smaller time frames for delivery. From a development point of view the total project time is shortened by the application of new techniques and methods (e.g., Rapid Application Development). The test team is expected to respond quickly to the rapid and frequent deliveries of new releases, often under circumstances of unclear or insufficient specifications.

Unknown quality of the test
With every new project the customer has to assess how thorough and well performed testing is being executed. Many customers do not want to bother anymore and demand a well defined test process with a constant and predictable quality.

A Test Factory, or TSite, is an answer to the above mentioned demands.

2 TSite in Relation to Test Process Improvement

2.1 Purpose

TSite is a dedicated test organization that performs tests not just for one project but for many different projects, different systems and different customers. It may be an independent organization by itself or an independent part of the IT department of a larger organization. It focuses on the test *process* to improve its efficiency and quality. The goals of TSite are: cheaper, faster, better testing.

With this it is able to sell its test services at competitive prices. Such an organization we find under different names, including test factory, test plant, test service centre, test competence centre.

TSite is useful especially in situations in which reusability is feasible, or infrastructure or expertise are scarce. These include:

- Maintenance testing;
- Large scale projects such as Millennium and Euro;
- Special IT environments, such as Rapid Application Development or Object Orientation;
- As part of a 'software factory'.

2.2 Starting Conditions

The basis for TSite is a structured test approach. This means that the following four 'cornerstones' are well organized:

- In a test ***life cycle*** it is defined what activities have to be performed and when, who is responsible and what will be the result or product.
- Standard ***techniques*** are developed to support various activities in various phases. Examples are: techniques for determining test strategies, for specifying test cases (test design), for reports on quality or coverage, for test management techniques etc.
- The ***infrastructure*** is built in order to create a test environment for test execution. It consists of the hardware and software platform, the network, business applications and all associated databases and tables, tools etc.
- An ***organization*** is defined, which includes an organizational structure, defined test functions for personnel, training, and many procedures and tools to support communication and secondary processes such as test management, change control, fault report management.

So far TSite is not much different from any well organized test project. What makes it different are the extra requirements on the test process:

- Reusability
  It is not just for one project, but it must be used again and again for many more releases to come.
- Flexibility and adaptability
  It is not just for this one system or application under test. It must be able to be used for testing other systems, with a minimum of extra effort.

- Ready to go
  In classical test projects preparation takes a lot of time and effort. TSite enables us to start test execution as soon as possible.

3 Main Components of TSite

The extra requirements on the test process have specific consequences on how to organize and implement the structured test approach. The areas that are most affected are test automation, infrastructure and human resource management. Some typical characteristics are given below.

3.1 Test Automation

Automation is essential in order to be able to test cheaper, faster and better. Many tools are available to assist in various testing tasks, such as test project management, test specification, test execution, coverage measurement, problem management etc. In TSite the test process is progressively automated as part of a test process improvement programme.

Of all testing tasks, test execution is the one that often gets top priority to be automated, because it is on the critical path of IT projects. It is not surprising then that test execution tools attract much attention. A large range of record and playback tools are available, with increasingly attractive (graphical) user interfaces and exciting technological possibilities. Unfortunately, these expensive tools often end up as shelfware. The reason for this usually is that organizations have too high expectations of the tool, and that they underestimate the knowledge and skills required to use them properly.

For a long time, organizations required that test tools should be extremely user friendly, so that end users should be able to work with them. This suggested that end users should be able to automate tests. The result was often poor. The first step to successful test automation is to recognize that it takes an automation project to achieve it. It requires automation skills and automation techniques. The technique of 'data driven testing' is becoming widely recognized as a good approach for creating maintainable test suites, the (record & playback) test tool providing the development environment.

3.2 Reusable and Multi-Purpose Infrastructure

In TSite we need a ready-to-use infrastructure, that has enough flexibility to be used for different systems.

- All business applications that interact with each other are installed and configured on the same infrastructure. Different configuration settings, including user authorization files and shared tables, are saved to enable separate testing of individual business applications as well as integration between selected applications. Each test has its own initial settings saved (such as specific database contents) and is automated as much as possible.
- Test tools are installed and configured in such a way, that different tests can make use of them without messing up each other's results. In order to achieve this many things can be arranged or created: standard directory structures defining where to look for test input and where to store the output, reusable scripts for the performance of standard activities, such as logging in on a remote server or collecting certain types of test output, etc.

- TSite maintains standards and templates for all test deliverables, such as test strategies, test plans, test specifications, error reports. No time is wasted on inventing new ways to carry out old things.
- When TSite is requested to test a completely new and different application, most components can be used immediately. Procedures, templates, fault report, database, probably do not need to be adapted. Tools can be employed very quickly, because configuration settings and structures are ready and people are already experienced in using them correctly and efficiently. Under adverse circumstances, however, it may occur that a hardware platform must be changed heavily or a complete new one must be installed.

3.3 Human Resource Management

TSite has long term interests. It does not exploit people to successfully finish this one project on time, but it keeps in mind that it wants to stay in business for a longer period of time. Its staff should be encouraged to stay and improve their skills. Experience and knowledge should be preserved and shared, and applied in future projects for the improvement of the performance of TSite.

TSite has specialists for test tools used, in order to ensure maximum advantage of each tool.

TSite personnel is trained in all working procedures and techniques, such as developing test strategies, participating in problem management meetings, producing reports. This causes them to be able to start very quickly with a new test job.

4 Implementing TSite

4.1 Scope and policy

Essentially TSite is a line organization rather than a project organization. Line management in TSite must ensure that all resources (people and machines) are used efficiently. Test contracts must be acquired in an active way. All employees should realize that TSite must be competitive. It should not be obligatory to use TSite for test activities; IT projects can also organize tests themselves. However, they should not want to do this since TSite can do it faster, better and cheaper.

One TSite works most efficiently for one type of organization (banking, insurance, government, etc.). The support of more organizations costs more in terms of application platform and required business knowledge. The scope may also be one of the platforms (e.g. a TSite for mainframe applications only), test levels (e.g. functional acceptance testing), quality characteristics (e.g. usability testing), subject matter (e.g. billing, savings accounts, branch applications).

Although TSite is an independent organization, close communication with other parties (system development, users) is essential to know where to focus test attention and how to operate certain system functions.

4.2 Management of Change

The word 'TSite' and its synonyms turn out to have characteristics of a buzz-word. Managers want a test factory in their organization, without knowing the feasibility or realizing the consequences. The implementation of TSite must be treated as most other fundamental organizational changes: carefully.

To start with, does your organization have the right problem for this solution? Perhaps the problem more heavily lies in uncontrolled changes of the specifications, or improper use of tools, or testers messing up each others' test results. In many organizations the first steps in solving problems with test processes may well involve the introduction and optimization of an approach to structured testing. Process improvement methods can be of help in analysing which steps to take first. When an organization has successfully implemented structured testing, but finds out that for each project again it costs too much time and money to set up the test organization, TSite is a suitable next step to take.

The initial phases are crucial. It is wise to start small with a pilot project. Take time to learn. Standards and structures will change frequently in the beginning. On the other hand, enough test work must be acquired to avoid that people run out of work after a few months. Management commitment is essential to survive this difficult phase.

When all goes well and everything works in TSite, beware of the monster of bureaucracy. Employees must remain alert and maintain their competitive mentalities. If they don't, customers might look around for other test possibilities.

5 Examples of TSites

Some examples of existing test organizations are discussed here. We try to answer the question whether they can be considered genuine TSites, i.e. whether or not they fullfil criteria as discussed above.

Various usability labs exist, which are TSites concentrating on testing the quality characteristic Uability. They are permanent organizations, based on structured testing, with high test maturity, working for more than one customer, complete life cycle, continuous process improvement.

A test centre at a large Bank started out as an organization offering infrastructure to customers for them to perform their system tests on. Test automation was taken care of. At its start we could not consider this centre to be a TSite, since mainly the infrastructural component was taken care of and no test expertise was offered. Since then it has developed into an organization more actively involved in testing activities and focussing more on the complete life cycle. The test centre concentrates on testing applications for the Bank's branches. It is a permanent organization, now based on structured testing, with high test maturity.

A test service centre of a Telecom company was a TSite before knowing it. A large acceptance test team was concerned with tests of a number of vital business applications. However, the results were insufficient, costs were too high, and resources were employed inefficiently, communication with the external development team was inadequate. Test process improvement is now being applied in order to increase effectiveness and efficiency.

The IT department of a company for social insurances has recently decided to phase out the activities of their competence centre testing. They consider their TSite as a step to higher system test maturity in the organization. It used to be a permanent organization, based on structured testing, with high test maturity, working for various projects, carrying out all phases of the testing life cycle. Now it is a facilitating department planning and scheduling test teams. Their future is uncertain.

The Dutch Railways have been restructuring their testing framework in relation with their Y2K testing activities. The efforts that were put into the Y2K test centre at first may consolidate into a TSite later, although obviously attention originally was focused on short term activities.

A test centre of a Government department simply started out as the wish to have workplaces available elsewhere for a very small test team working on one information system. This itself, of course, is no TSite. Because of their isolation, however, they were

able to fully concentrate on structured testing in a good infrastructural environment. In doing so test maturity in the organization now is increasing. The centre may become a permanent one and may start to work for more than this one information system.

6 Conclusions

In general it can be concluded that the most successful TSites exist where test expertise, infrastructure or resources (or a combination of them) are scarce. Test maturity and structured testing are crucial in an organization for a TSite to flourish. Management support and commitment are vital.

The scope of TSite may be small at the start but should not stay too small later.

Care should be taken that test knowledge and expertise is shared in TSite although testers may work on different projects. At the same time it is wise to invest in knowledge of the subject matter of business applications under test.

However hectic testing may be, the completion phase of the testing life cycle should not be omitted. On a larger scale TSite should be aware of the threats of 'going concern'.

As yet there is evidence to be able to give a positive answer to the question of whether testing in TSite is faster and cheaper. It should be noted, however, that to some extent the gains on one side (dedicated testing, reusability, ready-to-use) also imply losses on the other (more difficult communication with other parties, working according to fixed agreements). Quality of testing definitely becomes more constant and predictable in a TSite environment.

Deploying SQA in Very Small Projects

Quality Week 99

Jay G. Ahlbeck
Linda J. Lin
Jon S. Scharer
Margaret A. Stocking

Roger M. Records
Associate Technical Fellow
Boeing Commercial Airplanes

Agenda

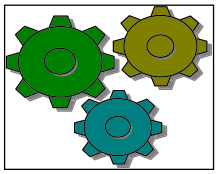
- What is the problem for small projects?
- What is the SQA goal for very small projects?
- What is a solution that works for very small projects?
- How is SQA deployed in very small projects?
- What does success look like for small projects?

What Is a Small Project?

Six to 24 Person-Months of Effort

Not Life Critical

Not Complex



SQA Deployment Challenge

SQA Body of Knowledge

Six SQA Staff

Provide SQA Installation Training

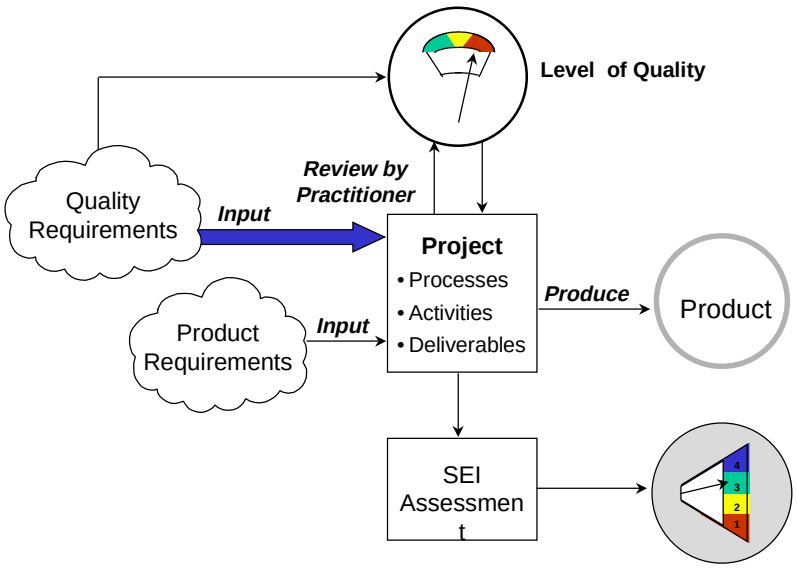
SQA Function Installed and Practiced

1,200 Software Projects

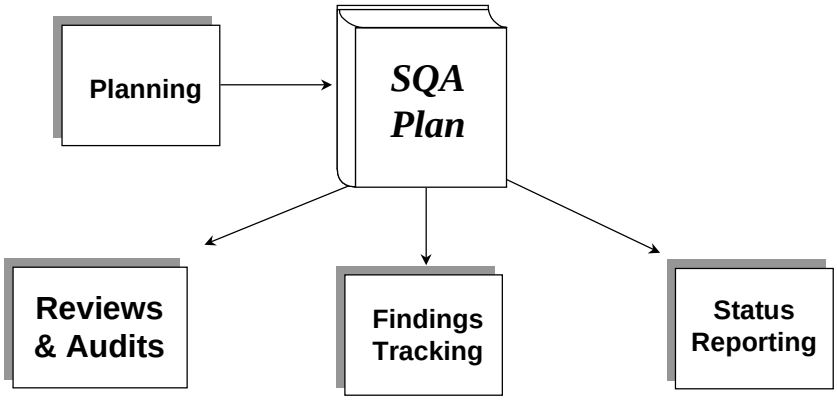
SQA Goal for Very Small Projects

- SQA function that can be accomplished.
- SQA activities that produce real value.
- Build quality in the software during development (not inspect it in at project's end).

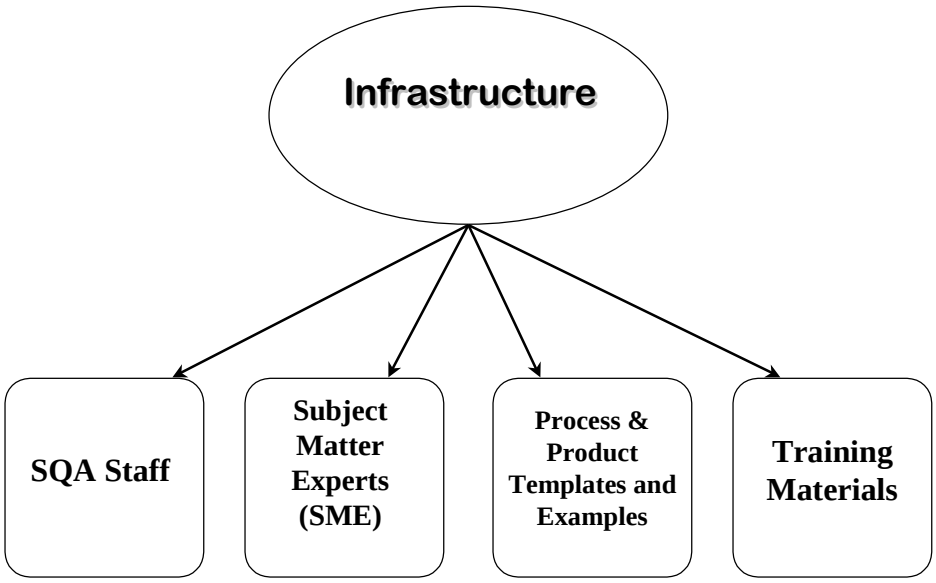
Recommended SQA Approach

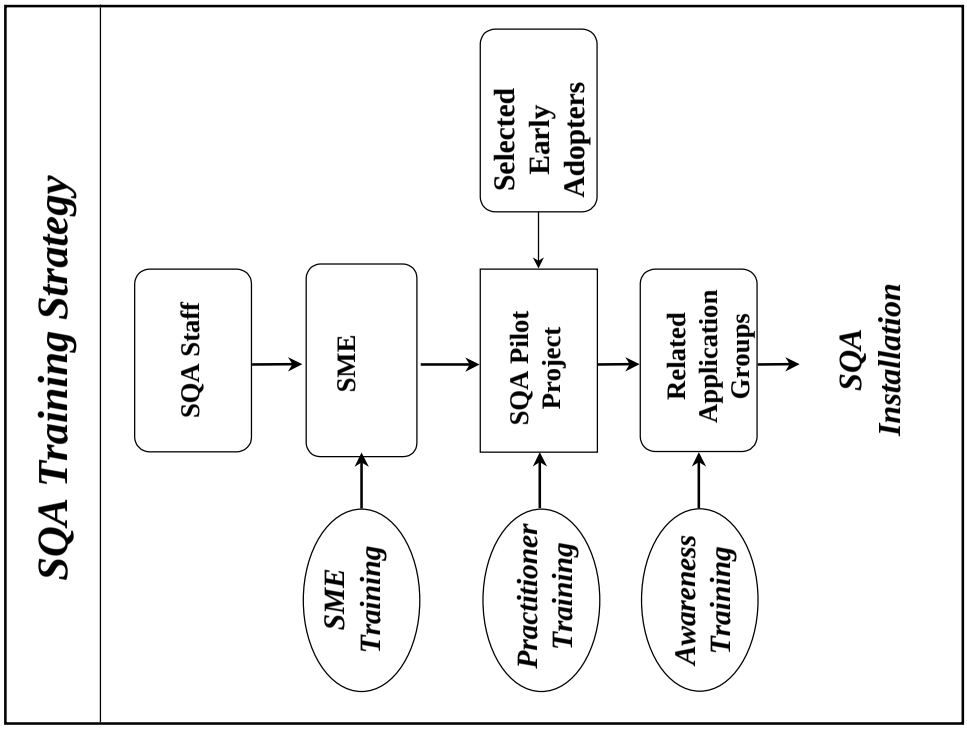


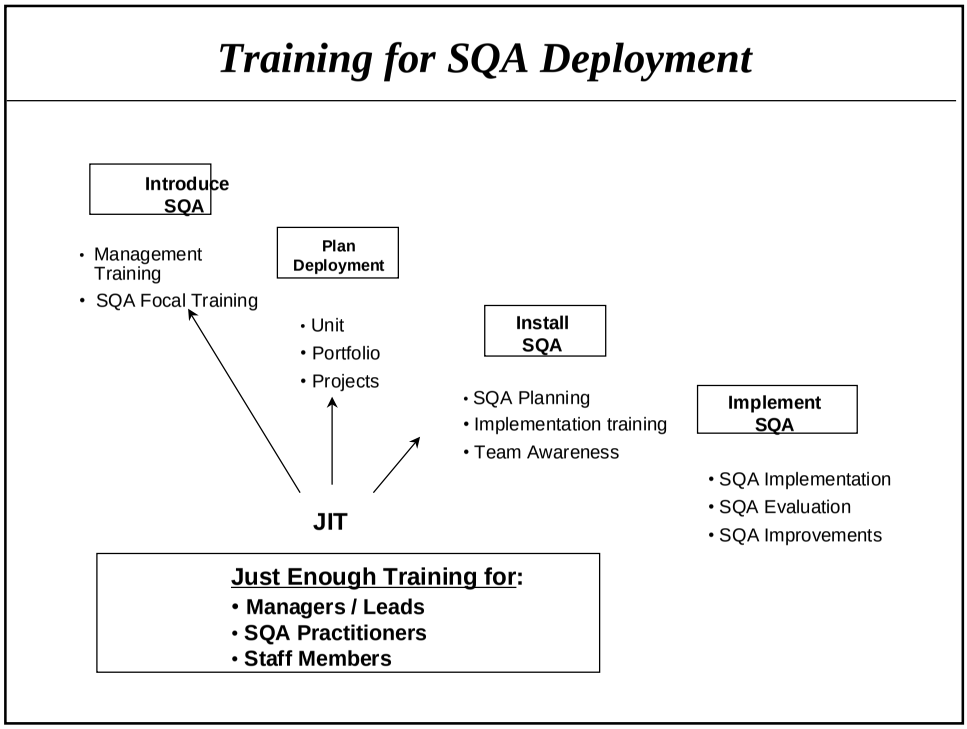
SQA Function

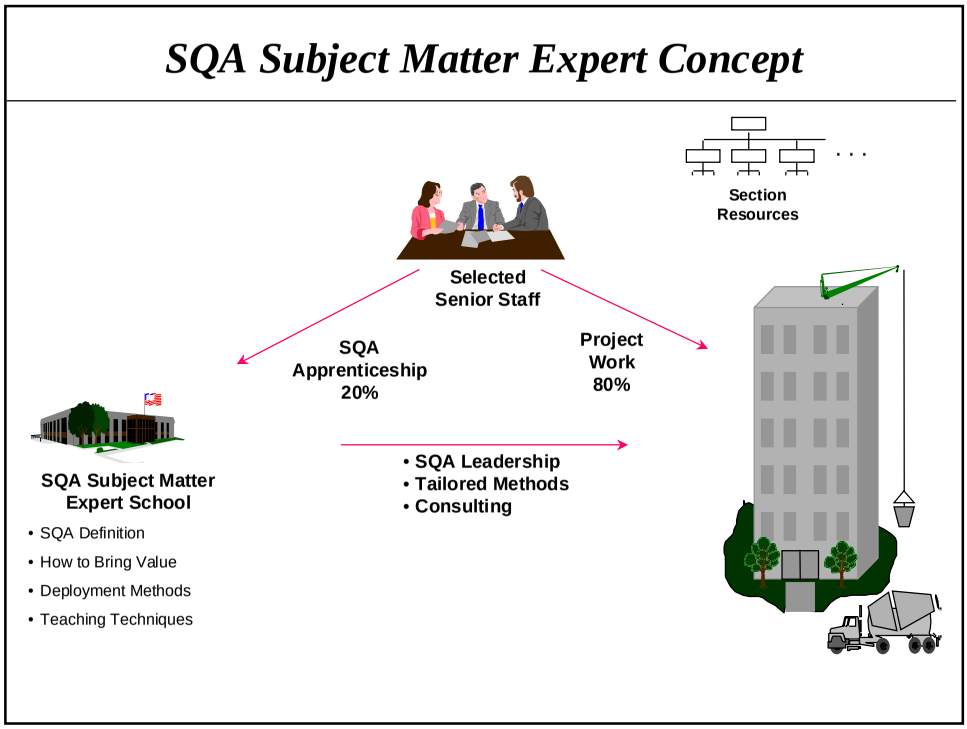


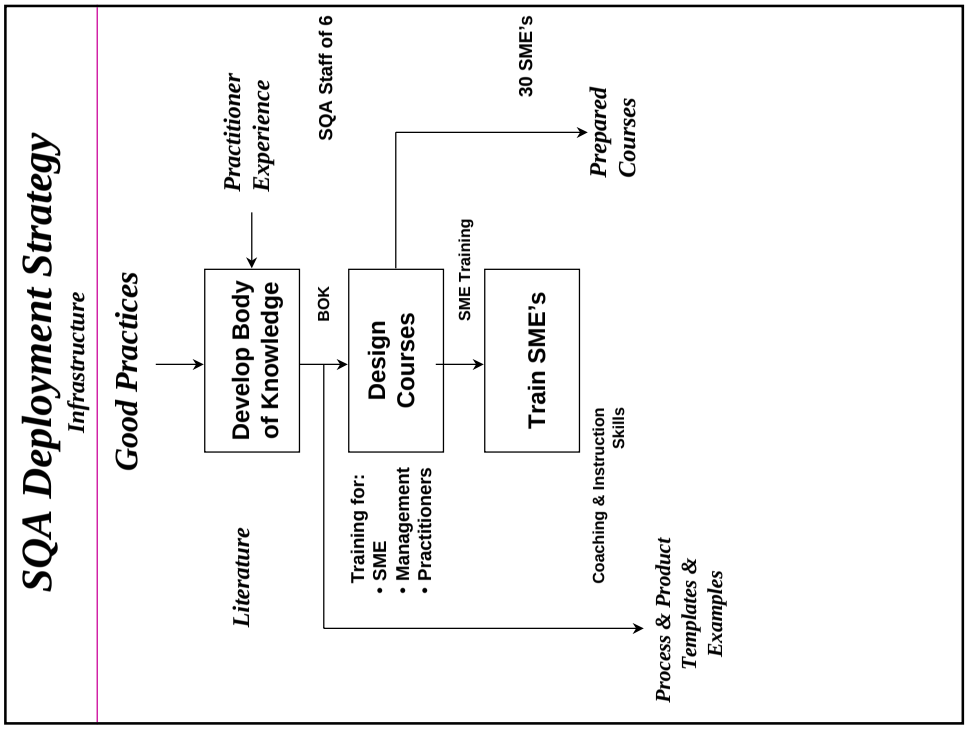
- Plan SQA activity and document in a plan
- Tailor and conduct reviews in accordance with the SQA plan
- Record observations and categorize by significance
- Report results to affected developers, process owners, and managers




SQA Training Strategy
SQA Staff
SME
SME Training
Selected Early Adopters
SQA Pilot Project
Practitioner Training
Related Application Groups
Awareness Training
SQA Installation

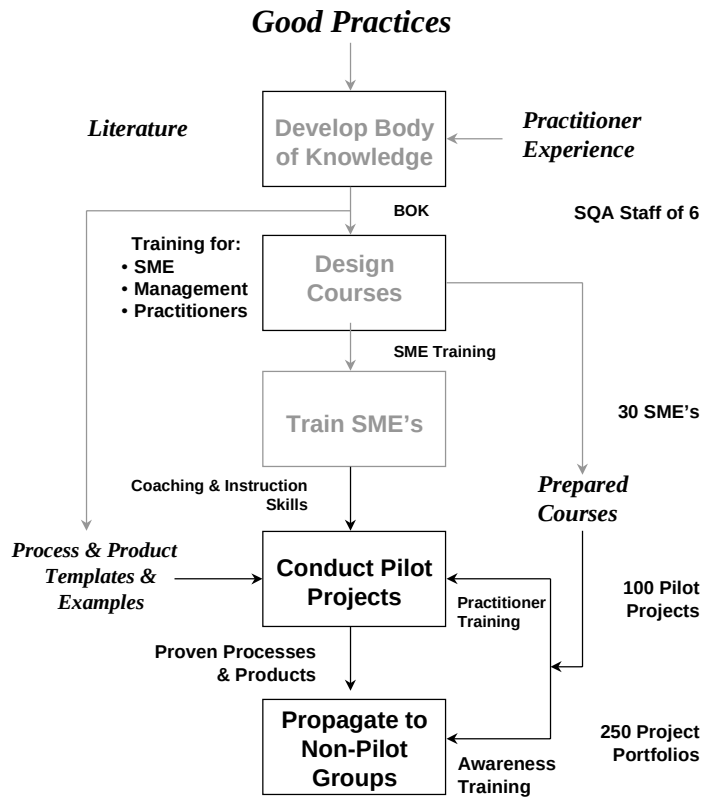

Training for SQA Deployment
Introduce SQA
• Management Training
• SQA Focal Training
Plan Deployment
• Unit
• Portfolio
• Projects
Install SQA
• SQA Planning
• Implementation training
• Team Awareness
Implement SQA
• SQA Implementation
• SQA Evaluation
• SQA Improvements
JIT
Just Enough Training for:
• Managers / Leads
• SQA Practitioners
• Staff Members


SQA Subject Matter Expert Concept
Section Resources
Selected Senior Staff
SQA Apprenticeship 20%
Project Work 80%
• SQA Leadership
• Tailored Methods
• Consulting
SQA Subject Matter Expert School
• SQA Definition
• How to Bring Value
• Deployment Methods
• Teaching Techniques

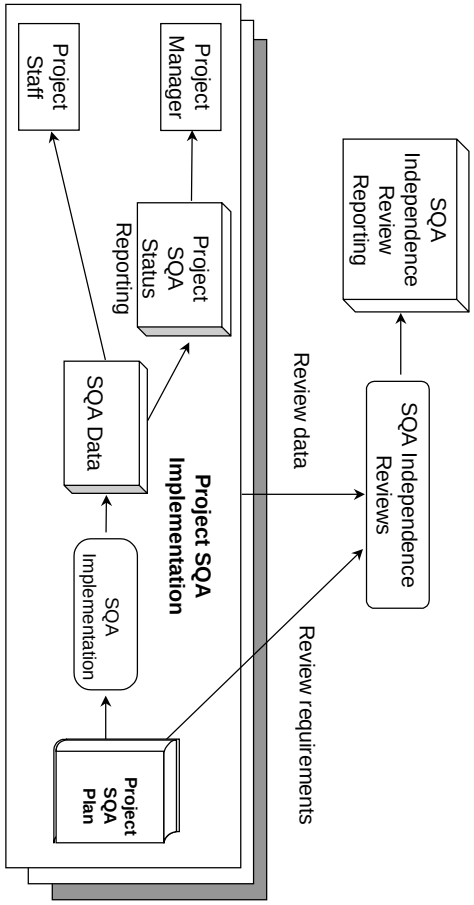

SQA Deployment Strategy
Infrastructure
Good Practices
Literature
Practitioner Experience
Develop Body of Knowledge
SQA Staff of 6
BOK
Design Courses
30 SME's
Prepared Courses
SME Training
Train SME's
Training for:
• SME
• Management
• Practitioners
Coaching & Instruction Skills
Process & Product Templates & Examples

SQA Deployment Strategy

Adopt and Tailor



SQA Independence for Small Projects



*"Adopt" Definition**

Definition:

To take up and practice or use as one's own

Migrate tailored, validated practices from pilot effort → **Understand and recognize the benefit potential** → **Accept and agree to support**

Tailored, Validated Practices + Benefit Recognition
- *Organizational*
- *Personal*

+ Decision
- *Organizational*
- *Personal*

= SQA Practices Installed and Used

*Adapted from David Duchesneau; used with permission.

*"Tailor" Description**

Definition:

To customize for a specific situation

Map existing practices to SQA needs → **Map available resources to shortfalls** → **Tailor / add processes, methods & tools as necessary**

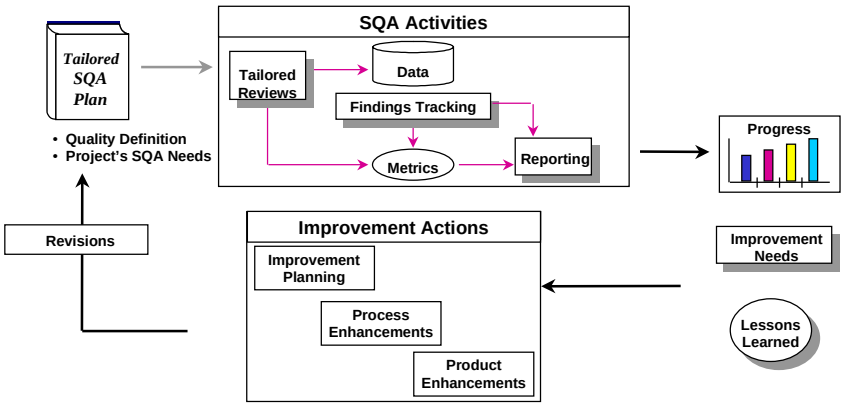
Existing Capabilities + Off-the-Shelf Capabilities + Augmentation of Capabilities = Tailored SQA Practice

*Adapted from David Duchesneau; used with permission.

How Is SQA Deployed In Small Projects?

Adopting Proven SQA Practices, Then by:

1. Tailoring an SQA plan template or example
2. Selecting and tailoring reviews to collect required data
3. Collecting and reporting value-added data to decision makers
4. Effecting SQA improvements



Success Criteria

- **Achieve Quality Goals**
  - SQA Function that can be accomplished
  - SQA activities that produce real value
  - Quality Built-In, not inspected in
- **Continuous Improvement of SQA Function**

What Out Customers Say

- ***"Our best training value for 1998 was the SQA Subject Matter Expert training."***

  Third Level Software Development Manager

- ***"You have . . . Helped us to change our fear and apprehension of doing SQA into one of a 'we can do this'. I believe this is directly a result of the quality of your products, processes and people."***

  SEPG Member

- ***"We appreciate your common sense approach and determination to have SQA make sense to us — and the way we do business."***

  Small Project Personnel

Deploying SQA in Very Small Projects

Roger M. Records, Associate Technical Fellow

Jay G. Ahlbeck, Linda J. Lin,

Jon S. Scharer, Margaret A. Stocking

Boeing Commercial Airplanes

Summary

We have demonstrated that SQA can be very practical and value-added (great ROI) in a small project environment. To deploy SQA in one or two person projects requires the definition of an SQA function that can be implemented with minimum resource investment, yet adding value to the small project. An infrastructure of instruction from Subject Matter Experts, model plans, processes and reviews that can be easily tailored to the small project needs has proven to be a feasible implementation.

1.0 What is the SQA Goal?

There are three quality goals for all software projects that are identified in the following paragraphs. Each goal is followed by a discussion of how that goal is made practical in a small project environment.

1.1 Goal 1: Define an effective SQA function.

There is a need to install an SQA function for the project that can be accomplished with the resources available.

For the small project, the limited investment of resources available for SQA requires a function that has been simplified. Simplification accomplishes two objectives: 1) Minimize the training required to implement the function. 2) Minimize the implementation investment. Simplification carries a challenge and a risk. The challenge for defining this function is to create a simple function that is not trivial. The primary risk in defining the SQA function is that it will be designed to satisfy some external criteria (such as SEI Level 2) but have no real value to practitioners. This is a formula for shelfware. It also is the motivation for the second goal.

1.2 Goal 2: Include SQA reviews that produce value.

A value-added SQA function must be based on SQA reviews that are deemed high leverage, resulting in real benefits for the project personnel, for their customers, and for the project's management.

One way of minimizing resources required for the small project's SQA is to constrain the scope. The criteria for scoping the SQA function is the requirement for real value in the reviews that are selected. The key to producing real value is to establish the correct criteria for the SQA function. If the SQA activity doesn't add value in the view of the small project developer, it is not worth doing. There are two components to consider. First, where are opportunities to add value? This is project specific, but it is not a difficult question for most small project developers to answer, since problem areas are usually known entities. The second component is to match an SQA activity with the real value opportunity. The small project has very little resource to create SQA reviews to match project needs. A workable SQA solution must provide real-valued reviews at a very low cost. Real value for the small project can be thought of as producing reduction in development cost or time, insuring the specified functionality is delivered, or otherwise increasing the satisfaction of the customer.

1.3 Goal 3: Build quality in.

The responsibility for quality lies with the developer (not a special SQA group) which requires building quality in during development, not inspecting for quality at the project's end.

Having an SQA function that is practical within the small project's budget and that still delivers real value to the project are both ambitious goals, and they are only going to be attained with the third goal - by building quality in, not inspecting for quality at the project's end. Building quality in means performing quality reviews on both products and processes as early in the development or maintenance cycle as possible. For the small project this also carries the implication of reviews performed by the developer rather than an external SQA group.

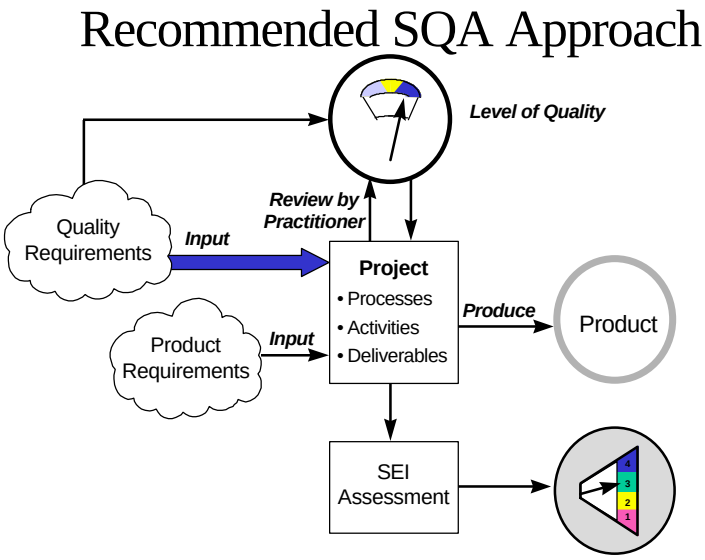


Figure 1. Building Software Quality In

As illustrated in Figure 1, the developer needs to know not only the product requirements, but also the process and deliverable quality requirements for reviews to be effective. There should be no quality surprises at the end of project for the software practitioner since the quality requirements are known while the process is being followed and the product is being produced. Quality reviews are performed during each phase of the software life cycle, giving just-in-time assessment of each life cycle deliverable from requirements specifications to design to code and test. This, in turn, helps attain the previous two goals (practical and value-added) by producing timely findings which, when corrected, will help project personnel avoid rework. Eliminating waste is where the real value is produced and where the time resource for the next SQA review is generated. A by-product of performing timely reviews is that the organization can be assessed with higher levels of both product functionality and process maturity.

2.0 What is the Problem for Very Small Projects?

Software developers live in a world of limited resources and high productivity demands. Everyone wants to create high quality software that functions as it was designed and delivers features which meet all the customer's requirements. So in this environment of "do it well, but be sure you are on time and within budget", what's a developer to do when the SQA folks come on the scene? If the developer is working on a very small project (6 to 24 person months, not life critical, and not highly complex), it is usually the case that any SQA activity is perceived as having little value to add to the project. The developer (or manager) asks, "So, why try to make room on my (already

The balancing issue is the desire of software developers to produce quality products. In a world of downsizing, most software developers want to be viewed by management as valuable employees. In our environment of advancing technology, most practitioners also want to be positioned for career advancement. If SQA offers potential to improve product quality, personal performance, and increase productivity, most practitioners are willing to consider change, but only if the change is practical.

3.0 A Proven Solution for Very Small Projects

Our proven SQA solution is based on 1) an infrastructure for small project support and 2) a workable SQA function for the software practitioner to utilize.

3.1 SQA Function

First, consider the SQA function that can be implemented efficiently for small projects. This function will require a minimal investment in training and minimal project resources to install and implement.

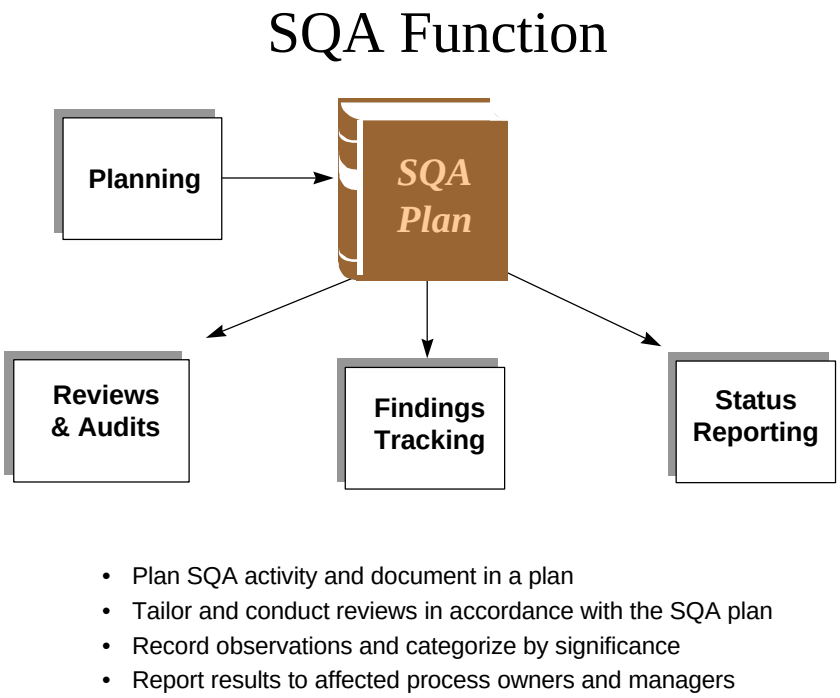


Figure 2. SQA Function for Small Projects

Figure 2 illustrates the four essential components for any SQA function. The planning activity consists of creating an SQA plan that can be completed by the small project personnel within a very short time allocation. This requires constraining the number of reviews as well as selecting reviews that will add value. Care must be taken to only collect data from the review that will be relevant to the small project personnel and that will address the quality goals. In addition to performing the reviews, the processing of review results (findings) must also be planned. A simple (but non-trivial) method of ensuring accountability for review findings is required. Finally, reporting the results of SQA activities must be planned to determine both the content of the reports and who the recipients should be. In a typical small project, SQA planning is completed within 20 hours of effort.

Once the SQA Plan is produced, the reviews specified must be defined and documented as well as performed. This requires some training for small project personnel. This training component is addressed below in the description of the supporting SQA infrastructure. The goal is to select and perform reviews to provide the collection of data that ensures that quality is present in each product and is being derived from each process used by the small project. The investment for selecting these reviews is reduced by using off-the-shelf examples that are tailored to the project's needs.

Processing and managing the data collected is the goal of Findings Tracking. Defects or problems identified in the review must be corrected. Corrective actions must be taken and a system of accounting or tracking can ensure that the problem found in the review is resolved. Once again, the criteria of simple, but non-trivial, tracking of the findings is important for the small project.

Finally, the results of the data collection and processing need to be documented for potential process improvement and to inform management that the SQA function is in operation. Decisions need to be made describing what kind of information to report and who to report it to. Status Reporting accomplishes this requirement.

3.2 SQA Support Infrastructure

In addition to the SQA Function, our solution for very small projects relies on a support infrastructure that consists of 1) a small group of trained Subject Matter Experts (SME)s, 2) a proven set of training materials and 3) examples and templates for the processes and products related to SQA to install and deploy SQA for very small projects.

3.2.1 Training Materials

Prior to deploying SQA in an organization with many small projects, our staff of SQA specialists defined the SQA body of knowledge for small projects, defined the SQA function which is used in the small project environment and developed the training materials The SMEs are recruited and given apprenticeship training.

In addition to SME training, instruction also needs to be distributed to address the needs of various project personnel. This must include instruction for management, the project SQA Focal and practitioners. Each student group has specific learning objectives. As shown in Figure 3, the training is customized to meet these specific learning objectives (just enough) and is delivered when it is needed (just in time). We usually formalize this training/coaching agreement in a Statement of Work, which is a contract for training and coaching services.

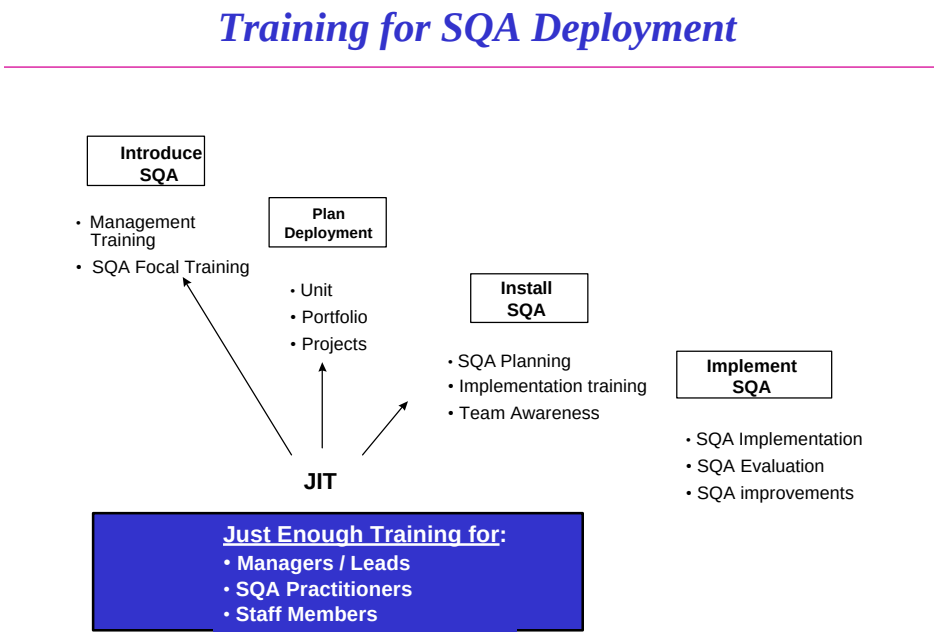


Figure 3. Implementing SQA with Just-In-Time Training

3.2.2 Examples and Templates

The second component of the infrastructure is a set of SQA templates and examples that are available to the small project personnel through their SME coach. Since the SQA function must be implemented with minimal resource investment, it is impractical to invent SQA plans, procedures or reviews. The goal for small projects is to obtain the examples and templates from the SME which are required to support the SQA activities for the small project. With minimal coaching, the example or template is then tailored to the project's needs.

Examples and templates for the SQA Plan are provided to the practitioners. They reuse most of the plan and customize the specific SQA activities and list of reviews to "fit" their project. In a similar fashion, examples and templates of specific reviews are provided and tailored. This enables the small project personnel to choose SQA reviews that match their need and offer the opportunity for adding real value. Finally, processes that support the SQA activities are selected from the set of examples and templates. These processes are also tailored, as needed, to meet the needs of the small project. Reuse and tailoring of plans, processes and reviews make the implementation of the SQA function cost effective.

3.2.3 SME Selection and Training

Given the constraints of budget and time, the small project personnel must have opportunity for a limited amount of formal training to accomplish each of the four SQA function components. A successful strategy is to minimize the formal training and provide access to a qualified SQA coach or organizational SQA expert SME to leverage that training into a working SQA solution. An SQA coach in our environment is a specialist from a central organization who also delivers training. A SME is a member of the software development organization who has received in-depth SQA training. In our experience a small number of SMEs are required for a successful SQA deployment. While the SMEs invest 60-160 hours to prepare for this role, the following benefits are realized:

1. This SQA expertise is made available to small project developers initially in pilot projects in which SQA is demonstrated to have real value. During the pilot project, the SQA function is installed for a specific application. Participation in an SQA pilot is part of the SME training.
2. The good practices developed during the pilot projects are then propagated to other small projects with a fraction of the training time for the SME. The propagation is one of adopting the proven SQA practices previously developed.
3. Becoming an SQA SME offers a new career enhancement opportunity for proven senior software developers. Expanding their set of software skills is a benefit for those participating. For most SMEs, this is not a bridge to a new career, but rather an opportunity to gain a better understanding of the complete software engineering environment.

4.0 How is SQA Deployed in Very Small Projects?

The process for deploying SQA, shown in Figure 4, begins with the development of an SQA Body of Knowledge (BOK), which is gleaned from SQA literature, good SQA practices and expert experience. The BOK is the foundation for SQA course design.

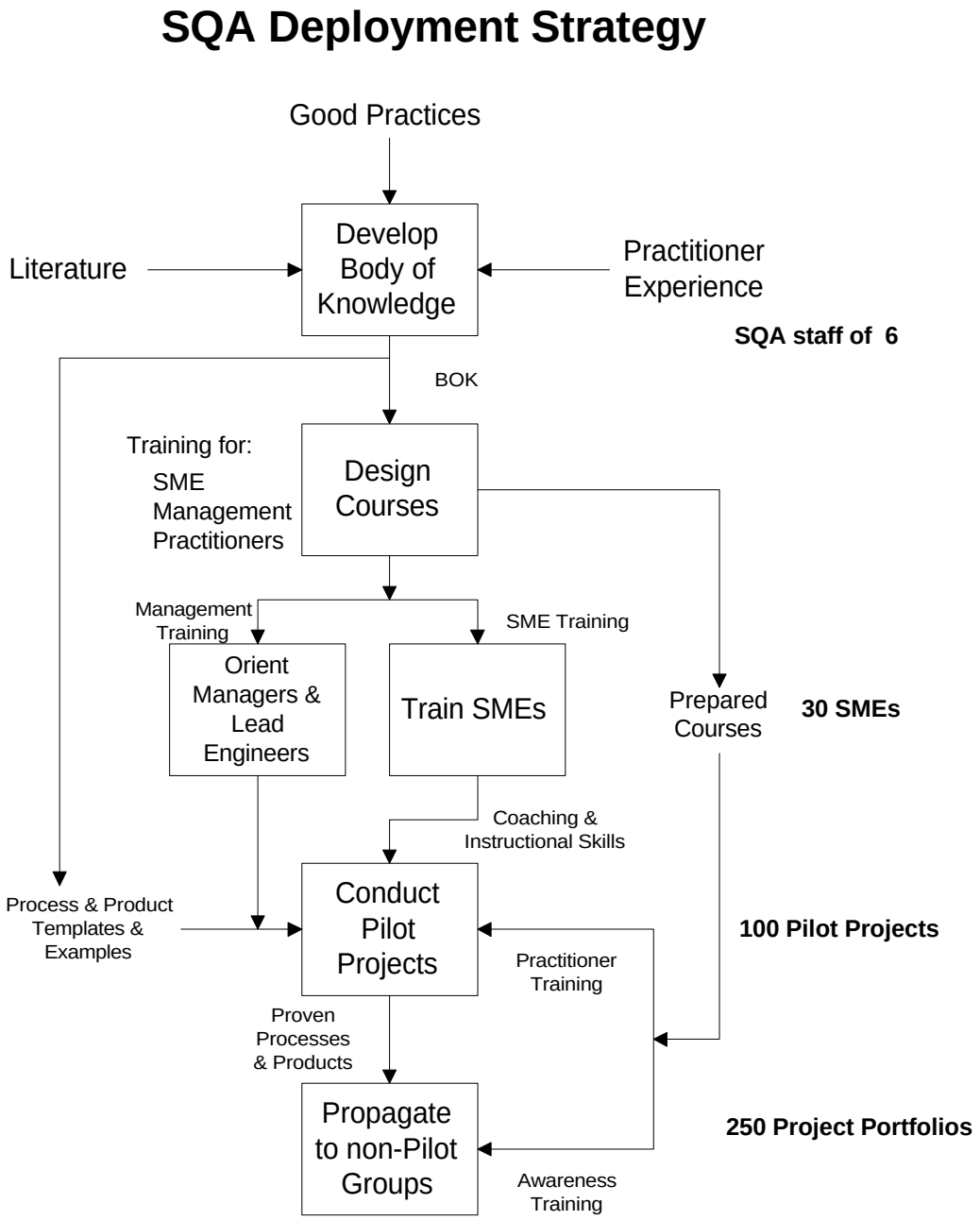


Figure 4. SQA Deployment Strategy

When the need for SQA installation surfaces, the training materials previously developed are used in a pilot with a small project. The SME participates in this first pilot project as an instructional intern to our SQA instructor. The follow-on SQA activities are

then lead by the SME with SQA coaches as consultants. The SQA plans, reviews and processes that have been adopted, tailored and validated during the pilot projects are then used for deployment in non-pilot projects within the same application group.

A prerequisite to the pilot activity is the SQA orientation for managers and lead engineers [see Figure 4 box "Orient Managers & Lead Engineers"]. A strong commitment from the project leadership is a necessity for SQA installation success. The manager and project leads first need to understand the three SQA goals. But they must also support the installation and implementation of the SQA function. This support comes in the form of scheduling and tracking SQA activity as an integral part of project planning. One manager, who successfully implemented an SQA function, admonished his development team following their SQA pilot with the following: "Now you need to do this SQA. We're not going to let this (SQA function) sit on the shelf!" The SQA activity became a topic for biweekly status reporting, in which the planned versus completed reviews were reported.

Deployment of the SQA function begins [see Figure 4 box "Conduct Pilot Project'] with a small amount of formal training (about 6-8 hours). This training is primarily focused on understanding the SQA Process Model. Participants in the pilot project define an area of focus for performing SQA reviews, and define the improvement goal for the pilot. Examples and templates that relate to the project's selected SQA pilot goals are introduced. The SME coaches the small project personnel during the pilot project. They tailor a plan for the SQA activity selected (based on the SQA function), and a procedure to implement the plan. Review topics are selected based on their ability to produce real value for the small project. These selected SQA reviews are developed in the form of checklists tailored from the set of examples. Practitioners are given practice in performing SQA reviews on their own SQA products and procedures.

During the pilot, the SQA function is installed within the project. All four components are developed. Tailoring of the SQA Plan is completed, tailored SQA reviews are performed, the findings generated during the reviews are recorded and corrective action (when required) is taken. Reports are generated using tailored versions of the report examples provided. Practitioners are encouraged to consider who else in their organization might use the materials developed during this pilot. The answer to this question may require that the attributes of the piloted materials be documented to facilitate judgment for use in propagation. This is the first step in utilizing the SQA function in non-pilot groups.

At the end of the project pilot the results of implementing the SQA function and related training are evaluated. Lessons learned are captured. There are two products from this evaluation. The instruction given may require enhancement. Change requests are generated by the SME and given to our SQA instructional developers. In addition, changes may be generated for the newly installed SQA function for the small project. These would be recorded as SQA findings for the reviews completed during the pilot project. Change requests could be written (if the small project has a Change Process in place) or changes can be described in the findings log.

During deployment training the practitioners are introduced to the concept of Quality-Built-In. Building in quality represents a shift in thinking for many with experience with "traditional" SQA methodology. The developer (initially assisted by either the SME or a unit SQA focal) performs the reviews for small project products and processes. Foundational to this approach is the notion that responsibility for quality belongs to the practitioner, not to someone "from the SQA group". This may raise a concern about the potential conflict of interest (does a developer want to log his/her own defects?) Our approach has been to emphasize the value of defect detection as related to the developer's desire to produce high quality software. Finding and logging defects early in the development is very beneficial for small projects and contributes to reduction in re-work.

An independent review of the small project's SQA activity is still important and is accomplished by having an individual knowledgeable in SQA and independent from the project's managerial structure perform an Independence Review. As shown in Figure 5, the Independence Review consists of inspecting the small project's SQA Plan and verifying compliance by looking at populated checklists, findings logs and action item artifacts such as change requests. So, for the small project, the key question to verify SQA independence is, "Are you performing the SQA activities called for in your SQA Plan, and do artifacts exist to support verification?" The product of the independence review is a report to project and organization management that gives the status of the small project's compliance to their SQA Plan

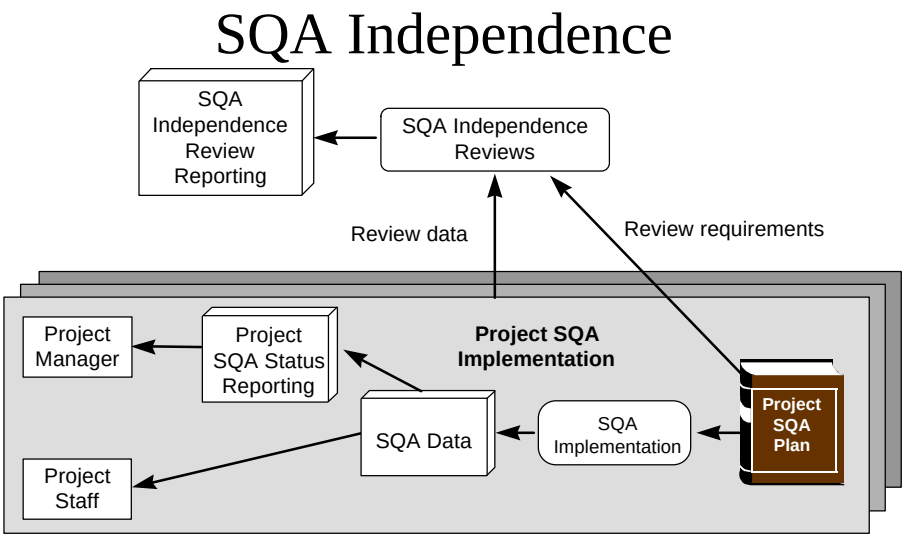


Figure 5. Independence Review for Small Projects

For readers interested in more detail regarding the deployment activity, the following paragraphs are offered to define our approach in propagating the SQA function to non-

pilot groups. For those who are not interested in such detail, paragraph 5.0 would be the appropriate place to resume reading.

At the conclusion of the pilot project, the SQA artifacts and lessons learned are shared with other small project personnel who perform similar work [see Figure 4 box "Propagate to non-Pilot Groups]. Important artifacts include the tailored SQA Plan, selected tailored SQA reviews, any processes documented or tailored from templates, and pilot project status reports. The SME has an opportunity to provide SQA implementation training for the personnel of peer projects needing an SQA function. The thrust of this training is SQA awareness and exposure to the SQA function that has been implemented within the pilot project. Additional training will equip peer projects to adopt the "good practices" developed during the pilot into their project for SQA deployment. As shown in Figure 6, validated SQA practices from the pilot are presented to a group of new practitioners. The benefits of using these practices are enunciated for "buy-in" by the organization and the individual practitioners. When value is recognized, the training required to install these proven processes and products is given by the SME, with additional mentoring and coaching as required.

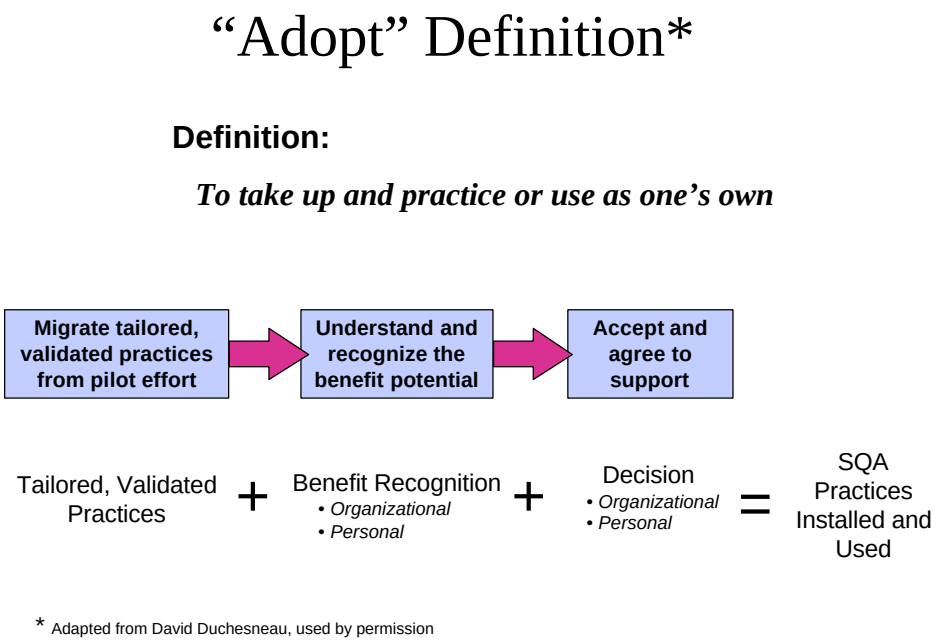


Figure 6. Define Adoption of SQA Processes

A secondary benefit is that adoption is viewed by peer projects, not as a top down edict from management, but as implementing what has been proven to work by other practitioners who are working on similar applications.

Once the SQA practices have been adopted by peer small project application groups, the related SQA plans, reviews and reports must be revised to reflect the specific quality concerns of the new application group. Additional tailoring of the SQA artifacts will satisfy this requirement. The tailoring, as shown in Figure 7, consists of reusing the set of SQA plans, reviews and documented processes.

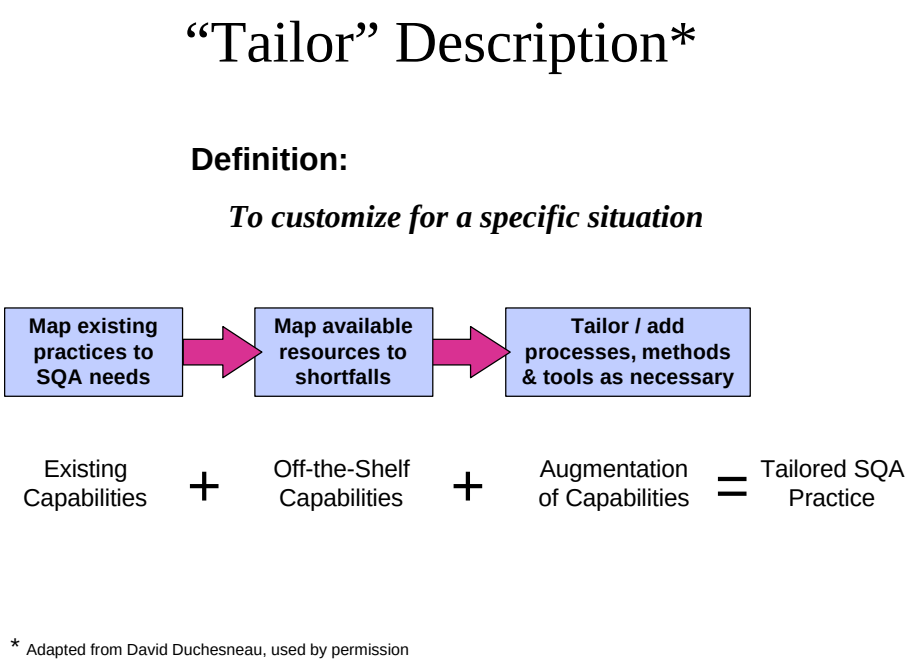


Figure 7. Define Tailoring Software Templates and Examples

An additional feature of the training is to use the actual project artifact (such as their SQA plan, a current process or existing report) for the training context instead of a hypothetical case or example. This creates more work for the instructor (in customizing the lesson), but produces real SQA findings and even produces project value during training activities. The students are able to see real value immediately and recognize that SQA does apply to <u>their</u> computing environment.

Following the completion of several pilot projects across several functional groups, an effort is made by a Software Engineering Process Group (SEPG) to collect the set of "good practices" and develop a set of "best practices" for utilization as a unit standard for performing SQA in a small project environment. Care needs to be taken to document attributes for the good practices that can be used to determine <u>where</u> these practices can be used. There is a tendency to think "bigger is better" when it come to a set of good practices. Our experience suggests that the opposite is true. A small number of good practices (screened by the SEPG) <u>will</u> be evaluated for adoption by a small project. But a large collection of unevaluated practices <u>will not</u> be considered.

5.0 What Does Success Look Like for a Very Small Project?

Successful deployment of an SQA function for small projects has two characteristics. First, the SQA activity is performed and has real value to the practitioner. It is our expectation that the performance of the SQA function will become a habit - a part of the regular project routine, even after the pilot is completed and the SME is no longer on the scene. Too often, the basic requirements of "practical" and "value-added" are ignored, which results in SQA plans, procedures and reviews that are, in reality, shelfware. Part of the job of the SME is to verify that the quality requirements are being met (that the small projects are using the SQA products and processes and that the result has real value to developers.) If either attribute is missing, modifications to the plan, the procedure or the review checklists must be made.

The second success criteria is that continuous improvement should result. If a product (such as an SQA Plan or a specific review) or a procedure is being used, someone will find an opportunity for improvement. This is a certainty for the first iteration of the product and the procedure. No one tailors a perfect product or process the first time! An unimproved SQA function is evidence of non-use. For the small project, non-use is one of the greatest risks. It frequently occurs in organizations which have a goal of receiving a sought after certification (such as SEI/CMM Level 2 designation) as opposed to seeking real value by installing their SQA function.

It should be noted that the order of performing and improving is also important. First we need to build a habit of doing SQA activity. It is only after a review is performed (sometimes several times) that the need for improvement is evident. When a specific review question is consistently answered with "not applicable," the reviewer will ask, "Why is this question being asked? We never do this in our group!" Improvement always follows activities that have been performed.

To ensure improvement to the SQA function, a self assessment should be performed. The appropriate time for initial self assessment is at the end of the SQA pilot project where the pilot's "lessons learned" are captured. The assessment is usually a joint effort involving the SME or SQA coach, the project manager and the practitioner(s) involved in the pilot project. Additional assessment of the effectiveness of the SQA function may be scheduled on a periodic basis or at the completion of a scheduled development phase or blockpoint.

Successful deployment of SQA into small projects will continue to follow the cycle illustrated in Figure 8. The observation should be made that a similar diagram would be used to describe SQA for a larger project. The difference is in key words like adopt and tailor. The small project SQA function is implemented with scaled down versions of all of the essential components of any SQA implementation. The challenge, as described in this paper, is to define the scaled version in a manner that preserves the quality goals.

How Is SQA Deployed In Small Projects?

Adopting Proven SQA Practices, Then:

By:
1. Tailoring an SQA plan template or example
2. Selecting and tailoring reviews to collect required data
3. Collecting and reporting value-added data to decision makers
4. Effecting SQA improvements

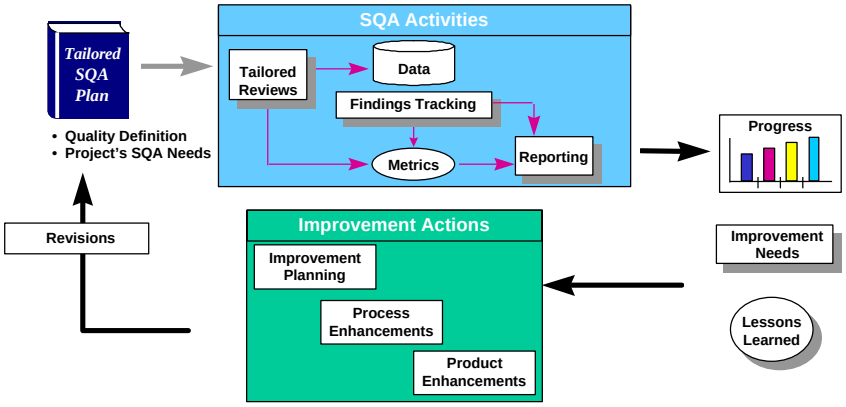


Figure 8. Successfully Deployed SQA Function

Our SQA SMEs continue to coach the piloted projects on an as-needed basis. Peer projects receive the awareness training required to allow the SQA function adoption (as suggested in Figure 4.) During this training they are asked for "buy-in" for the basic goals of 1) adopting an SQA function which can be accomplished and 2) that brings real value to their project. They are also instructed in the quality built-in perspective which puts responsibility for quality (and for performing quality activities) with the small project developer. Deployment of the SQA function continues as the piloted SQA plan and reviews are adopted and tailored by peer groups. The SME continues to mentor and coach as the SQA function is installed, practiced, and improved. The resource invested by the SME is smaller for an SQA function adoption than for a pilot project, and continues to decrease through each iteration of the cycle in Figure 8.

The final question about this approach for deploying SQA in very small projects is "Does it work?" We have collected feedback from management as well as practitioners. Perhaps the best answer is to let them speak for themselves.

"Our best training value for 1998 was the SQA Subject Matter Expert training"

Third Level Software Development Manager

"You have helped us to change our fear and apprehension of doing SQA into one of a 'We can do this.' I believe this is a direct result of the quality of your processes

SEPG Member

"We appreciate your common sense approach and determination to have SQA make sense to us - and the way we do business."

Small Projects Personnel

With this initial success, our goal will be to extend the application of the SQA function as our customer progresses in process maturity. Most of our instruction has been directed toward application of SQA in achieving repeatability in processes (SEI/CMM Level 2.) As our customers gain process maturity, we will focus our instruction on software product engineering and the associated key processes define in SEI/CMM Level 3. This will require Level 3 orientation for our SQA SMEs along with the production of plans, procedures and reviews that will support these key processes. We are confident that the infrastructure currently serving the needs of very small projects will be successfully extended to meet these new requirements.

Size *Does* Matter: Continuous Size Estimating and Tracking

by

Mike Ross
Quantitative Software Management, Inc.
5013 W. Vogel Ave.
Glendale, AZ 85302
(623) 435-9863 (phone) (623) 915-3351 (fax)
mike_ross@qsm.com
http://www.qsm.com

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide QuantitativeSoftwareManagement® QSM®

The Status Quo

- **Estimate the size of the system at the beginning of**
- **If disaster strikes, reorganize the team, re-estimate the size, and re-plan the project**

REACTIVE BEHAVIOR

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide QuantitativeSoftwareManagement® QSM®

What is Size?

- Size is a measure of *work* -- the *physical* stuff the development team creates -- units of human thought / creativity expressed in some accepted
  - Source statements
  - Sentences
  - Dialog elements
- Size is *not* synonymous with *labor*
- Size is *not* synonymous with *functionality*

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

What Do We Know About Size?

- Size Drives Cost, Schedule, and Reliability
- Size Estimation is Probabilistic
- Knowledge Increases with Time
- Change is Inevitable
- Developers are Optimists

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

Estimate Periodically and Record the Data

Edit Estimate Data for ESLOC (CUM)

Start Date	to	End Date
Jan-96		Dec-97

Month	Low	Most Likely	High	Expected	Convergence
Jan-96	45000	60000	75000	60000	0.00
Feb-96	45000	60000	75000	60000	0.00
Mar-96	45000	60000	75000	60000	0.00
Apr-96	45000	60000	75000	60000	0.00
May-96	45000	60000	75000	60000	0.00
Jun-96	53000	62000	70000	61833	43.33
Jul-96	55000	62500	70500	62583	48.33
Aug-96	56000	63000	71000	63167	50.00
Sep-96	57500	63500	72500	64000	50.00
Oct-96	58000	64000	74000	64667	46.67
Nov-96	59000	64500	76000	65500	43.33
Dec-96	60000	65000	78000	66333	40.00
Jan-97	62000	66500	81500	68250	35.00
Feb-97	65500	68000	84000	70250	38.33
Mar-97	67500	70000	86500	72333	36.67
Apr-97	70500	73000	88500	75167	40.00
May-97	72500	75000	90000	77083	41.67
Jun-97	75500	78000	91500	79833	46.67
Jul-97	78500	81000	92500	82500	53.33
Aug-97	82500	85000	93500	86000	63.33
Sep-97	86500	89000	94500	89500	73.33
Oct-97	89500	92000	95000	92083	81.67
Nov-97	91500	94000	95500	93833	86.67
Dec-97	92500	95000	96000	94750	88.33

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

Express Estimates as Ranges

- **Estimator's Inputs:**
  - **LOW VALUE: 1st percentile**
  - **HIGH VALUE: 99th percentile**
  - **MOST LIKELY VALUE: 50th percentile (best guess)**

3-point Estimate

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

Adjust for Bias

$$\text{Expected} = \frac{\text{Low} + 4 \times \text{Most Likely} + \text{High}}{6}$$

$$\text{Standard Deviation } (s) = \frac{\text{High} - \text{Low}}{6}$$

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

Look for Convergence

$$C_t = 1 - \frac{H_t - L_t}{H_{Baseline} - L_{Baseline}}$$

- **0% convergence = baseline uncertainty**
- **100% convergence = total absence of uncertainty**

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

Control the Process -- Traffic Light Metaphor

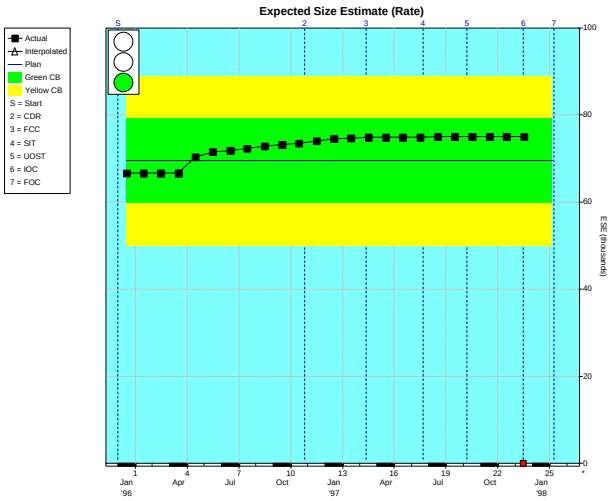
- **Green Status -- no corrective action recommended**
  - **default state**
- **Yellow Status -- determine cause, take corrective**
  - ***EXPECTED SIZE ESTIMATE* outside ±1σ *or***
  - ***SIZE ESTIMATE CONVERGENCE* outside ±1σ**
- **Red Status -- determine cause, take corrective action, replan project based on current estimate**
  - ***EXPECTED SIZE ESTIMATE* outside ±2σ *or***
  - ***SIZE ESTIMATE CONVERGENCE* outside ±2σ**

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

Example 1: Stable Sizing Expected Size Estimate Trend



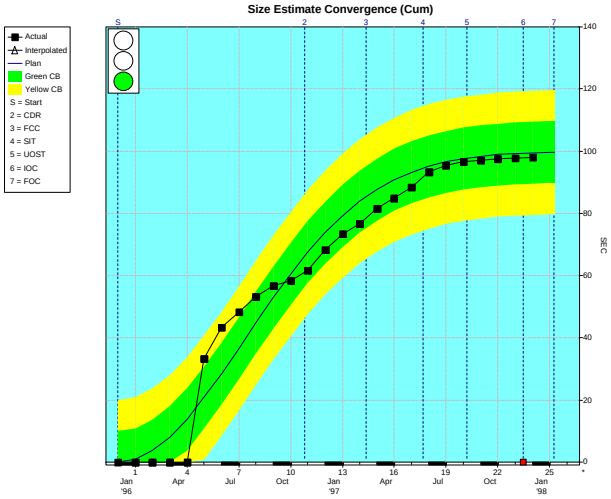
- **Within *±1 s* of replan**
- **Slope approaches *zero* early**

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

Example 1: Stable Sizing
High-Low Size Estimate Convergence



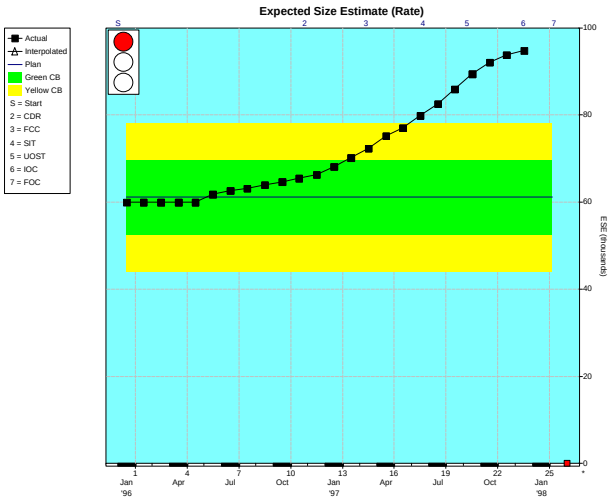
- **Convergence within ±1σ after replan**
- **Monotonically increasing**

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

Example 2: Unstable Sizing
Expected Size Estimate Trend



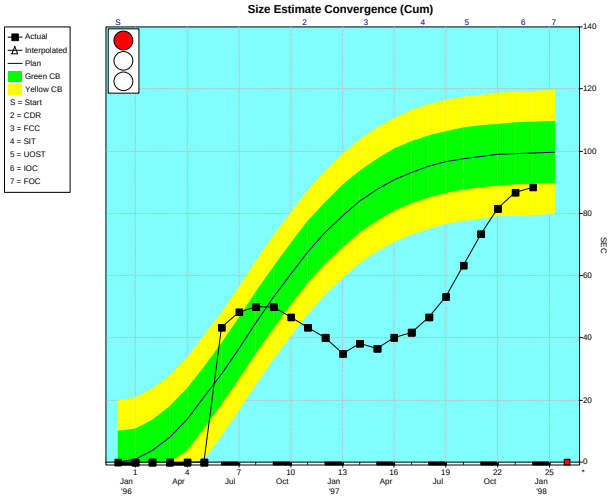
- **Expected size outside *±1 s* of replan after month 13**
- **Expected size outside *±2 s* of replan after month 16**
- **Slope *increasing* from month 7 through month 21 (only 3 months before delivery)**

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

Example 2: Unstable Sizing High-Low Size Estimate Convergence



- Convergence outside ±2σ after month 10
- Uncertainty increasing after month 9
- 8-month period of significant uncertainty after month 9
- Convergence never gets back on track

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

Conclusion: Why Continuously Update and Track Size Estimates?

- **Planning and Tracking**
  - Plans are based on size estimates
  - Estimated size is a key input to all software schedule/effort estimating relationships
  - Estimated size uncertainty is a key input to risk computations
- **Requirements Management**
  - Requirements are difficult to quantify
  - Requirements volatility/stability is directly related to changes in

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

Recommendations

- Success is when expectations are achieved
- Generally unreasonable expectations (rarely poor performers) lead to failure
- Base expectations on past performance
- Guard against expectations being set too high or
- *Measure*
- *Collect data*

© Quantitative Software Management, Inc. All rights reserved. Tracking Size Estimates, Slide

QuantitativeSoftwareManagement®

QSM®

Size Does Matter:
Continuous Size Estimating and Tracking

Mike Ross
Quantitative Software Management, Inc.
5013 W. Vogel Ave.
Glendale, AZ 85302
(623) 435-9863 (phone) (623) 915-3351 (fax)
mike_ross@qsm.com
http://www.qsm.com

Abstract. Estimating the size of a software system is a critical development process activity. Not only does size impact the *technical* solution; it also impacts the *project management* solution. It is therefore insufficient to estimate size only once, at the beginning of the project when the least is known about the system being developed. This paper describes a quantitative process for managing the size of software development projects through continuous estimation. The process includes a probabilistic approach to estimation coupled with tracking and assessment of trends to determine whether or not stability exists. This process has direct applicability to the SEI CMM Level 2 KPAs for Software Project Planning and Software Project Tracking and Oversight. It also serves as a Level 4 Quantitative Process Management tool for measuring the effectiveness of an organization's size estimating and requirements management processes. [3], [4], [5]

Introduction

Purpose

The purpose of this paper is to justify and describe a process for quantitatively managing the size of software development projects through continuous estimation.

Scope

The process described in this paper applies to all software development projects. Within the scope of the SEI CMM, this process applies to the Level 2 KPAs for Software Project Planning and Software Project Tracking and Oversight. It also serves as a Level 4 Quantitative Process Management tool for measuring the effectiveness of an organization's size estimating and requirements management processes. [3], [4], [5]

Background

> *Estimating size is the heart of the software-project estimating process.*
> Lawrence H. Putnam [6]

A casual glance at any software project estimating model illustrates the fundamental truth of the above statement. It follows, and historical data bears this out, that many software projects fail to meet cost, schedule, and reliability expectations because the actual size ends up being much larger than expected. Why, then, do so many organizations estimate size only once, at the beginning of the project when the least is known about the system being developed? These same organizations tend to revisit a size estimate only if, and more importantly when, the project is hopelessly out of control.

Intuition, Research, and Supporting Data

What is “Size”?

> *There are two, often-confused, notions of software size: one that relates to work and one that relates to functionality.*

The modern computing environment poses many challenges. Foremost among them is addressing totally new technologies. Trying to figure out how *big* computer programs are has challenged software engineering since its inception and is further complicated by the aforementioned dynamic nature of technology. [1]

Analysts have traditionally sized systems written in statement-oriented procedural languages expressed largely as text (large stacks of cards or reams of tractor-feed paper). Current technologies now take the form of more abstract representations such as diagrams, objects, spreadsheet cells, database queries, and Graphical User Interface (GUI) widgets. [1]

The secret to making progress in sizing these new environments is to identify the *unit of human thought* in the abstraction being used. Next, the organization must go through a calibration process, starting with projects for which the actual size can be determined in terms of that *unit of human thought*. The goal of calibration is to establish productivity as a function of actual size, actual time, and actual effort for completed projects. This newly-established productivity relationship can then be used to fine-tune the sizing process and to forecast time and effort on a new project. Once the project is complete, another iteration of the calibration process can be done. This cycle, repeated numerous times, yields sizing and forecasting methods that exhibit a high degree of accuracy with minimal variation. [1]

Sizing is one of the hardest things a development organization does, and the earlier in the life cycle it is done, the harder it is to do. [1]

Historically, statements (Source Lines of Code or SLOC) have been used for sizing systems, sometimes with poor results due to the difficulty of making the mental leap across the “abstraction chasm” from operational capability to programming language constructs. SLOC, however, is an excellent measure of the “work” done by the development process and is most effectively used in process productivity metrics. Advantages: 1) it can be unambiguously defined for a given language; 2) measuring the size of an existing product is automatable; 3) most of the world’s historical data contains SLOC as the sizing measure. Disadvantages: 1) the notion of SLOC becomes ambiguous when dealing with non-textual abstractions; 2) the measure has little meaning to the customer / end user. [1]

Function Points (FP) offer a way of narrowing the “abstraction chasm” by providing a level of abstraction between operational capability and programming language constructs. FP is an excellent measure of the ”functionality” or “value” produced by the development process and is most effectively used in “bang for the buck” type metrics. Advantages: 1) the customer / end user can likely relate to the entities being counted; 2) there are networks of people (e.g., IFPUG) dedicated to standardizing and improving the counting process. Disadvantages: 1) FP are limited to application domains for which their countable entities make sense (typically mainframe business applications); 2) the process of counting the number of FP in a finished product is not automatable, in fact many big FP shops do quick-and-dirty estimates, using shortcuts such as “backfiring” (back-calculating FP as a function of language and size in SLOC). [1]

A major qualifier on the use of FP in a productivity relationship is the fact that FP do not directly relate to development process “work” and must be scaled as a function of the programming language used. This scaling introduces an additional source of complexity and variability in the relationship. [1]

If all this isn’t enough to complicate the selection of sizing measures, consider that many new development methodologies employ abstractions that are neither textual nor do their components fit within the set of FP counting entities. [1]

Size Drives Cost, Schedule, and Reliability

> *If you underestimate the size of your next project, common sense says that it doesn’t matter which methodology you use, what tools you buy, or even who you assign to the job.*
> Ed Yourdon

Historical data shows that increasing the size of a system will increase its cost and schedule, and reduce its reliability at delivery (see Figure 1, Figure 2, and Figure 3). [6]

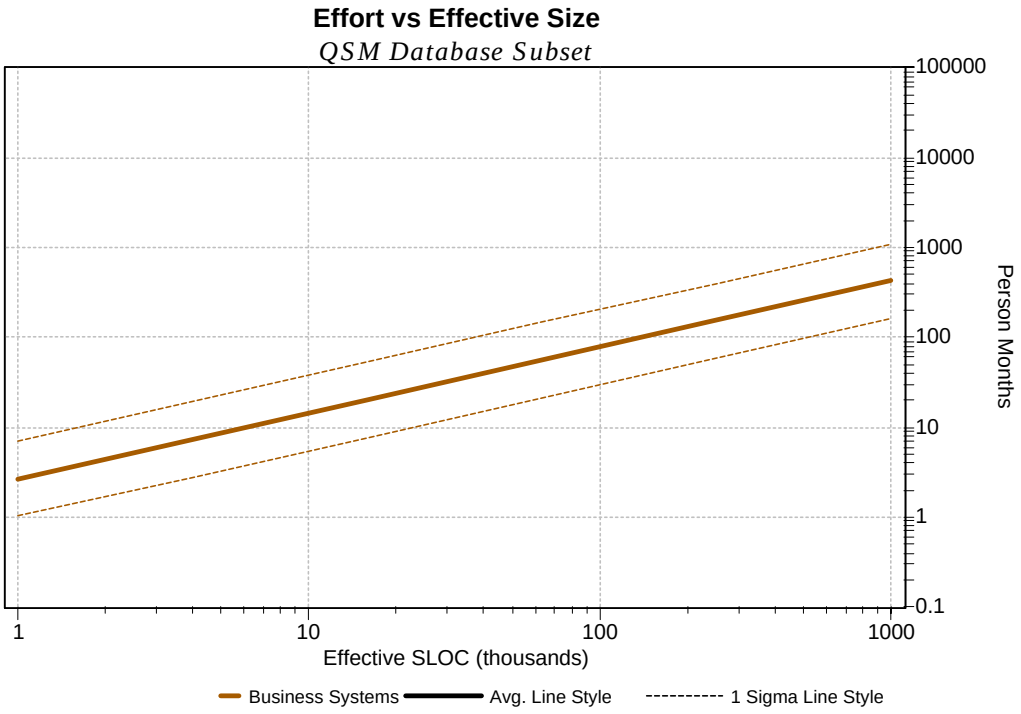


Figure 1: Effort (Cost) Increases with Size

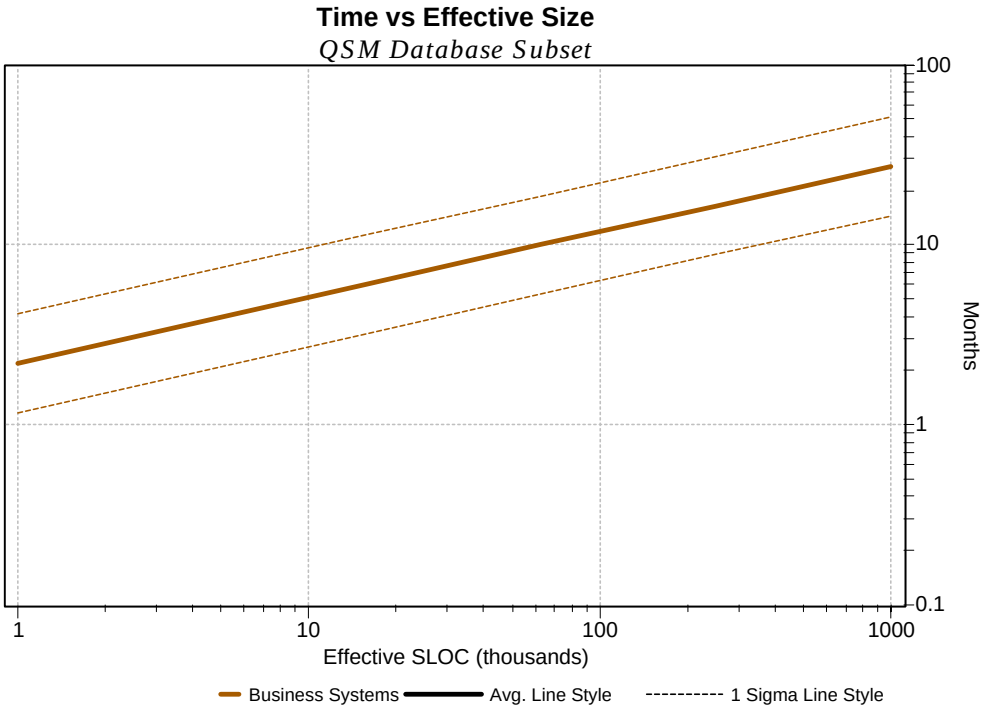


Figure 2: Schedule Increases with Size

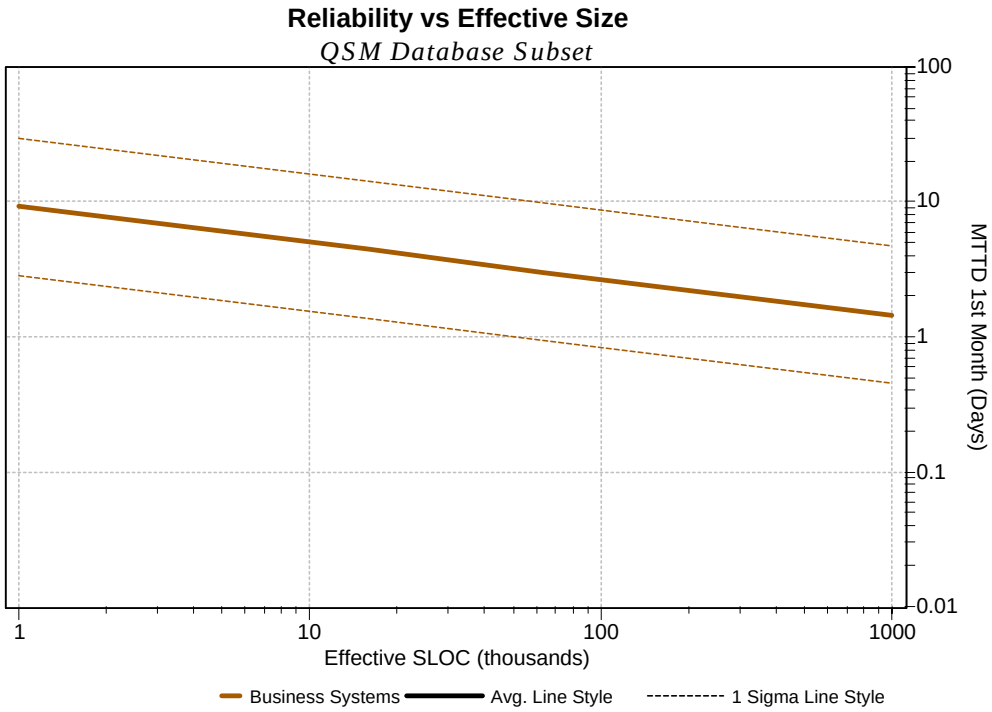


Figure 3: Reliability Decreases with Size

Size Estimation is Probabilistic

> *[Unfortunately, most estimates are] the most optimistic prediction that has a non-zero probability of coming true.*
> Tom DeMarco [2]

Size estimation is a probabilistic problem. This can be verified by simply analyzing the language that people use when they are asked to estimate something. Words and phrases such as *about*, *probably*, *somewhere around*, and *in the neighborhood of*, may seem, to some as weaseling or hedging; however, what they really indicate is the presence of uncertainty. Far too often, the tendency is to produce an estimate that represents the best case scenario.

Knowledge Increases with Time

> *In the beginning a software project is little more than a gleam in one person's eye. Yet his organization may need rough estimates of the cost and schedule to fit into advanced budgets for the next several years. As work on the concept proceeds, more becomes known about it and more precise estimates become*

> *possible.*
> Lawrence H. Putnam & Ware Myers [6]

Knowledge about any system being developed increases with time. Customers get more focused about the system's requirements. Developers get smarter about the technologies involved. Common sense suggests that as a project approaches completion:

- The rate of change of a system's estimated final size should approach zero.
- The uncertainty about a system's estimated final size should approach zero.

Change is Inevitable

> *"The only thing constant in the universe is change."*
> Unknown

A rare project experiences no requirements change or scope growth. Figure 4 illustrates that over 60% of all projects experience at least a 10% growth in requirements. A qualitative examination of the underlying data shows requirements growth to be perceived as a dominant negative factor. Intuition and experience suggest that the impact of a requirement change is inversely related to the time remaining in the schedule. In other words, requirements changes that occur late in the project have more impact than do those occurring early in the project.

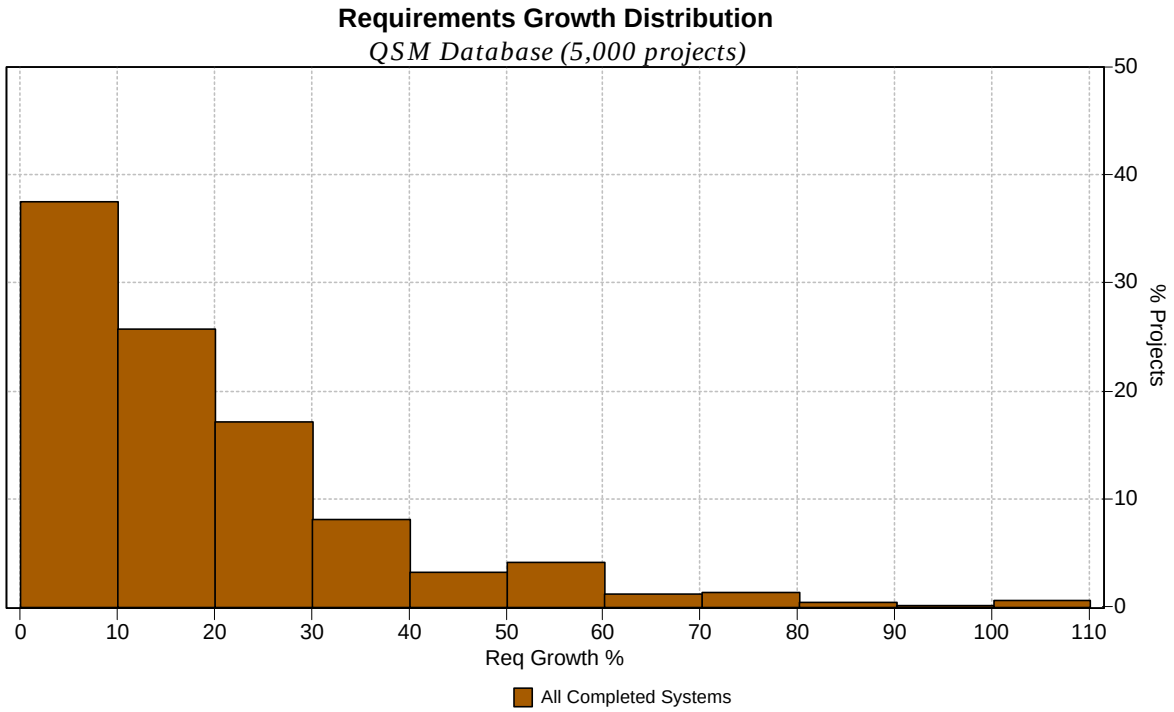


Figure 4: Requirements Growth Distribution

Developers are Optimists

> *Most people are congenitally optimistic (you have to be to get out of bed in the morning).*
> Ware Myers [7]
>
> *When you care a lot about the result, the quality of your estimate approaches zero.*
> Tom DeMarco [2]

Members of the development team are typically the ones called upon to provide estimates for the size of their systems. This would seem to make sense; after all, they know the most about what's being developed. The problem is, these same people are biased toward optimism by having a stake in the project's outcome. Generally, when someone is asked to provide an estimate of an outcome for which they are directly responsible, the tendency is to provide a best-case response.

Process Description

> *"If you don't measure, then you're left with only one reason to believe that you are still in control: hysterical optimism."*
> Tom DeMarco [2]

Estimate Periodically and Record the Data

A fundamental aspect to effectively managing the size of a system is to periodically re-estimate its size and to record the results. Figure 5 shows an example of an artifact for logging periodic size estimates.

Edit Estimate Data for ESLOC (CUM)

Start Date	to	End Date
Jan-96		Dec-97

Month	Low	Most Likely	High	Expected	Convergence
Jan-96	45000	60000	75000	60000	0.00
Feb-96	45000	60000	75000	60000	0.00
Mar-96	45000	60000	75000	60000	0.00
Apr-96	45000	60000	75000	60000	0.00
May-96	45000	60000	75000	60000	0.00
Jun-96	53000	62000	70000	61833	43.33
Jul-96	55000	62500	70500	62583	48.33
Aug-96	56000	63000	71000	63167	50.00
Sep-96	57500	63500	72500	64000	50.00
Oct-96	58000	64000	74000	64667	46.67
Nov-96	59000	64500	76000	65500	43.33
Dec-96	60000	65000	78000	66333	40.00
Jan-97	62000	66500	81500	68250	35.00
Feb-97	65500	68000	84000	70250	38.33
Mar-97	67500	70000	86500	72333	36.67
Apr-97	70500	73000	88500	75167	40.00
May-97	72500	75000	90000	77083	41.67
Jun-97	75500	78000	91500	79833	46.67
Jul-97	78500	81000	92500	82500	53.33
Aug-97	82500	85000	93500	86000	63.33
Sep-97	86500	89000	94500	89500	73.33
Oct-97	89500	92000	95000	92083	81.67
Nov-97	91500	94000	95500	93833	86.67
Dec-97	92500	95000	96000	94750	88.33

Figure 5: Sample Data Entry Form

Express Estimates as Ranges

An estimate, to be of any real value, must include two components: *magnitude* and *uncertainty*:

> ***Magnitude*** quantifies the most likely (best guess) outcome. Probabilistically speaking, this means that 50% of the time the actual outcome will be lower than the most likely outcome and 50% of the time the actual outcome will be higher than the most likely outcome.
>
> ***Uncertainty*** quantifies the distribution of all possible outcomes. This can be expressed as a low outcome and a high outcome between which 99% of all possible outcomes will fall.

An estimate expressed in this form is hereinafter referred to as a *3-point estimate*.

Adjust for Bias

Since most projects experience requirements growth, it follows that the difference between the *most likely* and *high* values in a 3-point estimate is usually much larger than is the difference between the *most likely* and *low* values. To dampen the effect of this high-side bias, the notion of an *expected* value is introduced as follows:

$$\text{Expected} = \frac{\text{Low} + 4 \times \text{Most Likely} + \text{High}}{6} \quad [6]$$

Viewing a 3-point estimate as defining some associated normal distribution, one standard deviation can then be approximated by the following:

$$\text{Standard Deviation } (s) = \frac{\text{High} - \text{Low}}{6} \quad [6]$$

Ideally, the expected size estimate value would remain constant throughout the project; i.e., the baseline value equals final actual value. This rarely happens; however, since it is the goal, this constancy forms the basis for the plan and variance threshold values in the tracking methodology.

Look for Convergence

Since the uncertainty about a system's estimated final size should approach zero, we need to establish a way to quantify and track this decreasing uncertainty. Here we introduce the notion of convergence. As knowledge and experience increase, uncertainty decreases; therefore, the high size estimate and low size estimate curves should converge. For a given elapsed calendar time t, convergence C_t is a function of the corresponding high H_t and low L_t estimates as follows:

$$C_t = 1 - \frac{H_t - L_t}{H_{Baseline} - L_{Baseline}}$$

Convergence in this context is represented as a percentage:

- A value of 0% (0) convergence represents the level of uncertainty contained in the baseline 3-point estimate.
- A value of 100% (1) convergence represents the total absence of uncertainty (i.e., the high size estimate and the low size estimate are equal).

Note that it is possible to have negative convergence (divergence) which happens when the current uncertainty is greater than the baseline uncertainty.

Cumulative effort (cost), cumulative code production, and cumulative defect discovery all behave according to the cumulative form of a Rayleigh distribution [6] (sometimes referred to in project management circles as an S curve). Uncertainty is a function of the problem-solving process, people solve problems, and people are applied to projects in a Rayleigh-like pattern. [6] A reasonable inference then is that, ideally, size estimate convergence should go from 0% at the beginning of the project and approach 100% by the end of the project according to the cumulative form of the Rayleigh function. This ideal behavior forms the basis for the plan and variance threshold values in the tracking methodology.

Track Over Time

Tracking of periodic 3-point size estimates provides valuable comparison and trend information. The idea is to track quantities that can easily verify expected project behavior and have the best chance of providing early warning when the project is in trouble. As a reasonable minimum, track $expected\ size\ estimate$ and $size\ estimate\ convergence$ as functions of elapsed calendar time. Additionally, add variance thresholds to the graphs to support the process control rules described in the next section (as exemplified by Figure 6, Figure 7, Figure 8, and Figure 9).

Control the Process

Within the context of quantitative project management, control means comparing the desired (planned) outcome to the measured (actual) in-process outcome and basing any corrective action on the difference. Plan the project and take action that causes the earliest possible convergence on what will be the final actual size value.

The following is an example implementation of a size-estimate-based control process that uses observations from the time-based tracking curves and a $traffic\ light$ metaphor with associated rules.

> ***Green Status***—No corrective action recommended.
>
> - The default state; i.e., the criteria for neither $Yellow\ Status$ nor $Red\ Status$ have been met.
>
> ***Yellow Status***—Determine the cause, take corrective action.

- Current $Expected\ Size\ Estimate$ is outside ±1σ of the baseline 3-point estimate ***or***
- Current $Size\ Estimate\ Convergence$ is outside ±1σ of the baseline plan Rayleigh convergence curve.

Red Status—Determine the cause, take corrective action, replan the project based on current 3-point estimate.

- Current $Expected\ Size\ Estimate$ is outside ±2σ of the baseline 3-point estimate ***or***
- Current $Size\ Estimate\ Convergence$ is outside ±2σ of the baseline plan Rayleigh convergence curve.

Examples of Application

Project with Stable Sizing

Figure 6 and Figure 7 illustrate the tracking of estimated size data for a project with stable sizing. After some early requirements misunderstandings, the project was re-planned during the fifth month. After the replan, the project experienced minor size growth that was well within the green threshold. The bulk of this growth occurred relatively early in the project; late changes were rare. Also, after the replan, size estimate convergence occurred close to plan and well within the green threshold. Even with the size growth, the project still managed to come in on time and within budget.

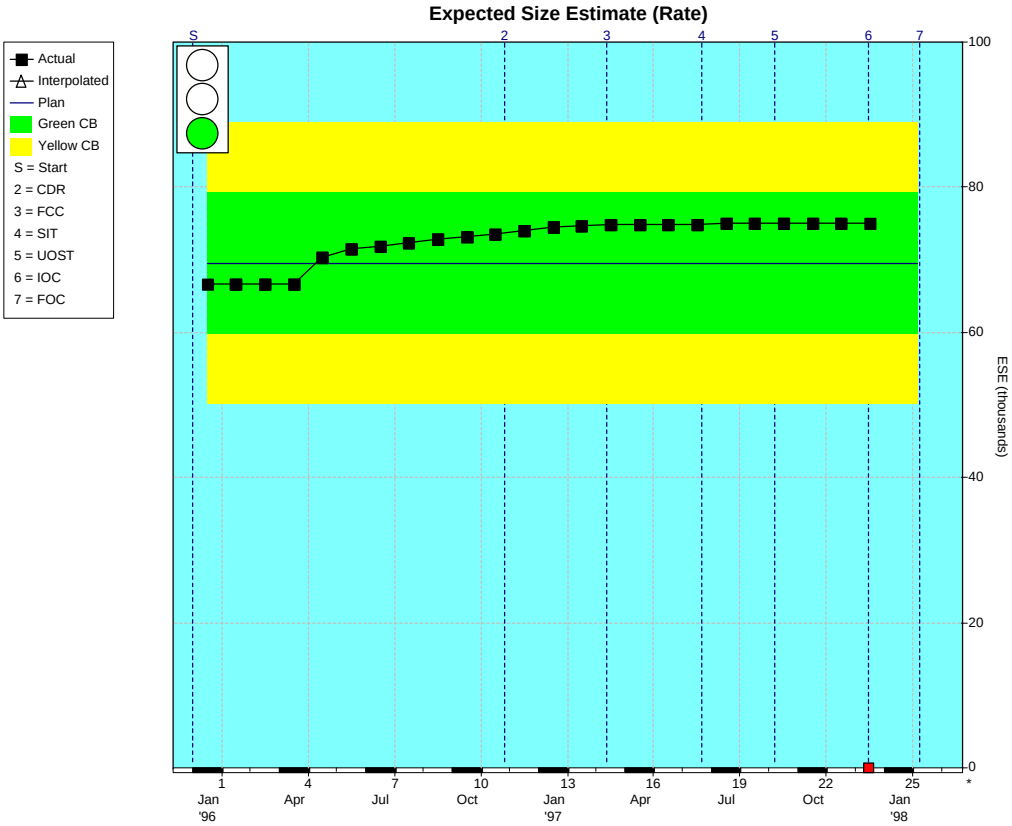


Figure 6: Expected Size Estimate Trend for Project with Stable Sizing

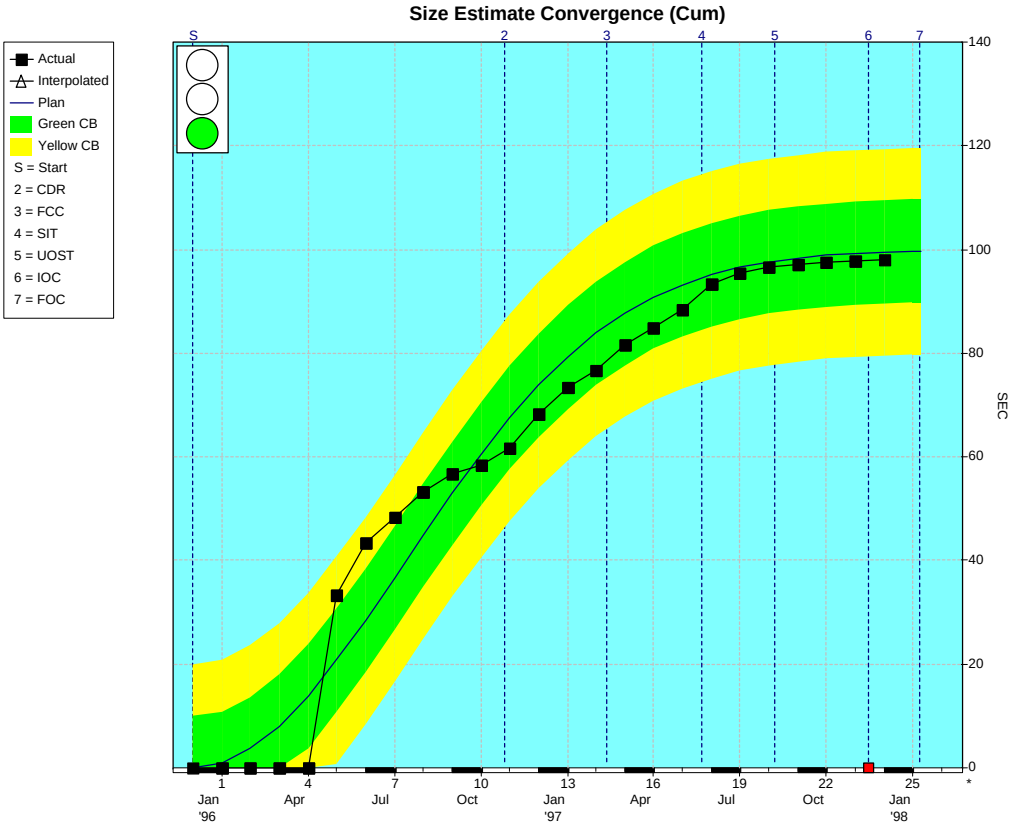


Figure 7: High-Low Size Estimate Convergence for Project with Stable Sizing

Project with Unstable Sizing

Figure 8 and Figure 9 illustrate the tracking of estimated size data for a project with unstable sizing. After some early requirements misunderstandings, the project was re-planned during the fifth month. Unfortunately, after the tenth month, some serious uncertainty began to creep in. The project was hit with several new requirements plus it lost a substantial amount of expected reuse benefit. Uncertainty increased as the bad news continued to roll in. Surprisingly, no replan was done and no get-well strategy was implemented. The expected size continued to grow, unchecked, until the customer finally initiated a code freeze and took delivery of the system as it was. The supplier subsequently lost in a competitive bid for the next phase of the program.

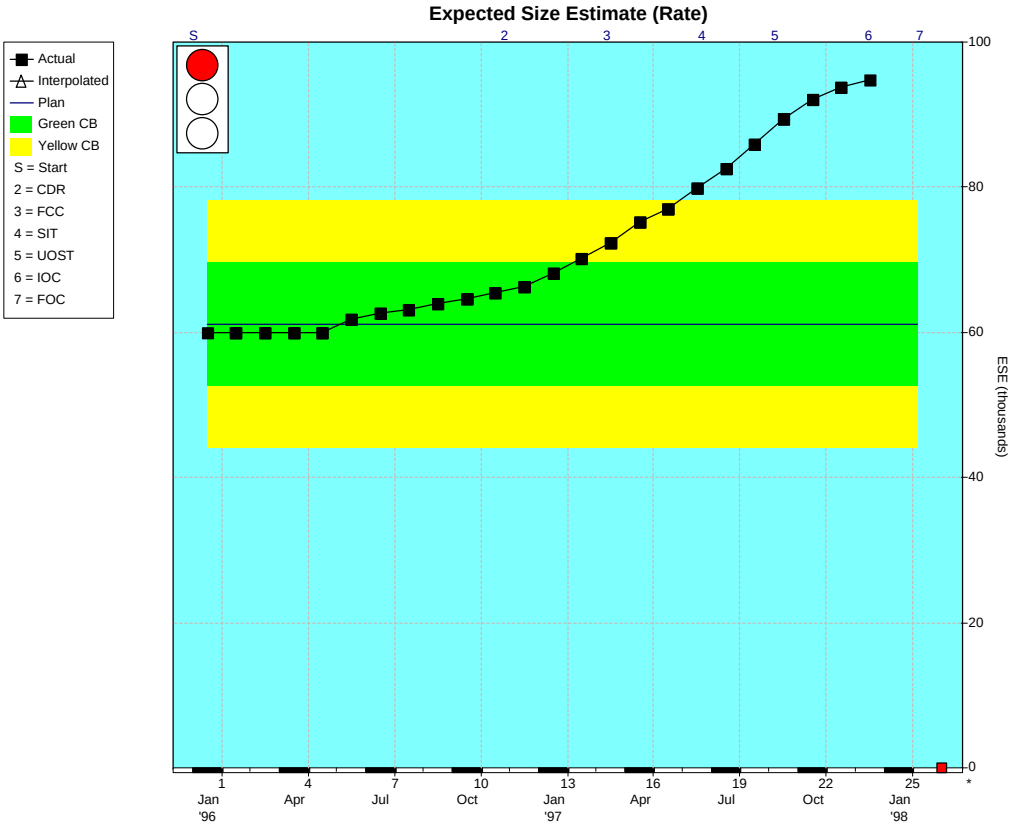


Figure 8: Expected Size Estimate Trend for Project with Unstable Sizing

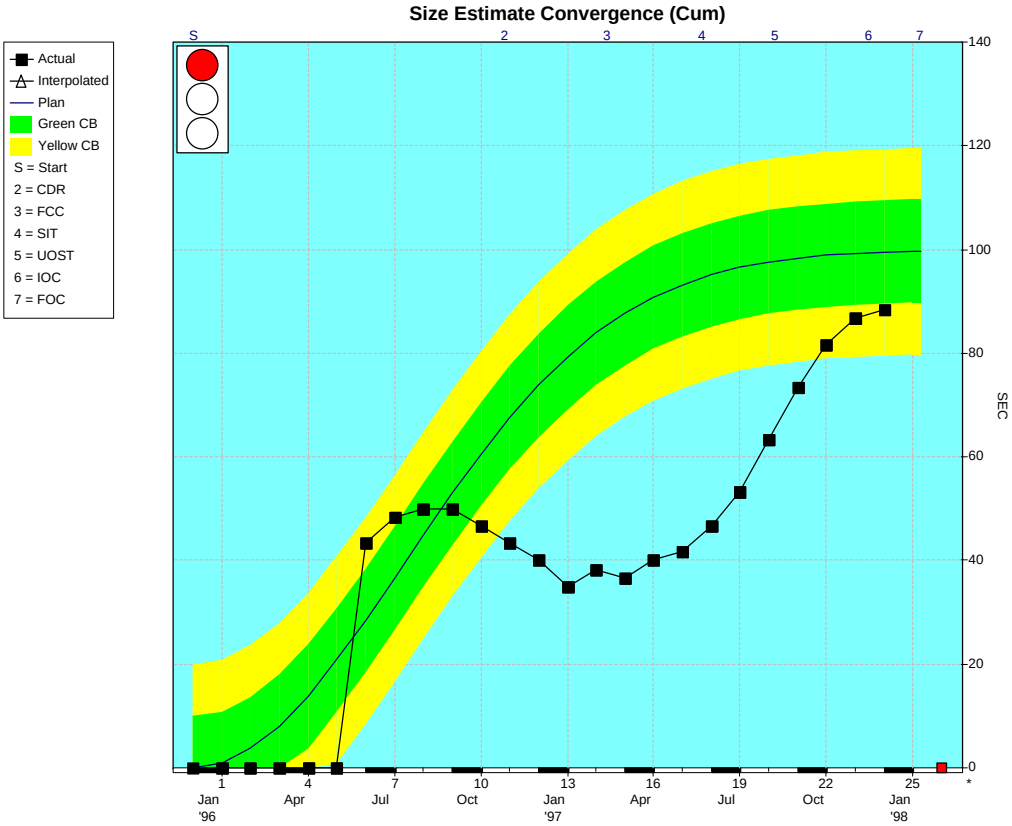


Figure 9: High-Low Size Estimate Convergence for Project with Unstable Sizing

Conclusion

Software size is a quantity that has a profound effect on development cost, schedule, and reliability. Quantitative management of software size is therefore necessary to ensure adequate quantitative management of cost, schedule, and reliability. Since software size is an estimated quantity (i.e., the final value cannot be measured) until the end of development, one must apply quantitative management techniques to the estimates themselves. This can be accomplished with continuous 3-point size estimation and time-based tracking of values derived from these 3-point estimates.

Project management success means achieving expectations. Unreasonable expectations lead to failure. Guard against expectations being too high and too low. Finally, you can't control what you can't measure. [2]

References

[1] Butler, J. & Ross, M., T., “Making the First Cut: Sizing New Technology.” *QSM Perspectives*, Fall 1997, Volume 20 Number 2, pp. 1-2, 4.

[2] Demarco, T., *Controlling Software Projects: Management, Measurement, and Estimation*. New York, NY: Yourdon Press, 1982.

[3] Humphrey, W., *Managing the Software Process*. Reading, MA: Addison-Wesley Publishing Co., 1989.

[4] Paulk, M.C., Curtis, B., Chrissis, M.B., Weber, C.V., *Capability Maturity Model for Software, Version 1.1*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993.

[5] Paulk, M.C., Weber, C.V., Garcia, S.M., Chrissis, M.B., Bush, M., *Key Practices of the Capability Maturity Model, Version 1.1*. Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1993.

[6] Putnam, L. & Myers, W., *Measures for Excellence: Reliable Software On Time, Within Budget*. Englewood Cliffs, NJ: Yourdon Press, 1992.

[7] Putnam, L. & Myers, W., *Industrial Strength Software: Effective Management Using Measurement*. Los Alamitos, CA: IEEE Computer Society Press, 1997.

Biography

Michael A. Ross is currently Managing Director of the Western Region Office of Quantitative Software Management, Inc. where, for the last four years, he has served as one of QSM's primary consultants and analysts working with Fortune 500 companies and government agencies in the areas of measurement, sizing, forecasting, and control.

Mr. Ross, during 17 years with Honeywell Air Transport Systems (formerly Sperry Flight Systems), developed or managed the development of embedded software for avionics systems installed in the Lockheed L1011-500, Boeing 757/767, Airbus A320, Douglas MD-11, British Aerospace BAe-146, and Boeing 777 airplanes. He also co-founded the division's process improvement team (later to become its SEPG), served as a corporate SEI CMM assessor, and served as the division's focal for software project management process improvement.

Mr. Ross did his undergraduate work at the United States Air Force Academy and Arizona State University, receiving a Bachelor of Science in Computer Engineering. He is a member of the IEEE, the International Function Points Users Group, the International Society of Parametric Analysts, the Arizona Software Association, and the Phoenix area Software Process Improvement Network.

Telcordia Technologies

Performance from Experience

Formerly Bellcore

Software Quality Assurance Independent Testing

"Life as a CMM Level 5 Test Organization"

May 26, 1999

John N. Romanak
Executive Director
Software Quality Assurance Independent Testing
(732) 699-8444
jromanak@telcordia.com

Copyright © 1999 Telcordia Technologies, Inc. Used by permission only.

QW'99 - Bellcore Independent Testing

Bellcore, with its long history of making innovative telecommunications technology work, is now

Telcordia Technologies™

Our name has changed, but our tradition of high standards and exceptional performance continues.

Telcordia Technologies

Formerly Bellcore…
Performance from Experience

QW'99 - Bellcore Independent Testing

"Quality, timeliness, cost-effectiveness"

. . . You cannot have all three

↓

Bellcore has proven otherwise!

Telcordia Technologies
Formerly Bellcore…
Performance from Experience

QW'99 - Bellcore Independent Testing

Who is Bellcore?

Corporate Facts

- Fast Facts About Bellcore: A Wholly Owned Subsidiary of Science Applications International Corporation (SAIC) Bellcore is a leading provider of Communications Software, Engineering and Consulting Services based on World-Class

Corporate Statistics

- 1998 Revenues exceeded $1 billion

Telcordia Technologies
Formerly Bellcore…
Performance from Experience

QW'99 - Bellcore Independent Testing

Software Quality System Profile

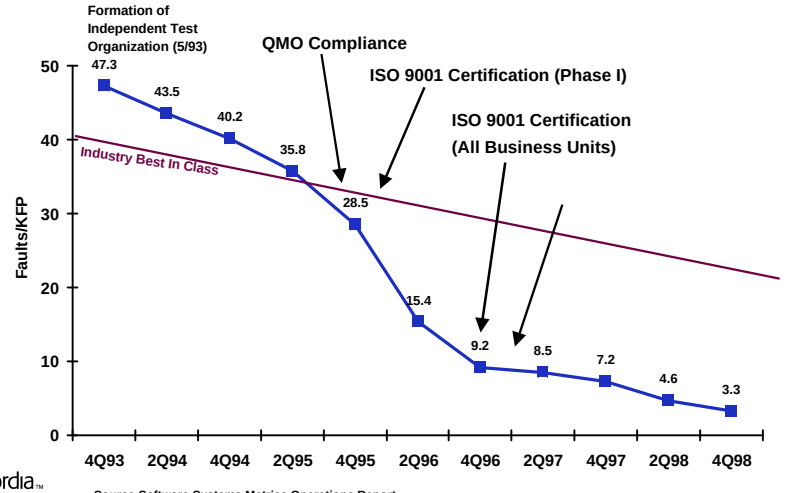
Key Facts

- CMM Level 3 and ISO9000 Certified
- 482 Technical Quality Engineers
- 164 Major Releases Shipped
- 214 Million Lines of Code Tested
- "Best in Class" Cumulative Field Fault results
- "Best in Class" Test Cost Efficiency results

Telcordia Technologies
Formerly Bellcore…
Performance from Experience

QW'99 - Bellcore Independent Testing

Field Fault Density- KFP



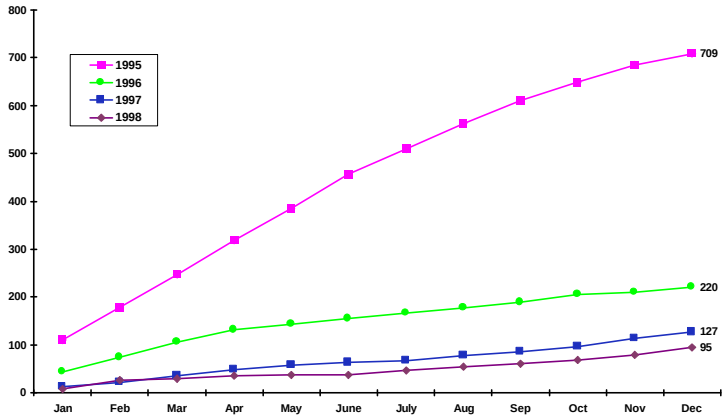
Source:Software Systems Metrics Operations Report

Telcordia Technologies
Formerly Bellcore…
Performance from Experience

Life as a CMM Level 5 Test Organization -

QW'99 - Bellcore Independent Testing

Sev1/2 Maintenance Requests



Data for 1995, 1996 and 1997 is displayed as provided at the end of each year and not retroactively adjusted

Source: Customer Service Center Metrics Database

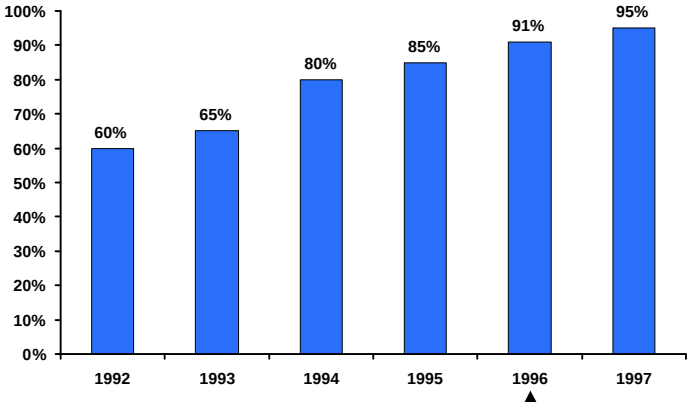
Telcordia Technologies

Formerly Bellcore…
Performance from Experience

QW'99 - Bellcore Independent Testing

Customer Satisfaction

Positive Trend in Customer Satisfaction



ISO 9001 Registered
CMM Level 3 Rating

Telcordia Technologies

Source: Bellcore Corporate Survey

Formerly Bellcore…
Performance from Experience

Life as a CMM Level 5 Test Organization -

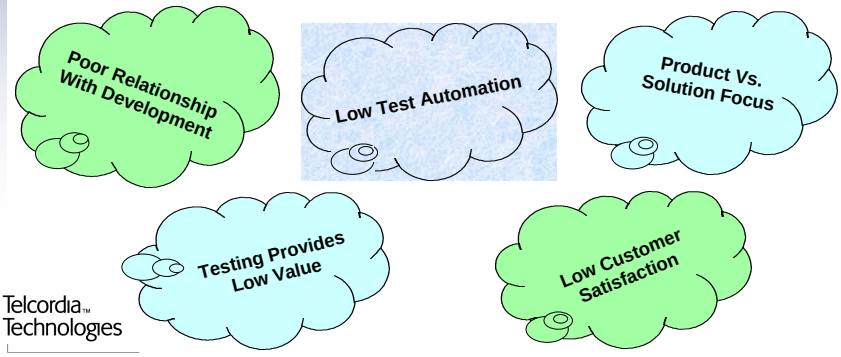
QW'99 - Bellcore Independent Testing

Start the Quality Journey

June, 1989: Why start:

"Mindset Change"

Independent Test - Clean Separation of System Test from Design & Development



Telcordia Technologies
Formerly Bellcore…
Performance from Experience

QW'99 - Bellcore Independent Testing

The Journey Continues

May, 1993: *"Corporate Organizational Change"*

Independent Test Launched - Software Systems Institutionalizes Clean Separation of System Test from Development.

Telcordia Technologies
Formerly Bellcore…
Performance from Experience

Life as a CMM Level 5 Test Organization -

QW'99 - Bellcore Independent Testing

The Journey Advances

April, 1995: *"Cultural Realization":*

Management Change - Corporate Alignment

Strict Quality Discipline - QMO

- **Testing is a Key Component Throughout the Process**
- **Early Testing is Critical to Success = "Prevention Oriented"**

Telcordia Technologies
Formerly Bellcore...
Performance from Experience

QW'99 - Bellcore Independent Testing

Manage Test like a Business

Know Your Products - Manage Them "Fundamentally" Different

SQA Testing Business Model

"Advance the Leadership"

- **Alignment with Business Unit Strategies**
- **Test Strategies Driven by Product Life Cycle**
- **Tactical Implementation of Business Model Goals**
- **Focus on Cost Reduction, Cost Avoidance, Time-to-Market**

Telcordia Technologies
Formerly Bellcore...
Performance from Experience

QW'99 - Bellcore Independent Testing

Strong Software Quality Assurance Structure

October, 1997: "*Software Quality Assurance Alignment*"

Corporate alignment of Quality Operation Center with new President & COO

SQA Web Site

- **Internal Access by Corporate Officers**
- **Business Operations Council Review**
- **Overall Quality Assessment - "Quality Conscience"**

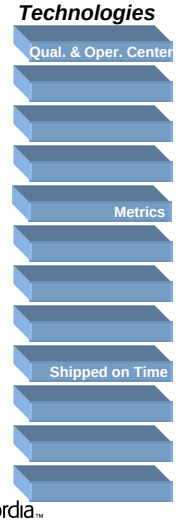
Telcordia™ Technologies

Formerly Bellcore… Performance from Experience

QW'99 - Bellcore Independent Testing

SQA Web Site - Home Page

Telcordia ä Technologies



The Software Systems Quality Assurance Testing Organizations, provide independent centralized test services for Bellcore's Software Systems Applications.

Telcordia™ Technologies

Formerly Bellcore… Performance from Experience

QW'99 - Bellcore Independent Testing

SQA Web Site - Product Status

10200 Software Quality Assurance Testing
Product Releases in Development - Product Status Summary

Department 10230 (J.N. Romanak - ED)

VP - GM

Business Unit

		Rel #	Ship Date
Y	PRODUCT 1	1.1	6/30/99
Y	PRODUCT 2	1.2.3	7/1/99
R	PRODUCT 3	4.0	7/29/99
Y	PRODUCT 4	2.2	6/30/99
Y	PRODUCT 5	2.2.2	10/31/99
Y	PRODUCT 6	2.3	6/30/99
G	PRODUCT 7	2.4	10/31/99
G	PRODUCT 8	3.1.2	9/30/99
G	PRODUCT 9	4.1	10/31/99

Telcordia Technologies
Formerly Bellcore... Performance from Experience

QW'99 - Bellcore Independent Testing

SQA Web Site - Detailed Metrics

HISTORICAL

Release #	1.0	2.0	3.0
Contractual Ship date			
Planned Ship Date			
Actual Ship Date			

RELEASE PROFILE

IN PROGRESS

Estimate	Current
4.0	
07/29/99	
07/29/99	

Assessment

Key Points

Milestones	Plan	Forecast	Actual

Telcordia Technologies
Formerly Bellcore... Performance from Experience

Life as a CMM Level 5 Test Organization -

QW'99 - Bellcore Independent Testing

Reset the Test Management Vision - '98

- **Soar with your Strengths**
  - √ **Technical Staff**
- **Advance the Leadership**
  - √ **"The Management"**
- **Dare to Soar**
  - √ **Rapid Growth**

Impact to the Corporation

Business Growth
- ↑ **204% Increase in Products Tested**
- ↑ **267% Increase in Deployed Code**
- ↑ **145% Increase in Major Releases Shipped**

Quality Improvement
- ↑ **306% Increase in Test Automation**
- ↑ **93% Improvement in Field Fault Density/KFP**
- ↑ **81% Improvement in Field Fault Density/MLOC**

Cost Efficiency
- ↓ **61% Decrease in Test Cost Per Line of Code**

Telcordia Technologies
Formerly Bellcore…
Performance from Experience

QW'99 - Bellcore Independent Testing

Proven Quality Results

Measures of Progress - Blue Chart

- * **Field Fault Density**
- * **Sev 1/2 Maintenance Requests**
- **Test Automation**
- **PT/MUT Test Faults**
- **Removal Efficiency**
- **Total Lines of Code Supported**
- **Pre-Shipment Fault Density**
- **Major Releases Shipped**
- **Product Test Fault Density**
- **Test Cost Efficiency**

Future
- **Automatic Test Case Generation**
- **Test Coverage**
- **Strong Partnership with Applied Research**

Telcordia Technologies
Formerly Bellcore…
Performance from Experience

QW'99 - Bellcore Independent Testing

Proven Strengths - Distinctive Competence Differentiators

- Industry Leader in Quality Process and Discipline
- Strong Quality Champion and Catalyst
- SQA Testing Business Model
- Solution/Integration Testing Focus
- Industry Leader in SQA Testing Measures of Progress Results
- State of the Art Test Tools
- "Best In Class" Independent Centralized Testing Operation

"Soar with your Strengths..."

Telcordia Technologies
Formerly Bellcore...
Performance from Experience

QW'99 - Bellcore Independent Testing

The Differentiator

Quality Is Our Differentiator

"You can have all three"

↓

Quality, Timeliness, Cost Effectiveness

"Life as a CMM Level 5 Test Organization"

Telcordia Technologies
Formerly Bellcore...
Performance from Experience

Telcordia Technologies

Formerly Bellcore…
Performance from Experience

Bellcore, with its long history of making innovative telecommunications technology work, is now

Telcordia Technologies™

Our name has changed, but our tradition of high standards and exceptional performance continues.

QW'99 Bellcore Paper

"Life as a CMM Level 5 Test Organization"

Introduction:

Quality, timeliness, cost-effectiveness – for years software professionals have believed you cannot have all three. Bellcore has proven otherwise. Bellcore's approach to Software Quality has made us a leading provider of World-Class software. Bellcore's Software Development organization numbers about 3500 employees, with a portfolio of 150 operations support systems and over 110 Million Lines Of Code (MLOC). Positioning Bellcore for SEI CMM Level Five in 1999.

Testing is needed now more than ever and testing is now more complex than ever. Testing is not a revenue-producing activity, but rather a revenue-preserving one. However, testing cannot be an add-on activity, after software development is complete. It must be integrated throughout the development lifecycle in order to be cost-effective. A more effective testing capability leads directly to increased customer satisfaction, as systems work and work right the first time. A more efficient testing capability leads directly to increased Quality, as test cost savings increase so does customer profitability and satisfaction.

Goals and Objectives:

- Journey through Development and Test Culture paradigm changes
- Understand how to Manage Test like a Business
- Experience what it's like to be a Corporate Independent SQA Testing Center aligning directly to the President of the Company
- Explore Bellcore's sophisticated SQA Web interface for overall Software Quality Assessment, which can be accessed by executives to view each system's overall quality "health" status based upon proven process and discipline
- Proven Measures of Progress historical metrics that serve as Bellcore's distinctive competence differentiation in being an Industry Leader in Independent SQA Testing operations

Meeting the Challenge for Test Excellence:

The quality of software testing is normally inconsistent, with different organizations performing testing in different ways. Although corporate quality models can exist, with clearly defined milestones and deliverables, the processes used to create deliverables are not always detailed and the model is not always used by all organizations. While many excellent test processes and knowledgeable people exist within the industry, this knowledge is not applied consistently to achieve predictable, consistent testing outcomes.

Key issues encountered in the Industry with its current testing environment are:

- Unclear ownership of the testing discipline in the company
- General lack of esteem in which testers are generally held compared to other software development professionals
- Inconsistent levels of involvement of users in the test planning process, and testers in the planning and software development efforts

Copyright © 1999 Telcordia Technologies, Inc. Used by permission only.

- Ineffective integration of software testing throughout the software development life cycle
- Proliferation of locally controlled testing facilities and testing within the production environment
- Lack of a centralized program to define, develop and maintain test skills.

Bellcore's testing challenge is for a robust, mature testing practice to ensure that corporate software quality goals are achieved consistently and predictably. This will require a corporate-wide testing methodology, tools, training and guidance, as well as infrastructure support for the initiative. Key goals of this practice are:

- Clear demonstration of management support for testing
- Well-defined test processes used by trained individuals
- Testing efforts that are aligned with business goals
- Testing measurements to ensure adequate test quality, performance and accountability
- High level of quality and overall customer satisfaction for testing.

Bellcore clearly understands testing vision and software quality assurance challenges that lie ahead. There are many similarities between software industry's current situation and its challenges and Bellcore's experiences with software development. Often, testing of software products is an afterthought - a requirement for sign-off or at best an opportunity to find the most obvious of the software "bugs" prior to production. The focus is typically on defect removal instead of on defect prevention. It is much more cost effective to avoid "bugs" before coding is completed than to test, identify and correct software anomalies after coding is completed. Bellcore's solution for software quality assurance testing is based on the premise that the software testers are involved throughout the Software Development Lifecycle. From the development of software testing strategies during the software sales process to the production implementation at a customer site, the Testers are always valued members of the software development team. The Testers provide important insights on the testability of the requirements, the

Copyright © 1999 Telcordia Technologies, Inc. Used by permission only.

usability of the designs, and the overall test strategy required to fully test the feature, product or solution at hand. Since the testers typically have the most “hands-on” experience (next to the actual customer), they bring highly valuable customer insights to the team that will help to improve overall customer satisfaction. The involvement of the Independent Test Staff throughout the Software Development Lifecycle will also help to raise the overall technical expertise of the Test staff as they increase their understanding of the product and the overall end-customer requirements and usage.

Experience Bellcore’s Journey for Test Excellence:

Bellcore has been an industry leader in testing techniques and quality assurance applications. Bellcore has developed a unique testing program and methodology that has led to a level of product and process interoperability that is unmatched in the industry. Bellcore’s Software Systems Organization has received industry recognition in two major quality arenas:

- In October 1996, Bellcore’s ISO 9001 Certification was expanded to include all of Software Systems. The quality system was examined for conformance to the twenty elements required by ISO through a random sampling across a wide spectrum of projects. With no open non-conformities at the end of the audit, we were recommended for immediate certification.

- In December 1996, Bellcore Software Systems successfully completed the Software Engineering Institute’s Capability Maturity Model (CMM) assessment and achieved a Level 3 rating with significant strengths noted in some key process areas at higher levels. Only 11% of organizations assessed have been rated as Level 3 or higher. In terms of organizational size and breadth of scope, Bellcore Software Systems was the largest assessment ever performed.

Bellcore’s software quality journey for “Best in Class” Independent Testing began in the 1992/1993 timeframe. At that time, only 30% of our testing was automated. Testing was viewed as providing low value and the relationship between the testing and development organizations was poor at best.

Copyright © 1999 Telcordia Technologies, Inc. Used by permission only.

Bellcore’s approach to Independent Software Quality Assurance Testing, a clean separation of system test from design and development, has been a key component of this success. Bellcore's testing methodology specifies the ownership of tasks and responsibilities and also the process of obtaining agreement and buy-in across organizations. In addition, goals are concise and quantitative and linked to Bellcore’s appraisal process. Bellcore's testing methodology requires that quality is everyone's responsibility across the organization. Bellcore's methodology specifies the roles of each organization and of each participant. Deliverables by all organizations involved in the system development are reviewed and signed off according to the criteria set by the team. Each test activity has roles and participants clearly identified. Bellcore’s strategy includes methods for determining the number of staff, level of expertise and particular job functions required for a software project/product based on where in the software lifecycle the project/product falls. In general, Bellcore’s methodology includes 4 primary job functions: Test Project Manager (responsible for initiatives related to the Quality Assurance Test function), Test Manager (responsible for the overall test activities for the product/project that his/her staff supports and the management of this staff), Lead Tester (responsible for the technical aspects of the testing and for mentoring the testers) and Tester (responsible for detailed test planning and execution).

It is a common behavioral characteristic of software development organizations to staff the software testing function with under-qualified personnel. Most often, talented market hires are directed to the software development shop before consideration is given to the software testing organization. It is not uncommon for a Test Organization to report to the same management as the Requirements and Development staff. This structure lends itself to the redirection of qualified technical staff to work on systems engineering and/or development efforts instead of software testing. The first step in the evolution of improving software quality is to improve the quality of the test staff. The key here is what Bellcore has done with the formation of an independent test organization, so there is a clean separation of system test from design and development. Make it known within the company that only “the best” will work in the Independent Testing Organization. Make sure this Test Organization has the best equipment – not “hand me downs” from Development. Hire highly technical, experienced people. Seed the Test

Copyright © 1999 Telcordia Technologies, Inc. Used by permission only.

Organization with a handful of stars from inside and outside the company to increase the credibility of the Test Organization and to act as role models, mentors, and trainers for the less technical staff. Do not allow poor performers to transfer to the Independent Testing Organization. Reward the test staff when they succeed. Make it clear that Testing is a profession with a clear career path. Set up clear job descriptions for each level of staff with titles that are comparable to those of the Development Organization. Make sure the Training Organization provides clear training tracks and courses for professional testers. Provide opportunities for the test staff to attend outside training and Test conferences in order to raise awareness that Testing is a valuable career choice for software professionals.

Bellcore's experienced test managers tailor and implement comprehensive testing solutions, while developing an overall test strategy and plan to support its Software Systems' Business. By following a well defined Test Strategy, we are able to implement a corporate-wide testing methodology, tools, training and guidance. Testing efforts will be aligned with business goals, and progress will be measured to ensure adequate test quality, performance and accountability, with a focus on quality and overall customer satisfaction. Bellcore's business of software testing is based on the maturity of the software product or solution being tested. For example, a product's test automation strategy will be very different for an emerging product than for an established product. The focus of the automation strategy is cost reduction, cost avoidance, and time to market. In addition, Bellcore's Independent Testing operations moves the decision-making and budget responsibilities for each product's test function to the line manager of the test group. The Test Manager is responsible for test cost estimation, test budget tracking, and personnel. Also, the test manager and the individual testers are responsible for signing the test readiness checklist and signoff sheet for each product release. Bellcore additionally calls for key learning sessions after each phase of the software process and the Independent Testing Organization provides valuable insights for each of those. In addition, our Test Organization uses a full range of in-process test metrics to manage each testing phase, and to determine areas for improvement going forward. These metrics are produced at the product level and rolled up to the corporate level with intermediate levels as necessary. The corporate level metrics are used to

Copyright © 1999 Telcordia Technologies, Inc. Used by permission only.

compare and contrast Bellcore's performance against software industry trends and to track overall progress as a company. Individual product metrics are used to determine the items to be entered in each product's Quality Improvement Plan and to measure the product's progress in achieving these planned improvements.

In addition to metrics, the Bellcore Independent Testing Organization incorporates the use of a SQA Testing Project Manager. The Project Manager is responsible for guiding the Independent Testing Management team in identifying key areas for improvement for the Test Function as opposed to the individual Products, and for identifying and implementing the steps necessary to move the Testing and overall Quality to best in class in the industry. Bellcore has dramatically increased the number of test cases executed per tester in its own testing organization by the effective use of test automation. Test tools play a critical role in an organization's overall quality program. Tools make it possible to improve the efficiency and effectiveness of the testing process. When used prudently, test tools can increase the rate of defect discovery and decrease test cycle times. However, successfully introducing tools into an organization requires an overall automation program. The program must be based on (1) goals with realistic expectations for return on investment, (2) an organizational structure with strong management support and (3) technology built with sound software engineering discipline. Experience has shown that automation goals should not be set uniformly across all application teams. Goals for automating should be established based on the following factors: application lifecycle, application life-span, relationship to critical business work-flows, and platform strategy. Developing automated scripts at the wrong time or for the wrong reasons can be much more expensive than manual testing. Therefore, the automation program should be initiated with specific and realistic expectations for the return on investment.

Metrics are critical to the successful implementation of a quality testing organization for several reasons. First of all, metrics give all members of the team visibility into the project as well as advance warning of problems. In addition, metrics allow the organization to track its progress on quality improvement and celebrate success. Some of the metrics that Bellcore uses successfully as its "yardstick" are:

Copyright © 1999 Telcordia Technologies, Inc. Used by permission only.

- Field Fault Density
- Severity 1 and 2 Maintenance Requests
- Test Automation
- System Test/Multi-Unit Test Faults Ratio
- Removal Efficiency
- Pre-shipment Fault Density
- Major Releases Shipped
- Test Cost Efficiency

In addition, Bellcore has established a considerable in-process metrics program. This program allows for the Management and Staff of the product teams to set goals and evaluate their progress against the goals on a consistent basis. In-process metrics can be used to determine if additional Management intervention is needed when products fail to meet these in-process goals. Bellcore has considerable experience in implementing and managing realistic jeopardy action plans that will guide a product back onto schedule. Bellcore has found that this set of metrics provides valuable insight into areas for improvement and the level of product quality.

Bellcore's testing methodology clearly states organizational testing goals, in both subjective measures and metrics. Goals are formed “top-down” and clearly support the overall corporate goals. Quarterly corporate test and quality goals are measured and the corresponding graphs and charts are strategically displayed in all locations so that each member of the product team (not only the test organization) is aware of the current status. In particular, the test metrics of effectiveness (number of MRs found by testing, number of MRs found in deployment, etc.) and efficiency (percent of test cases automated tracked to the business model for automation, etc.) combine to clarify each testing organization’s goals.

Copyright © 1999 Telcordia Technologies, Inc. Used by permission only.

Bellcore's Proven Results:

Since implementation of our Quality Process and our Independent Software Quality Assurance (SQA) Testing Organization, we have reduced the number of Severity 1 and 2 troubles our customers experience by 82% from 1995 to 1997, and, 51% through 1998, saving tens of millions of dollars for both our customers and ourselves, and improving our overall customer satisfaction from 60% to 95%.

To put these results in context, from 1993 to 1998, Bellcore's Software Systems organization experienced the following growth in its business:

- 204% increase in products tested
- 267% increase in deployed code
- 145% increase in major releases shipped

In terms of quality improvement, Bellcore's results are:

- 306% increase in test automation
- 93% improvement in Cumulative Field Faults/KFP
- 81% improvement in Cumulative Field Faults/MLOC

From a cost efficiency standpoint, Bellcore has achieved a 61% decrease in Test Cost per Line of Code Tested over the same period.

Bellcore's proven testing methodology involves culture, process, discipline, and organizational change. Bellcore acknowledges the difficulty to introduce and implement this new process. However, this methodology will lead to institutionalizing test quality assurance, with measurable Quality Improvement.

Copyright © 1999 Telcordia Technologies, Inc. Used by permission only.

Bellcore's Testing Paradigm Change:

Bellcore has created a world class software development quality assurance testing organization. We know the pitfalls and success factors, especially for an organization of the size and complexity of Bellcore's. We know what it takes to make such an enormous paradigm change to where software testing is a valued part of software development, software testers are highly critical members of the product/project team, and where the focus shifts from error detection to error prevention. We have a proven model for job and career satisfaction for testers and test managers that enables hiring and retaining excellent staff in Testing.

Benefits of Bellcore's Independent Testing Organization to the business include:

- Shortening implementation time for launching internal testing programs
- Assuring high quality results by working with a world class process
- Reducing costs of recruiting and retaining top personnel
- Directing investment to the right equipment and tools
- Giving personnel more time to focus on developing their own expertise and capabilities
- Improved planning and budgeting
- Exceeding customers' expectations by starting testing earlier so customers' services get to market on time
- Achieving "Best in Class" measurable Quality Improvement and Metrics.

In Closing:

Bellcore's SQA Independent Testing Organization is now industry-recognized as best in class. We have presented numerous papers describing our SQA Testing approach at

Copyright © 1999 Telcordia Technologies, Inc. Used by permission only.

industry conferences such as STAR, Software Quality week, EuroSTAR and ISSRE. In the last 3 years, we have provided workshops on Independent SQA Testing and have shared our success story for many large corporations.

References:

Special thanks goes to Anita Sauer, Executive Director, and Chris Spuler, Senior Director, for their experience and insights in shaping key points in this paper.

Copyright © 1999 Telcordia Technologies, Inc. Used by permission only.

Using the Testing Maturity Model SM to Assess and Improve Your Software Testing Process

Ilene Burnstein, Ariya Homyen,
Taratip Suwanassart, Robert Grom,
Gary Saxena

Computer Science Department
Illinois Institute of Technology, Chicago, IL

SM: Testing Maturity Model and TMM are service marks of Illinois Institute of Technology

Organization of the Talk

- Introduction
- TMM Level Hierarchy
- TMM Assessment Model
- Preliminary Experimental Results
- Maturity Model Integration Issues
- Summary and Future Research

Why Develop a Testing Maturity

- Process Quality Impacts on Product Quality
- Testing Provides Support for Quality Software
- Existing Process Improvement Models have not Adequately Addressed Testing Issues
- A Mature Testing Process has not been Adequately Defined

The TMM - Purpose and Usage

- PURPOSE

  -TMM can assist organizations with assessing and improving their testing processes

- USAGE

  -Assessors: identify current testing capability state
  : initiate test process improvement
  : improve testing capabilities
  define their role in testing

TMM Components

- Set of Levels
  -levels prescribe testing maturity hierarchy
  -level structure: goals,

- Assessment Model
  -assessment instrument, determines level of maturity
  -assessment procedure, guides team
  -team selection/training procedures
  -assessment forms and tools

The TMM Level Hierarchy

TMM Level 2 : Phase Definition

- Characteristics
  - -Testing is phase that follows coding
  - -Testing is separate from debugging
- Level 2 Maturity Goals
  - 1: Develop Testing and Debugging Goals
  - 2: Initiate Test Planning Process
    - Basic Test Tech/Methods

TMM Level 3 : Integration

- Characteristics
  - -Testing no longer phase following coding
  - -Testing is professional activity
- Level 3 Maturity Goals
  - 1: Establish a Software Test Organization
  - 2: Integrate Testing into SW Life Cycle
  - 3: Control and Monitor Testing
  - 4: Establish a Technical Training Program

TMM Level 4: Management and Measurement

- Characteristics

  -Definition of a testing activity broadened
- Level 4 : Maturity Goals

  1: Establish Org-Wide Review Program

  2: Establish Test Measurement Program

  3: Software Quality Evaluation

TMM Level 5: Optimization/Defect Prevention & Quality Control

- Characteristics

  -Infrastructure, leadership in place to evaluate, improve product, process quality
- Level 5: Maturity Goals

  1: Defect Prevention

  2: Quality Control

  3: Test Process Optimization

The TMM Assessment Model: the Questionnaire

- Used CMM, SPICE concepts for developm't
- Questions cover respondent data, TMM , tool use, trends
- MG questions organized by TMM levels
- Easy to complete - answers are input as "yes, no, doesn't apply, don't know"
- Trial use in industry

The TMM Assessment Procedure I

- Preparing for the Assessment
  - select and train team
  - select projects
  - prepare organizational units
  - develop assessment plan
- Conducting the Assessment
  - collect data (questionnaire, interviews, etc.) apply ranking algorithm to determine rank

The TMM Assessment Procedure II

- Reporting Assessment Outputs
  - test process profile
  - testing strengths and weaknesses

    assessment record
- Analyze Data
  - identify and prioritize improvement goals

    quantify improvement goals

    identify possible sub-processes/ practices for

The TMM Assessment Procedure III

- Develop Action Plans
  - involve assessment team, SQA, SEPG, managers

    develop action plans based on analysis of

    select pilot projects
- Implement and Track Improvements

    apply to pilot projects

    monitor and track process

    evaluate for organization-wide adaptation

TMM Ranking Process

1. Use Traceability Matrix, insure data integrity for ranking
2. First rate the Maturity Subgoals- use measure of “degree of satisfaction”
3. Then rate the Maturity Goals at each level
4. Determine Maturity Level using Maturity Goal Satisfaction Criteria (scale is 1-5)

Assessment Support

- Tools, forms and templates
  - assures assessments are consistent, repeatable
    reduce subjectivity
    ensure validity, usability, comparability of results
- Examples
  - process profile
  - team training data recording template
    web-based questionnaire at: www.
      supports local and distributed assessors by storing & organizing data at central site

Experimental Results

- TMM questionnaire applied at 2 industrial sites
  - Org1 -tests household hardware items
  - Org 2 - develops telecom software and hardware
    - helped to clarify and reorganize questions
      experimented with ranking algorithm
      generated example action plans
- TMM Ranking
  - both rank at TMM level 1
  - strengths and weaknesses vary
  - evidence for practices at higher levels of TMM

TMM - CMM Integration Issues

- TMM is a type of CMM
- Based on industry feedback- CMMs in all process domains should be integrated, and

  - common concepts and vocabulary
  - overlapping goals
  - parallel assessments
  - common training for assessors

Development Maturity Model (DMM)SM Framework

- Set of DMM Principles
- DMM Architecture
  - naming and numbering of levels
  - internal structure of levels
  - distribution of KFAs
- DMM Construction Process
  - discovery, architecture, synthesis, validation

SM: Development Maturity Model and DMM are service marks of Illinois Institute of Technology

Summary and Future Research

- Need for TMM established
- Components defined for TMM version 1.0
- Trial use of TMM for test process evaluation
- Current and Future Plans
  - model validation: wider use of TMM version 1.0 in
  - testers workbench
  - test process reuse
  - maturity model integration, TMM v 2.0

Using the Testing Maturity Model (TMM) to Assess and Improve Your Software Testing Process

Ilene Burnstein, Ariya Homyen, Taratip Suwanassart, Robert Grom, Gary Saxena

Computer Science Department
Illinois Institute of Technology
Chicago, IL 60616
csburnstein@minna.iit.edu

Abstract:

Our research has resulted in the development of a Testing Maturity Model (TMM) designed to assist software organizations in assessing and improving their software testing processes. The TMM consists of two major components, a set of maturity levels and an assessment model. In previous work we have reported on the set of maturity levels and maturity goals. In this paper we review the TMM level structure, and describe in detail the recently developed TMM Assessment Model, its supporting forms and tools, preliminary experiments with application in industry, and issues related to maturity model integration.

1.0 Introduction

Software systems are playing an increasingly important role in our society. They have a strong impact on vital operations in for example, the business, medical, and telecommunication domains. For this reason, it is imperative that we address quality issues that relate to both the software development process and the software product. Our research focuses on process. We are developing a Testing Maturity Model (TMM) designed to assist software development organizations in evaluating, and improving their testing processes [1-3]. Testing is applied in its broadest sense to encompass all software quality-related activities. We believe that improving the testing process thorough application of the TMM maturity goals will have a very positive impact on software quality, software engineering productivity, and cycle time reduction efforts.

In previous papers we have reported on our approach to building version 1.0 of the TMM [1,2]. We have also described the internal structure the TMM including its maturity levels, associated maturity goals, subgoals and activities/tasks and responsibilities. In this paper we focus on the recently developed TMM Assessment Model. We describe how the TMM Assessment Model can assist an organization with evaluating and improving its software testing process through use of its components: the questionnaire, the assessment procedure, and supporting templates and tools. We also discuss preliminary experimental results based on trial use of the TMM in industry, and issues in maturity model integration.

2.0 An Overview of the TMM

The development of version 1.0 of the TMM was guided by the work done on the SW-CMM, a process improvement model that has received wide-spread support from the software industry in the United States [4]. Other input sources to TMM development include, Gelperin and Hetzel's Evolution of Testing Model [5] that describes the evolution of the testing process in industry over a 40 year period, and the Software Testing Practices Survey Report [6] that identifies best test practices in industry as of 1993.

The TMM is a staged process improvement model with a 5 level hierarchy of goals that support incremental test process improvement. Version 1.0 of the TMM has two major components [1-3]:

1. SET OF LEVELS.

There are 5 levels in the TMM that prescribe a maturity hierarchy and an evolutionary path to test process improvement. The characteristics of each level are described in terms of testing capability and organizational goals. Each level with the exception of level 1 has a structure that consists of:

• A set of maturity goals. The maturity goals identify testing improvement goals that must be addressed in order to achieve maturity at that level. In order to be placed at a level, an organization must satisfy the maturity goals at that level. The TMM levels and associated maturity goals are shown in figure 1.

• Supporting maturity subgoals. They define the scope, boundaries and needed accomplishments for a particular level.

• Activities, tasks and responsibilities. (ATR) The ATRs address implementation and organizational adaptation issues at a specific level. Activities and tasks are defined in terms of actions that must be performed at a given level to improve testing capability; they are linked to organizational commitments. Responsibilities are assigned for these activities and tasks to three groups that we believe represent the key participants in the testing process, managers, developers/testers, and users/clients.

The maturity goals at each level of the TMM are shown in figure 1. They are fully described in previous work [1,2], and are also listed below along with a brief description of the characteristics of an organization at each TMM level. This will introduce the reader to the evolutionary path prescribed in the TMM for test process improvement.

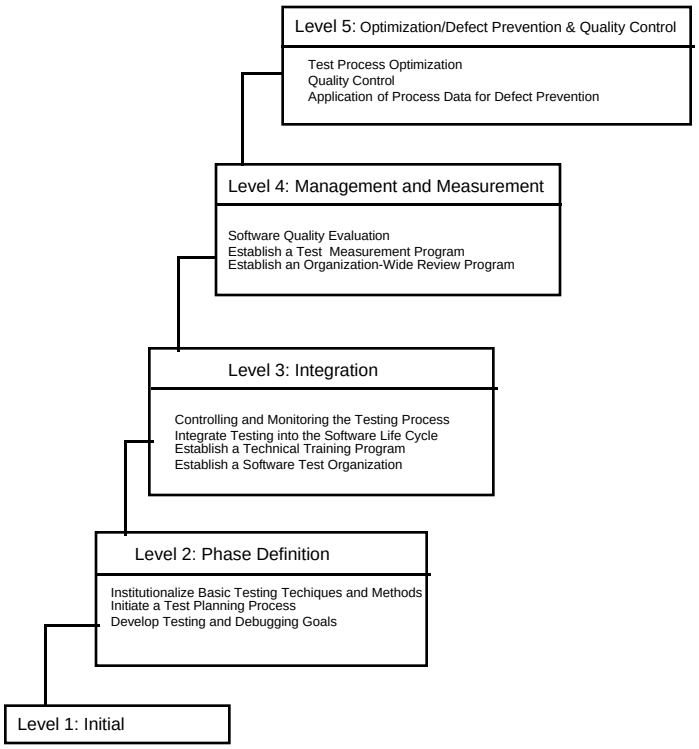


Figure 1: The 5-Level Structure of the Testing Maturity Model

Level 1 - Initial: (No maturity goals)

At TMM level 1, testing is a chaotic process; it is ill-defined, and not distinguished from debugging. Tests are developed in an ad hoc way after coding is done. Testing and debugging are interleaved to get the bugs out of the software. The objective of testing is to

show the software works [1,5]. Software products are often released without quality assurance. There is a lack of resources, tools and properly trained staff. This type of organization would be on level 1 of the CMM.

Level 2 - Phase Definition: (Goal 1. Develop Testing and Debugging Goals, Goal 2. Initiate a Testing Planing Process, Goal 3. Institutionalize Basic Testing Techniques and Methods)

At level 2 of the TMM testing is separated from debugging and is defined as a phase that follows coding. It is a planned activity, however test planning at level 2 may occur after coding for reasons related to the immaturity of the testing process. For example, there is the perception at level 2, that all testing is execution-based and dependent on the code, therefore it should planned only when the code is complete.

The primary goal of testing at this level of maturity is to show that the software meets its specifications [2,5]. Basic testing techniques and methods are in place; for example, use of black box and white box testing strategies, and a validation cross-reference matrix. Many quality problems at this TMM level occur because test planning occurs late in the life cycle. In addition, defects are propagated from the requirements and design phases into the code. There are no review programs as yet to address this important issue. Post-code, execution-based testing is still considered the primary testing activity.

Level 3 - Integration: (Goal 1. Establish a Software Test Organization, Goal 2, Establish a Technical Training Program, Goal 3. Integrate Testing into the Software Life Cycle, Goal 4. Control and Monitor Testing)

At TMM level 3, testing is no longer a phase that follows coding, but is integrated into the entire software life cycle. Organizations can build on the test planning skills they have acquired at level 2. Unlike level 2, planning for testing at TMM level 3 begins at the requirements phase and continues throughout the life cycle supported by a version of the V-model [2]. Test objectives are established with respect to the requirements based on user/client needs, and are used for test case design and success criteria. There is a test organization and testing is recognized as a professional activity. There is a technical training organization with a testing focus. Testing is monitored to ensure it is going according to plan and actions can be taken if deviations occur. Basic tools support key testing activities, and the testing process is visible in the organization. Although organizations at this level begin to realize the important role of reviews in quality control, there is no formal review program and reviews do not as yet take place across the life cycle. A formal test measurement program has not yet been established to qualify process and product attributes.

Level 4 - Management and Measurement: (Goal 1. Establish an Organization-Wide Review Program, Goal 2. Establish a Test Measurement Program, Goal 3. Software Quality Evaluation)

Testing at level 4 becomes a process that is measured and quantified. Reviews at all phases of the development process are now recognized as testing/quality control activities. They are a compliment to execution-based tests to detect defects. An extension of the V-model as shown in Figure 2 can be used to support the implementation of this goal [2]. Software products are tested for quality attributes such as reliability, usability, and maintainability. Test cases from all projects are collected and recorded in a test case database for the purpose of test case reuse and regression testing. Defects are logged and given a severity level. Some of the deficiencies occurring in the test process are due to lack of a defect prevention philosophy, and the porosity of automated support for the collection, analysis and dissemination of test-related metrics.

Level 5 - Optimization/Defect Prevention/Quality Control:(Goal 1. Defect Prevention, Goal 2. Quality Control, Goal 3. Test Process Optimization)

Because of the infrastructure that is in place through achievement of the maturity goals at levels 1-4 of the TMM, the testing process is now said to be defined and managed; its cost and effectiveness can be monitored. At level 5 mechanisms are in place so that testing can be fine-tuned and continuously improved. Defect prevention and quality control are practiced. Statistical sampling, measurements of confidence levels, trustworthiness and reliability drive the testing process. There is an established procedure for selecting and evaluating testing tools. Automated tools totally support the running and rerunning of test cases. Tools also provide support for test case design, maintenance of test-related items, and defect collection and analysis. The collection, and analysis of test-related metrics also has tool support.

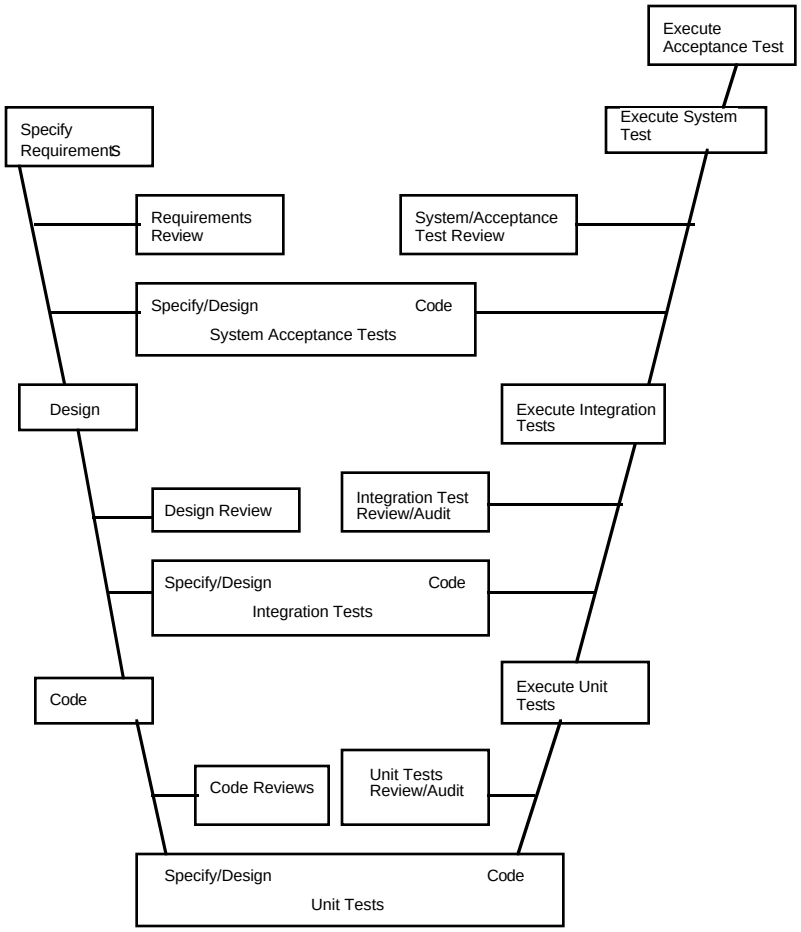


Figure 2: The Extended/Modified V-Model

2. THE TMM ASSESSMENT MODEL.

In order for a process improvement model to be effective, it must include as one of its components, a mechanism for process assessment so that an organization can evaluate the state of its current process. We have developed the TMM Assessment Model (TMM-AM) to fulfill this requirement. The TMM (Levels, Maturity Goals, Subgoals and ATRs) serves as a reference model for the TMM-AM. The outputs of a TMM assessment allow an organization to :

- determine its level of testing maturity (on a scale from 1-5)
- identify its testing process strengthens and weaknesses
- prioritize areas that need improvement

- develop action plans for test process improvement
- identify mature testing process elements that are candidates for reuse.

In the following sections we discuss the TMM-AM in detail.

3.0 The TMM Assessment Model : Development Approach

In order to support the research goals for the TMM and to provide a tool for self-assessment of the testing process we have developed the TMM Assessment Model (TMM-AM). Our research objectives for the TMM-AM were to: 1. provide a framework, based on a set of principles in which software engineering practitioners could assess and evaluate their software testing processes, 2. provide a foundation for test process improvement through data analysis, and action planning, and, 3. contribute to the growing body of knowledge in software process engineering.

We have used the SW-CMM and SPICE Assessment Models to guide development of the TMM-AM [7-9]. Our goals were to have the resulting TMM-AM be Capability Maturity Model Appraisal Framework (CAF) compliant [8] and integratable with the SW-CMM Assessment Model, so that in the future, organizations would be able to perform parallel assessments in multiple process areas. A set of 16 principles has been developed to support TMM-AM design. For example, a testing process assessment model should:

1. be based on a testing maturity model as its reference model
2. support test process improvement so that an organization is able to achieve software product and process quality goals
3. provide a profile of an organizations' testing process capability
4. assist an organization to make decisions on where to improve its testing process in order to achieve testing process maturity
5. be integratable with other assessment models
6. provide high quality data, and repeatable, reliable results
7. provide visibility to the testing process.

Based on the 16 principles, the SW-CMM Assessment Model, SPICE, the CAF [7-9], and the specific characteristics of the software testing domain, we have identified a set of inputs and outputs, and developed a set of components for the TMM-AM which will be described in the next sections of this article.

4.0 The TMM Assessment Model Components

The TMM-AM has three major components: 1. the assessment instrument (a questionnaire), 2. the assessment procedure, and 3. team training and selection criteria. A set of inputs and outputs is also prescribed for the TMM-AM. The relationship between these items is shown in Figure 3. A discussion of the components follows.

4.1 The Assessment Procedure

The TMM-AM assessment procedure consists of a series of steps that guides an assessment team in carrying out a testing process self-assessment. The principle goals for the TMM assessment procedure are: 1. to ensure the assessment is executed with efficient utilization of the organizations' resources, 2. to guide the assessment team as to who to interview, and how to collect, organize and analyze assessment data, 3. to support the development of a test process profile and the determination of a TMM level, and, 4. to guide the assessors in developing action plans for test process improvement. A brief summary of the steps in the assessment procedure follows:

<u>Preparation:</u> The activities called for in this step include: selecting and training the assessment team, choosing the team leader(s), developing the assessment plan, selecting and preparing the organizational units that are participating in the assessment. In this step

team members also prepare a statement of assessment purpose, scope, and constraints that is used to guide the development of the assessment plan.

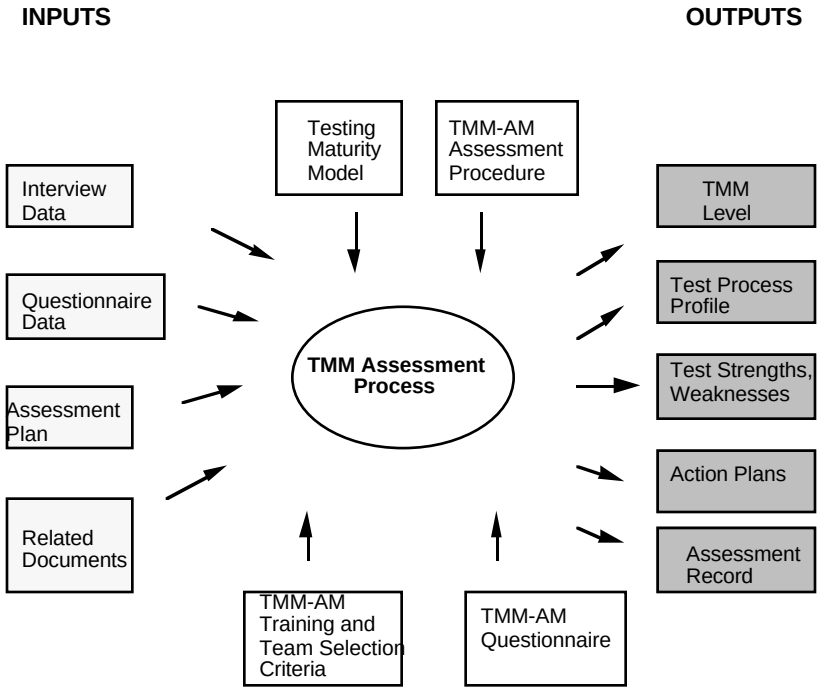


Figure 3. The TMM Assessment Process: Components and Input/Outputs

Conducting the Assessment: In this step the team collects and records assessment information from interviews, presentations, questionnaires, and inspection of relevant documents. All the information collected must be protected by a confidentiality agreement. The TMM level of the organization, which is a measure of its current testing maturity level, is determined by analysis of the collected data and use of a ranking algorithm.

The ranking algorithm we have developed for the TMM-AM is similar to the algorithm described in the work of Masters et. al. in their work on the Capability Maturity Model Appraisal Framework (CAF) [8]. The TMM ranking algorithm requires first a rating of the maturity subgoals, then the maturity goals, and finally the maturity level [10]. Our "degree of satisfaction" measure with respect to the maturity subgoals and goals is more fine-grained then the corresponding measure in existing assessment models. We believe that our approach provides more detailed information for identifying test process strengths and weaknesses.

Reporting the Assessment Outputs: The TMM-AM outputs include: a process profile, a TMM level, a statement of test process strengths and weaknesses, and the assessment record. The assessment outputs can be delivered as a presentation, and /or a written report (the final assessment report). The assessment team prepares the process profile which gives an overall summary of the state of the organizations' testing process. The profile is based on analysis of the assessment data, and results of the ranking process. The profile can be presented in the form of a matrix that indicates maturity goals and subgoals at each level of the TMM and their "degree of satisfaction". The profile also includes the TMM level, a summary of test process strengths and weaknesses, as well as recommendations for improvements.

The assessment record is also completed in this step. It is a written account of the actual assessment that includes: names of assessment team members, assessment inputs and outputs, actual schedules and costs, tasks performed, task durations, persons responsible, data collected, and problems that occurred.

Analyzing the Assessment Outputs: The assessment team now uses the assessment outputs to identify and prioritize goals for improvement. An approach to prioritization is described in Homyen [10]. Quantitative test process improvement targets need to be established in this phase. The targets will support the action plans that are developed in the next step.

Action Planning: In this step an action planning team develops actions plans that focus on improvements in the high priority areas that have been identified in the previous step. The action planning team can include assessors, software engineering process group members, SQA staff, and/or opinion leaders chosen from the assessment participants [11]. Inputs to action planning include the final assessment report, the process profile, and prioritized areas for improvement. The action plan describes specific actions needed to: 1) improve existing practices, and, 2) support the addition of missing practices, so that the organization can move up to the next TMM level. An action plan, like all other software engineering project plans should include: measurable goals, tasks, responsibilities, resources required, risks and benefits, and reporting and tracking mechanisms. Action planning can be accomplished through the convening of a workshop directed by the action planning team. The result should be a draft of an action plan. The workshop members should also identify pilot projects that will be implementing the new process.

Implementing Improvement: After the action plans have been developed and approved they can be applied to pilot projects that are selected. The process improvement projects need to be monitored and tracked to ensure task progress, and achievement of the target goals. Favorable results with the pilot projects set the stage for organizational adaptation of the new process.

4.2 The TMM Questionnaire

Assessment instruments are needed to: support the collection and recording of information from an assessment, maintain a record of results, and provide information for assessment post mortem analysis. We have selected the questionnaire as our assessment instrument for the following reasons. Use of a questionnaire supports CAF compliance, facilitates integration with other process assessment instruments, insures assessment coverage of all maturity goals and subgoals for each level of the TMM, and provides a solid framework in which to collect and store assessment data.

The TMM questionnaire consists of 8 parts which are: 1. instructions for use, 2. respondent background, 3. organizational background, 4. maturity goal and subgoal questions, 5. testing tool use questions, 6. testing trends questions, 7. recommendations for questionnaire improvement, and, 8. a glossary of testing terms [3,10].

Parts 2 and 3 of the questionnaire are used to gather information about the respondent, the organization, and the units that will be involved in the TMM assessment. The maturity goal and subgoal questions in part 4 are organized by TMM version 1.0 levels, and include a developer/tester, manger and client/user view. The questions are designed to determine to what extent the organization has in place mechanisms to achieve the maturity goals, and resolve maturity issues at each TMM level. The responses are input to the ranking algorithm that determines a TMM level. The testing tool portion of the questionnaire records information about type and frequency of tool use. This information can help the action planning team make recommendations for future tool usage. We added the testing trends section to provide a perspective on how the testing process in the organization has been evolving over the last several years. This information is useful for the assessment team when preparing the assessment profile and the assessment record.

The recommendations component allows each respondent to give the TMM-AM developers feedback on the clarity, completeness and usability of the questionnaire. A complete TMM questionnaire is found in [10]. The questionnaire can also be found on the web site noted in section 5 of this article.

4.3 Assessment Training and Team Selection Criteria

We have designed the TMM-AM with the goal of assisting an organization with self-assessment of its testing process. This will require a trained assessment team whose members are selected from within the organization [10]. Assessment team members should be selected in a manner that ensures that they: understand assessment goals, have the proper knowledge, experience, and skills to participate in an assessment, and are committed to test process improvement . The assessment team size should be appropriate for the purpose and scope of the assessment.

We have adapted SPICE guidelines for selecting and preparing an effective assessment team [9]. Preparation is conducted by the assessment team leader who is experienced in TMM assessments. Preparation includes topics such as: an overview of the TMM, process management concepts, interviewing techniques, data collection, and data analysis. The training activities include: team-building exercises, a walk through the assessment process, filling out a sample questionnaire, performing data analysis and learning to prepare final reports.

5.0 Forms, and Tools for Assessment Support

Carrying out a TMM assessment is complex task. To support the assessment team we have developed several forms and templates as well as a tool that implements a distributed version of the TMM questionnaire to support a TMM assessment team [10,12]. These tools are important to ensure that the assessments are performed in a consistent, repeatable manner, to reduce assessor subjectivity, and to ensure the validity, usability and comparability of the assessment results. Tools and forms help to collect, formalize, process, store and retrieve assessment information thus making effective use of organizational resources. The tools and forms we have developed include the Process Profile and Assessment Record forms which have been described in previous sections of this article, as well as the following:

<u>Team Training Data Recording Template:</u> This template allows the team leader to record and validate team training data. This template can be used in future assessments to make any needed improvements to the assessment training process.

<u>Traceability Matrix:</u> This matrix which is filled in as assessment data is collected, allows the assessors to identify sources of data, record any problems associated with the data, and resolve data related issues. This is to ensure the integrity of the data to be used in the ranking algorithm and the resulting process profile.

<u>Web-based Questionnaire:</u> A complete version of the TMM-AM questionnaire appears at the web site: "http:\\www.cs.iit.edu\~tmm". The web-based questionnaire was designed so that assessment data could easily be collected from distributed sites and organized and stored in a central data repository that could be parsed for later analysis [12]. The on-line questionnaire was developed using an Internet HTML-based development tool. The design allows the tool to run on multiple operating systems and collect data from users around the world, thus providing support for test process assessment to local and global organizations. A detailed description of tool development is given by Grom [12]. The web-based questionnaire and links to supporting information related to the TMM is found at the site named above. Comments and recommendations are welcomed by the authors.

6.0 Experimental Results Based on Questionnaire Usage

Software engineers from two software development organizations have evaluated the TMM questionnaire. They also applied the TMM questionnaire to 3 development groups in their organizations (one engineer evaluated one group, the other evaluated 2 groups). A brief background description of the engineers, organizations and groups follows.

Organization 1 tests and evaluates household hardware items. The development group evaluated using the TMM questionnaire consists of 4 software engineers. The group develops office and laboratory software to support the hardware testers. These products are for in-house use only. Engineer 1 manages this group. He holds a bachelors degree in Computer Science and has held a full-time position at this company as a software engineer for 6 years.

Organization 2 develops both hardware and software products for the telecommunications industry. The groups involved in the TMM evaluations consist of 8 developers and 5 dedicated testers for Group 1, and 14 developers and 3 testers for Group 2. Group 1 develops embedded software for the control of cellular test equipment. Group 2 develops software to collect and display real-time cellular network information. The size of the products developed ranged from 10,000 to 450,000 LOC for the two groups. Engineer 2 holds a masters degree in Computer Science. He has worked as both a software and hardware engineer, and is currently Manager of Data Collection Software.

The feedback we have gotten from the two engineers through this evaluation process has enabled us to: revise and reorganize some of the TMM questions, experiment with our ranking algorithm using actual industrial data, generate sample action plans, and study problems of testing process improvement in real-world environments. Obtaining and analyzing this industrial data, although on a small scale, has been very useful to our research team.

Applying the TMM ranking algorithm to the questionnaire data from our experiments resulted in a TMM rank of level 1 for each organization. However, the strengths and weaknesses of each of the groups were considerably different. Organization 2 did satisfy some of the maturity goals at the higher levels of the TMM. It was ranked at TMM level 1 because it did not satisfy the level 2 maturity goal to: "Establish Testing and Debugging Goals". Given the quality of the existing process for Organization 2, it should be able to reach TMM level 2 in a relatively short time period. Organization 1 did have some testing strengths such as the presence of a Software Quality Assurance Group, but responses to the TMM questionnaire indicated they could not satisfy any of the maturity goals at TMM level 2.

It should be noted that a full TMM assessment was not performed by these engineers. Data input to the TMM ranking algorithm and output to the test process profile was based only on responses to the questionnaire. In brief, although our experimental results look promising with regard to the usability of the TMM for test process assessment, larger scale experiments in industry are needed to further evaluate and improve the model.

7.0 Maturity Model Integration Issues

During the period in which we have been developing the TMM we have received feedback on our work from software engineers, software testers, mangers, and software quality professionals from over 40 organizations located throughout the world. The feedback was obtained through in-person discussions, phone conversations and correspondence by e-mail [13]. Comments confirmed the need for a Testing Maturity Model since most corespondents believed that existing process improvement models have not sufficiently addressed the special issues relevant to testing process assessment and improvement. An important issue for many practitioners was integration of process improvement models which would result in: 1. a common architecture and vocabulary, 2. common training requirements, and 3. support for performance of parallel assessments in

multiple process areas. Fulfilling these requirements would insure effective use of organizational resources both for assessment, and process improvement efforts.

Our initial development approach was to build the TMM as a complement to the SW-CMM. We believed that it would simplify parallel process improvement drives if both the SW-CMM and TMM had corresponding levels and goals. However in the course of our research we have realized that model integration issues are far more complex than simple level correspondences. Meeting industry requirements for maturity model integration as described in the previous paragraph required us to focus our research efforts in a new direction. This effort has resulted in the concept of a Development Maturity Model (DMM) [14]. The DMM concept is aimed at designers of process improvement models in software development sub domains such as design, testing, and requirements specification. The DMM framework is based on a set of principles which apply to all DMMs, a model architecture common to all DMMs, a process for construction, integration, and validation of a DMM, and a generic assessment model for DMMs. Our goal is to have the resulting DMMs be integratable with other DMMs, and ultimately with the SW-CMM.

Our current DMM research efforts address some difficult issues in process improvement model development and integration such as: how many levels are optimal in a given domain, how should one assign Key Focus Areas (KFAs) to the levels, how should designers integrate the DMM levels and features? The Software Engineering Institute has proposed an approach to CMM integration - the Capability Maturity Model Common Framework (CCF) [15]. We argue that their approach, which has now evolved into the CMM-Integration Project, has not fully addressed many of the important issues described above. The DMM Framework is now being applied to develop version 2.0 of the TMM, as well as a DMM for the software design process. Our goal for the DMM project is to support industry requirements for process assessment and improvement model integration.

8.0 Summary

We have been developing a Testing Maturity Model to assist organizations with assessing and improving their software testing process. Feedback from industry concerning the TMM shows that there is a need for a specialized test process assessment and improvement model. This is especially relevant in light of the Y2K problems that need to be solved.

In this paper we describe the two major components of the TMM: 1. the hierarchy of maturity levels, and 2. the TMM Assessment Model. We begin with a discussion of the internal structure of the levels, as well as the maturity goals for each of the five levels. We then describe our approach to TMM Assessment Model development, assessment inputs and outputs, our assessment procedure, and the forms and tools available to support the assessment team. Our assessment model has unique support in the form of a web-based questionnaire that allows distributed teams to provide assessment data to a centralized data repository for collection, and analysis. Preliminary experimental results using the TMM questionnaire in an industrial setting are also described. While these results are promising, wider industrial application of the TMM is planned to help us to further evaluate its usefulness and effectiveness for test process assessment and improvement. Our future plans also include completion of the DMM research so that we can provide strong support for process improvement model integration. We also plan on the development of more intelligent tools to aid process assessment teams.

9.0 References

[1] I. Burnstein, T. Suwanassart, C.R. Carlson, "The Development of a Testing Maturity Model", Proc. Ninth International Quality Week Conf., San Franciso, CA, May 21-24, 1996.

[2] I. Burnstein, T. Suwanassart, C. R. Carlson, "Developing A Testing Maturity Model", Part I, *CrossTalk*, Journal of Defense Software Engineering, Vol. 9, #8, August, 1996, pp 21-24, Part II, Vol. 9, #9, September, 1996, pp 19-26.

[3] I. Burnstein, A. Homyen, R. Grom, C. R. Carlson, "A Model for Assessing Testing Process Maturity", CrossTalk, Journal of Department of Defense Software Engineering, Vol. 11, #11, November, 1998, pp. 26-30.

[4] M. Paulk, C. Weber, B. Curtis, M. Chrissis, The Capability Maturity Model: Guidelines for Improving the Software Process, Addison-Wesley, Reading MA., 1995.

[5] D. Gelperin, B. Hetzel, "The Growth of Software Testing", CACM, Vol. 31, No. 6, 1988, pp. 687-695.

[6] J. Durant, Software Testing Practices Survey Report, Software Practices Research Center, Technical Report, TR5-93, May 1993.

[7] D. Zubrow, W. Hayes, J. Siegel, D. Goldenson, "Maturity Questionnaire", Technical Report, Software Engineering Institute, CMU/SEI-94-SR-7, June, 1994.

[8] S. Masters, C. Bothwell, "A CMM Appraisal Framework, Version 1.0", Technical Report, Software Engineering Institute, CMU/SEI-95-TR-001, Feb., 1995.

[9] ISO/IECJTC1/WG10, "SPICE Products", Technical Report, Type 2, June 1995.

[10] A. Homyen, "An Assessment Model to Determine Test Process Maturity", Ph.D. Thesis, Illinois Institute of Technology, July 1998.

[11] J. Puffer, A. Litter, "Action Planning", IEEE Software Engineering Technical Council Newsletter, Vol. 15, Number 2, pp. 7-10, Spring, 1997.

[12] R. Grom, "Report on a TMM Assessment Support Tool", Technical Report, Illinois Institute of Technolgy, April, 1998.

[13] I. Burnstein, private correspondence, 1997-99.

[14] G. Saxena, "A Framework for Development Maturity Models", Technical Report, Illinois Institute of Technology, Chicago, IL, March, 1998.

[15] Software Engineering Institute (SEI), Common CMM Framework, Vol. 3, CMM Developer Handbook. Draft E., Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1997.

Copyright (c) 1999, Ilene Burnstein, Illinois Institute of Technology, Chicago, IL. All rights reserved.

The impact of software practitioners on software quality

Nathan Baddoo
baddoon@sbu.ac.uk

Dr Tracy Hall
hallt@sbu.ac.uk

Centre for Systems and Software Engineering
South Bank University
103 Borough Road, London SE1 0AA
United Kingdom

Overview

- Introduction
- Research methodology
- Areas of investigation
- Pilot findings
- Projected work
- Conclusion

Introduction

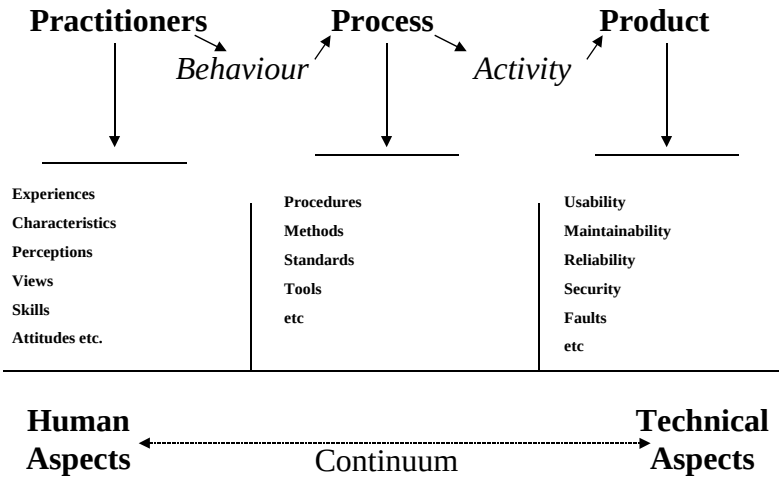
- **Logistics**

  Duration
  Funding body
  Personnel

- **Hypothesis**

  "The attitudes, experiences, perspective of software practitioners towards quality impacts on the processes implemented and hence the quality of the product"

Introduction



Introduction

Aims and Objectives

- **Understand quality views.**
- **Address misalignment.**
- **Develop implementation strategies**.

Introduction

Initial findings

- **Existence of a practice gap.**
- **Evidence of misalignment.**
- **Existence of effective implementation strategies.**

Methodology

Social science techniques.

- **Structured Interviews.**
- **Self-administered questionnaires.**
- **Repertory Grid Technique.**
- **Parasuraman's quality dimensions questionnaire.**

Methodology

Scientific analytical methods.

- **Statistical techniques.**

Current area of research

Process improvement and process assessment models.

- **CMM**
- **SPICE**
- **PCMM**
- **PSP**

Current area of research

Process improvement and process assessment models.
Standards:

- **ISO9001**
- **IEEE standards**
- **Awards**

Pilot findings

Limitations to quality:

- **Pressure.**
- **Speed of industry.**
- **Lack of quality culture.**
- **Commercial motives for certification.**
- **Lack of customised quality initiatives.**
- **Lack of motivation**.

Projected work

- **Larger research sample**
- **Motivation theories**.
- **Process theory.**
- **Psychological states in a job.**

Conclusion

“ Read my lips: no new models”

- Karl E. Wiegers, Head To Head,

IEEE Software September/October 1998.

“By itself, a model cannot lead to good practices - it simply provides a framework for the practices that implement it.”

- Tomoo Mastubara, Head To Head, *IEEE Software* September/October 1998.

The impact of software practitioners on software quality

Nathan Baddoo
Centre for Systems and Software Engineering (CSSE)
South Bank University
103 Borough Road
London SE1 0AA, United Kingdom
baddoon@sbu.ac.uk

Dr Tracy Hall
Centre for Systems and Software Engineering (CSSE)
South Bank University
103 Borough Road
London SE1 0AA, United Kingdom
hallt@sbu.ac.uk

Abstract

Most software quality initiatives focus on process to achieve product improvement. We accept the importance of this, but also believe that the human factors in the processes have not been properly examined. Current process improvement initiatives do not adequately address effective implementation strategies.

In this paper we examine the impact software practitioners have on product quality. We explore the softer aspects of quality improvement, and emphasise the continuum between the human and technical aspects of quality during the development process. We present our preliminary findings of a project analysing the critical factors in software quality improvement. We describe our work on developing an effective framework for the optimisaiton of software quality improvement initiatives.

Keywords: *software process improvement, process assessment, software quality, software practitioner, quality initiative*

1. Introduction

Over the years process assessment and improvement models have increased in prominence [Wiegers 1998]. As a direct result, most software quality initiatives focus on process to achieve product improvement. We accept the importance of this, but also believe that the human factors in the processes have not been properly examined. In particular we believe that the impact software practitioners have on product quality has not been explored.

It is also apparent that process assessment and improvement models have not always delivered improved software quality [Gray & Smith 1998]. In this paper we take a critical look at some of these approaches to the development of quality software and highlight how we feel these approaches have failed. We present preliminary findings of our current project analysing the critical factors in software quality improvement[1]. In this project we explore the *softer* aspects of quality improvement, and emphasise the continuum between the human and technical aspects of quality during the development process.

We analyse the relationship between software practitioners, software quality improvement processes and the quality of the software product. We introduce Soft Systems Methodology [Checkland 1990] to learn about what needs to be optimised in quality improvement initiatives. We show that the way software practitioners feel about software quality, their views of quality initiatives, their experiences of quality initiatives and in general their attitude towards quality, tends to influence how they, as practitioners, behave during the implementation of quality improvement initiatives. These improvement processes then influence the quality of the product. However, to what extent are the effectiveness of the processes predetermined by the attitudes and actions of the practitioner? It is the aim of this project to explore the link between practitioners' behaviour and the effect of this behaviour on the processes. Since it is believed that processes ultimately determine quality [Paulk et al 1995], we will analyse the indirect link from practitioner behaviour to product quality, see figure 1.

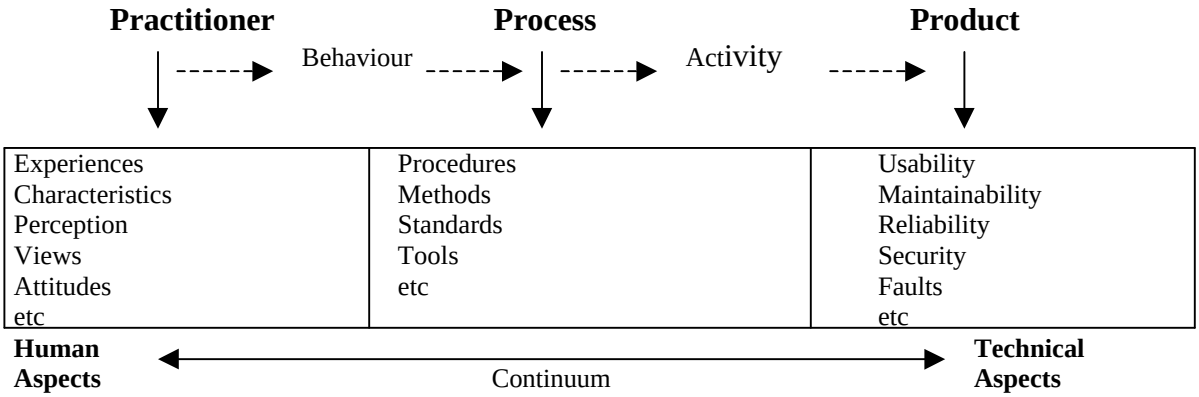


Figure 1: The PPP Quality Process Model (adapted from Hall and Wilson 1997)

[1] This project is being funded for three years by EPSRC under grant number **EPSRC GRL/91962**

The paper is structured thus; In Section Two we describe the research methods adopted for this project. In Section Three we introduce some of the existing software quality improvement initiatives and discuss the implementation limitations of these models. In Section Four we present findings from our pilot of this project. We summarise and outline our future intention in Section Five.

2. Research methodology

We make extensive use of social science research methods to measure and analyse practitioner attitude. Although it is unusual to use these methods in the study of software engineering they are widely used and have been thoroughly validated in other disciplines. For example, in the field of health care, the methods we describe have been successfully applied to improve quality [Edwards & Browme 1995].

We collected attitudinal data via various social science research methods including Repertory Grid Technique (RGT) [Fransella & Bannister 1977]. This has enabled us to investigate the differing perceptions, views and experiences of quality that exist within different groups of software practitioners.

In this paper we report on empirical work carried out with software developers in UK companies. We have sampled and collected information from a cross section of practitioners from five UK companies as a pilot to this project. The pilot study results have led us to be particularly interested in the attitudes of project managers, developers and senior managers. The findings of this earlier research serves as the basis of our current work [Hall and Wilson 1997]. We are now testing these findings on a larger sample of software companies within the UK.

The design of this project allows us to utilise a combination of evaluation techniques. We begin by exploring concepts via social science techniques and gradually move towards more scientific analysis. In the process we collect a combination of qualitative and quantitative data. We believe that this combined approach ensures the quality of our results.

3 Current quality improvement initiatives

In our previous work [Hall & Wilson 1997] we identified three key groups of practitioners in software quality: software developers, middle managers and senior managers. Within these groups we identify misalignment in the perception of software quality, for example: there is a gap between what developers and senior managers think about what constitutes effective quality initiatives. We suggest that the current quality initiatives do not contain measures that address this issue of misalignment. We examine some of the quality improvement initiatives in use at the moment and evaluate what can be done to optimise their deployment.

3.1 Process Improvement /Assessment Models

Process improvement and assessment models exist to guide software improvement programs [Wiegers 1998]. But how effectively are these models? What is it about these models that inhibit effective application? We believe that the expected improvements from using these models have not been fully realised because the models themselves have not been properly implemented. In this section we examine a few of the well established models currently being used in industry and some of the recent approaches to ease their implementation.

3.1.1 The Capability Maturity Model

The Capability Maturity Model (CMM) is highly relevant to this research. This is because we recognise that CMM has been the basis from which most other assessment models have been derived [Gray and Smith 1998]. Hence in order to examine the effectiveness of assessment models in getting practitioners on board the quality improvement initiative, it is imperative to examine the one model which has been the subject of the widest interest.

CMM is made up of clearly defined levels and detailed definitions of the key process areas and key practices. The model, however, fails to detail how to implement those key practices. In effect it serves as a guide to the principles and practices underlying process maturity [Paulk et al 1995], but fails to suggest effective implementation strategies. Like a road map it details the route to a destination, but falls short at explaining how to get there.

3.1.2 The People Capability Maturity Model

The People Capability Maturity Model (P-CMM) [Curtis et al 1995], offers a guide to improving the company workforce through evolutionary stages. P-CMM concentrates on workforce activities and offers a way of integrating these with process improvement to establish a culture of excellence. Its added advantage is that it serves both as a standard and a guide. It is however limiting in terms of the context of this research in that, very much like CMM, it does not specify how to implement the practices that form the processes which in turn bring about the required maturity.

3.1.3 SPICE

The Baseline Practice Guide (Software Process Assessment Part 2: A model for process management) defines methods for measuring the implementation and institutionalisation of selected processes in the software development cycle, but again fails to define any implementation strategies. SPICE harmonises existing approaches to process improvement though hesitates to recommend specific paths for improvement. It leaves the determination of a specific improvement path to the practicing company. We argue that this by itself is a shortcoming of the model as it gives no direction as to how to achieve improvement.

3.1.4 The Personal Software Process

The Personal Software Process (PSP) [Humphrey 1994] addresses how individual practitioners can inject quality practices into their software development process. PSP examines the Human Activity System (HAS) and concentrates on suggesting methods to improve defect management and accuracy estimation. As a model, PSP focuses on these two areas as a key to achieving personal process improvement. By addressing individuals, PSP is unique as a process model. Its failure, however, is its unwillingness to examine how individuals' perspective on software quality influence the practices that they undertake.

The process of collecting data to illuminate practitioners' work practices may lead to the elimination of defects and better quality software eventually. It is important to point out though that the process of collecting data itself does not change anything. It does not, in any way, offer suggestions as to how and why these defects are occurring. This makes the PSP diagnostic, but wouldn't it be better if we had a model that was prognostic?

3.2 Standards/Awards/Certification

"For standards to be relevant and useful, they must be guaranteed -to the extent possible – to improve quality and process efficiency."
Lin Zucconi [McConnel 1998].

Standards must also serve as an important information and knowledge base for the personnel in a company [McConnel 1998]. They should provide practitioners with common processes, hence working towards eliminating misalignment. However, in order to achieve this, standards must be developed by consensus, encompassing a variety of views, backgrounds and experiences [McConnel 1998]. Thus enabling the industry to achieve certification at a broader level.

However, the effort required to generate a generic set of standards that take into account the issues raised above is considerable. As a result what one finds in industry are compromises; standards written for specific products and processes.

In the light of the above we examine some quality standards/certification/awards and offer a critique as to where they fail in optimising quality improvement initiatives.

3.2.1 ISO Standards; e.g. ISO9001

ISO9001 provides broad guidelines to software developers on how to implement, maintain and improve a quality system capable of ensuring high quality software, [Ince 1994]. In a way it is a prescriptive model in a way that other standards that we will come to describe are not. It encourages companies to examine, critically, their internal

quality management systems. We find, however, that effective though ISO9001 may be at addressing basic issues like configuration management, documentation control, quality audit etc, it does not define a process of attaining the benchmarks that it measures. A failure to address implementation issues, represents a failure to address the human activity systems that enable successful implementation.

3.2.2 IEEE Standards

IEEE standards are not as international as ISO standards. They are very specific and effectively address specific processes. Some may argue, however, that IEEE standards have a US bias. Which means they fail to take into account the differences in the scale of software development in the US compared to other countries.

" For example, a small development group of 50 in the US may be considered a large development group in Australia"
Val Veraart [McConnel 1998].

Hence for example, practices designed for a large scale project in one context, may be totally inappropriate in another. Thus in tailoring IEEE standards to fit non US companies, some of the human activity inter-play may be misconstrued or lost.

3.2.3 Awards; Baldrige – The National Quality Award (NQA)

NQA is awarded annually to US companies which excel in quality achievement and quality management [Lizotte 1994]. The European quality award is derived from the Baldrige criteria and thirty two countries have quality awards derived from the same criteria [Wernham 1992]. We find, however, that the award focuses on the organisation as a whole as opposed to software development. Even though in applying for this award companies need to demonstrate their commitment to involving all employees in the total quality process, the award addresses aspects like profitability and customer satisfaction far more than it does software processes.[2] This, we feel, takes away the focus required for an extensive examination of software processes and hence fails to identify the human factors in these processes.

Overall, we find that what standards invariably do is remind software houses of what systems to have in place that will enable the production of high quality software, but never describe ways for implementing quality systems. Our previous research shows the implementation process is critical [Hall & Fenton 1997].

We also have found that standards provide little guidance on how to manage the people involved in the production process.

Furthermore, certification, *per se*, does not always promote the implementation of best practices: it serves as a benchmark which companies can use to measure themselves. This has the effect of reducing the benefits that companies can gain. As a result companies

[2] Http//www.software.org/quagmire/descriptions/baldrige.

may achieve certification without having adequately benefited from the measures deployed to get there. This can occur for example, when a company concentrates on achieving a particular maturity level without addressing the underlying processes.

"Maturity levels should be measures of improvement, and not goals of improvement …" [Gray & Smith 1998].

Our analysis suggests that irrespective of how sophisticated and encompassing current process improvement initiatives are, what practitioners lack from these initiatives are effective guidelines on implementation. However, in order to address this problem, we need to examine and understand practitioners' attitudes, views and experiences of software quality in general and towards quality improvement initiatives in particular.

We need to examine and understand not only the quality perspective of practitioners, but also how they arrive at their views. We need to address the cognitive processes that contribute to formulating specific attitudes towards quality. In the following section, we introduce the findings of our pilot research to ascertain the views of practitioners on software quality improvement initiatives [Hall & Wilson 1997].

4. Impediments to quality

We have identified the following areas as those that both impede and influence software quality initiatives, from practitioners' perspectives.

4.1 Pressure

Software practitioners are forced to compromise on quality due to the pressure brought upon them by users on the one hand and senior managers on the other [Demarco and Lister 1987]. We examine how this culture of "flight from excellence" [Demarco and Lister 1987] leads to ingrained attitudes about quality initiatives. The concept of flight from excellence explains that for software to be delivered quickly senior managers pressurise practitioners to meet development deadlines at the expense of quality. The market share lost to the development house, by virtue of the fall in quality standards, is more than made up for by the increased profitability of the product. As a result, there is a view within the software industry, and amongst writers on quality that the quality agenda is being dictated by market forces.

In our previous work, we have identified that on the one hand software practitioners and project managers find themselves under immense pressure to meet deadlines and on the other hand to maintain high quality standards. Consequently they find themselves sandwiched between the demands and wishes of users and of senior managers.

4.2 Speed of industry

The evolving nature of the software industry creates a situation where software practitioners view software process improvement initiatives as a facet of that evolutionary climate [Peat 1994]. Over the years there have been many initiatives introduced to address the issue of software quality improvement: the most current of which are process improvement and assessment models. We believe that, very much like the software product, improvement initiatives evolve too fast to enable practitioners to get used to them. Consequently, practitioners see these initiatives as an adjunct of intellectual fashion. *Here today, gone tomorrow*. Such an outlook is likely to influence practitioners' views on quality.

To overcome this, we believe that software process improvement initiatives need to be small and slow in order to be effective. Our previous research [Hall & Wilson 1997] confirms this fact and also reveals that practitioners are more responsive to initiatives which are nurtured and encouraged within the company.

4.3 Culture

We identify the problems that the lack of a quality culture will inevitably introduce. For example, we recognise the concept of the "maverick" practitioner; the individual who perceives quality initiatives as bureaucratic, disruptive and undermining creativity [Hovenden et al 1996]. Hovendan et al recognise that contrary to what may seem obvious, such individuals "are of the highest calibre and central to the technical success of the product". However without the integration of such individuals into quality improvement initiatives, the overall initiative is bound to be disrupted by their non-participation [Hovenden et al 1996].

On the other hand we identify the existence of the quality "enthusiast"; the individual who recognises the validity of quality improvement initiatives and adheres religiously to the quality models implemented in the company. However without the consolidated approach by a software development team as a whole, we envisage the efforts of the "enthusiast" being set back by the apathy of the "mavericks".

We have come across evidence of such misalignment in our previous work [Hall and Wilson 1997]. We argue that to overcome such misalignment, companies must consider more holistic quality improvement initiatives.

By developing a quality culture (for example key quality practices are instituted, software quality guidelines are documented and formalized into the development process) practitioners are more likely to adhere to long-term and sustainable process improvement initiatives [Ludewig 1994]. The assertion here is that a quality culture will sustain the individual's effort. That a company with an appropriate quality initiative in place and one that encourages quality improvements at every stage in the development process and within every aspect of their operation will have individuals practicing quality improvement measures without consciously being aware of it. Hence inspite of, or

irrespective of, individual practitioner's efforts towards or against quality, initiatives will only work if the company as a whole embraces a culture of quality.

4.4 Commercial motives for certification

Work that we have done in the past [Hall and Wilson 1997] suggests that companies embark on quality initiatives mainly for marketing purposes. Indeed we have identified some companies that obtain quality certification for marketing reasons. In a situation like this, we believe the whole quality improvement initiative is jeopardised as quality improvement schemes appear to practitioners as having been thrust upon them. Hence practitioners may resist these initiatives and weaken the coherency that is needed for successful SPI deployment, as stated earlier.

4.5 Lack of customised quality initiatives

We have also identified that for most quality initiatives to be successful, there is a need to tailor particular schemes to specific projects. A "blanket" approach to the application of quality schemes usually does not achieve optimum effectiveness. Current software process improvement initiatives tend to be too general in nature. Models like CMM are too static and do not picture different processes for different organisations and situations [Gray & Smith 1998].

We continue to investigate the assertion that some practitioners find such quality improvement initiatives difficult to implement because they view them as external standards which do not address the quality issues of their particular companies.

4.6 Motivation

A survey of software practitioners found that practitioners have motivational problems. [Warden & Nicholson 1995].

The above survey identified four key areas where the motivational profile of practitioners can be analysed [Warden & Nicholson 1995]. In their findings, Warden and Nicholson identified that there is a need for some sort of interaction, correlation even, between the "core job dimensions, the employee growth need strength, the critical psychological states and the personal and work outcomes" of the practitioner. For example, there is a need for some correlation between the strength of practitioners' growth needs and the motivational potential of their work. In a nutshell, if the core job dimensions of the practitioner embody a combination of tasks and skills which are highly motivating, then provided the practitioner has got matching needs for personal growth, they will be able to achieve critical psychological states that will be reflected in the quality of the work they produce [Warden & Nicholson 1995].

Work that we have done thus far suggests that current software quality improvement initiatives do not adequately tailor the core processes and key practices required to optimise quality, to any of the above motivational areas [Baddoo & Hall in progress]. Hence practitioners are involved in the implementation of practices which do not match

their motivational needs. This, we believe, impacts negatively on the coherence of the whole software quality initiative.

In our current work, we continue to examine motivation theories and investigate ways of incorporating suggested models into quality improvement initiatives. We will explore some process theories, for example stimulus response theory and job characteristics theory, which explain people's motivational behaviour. However, there is a general contention, echoed in many areas, that software practitioners have unique characteristics [Khalil et al 1997]. As a result it will be necessary to investigate specific motivational factors for practitioners before deciding on incorporating any specific characteristics into process improvement initiatives.

We will also look at some of the psychological states in a job that lead to high levels of satisfaction and quality of performance [Couger & Zawacki 1980] and suggest ways of harnessing these into appropriate frameworks.

5. Conclusion

Our research thus far suggests that the determinants of an effective quality system are very complex [Hall & Wilson 1997]. However, it also confirms our earlier findings that a significant part of that complex conundrum relates to effectively managing software practitioners. In this project we have discussed several ways by which that can be achieved. We have examined the weaknesses in some of the current software quality initiatives. We have also discussed what we believe are the influences and stumbling blocks to quality practices amongst practitioners.

We do not, however, recommend the adoption of new models to combat this shortcoming. We believe that there are sufficient SPI models and tools available at the moment, and that for the majority of companies, the current aids are sufficient. What we advocate and promote is a comprehensive strategy to address the implementation shortcomings of SPI initiatives. As Tomoo Matsubara eloquently puts it:

"By itself a model cannot lead to good practice.
It simply provides a framework for the practices that implement it"
Tomoo Matsubara [Wiegers 1998].

We re-emphasise that "how" to implement process improvement and assessment models, is vital.

We also subscribe to the view that in order to optimise quality improvement initiatives, it is necessary to customise and calibrate quality mechanisms to particular development environments [Hall 1995]. However, in order to have controlled calibration and hence improvement, companies need to understand their practitioners. Understand their views, experiences, perspective and in general, attitudes towards quality. These attitudes, as we

have asserted earlier on, vary amongst practitioner groups. Thus to address this, in our future work we will present a characterisation of the different attitudes that software managers and developers have towards product and process quality and suggest methods that will enable companies to understand the quality views of practitioners.

We will also develop a comprehensive set of strategies and guidelines, suggesting methods of reducing misalignment. These can be used to effectively manage software practitioners within the quality improvement process.

Finally, we will develop strategies and support mechanisms, based on alignment, for companies to effectively implement software quality improvement processes.

References

Baddoo, N. and Hall, T. (in progress), Optimizing Software Process Improvement Initiatives

Checkland, P. and Scholes, J. (1990), *Soft Systems Methodology In Action*, John Wiley & Sons

Couger, J. D. and Zawacki, R. A. (1980), *Motivating And Managing Computer Personnel*, John Wiley and Sons

Curtis, B., Hefley, W. E. and Miller, S. (1995) *Maturity Model - People Capability Maturity Model*, CMU/SEI-95-MM-02

DeMarco, T. and Lister, T. (1987) *Peopleware - Productive Projects And Teams*, Dorset House

Edwards, S. and Browne, M. (1995), Quality In Information Services, *Library And Information Science Research*, 17

Fransella, F. & Bannister, D., Eds. (1997) *A Manual For Repertory Grid Technique*, London, Academic Press

Gray, E. M. and Smith, W. L (1998) On The Limitation Of Software Process Assessment And The Recognition Of A Required Re-orientation For Global Process Movement, *Software Quality Journal*, (7), p.21-34

Hall, T. (1995) What Do Developers Really Think About Software Quality, In: Quality Management III Vol 1, [Ross et al eds], *Quality Management*, Computational Mechanics Publications, p. 359-368

Hall, T. and Wilson, D. (1997) *Views Of Software Quality: A Field Report*, IEE Procs On Software Engineering, April

Hall, T. and Fenton, N. (1997), *Implementing Effective Software Metrics Programmes*, IEEE Software, March

Hovenden, F. M., Sharp, H. C., Walker, S. D. and Woodman, M. (1996), *Building Quality Into Scientific Software*, Software Quality Journal, March (5), p.25-32

Humphrey, W. S. (1994), The Personal Process In Software Engineering, *Proceedings Of The Third International Conference On The Software Process*, Reston, Virginia, October 10-11, 1994, p.69-77

Ince, D (1994), *ISO 9001 And Software Quality Assurance*, McGraw-Hill

Khalil, O. E. M., Zawacki, R. A., Zawacki, P. A. and Selim, A. (1997), *What motivates Egyptian IS Managers And Personnel: Some Preliminary Results*, SIGCPR 97, ACM

Lizotte, R. (1994), NQA And ISO9000, *Software Quality Management*, Autumn (23), p.29-33.

Ludewig, J. (1994), People Make Quality Happen (Or Don't), In: *Software Quality Concern For People: Fourth European Conference On Quality*, VDF, p.11-21

McConnel, S., Moderator, (1998), Head To Head; Weighing In On Standards, *IEEE Software*, November/December, p. 92-102.

Paulk, M. C., Konrad, M. D., and Graydon, A. W. (1995), *CMM Versus SPICE Architectures*, submitted to IEEE TCSE Software Process Newsletter, p.7-11.

Peat, M. M. (1994), Quality - Yesterday's Fashion? In: Software Quality Management II Vol. 1: *Managing Quality Software*, [Ross, M. et al 1994], Computational Mechanics Publication.

Warden, R. and Nicholson, I. (1995), IT Quality Initiatives At Risk, *Software Quality Management*, New Year (24), p.24-27.

Weigers, K. E. (1998), Head To Head: Read My Lips: No New Models!, *IEEE Software*, September/October (5), p.10-13.

Wernham, B. (1992) Letter From America, *Software Quality Management*, Spring (14), p.20-21.

Biographies

Nathan Baddoo

Nathan Baddoo is a Ph.D. research student in the Centre for Systems and Software Engineering (CSSE) at South Bank University, UK. Previous research experience was an investigation conducted at Newcastle into the deployment of CASE in systems development in large organisations in the United Kingdom. Nathan spent a year in the information systems department of Nissan Manufacturing UK Limited, developing information systems.

Dr Tracy Hall

Tracy Hall is a principal lecturer and head of the Centre for Systems and Software Engineering (CSSE) at the School of Computing and Information Systems Management at South Bank University, UK. For the past six years she has been active in the area of software quality and software measurement. She has numerous publications in this area.

All of the research that Tracy Hall undertakes is in direct collaboration with companies in the software industry.

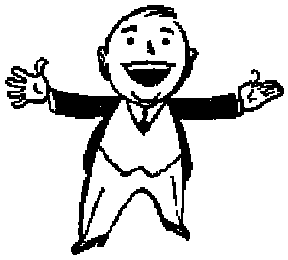
Over the last four years she has been a programme committee member of the British Computer Society's annual Quality Management conference. She is also an active member of the UK think tank: the Centre for Software Reliability Council. She is the co-editor of a book on professionalism of software practitioners, "The Responsible Software Engineer", Springer, 1997 and has chaired highly successful international conferences in that area.

Process Diversity

How I Stopped Worrying and
Learned To Love Chaos

Michael Deck
Cleanroom Software Engineering, Inc.

An SPI Story: Enthusiasm



- Desire for rapid process improvement to address key project risks
  - requirements traceability
  - control process change
  - requirements stability
  - documentation of project commitments
- Initial definition of a single SPI path
  - formal specification of requirements
  - traceability established through verification review
  - process and project documentation

Cleanroom
Software Engineering, Inc.

Copyright © 1999 Cleanroom Software Engineering, Inc. All Rights Reserved

SPI Story: Initial Success

- Initial goals mostly achieved
  - Functions delivered as specified
  - Highly maintainable, robust design
  - Requirements discoveries through spec process
  - Reviews enabled knowledge diffusion
- Although
  - Learning curve contributed to late 1st delivery
  - Significant investment of effort
- Development team enthusiastic about process!

Cleanroom
Software Engineering, Inc.

SPI Story: Other Risks Emerge

- Basic practices
- Management buy-in
- No project is an island
- Staffing stability
- New transfers-in had different backgrounds
- "Old timers" especially resistant to change
- Core team not able to enforce compliance

Cleanroom
Software Engineering, Inc.

Copyright © 1999 Cleanroom Software Engineering, Inc. All Rights Reserved

SPI Story: Resolution

- Multiple simultaneous processes
  - "Basic" process for most development
  - "Advanced" process for key components
  - "Experiment" process for prototypes and one-offs
- Additional management
- But more productive overall

Cleanroom
Software Engineering, Inc.

Lesson 1: Basic Practices

- Requirements management
- Software project planning
- Peer review
- Quality assurance
- Software configuration management
- Metrics
- Defect prevention

Cleanroom
Software Engineering, Inc.

Copyright © 1999 Cleanroom Software Engineering, Inc. All Rights Reserved

Broad Consensus

- Must be universally accepted or
  - lost in crunch mode
  - lost in cost-cutting
  - lost in staff turnover
- Identify minimum level to achieve universal acceptance
  - then strive for improvement based on results
  - quantify value of new practices
- Example: defect prevention

Cleanroom
Software Engineering, Inc.

Lesson 2: Practice Selection

- Use risk analysis
  - risk identification
  - risk prioritization
- Work backwards from WIBNI processes, tools, and models, e.g.
  - what risk(s) does obtaining ISO certification mitigate?
  - what risk(s) does using object-oriented analysis mitigate?
- Think about earlier problem areas
- Be honest!

Cleanroom
Software Engineering, Inc.

Copyright © 1999 Cleanroom Software Engineering, Inc. All Rights Reserved

Practice Improvement

- Smaller steps
  - easier to add to basic consensus of practice
- Quantitative value measurement
  - less disagreement over usefulness
- Example:
  - past situation: bad fix
  - risk areas: SCM, defect prevention
  - specific risks: current SCM tools, no reviews
  - near-term: SCM tool, peer review after baseline
  - success criteria: no defects due to SCM

Cleanroom
Software Engineering, Inc.

Lesson 3: Plan for Diversity

- Different projects and even components ought to have different process standards, e.g.
  - one-off interface test doesn't have same reliability needs as change to security kernel
  - avionics software very different from accounting
  - reliability, usability … all the "ilities" can be tailored
- Accommodate pilot projects for practice improvements

Cleanroom
Software Engineering, Inc.

Copyright © 1999 Cleanroom Software Engineering, Inc. All Rights Reserved

ProcessWeb Example

ID	Description	Basic	Advanced	Minimal
011	Use formal ‘B’ language specifications for every method	Optional	Required	Optional
012	Conduct ‘buddy’ review of all code created or changed	Required	Recom-mended	Optional
013	Conduct formal inspection of all code created or changed	Optional	Required	Optional
014	Track all labor hours spent in review/inspection	Recom-mended	Required	Optional

Cleanroom
Software Engineering, Inc.

Diversity and...

- SEI Capability Maturity Model
  - Maturity level probably defined by <u>common</u> subset
  - Evaluate all defined processes against KPA's
- ISO-9000
  - Each process must be well-defined within ISO framework with documents and records
  - Developers must be constantly aware of which process they are using for which work effort

Cleanroom
Software Engineering, Inc.

Copyright © 1999 Cleanroom Software Engineering, Inc. All Rights Reserved

Summary

- Define a universally-accepted baseline process
- Try new practices based on risk analysis
- Evaluate results of new practices for inclusion in baseline
- Manage process diversity to avoid chaos

Cleanroom
Software Engineering, Inc.

Copyright © 1999 Cleanroom Software Engineering, Inc. All Rights Reserved

Process Diversity

How I Stopped Worrying And Learned To Love Chaos

MICHAEL DECK
CLEANROOM SOFTWARE ENGINEERING, INC.

Process improvement efforts will fail if they assume processes are uniform across large organizations. The basic set of practices should be uniform, but certain projects or subprojects will have needs that require specialized processes. This experience report will share some techniques of process diversity management, and will describe lessons learned.

Introduction

This paper presentation describes techniques for managing **process diversity,** by which we mean having several different process variants in simultaneous use in a particular organization. The techniques presented are largely the result of one project's experience over a multi-year effort. The characteristics of this project which led to a need for process diversity are, I believe, common in the software industry.

Before going into the lessons learned from this project, let me describe why I think this project is typical of many projects in the software industry.

An SPI Story

The story of this project begins with an enthusiastic small team of developers. The developers are relatively inexperienced, which makes them a good target for software process improvement (SPI) because they haven't learned to succeed by working within an existing flawed process. These developers have almost no software engineering training or experience, although they have been programming for some time. I distinguish between programming and software engineering: the latter includes things like requirements analysis, metrics, team organization, project estimating and planning, conducting team and technical reviews, and developing design documentation.

Copyright © 1999
Cleanroom Software Engineering, Inc.,
All Rights Reserved.

Initial Risk Analysis

This team began their SPI effort with an analysis of the risks their project faced. The initial risk assessment used a combination of sources to come up with a starter set of "typical" project risks. The Software Engineering Institute's Taxonomy of Risks [1] was the primary source, but it was supplemented by a risk-centric analysis of the SEI CMM and the Baldrige award criteria. The risk assessment resulted in approximately 12 risk areas classified as "high," 9 risk areas classified as "medium," and 11 risk areas classified as "low." These counts reflect the combination of over 100 individual risk criteria that were assessed. Among the high risk areas were existing issues including:

- No established process for requirements tracking;
- Insufficient confidence in the requirements process;

and potential risks including:

- Inconsistent planning and/or documentation of project activities and commitments;
- Uncontrolled change to processes;
- Instability of requirements, specifications, and designs.

A Single SPI Path

Based on the risk analysis conducted, the team developed a process-improvement plan to treat these risks in approximate descending order of salience. The process improvements were integrated with an incremental development model, the idea being to deliver product code incrementally so that it could be integrated piecemeal with the target hardware (also under development at the time).

The process improvement plan had several characteristics that are important for this discussion. First, it set a very high initial standard for software engineering practices. These practices included:

- Formal requirements specifications using the "box structures" approach [2].
- Reliance on team verification review for most traceability analysis.

- Numerous new documents including a detailed project plan, user-view requirements, and formal specifications at the system, component, class, and method levels.

One aspect of this plan that did not seem significant at the time was the fact that it was intended to apply uniformly to *all* project code.

The team embarked on this learning process enthusiastically and unhesitatingly.

Initial Success

The first increment achieved most of its stated goals. Through the review process, it met an important requirement to train team members in the necessary domain knowledge. The formal specification approach uncovered numerous significant architectural and requirements "gotchas" that would likely have arisen much later in a traditional process. The documentation created was useful in focusing discussion of design tradeoffs and alternatives.

The project also delivered significant functionality in the first increment, demonstrating the capability of both the process and the team to meet critical user requirements.

One unexpected result of the intense documentation and review effort was the creation of a surprisingly robust and maintainable architecture. Although little of Increment 1 remains in today's product (due to a switch from C to C++), most of the architectural elements in today's Increment 7 product were present in Increment 2, after almost two years of additional work.

The large learning curve did delay delivery of the first increment, though there were a number of unplanned outside events that also took their toll on the schedule.

Perhaps most importantly, the team that developed increment 1 was extremely enthusiastic about the "Cleanroom" process they had used [3], and were committed to continuing its use.

Other Risks Emerge

Unfortunately, the initial risk assessment could not account for all of the actual risks that were to appear. Much of this was due to changes in the project's environment, but the fact remains that by increment 2 there were significant new risks. These risks did not fully manifest themselves until increments 3 and 4, which is another reason why they were not addressed quickly enough.

The most significant change in the project's environment was the assumption of staffing stability. The fact that this project's process was so different from that of other projects in the same organization meant that personnel who transferred into it had to climb its learning curve before they could become productive. That placed a large burden on the team to train new members.

We also underestimated the impact of not having the team trained in basic software engineering principles. This, too, contributed to a large learning curve as team members struggled with the basics of inspections, requirements, and formal testing.

With its own unique process, the project was expending more and more energy explaining and defending that process to upper management. The battles over "why should we do anything different" were waged both externally with upper management and internally with senior-level transfers from other projects. In the end, the original core team was not able to enforce process compliance and still meet their aggressive delivery schedule.

Resolution

After an increment of partial anarchy, a consensus began to emerge around the possibility that a single software process and a single SPI path might be unrealistic for this project. In particular, we observed that there were really three kinds of activities going on at any one time. Some of the code was being built to support critical elements of spaceborne scientific instruments. This code had to meet the highest level of quality and reliability, but could be commensurately more expensive to build and document. Some of the code was being built to support non-critical components and ground-based instruments. This code did not have to meet the same reliability requirements as the critical software. Finally, there were a lot of necessary experiments to run. For example, would a particular memory-management strategy meet performance objectives? The simplest way to find out was to write a little code and run an experiment, but nobody wanted to subject that code to a formal process of specification, design, review, testing, and documentation. The result is a diverse set of processes in simultaneous use, each of which is tailored to meeting key needs of a software subset.

Adopt Basic Practices

The first and most important lesson from this project's experience is to define a foundation of basic practices upon which there is universal agreement.

Copyright © 1999
Cleanroom Software Engineering, Inc.
All Rights Reserved

Rationale

If you don't develop broad, grass-roots acceptance of the fundamentals, three things are likely to happen when you try to progress to more advanced techniques. First, the staff will resist, either by active subversion or merely "waiting it out" in the hope that they will go away.

Without a base of universally-accepted practices, your newly-proposed improvements will not survive the inevitable project stress events like turnover, reorganizations, requirements changes, and "crunch mode" operation. The foundation practices must be part of the fabric of the organization so that people no longer question the practices' worth or try to eliminate them in rounds of "cost-cutting."

How To Do It

You should define foundation practices in all of the following areas:

- Requirements management
- Software project planning
- Peer review
- Quality assurance
- Configuration management
- Metrics
- Defect Prevention

Also define practices in any areas that you know to be risk-prone or that have been problematic in the past. These practices need to be tailored for your project and simple enough that they will be universally accepted.

It can be challenging to define a set of consensus practices in some areas. However, you may be able to achieve it with "baby steps." For example, although "Defect Prevention" is an *advanced* practice of the Software Engineering Institute's Software Capability Maturity Model (SW-CMM), you could probably find consensus to hold weekly informal discussion of significant software bugs at the regular staff meeting. Start by discussing any bug that took more than two hours to fix. Talk about how to prevent that bug or similar ones. You will probably see programmers at their desks after the meeting, looking to find similar problems in their code.

Use Risk Analysis To Select New Practices

After you have established (and documented) a baseline of universal agreement, you can look at areas of process improvement based on the project risks you see on the horizon. Then you can prioritize the risks, addressing those that are most significant—rather than those that have the coolest tools.

Continue to analyze and prioritize the risks throughout the project, so that you don't get stuck with a process that no longer addresses the current situation.

Rationale

If you don't choose the right targets for software process improvement, you will waste valuable resources fixing the wrong problems. For example, I saw one project that was cancelled because they did a poor job of prioritizing risks. The team spent months learning object-orientation so that their code would be extensible and maintainable, but their greater risk lay in not being the first product on the market. If you understand the threats to your project, you can choose the right techniques to overcome them.

How To Do It

Apply risk analysis informally by thinking about all of the ways you have seen projects (yours or others) fail in the past, and asking whether any of those factors are present in your current project. Use a risk taxonomy[1] or risk assessment process (e.g. [4]) that you tailor to your project.

Armed with the risk profile, you can evaluate tools and techniques. The highest risks will be expensive to address, so they must be subdivided into manageable units. Here is an example of how it can be done.

Past Situation: A new project member needed weeks of training and code-reading before being able to make even a small change.

General Risk Areas: documentation, design, and training.

Specific Risks: The current design documentation is organized to be more useful as a reference than it is for learning.

Possible Near-Term Practice Improvements: Create "roadmap" and survey documents that don't duplicate content but serve as a guide for learning. Produce examples of typical tasks. Investigate documentation-management tools that support version control.

Copyright © 1999
Cleanroom Software Engineering, Inc.
All Rights Reserved

Success Criteria: Significant decrease in amount of time between joining the team and making useful contribution.

Plan For Diversity

Finally, you should always plan to have more than one process in use at once.

Rationale

You are always going to have process variants unless you adopt a bulletproof process and never change it. In every other case, you will want to be trying out process improvements on a pilot project or component while leaving other parts of the project or organization alone.

Some of the process improvements you select through risk analysis may become part of your foundation process; others will be applied in special circumstances.

How To Do It

One approach is to segment your processes based on the software "ilities" such as capability, reliability, usability, and so on. The project described at the beginning of this paper segmented its processes based on reliability: the flight software had higher reliability requirements so it needed a tougher process. But you could also segment your process based on something like usability: certain projects will have mandatory usability testing, others will not. Or, based on maintainability: certain projects will produce code that is used and maintained over a long time, others produce one-off, short-term solutions.

There are two important attributes to managing diversity, once you have defined the required, recommended, and optional elements of each process and each segment.

Description	Basic	Advanced	Minimal
Use formal 'B' language specifications for every method	Optional	Required	Optional
Conduct 'buddy' review of all code created or changed	Required	Recommended	Optional
Conduct formal inspection of all code created or changed	Optional	Required	Optional
Track all labor hours spent in review/inspection	Recommended	Required	Optional

First, you must clearly define which products must use which processes, and who will decide. If this is not clearly defined in advance, "crunch mode" will reduce every project to the least strict process.

Second, you must have a mechanism through which the work products of one process can be used by another. For example, we don't want code developed as part of an experiment to be used in the flight kernel without some additional reviews and testing. On the other hand, we shouldn't mandate that all experiment code be thrown away either. The right thing to do is to treat experiment code as "raw material" which must be documented, reviewed, and tested within the flight process if it is to be incorporated into the space instrument.

Process diversity adds a little bit of complexity to the search for standards compliance such as CMM and ISO-9000. In general, the company's CMM level will probably be defined by the common subset of all of its processes. If you are seeking CMM compliance, you should evaluate all of your processes against the relevant KPAs. In the case of ISO, each process must be well-defined within the ISO framework with the appropriate documents and records. Each project team member must be aware of which process they are using for each work effort.

Conclusion

For a project to make long-term progress toward improved quality and productivity, it must agree on a set of basic engineering practices. Risk analysis is then used to identify advanced techniques that for trial. The resulting diversity must be carefully managed to prevent chaos and backsliding.

Biography

Michael Deck is a software practice consultant. His company, Cleanroom Software Engineering, Inc., specializes in training project teams to tailor and use software engineering practices to solve real-life software process problems. The company also practices what it teaches, applying Cleanroom practices to contract software-development projects. From 1982 to 1993 he was a member of the IBM Cleanroom Software Technology Center, where he worked closely with the inventors of the Cleanroom approach. He has a BA in Mathematics from Kalamazoo College and an MS in Computer Science from the University of Maryland, College Park. His current research interests include application of Cleanroom to object-oriented development, real-time and embedded software, and highly reliable systems.

Copyright © 1999
Cleanroom Software Engineering, Inc.
All Rights Reserved

References

Note items marked with can be obtained in soft-copy from the Cleanroom web site, www.cleansoft.com/cleansoft.

[1] Marvin J. Carr, Suresh L. Konda, Ira Monarch, F. Carol Ulrich, & Clay F. Walker, "Taxonomy-Based Risk Identification," *Technical Report CMU/SEI-93-TR-6*, June, 1993.

[2] Michael D. Deck, "Data Abstraction in the Box Structures Approach," *Proc. 3rd International Conference on Cleanroom Software Engineering Practices*, October, 1996.*

[3] Michael D. Deck, "Cleanroom Software Engineering: Quality Improvement and Cost Reduction," *Proc. Pacific Northwest Software Quality Conference*, October, 1994.*

[4] Karl Wiegers, *Know Your Enemy: Software Risk Management*, http://www.processimpact.com/articles/risk_mgmt.html

Copyright © 1999
Cleanroom Software Engineering, Inc.
All Rights Reserved

Using The Cost of Quality Approach for Software

Software Quality Week '99
San Jose, CA
May 27, 1999

Herb Krasner
President, Krasner Consulting
hkrasner@cs.utexas.edu

Copyright@1999 - Herb Krasner - all rights reserved

Introductions, Motivation, and the Economics of SW Quality

Myth - Quality software costs more and takes longer

Copyright@1999 - Herb Krasner - all rights reserved

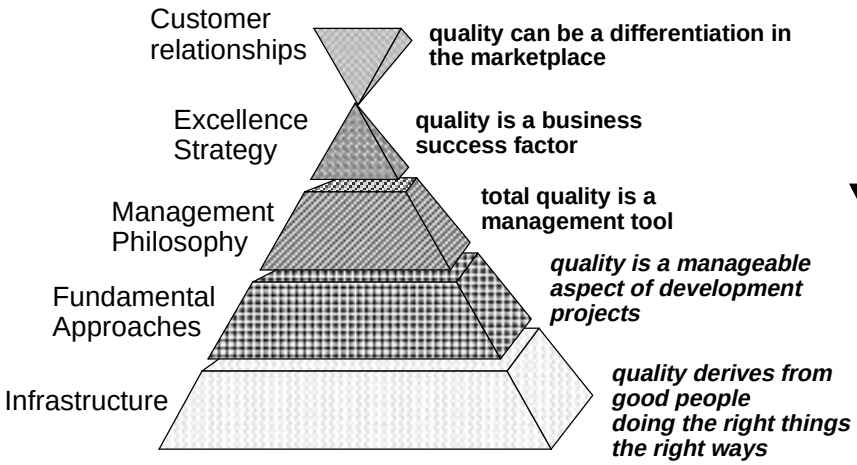
Different Philosophies on Software Quality

- Microsoft - “just good enough” to sell massive quantities to the consumers - shrink-wrapped
- Space Shuttle Flight Control - life critical, must be near zero defect, no failures in flight
- Typical government contractor - meets the requirements specification (telephone book)
- Motorola - 6 Sigma is required to get ahead of our competition
- Others ? Yours ??

Software quality is both a business and technical decision - but it is rarely treated that way

Copyright@1999 - Herb Krasner - all rights reserved

Good SW Quality is Good Business



Copyright@1999 - Herb Krasner - all rights reserved

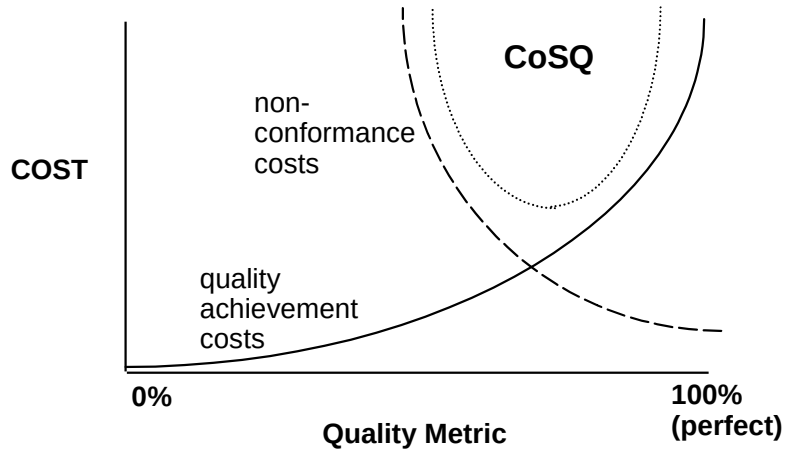
Definitions

Software Quality Costs

- a measure of the costs specifically associated with the achievement or non-achievement (nonconformance) of software product quality - encompassing all requirements established by the company, its customer contracts and society
  - ***Requirements*** - generally includes: wants, needs, constraints, etc. for both the product and related services
  - ***Nonconformance*** - generally, a deviation in one of the work products (e.g. SDP, SQAP, SCMP, SRS, SDD, STP, code, etc.) with respect to understood objectives, requirements, constraints and/or standards.
- Example terms associated with CoSQ
  - negative -> rework, wasted efforts, unacceptable software
  - positive -> preventing defects, validated software
- based on an emerging theory of the economics of software quality, reflecting the differences from manufacturing CoQ
- Non quality costs are the "normal" costs of creating SW

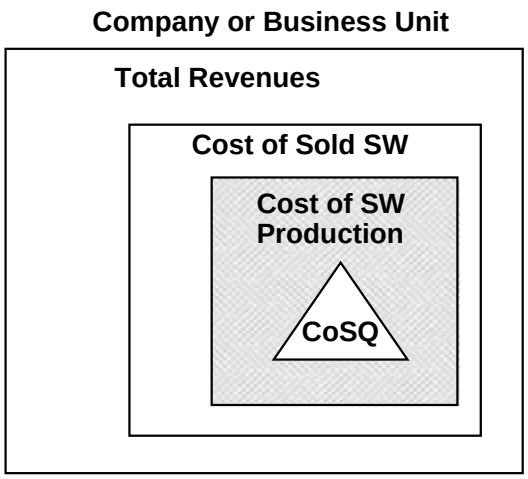
Copyright@1999 - Herb Krasner - all rights reserved

Cost of Software Quality
- Proposed Economic Theory -



Copyright@1999 - Herb Krasner - all rights reserved

Where CoSQ Fits in the Bigger Cost Model



Copyright@1999 - Herb Krasner - all rights reserved

CoSQ Categories

Major Category	Subcategory	Definition	Typical sub-items
Dealing with noncon-formances	Internal	quality problems detected prior to shipment	Pre-release defect mgt. rework, re-reviews, restesting, etc.
	External	quality problems detected after shipment	Post release technical support, problem handling, fixes, etc.
Appraising the level of quality	Discovering the condition	discovering the level of noncon-formances	Testing, SQA, inspections, reviews, etc.
	Assuring the achievement	quality control gating	Product audits, go/no decisions to proceed, release to distribution
Preventing poor quality from occurring	Project/process interventions	TQM, PIP, SPI, org. learning, canceling projects	Training, process improvements, metrics collection and analysis
	Quality basis mgt.	quality defn., goals, stds, thresholds, tradeoff analyses	Defining release criteria for acceptance, quality stds., quality goal model defn

H. Krasner, CrossTalk, November, 1998

Nonconformance - generally meaning a deviation in one of the SW work products with respect to understood objectives, requirements, constraints and/or standards.

Copyright@1999 - Herb Krasner - all rights reserved

Raytheon Electronic Systems SW Improvement Program

RES Cost Model Categories:

- **Nonconformance** - cost of rework
- **Appraisal** - cost of reviews, audits and testing
- **Prevention** - cost of preventing nonconformances
  - included their improvement initiative investment

} **CoQ**

- **Performance** - cost of building it right the first time (i.e. creation)

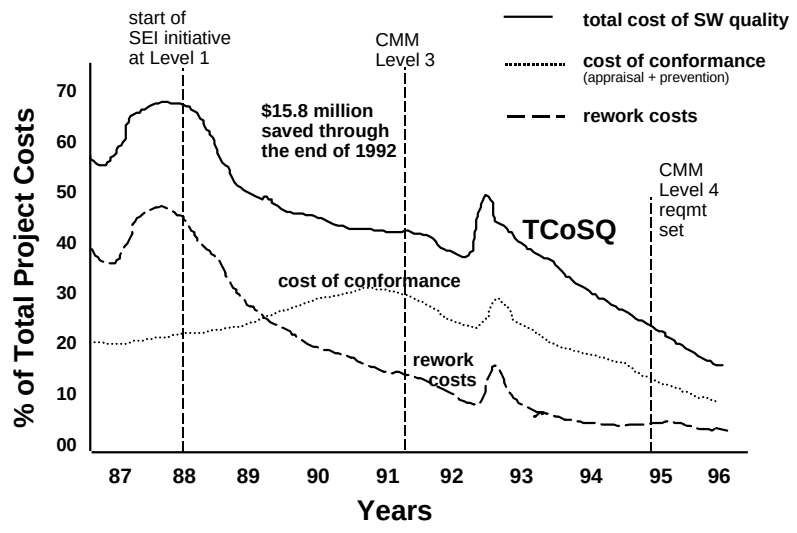
Changes experienced in average % of project time by cost type

	Performance	Nonconformance	Appraisal	Prevention
1988	34%	44%	15%	7%
1990	55%	18%	15%	12%
1992	66%	11%	-	23% -
1994	76%	6%	-	18% -

see Haley, et. al., 1996 for details

Copyright@1999 - Herb Krasner - all rights reserved

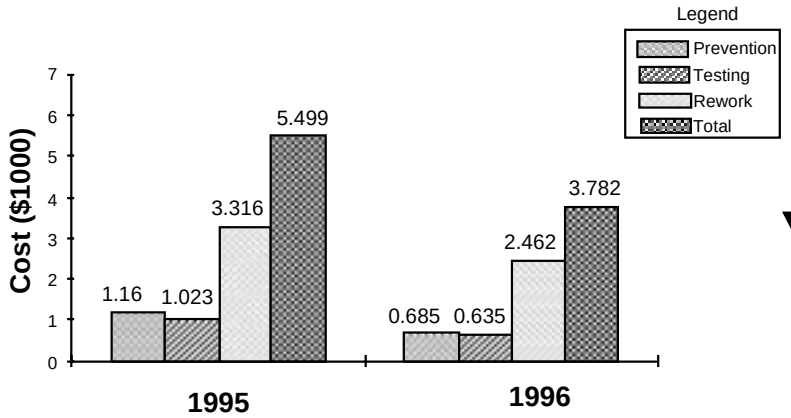
The Cost of Software Quality at RES



Raytheon ESD/RES, see Dion, 1992, 1993, and Haley, et al 1995,1996

Copyright@1999 - Herb Krasner - all rights reserved

Honeywell BSCE Cost of Software Quality



- 80 person org.
- do advanced building control systems
- in Germany

- 30% decrease in CoSQ
- Achieved CMM Level 2
  + implemented design & code reviews
  + earlier testing

from D. Houston, CoSQ:Selling SPI to Managers, 1998, ASQ SW Quality Journal

Copyright@1999 - Herb Krasner - all rights reserved

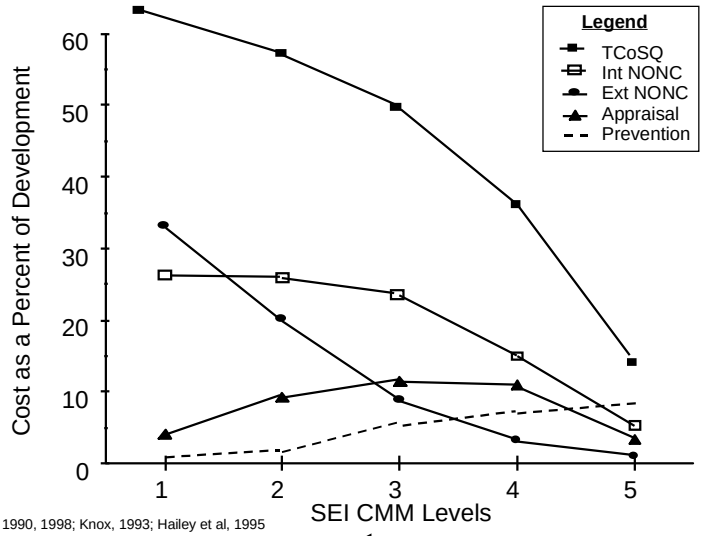
Notional Relationships Observed in Lockheed

Process Maturity (characteristic)	Rework (% of total development effort)	Product Quality (defect density)
Immature	>= .50	double digit
Project process controlled&stable	.25-.50	single digit
Defined org. process	.15-.25	.X
Management by fact	..05-.15	.0X
Continous Learning & improvement	<= .05	< .00X

Krasner, 1990

Copyright@1999 - Herb Krasner - all rights reserved

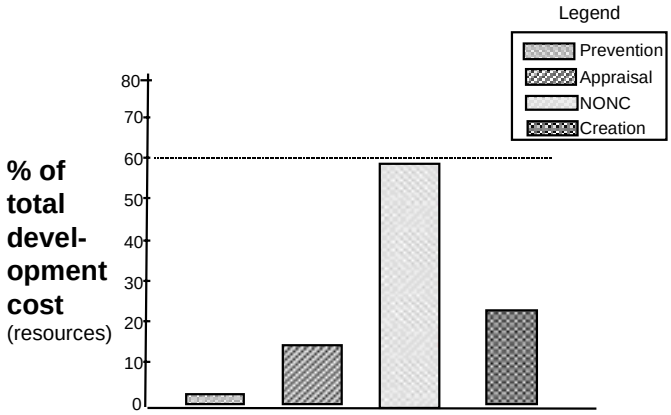
Relating CoSQ and CMM Level



Krasner, 1990, 1998; Knox, 1993; Hailey et al, 1995

Copyright@1999 - Herb Krasner - all rights reserved

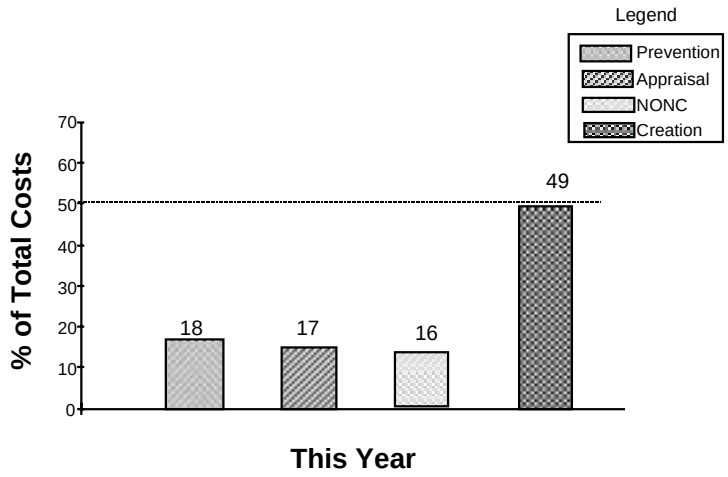
Snapshot of a Level 1 Performer



Current CoSQ Profile

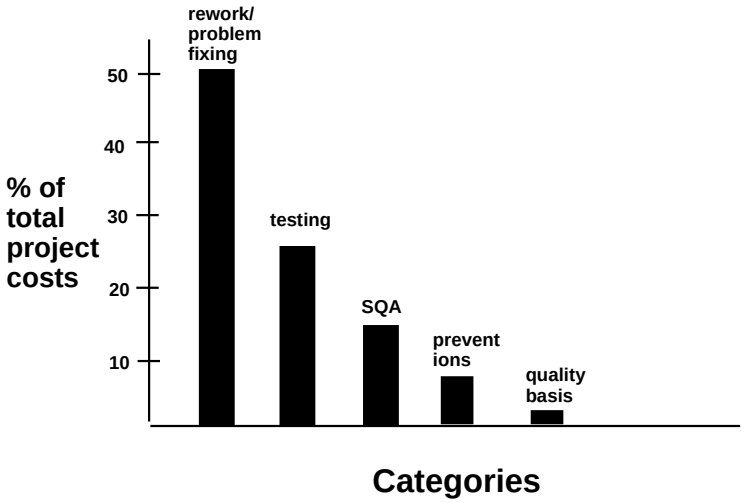
Copyright@1999 - Herb Krasner - all rights reserved

Snapshot of a Level 3 Performer



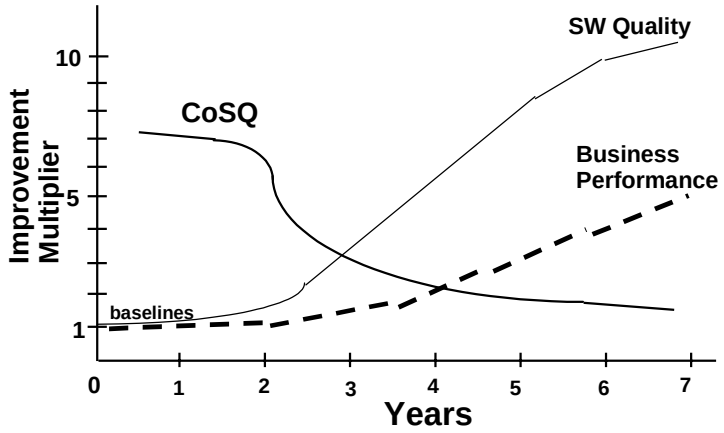
Copyright@1999 - Herb Krasner - all rights reserved

SW Quality Costs - Pareto Analysis



Copyright@1999 - Herb Krasner - all rights reserved

CoSQ Vs Performance Improvement

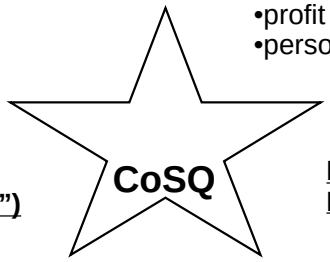


Copyright@1999 - Herb Krasner - all rights reserved

Software Quality Perspectives and CoSQ

Customer Satisfaction
- Conformance to requirements/needs
- Customer and user satisfaction levels
- Fitness for use

Value to Stakeholders
- profit
- personnel satisfaction



Properties("illities")
- Reliability(MTBF)
- Portability
- Maintainability
- Complexity

Development Effectiveness
- Schedule & budget performance
- process quality

Defectiveness
- Defect levels
- Defect severity
- Defect removal

Copyright@1999 - Herb Krasner - all rights reserved

CoSQ Reading References

- Dion, R., Process Improvement and the Corporate Balence Sheet, IEEE Software, July, 1993
- Haley, T., et al, Raytheon Electronic Systems Experience in Software Process Improvement, CMU/SEI-95-TR-017, November, 1995, pp 47-51, also IEEE Software, Nov. 1996
- Houston, D. and Keats, J.B., Research Shows the Cost of Quality Decreases for Maturing Software Organizations, Software Quality Matters, Vol. 5, No. 1-3, 1997 - see http://www.utexas.edu/coe/sqi
- Krasner, H., 1999, *Software Quality Costs* in Principles of Quality Costs, 3rd Edition, J. Campanella, Editor, ASQ Press, 1999
- Krasner, H., 1997; The Cost of SW Quality:Empowering SPI, Keynote Speech, 7th International Conference on Software Quality, see www.utexas.edu/coe/sqi/archive
- Krasner, H., 1992, Continuous Software Process Improvement, in *Total Quality Management for Software*, Van Nostrand Reinholt Co. Inc., NY,NY; ISBN 0-442-00794-9
- Krasner, H., 1997; The Payoff of SPI: What it is and How to Get it, in *The Elements of Software Process Assessment and Improvement*, 1998, IEEE Computer Society Press, 1998, see http://www.utexas.edu/coe/sqi/archive
- Krasner, H. 1998, Using Cost of Quality in Software, CrossTalk, Nov. 1998

Copyright@1999 - Herb Krasner - all rights reserved

USING THE COST OF QUALITY APPROACH FOR SOFTWARE

by Herb Krasner, Krasner Consulting

Abstract - Cost of software quality (CoSQ) is an accounting technique that is useful to enable our understanding of the economic tradeoffs involved in delivering good quality software. Commonly used in manufacturing, its adaptation to software offers the promise of preventing poor quality, but, unfortunately has seen little use to date. This article discusses the rationale and context for using CoSQ, then defines a basic CoSQ approach that differentiates the costs involved with: handling nonconformances due to a lack of quality, activities performed for the achievement of acceptable quality, and efforts to prevent poor quality. CoSQ implementation issues and suggestions are discussed. The proliferation of this approach will help eliminate the wide-spread and debilitating effects of poor software quality.

Software is pervasive in modern society, but we are often unaware of its presence until problems arise. Software is one the most important and yet one of the most economically challenging technologies of the current era. As a purely intellectual product, it is among the most labor-intensive, complex, and error-prone technologies in human history. Even though many successful software products and systems exist in the world today, an overall lack of attention to quality has also led to many problematic systems that don't work right, as well as to many software projects that are late, over budget, or canceled. In short, Software Quality Matters [14].

Although no standard industry definition exists for what constitutes good quality in software, it is generally taken to mean [15] that a software product provides value (satisfaction) to its users, makes a profit, generates few serious complaints, and contributes in some way to the goals of humanity (or at least doesn't do harm). Software quality is difficult to define because there is no single comprehensive and complete standard definition of its lexicon. Various aspects and terms are found in sources such as ISO 9000-3, IEEE Software Engineering Standards, and various books on the subject. The following are the key dimensions of software quality.

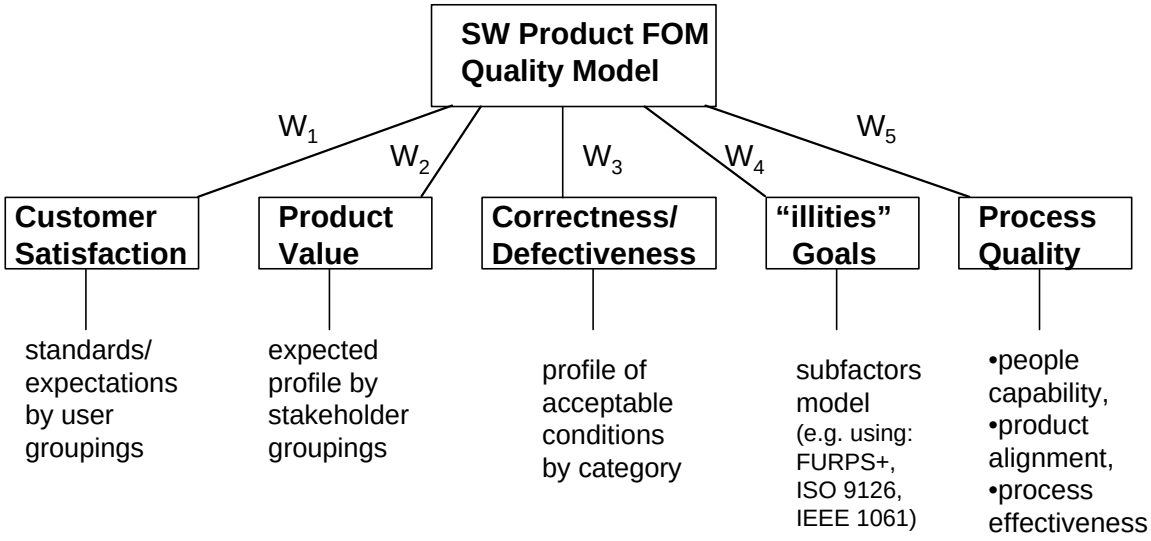
1. **Level of satisfaction**: it is the degree to which a customer or user perceives that a software product meets his or her composite needs and expectations.
2. **Product value**: it is the degree to which a software product has value for its various stakeholders, relative to the competitive environment.
3. **Key attributes** ("ilities"): it is the degree to which a software product possesses a combination of desired properties (e.g. reliability, usability, maintainability).
4. **Defectiveness**: it is the degree to which a software product works incorrectly in target user environments, due to debilitating operational defects.
5. **Process quality**: in relation to the development process by which the product is produced, it means good people doing the right things in an effective way.

A definition fashioned from the above aspects should be created for your own organization and for each project. Every application/business domain faces a specific set of software quality issues, and software quality must be defined accordingly. For example, mission critical applications have very stringent operational needs, whereas typical information system applications must focus on general measures of customer satisfaction. It is also important for each software development project to define it's specific meaning of software quality during the planning phase. Such a definition contributes to the basis for

Copyright 1999@Herb Krasner - all rights reserved - 4/28/99

setting objectives and practical measures of quality progress and determination of readiness for release to customers. An example of such a definition is shown below as a Figure of Merit (FOM) Quality Factors Model.

Figure 1 - Software Quality FOM model example



The categories and subcategories of the FOM model can be weighted as needed for use by domain, system and/or project. The FOM would be the sum of the weighted factors.

Now that we understand what software quality is, and that it is of utmost importance, we turn our attention to the cost perspective of the *economics of software quality*, a subject in serious need of an underlying theory.

Why is CoSQ Important Now?

If it is an organizational goal to improve business success through software quality, then we need the answers to a few simple questions which are too often not even asked in today's software development groups. These questions are:

- how much does poor software quality cost?
- how much does good software quality cost?
- how good is our software quality?

Once the answers to the above questions are factually known then

- quality costs can then be compared to overall software production costs, and software profits
- quality costs can be compared to benchmarks and norms
- deeper analysis can lead to actions taken to improve the situation

Copyright 1999@Herb Krasner - all rights reserved - 4/28/99

- the bottom line effect of quality programs and improvement actions can be measured
- previously hidden costs related to poor quality become visible
- the economic tradeoffs involved with software quality become visible, thus leading to better decision making

Software companies that are concerned about both product quality and economics can successfully apply cost of quality principles to their software developments as shown in the remainder of this paper. CoSQ is the framework that we use to discuss how much good and poor software quality costs.

Adapting CoQ Principles to Software

The principles behind the modern Cost of Quality concept were derived for manufacturing applications and can be found in the works of Juran [16]. In the conventional quality literature, Crosby[3] asserted that "it is always cheaper to do the job right the first time." However, this statement must be reconsidered with respect to software development. Software is, to borrow a metallurgical term, inherently malleable, capable of being readily shaped, formed, and reworked to alter or refine its function, its quality, or even its purpose.

Malleability is an important reason for developing technical solutions in software rather than in hardware. It allows business and technology to adapt to rapid changes in the world, revising objectives and requirements to address new opportunities as they arise. Both customers and producers have come to rely on software's ability to accommodate changing requirements, giving new meaning to "do the job right." A static sense of what is right cannot be presumed during the lifecycle of many software development projects, thus giving rise to non-manufacturing oriented lifecycle models for software (e.g. Spiral, Incremental, Evolutionary, etc.). This additional dynamism strongly influences the economics of the software life cycle and therefore, the application of quality cost principles to software. Establishing and maintaining a baseline definition of what is acceptable quality becomes a key component in the new model for CoSQ.

2. Applying Cost of Quality Principles to Software

The basis for the new model of CoSQ is the accounting of three different types of costs:
1. those which are incurred due to a lack of quality,
2. those which are incurred in the appraisal and achievement of acceptable quality, and
3. those which are incurred in order to prevent poor quality from occurring.

Costs due a lack of quality are further divided into costs of internal nonconformances and costs of external nonconformances. Costs of achieving quality are further divided into appraisal costs and assurance costs. Prevention costs are found both in the development cycle and in organization-wide activities, such as process improvement and metrics collection and analysis, as well as, in quality basis definition and management.

Table 1 provides definitions of the three main CoSQ categories with the next level of breakdown for typical sub-items. The term **nonconformance** means a deviation in one of the software work products with respect to understood objectives, requirements, constraints and/or standards. A more detailed taxonomy of CoSQ categories is available in [2].

Table 1. Costs of Software Quality Model Categories

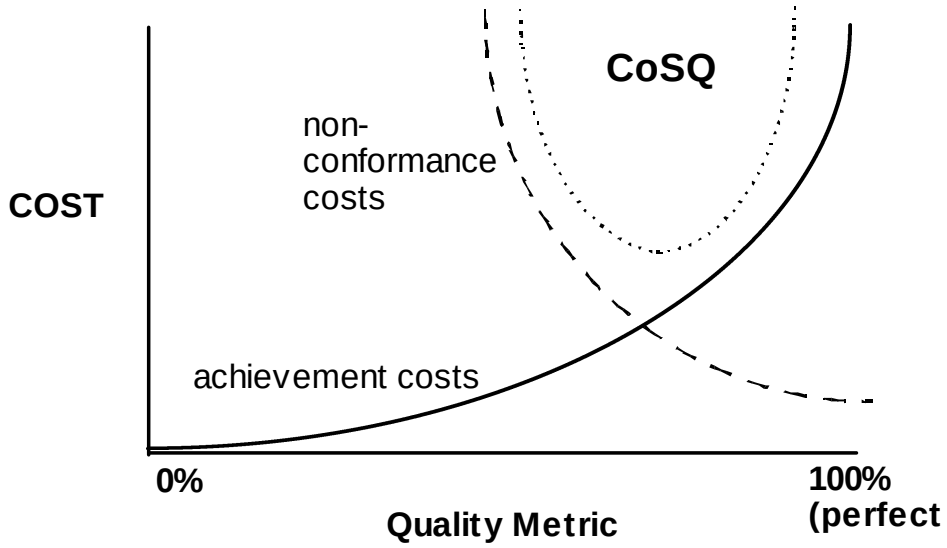
Copyright 1999@Herb Krasner - all rights reserved - 4/28/99

Major Category	Sub-category	Definition	Typical Cost Sub-Items
Dealing with	Internal nonconformances	Quality problems detected prior to product shipment	Pre-release defect management, rework, re-reviews and retesting.
Nonconformances	External nonconformances	Quality problems detected after product shipment	Post release technical support, complaint investigation, defect notification, remedial upgrades and fixes
Appraising the	Discovering the condition of the product	Discovering the level of nonconformances	Testing, SQA, inspections, reviews
level of quality	Assuring the achievement of quality	Quality control gating	Product quality audits, go/no decisions to release or proceed
Preventing poor quality from	Quality basis management	Efforts to define quality, set quality goals, standards and thresholds. Quality tradeoff analysis.	Defining release criteria for acceptance testing, and related quality standards.
occurring	Project and process oriented interventions	Efforts to prevent poor product quality, and/or improve process quality	Training, process improvements, metrics collection and analysis

An Economic Model of Software Quality Costs

There is no validated economic theory of software quality in existence today. This is clearly a ripe subject for multidisciplinary research. The software community currently uses a cost of quality theory borrowed from manufacturing, which is exhibited in the figure below.

Figure 2 - Economic Model of Software Quality Costs



Copyright 1999@Herb Krasner - all rights reserved - 4/28/99

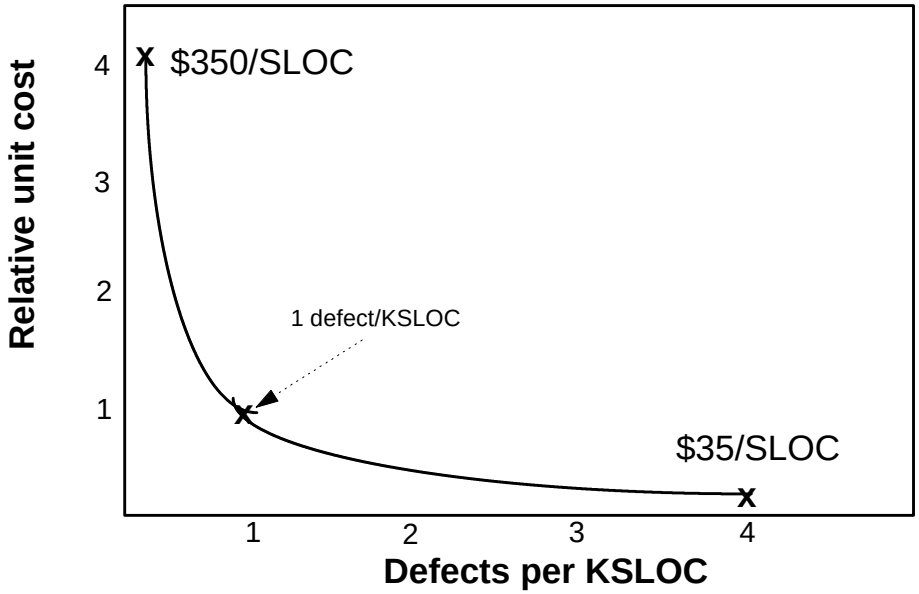
Figure 2, for instance, represents what Juran's Quality Control Handbook [Gryner 1988] calls a model for "traditional processes" in contrast to a different set of curves - with no point of diminishing returns - for what was seen as "emerging processes." These latter processes take advantage of higher priorities on prevention, the automation of appraisal activities, and new technology to reduce inherent failure rates. It remains an open question whether software development follows this "emerging process" model: the ability to achieve as near perfection as required at finite costs.

Once quality is defined, the costs of achieving quality (costs of conformance) and the costs due to lack of quality (costs of nonconformance) have an inverse relationship to one another: as the investment in achieving quality increases, the costs due to lack of quality decrease. This relationship is seen in Figure 2. The quality metric for software is usually a defectiveness level, such as number of defects per system (or part). In traditional CoQ models, the TCoQ has a point of diminishing returns, a minimum prior to achieving 100% of the quality measure. Current research is investigating whether or not the law of diminishing returns applies to the CoSQ in all cases.

As an industry we have collected very little data about cost of software quality which could be used to either challenge or validate this theory. The little we do have suggests that this economic model may not account properly for the dynamics of software development, since perfection is either not a goal or is a quickly moving target.

The NASA Space Shuttle software program collected and reported on data in this area. In this case failure free software is the goal for much of the software system that flies the space shuttle. As shown in the figure below, Keller and Rhone (1990) [10], were able to show the increasing cost of achieving extremely high reliability in the mission critical parts of the flight control software of the shuttle. After studying the defect data for mission critical software, it is apparent that the cost increases dramatically below a certain level of defect density, under which it becomes very expensive to test, find and fix problems.

Figure 3 - The Cost of Ultra High Reliability in the Shuttle Software



CoSQ: Data Found in the Open Literature

Copyright 1999@Herb Krasner - all rights reserved - 4/28/99

While the costs of software quality assurance and process improvement have been a topic of concern for over 20 years (Alberts 1976), very limited data has been available in the open literature that discusses the cost of software quality (CoSQ). The main sources to date are a Price Waterhouse study [12], Krasner's report [8], Knoxís article [7] and the Haley report on Raytheon efforts [6], which all discuss trends in software rework costs.

A 1988 Price Waterhouse study analyzed the costs and benefits of software quality standards from a survey of 19 United Kingdom (UK) software suppliers. The study estimated the cost of a quality control effort (prevention and appraisal costs) to be 23% to 34% of development effort. The study also estimated failure costs at 15% of development effort for a TCoSQ of 38% to 49% of development effort. It must be noted that this study excluded the costs of unit testing and rework because the suppliers could not separate these costs. With increases in the estimates to account for this oversight, TCoSQ in a software organization with a quality system can range from 40% to 55% of development costs with a conformance costs to nonconformance costs ratio from 1.5 to 2.

Another recently published case study [19] has reported on software quality initiatives that reduced the total quality cost to 50% of its pre-initiative value. It indicated that there were greater marginal cost reductions at the (initial) higher defect densities, but that all investments in quality improvement could be cost-justified. They also used CoSQ data to compare the value of four process improvements at BDM International.

Dion [5] used the CoQ model as one means of interpreting the results of improvement initiatives undertaken at Raytheon Electronic Systems (RES). Recently Haley, et al [6] updated this study. Using the results of tracking 15 projects, they recorded significant results in a little over three years. In the Level 1 stage, RESís CoSQ fluctuated between 55 and 67% of total development costs and when reaching Level 3 process maturity, their CoSQ had dropped to approximately 40% of total project cost. The ratio of conformance to nonconformance costs was 1.5. By 1996 this organizationís TCoSQ was approximately 15% of development costs, and the rework due to both internal and external nonconformances has been reduced to less than 10% of development costs. The case study of RES is presented in detail in [2].

Based on the results of my study of Lockheed projects at various CMM levels along with anecdotal and quantitative data collected in the mid and late 80's, I predicted the relationship of SEI CMM-based process maturity level to typical rework rates and quality levels that could be expected [8]. The table below is a slice of that presentation.

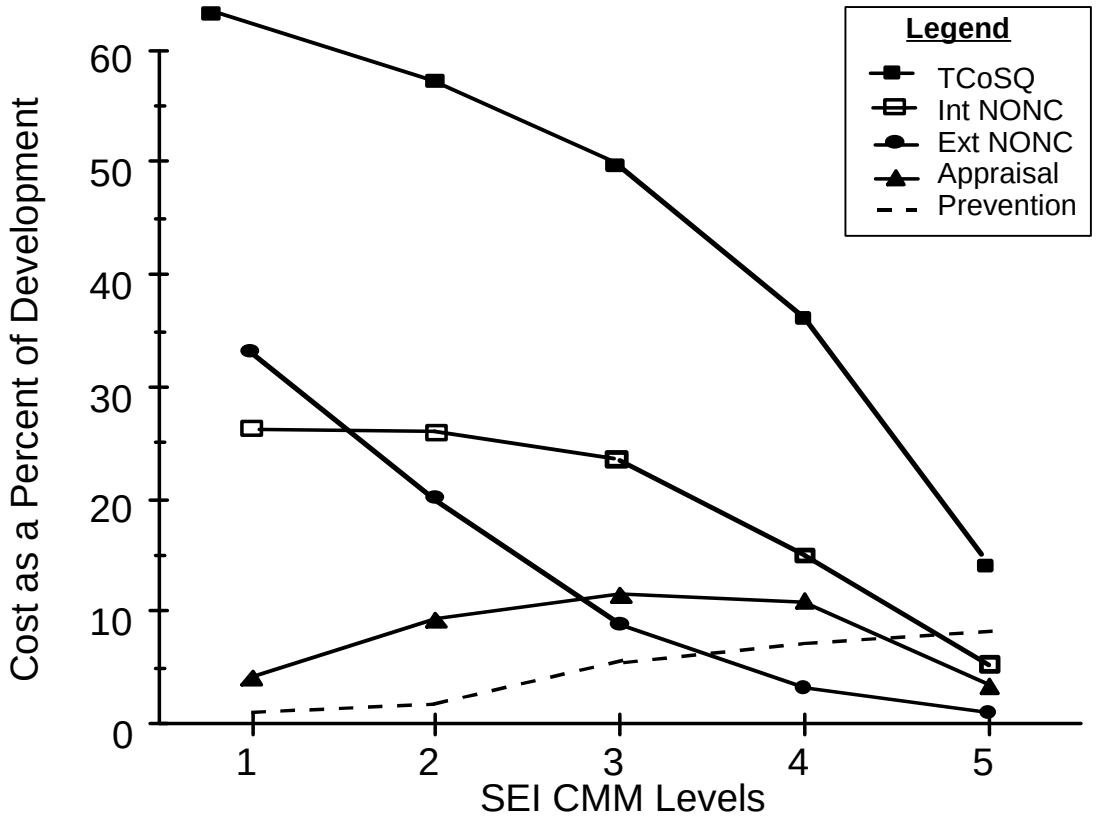
Table 2 - Process Maturity, Quality and Rework Results

Copyright 1999@Herb Krasner - all rights reserved - 4/28/99

Process Maturity (characteristic)	**Rework** (% of total development effort)	**Product Quality** (defect density)
Immature	**>= .50**	double digit
Project controlled	**.25-.50**	single digit
Defined org. process	**.15-.25**	.X
Management by fact	**..05-.15**	.0X
Continous Learning & improvement	**<= .05**	< .00X

Due to the SEI CMM process maturity movement, we have an aggregation of the payoff data that has been collected as a result. See [9] for more information on how software process maturity is related to CoSQ, software defectiveness levels and other measures of success. My ongoing collection of such data from my clients, along with the early efforts of Knox [7] have led to the model presented below. The following model makes predictions about CoSQ across the levels of the Software Engineering Instituteís Capability Maturity Model (Figure 3).

Figure 4 - Cost of Software Quality and CMM Level



Starting with the total CoSQ (TCoSQ) at 70% of development costs (based on several industry figures) for CMM level 1 organizations, the model hypothesizes that CMM level 5 organizations can cut this

Copyright 1999@Herb Krasner - all rights reserved - 4/28/99

CoSQ by more than 75%. The model predicts that a CMM level 3 organization would have a TCoSQ of 50%, but with a conformance to nonconformance cost ratio of .5. It appears that the model may be a fair predictor of TCoSQ for maturing software organizations.

Typical manufacturing CoQ, ranging from 5 to 25 percent of company sales, contrasts significantly with CoSQ. With the present state of software engineering practice we can expect CoSQ to range from 10 to 70 percent of development costs. Even accounting for the margin between production costs and sales, CoSQ appears to be roughly twice manufacturing CoQ. Also, the optimum manufacturing CoQ is often in the range of 95 to 100 percent of conformance to quality standards. The open literature lacks sufficient data for CoSQ as a function of conformance to quality, but the above data suggests that software producers have yet to reach such an optimum.

3.0 Elements of a CoSQ Program

3.1 CoSQ Program Purposes

There are many possible ways to apply the CoSQ approach. To date, CoSQ techniques are only being used after the fact to document the ROI for software improvement initiatives because executives want to know that there is a payoff from the up-front investments. This type of CoSQ application is expected to accelerate as more process improvement programs take off.

Other ways in which the CoSQ approach can be used are to

- provide a basis for budgeting the quality management and assurance functions
- identify specific quality improvement candidates through causal analysis
- compare proposed process improvements and identify the most cost effective ones
- provide a (one) measure to compare the success of various projects
- reduce the quality costs on a particular project by altering the process prior to, or even in situ
- determine the potential cost/risk impact of specific quality tradeoff decisions on specific projects
- determine a company's potential legal exposure due to customer experienced defects
- provide cost data to demonstrate the relationship of employee efforts to the bottom line

3.2 CoSQ Programmatics

Several points can be made with regard to the programmatic aspects of measuring and using cost of quality information specifically for software development organizations. These are:

- initiating a CoSQ effort
- accounting and gathering the quality cost data
- gathering the quality metrics
- presenting the results
- improving the CoSQ program continuously

Initiating the CoSQ Effort

Convincing management of the value of tracking CoSQ may be the initial hurdle one encounters in using this technique. There is a modest up-front investment required in order to educate those to be involved.

Initially, rough estimates of software quality costs may suffice very well for several reasons.

Copyright 1999@Herb Krasner - all rights reserved - 4/28/99

- Usually the largest CoSQ costs can be estimated readily from time and activity reports, so the expense of data gathering can be limited until its value is demonstrated.
- Controlled, scientific studies are unlikely and incomplete data can suffice in beginning a software cost benefit analysis.
- The published data indicates that the quality cost difference between improved and unimproved organizations is quite large.
- The primary purpose of the initial CoSQ effort will be to show the opportunity for cost savings.

The best advice is to keep it simple when starting a CoSQ initiative.

Accounting

Gathering quality cost data assumes that costs have been accounted using task categories which can be summed into the four major categories of quality costs. Many software organizations track costs in a manner amenable to quality costing, but many others do not. In the latter case, a preliminary step of defining and installing such a chart of accounts is required. A sample of such a chart of software quality costs can be found in [3]. The quality categories in a software organizationís chart of accounts must be tailored to reflect its software process. To realize the full benefit of CoSQ, it must also allow for the addition of continuous improvement tasks.

In the best cases, quality costs can be taken directly from departmental accounting (salary and expense) reports. In other cases, it may be necessary to resort to basic accounting and engineering records, such as schedules, time reports, defect reports, and purchasing records. In the worst cases, one may fall back on interviews with members of the software organization in order to construct estimates of each quality cost category. Exceptions are in the external failure category.

One of the pitfalls of a CoSQ program is ìcontroversial cost categoriesî. Usually the question is about which costs are normal operating costs and which are quality costs. An example would be the cost of producing a project management plan. While this plan is produced for the sake of managing a projectís expenses and schedule, it also influences product and process quality. Here it is helpful to keep in mind the following points.

- The trend among quality specialists has been to view quality costs as those incurred to directly prevent, appraise, and address the nonconformances of *poor quality*.
- Arguments over controversial categories have been known to sabotage cost of quality programs.
- The largest quality costs are those which are most easily discerned, for example reviews, software quality assurance (SQA), testing, and rework. Therefore, it is often safe to exclude controversial categories without unduly affecting the TCoSQ.
- Consistency throughout a CoSQ program is more important than thorough inclusion of quality costs because consistency allows for clear identification of improvements and candidates for improvement.

Concerns may also arise as to how quality costs should be categorized. Again, consistency is important. For example, the costs associated with formal inspections (peer reviews) can be treated as prevention costs rather than as appraisal costs. This is a matter of interpretation, depending on when a work product is considered ready for appraisal. Although manufacturing inspections are conducted on pieces after they are produced, in software production inspections may be incorporated into the production process. For documentation, this means that a document is not complete until it has undergone a peer review and

Copyright 1999@Herb Krasner - all rights reserved - 4/28/99

been revised. The same is true for code, especially when code inspections precede unit testing, clearly an appraisal activity.

Quality Metrics Collection

With regard to measures of quality, the CoQ has been used primarily with a fundamental approach to quality, that is defect rates (manufacturing) or service problem reports (service industries) rather than broader approaches that would take into account factors such as usability, testability, maintainability, and so forth. The fundamental approach has the advantages of straightforward measurement and ease of understanding. It also allows comparison of dissimilar products. Furthermore, if failure costs are collected in a defect tracking system, the most expensive defects can be identified for root cause analysis (Mandeville 1990). This discussion recognizes that most software producers take a fundamental approach to quality, concentrating on defect measurement, prevention, and removal.

Defect density is a good metric to start with measuring CoSQ improvements. Specifically, CoSQ can be plotted against defect density at the completion of system testing. This metric may be obtained from defect reports during alpha and beta tests and for a time period, say six months, following product release. Better yet, it may be generated statistically based on post-release defect reports for previous products from the same organization. Stoddard and Hedstrom [17] offer a recent example of this approach using Bayesian statistics in a defect prediction model. External failure costs can be estimated from the defects-at-release metric.

Presenting CoSQ Information

The relationships which have the greatest impact on management are:

- Quality costs as a percent of sales and profit
- Quality costs as a percent of total development costs
- Quality costs compared to the magnitude of the current problem

Figure 5 below shows an example of a snapshot report of an organization's CoSQ profile as a percentage of total development costs at that time. The example comes from a CMM Level 1 shop.

Figure 5 - CoSQ snapshot profile

Copyright 1999@Herb Krasner - all rights reserved - 4/28/99

- how the CoSQ model is defined at the detailed levels
- how the approach is implemented consistently in the organization
- how CoSQ is used for root cause analysis
- how CoSQ is used to stimulate improvements

These difficulties can be overcome with appropriate training and coaching.

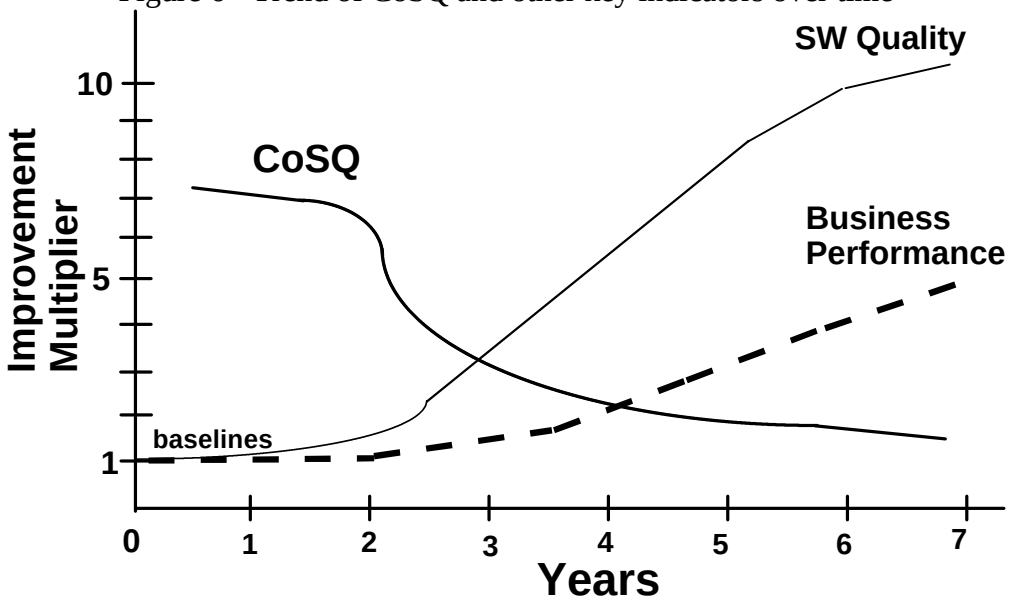
Feedback on the usefulness of the CoSQ data presented can guide how the organizational CoSQ program should be evolved over time. The lessons learned from trials and early adopters will be invaluable. The technology to support CoSQ will emerge quickly in response to the needs as they grow, once a

consensus on the CoSQ model is reached. Many of the TQM and CoQ tools available for manufacturing can be adapted for use in software. CoSQ tools appear to be a significant market opportunity yet to be explored.

4.0 Summary

CoQ is a proven technique in manufacturing industries both for communicating the value of quality initiatives and for indicating quality initiative candidates. CoSQ offers the same promise for the software industry, but has seen little use to date. Initial uses of CoSQ show that it can be a very large percentage of development costs, 60 percent or higher for organizations which are unaware of improvement opportunities. CoSQ has demonstrated its value in measuring the ROI of a software improvement program, as in the RES case. If your improvement program is effective, and you track CoSQ over time against the key performance indicators of your organization you should observe a trend similar to that shown in Figure 6.

Figure 6 - Trend of CoSQ and other key indicators over time



CoSQ is a technique that is most useful in enabling our understanding of the economic tradeoffs involved in delivering good quality software. Applying CoSQ in your organization requires a small investment that pays off handsomely in your increased understanding of the complexities and hidden issues involved in the delivery of quality software. The proliferation of CoSQ will help eliminate the debilitating effects of poor software quality.

5.0 References

1. Alberts, D.S. 1976. The Economics of Software Quality Assurance. *National Computer Conference 1976*, 433-441.
2. Campanella, J., ed. 1999. *Principles of Quality Costs*, 3rd edition. Milwaukee, WI: American Society for Quality Control, ISBN 0-87389-443-X
3. Crosby, P.B. 1988. *Quality Without Tears*. New York: McGraw-Hill.
4. Daugherty, T. 1988. The Search for Software Quality. *Quality Progress*. (November): 29-31.
5. Dion, R. 1993. Process Improvement and the Corporate Balance Sheet. *IEEE Software*. 10(July): 28-35.

Copyright 1999@Herb Krasner - all rights reserved - 4/28/99

6. Haley, T.J. 1996. Software Process Improvement at Raytheon, *IEEE Software*, 13(November): 33-41.
7. Knox, S.T. 1993. Modeling the Cost of Software Quality. *Digital Technical Journal* 5 (4): 9-16.
8. Krasner, H., 1990, Self Assessment Experiences at Lockheed, Proceedings of the SEI/AIAA Software Process Improvement Workshop, November 8, 1990, Chantilly, VA
9. Krasner, H.,1997, Accumulating the Body of Evidence for the Payoff of Software Process Improvement(1997 version), on the web at www.utexas.edu/coe/sqi/archive, also published in The Payoff for Software Process Improvement: What it is and How to get it, in *The Elements of Software Process Assessment and Improvement*, IEEE Computer Society Press, 1999
10. Krasner, H. (1994), *A Case History of the NASA Space Shuttle Onboard Systems Project*, SEMATECH Technology Transfer Report #94092551A-TR, October 31, 1994
11. Mandeville, W.A. 1990. Software Costs of Quality. *IEEE Journal on Selected Areas in Communications* 8 (2): 315-318.
12. Price Waterhouse. 1988. *Software Quality Standards: The Costs and Benefits. A review for the Department of Trade and Industry*. London: Price Waterhouse Management Consultants.
13. Putnam, L.H. 1994. The Economic Value of Moving up the SEI Scale. *Managing System Development* 14 (7): 1-6.
14. Software Quality Matters, 1992-present, the quarterly newsletter on subjects of software quality published by the University of Texas - Software Quality Institute, see back issues on the web at www.utexas.edu/coe/sqi
15. Davis, A. 1997, IEEE Software, Editors Column, December, 1997
16. Juran, J.M., and Frank M. Gryna. *Juran's Quality Control Handbook*, 4th ed., New York: McGraw-Hill Book Company, 1988.
17. Stoddard, Robert, and John Hedstrom. 1995. "A Bayesian Approach to Deriving Parameter Values for a Software Defect Predictive Model," *Proceedings of the Sixth Annual Conference on Applications of Software Measurement*, Oct 30 - Nov 2, 1995, 323-346
18. Houston, D. 1998, Cost of Software Quality: Selling Software Process Improvement to Managers, in ASQ's Software Quality Journal, 1998
19. Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan, "Evaluating the Cost of Software Quality," Communications of the ACM 41:8(August 1998) 67-73.

Copyright 1999@Herb Krasner - all rights reserved - 4/28/99

Version 3.1 March 20 1999

For Quality Week San Jose CA, Thursday 27th May 1999 at 10:30 am

"Risk Management : A practical toolkit for identifying, analyzing and coping with project risks"

By Tom Gilb,
Senior Partner, Result Planning Limited

Paper Summary

Risk management must be fully integrated into all the development and maintenance processes for systems. It involves more than applying risk assessment methods to identify and evaluate system risks.

To explain this broad approach to risk management, this paper discusses the way in which Requirements Driven Management (RDM) methods contribute to handling risks.

Definition of 'Risk'

Risk is an abstract concept expressing the possibility of unwanted outcomes.
A 'risk' is anything which can lead to results that deviate from the requirements.

It is in the nature of risk that the probability of risks actually occurring, and their actual impact when they do so, can only be predicted to varying degrees of accuracy. Not all risks can be identified in advance.

Risk Management is any activity which identifies risks, and takes action to remove, reduce or control 'negative results' (deviations from the requirements).

Principles of Risk Management

In my view, the fundamental principles of risk management include:

1. Quantify requirements

All critical quality and resource requirements must be identified and quantified numerically.

2. Maximize profit, not minimize risk

Focus on achieving the maximum benefits within budget and time-scales rather than on attempting to eliminate all risk.

3. Design out unacceptable risk

Unacceptable risk needs to be ‘designed out’ of the system consciously at all stages, at all levels in all areas, e.g. architecture, purchasing, contracting, development, maintenance and human factors.

4. Design in redundancy

When planning and implementing projects, conscious backup redundancy for outmaneuvering risks is a necessary cost.

5. Monitor reality

Early, frequent and measurable feedback from reality must be planned into your development and maintenance processes, to identify and assess risks before they become dangerous.

6. Reduce risk exposure

The total level of risk exposure at any one time should be consciously reduced to between 2% and 5% of total budget.

7. Communicate about risk

There must be no unexpected surprises. If people have followed guidelines and are open about what work they have done, then others have the opportunity to comment constructively. Where there are risks, then share the information.

8. Reuse what you learn about risk

Standards, rules and guidance must capture and assist good practice. Continuous process improvement is also needed.

9. Delegate personal responsibility for risk

People must be give personal responsibility in their sector for identification and mitigation of risks.

10. Contract out risk

Make vendors contractually responsible for risks, they will give you better advice and services as a result.

Let's now consider, each of these principles in turn and describe some (not all!) of the roles that the RDM methods play in risk management. However, first here is an outline sketch of the RDM methods:

- **Planguage**; a requirements specification language insisting on quantified values.

- **Impact Estimation** (IE); an analysis tool (a table) allowing evaluation of the likelihood of achieving requirements and, the evaluation and comparison of different designs (strategies). A strength of IE is that it also helps identify new designs and uncover previously unstated requirements.

- **Evolutionary Delivery** (Evo); based on the work by the quality gurus Deming and Juran, a way of working that focuses on evolutionary delivery of early, measurable, system benefits to the customers. A system is developed, by small risk steps, in a series of plan, develop, deliver and evaluate cycles.

- **Inspection**; a technique for measuring and improving technical document quality. Technical documents are evaluated against their source documents and any prevailing standards by Inspection teams consisting of individuals with specially assigned roles. The overall aims are to identify defects, to identify patterns in the introduction of defects (leading to process improvement), to help train individuals to avoid creating defects and, to assist team-building.

Readers wanting a more detailed explanation of these methods should look in the References.

Principle 1. Quantify requirements

All critical quality and resource requirements must be identified and quantified numerically.

Risk is negative deviation from requirements. So, if we are going to understand risk, we must have some way of specifying exactly what we want. If we use vague ways like "State of the Art, World Class, Competitor-Beating Levels of Quality", we cannot understand and assess risk.

Planguage helps because it demands numerically quantified requirements. Using Planguage, we must go through the following steps:

- Identify *all critical* quality and resource *attributes* of the system. In practice, this could be ten or more critical qualities (e.g. availability) and, five or more critical resources (e.g. operational costs).

- Define exactly how to *understand variation* in each attribute by specifying a scale of measure, e.g. 'Scale: Probability of being fully operational during the office day' and 'Scale: Total of all monetary operational expenses including long term decommissioning costs'.

- For each attribute, define one or more critical points on the defined scale of measure which are needed for the system to function properly and profitably. There are two important categories: 'Must' and 'Plan'. A 'Must' level defines the system survival level. A 'Plan' level defines the planned point for success. For risk management, 'Must' is the first level and 'Plan' is the second level for risk determination. A value for any attribute less than its required Must level means total system failure. Only when all Plan levels for all the attributes have been met can a system be declared a success.

- For all the Must and Plan levels, define additional qualifying information. We call this using 'qualifiers'. You are basically defining time, place and event, i.e. when it is critical for you to achieve a certain level of an attribute, where it is critical and under what conditions. For example,

Plan [1999,Europe,IF European Monetary Union implemented anywhere] 99.98%

We can even give direct expression to the amount of risk we are prepared to take by a statement such as :

Must [2001, UK, IF Euro is used in Norway & UK] 60% ±20%

In other words the range of results 40% to 80% is an acceptable upper and lower limit, but below 40% is unacceptable.

Here is a more complete example:

Usability: Scale: Mean time to learn [defined tasks] to minimum proficiency. Must [Release 2.0, English Version, Task: Modifying Files] 10 minutes. Plan [Release 2.0, English Version, Task: Modifying Files] 7 minutes. Plan [Release 3.0, English Version, Task: Modifying Files] 5 minutes. Plan [Release 3.0, French & Dutch Versions, Task: Finding a File by Content] 5 minutes.

In the example, the most critical (failure of system) risk is the Must level. The other statements are only of secondary risk; they indicate the levels required to declare success.

It should be obvious that the degree of risk can be expressed in terms of the deviation from the target levels. For example,

Method A can sometimes result in a learning time of 10 minutes, while method B can never result in a learning time exceeding 4 minutes.

This means that for the specified requirements, method A poses a real risk, but method B does not.

A template specification of risk levels

In addition to the basic statements described above, it should be noted that there are a wide variety of ways within Planguage to indicate that the information contains some element of risk. Here are some examples:

Plan 60-80	Specification of a range
Plan 60±30	Specification of an upper and lower limit
Plan 60 → 90	
Plan 60?	Expressing that the value is in doubt
Plan 60??	Expressing that the value is in serious doubt
Plan 60 ← A wild guess	Using the source of the information to show the doubt
Plan 60 ← A.N. Other	Depends on A.N. Other's credibility in setting this value
Plan <60>	Fuzzy brackets indicate data needing improvement

All of the above signals can be used to warn of potential risk. Of course, the culture must encourage such specification rather than intimidate people from using it.

Plan [IF Euro is used in UK] 99%

The above is an example where the risk is controlled by making the specification totally dependent on the IF condition. There is no risk that anyone will plan to achieve 99% if the condition is false. However, they are warned to plan to achieve 99% should the condition turn true.

Note, you can also use IF qualifiers to constrain the use of a strategy (a means for achieving a goal). This reduces the risk that an expensive strategy is applied under inappropriate conditions.

Strategy99 [IF hunger famine in a country, IF road and rail transport unavailable] Aerial Supply of Food.

Principle 2. Maximize profit, not minimize risk

Focus on achieving the maximum benefits within budget and time-scales rather than on attempting to eliminate all risk.

Elimination of all risk is not practical, not necessary and, not even desirable.
All risk has to be controlled and balanced against the potential benefits. In some cases, it is appropriate to decide to use (and manage) a strategy with higher benefits and higher risks. I use Impact Estimation (IE) to help me assess the set of strategies I need to ensure I meet the required objectives. My focus is always on achieving the objectives *in spite of* the risks.

Outline Description of Impact Estimation (IE)

The basic IE idea is simple: estimate quantitatively how much your design ideas impact all critical requirements. This is achieved by completing an IE table. The left-hand column of the table should contain the objectives and, across the top of the table should be the proposed strategies. For the objectives, assuming you have expressed them using Planguage, it is a question of listing down all the quality and resource attributes you wish to consider. You need next to decide on a future date you want to use. This should be a system ‘milestone’; a date for which you have specified Must and Plan levels. Then, against each attribute, you state the current level and the Plan level for your chosen date. (If you are especially risk averse you would use the Must level!) For the strategies, you simply list them across the top of the IE table.

You then fill in the table, for each cell you answer the question, ‘How does this strategy move the attribute from its current level towards the Plan level?’ First you state the actual value you would expect and then you convert this into a percentage of the amount of required change.

For example, Training Time for Task A is currently 15 minutes and you require it to be 10 minutes within six months. You estimate Strategy B will reduce Training Time for Task A to 12 minutes. In other words, Strategy B will get you 60% of the way to meeting your objective. See Table 1.

TABLE 1

	Strategy B	
	Real Impact	% Impact
Training Time Past = 15 minutes in June 1998 Plan = 10 minutes by end of Dec. 1998	12 minutes	60%
Resource = Development Budget Plan = $2000 up to end Dec. 1998	$1,000	50%

Further improvements to specifying the impacts

There are a number of improvements to this basic idea, which make it more communicative and credible. Here is a brief summary of them :

Uncertainty of Impact: you can specify a range of values rather than a single value.

Evidence for Impact Assertion: you can state the basis for making your estimate. For example: "Strategy B was used for 5 projects last year in our company, and the percentage improvement for Training Times was always 60% to 80%".

Source of Evidence for Impact Assertion: Of course, some skeptic might like to check your assertion and evidence out, so you should give them a source reference, e.g. "Company Research Report ABR-017, pages 23-24."

Credibility Rating of the Impact Assertion: We have found it very useful to establish a numeric 'credibility' for an estimate, based on the credibility of the evidence and the source. We use a scale of 0.0 to 1.0 (because it can then be used later to modify estimates in a conservative direction). See Table 2.

TABLE 2

Credibility Rating	Meaning
0.0	wild guess, no credibility
0.1	we know it has been done somewhere
0.2	we have one measurement somewhere
0.3	there are several measurements in the estimated range
0.4	the measurements are relevant to our case
0.5	the method of measurement is considered reliable
0.6	we have used the method in-house
0.7	we have reliable measurements in-house
0.8	reliable in-house measurements correlate to independent external measurements
0.9	we have used the idea on this project and measured it
1.0	perfect credibility, we have rock solid, contract-guaranteed, long-term, credible experience with this idea on this project and, the results are unlikely to disappear

Further Analysis of the IE data

Once you have completed filling in all the impacts, there are a number of calculations, using the percentage impact estimates (%Impact), that help you understand the risks involved with your proposed solution.

Let me stress that these are only rough, practical calculations. Adding impacts of different independent estimates for different strategies, which are part of the same overall architecture, is dubious in terms of accuracy. But, as long as this is understood, you will find them very powerful when considering such matters as whether a specific quality goal is likely to be met or which is the most effective strategy. The insights gained are frequently of use in generating new strategies.

Impact on a Quality: For each individual quality or resource attribute, sum all the percentage impacts for the different strategies. This gives us an understanding of whether we are likely to make the planned level for each quality or cost. Very small quality impact sums like '4%' indicate high risk that the architecture is probably not capable of meeting the goals. Large numbers like 400% indicate that we might have enough design, or even a 'safety margin'.

TABLE 3

Example: Adding the percentage impacts for a set of strategies on a single quality or cost can give some impression of how the strategies are contributing overall to the objectives. Note Strategies A, B and C are independent and complementary.

	Strategy A	Strategy B	Strategy C	Sum of Strategy Impacts	Sum Uncertainty
Reliability 900->1000 hours MTBF	0+/-10%	10+/-20%	50+/-40%	60%	+/-70%

Impact of a Strategy: For each individual strategy, sum all the percentage impacts it achieves across all the qualities to get an estimate of its overall effectiveness in delivering the qualities. The resulting estimates can be used to help select amongst the strategies. It is a case of selecting the strategy with the highest estimate value and the best fit across all the critical quality requirements. If the design ideas are complementary then the aim is to choose which strategies to implement first. If the strategies are alternatives, then you are simply looking to determine which one to pick.

TABLE 4

A measure of the effectiveness of strategy ‘Big Idea’ can be found by adding together its percentage impacts across all the qualities

QUALITY	PAST-PLAN	Big Idea	
Reliability	900->1,000 hours MTBF	50%+/-10%	
Maintainability	10 min. fix to 5 min. to fix.	100%+/-50%	
		150%+/-60%	Estimate of total effect of Big Idea on all goals

In addition to looking at the effectiveness of the individual strategies in impacting the qualities, the cost of the individual strategies also needs to be considered, see next section.

Quality to Cost Ratio: For each individual strategy, calculate the quality-to-cost ratio (also known as the benefit-to-cost ratio). For quality, use the estimate calculated in the previous section. For cost, use the percentage drain on the overall budget of the strategy or use the actual cost.

The overall cost figure used should take into account both the cost of developing or acquiring the strategy and, the cost of operationally running the strategy over the chosen time scale. Sometimes, specific aspects of resource utilization also need to be taken into account. For example, maybe staff utilization is a critical factor and therefore a strategy that doesn't utilize scarce programming skills becomes much more attractive.

My experience is that comparison of the 'bang for the buck' of strategies often wakes people up dramatically to ideas they have previously under- or over-valued.

Average Credibility / Risk Analysis: Once we have all the credibility data (i.e. the credibility's for all the estimates of the impacts of all the strategies on all the qualities), we can calculate the average credibility of each strategy and, the average credibility of achieving each quality. This information is very powerful, because it helps us understand the risk involved. For example, "the average credibility, quality controlled, for this alternative strategy is 0.8". Sounds good! This approach also saves executive meeting time for those who hold the purse strings.

Principle 3. Design out unacceptable risk

Unacceptable risk needs to be 'designed out' of the system consciously at all stages, at all levels in all areas, e.g. architecture, purchasing, contracting, development, maintenance and human factors.

Once you have the completed initial IE table, you are in a position to identify the unacceptable risks and design them out of the system. Unacceptable risks include:

- Any quality or resource attribute where the sum of the %Impacts of all the proposed strategies does not total 200%. (A 100% safety factor has been assumed to reduce the risk of failure.)

- Any strategy providing i) a low total for the sum of its %Impacts, ii) very low credibility or iii) low benefit-to-cost ratio.

New strategies will have to be found that reduce these risks. In some cases, it may be decided that the levels set for the objectives are unrealistic and they may be modified instead.

Within software engineering, the art of designing a system to meet multiple quality and cost targets, is almost unknown [GILB88]. However, I have no doubt that there is great potential in conscious design to reduce risks. For example, it is a hallowed engineering principle to be conservative and use known technology. However, this concept has not quite caught on in software engineering technology, where 'new is good', even if we do not know much about its risks. At least, with the use of an IE table there is a chance of expressing and comparing the risk involved in following the differing strategies.

Principle 4. Design in redundancy

When planning and implementing projects, conscious backup redundancy for outmaneuvering risks is a necessary cost.

Under Principle 3, we have discussed finding new strategies. Principle 4, takes this idea a step further. Actively look for strategies that provide backup. An extreme example of this practice is NASA's use of backup computer systems for manned space missions.

Principle 5. Monitor reality

Early, frequent and measurable feedback from reality must be planned into your development and maintenance processes to identify and assess risks before they become dangerous.

I expect the IE information only be used as an initial, rough indicator to help designers spot potential problems or select strategies. Any real estimation of the impact of many strategies needs to be made by real tests (Ideally, by measuring the results of early evolutionary steps in the field). Evolutionary Delivery (Evo) is the method to use to achieve this (See next Principal).

Principle 6. Reduce risk exposure

The total level of risk exposure at any one time should be consciously reduced to between 2% and 5% of total budget.

IE can also be used to support Evolutionary Delivery (Evo) as a budgeting and feedback mechanism during project building and installation of partial deliveries [GILB98, MAY96].

The Evolutionary Delivery (Evo) method typically means that live systems are delivered step by step to user communities for trial often (e.g. weekly) and early (e.g. 2nd week of project).

One of the major objectives of Evo is to reduce and control risk of deviation from plans. This is achieved by:

- getting realistic feedback after small investments
- allowing for change in requirements and designs as we learn during the project
- investing minimum amounts at any one time (2% to 5% of project time or money) so that total loss is limited if a delivery step totally fails.

IE is of use in helping to plan the sequencing of Evo steps. IE tables also provide a suitable format for presenting the results of Evo steps. See Table 5.

TABLE 5

Step-> Attribute	**STEP1** plan %	actual %	Devia-tion %	**STEP2** to **STEP20** plan	plan cumul-ated to here	**STEP21** [CA,NV,WA] plan	plan cumul-ated to here	**STEP22** [all others] plan	plan cumul-ated to here
QUAL-1	5	3	-2	40	43	40	83	-20	63
QUAL-2	10	12	+2	50	62	30	92	60	152
QUAL-3	20	13	-7	20	33	20	53	30	83
COST-A	1	3	+2	25	28	10	38	20	58
COST-B	4	6	+2	38	44	0	44	5	49

Table 5 is a hypothetical example of how an evolutionary project can be planned and controlled and risks understood. The 'deviation' between what you planned and what you actually measured in practice is a good indicator of risk. The larger the deviation, the less you were able to correctly predict about even a small step. Consequently there is a direct measure of areas of risk in the 'deviation' numbers.

The beauty of this, compared to conventional risk estimation methods [HALL98] is that it:

- is based on *real* systems and *real* users (not estimates and speculation *before* practical experience)
- is *early* in the *investment* process
- is based on the results of *small* system increments, and the *cause* of the risk is easier to spot, and perhaps to eliminate, or to modify, so as to avoid the risk.

Evolutionary Project management does not ask what the risks *might* be. It asks what risks have shown up in practice. But it does so at such an *early* stage, that we have a fair chance to do something about the problems.

Principle 7. Communicate about risk

There must be no unexpected surprises. If people have followed guidelines and are open about what work they have done, then others have the opportunity to comment constructively. Where there are risks, then share the information.

Hopefully, readers will by now have begun to understand that Planguage and IE are good means of communicating risk. Let me now introduce Inspection as a third useful method.

Inspection is a direct weapon for risk reduction. [GILB93]. Early Inspections on all written specifications is a powerful way to measure, identify and reduce risk of bad plans becoming bad investments. The key idea is that Major defects are measured, removed, and that people learn to avoid them, by getting detailed feedback from colleagues. A *defect* is a violation of a 'best practice' rule. A *Major* defect is defined as a defect which can have substantial economic effect 'downstream' (in practice, in 'test' phases and in the field). By this definition, a Major defect is a *'risk'.* So Inspection measures risks!

Many people think that the main benefit from Inspection is in identifying and removing Major defects early (e.g. before source code reaches test phases). This is not the case. (My experience is that Inspection is as bad as testing in % defect-removal effectiveness. In very rough terms half of every defect present is not identified or removed.) The really important economic effect of Inspection is not what happens at the level of a single document, but in teaching the people and the organization. The real effect of Inspection is in:

- • teaching individual engineers exactly how often they violate best practice rules
- • motivating the engineers to take rules seriously (<u>really</u> avoid injecting Major defects)
- • regulating flow of documentation, so that high Major defect documents can neither exit nor enter adjacent work processes.

Staff involved in Inspections learn very quickly how to stop injecting defects. Typically, the defects introduced by an author reduce at the rate of about 50% every time a new document is written and Inspected. For example, using Inspection, Raytheon reduced 'rework' costs, as a % of development costs, from 43% to 5% in an eight year period [DION95].

Sampling

One other little-appreciated aspect of Inspection is that you can use it by *sampling* a small section of a large document, rather than trying to 'clean up' the entire document. If the sample shows a high Major defect density (say more than one Major/Page) then the document is probably 'polluted' and action can be taken to analyze the defect sources. A complete rewrite may be necessary using appropriate specification rules or new/improved source documents. This is generally cheaper than trying to clean up the entire document using defect removal Inspection or testing.

Principle 8. Reuse what you learn about risk

Standards, rules and guidance must capture and assist good practice. Continuous process improvement is also needed.

In the previous section, the importance of Inspection was discussed and rules were highlighted as one of the essentials required to support it. It is worth emphasizing the aspect of reuse that is occurring here. The more effort that is put into making rules more effective and efficient by incorporating feedback from Inspections, the more productive the Inspections and the greater the reduction in risk.

Even more benefit can be achieved if what is learnt from Inspection is used to modify the processes that are causing the defects; Continuous Process Improvement has been shown to have a major influence on risk. For example, Raytheon has achieved zero deviation from plans and budgets over several years. They used a $1million/year (for 1,000 software engineers) for 8 years to do continuous software process improvement. They report that the return on this investment was $7.70 per $1 invested on improving processes such as requirements, testing and Inspection itself. Their software defect rate went down by a factor of three [DION95].

Using Inspection, analysis of the identified defects to find process improvements is carried out in the Defect Prevention Process (DPP). DPP was developed from 1983 at IBM by Robert Mays and Carole Jones and, is today recognized as the basis for SEI CMM Level Five. The breakthrough in getting DPP to work, compared to earlier failed efforts within IBM, was probably in the decentralization of analysis activity to many smaller groups, rather than one 'Lab Wide' effort by a Quality Manager. This follows what the quality Guru Dr. W Edwards Deming taught the Japanese; factory workers must analyze their own statistics and be empowered to improve their own work processes.

Analysis of 'root causes' of defects is very much a risk analysis effort [HALL98] and a handful of my clients are reporting success at doing so. But, most are still working on other disciplines like Inspection and others elsewhere in this paper.

Principle 9. Delegate personal responsibility for risk

People must be give personal responsibility in their sector for identification and mitigation of risks.

To back up communicating about risk, people must be given ownership of the risks in their sector (e.g. allocating ownership/sign off of IE tables and giving people specific roles within Inspections).

Principle 10. Contract out risk

Make vendors contractually responsible for risks, they will give you better advice and services as a result.

I would like to point out that contracting for products and services gives great opportunity to legally and financially control risks by squarely putting them on someone else's shoulders.

The effect of contracting a risk to someone else is that:

- you have gotten rid of the risk in some senses, but if they fail, you will still be affected!
- the supplier (assuming they get the risk) will be more motivated to take steps to eliminate the risks,
- be motivated to tell you exactly what you have to do the avoid being hit by risks,
- might come up with a more realistic bid and time plan to cope with the risks.

Summary

Risks can be handled in many ways and at many levels. I have tried to point out some risk management methods which are not so well known or well treated in existing literature. Hopefully, the need to fully integrate risk management into all the development and maintenance processes is clear.

Table 6 and Table 7 recap the ideas presented in this paper. Table 6 is a set of policies for risk management [See GILB98 for more detail]. Table 7 contains 'Twelve Tough Questions' to ask when assessing risk.

TABLE 6: Policy Ideas for Risk Management

- EXPLICIT RISK SPECIFICATION

All managers/planners/engineers/testers/quality assurance people shall immediately in writing, integrated in the main plan, specify any uncertainty, and any special conditions which can imaginably lead to a risk of deviation from defined target levels of system performance.

- NUMERIC EXPECTATION SPECIFICATION

The expected levels of all quality and cost attributes of the system shall be specified in a numeric way, using defined scales of measure, and at least an outline of one or more appropriate 'Meters' (test or measuring instruments for determining where we are on a scale).

- CONDITIONS SPECIFIED

The requirements levels shall be qualified with regard to when where and under which conditions the targets apply, so there is no risk of us inadvertently applying them inappropriately.

- COMPLETE REQUIREMENT SPECIFICATION

A *complete* set of all critical quality and cost aspects shall be specified, avoiding the risk of failing to consider a single critical attribute.

- COMPLETE DESIGN SPECIFICATION and IMPACT ESTIMATION

A complete set of designs or strategies for meeting the complete set of quality and cost targets will be specified. They will be validated against all specified quality and cost targets (using Impact Estimation Tables). They will meet a reasonable level of safety margin. They will then be evolutionarily validated in practice before major investment is made. The Evo steps will be made at a rate of maximum 2% of budget, and 2% of 'project time', per 'incremental trial' (Evo step) of designs or strategies.

- SPECIFICATION QUALITY CONTROL NUMERICALLY EXITED

All requirements, design, impact estimation and Evolutionary project plans, as well as all other related critical documents such as contracts, management plans, contract modifications, marketing plans, shall be 'quality controlled' using the Inspection method [GILB93]. A normal process Exit level shall be *that 'no more than 0.2 Major Defects per page maximum, can be calculated to remain, as a function of those found and fixed before release, when checking is done properly '*(e.g. at optimum checking rates of 1 logical page or less per hour).

7. EVOLUTIONARY PROOF-OF-CONCEPT PRIORITIES

The Evolutionary Project Management method [GILB98, GILB88] will be used to sense and control risk in mid-project. The dominant paradigms will be

- 2% steps,
- high value to cost with regard to risk delivered first.
- high risk strategies tested 'offline to customer delivery', in the Backroom of development process, or at cost-to-vendor, or with 'research funds' as opposed to project budget.

TABLE 7: Twelve Tough Questions

1. Why isn't the improvement quantified?
2. What is degree of the risk or uncertainty and why?
3. Are you sure? If not, why not?
4. Where did you get that from? How can I check it out?
5. How does your idea affect my goals, measurably?
6. Did we forget anything critical to survival?
7. How do you know it works that way? Did it before?
8. Have we got a complete solution? Are all objectives satisfied?
9. Are we planning to do the 'profitable things' first?

10.	Who is responsible for failure or success?
11.	How can we be sure the plan is working, during the project, early?
12.	Is it 'no cure, no pay' in a contract? Why not?

References

DION93: Raymond Dion, "Process Improvement and the Corporate Balance Sheet", IEEE Software, July 1993, Pages 28-35.

DION95: Raymond Dion, Tom Haley, Blake Ireland and Ed Wojtaszek of Raytheon Electronic Systems, "The Raytheon Report: Raytheon Electronic Systems Experience in Software Process Improvement", November 1995, SEI web-site, http://www.sei.cmu.edu/products/publications/95.reports/95.tr.017.html/.
This is an important update of earlier reports.

GILB88: Tom Gilb, "Principles of Software Engineering Management", Addison-Wesley, 1988, 442 pages. ISBN 0-201-19246-2. See particularly Chapter 6, Estimating the Risk (reproduced in Boehm, Software Risk Management, IEEE CS Press, 1989 page 53).

GILB93: Tom Gilb and Dorothy Graham, "Software Inspection", Addison-Wesley, 1993, ISBN 0-201-63181-4, 5TH printing 1998, 471 pages.
This book covers the Defect Detection Process and the Defect Prevention Process, as well as giving sample Rules to check by, defined processes and a well defined set of Glossary terms to aid quantification and comparison. It is a next-generation Inspection, with hundreds of larger and smaller improvements over initial Inspection practices.

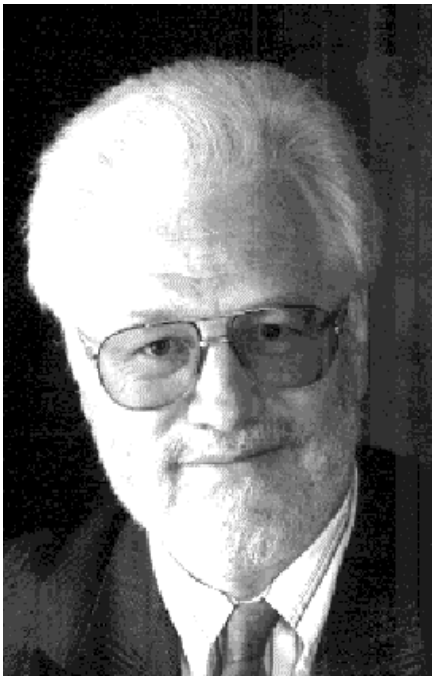
GILB98: Tom Gilb, Various papers and manuscripts on http://www.Result-Planning.com/. The manuscripts include:

. 'Requirements-Driven Management using Planguage' (1995-6)
. 'Evolutionary Project Management' (1997)
. 'Requirements Engineering Language' (1998).

HALL98: Elaine M. Hall, "Managing Risk: Methods for Software Systems Development. SEI Series in Software Engineering", Addison Wesley Longman, USA. (enq.orders@awl.co.uk) , £31.95, 1998, ISBN 0-201-25592-8, 374 pages.
This book is impressive and contains a lot of useful detail and original thought. Anyone interested in risk will enjoy and learn from the book as I did. It does not however deal with most of the subjects in this paper {specification languages, impact estimation, inspections, evolutionary delivery}. This in no way detracts from the book's favorable recommendation. It does tackle 'quantified objectives' much better than other texts.

MAY96: Elaine L. May and Barbara A. Zimmer, "The Evolutionary Development Model for Software", Hewlett-Packard Journal, August 1996, Vol. 47, No. 4, pages 39-45.

The author was at HP in 1989 on a project team who were taught early versions of the Planguage method. The article is full of practical advice and case studies gleaned from ten major projects in eight HP divisions. It must be strongly recommended to anyone interested in the practical implementation of Evo in a project and especially in an organization for many projects. See also article by Todd Cotton in same issue on Evo. HP Journal subscription free to qualified individuals: Write Distribution Manager, HP Journal, M/S 20BH, 3000 Hanover Street, Palo Alto, CA, USA-94304, or Email: **hp_journal@hp_paloalto-gen13.om.hp.com**. HP Journal is available on World Wide Web at **"http://www.hp.com/hpj/Journal.html**." *(Warning HP may have moved this site, but you can get to new site from here)*



Author Biography

Tom Gilb is the author of "Principles of Software Engineering Management" (1988) and "Software Inspection" (1993). His book "Software Metrics" (1976) coined the term and, was used as the basis for the Software Engineering Institute Capability Maturity Model Level Four [SEI CMM Level 4]. His most recent interests are development of true software engineering and systems engineering methods. His sons, Kai and Tor, now work with him.

Tom Gilb was born in Pasadena CA in 1940. He moved to England in 1956, then two years later he joined IBM in Norway. Since 1963, he has been an independent consultant and author.

This paper was edited by Lindsey Brodie, lindsey@brodie.source.co.uk

-----*End of Paper*-------

REvolutionary?
A Software Development Method That Works

Philip Lones
Lucent Technologies
Octel Messaging Division

Introduction

- In the software development environment, projects are always delivered on time, within budget and just what the customer wanted.

NOT

- In reality, software projects are usually over budget, late, and not always what the customer wants.

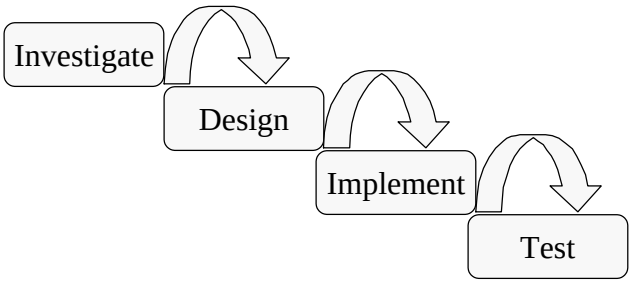
Introduction

In order to be successful and compete in a global market, organizations must have a process that will move reality closer to the ideal environment, i.e. on time, within budget, and the right product.

3

Methodologies

- **Waterfall Model**



Static Model. Each Phase completed before the next phase is initiated.

4

Methodologies

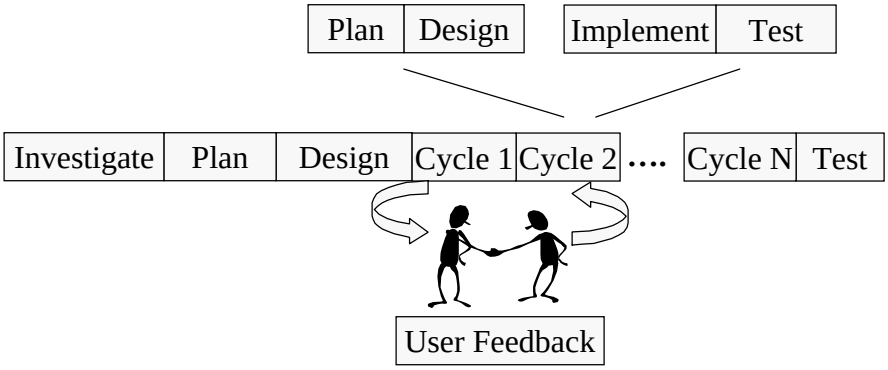
- **Several other methodologies**
  - Rapid Prototyping
  - Spiral
  - Incremental
- **Commonality:**
  - Lack of user involvement throughout the software development lifecycle (SDLC)
- **A different approach is needed**

5

EVO

- **Evolutionary Delivery Model - EVO**

  Tom Gilb (http://www.Result-Planning.com)



6

EVO

- **Key Concepts**
  - Early, frequent iteration
  - Plan, design, implement and test in each cycle
  - Result orientation, not software development process orientation
  - User orientation

7

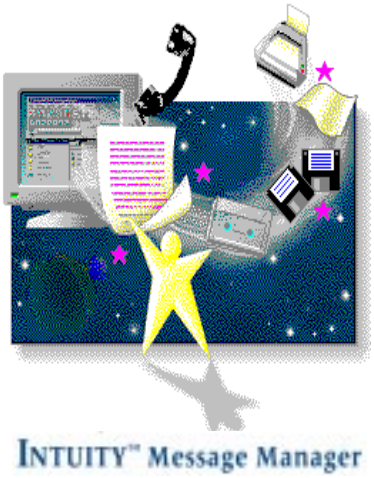
EVO

- Each delivery step is 1%-5% of total project effort
- “How little can we do”, not “how much”
- Give the customer something of value in each delivery
- “Best” features first
- Customer involvement

8

Message Manager

- Desktop client
- Visual representation of INTUITY™ AUDIX messaging system
- Voice, fax, text, binary can be viewed/heard and created via PC
- Can send and receive email with INTUITY™ Internet Messaging
- Over 1,000,000 seats sold



9

Message Manager

- First release 1994
- 6 major releases
- Visual Basic client
- As product evolved so did process - EVO model

10

Message Manager

- **MM incorporates key concepts**
  - Plan, implement, test in each cycle
    Review feedback, modify design, test features
  - Early, frequent iteration
    average 15-20 build cycles per release
  - User oriented
    Base of internal and external “friendly” users

11

Message Manager

- **Plan, Implement, Test**
  - Technical Prospectus
    - High level description of product
    - Prioritized list of features
  - Detailed Requirements
  - User Interface
  - Prototype GUI
  - System Architecture

12

Message Manager

- **Plan, Implement, Test (cont.)**
  - Features assigned by Project Manager and incorporated into delivery cycles
  - Modification Requests (MRs) assigned and incorporated into delivery cycles
  - Code read and unit test

13

Message Manager

- **Plan, Implement, Test (cont.)**
  - End of cycle
    - Sanity test - acceptance test cases
    - MR test - functionality of bug fix
    - Features - functionality of new features
  - End of project
    - System - customer based scenarios
    - Regression - does it still work

14

Message Manager

- **Early, Frequent Iterations**
  - 15-20 deliverables per release
    - Weekly update to code base
    - Available to customer when base is stable

15

Message Manager

- **User Oriented**
  - Established beta users: 700
  - Internal: 400
    - Department
    - Lucent
  - External: 300
    - 12 Sites

16

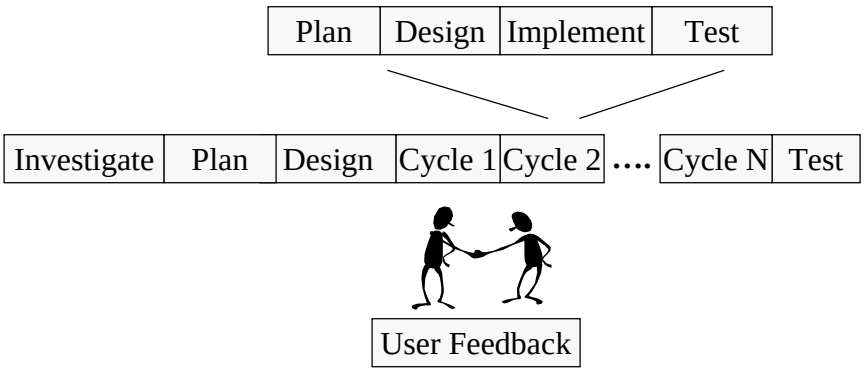
Message Manager

- **User Oriented (cont.)**
  - Solicit feedback
    - MM Hot Line
    - Surveys

17

Message Manager

- **Analyze feedback, redesign, implement, test**



18

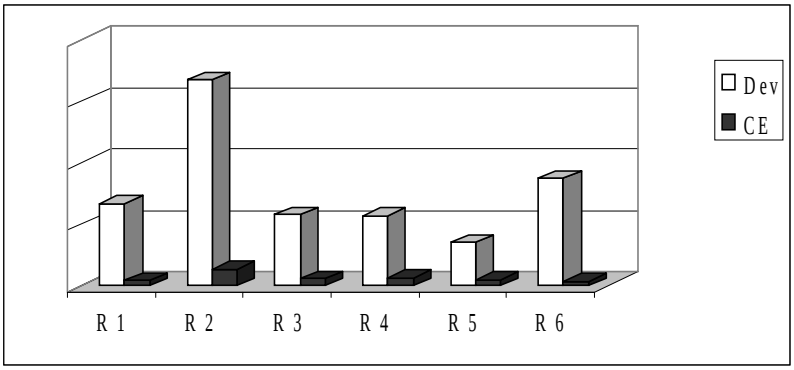
Message Manager

- **Impact**
  - Respond quickly to usability issues
  - Allowed testing of features much earlier in the SDLC
  - Review scope of release and make changes early in the SDLC
  - Product the user wants, not one we think they want

19

Message Manager

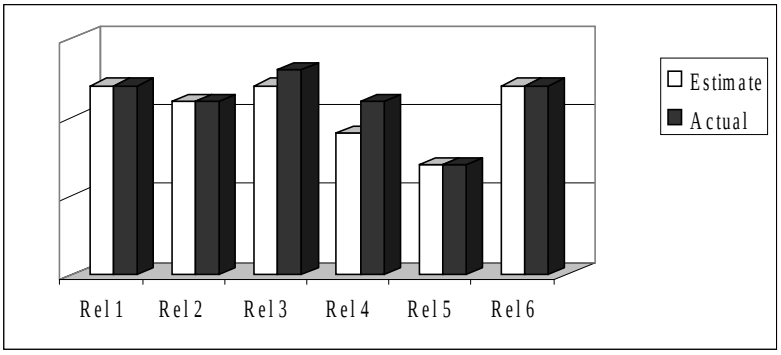
MRs during development vs. MRs during CE



20

Message Manager

Estimated Development to Actual Development Time



21

Critical factors

- Project Manager
- Team commitment
- Open communications
- Manage the “friendly” customer base

22

Conclusion

- Competition in the global market requires successful process
- EVO model
  - Early, frequent builds
  - Value to the customer
  - User involvement
- Product to market the customer wants
  - On time and within budget

23

References

- Principles of Software Engineering, Tom Gilb, Addison-Wesley, 1988
- The Evolutionary Project Managers Handbook, Tom Gilb, manuscript, (http://www.Result-Planning.com)
- Hewlett Packard Journal
  (http://www.hp.com/hpj/aug96/augart3.htm)
  (http://www.hp.com/hpj/aug96/augart4.htm)

24

Seizing Control of the Development Lifecycle

Nick Borelli
Microsoft Corporation

Introduction

- My background
  - Life B.T. (before testing)
  - The tester awakens
- Expectations of this talk
  - Why I'm here today
  - Management and Process Oriented
  - Empowering the Test Org to improve Development Methodology

Addressing our differences: Products & Paradigms

- The labels we live by
  - CMM, ISO, SEI, etc.
  - MIL-Spec
  - Shrink Wrap
- Common ground
  - We all want a better process
  - Higher quality
  - Accurate Schedules

Why should Testing drive this?

- Biggest stakeholder?
- Desire
- Pragmatism
- Can any one else do it better?
- Don't forget the Tester's Mission

Seizing Control: Step 1, Collecting Data

- In-depth Post-Mortem: Test
  - Various methods to collect data
  - Spend as much energy on your group as critiquing others
  - Important to reach consensus on items in conflict
  - Roll-up big ticket items from Cross-Functions only

Seizing Control Step 1: Collecting Data

- In-depth Post-Mortem, Cross-Functional
  - Powerful Process to collect both Best/Worst Practices
  - Techniques
    - Moderated Post-Mortems
    - Self-Administrated Post-Mortems
  - Frequency & Follow-up

Seizing Control Part 2: Analysis

- How can the current Development model hurt test?
  - Identify areas both from Post-Mortem and ones missed
  - How could these change?
  - How would change affect other teams?
  - Would these improve the chances for higher quality and earlier ship date?
  - Refine your "laundry list"

Seizing Control Part 3: Communicating Results

- Rollout of Changes to your Test/Quality Organization
  - Hash out with your leads/managers
  - Discuss changes with entire team
  - Pitfalls
    - Solving the right problem the wrong way
    - Process change overload
    - Cost/Benefit analysis

Seizing Control Part 3: Communicating Results

- Rollout of Changes to Cross-Functions
  - Meet with key stakeholders
  - Identify benefits
  - Review issues that the Post-Mortem identified
  - Install feedback process

Seizing Control Part 3: Communicating Results

- Testing-driven Development Lifecycle
  - Rolling up changes in Test/Development Plans
  - Modifying Milestone Criteria
  - Milestone Post-Mortems
  - Use Microsoft Project™ for tracking

Seizing Control Part 4: Examples

- Real-World Examples of Process Changes
  - Specification feedback
  - Defect discovery: How can it be earlier?
  - Automation re-work too costly
  - Test documentation taking too much time
  - Quality of code
  - Development schedules/process out of sync with milestones

Seizing Control: Conclusions

- More effective organization
- Higher quality
- Less conflict
- Improved communication
- Improved morale

Seizing Control: Conclusions

- Success breeds success
  - Peer test groups
  - Cross-functional groups in your company

Thanks for listening!

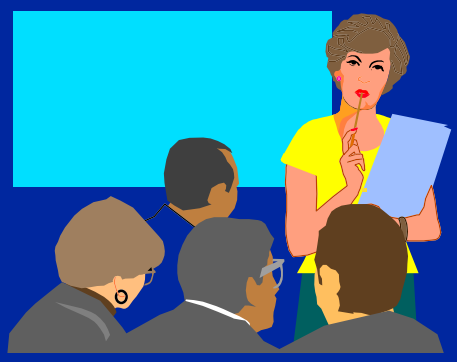
Questions or additional notes, send email to:

Nickbo@Microsoft.com

Tactical Improvement Projects:
Real-Life Lessons in Leading Change

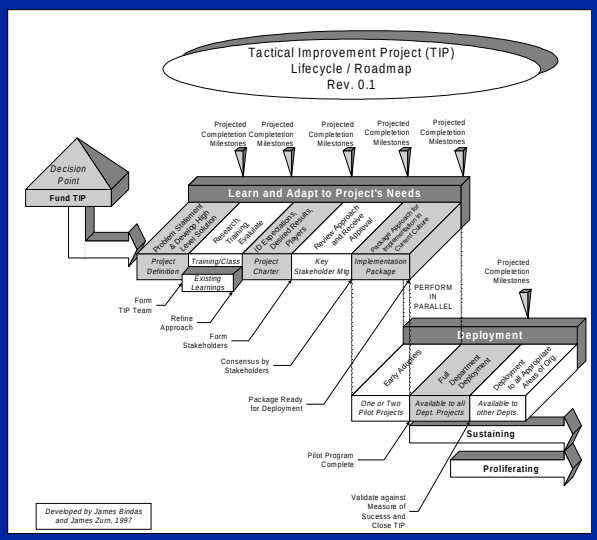
James R. Bindas
james.r.bindas@intel.com
Microprocessor Products Group / Design Technology
Intel Corporation

Tactical Improvement Project (TIP)



- **What is a TIP?**
  - **A specific, bounded improvement project**
  - **A component of a strategic improvement roadmap**
  - **Authorized and monitored by senior management**

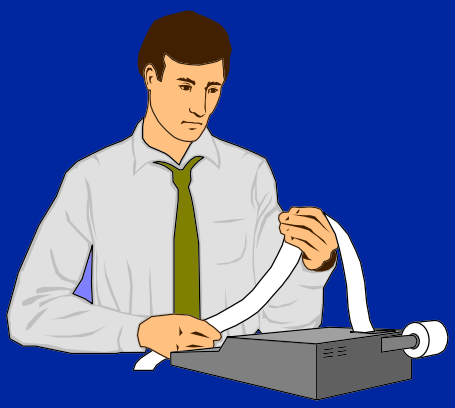
Typical TIP Lifecycle/Roadmap



James. R. Bindas

Slide 3

Key Values of a TIP

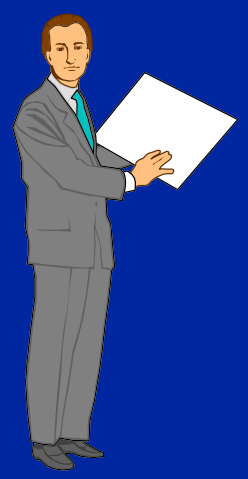


- Can be led by an outside person
- Working groups -
  - Are formed to develop and drive a group solution
  - Consist of stakeholders
    - Must be cross-organizational
- The process should be simple
  - I.e. Low Overhead

James. R. Bindas

Slide 4

First Steps in Leading a TIP



- Develop a Project Definition which includes:
  - Team Members / Roles
  - Project Overview
    - Including problem statement
  - Stakeholders
  - Linkage to Business Objectives
  - TIP Scope & Complexity
  - Mission Statement, Objective and Success Criteria
  - Key Milestones and Deliverables
  - Key Assumptions
  - Critical Success Elements
  - High Level Implementation Roadmap

Mistake Number #1
"Not Writing a Project Plan"

- Project Definition:
  - Confirms assumptions
  - Ensures a common understanding
- Experience has shown, projects without a good understanding...
  - Will result in lost effort and time

Researching the TIP

- TIP Working Groups are formed
  - Members should include stakeholders and be cross-organizational in nature
  - Ensures a cross-organization solution
- Understanding Involvement
  - A combination of formal instruction and transferred knowledge
  - Can begin the process before a TIP Working Group is formed
- The Project's Honeymoon
  - Members in high spirits
  - Confident they are going to make a difference

Mistake Number #2

"Not Taking Advantage of Previous Knowledge"

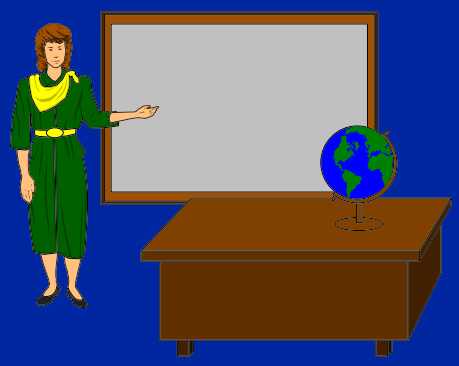
- Odds are that other departments and/or companies attempted similar efforts
- Avoid repeating the same mistakes

Mistake Number #3

"Failure to Document Agreements and Check Assumptions"

- Early on many perspectives and agreements are expressed
- Later during execution, interpretations of these discussions vary
  - This causes conflicts
- Keeping these agreements in a central, accessible location, e.g. intranet web site, will help prevent conflicts

Forming and Approving the Project Charter



- Broken down into:
  - Mission Statement
  - Objectives
  - Approach
  - Deliverables
  - Success Criteria
- Important: Keep the project focused and short-term
  - Hard to maintain momentum
  - Team dynamics change
  - Management values change
  - Re-orgs can devalue the TIP

Mistake Number #4

"Skip the Project Charter or Decrease its Content"

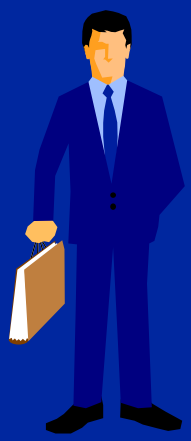
- Expectations need to be set for management
  - Otherwise, the project may never be viewed as a success by management

James. R. Bindas

Slide 11

Mistake Number #5

"Rely on Selling the TIP based on its Benefits"



- Classic approach is to sell the TIP based on its benefits
  - TIP's "cause & effect"
- Often, the TIP "buyer":
  - Does not possess the "ability"
  - Or, may not desire the outcome
- E.g. Exercise may prolong your life
  - If true, why doesn't everyone exercise?

Reference: Emery, Dale H., Resistance as a Resource, (Slide handout, 1/13/1998), © 1997-98

James. R. Bindas

Slide 12

Pilot Project Implementation

- A working group's efforts tend to split between:
  - Developing the approach
  - Implementing the approach, via a pilot
- Working Group members will start to disagree
  - Not on the intent
  - But, on the implementation
    - Sometimes, petty concerns can turn ugly

James. R. Bindas

Slide 13

Mistake Number #6

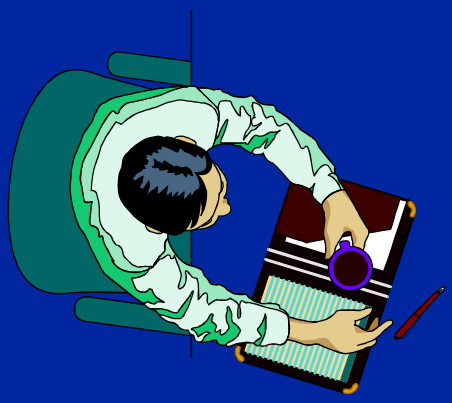
"Select Challenging Pilots"

- An urge may exist to select a pilot that will make a difference
- Remember, the intent of the pilot is to confirm the TIP's approach
  - Selecting a complicated pilot will put the TIP's success at risk

James. R. Bindas

Slide 14

Mistake Number #7
"Not Continuing after the Pilot"



- After the pilot is completed, a let down can occur
  - No follow-up
  - No planning for future deployment
  - Key lessons are not recorded and/or acted upon
- The TIP fades away

James. R. Bindas

Slide 15

TIP Deployment

- The TIP approach has been proven
- Key Stakeholders have declared the TIP a success
- Deploying the TIP is now the focus
- Working Group membership begins to change

James. R. Bindas

Slide 16

Mistake Number #8 and #9

"Expanding the Deployment Period" and "Assuming the Allocated Resource will Always be Committed"

- The longer the deployment, the greater the risk your TIP will die a slow death
- During the TIP, some of the following "will" happen
  - "Higher Priority" projects will come along
  - Re-org or management values will change
  - Personnel will turnover or become re-assigned
- Rule of thumb
  - If commitment is not obtained, come back when obtained

James. R. Bindas

Slide 17

How to keep the TIP "Alive"

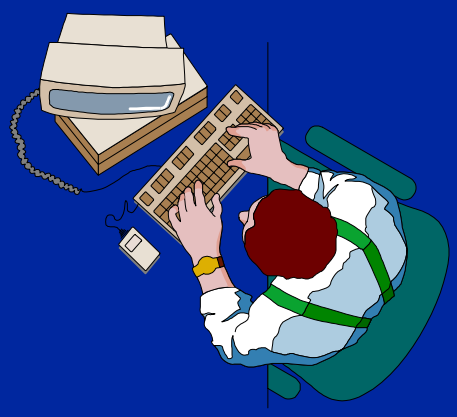
- TIPs can easily become derailed
  - For any number of reasons
- Three preventative methods
  - Patience and Persistence
    - Keep reminding them
  - Make it competitive
    - Make a scorecard and distribute it to senior management
  - "Hail Mary"
    - Send out an email stating
      - *No progress has been made*
      - *I am ceasing my involvement*
    - Worked for me in the past

James. R. Bindas

Slide 18

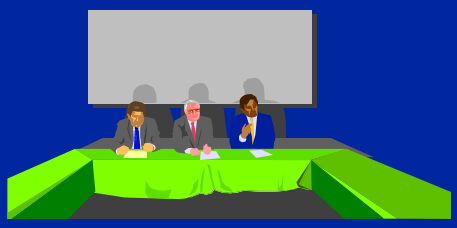
Mistake Number #10

"Thinking the Same Approach will work Every Time"



- Over time:
  - Key lessons are applied
  - The approach is optimized
  - Environment changes
  - Personnel changes
- Need to provide "headroom" to allow the TIP grow over time

Conclusion



- Discussed:
  - TIP Basics
  - TIP Lifecycle
  - TIP Implementation
  - TIP Mistakes
- Above all else, TIPs:
  - Need to be managed constantly
  - Require frequent attention to details
  - Need key-insights to handle human-related activities

Tactical Improvement Projects - Real-Life Lessons in Leading Change

James R. Bindas
james.r.bindas@intel.com

Microprocessor Products Group / Design Technology
Intel Corporation

Mail Stop JFT-102
2111 NE 25th Avenue
Hillsboro, OR 97007
(503) 264-8869

Abstract

This paper describes the typical lifecycle of an improvement team, how to lead an improvement effort and the common experiences encountered during improvement projects.

Keywords

Keywords include Tactical Improvement Projects, Project Definitions, assumptions, stakeholders, pilot, deployment and common mistakes.

Biography

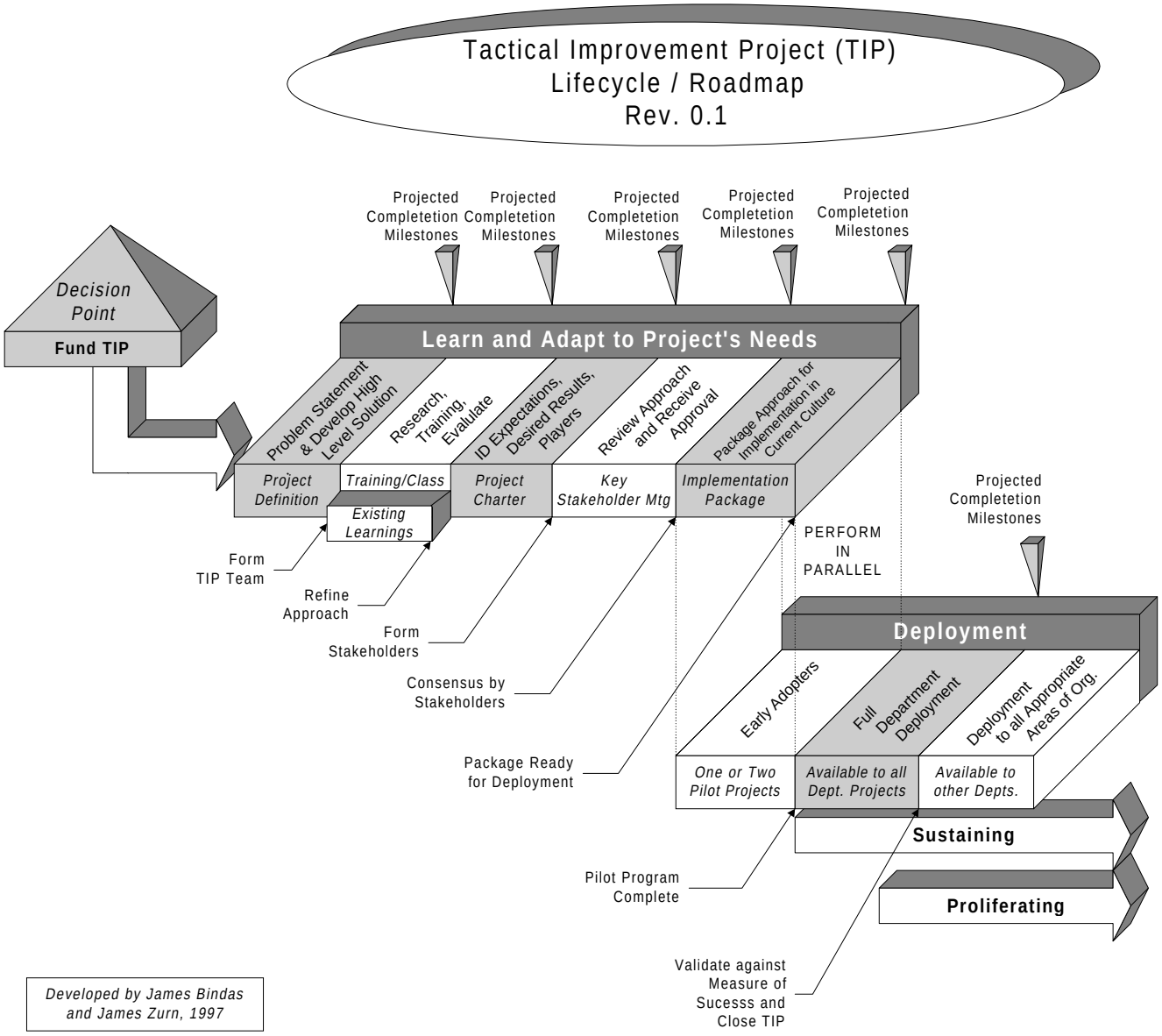
James R. Bindas is a Software Process Engineer with Intel Corporation in Hillsboro, Oregon. His tasks include implementing, standardizing and improving organizational and software development processes. James has been working with Intel for ten years in various software roles ranging from software tester to project leader. Before joining Intel, James worked for RCA/GE Solid State and Harris Semiconductor as a Quality Assurance Technician. James holds a Master of Science degree in Computer Science from Stevens Institute of Technology, and a Bachelor of Science in Graphic Communications from California University of Pennsylvania.

Introduction

Leading a Tactical Improvement Project (TIP) team is a challenging task. My experiences as a TIP leader in a cross-organizational setting have enhanced my knowledge in this subject. A sample of my experiences are shared in this paper.

Typical TIP Lifecycle / Roadmap, Values and General Leadership Process

To be effective in leading a TIP, creating a roadmap is essential. Without it, the leader's efforts often result in confusion and frustration in accomplishing tasks. No "silver bullet" approach to leading a TIP exists. Most often, a set of values, a roadmap and a process (similar to what is detailed below) is followed.



Key Values to incorporate into a TIP

- Can be lead by an outside person, but working groups need to be stakeholders – Working groups must have a cross-organizational mix.
- Working groups should adhere to a short life cycle.
- The process should be simple. (i.e., Low Overhead)

General TIP Leadership Process

- Inputs to the process -
  1. Select an individual or group to organize an improvement team to solve a specific problem.
- Before starting a checklist -
  1. Obtain senior management sponsorship.
  2. Devise a high-level project scope with the sponsor.
- Task list -
  1. Develop a Project Definition that solidifies the project's purpose/goals.
  2. Research the project by:
     a. Examining past or existing related-experiences and projects.
     b. Forming a working cross-organizational group to develop and implement an approach.
     c. Obtaining training, if necessary.
  3. Combining training with the Project Definition to form an approach that will implement the TIP.
  4. Presenting the approach to the Key Stakeholders for validation and settling any issues that require their attention.
  5. In parallel, perform the following:
     a. Develop the implementation package of processes, templates, and checklists.
     b. Pilot the implementation packages on one or two projects.
  6. Feed Learned Lessons back into the TIP's process.
  7. Deploy the project within the department.
  8. Deploy the project within the greater organization.
- Success is achieved when...
  1. The TIP team and key stakeholders agree upon the TIP project definition, approach and success criteria.
  2. The sponsor approves an implementation plan, success criteria and schedule.
  3. The sponsor accepts the pre-determined success criteria that are met according to plan.
- The process is completed when...
  1. Success criteria are met.
  2. The intended receiving party accepts the deliverables.
  3. A post-project review has taken place and its results are recorded.
  4. A project notebook or an intranet web site is completed, with Best Known Methods documented and transferred to the responsible organization for storage.
- Process will output…
  1. Stated deliverables.
  2. Feedback will be generated between the sponsor and team members.
  3. A Process Improvement (3 Ring) Notebook and/or Web Site will be produced that contains:
     - A plan and schedule.
     - Meeting minutes and other related correspondence.
     - Deliverables.
     - A post-project report.
     - Any other related documentation.

First Steps in Leading a TIP

Once you are assigned to lead a TIP, the first step is to interview the sponsor about the project. The purpose of the interview is to gain an understanding of what is expected in terms of deliverables, what particular process is to be followed, who are the people involved, and what is the intended timetable. All this information should be included in the project definition and should be presented at a stakeholder meeting.

Typically, the items to include in this document are:

- Team Members (with roles and responsibilities)
- Project Overview
  – Including problem statement
- Stakeholders (listing interest in project and benefits)
- Linkage to Department Business Objective
- TIP Scope
- TIP Complexity (in terms of level of organization involvement, technology complexity, business complexity and geographical involvement)
- Project's Mission Statement, Objective and Success Criteria (NOTE: Success criteria are optional at this point, but needs to be defined in the Project Charter)
- Key Milestones and Intended Delivery
- Key Assumptions (invalid and valid)
- Critical Elements needed by the TIP to be successful (may involve management support, tools and customer needs.)
- High Level Implementation Roadmap (keep this simple, otherwise, it will drive-off needed support.)

Mistake Number #1: Not writing a Project Definition!

Many people don't understand the value of spending time writing a project definition. After all, "we all know what the project covers, right?" These fatal words can doom a project from the start. While some may not be convinced at the outset that writing a project definition is critical, once the project definition is presented to the stakeholders, they will overwhelmingly appreciate the document, without exception.

One benefit of a project definition is to verify assumptions. Most of the time, a few assumptions exist that are not universal within the team. For example, a manager and two co-workers were requested to introduce a major process improvement into a department. Six weeks were spent analyzing the business practices, developing process flows and developing process tools. The improvement processes portfolio was presented to the department manager. He was really impressed, however, all that he really wanted was a simple scorecard to keep track of process steps and dates that had been agreed to. Another department had already developed a similar scorecard, which could have easily been adapted to their needs. Six weeks of work were never utilized.

Researching the TIP

At this point, more research is performed on the TIP. This section is broken down into two sections:

- Form TIP Working Groups
- Formal Instructions / Transfer Leanings

Working groups or teams are formed, if they have not already been established. The team members should represent each of the stakeholder groups. These representatives should be empowered to lead change in their organization. This will ensure that their interests will be taken into account and decrease resistance when the approach is implemented. For example, resistance can be heard as, "this is our approach, not your approach."

The working group researches the TIP by a combination of collecting ideas and taking formal instruction. This combination will provide a better understanding of expectations in implementing the approach.

At times, research is begun before the working teams are formed. This is fine as long as information is documented and available to the group when it is formed.

Mistake Number #2: Not Taking Advantage of Previous Leanings!

Chances are that other departments, inside and outside your company, had tried similar improvement efforts. Question these efforts and learn from them. Understand where the projects had successes and failures. Learn and build upon past experiences. Otherwise, needless repetition of failures may occur.

This period tends to be the project's honeymoon because the team's spirits are high. Each member believes they are part of a greater solution and they are going to make a difference. Disagreements are insignificant and workable at this point. Take advantage of this environment and nurture it, as long as you can. The longer you maintain this environment of fellowship, the more productive your group will be.

Mistake #3: Failure to Document Agreements and Check Assumptions!

Take advantage of goodwill at this time and define the project, as much as possible. Document all meetings and agreements. This information should be placed in a central repository, such as an intranet web site. Some feel this is unnecessary. However, my experience reveals that conflict can occur over past agreements. In a previous TIP, two out of three members assumed that write-access would not be granted for group members. The third person believed the contrary. If documentation has been available, this episode could have been avoided.

Forming and Approving the Project Charter

Form an approach on how to complete your effort and how to measure its success. Writing a Project Charter forms the foundation for this.

Mistake #4: Skip the Project Charter or Decrease the Content!

The TIP effort will be judged on its success. The question is who will judge the effort and how will they judge this one? Success is translated to the TIP's approach, deliverables and timeline. These expectations need to be decided with management before work is begun. Otherwise, the project risks the danger of being redirected .

Traditionally, the charter is broken into the following categories:

- **Mission Statement** – *Why are we here? Why are we investing in this effort?*
- **Objectives** – *What are we trying to accomplish?*
- **Approach** – *How are going to accomplish this?*
- **Deliverables** – *What are we going to deliver and when?*
- **Success Criteria** – *How do we know when we are successful?*

It is not important whether the charter is in a fancy or a simple straightforward presentation. Focus on small, short-term simple and achievable goals, while building a long-term roadmap. I based this approach on the following personal experiences:

- It is difficult to keep the project's momentum going for longer than a quarter or two.
- Team dynamics change, as people leave and join the team.
- What is currently important to management may not be important later.
- Departments can reorganized and devalued the TIP's potential benefit.

Mistake #5: Rely on Selling the TIP on its Benefits.

In any TIP, the standard selling point is demonstrating its benefit, especially when tied to a business objective (values). Don't stop short at this point. Initiating a TIP will require people to change. Therefore, people need to believe that the TIP will make a difference (cause and effect beliefs) and it is achievable (abilities), as well as beneficial to a business objective. Dale Emery[1] once said:

People will change whenever:

- ***They believe they can perform the new behavior (abilities).***
- ***They believe they know the likely outcome (cause and effect beliefs).***
- ***They want the outcome (values).***

Pilot Project Implementation:

At this stage, working teams now become focused on implementing the process into pilot programs. A parallel effort is involved: one effort is developing the approach, while another effort is managing the pilot. (Actually, much of the development approach occurs before implementing it to pilot programs.) At this time, the approach is adjusted according to feedback from the pilot programs.

Mistake #6: Select Challenging Pilots

Generally, two pilot projects are launched to support the TIP. The criteria in selecting a pilot program can be characterized as, "picking low hanging fruit," or, in other words, choosing easy-to-implement pilots. The intent is to validate the TIP approach, which is complicated enough; adding more complexity by choosing difficult pilots only increases the chance of the TIP to fail.

Another characterization of the pilot-implementation phase is that team members begin to disagree openly about implementing the approach, while agreeing with the TIP mission, objectives and the basic approach. At this point, everyone has a different perspective on the implementation. An example from experience is about the wording and format of an e-mail notification message. One person had the idea to write the message with less information, but formatting it in a way to gain maximum attention. Another person had the idea of e-mailing the maximum information possible and letting the receiver choose important information. These disagreements need to be resolved as soon as possible. Otherwise, a snowball effect could occur with each party disagreeing with each other and expending large amounts valuable energy over a small matter. This example also proves that "documenting" agreements in advance will help minimize these unnecessary confrontations.

Mistake #7: Not Continuing after the Pilot

Launching pilots is taxing. The planning requires preparation, training and execution, which drains valuable resources. Despite the considerable resource dedication, things do not always go as planned. It is my experience that key lessons are often not utilized or acted upon, which wastes experiences learned from the pilot and jeopardizes the pilot's success.

Deployment

After the pilot approach has been proven, the working group will deploy the pilot to other parts of the department and organization. Group membership most likely will change in size and membership. This is a normal occurrence because of the changing dynamics of the organization and the change in mission and objectives.

Mistake #8: Expanding the Deployment Period

If the TIP is successful, people may begin asking you to deploy the TIP in their area. Before responding yes, confirm their schedule and priorities. If you suspect that they are unable to commit to the deployment in a short time- period, pass on them until they are able to commit. In my experience, they may be interested, but if they cannot commit to a time frame, they never will.

[1] Emery, Dale H., *Resistance as a Resource*, (Slide handout, 1/13/1998), © 1997-98

Mistake #9: Assume that an Allocated Resource will Always be Committed

Things often change. Resources, once allocated, can vanish without warning. A host of "valid" business reasons can affect the allocated resource. This can range from higher priority projects to the resource leaving the organization. I was pulled from leading a TIP, at the most critical time, to work on a higher priority task. This reinforces the need to keep the TIP deployment period as short as possible, while following a long-term improvement roadmap.

During deployment time, it is important to keep focused. Other priorities will emerge and organization changes will threaten to derail the deployment progress.

I was asked to drive the transfer of a software development process from one group to another. When I met with both teams in late November 1997, we agreed to start the transfer process in January and conclude by March. Delays caused the formal transfer process to be completed in August. Unforeseen events occurred that had a higher priority and conflicted with this project. I was lucky that the process transferred at all. Another process started in late September 1997 and required over ten months to transfer. When it was transferred, the intended receiver changed jobs and the process was no longer valued. The perceived need did not exist for the new receiver.

Methods to keep the project focused may be considered passe, but often prove to be effective. These tools include persistence, patience, completion, and the occasional "hail Mary" ploy. Persistence means constantly reminding the parties of the TIP benefits, roadmap, and their obligations to the roadmap. Patience means understanding the organization culture, current working environment and other tasks that need attending.

At times, it is beneficial to use internal competition to insure the success of a TIP. My former manager led a TIP where department managers reported progress of their obligations to him. He gathered the results, assigned them into a scorecard format, and presented them at department meetings. The department members, who were competitive by nature, then devoted more energy into completing their obligations.

If you reach a point where nothing works, you may try using the "hail Mary" ploy, which is a high-stakes gamble and should be used as a last resort. Basically, send out e-mail or hold a meeting stating:

- No progress has been made on the TIP.
- My involvement in the TIP will cease.

This ploy has worked for me in two-out-of-three attempts. If the TIP is highly valued, but somehow fell in priority, the involvement level will rise to the occasion. If the TIP is not valued, the involvement level will further decline. Any further involvement will not be valued. This will allow you to exit the TIP gracefully.

Mistake #10: Thinking that the Same Approach will work Every Time!

What makes the software world exciting is despite everyone having the same type of challenges, the particular solution implementation is a bit different for each situation. Software processes, in any one organization, are a dynamic combination of the organization's culture, business goals, customers, and capabilities. If an organization is weak in project scheduling, the solution is to utilize a standard-based and industry-proven estimation process. The challenge is how to tailor and integrate a standard technique in a dynamic environment. This only comes with experience.

As you proceed through a broader deployment, the pilot approach will evolve over time. Sometimes, it will change radically. This is to be expected for a number of reasons, but not limited to:

- Key pilot lessons are applied to the approach.
- The approach was optimized for maximum benefit.
- The environment changed.
- Different organization personalities exist.

While we always strive to create a "cookie cutter" approach that works in any situation over time. The fact remains that reality evolves over time and once common approaches may not be as effective as it once was. If your TIP approach does not provide "headroom" to evolve, then it will be destined to become an artifact of history.

Summary:

I have discussed the details of a TIP, a typical TIP Lifecycle and how TIPS are implemented. I have also shared some personal experiences to assist you in avoiding some common mistakes.

Remember, above all else, the challenge of leading a TIP lies in constant project management, attention to details, and ESPECIALLY having key insights in handling human-related issues. Despite all logic about why a TIP is a good idea, if the TIP is not humanized for the culture intended it will fail.

Quality Week '99: 8M2

Them and Us: Communication Between Development and Test

Scott Young

perotsystems™

The Argument

The Test Team do a better job if they listen to, learn from and build a strong relationship with the Development team.

Not Leadership 101: We are all supposed to be on the same side

perotsystems™

Why Be Concerned

- The earlier in the development process a problem is found, the cheaper it is to fix
- Technology is easy, people are difficult

perotsystems™

The Problem

Common goals but

- Different focus
- Different skills
- Understanding not always in step

perotsystems™

Shared Goals

The Development and Test teams:

- Need to ask the same questions and resolve the same issues of interpretation and ambiguity of the requirements
- Need to speak to the application users to understand workflow

perotsystems™

Different Focus: Development

- Building the application
- Technology
- Specifying an architecture
- Writing code
- Designing a database

perotsystems™

Different Focus: Test

- V&V of the system
- Building test conditions, cases and expected results
- Gathering of realistic data
- Application look and feel

perotsystems™

Timing

- The Development team are ahead in the analysis
- Test will benefit from talking to Development early in the process
- Development will benefit from talking to Test late in the process

perotsystems™

Building a Strong Team

Factors

- Environmental
- Procedural
- Organizational

perotsystems™

Environmental

A problem

- Development projects are temporary

Some solutions

- Put the teams together. If this is not possible use technology
- Appoint ambassadors

perotsystems™

Procedural

- Write Requirements
- Use problem tracking software
- Developers ‘clarifying’ requirements

perotsystems™

Problem Tracking: The Observation

- Its an observation first
- Teach the team what to put in Observations
- Teach the team what **not** to put in Observations
- Never forget to talk to the Developer

perotsystems™

Organizational

- Teach the development team about testing
- Teach the test team about the technology
- Use team building

perotsystems™

Putting it all Together

- Effective communication between Test and Development is an integral part of any successfully development
- The best location for teams is side by side
- Formalize defect tracking to facilitate communication
- Enforce configuration control
- Organize cross team groups, appoint ambassadors

perotsystems™

Them and us: Communication between Development and Test

Contents

INTRODUCTION **2**

THE SOFTWARE CHALLENGE 2
DEVELOPMENT AND TEST 2

THE IMPORTANCE OF COMMUNICATION **3**

THE IMPORTANCE FOR THE TEST TEAM 3
THE IMPORTANCE TO THE DEVELOPMENT TEAM 4
THE BALANCE OF THE RELATIONSHIP 4
FAILURE TO COMMUNICATE 4

FACTORS **5**

ENVIRONMENT 5
Proximity of Teams *5*
Workflow *5*
PROCEDURE 6
Defect Tracking *6*
Observation Tracking *6*
Communicating with Observation Reports (OR's) *6*
Classic Mistakes to Avoid with OR's *7*
Configuration Control *8*
ORGANIZATIONAL 8
Team Structure *8*
Training *9*
Project Size *9*

SUMMARY **10**

© Copyright 1999 Perot Systems Corporation
All Rights Reserved

Introduction

The Software Challenge

For all software development projects there are many challenges that the project leadership and team must overcome to produce an application that is a business, financial and technical success. The traps for the unwary exist from the initial conception of the project and appear in many different forms through to initial implementation and beyond. The same causes appear on the post mortem for most failed projects: feature creep, loss of key staff, no project sponsor, technical complexity, lack of quality control, failure to reach market in time, bad staff morale etc. Even when a project does achieve a measure of success it has more than likely been affected by one or more of the most common problems. After decades of progress, the ideal, on-time, on-budget, successful-in-the-marketplace project, still draws attention.

One linking factor that influences every aspect of a project's success is effective communication between the teams involved in the application development: business sponsors, marketing, customers, developers, and testers. Communication weaknesses can be the root cause of many symptoms from not being able to say 'no' to feature creep to not listening to what the market needs.

Development and Test

This paper addresses the internal communication between the main teams responsible for the technical delivery of an application: Development and Test.

The Development team, as the term is used here, is responsible for the architecture, design and implementation of all the required functionality. The skills of the team are analysis and technological. They know about which development tools to choose, about third normal databases, about the theory of human computer interaction and user interface design standards and apply that knowledge to the problem in hand. They analyze, design and implement a new application, creating a working product from a simple specification.

The Test team is responsible after-the-fact for the quality of the application[1]. They take the application and complete both validation and verification prior to implementation or product release. The test team is intent on the application look and feel, the gathering of realistic data, the identification of business scenarios to use for test cases and the application's overall conformance to requirements. The testers are the first users of the system and become advocates for the real users.

The Development team's technical focus and the test team's functional focus can often lead to difficulties in communication between the teams. The two teams have different skill sets and focus on (and like) different aspects of the development process. If ignored this difficulty can grow into a them-and-us mentality that can represent a serious risk to a

[1] I am ignoring the Quality Assurance function here.

© Copyright 1999 Perot Systems Corporation
All Rights Reserved

project. Some symptoms of failure that would appear include: low product quality, significant rework, low team morale, loss of key staff and extended cost and schedule.

The key tool to preventing a them-and-us environment is effective communication between the teams. Only through such communication can the necessary trust and understanding be developed that will support a successful implementation. Effective communication can be established through environment, procedure and organization:

- *Environment* includes those aspects that influence face to face communication.
- *Procedure* includes establishment of a method for the proper control of requirements and an efficient and appropriate defect tracking system
- *Organization* includes those influences that related to team structure including both project and enterprise wide factors.

The Importance of Communication

The Importance for the Test Team

For the Test team, communication with the Development team is essential to completing the testing task successfully. There are other communication lines that are critical, such as with the functional specialists but interaction with the Developers is the most important internal communication. The Development team are important to the Test team because:

- While code writing is progressing the Test team are developing test cases. This is their first major deliverable and to get it right the Test team need to talk to and learn from the developers.
- Developers can't learn everything about the system from the documentation they must learn by asking the functional specialists. If the Test team is close to the Development team they can learn through their discussions with the specialists. The alternative is for the Test team to ask the same questions later, irritating the specialists, probably wasting their time and worst of all perhaps getting a different answer.
- As the design and development progress the detail of the application changes; screens are redesigned, menus re-ordered and single batch programs split into several. These details affect the design of the test cases and the Test team can only find out such implementation detail by talking with the developers
- During system testing a lot of problems are going to be discovered. Some of these problems will be bugs, some execution errors, while others could be requirement problems. If there is no rapport between the teams, the developers may take exception to constant 'criticism' that the bugs represent.
- When system testing begins in earnest, it becomes the primary focus of the project and is probably on the critical path. The Test team should not be shown wanting when the spotlight falls on them. This requires the best possible preparation.

The relationship between the two teams is not balanced. The Development team can go through almost the whole initial development, all the way up to the beginning of system test without ever *needing* to talk to Testers to complete their initial implementation. This is not true for the for the Test team. They need to talk to Development. It is therefore the

© Copyright 1999 Perot Systems Corporation
All Rights Reserved

Test team who must work on the relationship, build the trust and learn because it is the Test team who needs the relationship most.

The Importance to the Development Team

The Development team has a different perspective on the relationship between the two teams. Developers have a great sense of ownership of their code and the quality of the product they produce. Developers also know, even if they have not expressed it, that testing can only verify the quality of their software once it is complete. The Test team cannot influence the quality of the code while it is still in pieces.

The Development team starts looking outward and becomes interested in talking to Test when the initial coding is complete. This is the time when defects start to get raised and generally when a single developer's code is first integrated with the rest of the system. Individual developers now must talk to and understand the Test team to clear up the defects identified.

The Balance of the Relationship

As the system development progresses it is clear that the balance changes between the two teams. Initially it is the Development team that has more information, better understanding of the domain and a fuller picture of the application architecture. It is at this time during the design and code phase that the Test team has most to gain by developing a strong relationship.

Once into the final stages of development and certainly after system testing starts, the balance shifts to the Development team being dependent upon Test for information on tests and defects.

The responsibility for a strong relationship is there for all members of the project team, whether they are developers or testers but the timing of the need means that the Test team must be more conscious of the relationship and work at it earlier.

Failure to Communicate

There is unlikely to be a catastrophic event that will affect a development project if the two teams do not talk. The effects are much more gradual and much harder to identify and influence. The effects include:

- Failure to identify inconsistencies as early as possible. If the test team has a different interpretation of the requirements than Development that should be dealt with when the tests and the application are being designed. It is expensive and wasteful to wait until test execution. The sharing of information does not compromise 'independent testing' because it only influences when the information is shared between the teams, without any prejudice about which interpretation is correct.
- A loss of morale within the Test team when their opinions and observations are ignored. The test design requires similar level of analysis detail as the system design done by the Development team. If these efforts are ignored or given superficial attention this will have an effect on tester morale.

© Copyright 1999 Perot Systems Corporation
All Rights Reserved

- A loss of morale amongst the Development team when they receive poor or irrelevant defects because the tester's or test's information is out of date.

Factors

Environment

Proximity of Teams

Most development projects suffer from the same problem: they are temporary: a need is identified, a system specified, an application built, a product delivered. During the project's, sometimes long, lifetime the team grows from a small number to a peak during the coding stage and then declines gradually until a core support group remains. Because of this tremendous change in size, projects often must make do with ad-hoc office arrangements.

There are many lessons to be learned from all the bad arrangements. The most important of these however is physical proximity. Test and Development teams work best together if there is a chance to develop informal relationships. If the teams share the same kitchen and restrooms and pass each other in the passageway 20 times a day you have the ideal situation.

However there are many cases when physical proximity is not possible. There are a number of ways of dealing with this separation:

- Regular communication using conference calling and video conferencing. This cannot always be a substitute for face-to-face meeting. When initially establishing a relationship, there needs to be face-to-face communication. If two teams are separated, someone will have to get on a plane. It is the only way. Once the relationship is established the conference call and video are much more effective.
- Pick a single representative who will act as ambassador between the two teams. Make sure this person is able to talk both languages, test and development. The ambassador should be familiar with the technology and understand the testing process.
- Specify and enforce a formal defect tracking system. When teams are separated it is easy for misunderstandings to become magnified. A formal tracking system should prevent these misunderstanding.

Workflow

The issue of efficient workflow is related to physical proximity. Most of the delays in processing work come not from doing the work itself, but from handing over the task to the next person. This applies equally to organizing meetings, reviewing documents, fixing defects and retesting code. Not having the teams located side by side immediately introduces these delays into all work, even if when the work gets started it is done efficiently.

© Copyright 1999 Perot Systems Corporation
All Rights Reserved

Procedure

Defect Tracking

Implementing a formal defect tracking system is an important part of ensuring effective communication between the teams. Without an established system that is understood and supported by both teams, system testing can quickly become uncontrolled and a drag on the project schedule.

Defect tracking often comes as part of another tool or is developed and maintained by the organization. For example defect tracking comes as part of the SQA Suite of test automation tools. However there are definite advantages to establishing a project wide system that covers the whole life cycle including change control, issue tracking and defects. These are variously called issue, problem or defect tracking systems but I prefer the term *Observation* Tracking.

Observation Tracking

When running tests, what the Test team is looking for are inconsistencies, that is behavior that is different than that expected by the test. Different does not automatically mean the code is bad. There could be many reasons:

- A problem with the test definition,
- A problem in the test execution,
- Inaccurate or ambiguous requirements,
- A coding problem,
- A design problem.

With an observation-based system, the Test team raise Observation Reports (OR). What the OR says is that there is an observed difference between what was expected and what the system does. It is then a *joint* effort to resolve the discrepancy. Code being corrected because of defect is only one outcome.

This is also why an integrated tracking system is essential, the OR could easily become a Change Request. If these are tracked separately then the same problem is tracked in two different places.

The use of Observation Reports will improve the relationship between the teams. The Test team is not out to find as many bugs as possible only to then dismiss them as Development team problems. It becomes a joint problem to *investigate* observations and find the real problem; be it requirements, test design or code.

Communicating with Observation Reports (OR's)

Whether they are called defects, problem reports, issues or observation the idea is the same: the tester writes down the problem, the developer fixes it. However there are many things that can go wrong:

© Copyright 1999 Perot Systems Corporation
All Rights Reserved

- Misunderstanding the problem and changing the wrong thing,
- Fixing the right thing but in the wrong way,
- Sending a real problem back as ‘functions as designed’,
- Partially fixing the problem,
- Fixing the problem but breaking something else,
- Not fixing the problem and breaking something else,
- Fixing the problem in one place but not all the occurrences,
- Raising the same issue twice with two different recommended solutions.

The OR process should be designed to minimize these failures by addressing both the way the OR’s are written and the teams’ commitment to the problem tracking system as a whole. There are many projects where the OR tracking is problematic because only lip service is paid to its use by some or all of the team.

To address the weaknesses in communication that can be present in an OR system try the following:

- Make sure that the team is committed to using the system.
- Establish a common language when talking about the system e.g. functional requirements, use cases, screens or components. It is important for the tester to be able to say *why* they think there is a problem and *where* it is.
- Make sure that everybody has a basic understanding of the technology e.g. what a dialog box is or what client/server is. Without a basic understanding it is very difficult to describe problems effectively.
- Make sure everyone knows how to write a problem down. Frequently descriptions or problems are incomplete, ambiguous or inaccurate.
- Include in each report how to recreate the problem: step by step instructions either directly in the defect report or by reference to a test case.
- Make sure that the date and name of the person making the entry is included in every comment made on the report including the original observation, the investigation and description of the fix.

In addition to writing problems down, the personal touch can make the difference between a ‘good’ OR and a ‘bad’ OR. If the developer is available, show the problem immediately. This should not be a substitute for writing the OR but it does mean that the report will not be returned as ‘not reproducible’. Having seen the problem in the Test environment, it cannot be just the tester ‘dreaming’.

Even if the developer is not immediately available, follow up the OR by going and talking to the developer while the problem is still fresh in your mind and the data required to demonstrate the problem is still available.

Classic Mistakes to Avoid with OR’s

As well as positive steps there are also things that the Test team should refrain from doing:

© Copyright 1999 Perot Systems Corporation
All Rights Reserved

- Keep emotive language out of Observation Reports. Developers are rightly proud of their creation and attacking it or cracking jokes is not going to win you any friends. I have witnessed several times developers shout and rage at a tester for writing the wrong thing. It is unnecessary and avoidable.
- Never make unsupported claims, always say why you think it is a defect. Requirements are always good for this and so are GUI style guides and Interface Control Documents.
- Observation reports that have a life of their own should be closed and opened anew. Frequently a defect report has gone through much investigation and perhaps many attempted fixes. After a while the current problem has no real bearing on the original issue and the title of the report is out of date.
- If too many individual problems are included in one observation report it becomes difficult to track which ones have been fixed and which need to be fixed again.
- Creating an individual observation reports for every small item is irritating for everyone who must go through all the workflow actions to record progress on the report.
- Don't let reports languish, deal with the important ones first not the easy ones. A report that is untouched is forgotten about eventually and someone will end up documenting the same problem in another report.

Configuration Control

When the Development team hands the application over for the start of system testing, it is 'finished', that is, all the functionality is present and no further new development is required. The remaining work is defect, fixing, performance tuning and documentation. All changes to the structure and logic of the application should be done only when documented in a defect report. Without enforcement of this control, the test team can quickly lose sight of what is tested, what is not tested and what needs to be retested. There are few things more frustrating for the test team than finding something that has changed for no apparent reason and now does not work.

Although configuration control of software during System Testing is common to most projects, it is clearly an important factor in establishing trust and maintaining effective communication between the teams. Proper use of Release Notes to identify the defects that have been fixed in each build reinforces the Observation Report communication and provides additional confidence that the application is under strict control.

Organizational

Team Structure

Most project teams are split into management, requirements, development and testing teams. This allows convenient reporting and allows each member of the team to draw on the skills and experience of the others. However, we have tried and succeeded with cross team groups, for example one person from each of the requirement, development and test teams. These cross-teams can work on a very narrow set of tasks very efficiently by

© Copyright 1999 Perot Systems Corporation
All Rights Reserved

working closely together. Tasks that seem to be most suitable for this kind of group are reports and batch processes.

Cross-teams create a sense of shared responsibility, breaking down the barriers between each of the groups. However it is important to ensure that each person of the group maintains strong ties with the rest of their team to ensure conformance to standards.

If the creation of formal groups is not appropriate, a less radical idea is to appoint an ambassador from each team. The role of the ambassador is to represent the team at the working level and to communicate what is happening back to his or her own team.

An ambassador for the Test team might attend design reviews, architecture or database discussions or sit in on the regular team meetings of the Development team. The Developer ambassador might be involved in the review of test conditions or the structure of user acceptance or beta testing.

Training

There are several ways that both informal and formal training can be used to encourage effective communication between the teams. Some of the most immediate are:

- Give the Test team training in how to write defects. The quality of most reports is low. Training should aim to improve consistency and the effectiveness of the OR's.
- Give the Test team grounding in the technology used in development. Testing is often seem as a good introduction to IT and it is not uncommon for testers have only limited technology experience.
- Teach the Development team about the testing process.
- Establish a corporate testing program to support the long term development of testing.

Project Size

As the size of a development project increases communication becomes much more of an issue, not just between Development and Test but across the whole project team. Eventually the problems of communication dominate all others and the single project needs to be split into multiple, smaller projects.

Project size is one of the main determinates on the structure and formality of communication between Development and Test[2]. On very large teams, there my just be too many people to exploit the benefits of informal communication between the teams. If the development team has 30 people on it, it will be difficult to ensure physical proximity to a Test team of a similar size. As the effectiveness of informal communication reduces formal communication replaces it: formal problem tracking, strict configuration control.

[2] Others factors include the application domain; generally commercial applications are developed under a different quality environment than safety critical applications.

© Copyright 1999 Perot Systems Corporation
All Rights Reserved

It is in large projects that communication is seen as an issue. However there are also issues at the other end of the project spectrum. Because defect reports are designed primarily to facilitate communication there should not be over enthusiastic use of tools or corporate procedures. Too much formalization can be just as counterproductive and wasteful as too little. If there are only a few developers and testers why use the corporate standard for defect tracking when a spreadsheet or a simple MS Access database would suffice? Even a paper-based system may be appropriate in some instances.

Summary

The effects of ignoring the relationship between Development and Test teams can be very damaging. A breakdown in communication can lead to much extra rework both in code fixes and testing. Without a sufficient understanding of the system, testing will leave many undetected bugs.

To conclude:

- Effective communication between the Test and Development teams is an integral part of any successful development project
- It is the Test team who benefit initially from the relationship and it is to their advantage to promote and develop the relationship as early as possible
- The best location for the to teams is side by side. If this is not possible use technology to facilitate communication
- A formal defect tracking system is essential on all but the smallest development projects. It enforces a consistent and professional communication
- Enforce proper configuration control so that during System Testing, the test team has a known configuration to test
- Organize testers and developers so that they have areas of shared responsibility, for example evaluating change requests.

© Copyright 1999 Perot Systems Corporation
All Rights Reserved

Test Engineering - A “Value-Add” Career Path!

Thomas L Wissink
Lockheed Martin Mission Systems
E-Mail: tom.wissink@lmco.com
Phone: 301-918-7448

Copyright, Lockheed Martin Mission Systems, 1999.

Test Engineering - A “Value-Add” Career Path

Agenda

- Introduction
  - Past views of the Test Profession
  - A new view of the Test Profession
- Test Engineering Skills
- Other Disciplines the Test Engineer should Know
- The Test Lead a Major Contributor
- The Steps in a Test Engineering Career Path
- Summary

Test Engineering - A "Value-Add" Career Path

Introduction

- **Purpose**
  - **Discuss Testing Engineering as a computer science discipline**
  - **Explain how Testing Engineering is a "value-add" career path**
- **Definitions**
  - ***Testing***
    - » **William Hetzel 1973,** testing is the process of establishing confidence that a program or system does what is supposed to
    - » **IEEE 1983,** testing is the process of exercising or evaluating a system by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results
    - » **Dr. Boris Beizer 1995,** testing is defined as the act of designing, debugging and executing tests

Test Engineering - A "Value-Add" Career Path

Introduction (cont'd)

- **Definitions**
  - ***Test Engineering,***
    - » a career path or computer science discipline that focuses on planning and implementing testing throughout the system or software development lifecycle. It starts with the requirements phases and continues into the support/maintenance or production phase.
  - ***"Value-Add"***
    - » the test engineer provides input, ideas, suggestions and results that:
      - contribute to a timely product/system
      - improves the product/system
      - conforms to the company vision/mission

Test Engineering - A "Value-Add" Career Path

Past views of the Test Profession

- **My 26 years in and around testing as an example**
  - **1970's**
    - » **Diagnostics - HW/SW testing**
      - Diagnostodian
      - Our new name - Diagnostician, with white lab coats
  - **Early 1980's**
    - » **Validation Team**
      - Validators (Terminators), Verifiers, and Testers
    - » **Test staffing realization**
      - training gound
      - holding area
      - viewed as not a place to stay

Test Engineering - A "Value-Add" Career Path

Past views of the Test Profession (cont'd)

- **Testing career example continued**
  - **Late 1980's**
    - » **System Test Team Manager**
      - Very fulfuling
      - Plenty of support
      - Told this was not a place to stay
        - – Work in other areas (i.e. SW, SE)

Test Engineering - A "Value-Add" Career Path

A new view of the Test Profession

- **Testing career example continued**
  - **The mid-1990's**
    - » **After some time in others disciplines**
      - returned to Testing
    - » **Became the Integration and Test Architect for a large and very complex satellite control system**
      - Viewed by the management and technical team as "value-add"
      - For me - Testing as a career began to take shape
    - » **Over the same 20 plus years:**
      - Engineers became System Engineers
      - Programmers became Software Engineers
      - Testers stayed Testers

Test Engineering - A "Value-Add" Career Path

A new view of the Test Profession (cont'd)

- **Testing career example continued**
  - **The late-1990's**
    - » **Another realization**
      - All the advice was correct - they said try other areas
      - For the wrong reason - they said it was not a place to stay
        - – What it did was help make me a better test engineer and provided a solid foundation for being "value-add"
      - Test Engineering can and should be a distinct computer science discipline with it's own career path
- **The remaining part of this presentation may:**
  - – seem like common sense - it is!
  - – seem unnecessary - it isn't!
  - – seem like company politics - it is, everything has it's politics!

Test Engineering - A "Value-Add" Career Path

Test Engineering Skills

- **My first recommended skill probably isn't a skill at all**
  - **A good sense of humor**
    - » That part of testing that is a the back end of the schedule will most often be under schedule pressure
      - you will go insane without this skill
- **Recommended skills - not necessarily in textbooks or in "How to write a resume" book**
  - Many good books on the market today that talk recommended technical test methods/skills.
  - This list should be viewed as a basis, fundamental or starter set of skills for a test engineer

Test Engineering - A "Value-Add" Career Path

Test Engineering Skills (cont'd)

- **Recommended Fundamental Skills**
  - ***Analytical* - two definitions or aspects**
    - » Breakdown a program or system into component parts and define tests for each part - through and specific
    - » Search out the cause of problems - dogged determination to understand exactly why something occurred
    - » It might be a gift or knack but can be developed and improved
  - ***Detailed***
    - » Pulling out or Putting in all that is need to ensure completeness and correctness
    - » Sufficiently document what happens - during and after each test phase
    - » Can be taught, developed and improved

Test Engineering - A "Value-Add" Career Path

Test Engineering Skills (cont'd)

- **Recommended Fundamental Skills (cont'd)**
  - ***Tenacious* - this is the one that really explains the friction**
    - » Keep testing over and over with only minor changes in data, structure, and/or environment
      - To get it to run right
      - To find that error you saw one time
    - » Can be taught, developed and learned
  - ***Knowledgeable***
    - » Knows and understands the product or system
      - requirements, processes/methodologies, cm, labs, etc.
      - pervasive project awareness and involvement
    - » Can be taught, developed and learned

Test Engineering - A "Value-Add" Career Path

Test Engineering Skills (cont'd)

- **Recommended Fundamental Skills (cont'd)**
  - ***Teachable***
    - » Always learning
      - no matter what the source - even engineers and programmers can teach a tester new things
    - » Open to new ways of doing the testing job
      - New tools, new methods/processes, new ideas
  - ***Entreprenurial* - defined as organizing, managing and assuming the risk of a business or enterprise**
    - » Knows why the company is in business - Tests to the Vision
    - » Knows Test is the last line of defense - must assume responsibility for reducing the risk of errors in the product or system

Test Engineering - A "Value-Add" Career Path

Other Disciplines the Test Engineer should Know

- **The goal is to gain a working understanding**
  - Training & Performing or less desirable just Training
  - technical and administrative aspects
- **Here are a few examples - there are many more possibilities**
  - **System Engineer**
    - » write "testable" requirements
    - » support architecture development and/or analysis
  - **Software Development**
    - » moderate a design and/or code inspection
    - » generate and unit test a piece of software
  - **Software Configuration Management**
    - » support the build, delivery and installation processes

Test Engineering - A "Value-Add" Career Path

The Test Lead as a Major Contributor

- **A Test Lead, Manager or Supervision**
  - Viewed as a major contributor
  - Less technical and more administrative
    - » cost, schedule, staffing, training, etc.
- **Test Engineers at any level in the organization can be viewed as major contributors**
  - ***awards for savings, improvements, heroic efforts, etc.***
- **Having all the skills we've talked about and performing them effectively, makes us** *"value-add"* **and** *major contributors!*
- **How long does it take?**
  - Varies, but at a minimum several years

Test Engineering - A "Value-Add" Career Path

The Steps in a Test Engineering Career Path

- **I can't dictate to your company how to develop it's people**
  - I can recommend a set of steps that will result in a career path for test engineers and it will be "value-add"
- **The recommended Steps in a Test Engineering Career Path**
  - A computer science degree (at least as a minor)
  - Work assignment(s) in testing
  - Test courses every other year (internal & external)
  - Other Discipline courses on the alternate years
  - Work assignment(s) in other discipline(s)
  - Test Lead/Manager assignment
  - Get certified (internal or external)
  - Work assignment(s) in other discipline(s) or a Staff position

Test Engineering - A "Value-Add" Career Path

The Steps in a Test Engineering Career Path (cont'd)

- **The recommended Steps in a Test Engineering Career Path (cont'd)**
  - **Next steps depends on personal goals and abilities**
    - » If you earn the right or are given a chance at upper/executive management - do it!
    - » If you return to or stay technical
      - get published and/or work to improve the technology and career paths within your company
- **Some companies do a good job of helping achieve personal goals - others don't (not intended as a slam)**
  - *In either case - Take ownership of your career*
- **A final comment about your career path - *Be Flexible!***
  - Perform undesirable assignment very well, train replacement & move-on
  - Management will appreciate it and work harder to assist your career

Test Engineering - A "Value-Add" Career Path

Summary

- **A "Value-Add" Test Engineer**
  - Knows and has performed several disciplines
  - Has improved the process (in testing and other areas)
  - Implements a robust testing program throughout the lifecycle.
- **This presentation is one more nuance in the process of improving the Test Engineering profession**
- **Facing the Future in the next Millennium**
  - Test Engineering is poised to become a mainstream career path within the computer industry
  - Dr. Boris Beizer says - *"Losing the test technology game is losing the software quality game which means losing the software game…"*
- **Let's go out and make Test Engineering all it can be!**

Test Engineering - A "Value-Add" Career Path!

Thomas L. Wissink

Lockheed Martin Mission Systems
Phone: 301-918-7448
Fax: 301-918-7474
e-mail: tom.wissink@lmco.com

Copyright, Lockheed Martin Mission Systems, 1999.

Abstract

In the not too distant past software testing was viewed as an unattractive career path and test personnel had few training or career enhancement options. In the past test organizations have usually been insufficiently and inappropriately staffed. The staffing that was provided, came from assigning new personnel to test for training purposes or some of the project less desirable personnel were given the "opportunity" to test. Those who enjoyed testing and wanted it as a career usually had to make up for these situations by working harder and longer to overcome this "stereotype" view of testing.

However, the environment today provides a more positive opportunity for the test engineer. We are now frequently viewed as important team members and as contributors to the overall quality and timeliness of software products. The recognition that testers must be involved early and throughout a product or project lifecycle, the plethora of software testing companies, test products, test conferences and even test engineering certification are all examples of the growth of the testing discipline.

This paper will provide insights into the following topics: 1) the past views of the testing profession, 2) the skills that make a successful test engineer, 3) the other computer industry disciplines that a test engineer should understand, 4) how the "Test Lead" can be a major project/company contributor, and 5) the steps that should be in a test engineering career path.

1. Introduction

This paper is written primarily for the test engineer, but provides insight for anyone with an interest in the career aspects of testing. The paper has two main purposes: 1) to present the notion that Test Engineering is a unique computer science discipline, with a defined career path, and 2) to explain that test engineering can be a "value-add" career path for the person and his or her company.

Today some companies have a specific career path for test engineers and there are formal certifications available for testers. This progress is great, but the idea of a test career path is not pervasive in the computer science industry and not many universities have a election of test courses or a degree in test engineering.

Definitions

Here are a few simple definitions.

Testing, has been defined very well by many authors – here are a few good examples from the past and present:

1) from William Hetzel in 1973, testing is the process of establishing confidence that a program or system does what it is supposed to,
2) from Glen Myers in 1979, testing is the process of executing a program or system with the intent of finding errors,
3) from the IEEE standard glossary in 1983, testing is the process of exercising or evaluating a system by manual or automated means to verify that it satisfies specified requirements or to identify differences between expected and actual results, and
4) from Dr. Boris Beizer in his 1995 book "Black-Box Testing", testing is defined as the act of designing, debugging, and executing tests.

Value-Add means that the test engineer provides input, ideas, suggestions and results that: 1) contributes to a timely product or system delivery, 2) improves the product or system quality and 3) conforms to the company vision and mission.

Test Engineering is the term used to describe a career path or computer science discipline that focuses on planning and implementing testing throughout the system or software development lifecycle. It starts with the requirements phase and continues into the support/maintenance or production phase.

Past Views of the Test Profession

I will use my personal 26-year career in the computer industry as an illustration. Those of us who have had the "career opportunity" to test software and software based systems have, over the years, been called by many different titles. These titles have included: diagnostodian, verifier, checker, validater and tester. Let me explain the source of some of these "titles".

The term diagnostics refers to computer software that's purpose is to: 1) test or checkout new hardware, software and firmware as it comes off the assembly line, or 2) troubleshoot hardware or system problems. When I wrote diagnostics for a small computer firm in the 1970's I was sometimes referred to as a diagnostodian. This title was used half in jest but it was used. We diagnostodians coined our own title, "Diagnostician" and we even wore white lab coats to emphasis our point. This was all in fun but the "Engineers" were still the folks who made all the important technical decisions.

In the early 1980's I was part of a "Validation Team" where we were either called verifiers, validaters or testers (Note: If the movie *Terminator* had been around then I am sure we would have been called Terminators – because that is what we usually did to the software under test). During this particular assignment I began to realize that the "testers" were not the "cream of the crop" when it came to the staff performing on a project. The management team would say things like "it is good to do a little time in test so you understand that part of the development process but you should not stay there". Testing was not viewed as a long-term career path for those who wanted to progress up the corporate ladder or wanted to become a technical leader.

In late 1980's I was the manager of a System Test Team. The management job was much more visible, very fulfilling and I received all the help and support that was needed. I was told however, that to really progress in the company I needed to be involved in other disciplines like Systems Engineering and Software Development. Again, testing was not a career that you could stay in for very long and have it continue to progress.

A New View of the Test Profession

In the mid 1990's I finally decided that after being a line manger, a middle manager, a software developer, and systems engineer, I would return to "testing". I became the Chief Integration and Test Architect for a very large and complex computer system used to control a constellation of Air Force satellites. After several months I was viewed as a critical people resource to project management and my opinions were sought out when new or enhanced features were being considered for the system.

It was at this point in time for me that the whole idea of a testing career really began to take shape. During the same 20 plus year period of time the other computer science disciplines were evolving - engineers turned into Systems Engineers, programmers turned into Software Engineers but testers stayed testers.

It finally dawned on me in the late 1990's that much of the advice I had been given in the past was not totally incorrect or untrue. It was however, given for the wrong reasons. In the remainder of this paper I will pass along lessons I have learned concerning a career in test. It will include some recommendations that can make the "tester" a key "value-add" person to any company and earn him or her, my preferred title for this profession, "Test Engineer". Some of my recommendations may seem like common sense – they are! Some may seem unnecessary – they are not! Still others may seem to be more about company politics – they are – everything has its politics!

2. Test Engineer Skills

The first skill is a non-technical skill, if it can be considered a skill at all – a good sense of humor. A positive attitude in any testing situation will serve you very well throughout your test career. If you have not already encountered it, testing always takes the brunt of the schedule impacts. Since part of testing is at the back end of the schedule you will more than likely get pressure because of other project problems. So having a good sense of humor will at least keep you sane.

There are several good testing books on the market today that list and define the many phases of testing and the specific technical methods and/or skills that are needed to perform a complete test program. These all tend to be very specific and they need to be. This paper however, is intended to identify a basic set of fundamental skills for test engineering.

You may not find these skills in one of those textbooks or in a "How to Write a Resume" book. But they are skills that will make you a very successful test engineer. The skills are not listed in any particular order and not every Test Engineer will have or is required to have all these skills mastered – but they should constantly work to improve them.

1. *Analytical* – there are two important definitions for analytical. The first is being able to break down a system or a software program into its component parts. Then being able to define a set of tests that will ensure a quality product is the result. The second definition of analytical, and the most important, is that a test engineer is always looking for and searching out the cause of a problem. Working (not just wanting) to understand why a function works the way it does, why an error occurred under given circumstances or why something runs one way on one device and another way on another device. Getting to concrete reasons and rationale is the goal. This skill (especially the second definition) appears to be a knack or a gift, but it can be developed and improved.

2. *Detailed* – able to determine the steps that are needed, the items to be queried or the paths to be executed. Fleshing out the detail is crucial, it helps with determining the testability of a requirement, the completeness of an inspection or

the robustness of a test procedure. Putting in and pulling out the detail is another tool in the test engineer's tool kit of skills (Note: You need to be able to document this detail also). This skill can be taught, developed and improved.

3. *Tenacious* – this skill has to do with the ability to keep working on something over and over until it works exactly right. That may mean running the same test case or procedure with only minor differences in data or structure many, many times to ensure the quality of test and the quality of the product. You may also be running the test over and over because you saw something that did not look right one time and your looking for that case again. Tenacity also aids the test engineer when a software engineer declares the failure is "not repeatable". This skill can also be taught, developed and improved.

4. *Knowledgeable* – knows the system or software product that is being tested. A deep understanding of the requirements, the architecture, the design and development methodologies, the method and process of software configuration management, the structure and integration plan of the labs that are being used, etc. Being aware and interested in all aspects of a project is vital. This skill has to do with paying attention, attending meetings, reviewing documents and just plain being interested in all that is going on within a given project – it can be taught, developed and improved.

5. *Teachable* – You must be able to be taught and be able to learn new things everyday – no matter who you learn it from. Excuse me if you are an engineer or programmer reading this paper - but yes, even engineers and programmers can teach testers a new thing or two about test. Teachable also means learning new ways of doing things. Because a given type of test worked in the past does not mean that it will work exactly the same in the future. Be open to new ideas, new tools and new processes.

6. *Entrepreneuria*l – Webster defines entrepreneur as "on who organizes, manages, and assumes the risk of a business or enterprise". This skill has two aspects – The first is the knowledge of why the company is in business. You should look for ways testing can contribute to and promote the vision and direction of the company. The other aspect to understand is that test engineers are the last line of defense for delivering a quality product or system. Test Engineers should take ownership to reduce the risk of delivering errors to the customer.

3. Other Disciplines the Test Engineer should Know

As noted in the Introduction of this paper, often well meaning people gave advice to move-out of testing and try other disciplines. The rationale was that testing was not a place to stay. In hindsight, this was good advice – but bad rationale. The correct rationale is that being involved in other disciplines provides an opportunity to really understand that discipline and how testing is involved and how it can be improved. Therefore, every test engineer should develop a working understanding of other major computer science disciplines. Systems engineering and software development are probably most important. However, configuration management, the specialty engineering fields (i.e. security, reliability, performance, safety, logistics, etc.) and others should also be investigated.

You gain a working understanding by one of two methods. First, and most important, you should have training in and have worked in another discipline. The second choice is to at least have training in the discipline and that training should include a workshop type environment (i.e. you did it at least once in the class setting).

Here are a few sample assignments that could be performed in another discipline: 1) be a moderator for the software and/or requirements inspection process, 2) be assigned and generate several small software programs, classes or components with unit test responsibility, 3) be involved in the software configuration management, build, delivery and installation processes, and/or 4) work in lab support as a system administrator.

Two final points about being involvement in another discipline. One, be involved in as many aspects of that discipline as possible, the technical aspects and the administrative aspects (i.e. costing, scheduling, etc.). Two – an Army phrase seems appropriate here – "Be the best you can be" and be easy to work with. It is virtually guaranteed, if you are viewed as easy to work with and helpful when you support another discipline, you will receive all the support you need, when again as a test engineer, you need something from that discipline.

4. The "Test Lead" as a Major Contributor

By the time a Test Engineer becomes the leader or manager of a test team, they need to have additional skills in their skill's toolkit. These skills tend to be less technical and more administrative and managerial. They include an understanding of the company's proposal or new business process, the cost and scheduling process and the people management process. Developing these skills definitely takes education and experience.

It is usually not until this point in a career that upper management views a test engineer as being "value-add" to the company or project. However, a test engineer at any position in the company can be "value-add". Many test engineers have received special company and

customer recognition for their contributions during all phases of a project. The test engineers and the lead test engineer can and should be major contributors on any project.

Having all the technical and administrative skills that have been discussed above and being able to productively use them will, by definition, make you "value-add". Management, customers, and other team members will appreciate and listen to your input, suggestions, and recommendations. Your input should span the project lifecycle.

How long might it take for someone to become a contributing test lead? It will more than likely take several years. There are people who can progress through the different steps in a test career very rapidly and do a great job. Then there those who will take longer. Longer is not bad – producing and contributing throughout the career path is the important factor.

5. Steps in a Test Engineers Career Path

Obviously, it would be dangerous to try to dictate how a particular company will develop its people. But here is a suggested set of steps that are important for the development of a "value-add" test engineer. The following list is a minimum set of ordered steps that should be included in a test engineering career path:

1. A computer science degree is desirable.
2. Work assignment(s) in testing.
3. Test courses (either via your own company or one of the many testing companies that are now available).
   - As with any computer science field, education is an on-going activity. Plan on a test course at least every other year. The alternate year should be a course in some other discipline or in leadership/management.
4. Work assignment(s) in another discipline(s).
5. A work assignment back in testing, possibly as a technical lead or manager
6. Get certified as a Test Engineer (could be done earlier with the right experiences)
   - Inside or outside your own company – there are several companies that have certifications and/or specialist programs.
7. A work assignment in another discipline or on staff to a Project Manager.
8. The next set of steps depends on your own personal goals – do you want to try an upper management/executive path or a technical path.
   - If you earn the right or are given the chance at an upper management/executive position, do it.
   - If you decide to stay technical or return to a technical position after a management assignment, work to get published and/or help define your company's technical direction.

In today's market place you need to take ownership of your own career. Some companies are very good about helping you develop along the path you desire while others seem only

interested in the bottom line. Without judging any particular company or line of business, it is suggested, that if your goal is to be a test engineer, work at it with a specific plan in mind.

A final thought about your career plan. While it may seem contradictory with the previous paragraph, I suggest that you be flexible. There will be times when you are asked or directed to work in an area that you do not think fits your plan. My suggestion is that you work at it for a while, do well, look for and develop your replacement, and work to move on to another assignment as soon as possible. This approach gives you an opportunity to learn another discipline and will in most cases, ingratiate you to your management team.

6. Summary

To be a "Value-Add" Test Engineer means you have a working understanding of the entire process of getting a product or system designed, developed and delivered. It means you participate in the process and provide positive input to those processes. And finally, and most importantly, it means you can implement a testing program throughout the product or system development and support lifecycle.

Today, there are many people and companies working to improve the test engineering profession. This paper is another nuance of that effort and has been focused on providing career counseling to the test engineer.

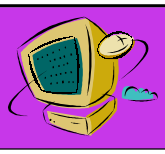
The year 2000 computer problem has done many things for the computer industry – not all of it good and not all bad. One of the good things is that the testing profession has been brought to forefront. Facing the future and looking toward the next millennium, the prospect exists for test engineering to become a mainstream "value-add" career path in the computer industry.

Creating a Testing Culture
Sharing Best Practices

Keith Stobie
QA Process & Test Architect
Keith.Stobie@BEAsys.com

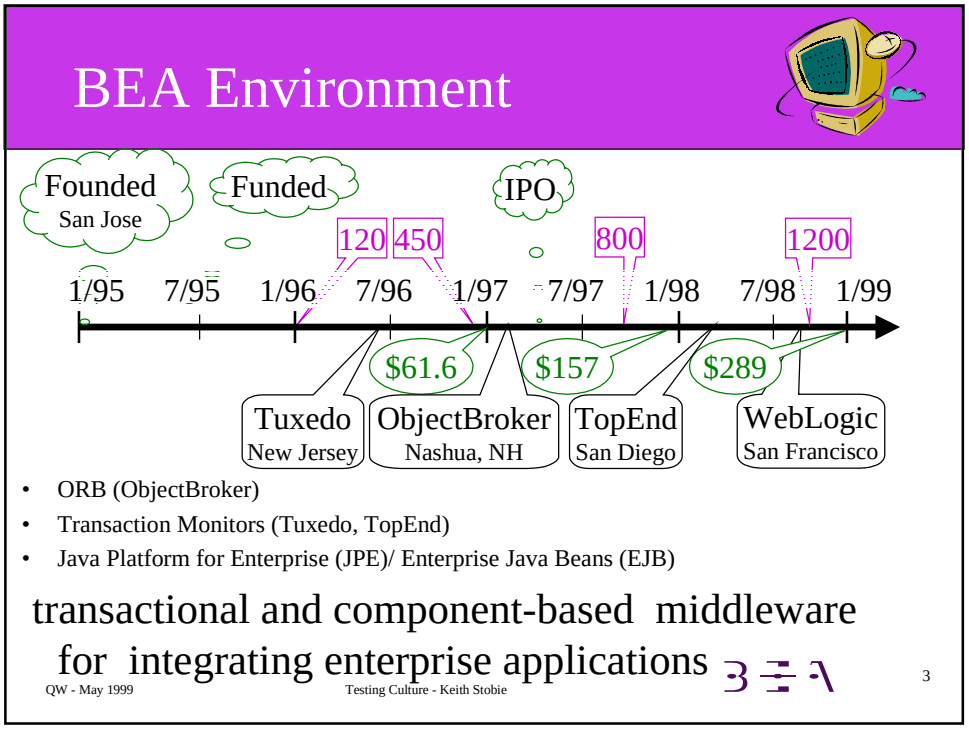
Acknowledgements:
Vern Peets, Michael Peters, Art Rofrano

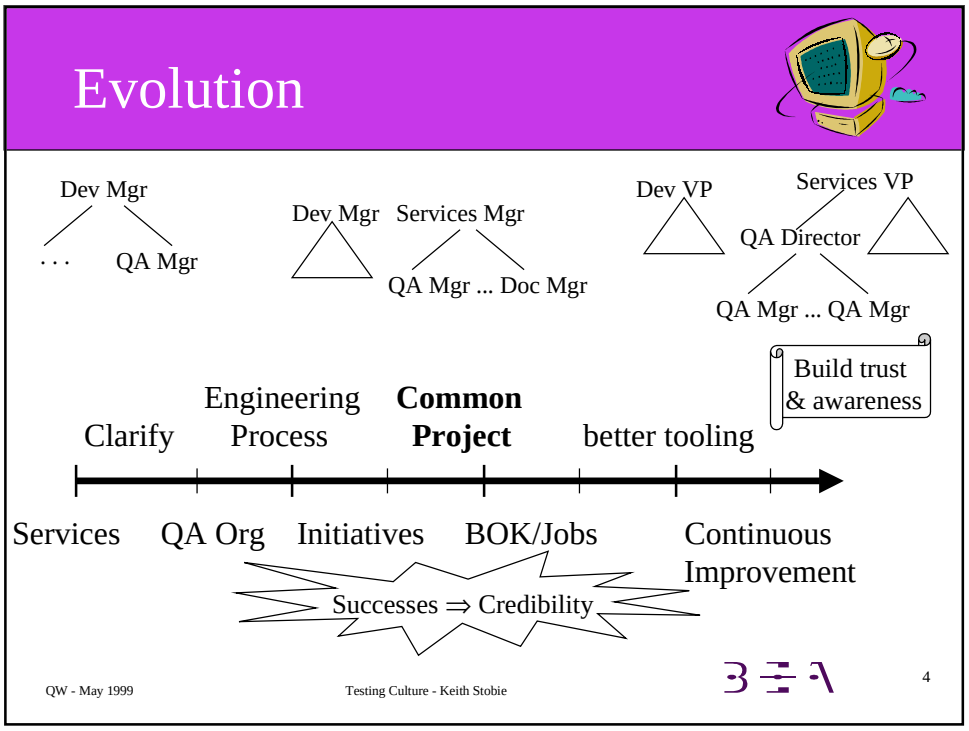
Culture



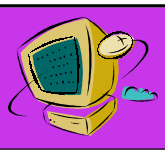
set of shared attitudes, values, goals, and practices that characterizes a company

- Define them: create Body of Knowledge
- Reward them: job advancement
- Teach them: define training curriculum
- Share them: define common patterns
- Measure them: development, execution, defects, etc.
- Tool them: provide tools to match practices


BEA Environment
Founded
San Jose
Funded
IPO
120
450
800
1200
1/95
7/95
1/96
7/96
1/97
7/97
1/98
7/98
1/99
$61.6
$157
$289
Tuxedo
New Jersey
ObjectBroker
Nashua, NH
TopEnd
San Diego
WebLogic
San Francisco
• ORB (ObjectBroker)
• Transaction Monitors (Tuxedo, TopEnd)
• Java Platform for Enterprise (JPE)/ Enterprise Java Beans (EJB)
transactional and component-based middleware
for integrating enterprise applications
QW - May 1999
Testing Culture - Keith Stobie
3

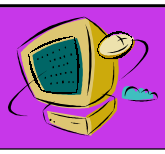

Evolution
Dev Mgr
. . .
QA Mgr
Dev Mgr
Services Mgr
QA Mgr ... Doc Mgr
Dev VP
Services VP
QA Director
QA Mgr ... QA Mgr
Build trust
& awareness
Clarify
Engineering
Process
Common
Project
better tooling
Services
QA Org
Initiatives
BOK/Jobs
Continuous
Improvement
Successes ⇒ Credibility
QW - May 1999
Testing Culture - Keith Stobie
4

Quality Initiatives



- mini-projects (initiatives) for improvement
- project lead & management sponsor additional initiative members
- Early areas identified included
  - job skills & training
  - test processes
  - metrics
  - automation
  - best practices

Body of Knowledge (BOK)



- Software Testing Labs (early web)
- ASQ's Certified Software Quality Engineer

Knowledge of

- Company
- Process
- Testing
- Technical (software areas)

BOK outline

- 1. BEA specific
- 2. Professionalism
- 3. Technical Knowledge
- 4. Software Quality
- 5. Test Planning
- 6. Test Design and Execution
- 7. Test Project Management
- 8. Documentation
- 9. Formal Inspection
- 10. Software Configuration Management
- 11. Software Project Dynamics & Software Development

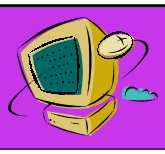
QW - May 1999 Testing Culture - Keith Stobie 7

BEA specific

- 1.1. Mission, Strategy and Values
  - What makes BEA special, where we've been, & where we're going.
  - History, Organization structure, Values, Goals, Competition
- 1.2. BEA Quality System
- 1.3. Standard Processes: How projects work in Engineering.
  - Engineering Process
    - Deliverables (Doc, Dev, QA, others)
    - Standards (coding, portability, ...)
    - Tools (CVS, ClearQuality, ...)
    - Procedures (Reviews, Doc Versioning, ...)
  - QA/Test Process
    - Tools (MegaTest, ARTTS, Test FrameWork, Vasir ...)
    - Patterns, Templates
    - Tracking (Test Dev, Test Run)

QW - May 1999 Testing Culture - Keith Stobie 8

Technical Knowledge



3.1. Software Product Architecture

(structure and design of enterprise software product components.)

Performance, Scalability; Components & frameworks; Applications; Documentation & packaging

3.2. Hardware Configuration

(architecture, maintenance, setup and testing of Windows and Unix)

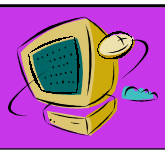
Memory; Mass Storage; O/S setup

3.3. Technology Market and Trends

Popular products & companies; Emerging technologies & application trends.

QW - May 1999 Testing Culture - Keith Stobie 9

Technology Risks



Architecture, development, maintenance, setup, tools, *risks and testing methods* associated with each technology.

3.4. Networks & Data comm (TCP/IP, SNA, OSI)

3.5. Client/Server (HTTP, ATMI, IIOP, JRMP)

3.6. Databases (/D, Oracle, DB2, Sybase, Informix)

3.7. GUIs (MFC, Java)

3.8. Component Models (CORBA, COM, EJB)

3.10. Installation: (InstallShield, Unix, Web)

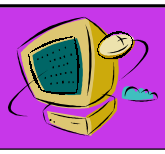
3.15. Windows/NT **Platform**

3.16. Unix **Platforms** (Testing of Unix applications.)

3.17. Other **Platforms** (AS400, MVS, VMS)

QW - May 1999 Testing Culture - Keith Stobie 10

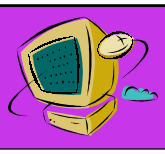
Test Case Design



- clues (specs, customer defect reports)
- requirements
- cases
- Equivalence Class partitioning
- Usage scenarios
- Fault Insertion / Design for Testability
- Authoring, Reviewing, Baselining
  Test Design Specifications (TDS)

QW - May 1999 Testing Culture - Keith Stobie 11

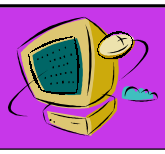
Test Techniques



- Boundary Value Analysis
- Syntax Testing
- Cause Effect diagrams
- State machines
- Orthogonal Arrays, Combinatorial Design
- Testing the accuracy of customer deliverables:
  user documentation, marketing and training materials
- Traceability mechanisms
  - system verification diagrams
  - requirements tracing
  - system messages
- Evaluating requirements for:
  consistency, completeness, and testability

QW - May 1999 Testing Culture - Keith Stobie 12

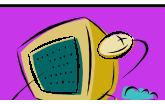
Test BOK Map



	8	9	10	11	12
1. BEA Specific					
1.1 Mission, Strategy, & Values	.	.	+	+	*
1.2. BEA Quality System		.	.	+	*
1.3. Standard Process		.	+	+	*
3. Technical Knowledge					
3.1. Software Product Architecture	.	.	+	+	+
3.2. Hardware Configuration		.	.	+	+
3.3. Technology Market and Trends				.	.
6. Test Design and Execution					
6.1. Test Case Design		.	.	+	*
6.2. Test Techniques			.	+	+

QW - May 1999 Testing Culture - Keith Stobie 13

BOK to Job Ladder Map



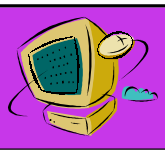
. = Familiarity - = Varies

+ = Proven Skill ■ = Creative Expertise

	Associate Tech 8	Associate Eng. 9	Software Eng. 10	Software Sr. 11	Staff Eng. 12	Staff Sr. 13
1. BEA Specific						
1.1. Mission, Strategy and Values	.	.	+	+	■	■
1.2. BEA Quality System		.	.	+	■	■
1.3. Standard Process		.	+	+	■	■
3. Technical Knowledge						
3.1. Software Product Architecture	.	.	+	+	+	■
3.2. Hardware Configuration		.	.	+	+	+
3.3. Technology Market and Trends				.	.	+
3.4. Networks and Datacomm		-	-	-	-	.
3.5. Client/Server		-	-	.	.	+
3.6. Databases		-	-	-	-	.
3.7. GUIs		-	-	-	-	.
3.8. Component Models		-	-	-	-	.
6. Test Design and Execution						
6.1. Test Case Design		.	.	+	■	■
6.2. Test Techniques			.	+	+	■
6.3. Test Tools and Automation		.	.	+	■	■
6.4. Test Execution and Reporting	.	.	+	+	■	■
6.5. Bug Investigation and Reporting	.	+	+	+	■	■

QW - May 1999 Testing Culture - Keith Stobie 14

Training - Basics



Basic Testing Process

- Plan
- Design
- Implement
- Execute
- Report

Roles

- Planner : Creates Test Plans
- Designer : Designs Test Cases
- Implementer: Implements & automates Test Cases
- Executor: Executes test cases

Basic Test Techniques

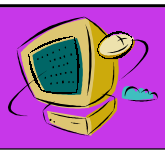
- Equivalence classes
- Boundary Values

Basic Courses

- Software Testing 101 & Black Box Software Testing
- System Testing Techniques
- Testing in the Real World
- Systematic Software Testing

QW - May 1999 Testing Culture - Keith Stobie 15

Training - Advanced



The World of Testing: 5-day
covers 38 different test techniques

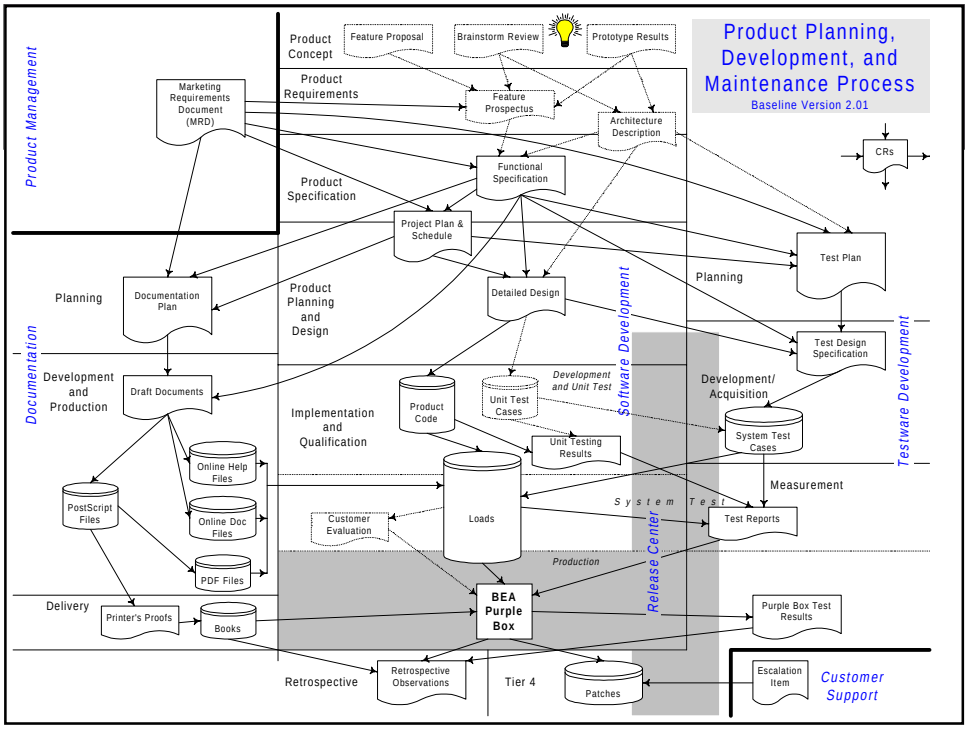
Advanced Test Design: 2-day
12 pragmatic techniques and
36 tools for model construction/test generation

Software Testing Tools 2-day
roadmap for automating the testing process & overviews (with demos) of leading testing tools

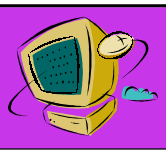
Test management: 2-day
What Every Test Manager Needs to Know

Software Inspections

QW - May 1999 Testing Culture - Keith Stobie 16



Common Test Designs

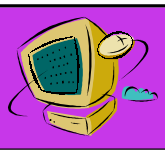


- Brian Marick's Catalogs
- Cem Kaner's Test Matrices
- Robert Binder's Test Patterns

- Installation Test Design
- Host & Port test idiom

QW - May 1999 Testing Culture - Keith Stobie 18

Test Catalogs

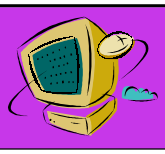


- "a catalog is a collection of test requirements, indexed by variables and operations"
- simple example
  counts: *The number of iterations, number of data items, etc.*

	Specification only.
-1	IN
0	IN/OUT
1	IN
>1	IN/OUT
maximum possible value	IN/OUT
one larger than maximum possible value	IN

QW - May 1999 Testing Culture - Keith Stobie 19

Test Matrices

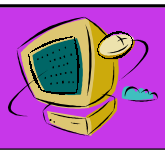


- "matrices for whenever you can specify multiple tests to be done on one class of object"
- simple example
  Test Matrix for a Numeric Input Field

		Nothing	Valid value	at Lower Bound (LB) of value	at Upper Bound (UB) of value	At LB of value-1	At UB of value+1	Outside of LB of value	Outside of UB of value	0	Negative	Empty Field	Non-digits

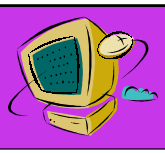
QW - May 1999 Testing Culture - Keith Stobie 20

Test Patterns



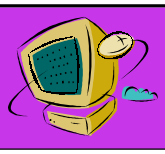
- "Test patterns simply frame and focus the issues that must be resolved to accomplish any kind of testing."
- simple example
  Category-Partition:
  Design test suite based on input/output analysis.

Installation Testing



- File Handling
  pathnames (from test catalog) :
  - empty name (legal? Default?)
  - minimum name
  - maximum name
  - alphabet:
  - network name
  - alternate filesystem/drive
- System Settings
  configuration
  - minimum
  - typical
  - maximum
- Error/Exception Handling
- Compatibility
- Usability
- Special Capability Issues
- AutoPlay
- Custom Install
  - all defaults
  - no defaults
  - other non-defaults
- Uninstall
  - fresh install
  - new (release) install
  - re-install (over same version)
  - parallel install
- Partial/Add Install
  - single component at a time
  - all components
- Upgrades
- On-line Registration

Host test idiom



<host-name> ::= <dns-name> | <dotted-address>

Test Requirements

1) address length:

Reqt#	Value	Reason
2.	7	min legal: #.#.#.#
3.	15	max legal (255.255.255.255)

4) octet value

Reqt#	Value	Reason
2.	0	min
3.	128	middle
4.	255	max

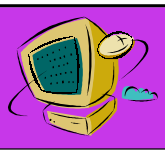
2) octet length
3) octet characters
5) name length
6) dns components
7) name characters

Test Values

Reqts	test case	value
1.2, 4.2	Simple short addr:	"0.0.0.0"
1.3, 4.3, 4.4	Simple safe (15) longest name:	"000.127.128.255"

QW - May 1999 Testing Culture - Keith Stobie 23

Interview Questions: API

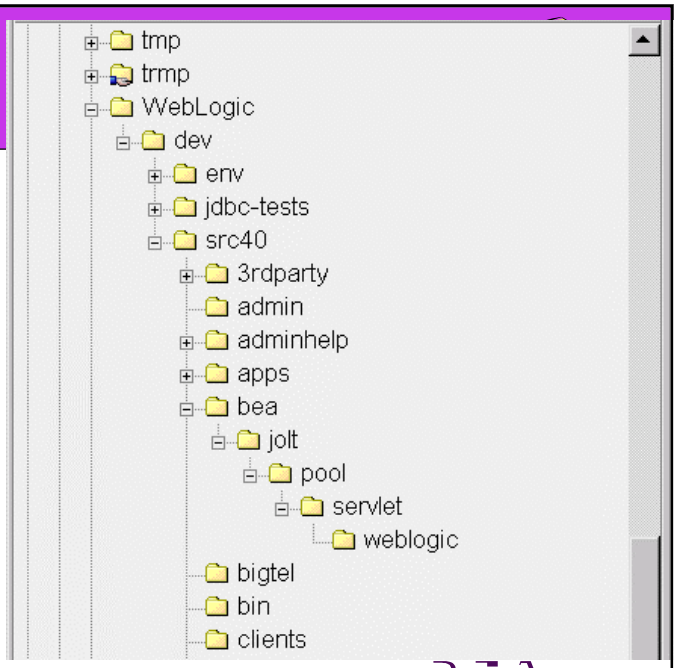


- Code method to merge 2 sorted linked lists:
  What are the requirements?
  Can they outline/design/code on the fly?
- How would you test a routine merging two sorted linked lists?
  Boundaries? Stress? Illegal inputs?

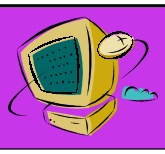
QW - May 1999 Testing Culture - Keith Stobie 24

Sample

- How would you test a tree control based widget?

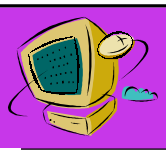


Metrics

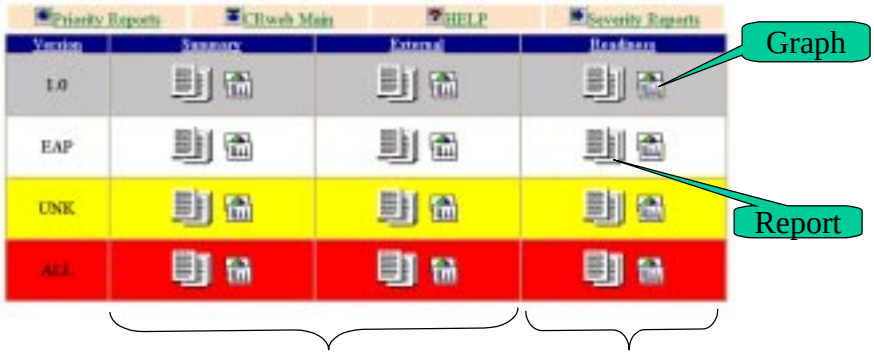


- Defects (from Clarify database)
  [CR = Change Request = Defect/Bug report]
- Code (from CVS repository) metrics (from RSM)
- Web based standard reports
- Drill downs

Product Birds Eye View



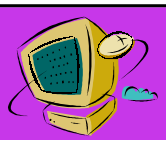
Product B CR Summaries



CRs by version FOUND (no replicates)

CRs by version FIXED (includes Replicates)

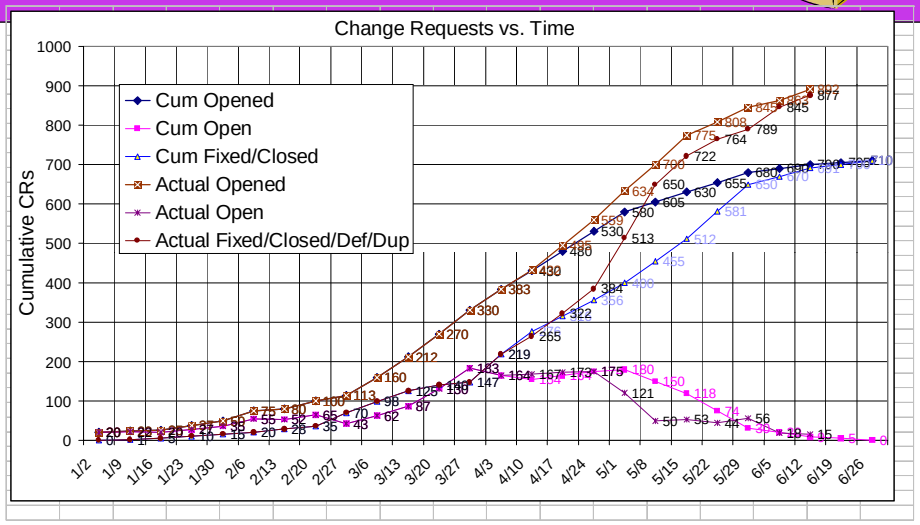
CRs by Component



CR Summary for Product A (Report generated: Tue Mar 2, 1999 09:43 PST)

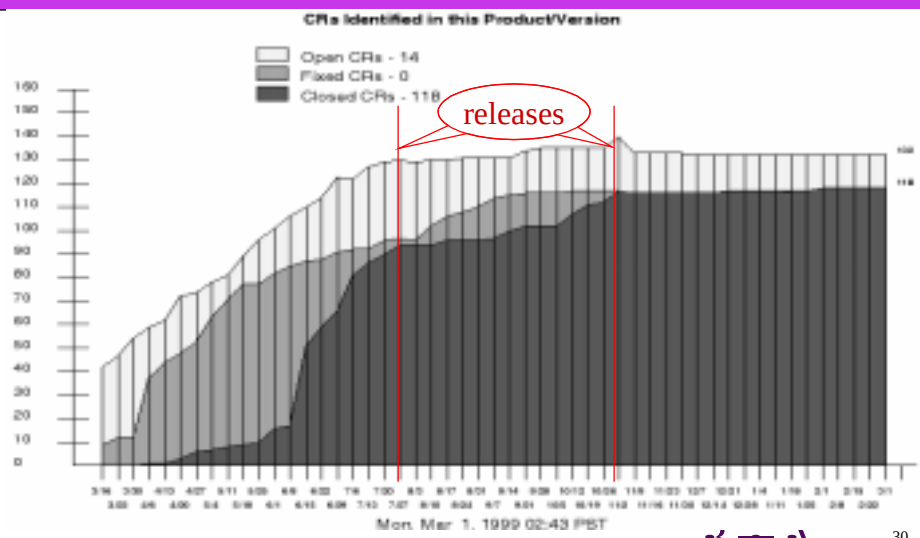
Component	Change Request Statistics													
regression	Priority	1	2	3	4	5	Total	Severity	1	2	3	4	5	Total
	OPEN	0	3	1	1	0	5	OPEN	0	2	1	1	1	5
	FIXED	0	0	0	0	0	0	FIXED	0	0	0	0	0	0
	CLOSED	1	14	0	2	8	25	CLOSED	1	15	0	0	9	25
sanity	Priority	1	2	3	4	5	Total	Severity	1	2	3	4	5	Total
	OPEN	0	0	0	0	0	0	OPEN	0	0	0	0	0	0
	FIXED	0	0	0	0	0	0	FIXED	0	0	0	0	0	0
	CLOSED	0	0	2	1	1	4	CLOSED	0	0	2	1	1	4
stress	Priority	1	2	3	4	5	Total	Severity	1	2	3	4	5	Total
	OPEN	0	1	0	3	0	4	OPEN	0	1	0	0	3	4
	FIXED	0	0	0	0	0	0	FIXED	0	0	0	0	0	0
	CLOSED	0	1	1	0	1	3	CLOSED	0	1	1	0	1	3
TOTAL	Priority	1	2	3	4	5	Total	Severity	1	2	3	4	5	Total
	OPEN	0	4	1	4	0	9	OPEN	0	3	1	1	4	9
	FIXED	0	0	0	0	0	0	FIXED	0	0	0	0	0	0
	CLOSED	1	15	3	3	10	32	CLOSED	1	16	3	1	11	32

Sample: Projected vs. Actual



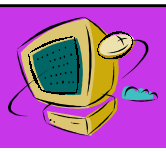
QW - May 1999 Testing Culture - Keith Stobie 29

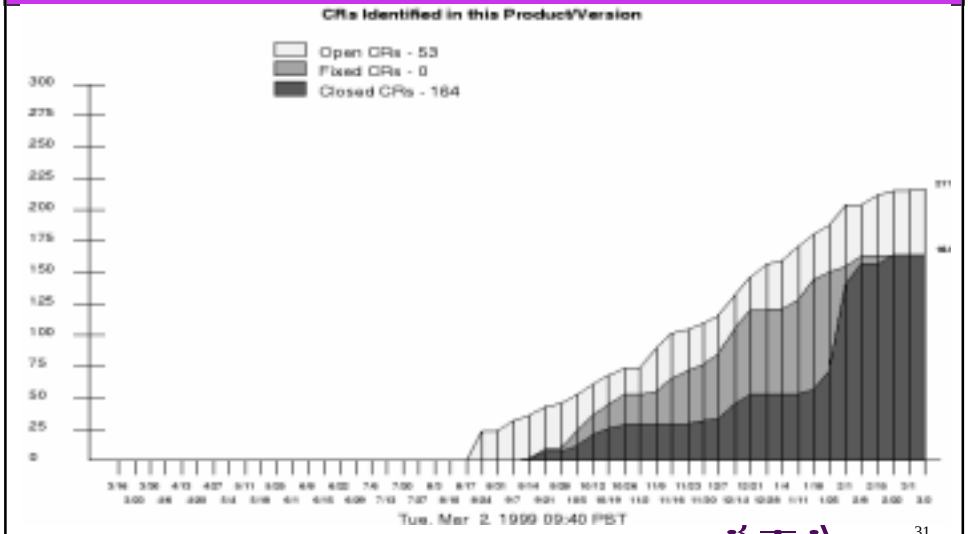
Open/Fixed/Closed over Time



QW - May 1999 Testing Culture - Keith Stobie 30

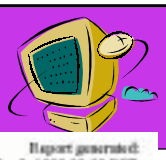
New Product Version





QW - May 1999 Testing Culture - Keith Stobie 31

Report Detail



Active CRs for Prdj 1.1

Report generated:
Tue Mar 2, 1999 09:32 PST

Priority 1

	CR Number	Type	Current Status	Owner	Component	Release Found	Target Release	Summary
1	CR013644	def	CO	chasc	prdjinstall	1.1FCS	Patch 7	patch7 rules repository directory on unix platform for Third Party CD

Priority 2

	CR Number	Type	Current Status	Owner	Component	Release Found	Target Release	Summary
1	CR011558	per	PN	chasc	prdjinstall	1.1Pch2	1.0	Install on NT fails, does not identify Tuxedo
2	CR012621	imp	CO	chasc	prdj_patch_install	1.1Pch2	1.1	Prdj1.1 patch 5 installation script gave wrong "Info"
3	CR012875	sub	CO	lau	prdjClassLibr	1.1	1.1	JSE should send the COMMIT response prior to if before completing the transaction
4	CR012909	sub	PN	lau	WebAppServices_Class_Lib	1.1	2.0	enhancement request for README of Web Application Services IIS filter example
5	CR012911	per	CO	sagr	Prdjbeans_Class_Library	1.1	2.0	Prdj output events are not independent
6	CR012952	per	CO	sagr	Prdjbeans_Class_Library	1.1	2.0	RS The checkboxGroup Custom Property Editor does not support code generation
7	CR013279	per	CO	sohn	Prdjbeans_Class_Library	1.1	1.1	QuickBeans bug

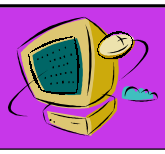
Priority 3

	CR Number	Type	Current Status	Owner	Component	Release Found	Target Release	Summary
1	CR013936	perf	PN	chasc	prdjinstall	1.1	Patch 4	Prdj 1.1 Patch 4 Install does not clean up after install is finished
2	CR013938	per	PN	chasc	prdjinstall	1.1	Patch 4	Prdj 1.1 Patch 4 "readme" needs to be more specific when to use "root" id.
3	CR013782	sub	CO	garmsb	prdjServer	1.1	1.0	T_CLOSE for file transfer causes JSE server socket into TIME_WAIT state

Active CRs: 11

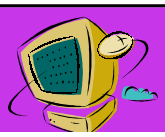
QW - May 1999 Testing Culture - Keith Stobie 32

Code metrics



- Across Releases/Loads
  - Drill down:
    Product ⇒ Load ⇒ Component ⇒
    Directory ⇒ File ⇒ Method
- Lines of Code (LOC),
  Effective Lines (eLOC : exclude '{'/'}' only lines),
  Blanks (Blk), Comments (Com)
- Complexity [Comparative (Cmp),
  Cyclomatic (Cyc),
  Interface (Int=Parameters+Returns),
  Logic (Log=Cyc+Int)]

Load to Load comparisons

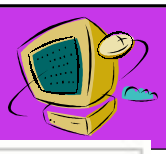


Metrics Summary

Revision	DEV Source Code				QA Source Code				Total Source Code			
	Total Lines	Compared to			Total Lines	Compared to			Total Lines	Compared to		
		add	chg	del		add	chg	del		add	chg	del
2.2 (j6)	2,517,960	j5			1,307,311	j5			3,825,271	j5		
		144	2,707	538		5,849	1,620	546		5,993	4,327	1,084
2.2 (j5)	2,518,370	j4			1,302,531	j4			3,820,901	j4		
		1,144	463	1,222		13,845	1,599	480		14,989	2,062	1,702
2.2 (j4)	2,518,448	j3			1,295,242	j3			3,813,690	j3		
		2,185	3,170	1,201		585,453	1,142	582		587,638	4,312	1,783
2.2 (j3)	2,517,468	j2			710,371	j2			3,227,839	j2		
		29,888	6,669	2,253		12,298	518	53		42,186	7,187	2,306
2.2 (j2)	2,517,190	p12			1,211,834	p12			3,729,024	p12		
		7,791	3,634	2,431		43,839	4,346	2,870		51,630	7,980	5,301
2.1 (p12)	2,511,833	p10			1,173,875	p10			3,685,708	p10		
		61,082	1,066	1,226		3,803	474	1,393		64,885	1,540	2,619
2.1 (p10)	2,512,236	p8			1,171,466	p8			3,683,702	p8		
		242,058	3,126	1,587		1,042,108	1,446	763		1,284,166	4,572	2,350
2.1 (p8)	2,271,765	p5			130,113	p5			2,401,878	p5		
		71,576	20,285	23,687		92,797	1,277	722		164,373	21,562	24,409
2.1 (p5)	2,224,177				440,424				2,664,601			

Source Code Metrics

Single load by component level

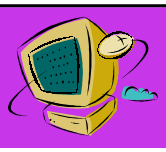


Summary									
filename/subdirectory		LOC	eLOC	Com	Blk	total	Lines compared to CVS tag: j5		
							Added	Changed	Deleted
metrics.dat		0	0	0	0	4	4	0	0
MB	(74 files)	2,118	1,851	483	350	21,150	0	0	0
...	(... files)	...	...	...	...	...	...	...	...
objm	(473 files)	56,306	47,413	17,845	11,865	103,497	0	4	4
orb	(604 files)	247,090	208,096	164,584	66,082	489,784	0	3	0
qa	(8,056 files)	409,271	339,001	123,785	132,485	1,307,311	5,849	1,620	546
...	(... files)	...	...	...	...	...	...	...	...
rpc	(55 files)	16,700	14,569	4,496	[illegible]	33,205	0	0	0
...	(... files)	...	...	...	...	...	...	...	...
uuort	(11 files)	3,109	2,585	707	[illegible]	4,677	0	0	0
web	(176 files)	29,032	25,403	8,981	5,017	51,990	0	0	0
wn	(166 files)	34,856	30,171	9,364	4,771	66,724	0	0	0
TOTAL	(14,589 files)	1,576,729	1,337,266	562,846	363,806	3,825,271	5,993	4,327	1,084

QW - May 1999 Testing Culture - Keith Stobie 35

Source Code Metrics

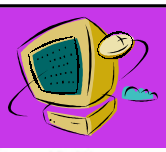
file level: logoncrypt.c with methods



function/method	LOC	eLOC	Com	Blk	Par	Ret	Complexity			
							Int	Cyc	Cmp	Log
des_setkey	29	23	18	1	2	1	3	1	0	1
des_encrypt	48	42	42	4	2	1	3	1	0	1
des_decrypt	48	42	42	4	2	1	3	1	0	1
fix_des_parity	16	12	7	2	1	1	2	2	1	3
_tl_j13	30	23	8	5	2	3	5	5	5	10
_crypt	56	45	14	11	6	2	8	6	9	15
_tl_j019	8	6	0	0	5	1	6	1	0	1
_tl_j0230	19	12	15	6	1	3	4	4	3	7
_tl_j0110	72	59	44	19	2	6	8	10	12	22
TOTAL	463	401	325	95						

QW - May 1999 Testing Culture - Keith Stobie 36

Project Milestone Tracking



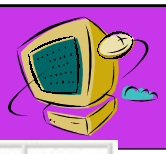
ProductQ Release 5.0 View Milestones or Deliverables

Milestones for Discipline: *all disciplines*

ID	Discipline	Milestones	Owner	Target	Revised	Actual	Status
1	Product Marketing	Marketing Requirements Complete	vijaya	7/7/98			Complete
2	Development	Software Feature Complete	sdf	7/30/98	6/29/98	6/29/98	
3	Development	Software Development Complete	sdf	7/30/98	6/29/98	6/29/98	
6	Development	Unit Test Execution Complete	sdf				
7	Development	Code Review Complete	sdf				
4	Doc Development	Documentation Feature Complete	rbrown				
5	Doc Development	Documentation Development Complete	rbrown				
8	QA/System Test	System Test/QA Start	awr	7/2/98	7/2/98	7/2/98	
9	QA/System Test	System Test/QA Complete	awr	10/2/98	10/30/98		
24	QA/System Test	System Test/QA Test Development Complete	awr	7/2/98	8/7/98		addl dev required beyond dev cmpl

QW - May 1999 Testing Culture - Keith Stobie 37

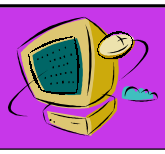
Deliverables Tracking



ID	Deliverable	Owner	First Draft			Baseline Complete			Status
			Target	Revised	Actual	Target	Revised	Actual	
12	Marketing Requirements Document	vijaya	3/30/98		3/30/98	7/14/98			
25	ProductQ 4.1 New product Information	vijaya	7/7/98	7/14/98					
14	ProductQ 4.1 Load Schedule	sdf			5/1/98			6/1/98	
27	ProductQ/TUX Bridge Func Spec	sdf	3/30/98		3/30/98	6/29/98			Version 1.1
28	FML32 Functional Spec	sdf	4/15/98		4/15/98	6/29/98			Version 1.0
29	API Modifications Func Spec	sdf	4/30/98		4/30/98	6/30/98			Version 1.1
30	Licensing Design Spec	sdf	4/15/98		4/15/98	6/30/98			Version 1.1
31	Tracing Functional Spec	sdf	6/29/98		6/29/98				Version 1.0
16	Documentation Development Plan	rbrown	6/15/98		6/15/98				
17	Release Notes	rbrown	9/9/98						MT on 9/15
18	Online Docs Available	rbrown							
19	ProductQ 4.1 Master Test Plan	awr	5/15/98		5/15/98	7/16/98			Version 03
20	QA/Test Results Report	awr	11/13/98						
21	MQ 4.1 API Modifications Test Spec	awr	5/22/98		5/18/98	7/16/98			Version 1.01
22	ProductQ/Tuxedo Messaging Bridge TDS	awr	5/26/98		5/26/98	7/16/98			Version 1.1
23	FML32 Support Test Design Spec	awr	5/29/98	6/10/98	6/10/98	7/16/98			Version 1.01
26	Install/Licensing TDS	awr	8/6/98		8/5/98				
32	ProductQ/TUXEDO Messaging Bridge TDS	awr			[2/8/99]				Version 1.1

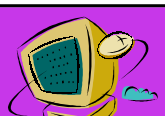
QW - May 1999 Testing Culture - Keith Stobie 38

Test Reporting



- **Test Case**: (unit of Development)
  code or script with pass/fail result
- **Test Point**: (unit of Execution)
  Test Case against a *configuration*
- **Configuration**:
  Hardware, OS, Database, Compilers, etc.

Test Development Tracking

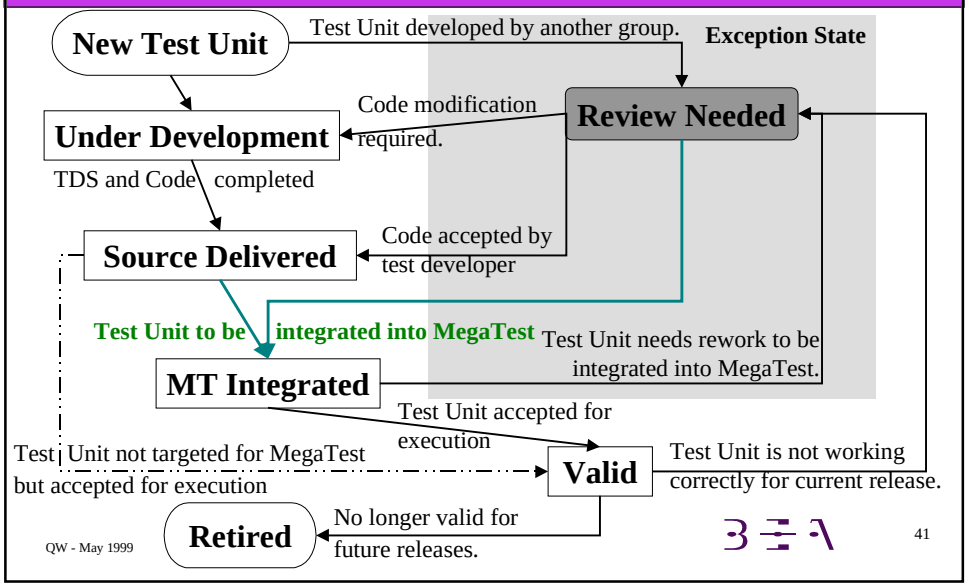


Test Suite	Tester	Site	Total Tests			7/16/98			...	8/15/98			Other Target			Development		Automation	
			NS	Man	Auto	Plan	DC	AC	...	Plan	DC	AC	Plan	DC	AC	Plan	Comp	Plan	Comp
mqapi100	ram	LC	0	0	87	87	87	87	...	0	0	0	0	0	0	87	87 (100)	87	87 (100)
FMLJA	kathomas	LC	0	0	28	0	0	0	...	0	0	0	0	0	0	28	28 (100)	28	28 (100)
mqb100	kats	LC	0	0	32	20	20	20	...	12	12	12	0	0	0	32	32 (100)	32	32 (100)
mqb200	kats	LC	0	0	63	27	27	27	...	30	30	30	0	0	0	63	63 (100)	63	63 (100)
mqb300	kats	LC	0	0	19	0	0	0	...	19	19	19	0	0	0	19	19 (100)	19	19 (100)
mqapi200	ram	LC	0	0	72	72	72	72	...	0	0	0	0	0	0	72	72 (100)	72	72 (100)
mqapi300	ram	LC	0	0	12	12	12	12	...	0	0	0	0	0	0	12	12 (100)	12	12 (100)
FMLSB		LC	4	0	128	0	0	0	...	0	0	0	4	0	0	132	128 (96)	128	128 (100)
FMLMB	kathomas	LC	0	0	128	0	0	0	...	0	0	0	0	0	0	128	128 (100)	128	128 (100)
FMLCA		LC	4	0	128	0	0	0	...	0	0	0	4	0	0	132	128 (96)	128	128 (100)
mqapi400	ram	LC	0	0	72	72	72	72	...	0	0	0	0	0	0	72	72 (100)	72	72 (100)
mqapi500	ram	LC	0	0	26	26	26	26	...	0	0	0	0	0	0	26	26 (100)	26	26 (100)
..		..	.	.	.	.	.	.	...	.	.	.	.	.	.	.	. (.)	.	. (.)
mqb110	kats	LC	0	0	2	0	0	0	...	0	0	0	2	2	2	2	2 (100)	2	2 (100)
mqb120	kats	LC	0	0	2	0	0	0	...	0	0	0	2	2	2	2	2 (100)	2	2 (100)
TOTALS			12	1	840	357	357	357	...	61	61	61	17	4	4	853	840 (98)	840	840 (100)

Development Complete Testing Culture - Keith Stobie Automation Complete 40

Test Unit

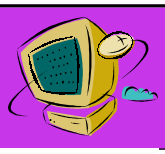
Development Life Cycle Diagram



Test Execution Detail

Feature/Regr Suite	Log Date	Tester	Stat	Plan	Run	Pass	Fail	NA	%compl	Comments
All Delivery Modes-1_2	10/27/98 Alpha_NT_4.0	ebrown		156	156	156	0	0	100	
All Delivery Modes-3_4	10/27/98 Alpha_NT_4.0	ebrown		156	156	156	0	0	100	
API Range Check Sect A	10/23/98 AIX_4.1.4	dampip		133	0	0	0	133	100	
API Range Check Sect B	10/23/98 AIX_4.1.4	dampip		133	0	0	0	133	100	
AVAIL msg-based svc	10/29/98 Alpha_NT_4.0	ebrown		8	8	8	0	0	100	
Bus Isolation	10/27/98 Solaris_2.5.1	xiaod	GRN	1	1	1	0	0	100	
Client Library Compat	10/29/98 Alpha_NT_4.0	ebrown		2	2	2	0	0	100	
Confirmation Style	10/28/98 Solaris_2.5.1	xiaod	GRN	3	3	3	0	0	100	
Dynamic Group Alteration	11/3/98 Alpha_NT_4.0	ebrown		13	13	12	1	0	100	
Destructive Interoperation	10/28/98 AIX_4.2.1	koingh	RED	3	2	2	0	1	100	[illegible] Waiting for [illegible]
Duration	10/30/98 Solaris_2.5.1	xiaod	GRN	1	1	1	0	0	100	
Link Conn Grp Init	11/3/98 Alpha_NT_4.0	ebrown		1	1	1	0	0	100	
Link Cycling	10/29/98 HP-UX_10.20	hsum		3	3	3	0	0	100	
List-Notify Msg-Based Svc	10/28/98 AIX_4.2.1	koingh		2	2	2	0	0	100	
Link Mgmt Msg-based Svc	10/28/98 AIX_4.2.1	koingh		5	5	5	0	0	100	
Msg Transport Bound Chk	10/27/98 AIX_4.2.1	koingh	RED	4	4	4	0	0	100	[illegible] failed on [illegible]
...	...	...		...	...	...	...	...		
TOTALS				1677	1003	1001	2	674	100	

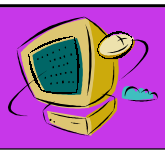
Weekly Execution & CRs



Week	Load	NA	Plan	Run	Pass	Fail	% Run	% Pass	ARITS CRs Filed to Date	Total Weekly CRs Filed	
										SW	Doc
8/20-8/25/98	load3	16283	10866	10662	7925	2737	98	74	42	19	0
8/26-9/1/98	load4	121	27028	0	0	0	0	0	0	15	0
9/2-9/8/98	load4	9792	17867	3677	3628	49	20	98	2	16	0
9/9-9/15/98	load4	12523	15340	9054	8071	983	59	89	16	30	0
9/16-9/21/98	load4	14893	13106	10642	9577	1065	81	89	47	14	0
9/22-9/28/98	load5	5648	22487	246	235	11	1	95	2	11	0
9/29-10/5/98	load5	18505	9630	3395	3241	154	35	95	14	6	0
10/6-10/12/98	load5	19918	8387	6752	6556	196	80	97	17	16	0
10/13-10/16/98	load5	20413	7892	7149	6947	202	90	97	18	13	0
10/19-10/23/98	load6	20786	7536	1491	1179	312	19	79	1	12	0
10/26-10/30/98	load6	20533	7789	5432	5432	0	69	100	1	4	0
11/2-11/6/98	load6	21134	7205	7201	7197	4	99	99	7	6	0

QW - May 1999 Testing Culture - Keith Stobie 43

Test Execution Tracking



Platform Test Execution Summary for load

Platform	Suites	Plan	Run	Pass	Fail	%Run	%Pass	%Fail
AIX_4.1.4	11/11	462	462	461	1	100	99	0
AIX_4.2.1	29/29	563	562	561	1	99	99	0
AIX_4.3	12/12	473	473	465	8	100	98	1
Alpha_NT_4.0	25/25	552	552	551	1	100	99	0
DUX_4.0	28/28	568	568	568	0	100	100	0
DYNIX_4.4.1	16/16	580	579	579	0	99	100	0
HP-UX_10.20	29/29	582	581	581	0	99	100	0
HP-UX_11	16/16	583	583	583	0	100	100	0
Intel_NT_4.0	11/11	297	297	297	0	100	100	0
...	...	...	...	...	...	...	...	...
Solaris_2.5.1	19/19	530	530	529	1	100	99	0
Solaris_2.6	6/6	189	189	189	0	100	100	0
Win_95	5/5	104	104	104	0	100	100	0
TOTALS	256/256	7277	7273	7261	12	99	99	0

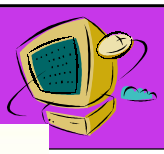
QW - May 1999 Testing Culture - Keith Stobie 44

Release

Test Platform		load1	load2	load3	load4	load5	load6
New Feature	Planned	272	272	5568	8392	4967	5015
	Run	0	0	5161	8253	4967	5015
	%Run	0	0	92	98	100	100
	Passed	0	0	2512	7421	4803	5015
	%Passed	0	0	48	89	96	100
special	Planned	323	11553	5841	3415	2381	1912
	Run	0	512	5487	2603	2381	1912
	%Run	0	4	93	76	100	100
	Passed	0	499	5401	2449	2364	1909
	%Passed	0	97	98	94	99	99
Automated Regression	Planned	714	714	714	747	399	290
	Run	0	0	0	297	174	286
	%Run	0	0	0	39	43	98
	Passed	0	0	0	198	150	285
	%Passed	0	0	0	66	86	99
Manual Regression	Planned	170	170	170	170	170	60
	Run	0	0	0	0	0	60
	%Run	0	0	0	0	0	100
	Passed	0	0	0	0	0	52
	%Passed	0	0	0	0	0	86
Over all	Planned	1479	12709	12293	12724	7917	7277
	Run	0	512	10648	11153	7522	7273
	%Run	0	4	86	87	95	99
	Passed	0	499	7913	10068	7317	7261
	%Passed	0	97	74	90	97	99

QW - May 1999 45

Test Report Summary



Summary Title	Load02	Load03	Load04	Load05
Test Points Pass Rate	90.40%	71.80%	68.47%	86.77%
Test Points Fail Rate	9.52%	26.12%	31.53%	3.31%
Test Point Problems Rate	0.00%	0.00%	0.00%	0.47%
Test Points in Review Rate	0.00%	0.00%	0.00%	9.45%
Test Points Blocked Rate	0.00%	0.00%	0.00%	0.00%
Test Point Pass/Fail Ratio	90.40%	71.80%	68.47%	86.77%
Test Points Run Plan	55.00%	65.54%	98.74%	98.84%
Test Cases Pass Rate	90.00%	67.81%	62.81%	82.63%
Test Cases Fail Rate	10.00%	32.17%	37.19%	3.79%
Test Case Problems Rate	0.00%	0.00%	0.00%	0.47%
Test Cases in Review Rate	0.00%	0.00%	0.00%	13.14%
Test Cases Blocked Rate	0.00%	0.00%	0.00%	0.00%
Test Case Pass/Fail Ratio	90.00%	67.81%	62.81%	82.63%
Test Cases Run Plan	53.19%	64.41%	98.61%	91.03%

QW - May 1999 Testing Culture - Keith Stobie 46

Test Report Failures

F=Fail T=Test problem nt=Not Tested P=Pass R=Reviewed -=No feature

Test Group Installation and license test - install/uninstall using different license

Click to see CR

Test Case	NTint	NTalpha	HP1020	alpha40b	CRs	Note num
y2kInstAI_AC_AK_LC_fresh	F	F	F	F	CR015637	13
Perm_AI_AK_fresh	F	F	F	F	CR015683	14

Test Group Log Central testing - the basic functionality test for Log Central

Test Case	NTint	NTalpha	Sol251	Sol26	HP1020	HP11	Ais421	Ais45	alpha40b	alpha40d	CRs	Note num
msgBrowser_rpt_sum_mngNd	F	F	F	F	F	F	F	F	F	F	CR015601	1
msgdef_export_big	F	F	F	F	F	F	F	F	F	F	CR015626	2
lc_online_help	F	F	F	F	F	F	F	F	F	F	CR015630	3
msgBrowser_flk_combine	F	F	F	F	F	F	F	F	F	F	CR015633	4
log_monitor_oracle	F	F	F	F	F	F	F	F	F	F	CR015646	5
msgdef_sort	F	F	F	F	F	F	F	F	F	F	CR015666	6
lc_launch_central_browser	F	F	F	F	F	F	F	F	F	F	CR015675	7
pm_snmpd_set_ipckey	F	F	F	F	F	F	F	F	F	F	CR015683	8
msgdef_detail_add	F	F	F	F	F	F	F	F	F	F	CR015690	9

QW - May 1999 Testing Culture - Keith Stobie 47

CR015637 (web link/report)

DETAILS	
CR	CR015637
Product	ProductM
Component	Mgr Install
Release Found	2.0
Release Fixed	3.0
Priority	2
Severity	2
Class	Defect
Type	Software
Condition	Open
Status	Def
Owner	adrian
Customer	
Date	12/07/1998
OS	Not Applicable
CPU	ALL PLATFORMS
Entered by	vuong

DESCRIPTION

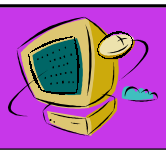
Summary: Year 2K licence doesn't work.

Description: Install AI+AC+AK+LC with a licence Feb. 1,2000 (:
285B) on Windows NT4.0, Alpha NT4.0 and UXHP 10.20. Set date to 2003, but build_agent, start_messaging ... can be started. The licen
ce should be expired.

Notes: *** CHG STS 12/09/98 11:48:23 AM adrian

QW - May 1999 Testing Culture - Keith Stobie 48

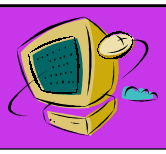
Test Failure Notes



V. Notes New Problems Found	Note	CR#	First Found	CR summary (or explanation)
msgBrowser_rpt_sum_mngNd	1	CR015601	N10	Log Central gives wrong directory for ...
msgdef_export_big	2	CR015626	N10	Got error messages when running msgdef_...
lc_online_help	3	CR015630	N10	The online help page for"Operation ...
msgBrowser_filt_combine	4	CR015633	N10	Filter for msg_body doesn't work when ...
…				
y2kInstAI_AC_AK_LC_fresh	13	CR015637	N10	Year 2K license doesn't work(it doesn't expire)
Perm_AI_AK_fresh	14	CR015638	N10	beamutil displays the license for TA+AK+...

QW - May 1999 Testing Culture - Keith Stobie 49

Test Automation



- Java API framework (AutoTest)
- Multi-Platform execution (MegaTest)
- GUI testing (commercial tool: SilkTest)

QW - May 1999 Testing Culture - Keith Stobie 50

AutoTest: Java API framework

```
public class SimplestTest extends AutoTest {
  boolean canA = false;  boolean canB = true;
  public SimplestTest(TestContext tc) { super(tc); }
  /** Is canA true? */
  public boolean testA() {
    if (canA) return success("Can A");
    else      return failure("Can't A");
  }
  /** Is canB true? */
  public boolean testB() {
    require("A");
    if (canB) return success("Can B");
    else      return failure("Can't B");
  }
```

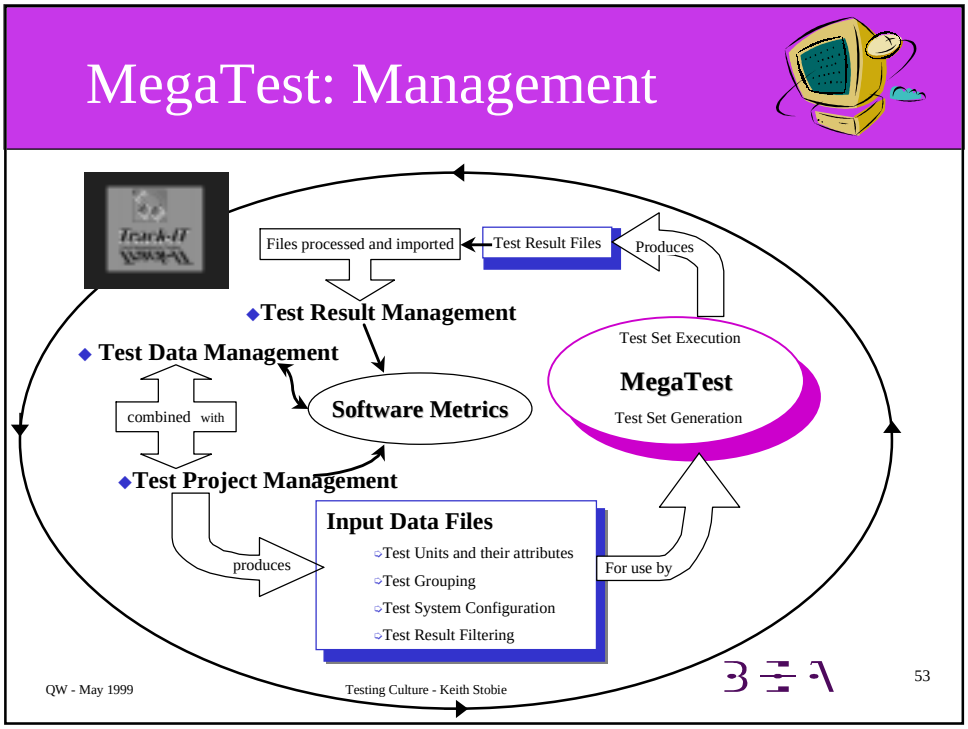
QW - May 1999 Testing Culture - Keith Stobie 51

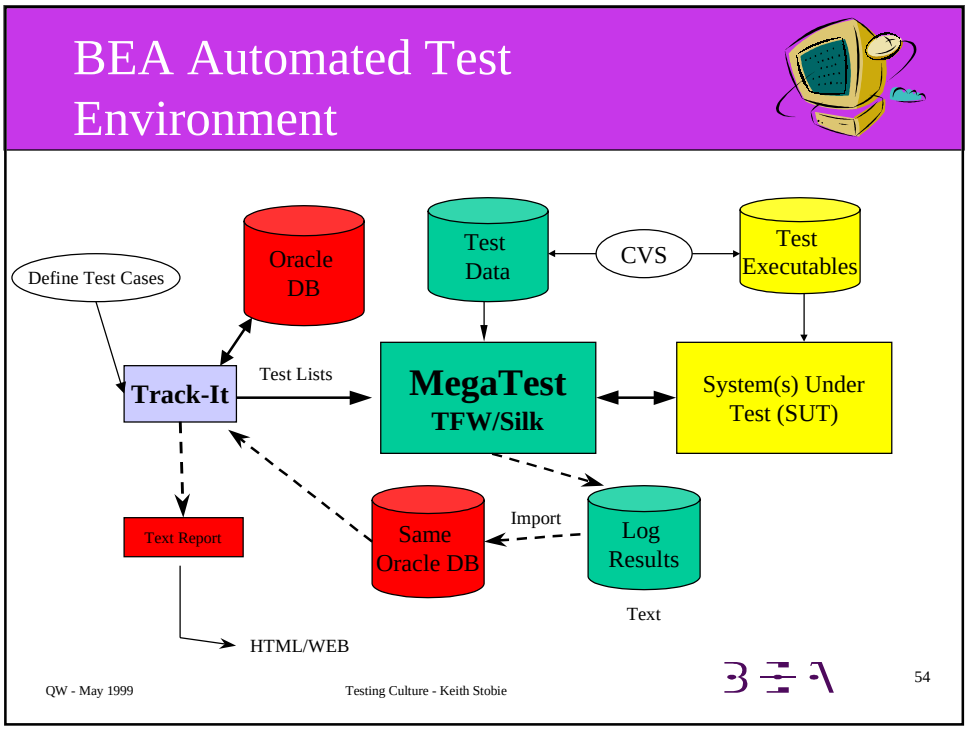
AutoTest: Dependencies

```
public class TestStorage {
  TestStorage(TestContext tc) { super(tc);
    group("allLookups","Lookup1,Lkup2,Lkup3,Lkp4");}
    public boolean testBind() { ... }
    public boolean testLookup1(){require("Bind");... }
    public boolean testLkp2() { require("Bind"); ... }
    public boolean testLkp3() { require("Bind"); ... }
    public boolean testLkp4() { require("Bind"); ... }
    public boolean testUnbind()  {
      require("Bind,%allLookups"); ... }
  }
```

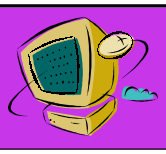
% = run, but not necessarily to have passed

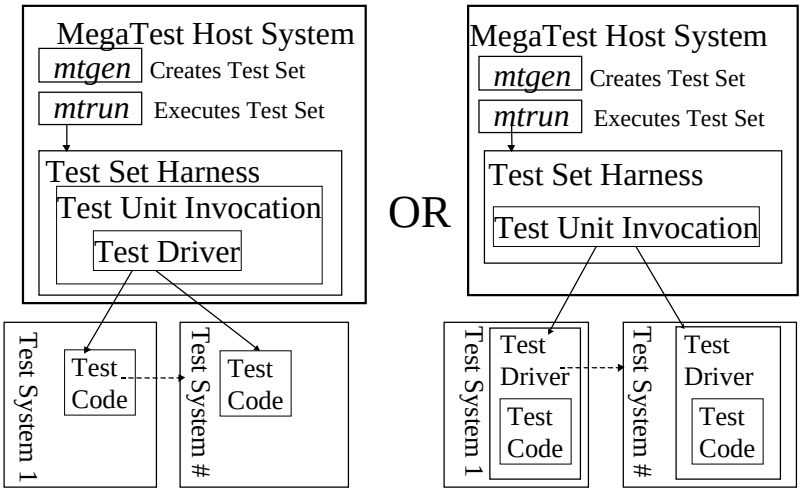
QW - May 1999 Testing Culture - Keith Stobie 52


MegaTest: Management
Track-IT
Files processed and imported
Test Result Files
Produces
Test Result Management
Test Data Management
Test Set Execution
MegaTest
Test Set Generation
combined with
Software Metrics
Test Project Management
Input Data Files
Test Units and their attributes
Test Grouping
Test System Configuration
Test Result Filtering
produces
For use by
QW - May 1999
Testing Culture - Keith Stobie
53


BEA Automated Test Environment
Define Test Cases
Oracle DB
Test Data
CVS
Test Executables
Track-It
Test Lists
MegaTest
TFW/Silk
System(s) Under Test (SUT)
Text Report
Same Oracle DB
Import
Log Results
Text
HTML/WEB
QW - May 1999
Testing Culture - Keith Stobie
54

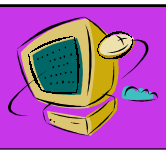
MegaTest: Test Driver





QW - May 1999 Testing Culture - Keith Stobie 55

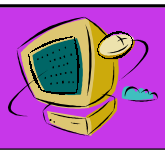
Unattended GUI testing



- Remote execution from MegaTest
- Unattended script execution & automated gathering of results

QW - May 1999 Testing Culture - Keith Stobie 56

Summary / Future



- Hire / train / cultivate outstanding Test Engineers
- Develop common Test Design patterns
- Improve metrics & status automation and integration
- Improve test tools flexibility and integration
- Continuously raise the bar

QW - May 1999 Testing Culture - Keith Stobie 57

Software Quality Week 1999 - San Jose

The Millenium is Getting Closer: The Quickstart to Y2K Testing

Andreas Rudolf / Rainer Pirker
AD Consultants
IBM Global Services Austria

Presentation overview

- Our experience
- The Quickstart: What can you do realistically?
- Good practices
- Bad practices
- Measurements
- Tools used
- Y2K testing differences
- Contingency management
- Best practices summary

IBM Global Services Austria

Our experience has been extended by large Y2K testing projects in different industries

Our Department is strongly focussed on:

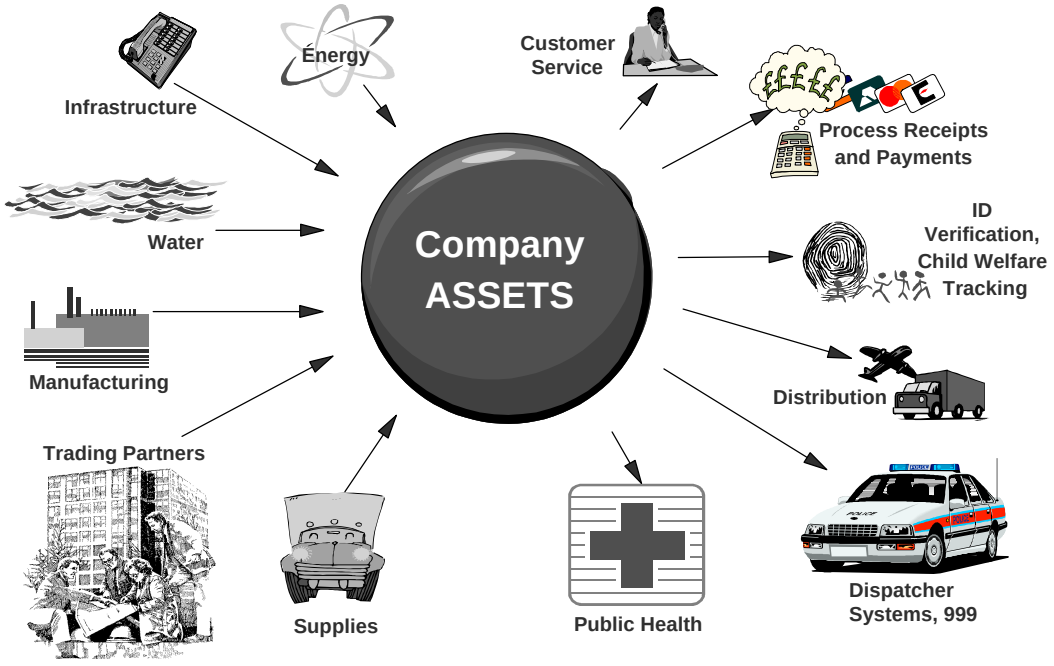
- Test Consulting (EURO, Y2K, e-business, new product developments etc.)
- Application Development Effectivness Consulting (supporting the client to assess and improve their application development processes)

Our experience includes:

- Test consulting and test management for half a dozen large Y2K projects
- Marketing and workshop activities for dozens of customer projects
- Test consulting in the finance, insurance, transport and utility industries

IBM Global Services Austria

External Y2K problems can disrupt a firm's ability to produce, deliver, sell and support products and services

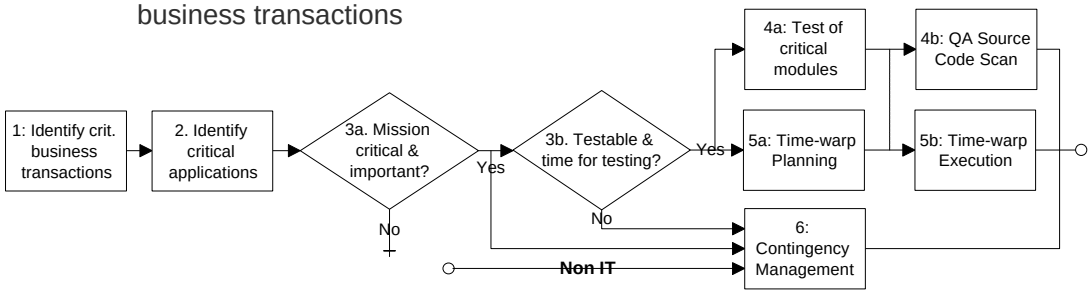


IBM Global Services Austria

As time runs out our Quickstart will show you how to speed up your testing!

The Quickstart essentials:

- Focus on testing critical business transactions
- Plan your test activities in parallel wherever possible
- Key is a complete inventory of your software portfolio and the supported business processes (business transactions)
- Establish contingency management for those applications which cannot be tested as well as those which support mission critical business transactions



IBM Global Services Austria

Identifying the critical business transactions is the key initial task for both time-warp test and contingency management

Quickstart Step 1:

- Identify critical business transactions

Advantages:

- You do not test everything, you select and prioritorise what needs to be tested based on business impact, just as for contingency management
- Business transactions are understood by management, modules, lines of code etc. are irrelevant to the business world

Steps to take:

- Identify and describe the critical business transactions. Do this with people who are subject matter experts.
- Identify the most important parts of a business transaction, these are the results, sequence, schematic input & output description

IBM Global Services Austria

Identifying the applications gives you the technical view of your business critical transactions

Quickstart Step 2:

- Identify critical applications

Advantages:

- Limits the technical support requirement for the test environment, data-aging and problem fixing

Steps to take:

- Identify the applications that support the critical business transactions
- Identify the technical product owners and their deputies

IBM Global Services Austria

If an application is involved in a mission critical & important business transaction contingency management is needed

Quickstart Step 3:

- Applications that are not tested and those which support mission critical & important business transactions need to be covered by contingency management

Steps to take:

- Inventory is neccessary to support this step with links from business transactions to IT applications
- Decide which business transactions can be tested and which are primarily covered by contingency management
- Decide if enough time is left for testing => otherwise contingency management

IBM Global Services Austria

Critical module tests give you a good return on your investment for little effort and QA source code scan secures that your conversion effort do not miss the target

Quickstart Step 4:

- Test of critical modules and perform a QA Source code scan after completion of the conversion phase

Advantages:

- If you find an error it will be a critical one, since many applications depend on the central date routines this module
- Reduces the risk of Y2K failures

Steps to take:

- Test of critical Modules:
  1. Identify the central modules, where dates are calculated;
  2. Build test driver and test cases;
  3. Execute the test cases
- QA source code scan:
  1. Select a pilot application to verify if this step is needed;
  2. Choose an independent team;
  3. Use a different tool as the analysis and conversion team used

IBM Global Services Austria

Time-warp test planning and test execution is the most effective way to find Y2K related bugs

Quickstart Step 5:

- Time-warp test planning and test execution

Steps to take:

- Test system preparation
- Test data aging
- Identify test dates (i.e. rollover 1999-2000, 29.02.2000)
- Identify test cases
- Execute the test cases
- Document the results of the executed test cases
- Problem management, Change management

Recommendation:

- Use a hardware solution for the date manipulation in time-warp tests

IBM Global Services Austria

Good practice #1: Establishing a test strategy is the starting point of your test process

A sample for a proven Y2K test strategy:

Black box test

- Use test cases based on business transactions
- Test 20% of general transactions weighted by business impact
- Test 100% where date is affected

Check with the customer

- Test cases
- User documentation
- Requirement, Design and Program documentation
- Test environment
- available test tools

Test level

- Perform function tests in a time-warp environment
- Integration tests for inter-business line transactions
- Unit test for critical modules like date calculation routines

Test environment

- Dedicated test environment (mainframe + bank branch office)
- Hardware solution for time-warp test

IBM Global Services Austria

Good practice #2: Have the right skills in your test team

Technology

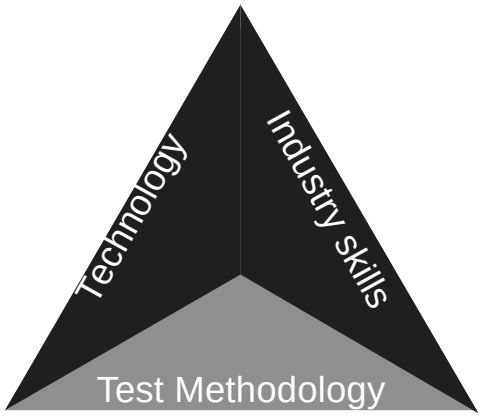
- OS/2, Windows
- Microsoft Office, Lotus Notes
- S/390, MVS, CICS, IMS, JCL

Test methodology

- Black & White box
- Equivalence Class Definition
- Metrics

Industry skills

- i.e for banking: Knowledge of banking systems (stock trading, savings account management etc.)
- Company internal know-how



IBM Global Services Austria

Good practice #3: Plan frequent time-warp cycles and plan them in detail

Week 1:

Application bundles \ Time-warp days	MO	TU	WE	TH	FR	SA	SU
	02.03.1998	03.03.1998	04.03.1998	05.03.1998	06.03.1998	07.03.1998	08.03.1998
	Synchronisation & Preparation (computer-centre)					*Backup*	
Telefone	*time-warp environment not available*						
Bonds							
Correcting values	*testing not possible*						
Foreign money transfer							
Bankbook							

Week 2:

Application bundles \ Time-warp days	MO	TU	WE	TH	FR	SA	SU
	09.03.1998	10.03.1998	11.03.1998	12.03.1998	13.03.1998	14.03.1998	15.03.1998
	Realtime	*Realtime*	*29.12.1999*	*31.12.1999*	*31.12.1999*	*01.01.2000*	*02.01.2000*
Telefone	*- recording test scripts*		*can*	MUST			
Bonds			*can*	MUST			
Correcting values	*- data aging*		*can*	MUST			
Foreign money transfer			MUST	MUST			
Bankbook	*- create comparable datas*		MUST	MUST			

real dates

time-warp dates

Week 3:

Application bundles \ Time-warp days	MO	TU	WE	TH	FR	SA	SU
	16.03.1998	17.03.1998	18.03.1998	19.03.1998	20.03.1998	21.03.1998	22.03.1998
	03.01.2000	*04.01.2000*	*05.01.2000*	*31.01.2000*	*01.02.2000*		
Telefone	*can*	*can*	*can*	*can*	*can*		
Bonds	MUST	MUST	*can*	MUST	MUST		
Correcting values	*can*	MUST	*can*	*can*	*can*		
Foreign money transfer	MUST	*can*	*can*	*can*	*can*		
Bankbook	MUST	MUST	MUST	*testing not possible*			

Week 4:

Application bundles \ Time-warp days	MO	TU	WE	TH	FR	SA	SU
	23.03.1998	24.03.1998	25.03.1998	26.03.1998	27.03.1998	28.03.1998	29.03.1998
	29.02.2000	*29.02.2000*	*01/30.03.2000*	*31.03.2000*	*03.04.2000*		
Telefone	MUST		*can*	*can*	*can*		
Bonds	MUST		MUST	MUST	MUST		
Correcting values	MUST		*can*	MUST	MUST		
Foreign money transfer	MUST		*can*	*can*	*can*		
Bankbook	*testing not possible*						

IBM Global Services Austria

Good practice #4: Establish measurements right from the start

i.e. sample project estimation:

Only one input available: lines of code

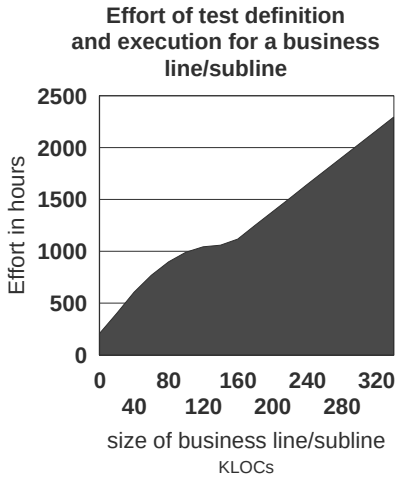
- 50 application bundles
- 6.0 MLOC code to test

leads to an effort of 410 person months

- project length: 18 months leading to 23 people working in parallel for testing.

Due to management and budget restrictions

- only 15 people hired for testing



IBM Global Services Austria

Bad Practice #1: Functional test planned by developers

Developers have limited understanding of the underlying business transactions

- Which is ok, remember it is not their job!

The developer checks his own work

- The person takes on the roles of developer and tester

Steps to do to avoid or get out of this situation:

- Convince the management to get the business people onto the Y2K project
- Priorisation based on business transactions can only be done by the business and people who are subject matter experts

IBM Global Services Austria

Bad Practice #2: The test system is different from the production system

Production system runs on 6 large IBM S/390 processors

The test system runs on just one IBM S/390 processor

- With all production data and the complete batch job network
- Which means the batch job network does not complete overnight, but the clocks do not stop!!!
- The batch job network has to be adapted for one processor

Steps required to avoid this situation:

- Get the same hardware for your test system as for your production system
- or, reduce test data and batch job network (skills of the developers needed)

IBM Global Services Austria

Bad Practice #3: The standard software syndrom

Message from SAP: SAP R3 is Y2K ready.

- Okay. But, what about your self written add-ons? If you have not checked them, these add-ons are by default not Y2K ready!

Car production system:

- A car producer uses a car production system from a partner, which claims to be Y2K ready
- The car producer has written add-ons to this system and five(!) different versions of this system are running now in his production
- This system is the core system for the car producer!

Steps required to avoid this situation:

- Assess carefully
- Divide into standard software and add-ons
- Test when ever possible!

IBM Global Services Austria

Bad Practice #4: Assessing importance of applications based only on date impact

Car production system:

- Message from the Y2K project team: The code of the car production system is only to 0,2% impacted with date calculation relevant code
- Yeah, that is true but this is the most important system for car production
- A failure goes quickly in Mio of $!

Steps required to avoid this situation:

- Assess carefully
- Testing decisions should be based on business impact

IBM Global Services Austria

Bad Practice #5: Technical comparision of the complete production data without link to business transactions

Technical test with all production data

- No functional test is strictly technical
- Almost no priorisation is possible when there is no link from code to business transactions
- After comparision with the baseline data many differences were found
- Very difficult to classify the problem (real Y2K problem, data-aging problem, problems partly caused by differences between production and test system)

Steps to do to avoid or get out of this:

- Reduce data in test!
- Go for the functionally oriented test of the core business transactions!

IBM Global Services Austria

Measurements from a sample test project to address the main objectives in Y2K testing

Customer:

- Large bank in Austria

The software portfolio of the bank

- 17.5 Millon lines of code (MLOC), 6.0 MLOCs under active maintenance
- Programming Languages: 70% Assembler, 16 % PL/I; both under MVS, 14% C under OS/2
- The portfolio is partitioned into 50 application bundles

Goal:

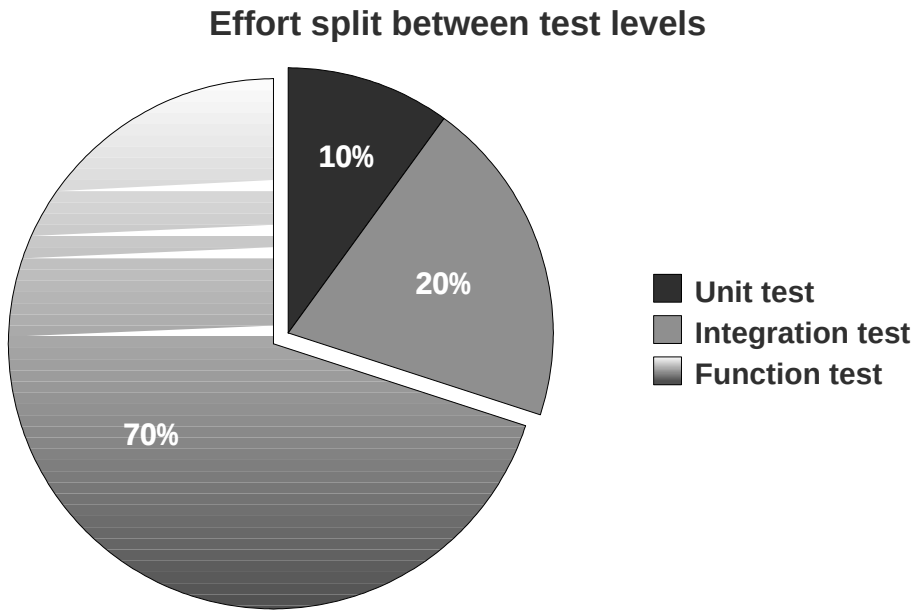
- Testing the software portfolio for Year 2000 readiness
- Establish a test center which can later be used for general regression and stress tests

Conversion Techniques:

- mainly: static window
- partial: flexible window
- only when absolutely neccessary: data expansion

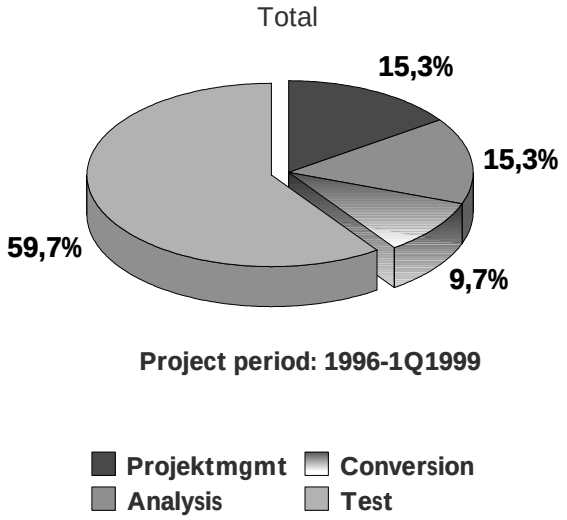
IBM Global Services Austria

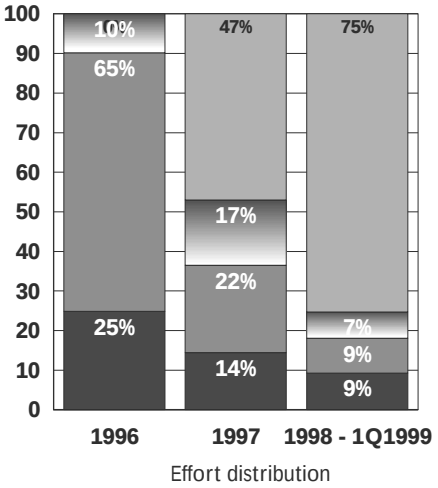
Testing the main functions of applications is 70% of the whole testing effort



IBM Global Services Austria

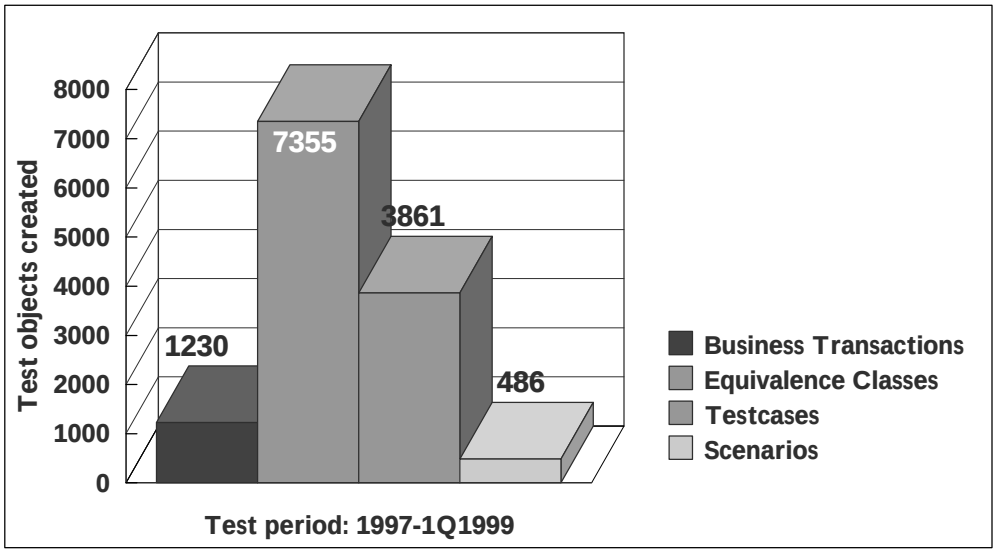
Testing is 60% of the total Y2K effort





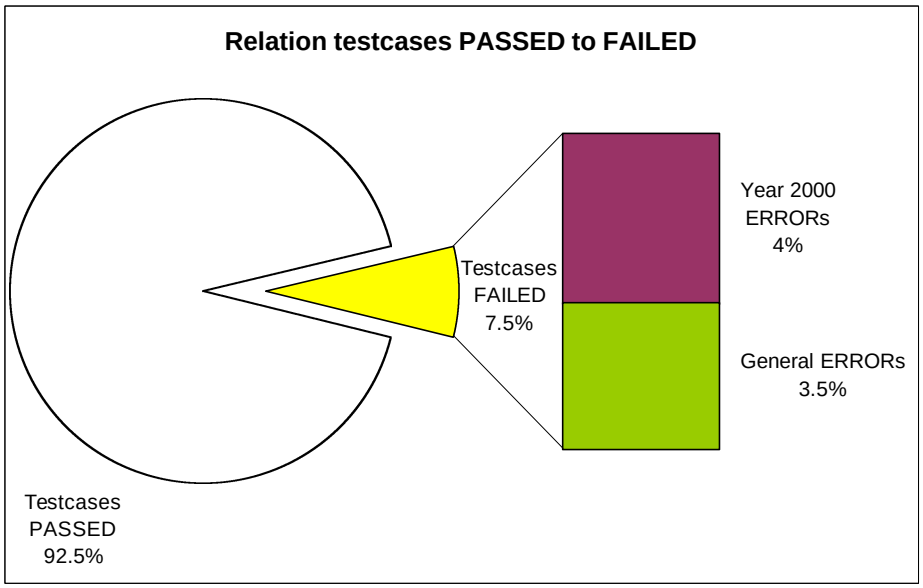
IBM Global Services Austria

Test objects for 50 applications bundles were created



IBM Global Services Austria

7.5% of 14.500 executed Testcases failed



IBM Global Services Austria

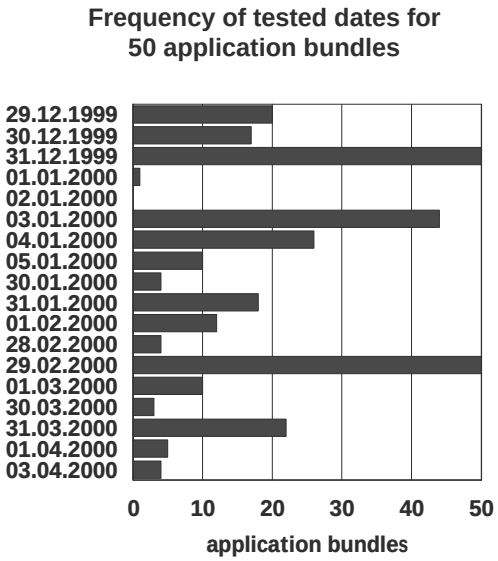
All applications bundles were tested on 31.12.1999 and 29.02.2000 in a time-warp cycle

3 common time-warp slots

- 29.12.1999 -> 04.01.2000
- 30.01.2000 -> 01.02.2000
- 29.02.2000

+ application specific dates

- for example: end of week, end of quarter, bi-weekly interval, first workday after Easter



IBM Global Services Austria

In total we found 469 errors. 91 of them were high severity Y2K bugs

Examples of impact:

Foreign money transfer application:

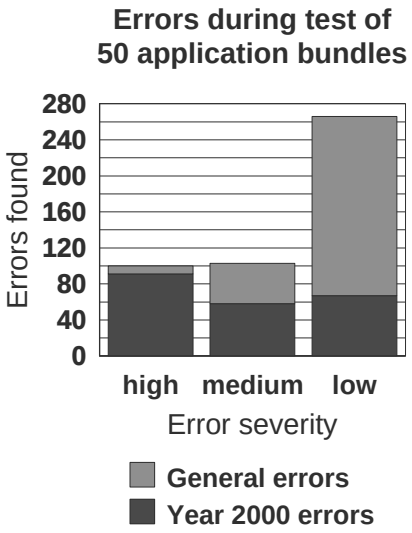
- No money transfer possible in Year 2000

Common used date module:

- Central date calculation routine traps at 29th February 2000

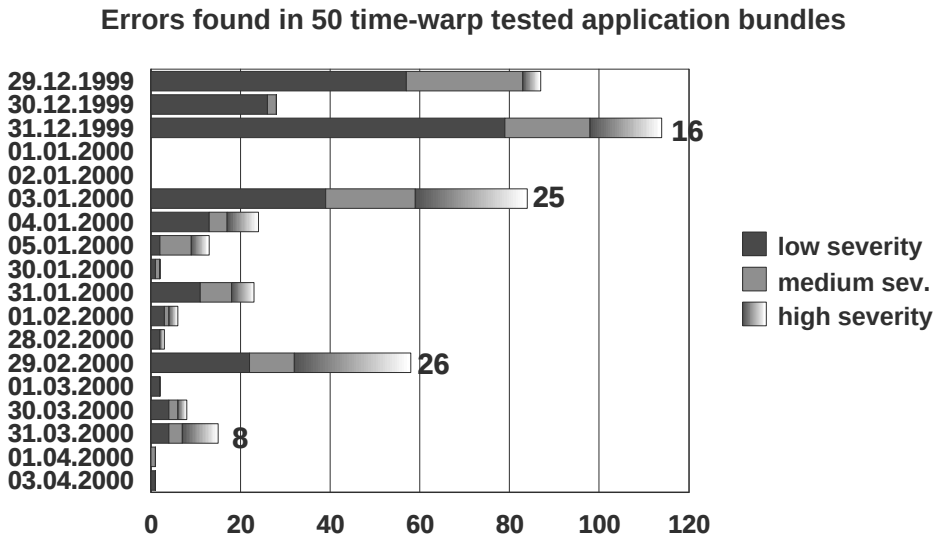
B/L (bank bill) management:

- Acceptance of a draft not possible at 29th February 2000



IBM Global Services Austria

Most severe errors were found on 29.02.2000



IBM Global Services Austria

Some examples of date related problems in the period: December 1998 / January 1999

All Taxi Meters in Stockholm failed

The largest Petrol companies credit cards failed

Post Office postal bank account disrupted

UK Bank letter

- ..... Limiting liability for year 2000 insurance incidents
- Insurance Period: 1st January 1999 to 1st January **1900**

Electronics manufacturer - three weeks before Christmas found 140 occurences of "*IF YEAR equals 1999*"

Irish Electricity Board

- Year 2000 ready solution implemented in December 1998
- 3 weeks of major disruption
- Still some supply chain problems

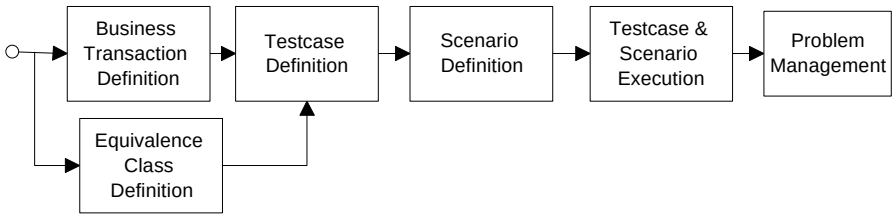
IBM Global Services Austria

Testcase administation and problem management databases were developed to support our Y2K test methodology

Our cornerstone tool suite is based on Lotus Notes and MS Access and integrates:

- Business transaction definition
- Test case definition
- Equivalence classes definition
- Test case and scenario execution
- Problem management

Methodology and Tool: From Test Case Definition through Execution



IBM Global Services Austria

Tools are used to improve testing productivity and to support data manipulation

Capture/replay tool

- QAHiperStation™ from CompuWare (mainframe applications)
- QA Partner™ from Segue (client/server systems)

Advantage of using a capture/replay tool for regression tests

Activity	Time using QA Partner™	Time without QA Partner™
Test case creation	160 h	150 h
Correcting generated scripts	5 h	
Initial learning effort	5 h	
Test case execution once	10 h	40 h
Total	**180 h**	**190 h**
Test execution 3 more time	30 h	120 h
Total	**210 h**	**310 h**

Data manipulation

- File-AID™ from CompuWare (prepare and verify test data)
- File-AID/Data Ager™ from CompuWare (data aging)

IBM Global Services Austria

Y2K testing differences to normal testing

Short testing time at fixed date

- The date is definitely fixed, there is no way to stretch the testing time

No quality improvement

- The goal is simply to fix the Y2K related bugs

Affects the whole company

- Interfaces to other companies or the government may be affected

Special test beds needed

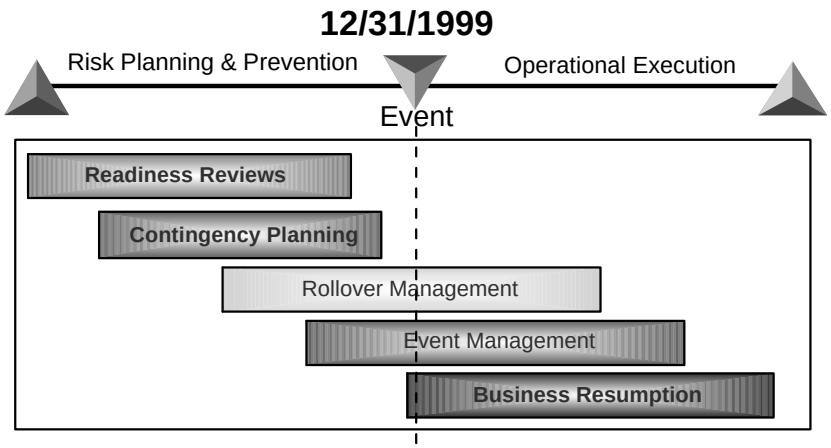
- For mainframe tests: Updated licenses, advance system clock

No pilot users

- No pilot phase with pilot users can be done as with normal rollouts

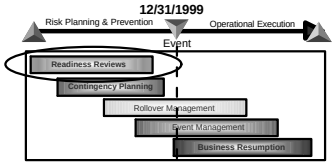
IBM Global Services Austria

Contingency Management is a framework that addresses business continuity strategy, risk planning, prevention, and recovery from Y2K related problems



IBM Global Services Austria

Readiness Reviews provide a high-level analysis of Y2K initiatives in order to identify opportunities for improvement based on progress to-date.



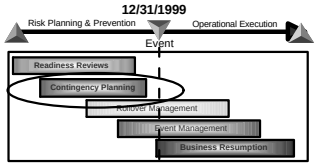
Launch Readiness Review → Gather Data → Analyze Findings → Present Findings

Characteristics:

- **High level review**
- **Quick completion - One Week per area**
- **Focus on the key Y2K project areas**
- **Leverage questionnaires and templates**
- **Identify Issues and Recommendations**

IBM Global Services Austria

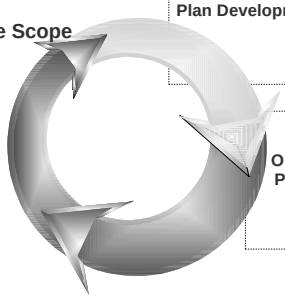
The contingency planning model covers strategy, plan development and testing the plan.



Phase 1
Strategy Definition

Launch Contingency Planning Strategy → Analyze Business Processes / Analyze Assets & Vendors / Analyze External Agents → Prioritize Processes, Develop Strategy → Review Strategy With Management

- **Refine Contingency Plans**
- **Refine Scope**



Phase 2
Plan Development

Launch "Plan Development" Activity → Develop Contingency Plans → Review Contingency Plans

Phase 3
Organizational Preparedness

Launch Organizational Readiness → Train Personnel on Plans → Test Contingency Plans → Monitor & Modify Plans

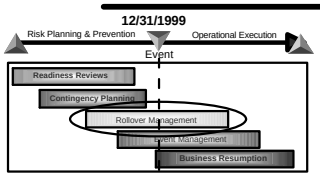
IBM Global Services Austria

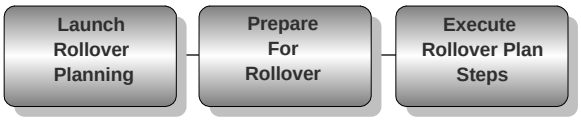
Example from a major Swedish Bank - the resulting contingency plans have a wide priority spread and a wide variety of contingency activities

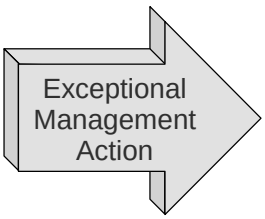
- Contingency plans have been considered for 339 prioritised business sub-processes across all business areas of the Bank.
- The defined contingency preparations include:
  - development of information plans
  - education, training and extra staffing
  - development of contact lists, contingency organisations
  - assessment of alternative routines and information channels
  - synchronisation of contingency planning including mutual testing with external partners
- Some examples of contingency actions defined in the plans are the use of:
  - stand alone PCs with the clock reset to a time before Y2K
  - hardcopies of customer information, positions and engagement status printed in advance
  - other backups; telephone alternatives; multiple operators, mobiles
  - manual routines

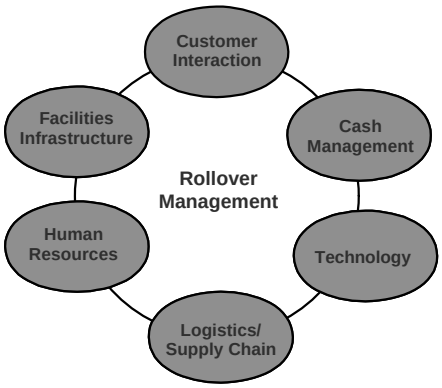
IBM Global Services Austria

Rollover Management establishes procedures, people, equipment, materials and agreements to avoid or minimize Y2K exposures around the rollover period



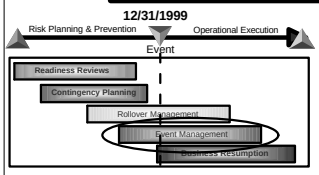




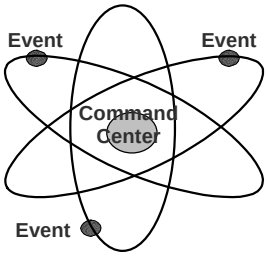


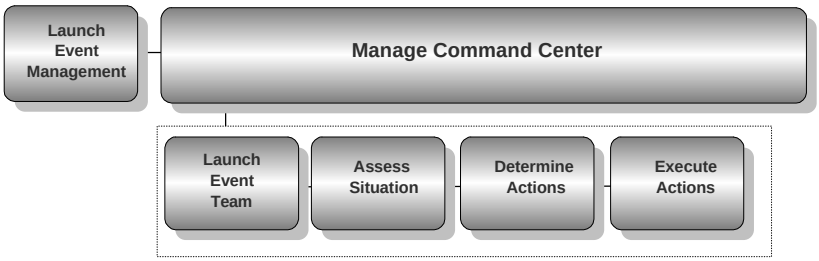
IBM Global Services Austria

Event Management assists in establishing a Command Center to monitor for events, prioritize events, deploy resources and report status to executive management.



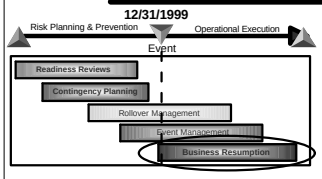
Event Management

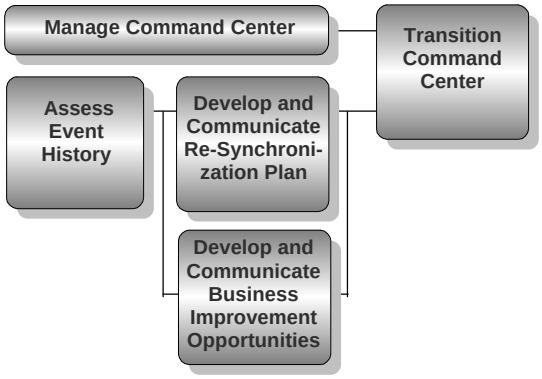


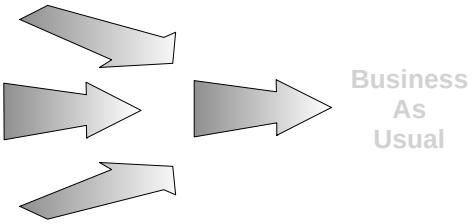


IBM Global Services Austria

Business Resumption provides support for the analysis of an event, the implementation of corrective actions and the re-establishment of normal business processes







IBM Global Services Austria

Best practices summary

Testing:

- Use test cases that test critical business transactions
- Follow a black box testing strategy
- Use exact values from equivalence class definitions, for simple test cases use the equivalence class definitions
- Establish metrics right from the start
- Establish a test system that is a close copy of the production system

Contingency management:

- Even if you are confident of your own Y2K work, you must protect yourself against external failures
- It is prudent to begin now to identify and prepare for the most serious failures
- Contingency management focuses on the business processes essential to business continuity, and on the Y2K-affected assets held by you and outside parties on which those processes depend

IBM Global Services Austria

Open questions?

Ask them right now!

or if they come up later contact us:

- Andreas Rudolf
- e-mail:Andreas_Rudolf@at.ibm.com
- Phone: +43-1-1706-4347
- Fax: +43-1-1706-2393

- Rainer Pirker
- e-mail:Rainer_Pirker@at.ibm.com
- Phone: +43-1-1706-4163
- Fax: +43-1-1706-2393
- Web: www.geocities.com/Vienna/Strasse/7559/

IBM Global Services Austria

Evolutionary Delivery Project Management

- or "EVO method.
- Quickstart Presentation
- Quality Week, San Jose
- 1:30 PM Wednesday 26th May 1999

by Tom Gilb

© Gilb. Permission to copy and share freely, with credit, is granted.

For detail:

Gilb, "Principles of Software Engineering Management", Addison Wesley Longman.

Gilb, 'Evo: The Evolutionary Project Managers Handbook' at

http://www.Result-Planning.com

Introductory Literature

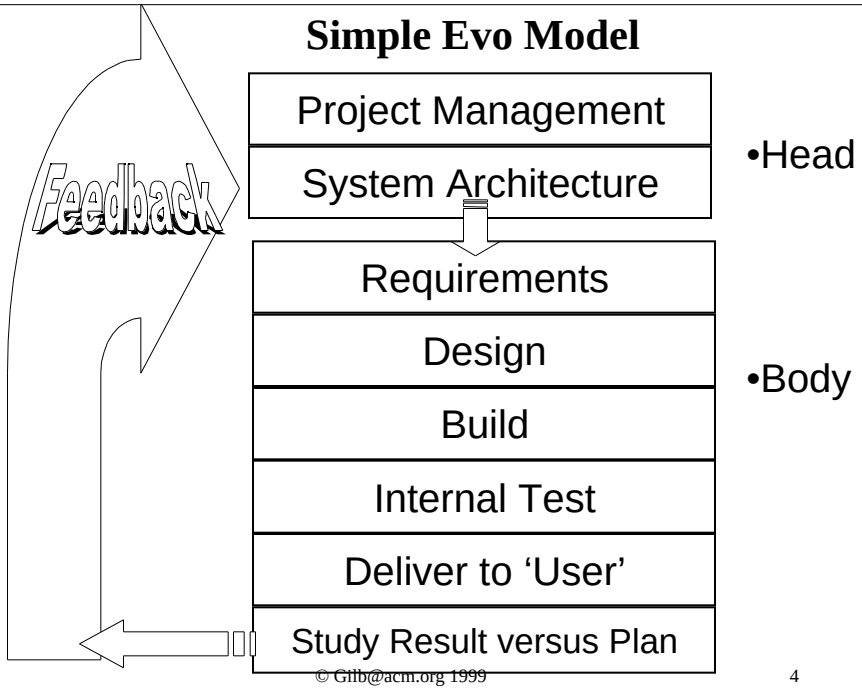
- See Chapters 7, 13, 15 (Literature survey)

Conventional Project Model

Frozen		**Build to design**	**=Requirements?**
Requir e- ments	**Engin- eering**	**Construction/ Acquisition**	**Test (system, acceptance)**

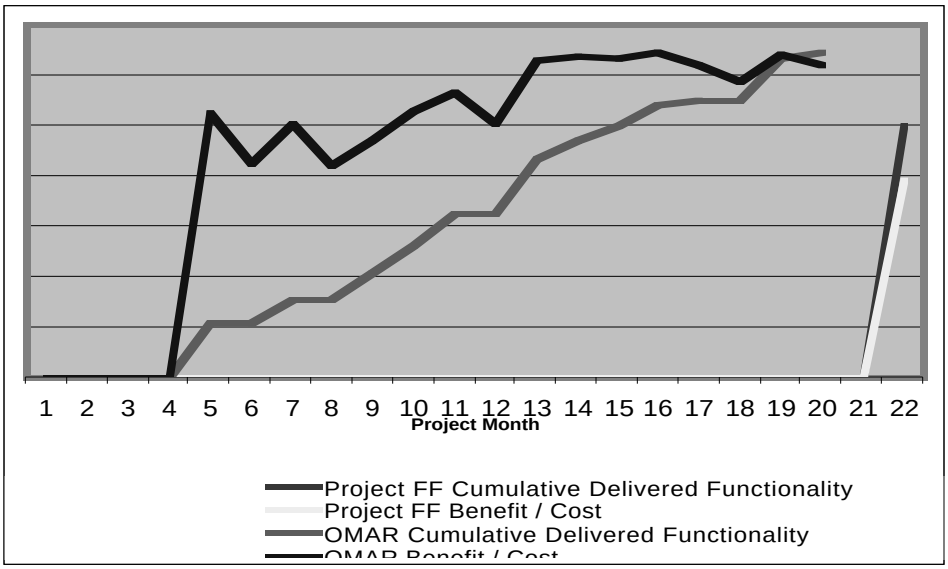
© Gilb@acm.org 1999

Simple Evo Model



© Gilb@acm.org 1999

OMAR Case delivery value vs Waterfall (1998)



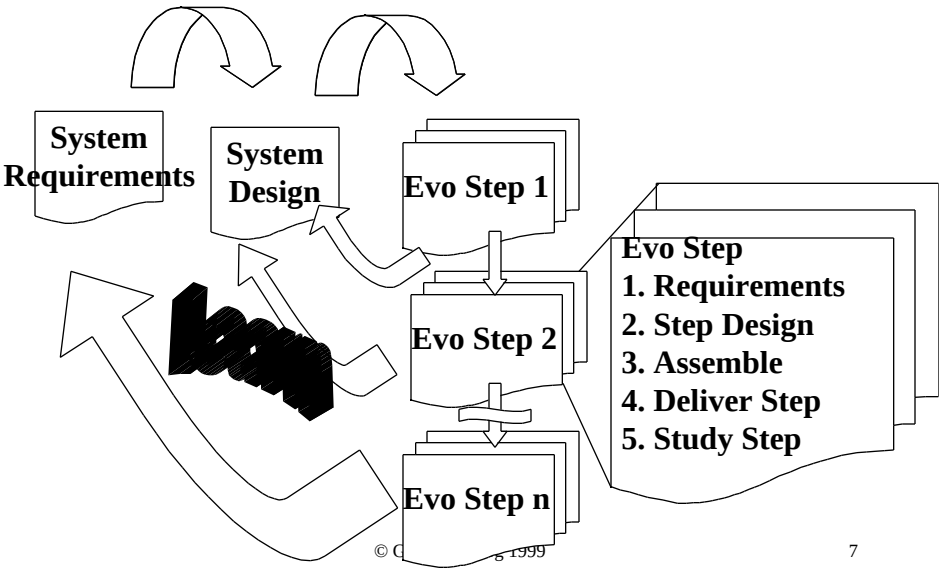
Using Evolutionary Project Management, To Get More Quality, From Fewer Resources, In Less Time; By Stuart Woodward, DoubleHelix Software & Services Ltd.

© Gilb@acm.org 1999 5

Basic Principles of Evo Delivery

- **RD1. Any Project can be managed better using process control.**
- **RD2. Any 'Improvement' can be delivered as a series of smaller changes.**
- **RD3. No person knows all the results of a design in advance.**
- **RD4. No person can know what all the goals should be, in advance.**
- **RD5. You must be prepared to compromise intelligently with reality.**
- **RD6. Early delivery means early payback.**
- **RD7. The customer is always right, even when they change their goals.**
- **RD8. There is no real end to a project, if we have competition.**
- **RD9. You cannot foresee every change, but you can foresee change itself.**
- **RD10. Useful results are your only justification for existence.**

“Evo” model


System Requirements
System Design
Evo Step 1
Evo Step 2
Evo Step n
Evo Step
1. Requirements
2. Step Design
3. Assemble
4. Deliver Step
5. Study Step

Evo Cousins: Process Control

Evo	DPP Defect Prevention CMM 5	Inspection (Doc. Qual. Ctl.) CMM 3	SPC Statistical Process Control
Project oriented	*Organization orientation*	*Document, and work process*	*General, but manufacturing bias*
Meet project goals or change them	*Improve general org. ability work*	*Approve work, clean work, analyze tasks*	*Improve work processes*
Project Manager	*Quality Director*	*Inspection process owner*	*Quality Control*

© Gilb@acm.org 1999 9

The Evo Cousins' Commonality

- **Learning**
- **Measurement**
- **Future Improvement orientation**
- **Process Improvement**
- **The Deming/Shewhart (Juran) Statistical ideas**
- **Eternal learning**
- **Distinguish between 'chance causes' and 'common causes' (fix the common causes)**

© Gilb@acm.org 1999 10

EVO differs from conventional project management methods in these ways:

- detailed design is integrated into entire system life cycle, rather than a distinct initial phase
- specific provision is made for adjusting requirements, design, and resource estimates as practical result delivery experience dictates;
- it is heavily oriented towards getting control over multiple critical result (qualities and resources) parameters simultaneously;
- it attempts to deliver some concrete improvement results to real users extremely early;
- it attempts to deliver results to users frequently, at typically 2%-of-total-cost intervals;
- it is oriented towards project survival and success in spite of high risks, and turbulent environments;
- by early frequent realistic deliveries it gets credible feedback to project management, which permits adjustment in time of anything threatening project success.

EVO aims to:

- deliver early: first deliveries (of some improvement) "next week" if you are committed, even for large total projects;
- deliver partial improvements regularly: weekly, monthly (non-disturbing for user/customer);
- prioritize customer's most -important improvements for early delivery;
- discover "true costs", in practice, before larger commitment is made;
- involve the customer early, practically, frequently;
- control quality aspects quantitatively;
- control costs and deadlines, much better than conventional project management;
- exploit all other aspects of ”Result Driven Management" methods (Impact Estimation, Planguage, Document Quality Control).

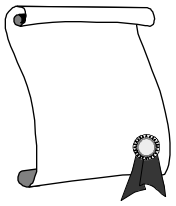
What EVO is not

- **phased development**
  - **(Evo uses much smaller increments, 50 steps, not 3);**
- **prototyping**
  - **(Evo delivers final cumulative results to users);**
- **incremental development**
  - **DoD 498: no change to requirement and design permitted**
  - **("delivery" is the operative word for us here);**
- **conventional critical path method**
  - **(our 'critical paths' are improvements, not just costs or time);**
- **a "waterfall" model method**
  - **(it is more like a series of many small waterfalls).**

© Gilb@acm.org 1999 13

Evolutionary Results Delivery Method Project Planning Policy

- **PP1.(Budget) No project cycle shall exceed 2% of total budget before delivering measurable results to a real environment.**
- **PP2. (Deadline) No project cycle will exceed 2% of total project time (one week for a year's projects) before it demonstrates practical measurable improvement, of the kind targeted.**
- **PP3.(Priority) Project cycles which deliver the most planned results to customers, for the resources they claim, shall be delivered first, to the customer.**



© Gilb@acm.org 1999 14

Priority: a definition

- *'priority'*

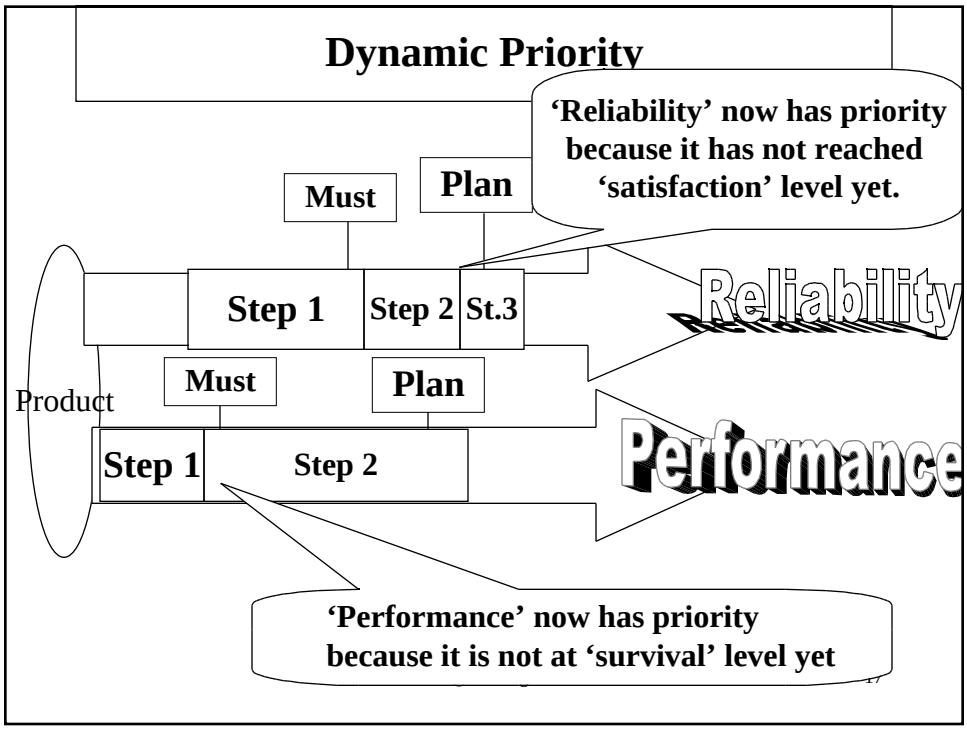
- ***A choice you make when resources are limited***

© Gilb@acm.org 1999 15

Dynamic Priority Principles

- **Priority: is something which has strongest claim on your limited resources.**
- **Priority is set by your "targets" (Must, Plan)**
- **Priority is determined 'as time goes by'**
  - **by need for <u>survival</u> (distance to Must levels)**
  - **then by need to <u>succeed</u> (distance to Plan levels)**
- **when all Plan levels are achieved**
  - **there is nothing which has claim on your resources**
  - **you are done with the project**
  - **free remaining resources for other projects**

© Gilb@acm.org 1999 16



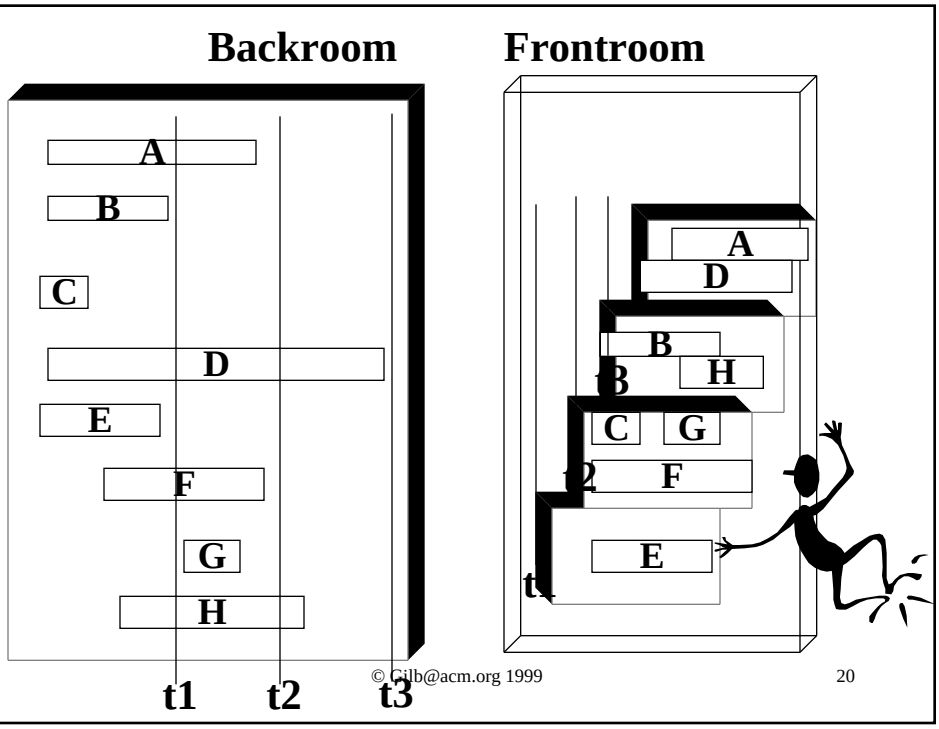
Practical priorities

- Survival Priorities
  - Real , but undocumented factors
  - MUST levels of stated requirements
    - with respect to time and other [conditions]
- Satisfaction Priorities
  - Enough resource to satisfy PLAN levels
  - Real, but undocumented factors
  - PLAN levels of stated requirements
    - with respect to time and other [conditions]
- IF no Priorities remain (no claim on resource)
  - *all PLAN levels are satisfied [on time, place]*
  - *Project stop*
  - *Redistribute or save resources*

Backroom Development, Frontroom Delivery.

- The Evo deliveries are a *'cyclical change'* from the user/customer/potential market ("**recipient**") point of view.
- Some of the components which make up a delivery step may take much longer time to purchase or build, than the **delivery cycle** timing.
- These are prepared in the background ('**backroom**'), early enough to deliver at the appropriate step, from the '**frontroom**'.
- The time needed to get backroom step components ready for delivery is invisible to the step recipient.
- Until the step components are actually ready, they cannot be considered for scheduling as a frontroom step delivery.

© Gilb@acm.org 1999



© Gilb@acm.org 1999

Why do people have difficulty finding smaller delivery steps?

- **people have almost no formal experience, or teaching, on how to sub-divide**
- **they assume falsely, that we are going to sub-divide their 'design idea' (we divide results)**
- **they don't realize we are going to sub-divide the result delivery (increments of user pleasure)**
- **they have little faith that there is a solution, and give up easily (there is always a way to divide)**
- **they have no managers who insist on an Evolutionary solution,**
  - **and give people help to do it.**

© Gilb@acm.org 1999 21

Here are some principles of dividing up a project into smaller steps:

- **Divide the result, not the system.**
- **Get some real improvements now, don't wait for final goals.**
- **Ask your customer what they want most improved.**
- **Don't be afraid to use the "old system" as a temporary crutch.**
- **Don't hesitate to use some temporary "scaffolding" to get results now.**
- **Make unusual agreements with suppliers, like 'pay as you actually use'.**
- **Don't give up. Call it a challenge. Try other minds. There is a way.**
- **Invent new design ideas,**
  - **better for incrementing than those initally specified.**
- **Go to the "customer" to get permission to change constraints.**
- **Use the policy (2% resource cycles, biggest bang for buck first)**

© Gilb@acm.org 1999 22

Practical Tip . Advice on finding smaller implementation cycles when it seems difficult or impossible to achieve. PART 1 of 2

- • **Believe there is a way to do it, you just have not found it yet!**
- • **Identify obstacles, but don't use them as excuses: use your imagination to get rid of them!**
- • **Focus on some usefulness for the user or customer, however small.**
- • **Do not focus on the design ideas themselves, they are distracting, especially for small initial cycles. Focus on getting results and feedback.**
- • **Think; one customer, tomorrow, one interesting improvement. When that succeeds, multiply it.**
- • **Focus on the results (which you should have defined in your goals, moving toward PLAN levels).**
- • **Don't be afraid to use temporary-scaffolding designs. Their cost must be seen in the light of the value of making some progress, and getting practical experience.**
- • **Don't be worried that your design is inelegant; it is results that count, not style.**
- • **Don't be afraid that the customer won't like it. If you are focusing on results they want, then by definition, they should like it. If you are not, then do!**
- • **Don't get so worried about "what might happen afterwards" that you can make no practical progress.**
- • **You cannot foresee everything. Don't even think about it!**

© Gilb@acm.org 1999

Practical Tip . Advice on finding smaller implementation cycles when it seems difficult or impossible to achieve. PART 2 of 2

- • **If you focus on helping your customer in practice, now, where they really need it, you will be forgiven a lot of "sins"!**
- • **You can understand things much better, by getting some practical experience (and removing some of your fears).**
- • **Do early cycles, on willing local mature parts of your user community.**
- • **When some cycles, like a purchase-order cycle, take a long time, initiate them early, and do other useful cycles while you wait. (Parallel activity is OK!)**
- • **If something seems to need to wait for "the big new system", ask if you cannot usefully do it with the 'awful old system', so as to pilot it realistically, and perhaps alleviate some 'pain' in the old system.**
- • **If something seems too costly to buy, for limited initial use, see if you can negotiate some kind of "pay as you really use" contract. Most suppliers would like to do this to get your patronage, and to avoid competitors making the same deal.**
- • **If you can't think of some useful small cycles, then talk/observe directly with the real "customer" or end user. They probably have dozens of suggestions.**
- • **Talk with end users in any case, they have insights you need.**
- • **Don't be afraid to use the old system and the old "culture" as a launching platform for the radical new system. There is a lot of merit in this, and many people overlook it.**

© Gilb@acm.org 1999

How to get EVO into an organization

- **Show that it works by <u>practical</u> demonstration.**
- **Get the policy of planning the Evo way adopted by <u>top</u> management**

© Gilb@acm.org 1999

How to sell EVO to managers

- **Show, in practice, that it:**
  - **Gives results early**
  - **Gives better project control versus plans and budgets**
  - **Pleases the customers**
  - **Is structurally incapable of a large disaster.**

© Gilb@acm.org 1999

How to sell EVO to planners and engineers

- ***Show them that:***
- ***they can get into real action with users and systems earlier***
- ***they get credibility with customers and their managers by the stream of real results***
- ***they can resolve many of their goal and design idea problems by the practical route***
- ***they can prove many points to customers and managers in practice***
- ***they can be more experimental with new technology, since failure is more limited and controlled.***
- ***Help them to make their first Evo plan in practice.***

© Gilb@acm.org 1999

How to train people to do EVO

- **An experienced Evolutionary planner must help them to make a real plan.**
- **Secondarily, the "Requirements Driven Management" book contains the detailed theory.**
- **You really need to master the art of specifying goals quantitatively,**
- **and estimating design impact versus those goals.**
- **Evolutionary result delivery is not just about chopping up the project into cycles. It is fundamental that those cycles be towards defined primary improvement goals.**

© Gilb@acm.org 1999

Step Impact Estimation and Accounting

	Cycle 1 I1[CA]		Cycle 2 I1[NY] & I4[CA]		Cycle 3 I1[DC] & I5		Cycle 4 I3[AZ]	
	Cycle	Sum	Cycle	Sum	Cycle	Sum	Cycle	Sum
GOAL-Q	30%	30%	40%	70%	10%	80%	30%	110%
GOAL-I	0%	0%	60%	60%	25%	85%	25%	110%
COST-C	2%	2%	5%	7%	4%	11%	39%	50%
Bene/Cost	30/2	30/2	100/5	130/7	35/4	165/11	65/39	220/50

- **An IE table for project management planning or feedback. It specifies the projected or actual impact of any set of design ideas done at a particular implementation cycle.**

© Gilb@acm.org 1999

Evo Book Chapters

- ***0: Overview. The essential character of Evo.***
- ***1: Requirements at Project Level: The Evo direction.***
- ***2: Design: The Evo 'Means' to the Target 'ends'.***
- ***3: Impact Tables: The Evo Accounting and Planning Mechanism***

- ***4: Evo Planning: How to specify an Evo Project plan.***
- ***5: Evo Step Objectives: Cycle Requirements***
- ***6: Detailed Evo Step Design: Extracting function and design to make a step.***
- ***7: Planning the Evo Step: The delivery cycle in detail.***
- ***8: The Evo Backroom: Readying components for packaging and delivery.***
- ***9: Evo Culture Change***

© Gilb@acm.org 1999

Quotations from Evo Practice

From the book manuscript by Tom Gilb
At www.result-planning.com

© Gilb@acm.org 1999

Any Complex System <-Todd HP

- "Evolutionary Development has been positioned here' *[in cited HP Journal article]* 'as a life cycle for software development, but it really has much broader application to any complex system."
- [COTTON96].

© Gilb@acm.org 1999

Early simple proof of concept (Ericsson)

- ***"Organic integration [Evo] is a way of getting rid of the myth [that problems don't exist] very early on.***
- ***You could say that organic integration demands of an organization that it do the specifications, the system, the design and the verification for one first very small task very quickly.***
- ***It also demands of the organization that it do this right in terms of delivering products correctly.***
- ***If the organization cannot even manage its first simple task in the time agreed, it certainly should question the ability to manage more difficult tasks.***
- ***This process of questioning is very healthy. It may for example prevent the delusions of grandeur so common in nearly all organizations".***
- ***[Ericsson94], page 26, Jack Järkvik, in the context of building mobile telephone base stations***

© Gilb@acm.org 1999

Steps to Users (JPL)

- ***Evo "involves a series of incremental deliveries.***
- ***Each delivery contributes an operable, functionally valuable, partial system.***
- ***The overall system is developed and delivered to its <u>users</u> (and thereby contracturally delivered to its sponsor) in small evolutionary increments.***
- ***The <u>users</u> employ the evolving system in the daily conduct of their mission."***
- [SPUCK93], Jet Propulsion Labs, JPL, on Rapid Development Method RDM hereafter called 'Evo'.

© Gilb@acm.org 1999

Upfront Requirements (JPL)

- "We assert that, in the case of an important class of systems, namely those that *automate human functions*, it is unreasonable if not impossible to expect system users or operators to be able to state Final Operating Capability requirements up front.
- An evolutionary approach is essential.
- This is true because staff functions change, user insight into operations increases, and concepts of operation are modified by the introduction of automation.
- Further, needs that are rejected as impossible, beyond the existing technology base, or simply heretofore inconceivable under Conventional Development Methods often are perceived as achievable at some point under [Evo]."

- [SPUCK93]

© Gilb@acm.org 1999

Synch and Stabilize (Ms)

- "We have labeled Microsoft's style of product development the 'synch-and-stabilize' approach.
- The essence is simple: continually *synchronize* what people are doing as individuals and as members of different teams, and periodically *stabilize* the product in increments
- – in other words, *as the project proceeds*, rather than once at the end."

- CUSUMANO95 , 14

© Gilb@acm.org 1999

User Feedback (JPL)

- Evo "expects active feedback from the experience gained from one incremental delivery to the requirements from the next.
- As Evo periodically delivers to the users an increment of capability, the users are able to provide understanding of how effectively that delivery is meeting their needs.
- As the users assess the impact of a delivery on their operations, the system developer is able to work with them to adjust the system requirements to better satisfy their operational needs.
- Evo lets that adjusted set of requirements be the basis for all subsequent incremental deliveries.
- This feedback process is formal and proactive. It is a key element in making Evo effective from a user's perspective."

- [SPUCK93]

© Gilb@acm.org 1999

Milestone Control (Ms)

- "Mike Conte, a senior program manager for Office
- "We actually break our development into three separate milestones.
- They might be six week milestones, [or] they might be ten-week milestones …
- At the end of the milestone our goal is to get all the features for that milestone that have been implemented … for that milestone at zero bugs….
- And then, when we get to the point where we get to 'ship quality', we can move on to the next milestone.
- The point of this is that we never get so totally out of control that we're at the end of a project and we have so many thousands of bugs that we can't ever tell when we're going to finish it."

- CUSUMANO95 , page 200

© Gilb@acm.org 1999

Cycle Length: short! (HP)

- ***The general rule of thumb is to keep the cycle length as short as possible. Within Hewlett-Packard, projects have used a cycle length as short as one week and as long as four weeks. The typical cycle time is two weeks.***
- ***The primary factor in determining the cycle length is how often management wants insight into the project's progress and how often they want the opportunity to adjust the project plan, product, and process.***
- ***Since it is more likely that a team will lengthen their cycle time than shorten it, it is best to start with as short a cycle as possible.***
- [COTTON96]

© Gilb@acm.org 1999

Early Sales Impact (HP)

- ***"Evo allows the marketing department access to early deliveries, facilitating development of documentation and demonstrations.***
- ***Although this access must be given judiciously, in some markets it is absolutely necessary to start the sales cycle well before product release.***
- ***The ability of developers to respond to market changes is increased in Evo because the software is continuously evolving and the development team is thus better positioned to change a feature set or release it earlier."***
- [MAY96]

© Gilb@acm.org 1999

Major Milestone Releases (MS)

- "Major Milestone Releases. A project organizes the development phases around three or four major internal releases, or 'milestone subprojects'.
- Microsoft intends these releases to be very stable. Theoretically, projects could ship them to customers,
- as Chris Peters observed "… What you do is you essentially divide a project into three [or so] pieces … and you pretend to ship the product after each piece."
- <-- CUSUMANO95 , page 200

© Gilb@acm.org 1999

Uncertainty Drives Evo (DOD)

- ***"When requirements uncertainty indicates an Evolutionary Acquisition approach, the program may involve little or no advanced development.***
- ***In contrast, when technological uncertainty indicates an evolutionary approach, significant amounts of advanced development are ordinarily involved.***
- ***Indeed, the evolutionary strategy has been derived as a means of dealing with just such uncertainties because development periods involved in making very large or "revolutionary" jumps at the limits of a state-of-the-art take so long and are so risky that U.S. readiness is being threatened.***
- [DODEVO95] quoting from other sources

© Gilb@acm.org 1999

Techno Risk/ User Risk (DoD)

- "While it is highly desirable that users be constantly knowledgeable about programs with technological uncertainty
  - _ indeed play a continuous, if reactive role in the acquisition of any DoD system
  - - the approach for these programs does not require user acceptance of any significant responsibility at any stage of the acquisition cycle.
- In contrast, for programs with requirements uncertainty,
  - succeeding blocks of work after the first cannot be adequately specified
  - until feedback from some user is received on the usefulness and needed modifications to prior blocks."
    - [DOD EVO 95] quoting from other sources

© Gilb@acm.org 1999

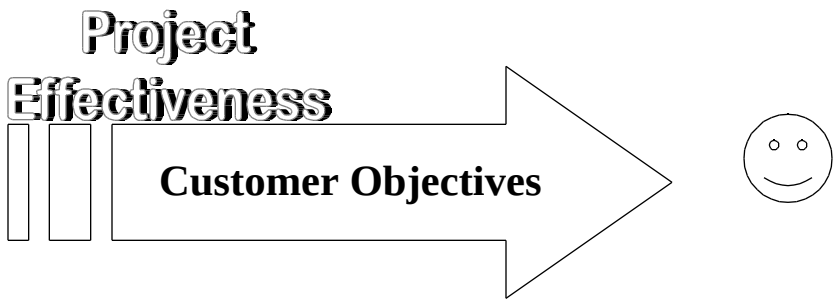
Evo shortens project by feedback (MS)

- "It appears that this incremental approach takes longer, but it almost never does, because it keeps you in close touch with where things really are"
- Brad Silverberg, Sr. VP for Personal Systems Microsoft in CUSUMANO95 , page 202

© Gilb@acm.org 1999

Evo acknowledges only one measure of effectiveness**.**

- It is 'measured progress towards defined **target** goals at the **recipient** level'.
  - User improvement.
  - Customer improvement.
  - As viewed by *them*, and *their* formal agenda.



© Gilb@acm.org 1999

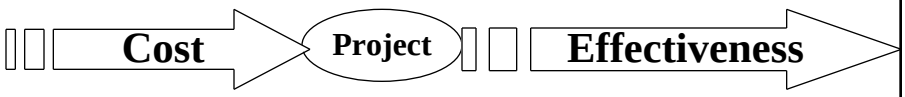
Costs of Evo (HP)

- ***"Adopting Evolutionary Development is not without cost.***
- ***It presents a new paradigm for the project manager to follow when decomposing and planning the project,***
- ***and it requires more explicit, organized decision making than many managers and teams are accustomed to.***
- ***In traditional projects, subsystems or code modules are identified and then parceled out for implementation. As a result, planning and staffing of large projects were driven by the structure of the system and not by its intended use.***
- ***In contrast, Evolutionary Development focuses on the intended use of the system. The functionality to be delivered in a given cycle is determined first. It is common practice to implement only those portions of subsystems or modules that support that functionality during that cycle.***
- ***This approach to building a work breakdown structure presents a new paradigm to the project manager and the development team. Subsystem and module completion cannot be used for intermediate milestone definition because their full functionality is not in place until the end of the project. The time needed to adopt this new paradigm and create an initial plan can be a major barrier for some project teams."*** [COTTON96] HP Journal August 1996 (web available)

© Gilb@acm.org 1999

The Evo measure of project 'efficiency' is targeted *results in relation to resources needed to deliver them.*

- The secondary measure of a project is **efficiency**.
- This is the effectiveness (delivered recipient-stipulated results) in relation to all 'resources consumed' to deliver the results.
- This is a measure of project profitability, or of return on investment.



© Gilb@acm.org 1999

Mills on Project Control

- ***"Software Engineering began to emerge in FSD (IBM Federal Systems Division, from 1996 a part of Lockheed Martin Marietta) some ten years ago [about 1970] in a continuing evolution that is still underway.***
  - ***Ten years ago general management expected the worst from software projects – cost overruns, late deliveries, unreliable and incomplete software.***
  - ***Today [1980] , management has learned to expect on-time, within budget, deliveries of high-quality software.***
- ***A Navy helicopter ship system, called LAMPS, provides a recent example.***
  - ***LAMPS software was a four-year project of over 200 person-years of effort,***
  - ***developing over three million, and integrating over seven million words of program and data for eight different processors distributed between a helicopter and a ship,***
  - ***in 45 incremental deliveries.***
  - ***Every one of those deliveries was on time and under budget.***
- ***A more extended example can be found in the NASA space program,***
  - ***where in the past ten years, FSD has managed some 7,000 person-years of software development, developing and integrating over a hundred million bytes of program and data for ground and space processors in over a dozen projects.***
  - ***There were few late or overrun deliveries in that decade, and none at all in the past four years."*** ***Harlan Mills [IBM80, page 415].***

© Gilb@acm.org 1999

Chapter 1 Requirements

The main point of difference between Evo and other project management methods is that ‘requirements achievement’ dominates, rather than formal processes (such as ‘approve design’, or ‘conduct field trials’).

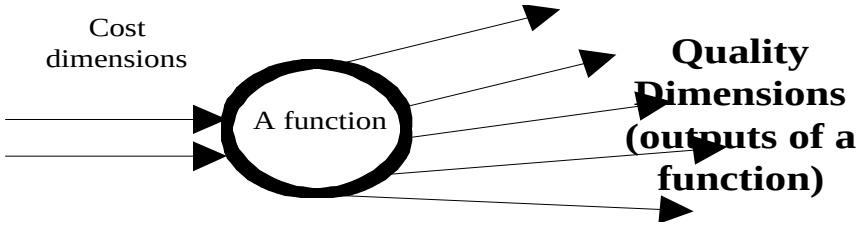
© Gilb@acm.org 1999

Failure due to goals (Kaplan, IBM)

- “We are continually amazed at how many managers fail to achieve results simply because their employees don’t understand the desired goal state.” [Kaplan94, p.203] IBM STL

© Gilb@acm.org 1999

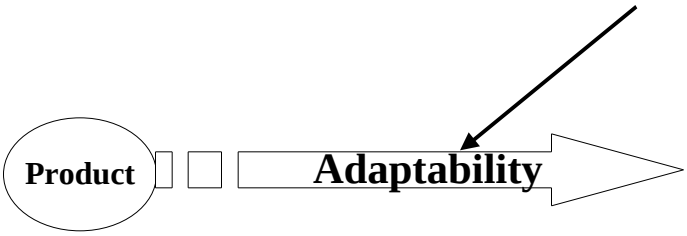
‘quality’ is any variable result from a system



© Gilb@acm.org 1999

Adaptability, simple goal

- **Adaptability:**
- **Scale: Engineering Hours needed to modify product for a new market**
- Plan [Product XX, Market YY] 1,000 Eng. H.



© Gilb@acm.org 1999

Usability Goal, Complex example

Usability:

Gist: the relative ease of learning and using a defined product compared to previously used products.

Scale: average minutes per [defined User type] to learn to use [defined Tasks to use the product].

Meter: at least 30 users of representative defined User type will be monitored doing at least 10 defined Tasks of defined function type.

Past [Old Product PP, Home Buyer, Adult, Task: Build telephone number list] 30 minutes.

Record [MM, Adult, Task: Dial Out] 10 seconds ← Consumer Reports, January

Trend [US Market, Children, Mix] 20 minutes ← Market Analysis Feb.

Wish [Our Customers, Mix] 5 minutes ← Chairman's Dream in Speech

MinLevel: Must [Our Customers, Mix, New Product] 10 minutes ← marketing minimum

Plan [Our customers, Mix, New Product, First Field Release] 50% of MinLevel? ← Guess by Project Mgr., [within 2 years of First Field Release] 30% of MinLevel ←Guess.

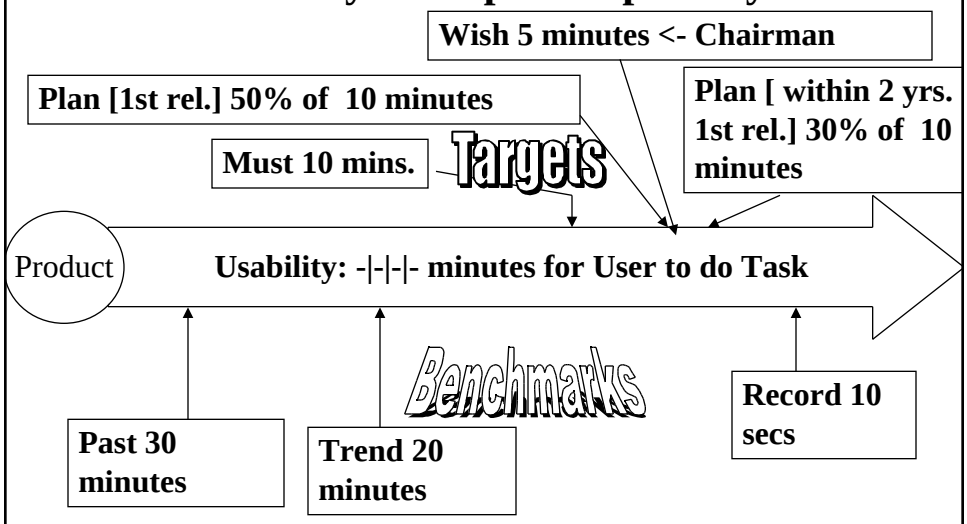
Local Definitions of Terms.

Mix: Defined: representative mix of common frequent user tasks.

User: Defined: person who intends to use the product in the long term, not a test person.

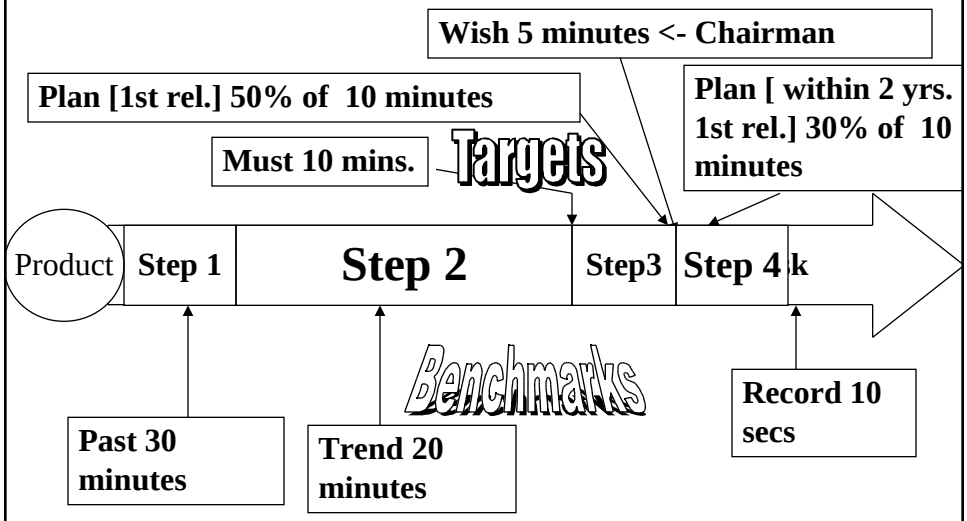
First Field Release: Defined: First sold releases to any public market after Field Trials.

Usability Example Graphically



© Gilb@acm.org 1999

Usability Evo Delivery



© Gilb@acm.org 1999 55

Requirements Control (DoD)

- ***"Change to functional requirements (especially additions to current requirements) can be controlled by accepting only very important changes.***
- ***The philosophy of permitting only crucial requirement changes is essential because:***
- **Feedback on effectiveness and suitability from actual operations and maintenance is almost always required to determine the value of proposed changes with any degree of certainty.**
- **For programs with short times between development increments, deferring requirements changes until the next program increment might be a better course of action because it preserves schedule and does not place delivery and fielding plans at risk.**
- **However, preserving schedule is of little value if feedback indicates an inability to meet or sustain specified performance thresholds or a lack of logistics supportability."** [DODEVO95]

© Gilb@acm.org 1999 56

Fixing Project Resource (MS)

- "Microsoft managers also try to 'fix' project resources - limiting the number of people and amount of time on any one project.
- The fixed project resources thus become the key defining elements in a product development schedule; in particular; the intended ship date causes the whole development team to bound its creativity and effort.
- The team must define intermediate steps and milestones that work backward from the target ship date, and co-ordinate product delivery with other Microsoft projects, product distributors, and third-party system integrators.
- Projects accomplish these goals even though the intended ship date almost always changes."
  - CUSUMANO95 , 188-9

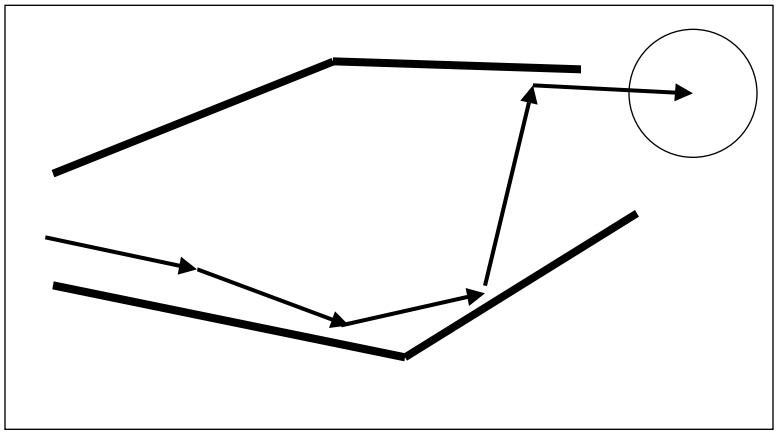
© Gilb@acm.org 1999

IBM FSD on ITERATIVE DESIGN-TO-COST, 1980

- **"Design is an iterative process in which each design level is a refinement of the previous level. At each stage, design and cost alternatives are examined. Those that best satisfy the project objectives are prepared for review and selection by the project sponsor.**
- **If no alternative fits the cost target, several courses of action are available.**
  - **The most common one is to go back to the designer and ask for a less costly, and perhaps a less attractive design.**
- **If the target has been missed by a large amount – and cost is critical - redesign may not produce an answer.**
  - **In this case the sponsor has to consider giving up some of the planned capability of the system.**
- **Otherwise he has to recognize that the capability cannot be acquired without increasing the cost target.**
- **The design [to-cost] process is followed until the program design for a specific software increment has been completed. From that point, development of each increment can proceed concurrently with the program design of the others.**
- **When the development and test of an increment are complete, an estimate to complete the remaining increments is computed.**
  - ***The algorithms used in this computation should reflect the various actual productivity rates experienced in developing and testing previous increments.***
  - ***An alternative plan is prepared and reviewed, as previously described, whenever a cost projection is inconsistent with its cost plan….***
  - ***The design-to-cost practice describes the management control procedures that balance cost, schedule, and functional capability." Robert Quinnan, [IBM80, page 474]***

© Gilb@acm.org 1999

Constraints



© Gilb@acm.org 1999

Arbitrary Delivery Steps (JPL)

- **"The delivery schedule is generally quite arbitrary.**
- **It is determined a priori that deliveries will occur .. at whatever… interval .. the project and sponsor select. …**
- **This approach might be called 'design to schedule' because only those functions that can be implemented in the allocated time are candidates for the next delivery."**
- **[SPUCK93]**

© Gilb@acm.org 1999

2: Design: The Evo 'Means' to the Target 'ends'.

© Gilb@acm.org 1999

Design Analysis Process (HP)

- ***"Because some design issues are cheaper to resolve through experimentation than through analysis,***
- ***Evo can reduce costs by providing a structured, disciplined avenue for experimentation."***
- [MAY96]

© Gilb@acm.org 1999

Fixed Budget, Variable Build (JPL)

- "In contrast to conventional project management,
  - the overall budget for an Evo project is taken as a given,
  - and the Evo budget process is closely analogous to the scheduling process;
  - it is 'build to cost' "
    - [SPUCK93]

© Gilb@acm.org 1999

Evo forces organized Project decision-making (HP)

- ***Many development teams lack a well-defined, efficient decision-making process. Often they make decisions implicitly within a limited context,***
  - ***risking the compromise of the broader project goals and slowing progress dramatically.***
- ***Evolutionary Development forces many decisions to be made explicitly in an organized way,***
  - ***because feedback on the product is received regularly***
  - ***and schedules must be updated for each implementation cycle.***
- ***The continual stream of information that the project team receives must be translated into three categories of decisions:***
  - ***changes to the product as it is currently implemented,***
  - ***changes to the plan that will further the product implementation, and***
  - ***changes to the development process used to develop the product.***
- ***Fortunately, because of Evo's short cycle time, teams***
  - ***have many opportunities to assess the results of decisions and adjust accordingly***
    - ***[COTTON96]***

© Gilb@acm.org 1999

Requirements Refinement (JPL)

- "a key benefit … is its ability to progressively refine requirements
  - and to respond easily to the refinements.
- Refinement is done on the basis of
  - developmental test,
  - training, and
  - operational experience.
- Requirements feedback facilitates working in an environment of change."
  - [SPUCK93]

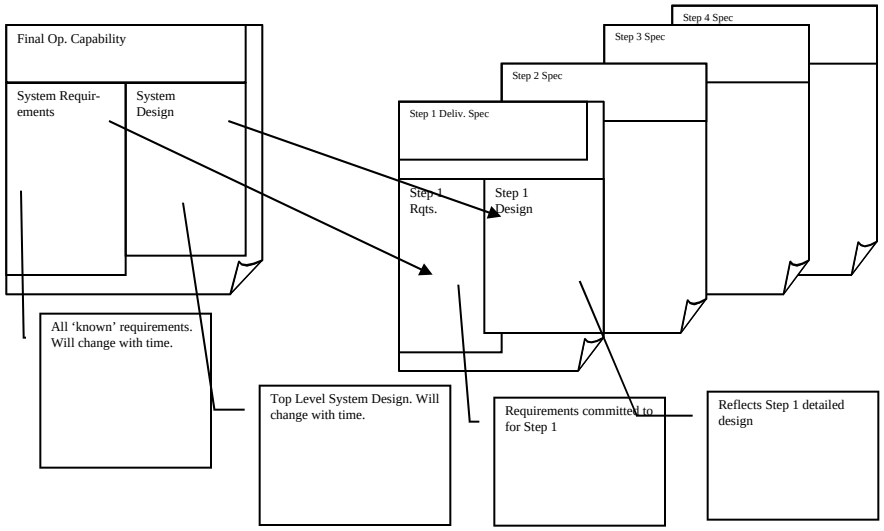
© Gilb@acm.org 1999

Final and Step Specification (JPL)

- "The [step] delivery specification is required to be much more complete and specific than the Final Operational Capability (FOC) specification,
  - although good [Evo] practice suggests that any requirements or design information that is known at any point in time be captured in the FOC specification regardless of the delivery in which it will be implemented.
  - Also the FOC specification is kept current with the [step] delivery specification so that the FOC specification evolves with time to become the complete 'as built' specification." [SPUCK93]

© Gilb@acm.org 1999

System Specs and Step Specs (JPL)



© Gilb@acm.org 1999

Users feedback to next cycles (HP)

- ***"In parallel with the development activities of the team,***
  - ***selected users or customers of the system are working with and***
  - ***providing feedback on the release from the previous cycle.***
- ***This feedback is used to adjust the plan for the following cycles."***
- [COTTON96]

© Gilb@acm.org 1999

Open-endedness Quantified

- **Adaptability:**
- **Gist: the ability of our system to easily tolerate unexpected changes. The set of all other ease-of-change qualities below { Extendibility, Portability, Serviceability }.**

- **Extendibility:**
- **Scale: the engineering effort needed to add [defined capacity] to the product.**
- **Plan [memory by factor 10] less than 10% of cost of memory itself.**

- **Portability:**
- **Scale: the engineering effort needed to move [defined system elements] to [defined target environments] using [defined tools or skilled people or processes].**
- **Plan [software logic and data, East Asian Markets, Average Programmers] 1 hour per 100 lines of code.**

- **Serviceability:**
- **Scale: The ease of giving [defined service types] in [defined service locations] by [defined levels of service people].**
- **Plan [Shop Counter, Major Chains, Certified Trained Specialists] 90% Service Cases within 30 minutes "in shop wait".**

© Gilb@acm.org 1999 69

Open-endedness Design Spec example

- **JAVA: Use Java programming Language → Portability.**
- **IEEE675: Use the IEEE 675 Interface Specification → Extendibility.**
- **Self Test: Build all components hardware and software with thorough self test and defect reporting capability in fully automatic mode →Testability & Adaptability.**
- **Accessories: Use the Corporate Product Line Interface for all accessories, or at least include necessary plugs, cables and software with each product to enable interface to it →Adaptability.**
- **Display: All design of displays will assume that future displays can be of any size and dimension both smaller and larger than initial releases →Adaptability.**

© Gilb@acm.org 1999 70

3: Impact Tables: The Evo Accounting and Planning Mechanism

© Gilb@acm.org 1999

Impact Tables Answer Us:

- **How much will a particular design idea impact the step goals?**
- **Which ideas should be *selected* for this *particular* step?**
- **What is the expected outcome of implementing this step in terms of quality improvements and costs?**
- **What is the *uncertainty* of the step implementation, and what is worst case results?**
- **What is the expected impact of a *planned series* of steps on our objectives?**
- **What was the cumulative impact of the *past series* of steps on our objectives**
- **What was the impact of the *last* step on our objectives?**
- **What were the things we understood least on the past step?**

© Gilb@acm.org 1999

Step Comparison Table

	Step Candidate **A**: {Design-X, Function-Y}	Step Candidate **B:** {Design Z, Design F}
Reliability 99%-99.9%	50%	100%
Performance 11sec.-1 sec.	80%	30%
Usability 30 min.-30 sec.	-10%	20%
Capital Cost 1 mill.	20%	5%
Engineering Hours 10,000	2%	10%
Performance/Capital Cost Ratio	80/20= *4.0*	30/5= *6.0*
Quality/Cost Ratio	120/22=**5.46**	150/15=**10.00**

© Gilb@acm.org 1999

Step Risk Analysis

	Step Candidate **A**: {Design-X, Function-Y}	Step Candidate **B:** {Design Z, Design F}
Reliability 99%-99.9%	50%±50%	100%±20%
Performance 11sec.-1 sec.	80%±40%	30%±50%
Usability 30 min.-30 sec.	-10%±20%	20%±15%
Capital Cost 1 mill.	20%±1%	5%±2%
Engineering Hours 10,000	2%±1%	10%±2.5%
Worst Case B/C ratio (1 to 3)	(0+40-10)/(21+3) = *1.25*	(80-20+5)/(7+12.5) = *3.33*
Best Case B/C ratio	(100+120+10)/(19+1) = **11.5**	(120+80+35)/(3+7.5)= **22.38**

© Gilb@acm.org 1999

Step Choice with 'Credibility'

	Step Candidate **A**: {Design-X, Function-Y}	Step Candidate **B:** {Design Z, Design F}
Reliability 99%-99.9%	50%±50%	100%±20%
Performance 11sec.-1 sec.	80%±40%	30%±50%
Usability 30 min.-30 sec.	-10%±20%	20%±15%
Capital Cost 1 mill.	20%±1%	5%±2%
Engineering Hours 10,000	2%±1%	10%±2.5%
Worst Case B/C ratio	(0+40-10)/(21+3) =*1.25*	(80-20+5)/(7+12.5) =*3.33*
æWorst WorstA case considering estimate credibility factor	0.8 x 1.25= ***1.00***	0.2 x 3.33= ***0.67***

A
Credibility=0.8
(High)

B
Credibility=0.2
(Low)

© Gilb@acm.org 1999

Building Up an Evo step to a 'biggest allowable bet' threshold.

	Design: Dual [USA]	Design Dual [Europe]	Next Step **Maximum** at the 2% level	Next Step *Totals* Before looking at possible additional Design ideas
Capital Investment	150,000±100,000	10,000 ±5,000	**200,000**	*180,000*
Project Calendar Time	0.1 to 0.5 weeks	0.05 week	**1 week**	*0.9 week*
Engineering Hours	20 ±10 hours	10±5 hours	**150**	*130 hours*

You can see, from this constructed example, that implementing the design idea 'Dual' in 'Europe' will work if added to the current step 'bucket'. But, that the same design idea in 'USA' would be consistently 'over the top' of the allowable risk. That one would be fine to think about for the next step after this.

© Gilb@acm.org 1999

An example of a typical one-week Evo cycle at the HP Manufacturing Test Division during a project. [MAY96]

	Development Team	Users
Monday	• System Test and Release Version N • Decide What to Do for Version N+1 • Design Version N+1	
Tuesday	• Develop Code	• Use Version N and Give Feedback
Wednesday	• Develop Code • Meet with users to Discuss Action Taken Regarding Feedback From Version Nû1	• Meet with developers to Discuss Action Taken Regarding Feedback From Version Nû1
Thursday	• Complete Code	
Friday	• Test and Build Version N+1 • Analyze Feedback From Version N and Decide What to Do Next	

© Gilb@acm.org 1999

Scaling to Funds Available (JPL)

- **"[Evo] adapts easily to changing programmatic environments of which … budget changes …are typical. ….**
- **Each succeeding delivery can be scaled to funds available……**
- **[Evo] has an excellent immune system to normal funding variations**
- **and finds little challenge in providing meaningful increments of capability under most real-life funding scenarios."**
- **[SPUCK93, 7.2]**

© Gilb@acm.org 1999

Direct Customer Input (MS)

- "Microsoft's general philosophy has been to ….. focus on evolving features and whole products incrementally, with direct input from customers during the development process." CUSUMANO95 , 13

© Gilb@acm.org 1999

WBS and Dependency (HP)

- ***"Although it is difficult and time-consuming, the work breakdown structure and dependency information must be done and done correctly." [MAY96]***

© Gilb@acm.org 1999

Schedule Risk Steps Early (HP)

- ***"Some of the criteria commonly used in setting priorities during this initial planning activity are …:***
  - ***Features with greatest risk.***
- ***The most common criterion used for prioritizing the development phase implementation cycles is risk.***
- ***When adopting [new] technology, many teams are concerned that the system performance will not be adequate.***
- ***Ease-of-use is another common risk for a project.***
- ***The [step content] that will provide the best insight into areas of greatest risk should be scheduled for implementation as early as possible."***

[COTTON96]

© Gilb@acm.org 1999

Impact Table for Step Management

	Step #1 **A**: {Design-X, Function-Y}	Actual	Difference. - is bad + is good	Total	Step #2 **B:** {Design Z, Design F}	Actual	Difference	Total	Step #3 Next step plan
Reliability 99%-99.9%	50% ±50%	40%	-10%	40%	100% ±20%	80%	-20%	120%	0%
Performance 11sec.-1 sec.	80% ±40%	40%	-40	40	30% ±50%	30%	0	70%	30%
30 min. -30 sec.	10% ±20%	12%	+2%	12%	20% ±15%	5%	-15%	17%	83%
Capital Cost 1 mill.	20% ±1%	10%	+10%	10%	5% ±2%	10%	-5%	20%	5%
Engineering Hours 10,000	2% ±1%	4%	-2%	4%	10% ±2.5%	3%	+7%	7%	5%
Calendar Time	1 week	2 weeks	-1week	2 weeks	1 week	0.5 weeks	+0.5 wk	2.5 weeks	1 week

4: Evo Planning: How to specify an Evo Project plan.

© Gilb@acm.org 1999

Simple Evo Plan.

- **Build customer files and produce published customer lists, replacing old style ones.**
- **Build product files and pricing information, replace old price catalogues.**
- **Take orders and produce a warehouse article picking list.**
- **Produce invoices for simple frequent cases.**
- **Produce more complex invoices with special discounts.**
- **Produce invoice reminders.**
- **Accept payment information and produce differential invoices.**
- **Do accounting for all invoices**
- **All other steps symbolized as included in this rough last one.**

© Gilb@acm.org 1999

Prioritizing Essentials (Ericsson)

- *"Customers do not like disappointments and broken promises.*
- *Promising the earth and then failing to deliver is not a good way of treating your customers.*
- *It means you have to negotiate constant upgrades in order, some time in the future, to achieve what was originally promised.*
- *It is far better to deliver something which just about meets the specification, and then sell enhancement packages, i. e.. additions.*
- *When every bit costs money, you only add what is really needed. At the center of this discussion … lies the importance of [us] understanding which features and functions are essential for our customers to be able to do what they intend to with our products.*
- *In the final analysis it is to make money.*
- *No amount of [product simplification] should ever threaten this.*
- *This is precisely the main reason why we must always, always be able to deliver on time.*
- *And exactly the same reason why we must always put basic functionality before special features.*
- *Even to the extent that special features should often be excluded altogether since otherwise there is a risk that they may threaten the product's basic functionality.*
- *Few people want to admit that this is in fact often the case."*
- **[Ericsson94], page 49-50, Jack Järkvik, in the context of building mobile telephone base stations**

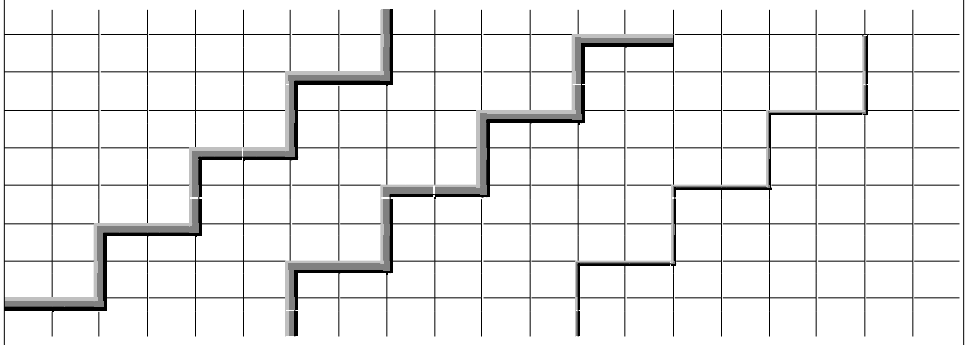
© Gilb@acm.org 1999

An Outliner Evo Plan

- **Project Evo Plan**
- **Rest of This Year**
- **November**
- **Week 45**
- **Deliver Change of Priority for Top Brass**
- **Week 46**
- **Increase Performance by at least 10%**
- **December**
- **All remaining performance improvements**
- **Next Year**
- **First Half**
- **Focus on Quality Improvement targets**
- **Second Half**
- **Focus on Customer Service Improvement**
- **All remaining Years**
- **Content and sequence dependent on what has been learned and**
- **what has happened until this time.**

© Gilb@acm.org 1999

The Step Plan Template



© Gilb@acm.org 1999

A multilevel Evo Plan

	Mega-	**steps**			2003 Step 6	?	**Mega-**	**Steps**	Rest of year 1.9	? rest tune	
				2002 Step 5	?			May 1.5	Bottle necks		
			2001 Step 4	Speed			April 1.4	Priority		1.1.4	Wk. 4
		2000 Step 3.	Capa-city			March 1.3	Nodes		1.1.3	Wk. 3	Clea n out
	1999 Step 2.	Usab-ility			Feb. 1.2	Com munic		1.1.2	Wk. 2	Auto tune	←
1998 1.	**Perfor-mance**	→	→	Jan. 1.1	**Tune Data base**	→	→	Wk. 1 1.1.1	**Reorg-anize**	←	***Deli-very steps***

© Gilb@acm.org 1999

Ladder Format

Step Name tag	*Description (Design Tag)*	*Other related information*
User Enhance	{Interact, Intelligent}	9/1 (Value/Cost ratio)
Port	{Convert Code [UNIX], Convert Data [Database, Tables]}	8/3
Miniaturize	{Mini Phone, Field Trial}	*Depends on Port 7/3*
Retail	{Market, Distribute, Discount}	5/1
Europe	{European Community, EEconC}	2/1 Do by Country Size, Small up

© Gilb@acm.org 1999

Operational vs. Earned Value (JPL)

- "For Conventional Development Methods (CDM), money is invested steadily in order to derive value only at the final delivery, Final Operating Capability(FOC).
- That is, under CDM there is no value to users until FOC.
- In contrast, [Evo] makes the same investment and derives value to users incrementally, shortly after the investment.
- It is this characteristic of CDM that has prompted sponsors to impose performance measurement systems (PMSs) that aggregate artfully composed measures of 'earned value' for accomplishment of the various processes of the CDM life cycle.
- [Evo] substitutes operational capability or delivered value to earned value."
- [SPUCK93]

© Gilb@acm.org 1999

Impact Table Top Level Plan

	Host System base	Mega-step **Next Year**	Mega-step **Jan Next**	Delivery step **Convert**	**Convert** Impact %	**Jan Next** Impact %			
Q1 99% to 99.9%	99.00%	99.5%	99.05%	99.05%	6%	6%			
Q2 10 min-1 min.	10 min.	5 min.	9 min.	10 min.	0%	11%			
Ca 0-$2mill.	0	$1,000,000	$100,000	$20,000	1%	10%			
Cb 0-20,000 Eng., hours	0	10,000 hours	5,000 hours	1,000 hours	5%	25%			
Q/C					6/6=1	17/35=0.49			

© Gilb@acm.org 1999

System Spec and Step Index (JPL)

- "Some [Evo] projects have very successfully implemented the notions with only a Final Operating Capability specification and a tabular appendix to the specification, listing which specification provisions are to be (or were) implemented at each delivery" [SPUCK93] JPL, CA, USA

© Gilb@acm.org 1999

Bill on Milestone Approach

- "the milestone approach is a major practice for us"
- Bill Gates in CUSUMANO95 , 18
- "Microsoft Secrets"

© Gilb@acm.org 1999

5: Evo Step Objectives: Cycle Requirements

© Gilb@acm.org 1999

‘Release Practice’ Makes Perfect (HP)

- ***“A second concern is that it will be too difficult to make so many releases.***
- ***If it is difficult to make one release every 9 to 18 months, how much more difficult will it be to release every two weeks?***
- ***The answer is that when you make frequent releases, you get better at it (if this is not the case, EVO becomes too inefficient).***
- ***Further, the small chunks in each cycle keep things to a manageable size.”*** [MAY96]

© Gilb@acm.org 1999

A Step with Quality at Top (MS)

- “When we reach milestones, we have not just functionality but
  - size,
  - performance requirements and
  - quality requirements. …
- So that milestone isn’t met until [then]…”

- Brad Silverberg, Sr. VP for Personal Systems Microsoft in CUSUMANO95 , page 202

© Gilb@acm.org 1999

Leverage in Early Steps (JPL)

- **"The first maxim is the '80/20 rule'.**
- **The 80/20 rule observes that 80 % of results can be achieved with 20 % of an effort; it takes the remaining 80 % of effort to accomplish the remaining 20 percent of results.**
- **Under [Evo] the general guideline is to apply the 20 % of effort to accomplish the 80% of results and to be satisfied with that until the next delivery…..**
- **The second maxim is 'just-in-time engineering' ……**
  - **Requirements are not completed until design work must be initiated.**
  - **Designs are not complete until shortly before they are to be implemented.**
  - **In particular, requirements and designs for a future delivery are postponed until work on that future delivery begins.**
  - **In general, engineering is completed just in time for the need.**
  - **Again, this maxim must not be pursued to absurdity.**
  - **In particular, inasmuch as the products of [Evo] engineering processes evolve throughout the successive deliveries, engineering data should be permanently deposited in these products as they become available." [SPUCK93, 6.3]**

© Gilb@acm.org 1999 97

More Project Management Decisions

- ***"Because many more project management decisions need to be made in Evo,***
  - ***handling decisions can also become a problem.***
- ***If the decisions are not timely or cause dissension, progress can be delayed.***
- ***Participatory decision-making techniques have been one solution at HP."***
- [MAY96]

© Gilb@acm.org 1999 98

Use this recipe for starters

- **Decide on one required quality objective which the recipient would most value improvement in (example 'Reliability').**
- **Decide on an interesting minimal increment for the recipient. Use this as Must level.**
- **Decide what might be accomplished by the best technology you can insert in the next cycle. Use this as a Plan level. *This technology will normally be extracted from the system architecture specification.***
- **Specify the technology you believe will get you the increase (in the Step specification). More on *choosing* the technology in next chapter.**
- **Estimate that resources needed to implement *that* technology. Put this in the step plan.**
- **If the resources exceed those available or permitted by step planning policy, go to step 2 and 'adjust'.**
- **Estimate the impact of the step technology on all *other* qualities. Document your estimates. Example: in an Impact Table.**

© Gilb@acm.org 1999 99

A Planning Impact Table

	Thus Far	Next Step est.	Worst Case est.	Risks	Long term target
Reliability	99.10%	99.20%	99.00%	NewTech =???	99.99%
Usability	5 minutes	5 minutes	=	none	1 minute
Budget	50,000	10,000	20,000	Weekend work	100,000
Calendar Time	300 days	5 days	10 days	Weather	300 days left

© Gilb@acm.org 1999 100

Analyzing Step Cost Types

Step Costs	Design idea costs	Functionality costs	**Acquisition** Costs	**Integration** and testing costs	**Deployment** costs
Budget	$20,000	$1,000	$2,000	$3,000	$4,000
Calendar Time			1 day	6 days	3 days
Engineering hours (Eh)			20 Eh	24 Eh	12 Eh
Post deployment costs/year oper.	$1,000/year	0	0	$1,000	

© Gilb@acm.org 1999

Early Use During Development (JPL)

- ”For example, comments received from document reviews can be incorporated in the update of the document emanating from the next delivery:
- thus, many typical document iterations are eliminated.
- The chance to use a system while it evolves and to fix deficiencies as they arise is simply not available under Conventional Development Methods” [SPUCK93]

© Gilb@acm.org 1999

6: Detailed Evo Step Design: Extracting function and design to make a step.

© Gilb@acm.org 1999

Microsoft Success Principles

- **"Microsoft is one of those rare firms that has sustained and extended its market power. We think two principles we cite in this chapter explain how.**
- **First, Microsoft frequently makes incremental improvements and occasionally introduces major advances in its products.**
- **While often doing little more than packaging many incremental innovations, these major changes make older product versions obsolete.**
- **With a continual cycle of incremental and occasionally more radical innovations, competitors have little opportunity to challenge the market leader.**
- **Microsoft has accumulated enormous financial and technical resources that enable it to sustain this level of R&D.**
  - **It has followed a somewhat unusual strategy:**
  - **Dominant firms generally hesitate to introduce new products that steal sales from their existing product lines." CUSUMANO95 ,129**

© Gilb@acm.org 1999

Competitiveness by ...

- ".. the synch-and-stabilize process … provides several benefits that serve Microsoft well …
- It facilitates competition based on
  - customer feedback,
  - product features, and
  - short development times by
  - providing a mechanism to
    - incorporate customer inputs,
    - set priorities,
    - complete the most important parts first, and
    - change or cut less important features." CUSUMANO95 , 17

© Gilb@acm.org 1999

CASE STUDIES

© Gilb@acm.org 1999

Do Evo!

Deliver results weekly to real users!

Case 1.
The Sales Manager System.

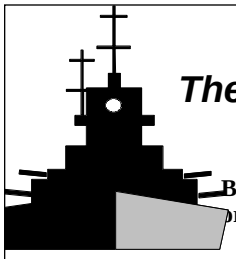
- • **Barrier: "We are not allowed by {somebody, Sales manager, Research Director} to contact our users or customers".**
- **About 1985 a U.K. Computer Manufacturer was making a product tailored for a new market, "Sales Managers". About 30 young technical experts were designing the product. None had sales experience of any kind.**
- **At an early stage I advised using weekly Evolutionary delivery to some real users. They ignored the advice. This was probably because I did not present my case with a concrete Evolutionary plan, I later learned. I made the mistake of giving advice in general terms to people who had no practical familiarity with the concepts.**
- **About a year later, 12 months into a projected 35 month project, they finally took one part of the advice. They invited a real sales manager from their own company to look at the emerging product prototype. He laughed at it. They realized they had to start again from scratch. Imagine if they had gotten him to laugh somewhat earlier!**
- **Not too long afterwards, I helped one of the team make a detailed multi-cycle plan with about 5 key result numbers for each Evolutionary cycle of delivery. We presented it to the project manager who up to then had said "no". He immediately warmed to the whole concept.**
- **"Why didn't you explain it this way before?", "I'd be very interested in using this method. I review the design every Friday. But all I review is non-working prototypes and more design documentation. I don't know if it will work or sell. What you are offering me is the chance to see a real live system being used by real sales managers every Friday. Let's do it!**
- **I felt I could have kicked myself for not presenting a more concrete results delivery plan earlier.**

© Gilb@acm.org 1999

Lesson Learned: Sales Manager Case

- **Make a concrete plan**
- **showing real results expected,**
- **especially for the first steps.**

© Gilb@acm.org 1999

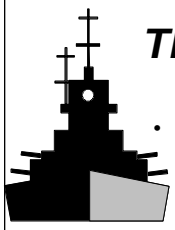


Case 2.
The Naval Weapons System. *Slide 1 of 2*

Barrier: "It cannot be done until the new {thing, building, organization, system}.... is ready in some years time".

- **British Naval Weapons System case: Once, when holding a public course on the EVO method in London, a participant came to me in the first break and said he did not think he could use this early Evolutionary method. Why? "Because my system is to be mounted on a new ship not destined to be launched for three years."**
- **I did not know anything about his system, at that point. But I expressed confidence that there is always a solution, and bet that we could find one during the lunch hour.**
- **He started our lunch by explaining that his weapons research team made a radar-like device that had two antennas instead of the usual one, which had their signals analyzed by a computer before presenting their data. It was for ship-and-air traffic, surrounding the ship it was on.**
- **I made a stab at the "results" he was delivering, and who his "customer was", two vital pieces of insight for making Evolutionary delivery plans. "May I assume that the main result you provide is 'increased accuracy of perception', and that your 'customer' is Her Majesty's Navy?" "Correct." He replied.**
- **"Does your 'box' work, more or less, now, in your labs?", I ventured. (Because if it did, that opened for immediate use of some kind) "Yes", he replied. "Then what is to prevent you from putting it aboard one of Her Majesty's current ships, and ironing out any problems in practice, enhancing it, and possibly giving that ship increased capability in a real war?" I tried, innocently.**
- **"Nothing!", he replied. And at that point I had won my bet, 20 minutes into the lunch.**

© Gilb@acm.org 1999



Case 2.
The Naval Weapons System. *Slide 2 of 2*

- **"You know, Tom", he said after five minutes of silent contemplation, "the thing that really amazes me, is that not one person at our research labs has ever dared think that thought!".**

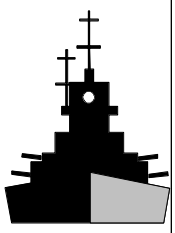
  The thing to notice here was that the customer was not the new ship, and that the project was not to put the electronics box on the new ship. The project was to give increased perception to the real customer, Her Majesty's Navy.
-
- **Notice the "method" emerging from this example:**
- **1. Identify the real customer, and plan to deliver results to them.**
- **2. Identify the real improvement results and focus on delivering those results to the real customer.**

- **in other words:**
- **1. Do not get distracted by intermediaries (the new ship)**
- **think "Her Majesty's Navy" or even "The Western Alliance".**
- **2. Do not get distracted by the perceived project product (the new radar device for the new ship):**
- **think "increased accuracy of perception".**

© Gilb@acm.org 1999

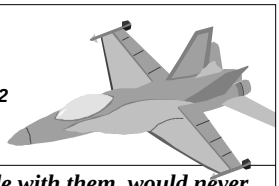
Lessons Learned: Naval Weapons Case

- **Evolutionary Projects are not normal thinking even amongst well educated engineers.**
- **Evo is a systems method not a software method**
- **Focus on 'evolving' the results of the project**
- **(increased accuracy of perception, not 'deliver a black box')**
- **Focus on your real customer (HM Navy, not a ship)**



© Gilb@acm.org 1999 111

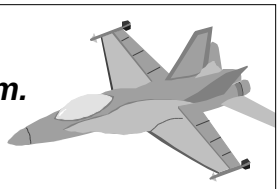
Case 3.
The Air Traffic Control System. *part 1 of 2*



- ***• Barrier: "Our customer, and the contract we have made with them, would never permit it."***
- ***Air Traffic Control System: In 1986-7 a Swedish client was building an air traffic control training system for another European government.***
- ***It was admittedly late. I was called in to help. I suggested we re-plan the project into a series of Evolutionary cycles, delivering the most vital ones earliest.***
- ***Fine, they said. One problem though. The client won't have it. They stick to their "all at once" delivery contract fanatically, like good bureaucrats.***
- ***"What would you want, if you were the customer?", I tried. The assembled executive team said they, of course, were rational people, and would prefer to get the system incrementally. But, they were sure the customer would have none of that nonsense.***
- ***Well, I had their attention for the rest of the day, and they were paying me. So, I asked if they would play the game of making an Evolutionary result delivery plan for the project with me, the way they would like it, if they were customers. The customer had a training building specially built. The hardware was ready, the "software" was late.***
- ***Our rough sketch included the design that the early stages should deliver enough of the system so that the instructors could begin to prepare a course for their students, and then, that simple air traffic situations were working before more exotic ones. They worked out a ten basic Evolutionary cycle plan that afternoon.***
- ***At the end of the day, I said: "Let's do it". They said. "Tom, you don't understand the mentality of our customer. There is no way they will accept this. They only think in terms of the contract as it is.***
- ***I asked them what it would cost them to phone the customer and ask what the customer wanted to do. They promised to try, no promises about results, after I left.***
-

Case 3.
The Air Traffic Control System.

part 2 of 2

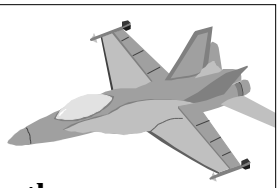


- ***What happened in later reality?***

- **A few weeks later, I was received back with almost a hero's welcome.**
- **The customer, to everyone's surprise, had accepted the evolutionary plan.**
- **The terrible deadline pressure was "off".**
- **The project could now concentrate on doing a quality job on the evo cycles.**
- **They finally did deliver, and the project was considered "not unsuccessful".**
-
- **Years later I met the main customer representative who made that decision, on my course in Berlin. At a lunch at a Yugoslavian (it was still there, so was the Wall) restaurant he told me: *"Of course we accepted the evolutionary plan. Do they think we are stupid?"***
- **Lesson: don't under-evaluate the customer's survival-intelligence, especially if you talk to the appropriate level of manager.**
- **Another lesson: evolutionary results delivery can be used to save large projects even after much initial effort has been placed in the wrong direction.**

© Gilb@acm.org 1999

Lessons Learned:
Air Traffic Control Case



- **Do not believe all the local experts,**
- **they are in trouble because they do not believe the answer.**

- **Get to the right level of management to get change decisions.**

- **Do not be afraid to use common sense: 'what would I want if I were the customer?'**

© Gilb@acm.org 1999

Case 4.
The Department of Defense Standard.

- ***Barrier: "The government insists on a "waterfall" method of doing projects. It is the standard".***
- ***This was certainly the case in the U.S. The Department of Defense Standard "2167" (now obsolete) apparently demanded a Waterfall Model (Straight line project development, requirements, design, build, test, deliver) as a Requirement. A lot of my clients acted as though the government required such a project management model (Big Bang or Waterfall Model).***
- ***I noticed that IBM's Federal Systems Division, (later 'Loral' then 'Lockheed') was unique in practicing an Evolutionary results delivery project management method since 1970. They were also unique in being successful in project management. "All projects using the method were on time and under budget". (Harlan Mills, IBM SJ 4/1980).***
- ***In 1991 I met a man in uniform, at a conference in Florida, who claimed that he had developed that standard (2167), and that it most certainly did not require the Waterfall method. He had had to correct that wrong opinion with prominent software engineering personalities who were spreading the wrong word.***
- ***The lesson? Don't take things on hearsay.* Even the government *likes to get things done on time and under budget. Any Standards were intended to help, not to stand in the way.***
- ***In 1990, I asked Harlan Mills, in Oslo, how they had come to develop their Cleanroom method, which included Evo. I said I guessed they had used the analogy of how rockets used process control to hit their target. He smiled and acknowledged that was part of it.***

Case 5. The German Telecommunications Company.

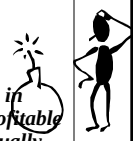
- **Barrier: "It is too late, we have already invested so much the old way, that we just have to see it through".**
- *At a large German telecommunications business in December 1984, about 985 software engineers had been working for three years on a major new world-market product. December 1984 was the deadline for delivery of the product, but their 40,000 node PERT chart, the Financial Director told me, estimated that they had 2 or 3 years more effort left. Corporate top management had given them one more year, to December 1985. Deliver, or forget the whole market, which by that time would be taken over by competitors.*
- *As usual, I suggested re-planning the project in smaller and critical increments first. They told me that this was unthinkable. The software was already written, they claimed, only testing remained. They also had a rather long list of other reasons why Evo would not work for them.*
- *Using common sense we worked out a basic Evolutionary scheme. The small-model software first (there were 35 signed contracts for it, none for the medium and large systems). Then fundamental telephone services before advanced fancy stuff.*
- *After what seemed like seven management layers of "you must present this to my boss", we ended up in the office of Herr Raab, The Project Director. He thought it was all good common sense, and stared coldly at his (cowardly, cautious?) sub-ordinates as he asked: "Can you do it this way?", (assenting nods) "Then do it!".*

- *They did too. By November 1985, on a return visit, they told me that the small systems had been operating for over six weeks with several real customers, with no problems whatsoever. Note, three months before the impossible deadline!*
- *As in many other cases, I had to spell out the basic steps myself. I had to make them obvious clear simple steps. I could not merely suggest the method. And then, in spite of the obviousness, I had to get to the right person to make a decision.*

16

Case 6. The European Chemicals Organization. part 1 of 2

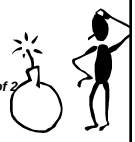
- **Barrier: "Sure, but it will take three years before we have enough basic system to start incrementing results in small cycles".**



- ***A 10,000 person strong division of a multinational European chemicals corporation was in trouble in 1981. Secretly (our planning team didn't know yet) the Board had said: Get profitable within three years or*** **we'll give the whole division a "golden handshake"** ***(they were actually quite generous). It was no idle threat. They had done such things a few years back.***
- ***The planning team of about a dozen people, half production planners, half computer specialists, was going to save the day by making a new production planning system which would permit much cheaper production.***
- ***They*** **(unknown to themselves)** ***had to reduce costs by $200 million annually (the current annual deficit) within 3 years maximum. That was their goal (and it was my job to help them clarify goals*** **- they thought the goal was to build a great new planning system!).**
- ***When the "real" ($200 million/annual cost reduction) goal was sensationally ferreted out of the woodwork (probably kept secret with good reason) I reflected on the fact that I had never heard of an improved production planning system, with such an effect, in the history of business.***
- ***I asked the team what was the best they thought they could do, in the best of worlds.***
- ***They replied that they could save between $10-40 million annually. At that point we ascertained, with the top manager, that we were the only planned effort! I told him that if he didn't mount several other parallel Evolutionary efforts, he might as well kill the Division instantly. He got that message.***

- ***(continued next slide)***

Case 6. The European Chemicals Organization. part 2 of 2



- ***Getting back to our "little effort", I suggested to the team that it was a bit daring to bet the workplaces of 10,000 colleagues (and 1/6 of the Corporation) on the fact that in exactly three years their marvelous system would save the day. But that "big bang" was their current plan.***
- ***I asked them if they would like to try to plan their project, so that the*** **most useful and profitable things were done early,** ***and in small Evolutionary deliveries. If nothing worked out, they would soon know there was no hope, and shut down.***
- ***If things worked out in the small, they could accelerate. They would have gained confidence that things would work out. They would actually be delivering*** **real savings years earlier** ***(even 5% of $200 million a year bears worrying about for most of us).***
- ***We met for a day and they developed an Evolutionary plan (I just tickled them at appropriate intervals). It had about seven hierarchical layers.***
- ***We got down to the detailed level where a special-exception cost report was the most profitable thing to do first. We costed it at about $400. They could even name the man who was going to use it, "Old Jim" in the factory production-planning department. At that level they suddenly realized that Jim was not motivated to make use of such advice yet, and they cleverly remembered to design-in the motivation to wake Jim up, so he took the report seriously.***
- ***I'm glad to report that the Division survived (others had not). Sometimes I think it might have been foreign exchange rates which did the trick. But it might have been getting clear goals, and doing projects in evolutionary result deliveries.***

© Gilb@acm.org 1999

Lessons Learned
The Chemicals Organization

- **Do not assume people are working towards even remotely correct objectives**
- **Do not assume top management understands that your project has no chance**
- **Do not assume people even want to talk about the real objectives**
- **Do not assume that the common sense of evolutionary results is understood by the culture**

© Gilb@acm.org 1999 119

Case 7.
The News Agency.

- ***Barrier: "There is just so very much to do <u>before</u> we are ready to deliver anything".***

- ***A world-famous UK news agency was, in 1986, going to electronically sell its' clippings files for the last ten years. They explained to me how much time it would take to get all their files (10 years worth) into electronic medium.***
- ***In thinking about possible Evolutionary cycles of result delivery I asked them if they could <u>just start working backwards, from yesterday's clippings "a year at a time"</u>. One theory was that the most recent past was also the most valuable for our industrial clients. I also pointed out that there were surely certain <u>types of reports which were more useful than others</u> initially in this context, for example company reports.***
- ***This simple Evolutionary thought seemed obvious enough to me. But I kept on hearing long afterwards how incredible they felt it was, that they had never considered the thought themselves.***
- ***If you don't look, you won't find.***

© Gilb@acm.org 1999 120

Case 8. The Swedish Map-making Institute case.

- ***Barrier : "Our system is so small and simple that we don't need many cycles, we'll have it all done shortly."***
- *A Norwegian systems house had contracted a **fixed price ($80,000), and fixed deadline** for making a map-drawing system, including computer hardware, for the Swedish Government Fixed-Property Directorate.*
- *There were only **two employees**, Gunnar and Anne-Lise, experienced in making a similar system, and they were assigned to the task.*
- *As an experiment, their top management asked me to spend a day with them.*
- *On that first day, we set **quantified goals**. One goal in particular was to dominate. The "**maintainability**" of the system by the customer, at the local customer site had to be very easy. This goal immediately led us to find suitable design ideas (software tools for error analysis and testing). The result of this was that it became clear that we had **twice as much work to do**, than the salesperson had envisaged, when making the fixed price and deadline commitment.*
- *That first day, we developed a **ten-cycle Evolutionary plan** with the two project people, over, hopefully, the deadlined three-month period.*
- *We planned to get the **basic map-making system up and running, in the first cycle** or so. We had to use a **crude temporary link** to their main computer, but it already existed, and cost nothing to make use of, until we in a later cycle built a customized link. Fancy polygon drawing features were also put off until later cycles. All things like **maintenance and training** documentation were in the **last** cycles.*
- *After about two, of the projected three, months a small **crisis** erupted. Both Gunnar and Anne-Lise had been off the job ill. Both had been forced to take time away from the project to service emergencies with previous projects.*
- *So, **after two months, three Evolutionary cycles had been delivered to Sweden**, and it became painfully clear that the last seven cycles were never going to be completed by the original deadline.*
- *"Hat in hand" Gunnar went to the customer to ask advice about what best to do about his "late" project. "LATE!?" the customer replied (and I was later to hear him and Gunnar both personally tell the tale at a conference in Sweden, where they specifically asked the audience to take my talk about Evolutionary delivery and Ideaware seriously, as that was the key to their success). LATE!? We are **already happily using the system weeks before the promised deadline.** We don't yet need the missing features. Please do not be tempted to do "crash program" actions which might threaten the quality of our system. Take your time. Do a good job. And we will only speak of the only system we have experienced, which was actually delivered **BEFORE the deadline.***

Case 8.
The British Life Assurance Computerized System cancelled by the Board.

- ***Spring 1993,***
- ***I was asked to analyze the situation at a well known British Life Assurance company. They had spent about £70 million on an "advanced" computerization project. But I found no useful objective was served for anybody. They were building it because advanced stuff was a good idea for a large company.***
- ***Their goal was to have a more-flexible advanced system (not much better defined than that either) - no mention of precisely which improvements, like in operating costs, were going to appear, and when. I believe that it was partly this <u>total lack of clearly-defined goals</u> which permitted the project to wander on, without any warning signals, <u>without any commitment to specific result delivery in small early increments.</u>***
- ***Finally the huge expenditure, and total lack of any planned or previous result delivery to anybody in the insurance company, got to the Board of Directors. They closed down the 200-person project about a month later. It was reported in the trade weeklies.***
- ***So was, a similar "Building Society" (Savings and Loan) system and a Stock Exchange reported, at the same time, with the same magnitude of losses.***
- ***There were plenty of "managers" involved at the assurance company. But they were not managing.***
- ***They did not, in my opinion understand <u>how to define useful results.</u>***
- ***And I had to document this thoroughly for them. I fear that the culture of poor definition of results is so prevalent that my message did not change their practice.***

© Gilb@acm.org 1999

This viewgraphs is the sole peoperty of Mr. Bill Deibler of SSQC. Do not copy or reproduce without written permission from Software Research Institute and Mr. Deibler. Any person copying or reproducing this material without permission is in violation of existing copyright laws.

Making the CMM Work: Streamlining the CMM for SMALL Projects

About SSQC

- Software Systems Quality Consulting
- Located in San Jose
- Since 1990 - assisting organizations to develop and improve their software engineering and management practices
  - CMM
  - ISO 9000
  - Software QA and Testing
  - Business Process Reengineering and Benchmarking
  - Education and Training

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

© Software Systems Quality Consulting
2269 Sunny Vista Drive, San Jose CA 95128
All rights reserved.

408-985-4476
ssqc@concentric.net
www.ssqc.com

Topics

- Background, guidance, assumptions
- Mapping the CMM
  - Levels, ratings, and other traps
  - Tailoring techniques and examples for small projects
- Strategy for transitioning to the CMM
  - Appraisal methods - return on investment
- The future of the CMM: CMMI

The CMM is …

… a procurement tool that has evolved into a process improvement tool

- Selection
- Expectation
- Confidence
- Extent of control

CMU/SEI-87-TR-23, p. 3

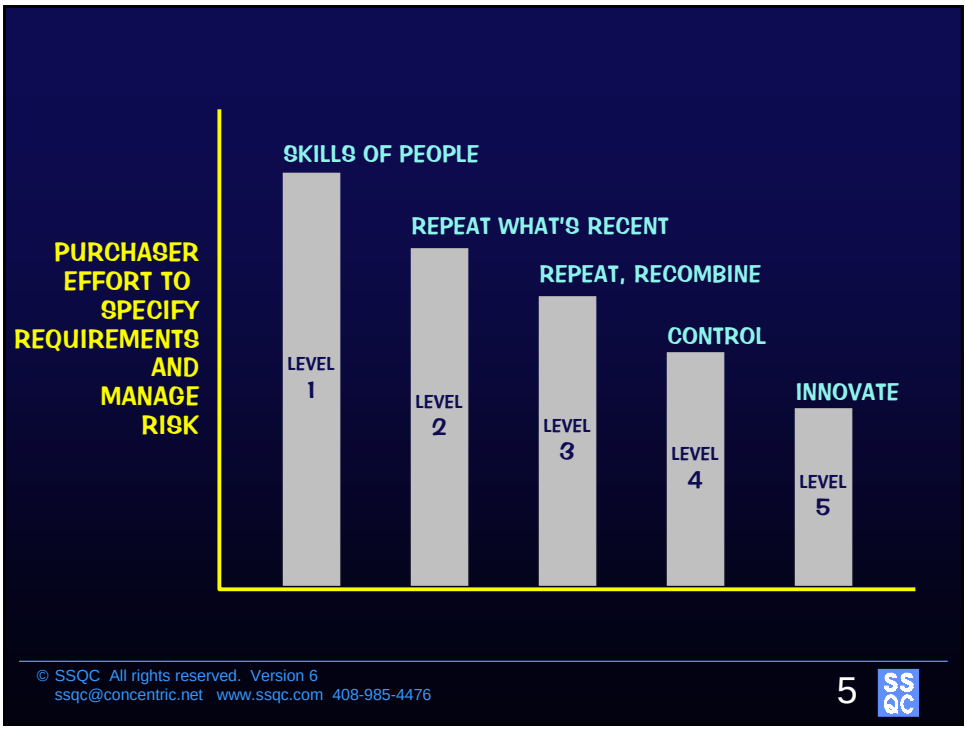
Provide a method for US DoD *to more effectively evaluate the abilities of their software contractors to competently perform on software engineering contracts.*

CMU/SEI-93-TR-24, section 1.3

The CMM was designed to guide software organizations in selecting process improvement strategies by determining current process maturity and identifying the few issues most critical to software quality and process improvement.

All rights reserved.

408-985-4476
ssqc@concentric.net
www.ssqc.com



The CMM is ...

… structured to support a large organization executing large-scale projects for a specific customer

CMU/SEI-93-TR-24, Section 2.2
The detailed practices characterize the … behavior … expected in an organization doing <u>large-scale projects</u> in a <u>government contracting</u> context.

CMU/SEI-93-TR-25, Section 1.1
The CMM must be appropriately interpreted when the business environment of the organization differs significantly from that of a <u>large contracting organization</u>.

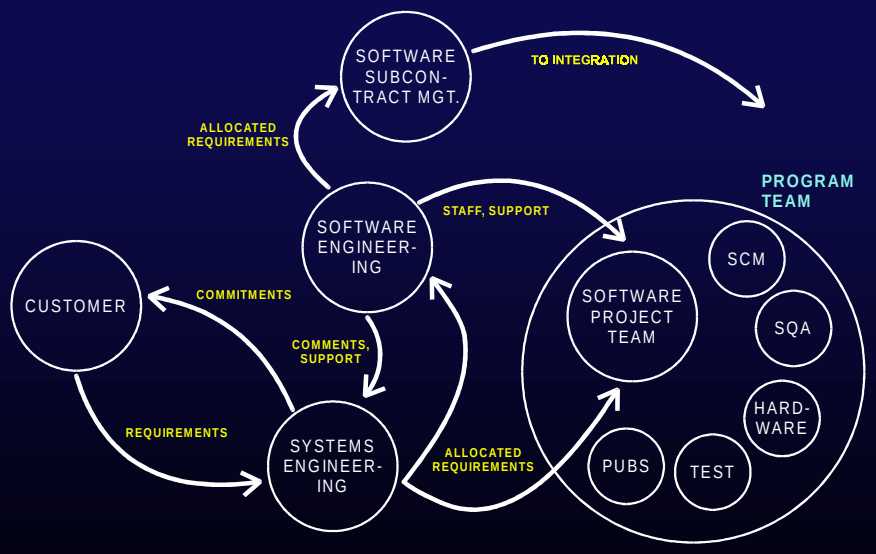
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
6

Implications and assumptions

- Formal systems and an extensive infrastructure and overhead are justified and required
- Market/customer-specified and funded development activities, methods, and tools
- Proliferation of tools and methods - size, variety of technologies, acquisition

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

The implied organizational model



© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

© Software Systems Quality Consulting
2269 Sunny Vista Drive, San Jose CA 95128
All rights reserved.

408-985-4476
ssqc@concentric.net
www.ssqc.com

Key points of comparison

Small Project	Large Project
Team size (1-15, 1-3 teams)	≧ ~15
Established individual relationships and interpersonal skills	Formal relationships
Work compartmentalized	Overlap, shared responsibility
Individual skills	Collective skills
Multiple and/or specialized responsibilities	Narrow assignments
High key-person dependency	Lower key person dependency
Short term (≦~6 months)	Longer duration
Low overhead	High overhead

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

9

Focus on established principles

CMU/SEI-93-TR-25, Section 1.1
The CMM must be appropriately interpreted when the business environment of the organization differs significantly from that of a large contracting organization. The role of professional judgment in making informed use of the CMM must be recognized.

CMU/SEI-93-TR-25, Section 4.1
Although the key practices are meant to be independent of any particular implementation, specific terms and examples are consistently used … Organizations … should map [these terms] appropriately to their own organization, project, and business environment.

CMU/SEI-93-TR-25, Section 4.4.2
A group could vary from a single individual assigned part time, to several part-time individuals assigned from different departments, to several individuals dedicated full time.

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

10

Focus on "small"

- Techniques
  - Map the CMM
    1. Select KPAs
    2. Translate definitions and conventions
    3. Consolidate practices (e.g., reviews) and share responsibilities
    4. Combine documents (e.g., plans)

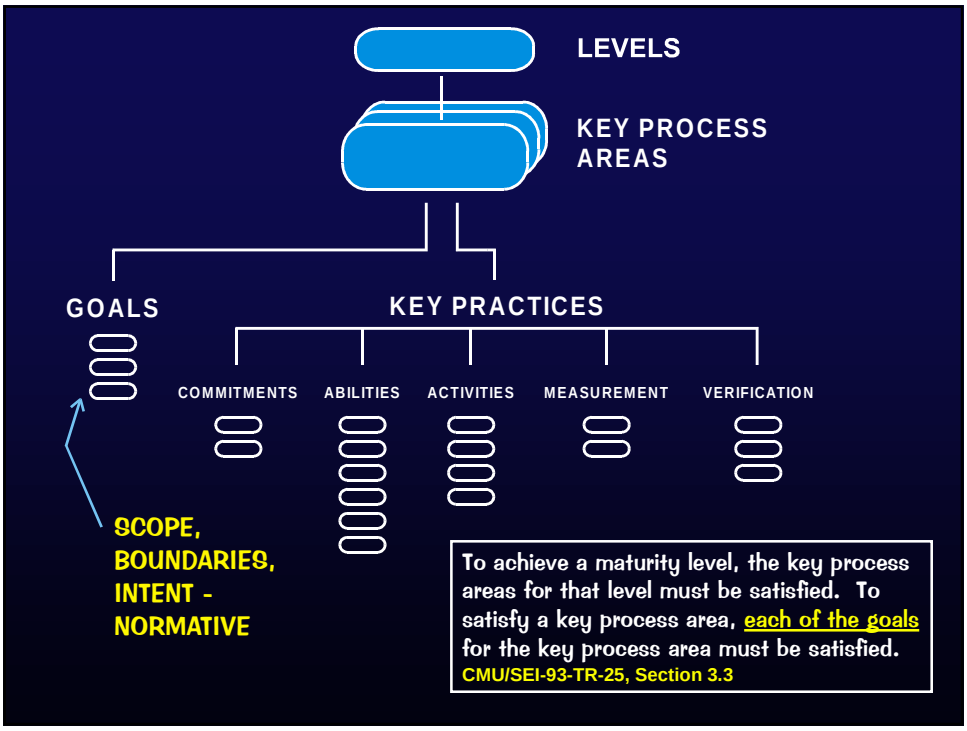
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

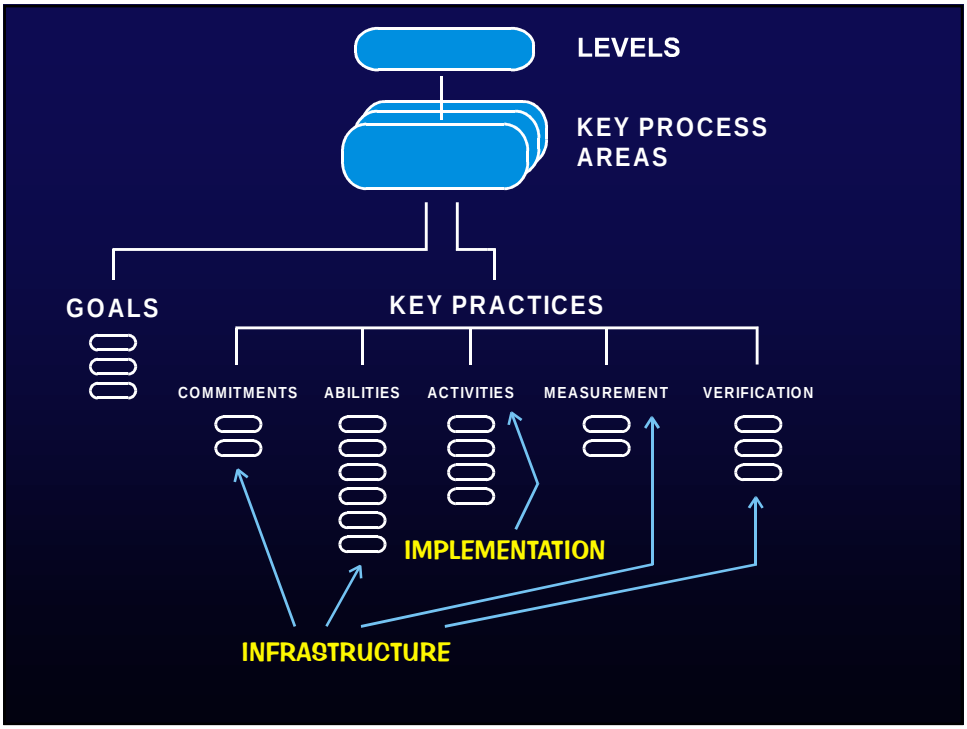
Map the CMM

- Reduce the information overload
- Expose the goals and principles
  - 52 key sentences, 500 pages of guidance
- Leverage parallels - maximize opportunities to minimize overhead and repetition
  - Between KPAs
  - Across levels

Basis for:
- Appropriate interpretation
- Professional judgement

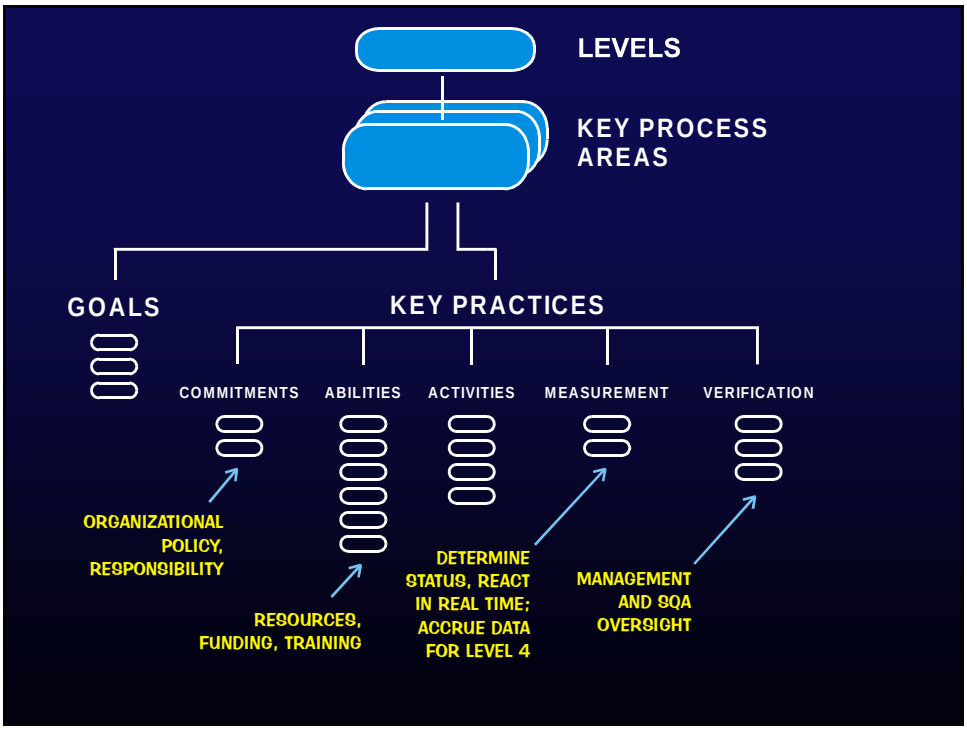
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476





© Software Systems Quality Consulting
2269 Sunny Vista Drive, San Jose CA 95128
All rights reserved.

408-985-4476
ssqc@concentric.net
www.ssqc.com



Map the CMM - at the KPA level

Focus on GOALS

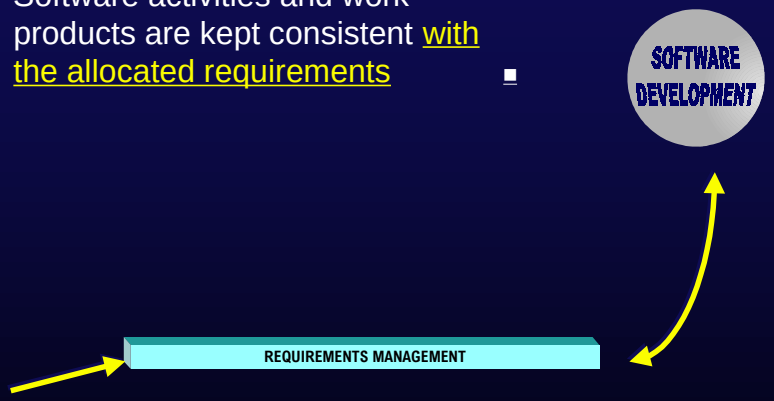
❶ Select KPAs

© SSQC All rights reserved. Version 3
ssqc@concentric.net www.ssqc.com 408-985-4476

16

Go1 Allocated requirements are controlled

Go2 Software activities and work products are kept consistent with the allocated requirements



LEVEL 2

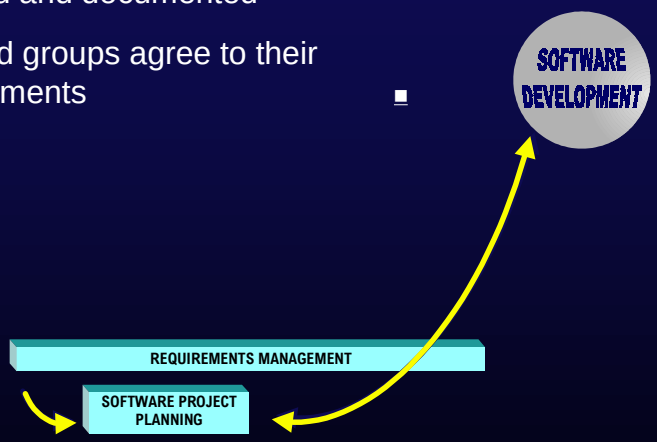
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

17

Go1 Estimates are documented

Go2 Activities and commitments are planned and documented

Go3 Affected groups agree to their commitments



LEVEL 2

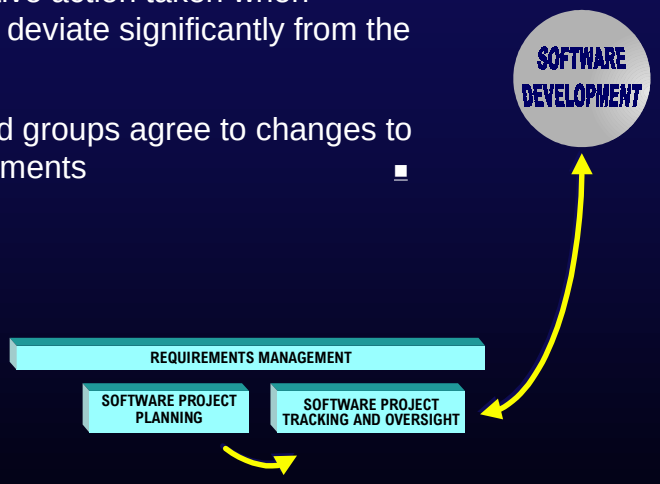
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

18

Go1 Actual results and performance tracked against the plan

Go2 Corrective action taken when actuals deviate significantly from the plan

Go3 Affected groups agree to changes to commitments



LEVEL 2

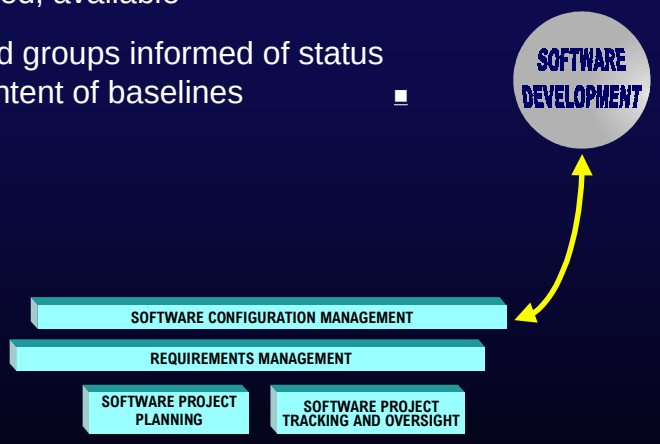
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

19

Go1 SCM activities planned

Go2 Selected work products identified, controlled, available

Go3 Affected groups informed of status and content of baselines



LEVEL 2

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

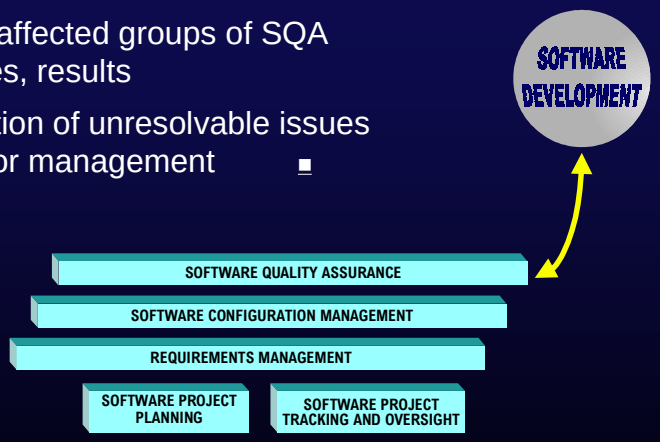
20

Go1 SQA activities planned

Go2 Adherence of activities, products objectively verified

Go3 Inform affected groups of SQA activities, results

Go4 Escalation of unresolvable issues to senior management



LEVEL 2

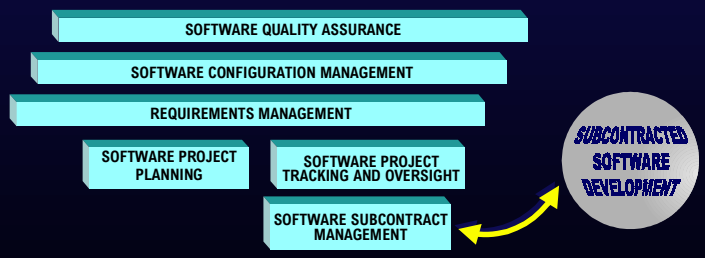
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

21

Go1 Qualified subcontractors selected

Go2 Commitments agreed to

Go3 On-going communication and tracking of performance against commitments

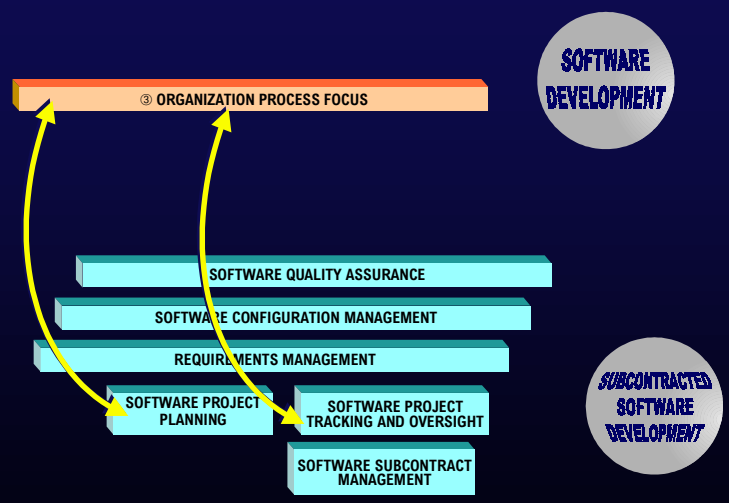


LEVEL 2

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

22

Go1 Software process development and improvement coordinated and planned



LEVEL 3

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

23

Go1 Standard software process developed, maintained

Go2 Information on use of standard process is collected, reviewed, available

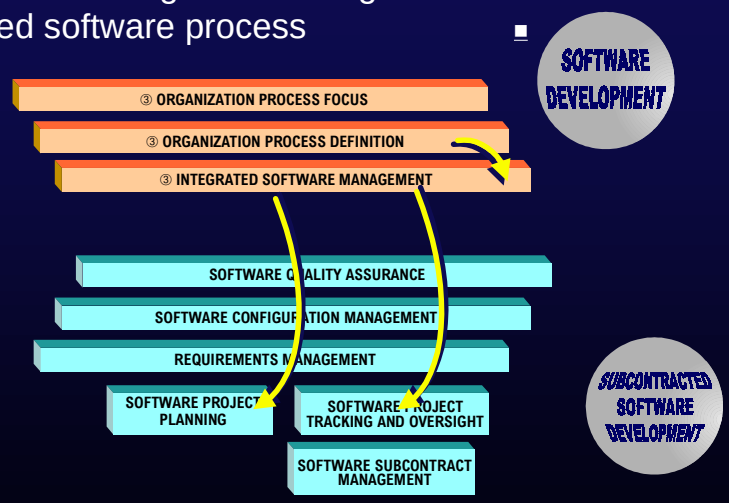
LEVEL 3

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

24

Go1 Project's defined software process tailored from standard software process

Go2 Project is managed according to its defined software process



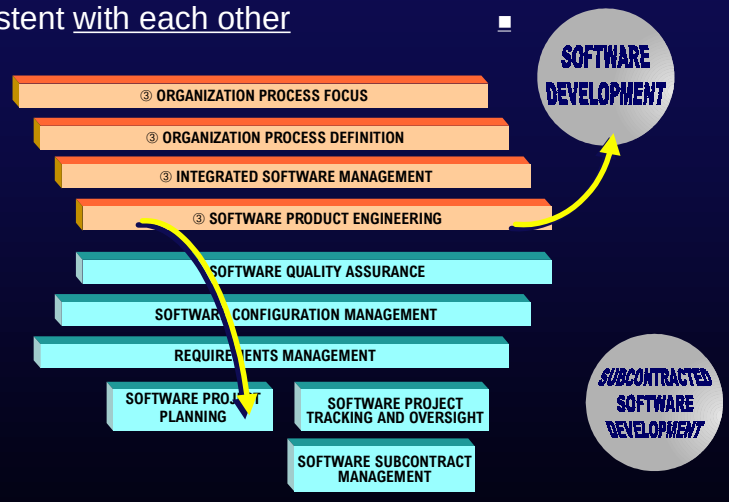
LEVEL 3

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

25

Go1 Software engineering tasks are defined, integrated, and consistently performed

Go2 Software work products are kept consistent with each other



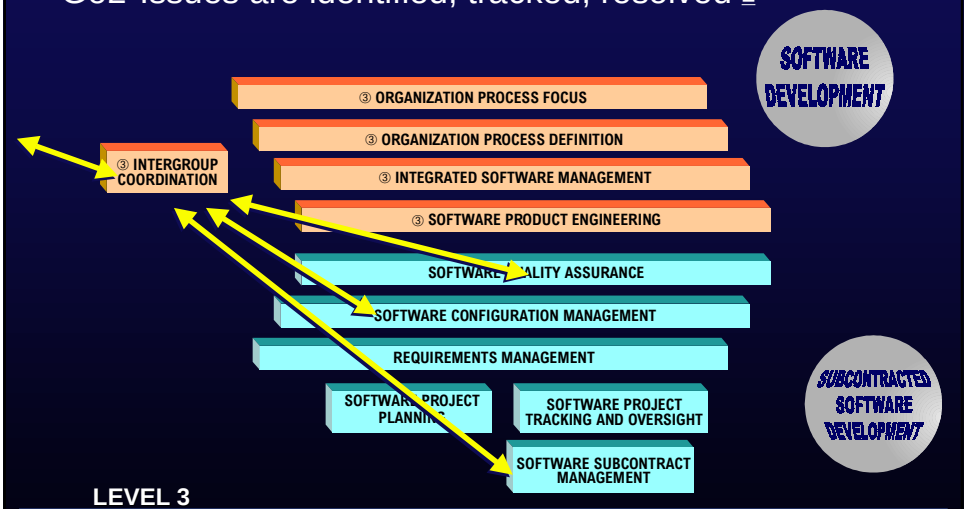
LEVEL 3

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

26

Go1 All affected groups agree to customer requirements and to inter-group commitments

Go2 Issues are identified, tracked, resolved



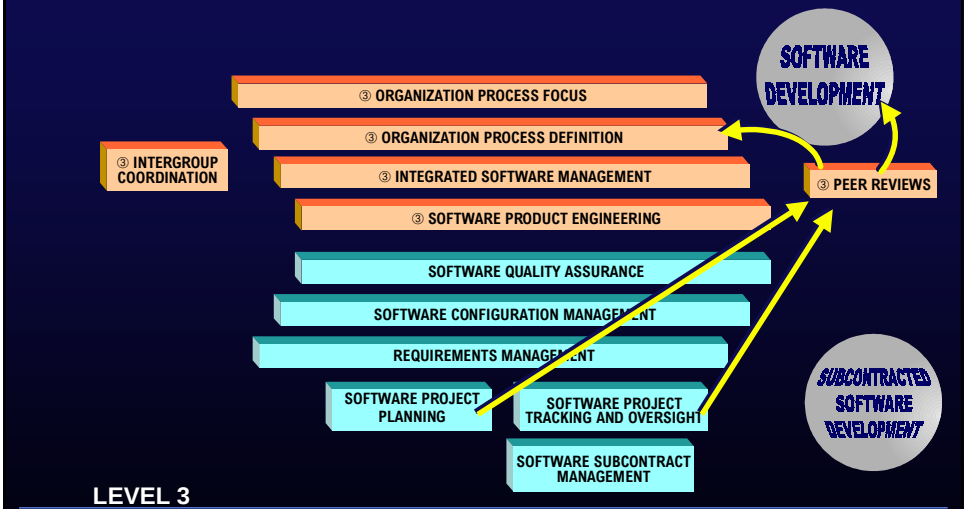
LEVEL 3

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

27

Go1 Peer reviews are planned

Go2 Identified defects removed



LEVEL 3

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

28

Go1 Management and technical training are planned and provided

Go2 Individuals receive required training

LEVEL 3

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

29

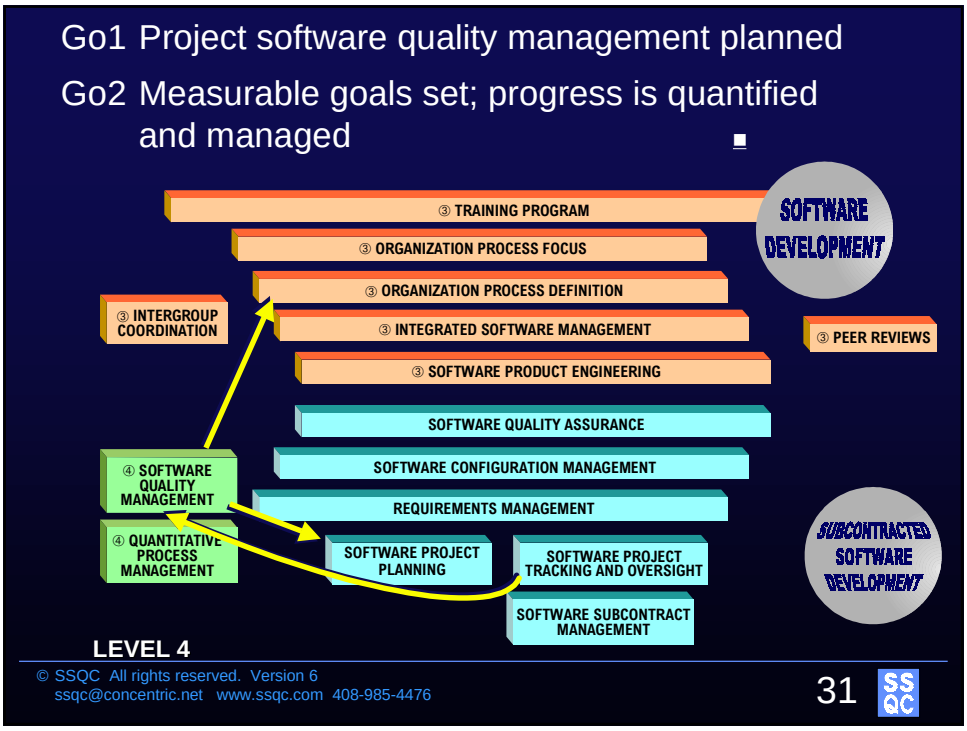
Go1 Achieve quantitative control of the project's software process

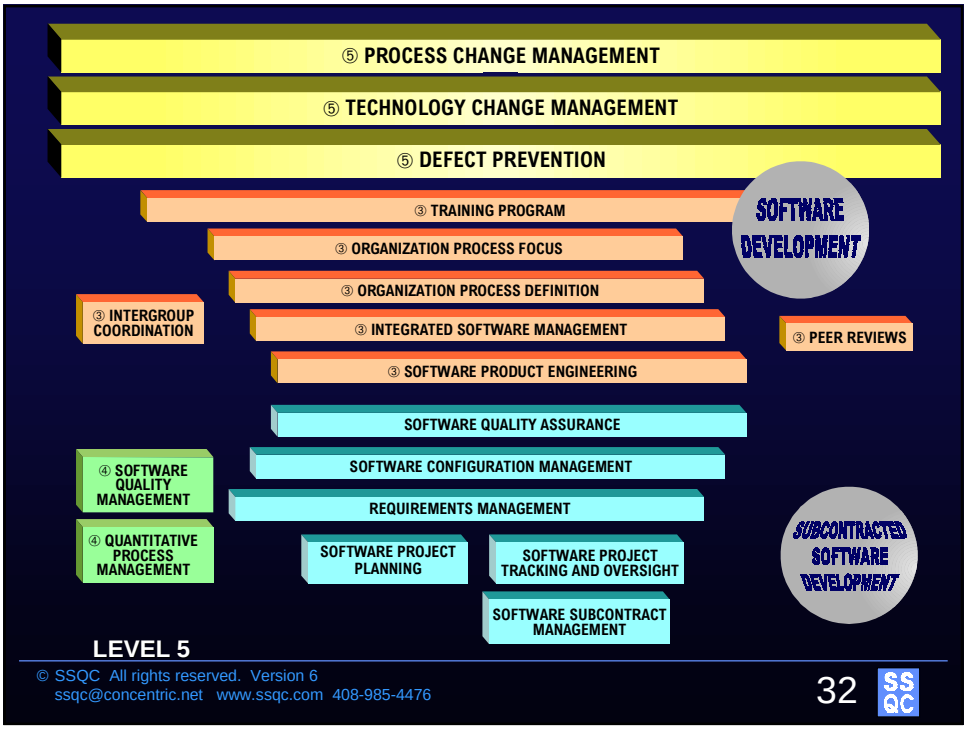
Go2 Capability of the organization's standard process is quantified

LEVEL 4

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

30


Go1 Project software quality management planned
Go2 Measurable goals set; progress is quantified and managed
③ TRAINING PROGRAM
③ ORGANIZATION PROCESS FOCUS
③ ORGANIZATION PROCESS DEFINITION
③ INTERGROUP COORDINATION
③ INTEGRATED SOFTWARE MANAGEMENT
③ PEER REVIEWS
③ SOFTWARE PRODUCT ENGINEERING
SOFTWARE DEVELOPMENT
SOFTWARE QUALITY ASSURANCE
SOFTWARE CONFIGURATION MANAGEMENT
④ SOFTWARE QUALITY MANAGEMENT
REQUIREMENTS MANAGEMENT
④ QUANTITATIVE PROCESS MANAGEMENT
SOFTWARE PROJECT PLANNING
SOFTWARE PROJECT TRACKING AND OVERSIGHT
SOFTWARE SUBCONTRACT MANAGEMENT
SUBCONTRACTED SOFTWARE DEVELOPMENT
LEVEL 4
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
31


⑤ PROCESS CHANGE MANAGEMENT
⑤ TECHNOLOGY CHANGE MANAGEMENT
⑤ DEFECT PREVENTION
③ TRAINING PROGRAM
③ ORGANIZATION PROCESS FOCUS
③ ORGANIZATION PROCESS DEFINITION
③ INTERGROUP COORDINATION
③ INTEGRATED SOFTWARE MANAGEMENT
③ PEER REVIEWS
③ SOFTWARE PRODUCT ENGINEERING
SOFTWARE DEVELOPMENT
SOFTWARE QUALITY ASSURANCE
SOFTWARE CONFIGURATION MANAGEMENT
④ SOFTWARE QUALITY MANAGEMENT
REQUIREMENTS MANAGEMENT
④ QUANTITATIVE PROCESS MANAGEMENT
SOFTWARE PROJECT PLANNING
SOFTWARE PROJECT TRACKING AND OVERSIGHT
SOFTWARE SUBCONTRACT MANAGEMENT
SUBCONTRACTED SOFTWARE DEVELOPMENT
LEVEL 5
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
32

Level 5 Goals

DEFECT PREVENTION

Go1 Prevention activities are planned

Go2 Common causes of defects are identified, prioritized, and systematically eliminated ■

TECHNOLOGY CHANGE MANAGEMENT

Go1 Incorporation of technology change is planned

Go2 New technologies are evaluated for impact on quality and productivity

Go3 Appropriate new technologies are implemented across the organization ■

PROCESS CHANGE MANAGEMENT

Go1 Continuous process improvement planned and achieved

Go2 Organization-wide participation ■

Levels

- Basis for one set of ratings from the current assessment methods
  - CMM-Based Appraisal for Internal process Improvement (CBA-IPI)
  - Software Capability Evaluation (SCE)
- Potential for abuse
  - Theory: Walk before you run.
  - Practice: What do I have to do?
  - Best practice: What do I need to do?

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

34

Levels 2 and 3 for small organizations

- Typical Silicon Valley company
  - No proliferation of tools and methods across projects - commonality
    - Single approach across projects - documented or not
  - High degree of vocal participation and consensus
- Organizational focus = Level 3
- Transition between ISO 9001 and CMM Level 3

Case Study
❶ Select KPAs

Financial Algorithms, Ltd. (FAL)

© SSQC All rights reserved. Version 3
ssqc@concentric.net www.ssqc.com 408-985-4476
36

Map the CMM - at the key practice level

Across the KPAs

❷ Translate definitions and conventions

❸ Consolidate practices, share responsibilities

SILOS

CON SOLID ATE

© SSQC All rights reserved. Version 3
ssqc@concentric.net www.ssqc.com 408-985-4476

37

COMMON FEATURES		LEVELS 2						3							4		5		
		RM	SPP	SPT	SSM	SQA	SCM	OPF	OPD	TP	ISM	SPE	IC	PR	QPM	SQM	DP	TCM	PCM
GOALS	1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	2	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	3		●	●	●	●	●	●		●			●		●	●	●	●	●
	4				●	●	●												
✔ COMMIT-MENTS	1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	2		●	●	●			●							●	●	●	●	
	3																		
✔ ABILITIES	1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	2	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	3	●	●	●	●	●	●	●		●	●	●	●	●	●	●	●	●	●
	4	●	●	●		●	●	●		●		●	●		●		●	●	●
	5			●			●						●		●			●	
✖ ACTIVITIES	1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	2	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	3	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	4		●	●	●	●	●	●	●	●	●	●	●		●	●	●	●	●
	5		●	●	●	●	●	●	●	●	●	●	●		●	●	●	●	●
	6		●	●	●	●	●	●	●	●	●	●	●		●		●	●	●
	7		●	●	●	●	●	●			●	●	●		●		●	●	●
	8		●	●	●	●	●				●	●					●	●	●
	9		●	●	●		●				●	●							●
	10		●	●	●		●				●	●							●
	11		●	●	●						●								
	12		●	●	●														
	13		●	●	●														
	14		●																
	15		●																
✔ MEASURE-MENT	1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	2									●		●							
✔ VERIFICA-TION	1	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
	2	●	●	●	●	●	●			●	●	●	●		●	●	●	●	●
	3	●	●	●	●	●	●			●	●	●	●		●	●	●		
	4						●												

52

316

✔ = INSTITUTIONALIZATION ✖ = IMPLEMENTATION

368/594 (18x33)

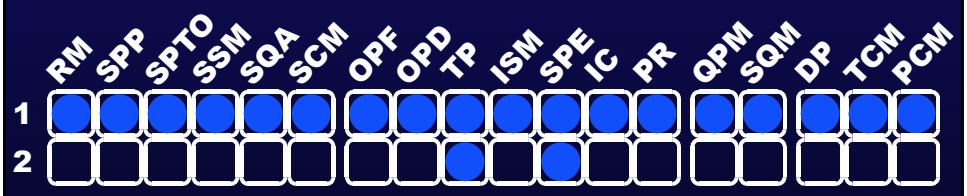
Example: Map institutionalization key practices across the KPAs

Commitments, Abilities, Measurement, Verification

© SSQC All rights reserved. Version 3
ssqc@concentric.net www.ssqc.com 408-985-4476

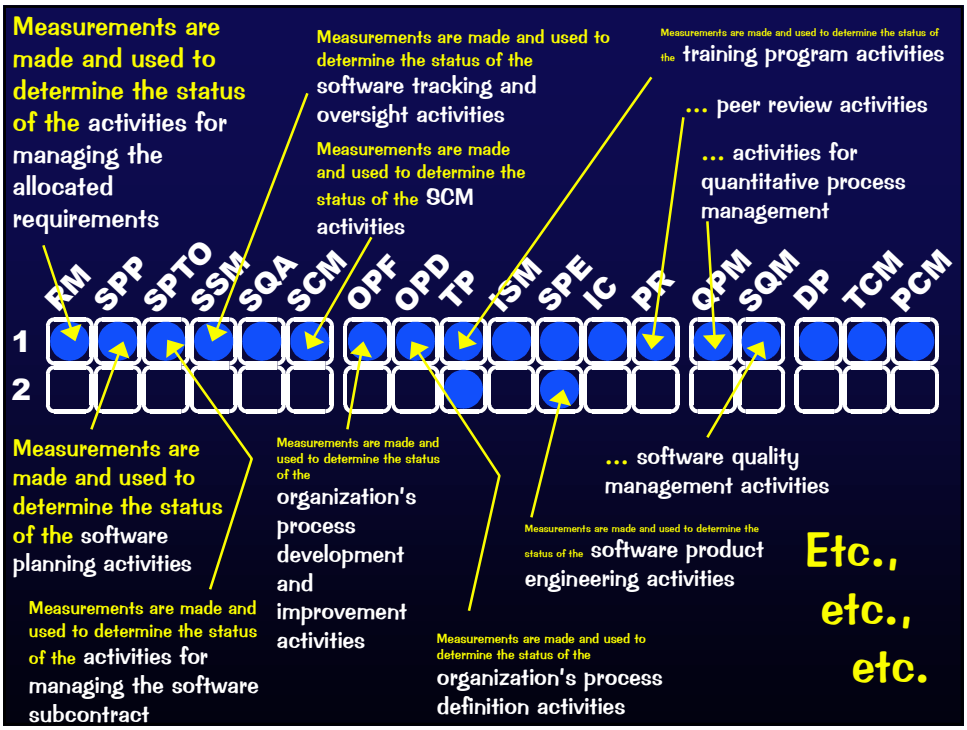
39

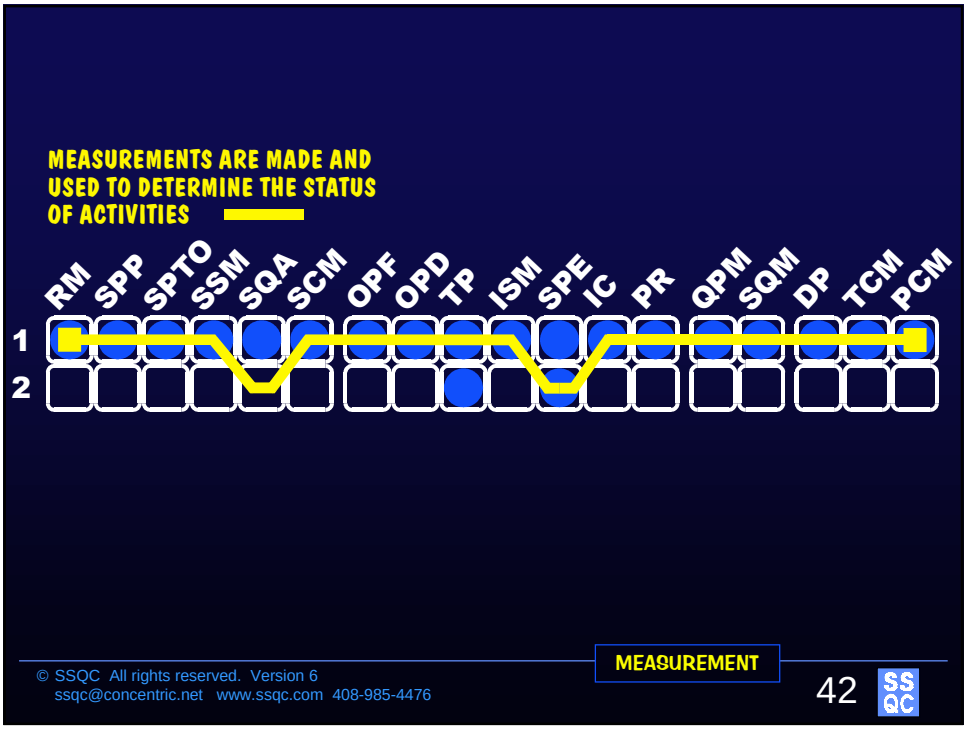
Measurement and analysis

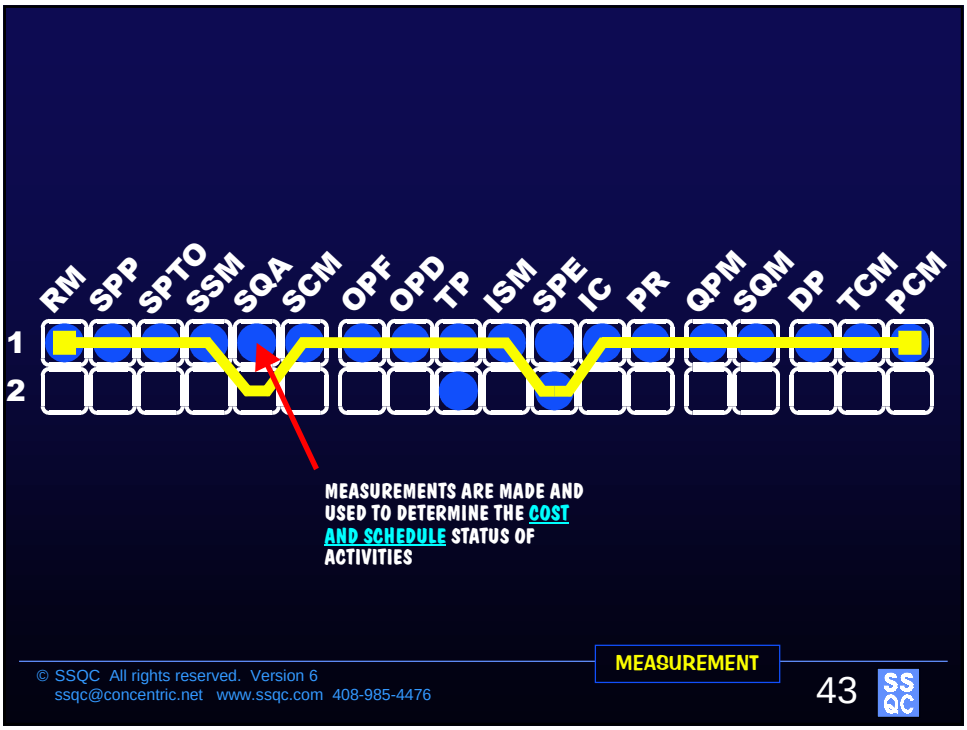


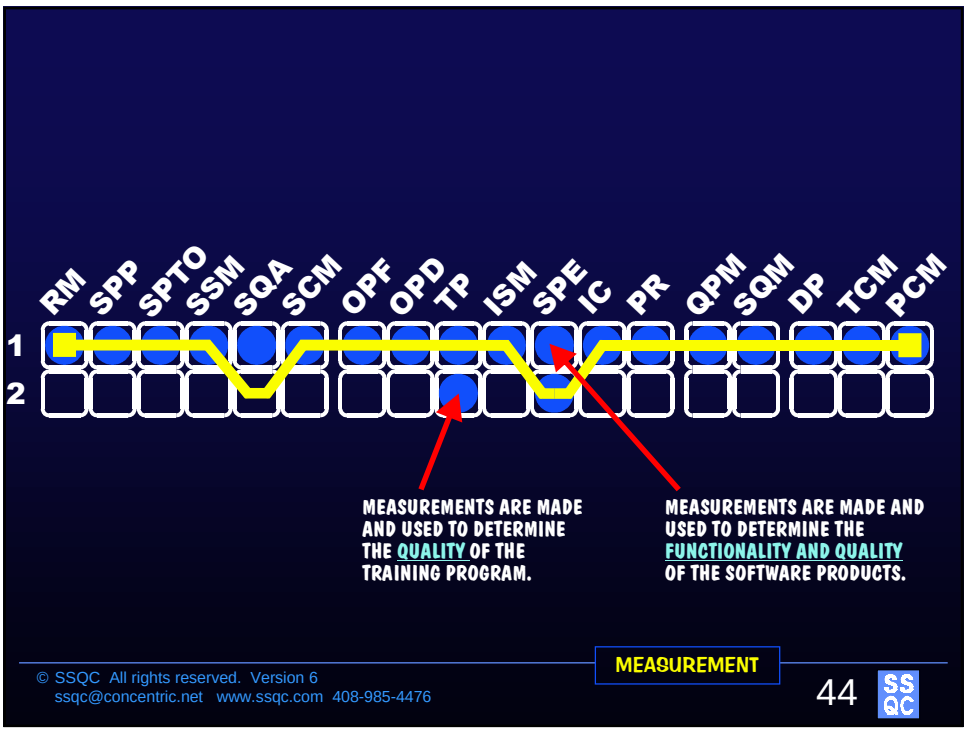
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

40


Measurements are made and used to determine the status of the activities for managing the allocated requirements
Measurements are made and used to determine the status of the software tracking and oversight activities
Measurements are made and used to determine the status of the SCM activities
Measurements are made and used to determine the status of the training program activities
... peer review activities
... activities for quantitative process management
RM SPP SPTO SSM SQA SCM OPF OPD TP ISM SPE IC PR QPM SQM DP TCM PCM
1
2
Measurements are made and used to determine the status of the software planning activities
Measurements are made and used to determine the status of the activities for managing the software subcontract
Measurements are made and used to determine the status of the organization's process development and improvement activities
Measurements are made and used to determine the status of the organization's process definition activities
Measurements are made and used to determine the status of the software product engineering activities
... software quality management activities
Etc., etc., etc.


MEASUREMENTS ARE MADE AND USED TO DETERMINE THE STATUS OF ACTIVITIES
RM SPP SPTO SSM SQA SCM OPF OPD TP ISM SPE IC PR QPM SQM DP TCM PCM
1
2
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
MEASUREMENT
42


RM SPP SPTO SSM SQA SCM OPF OPD TP ISM SPE IC PR QPM SQM DP TCM PCM
1
2
MEASUREMENTS ARE MADE AND USED TO DETERMINE THE COST AND SCHEDULE STATUS OF ACTIVITIES
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
MEASUREMENT
43
SS QC


RM SPP SPTO SSM SQA SCM OPF OPD TP ISM SPE IC PR QPM SQM DP TCM PCM
1
2
MEASUREMENTS ARE MADE AND USED TO DETERMINE THE QUALITY OF THE TRAINING PROGRAM.
MEASUREMENTS ARE MADE AND USED TO DETERMINE THE FUNCTIONALITY AND QUALITY OF THE SOFTWARE PRODUCTS.
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
MEASUREMENT
44
SS QC

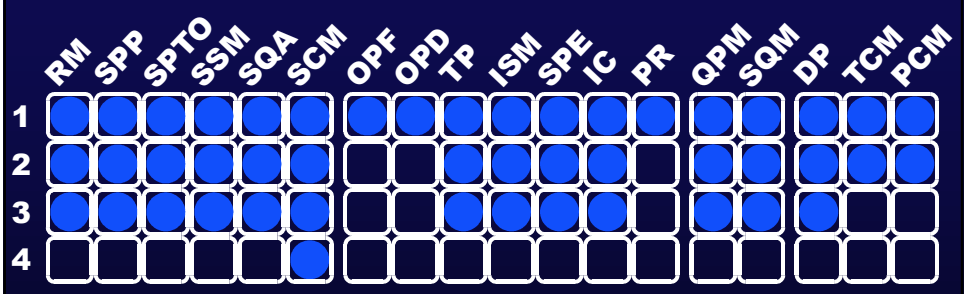
Measurements for small projects

- Useful for managing the current project (short-term) and future projects
- Easy to report and gather
  - Progress against the schedule
  - Accuracy of estimates
  - Defects - found, closed

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

45

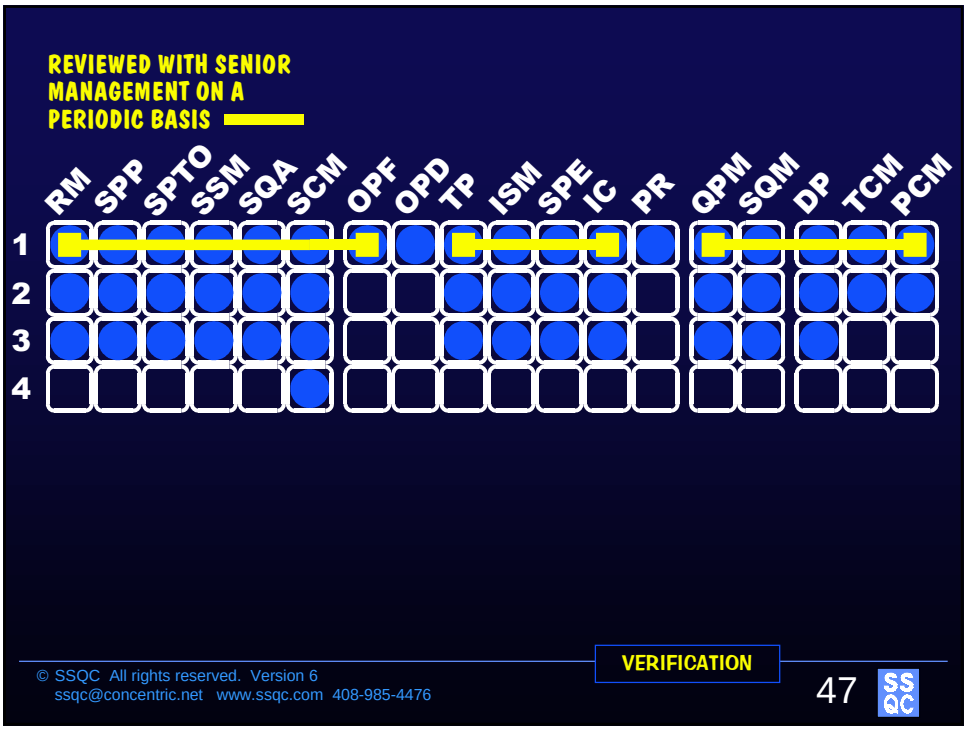
Verifying implementation

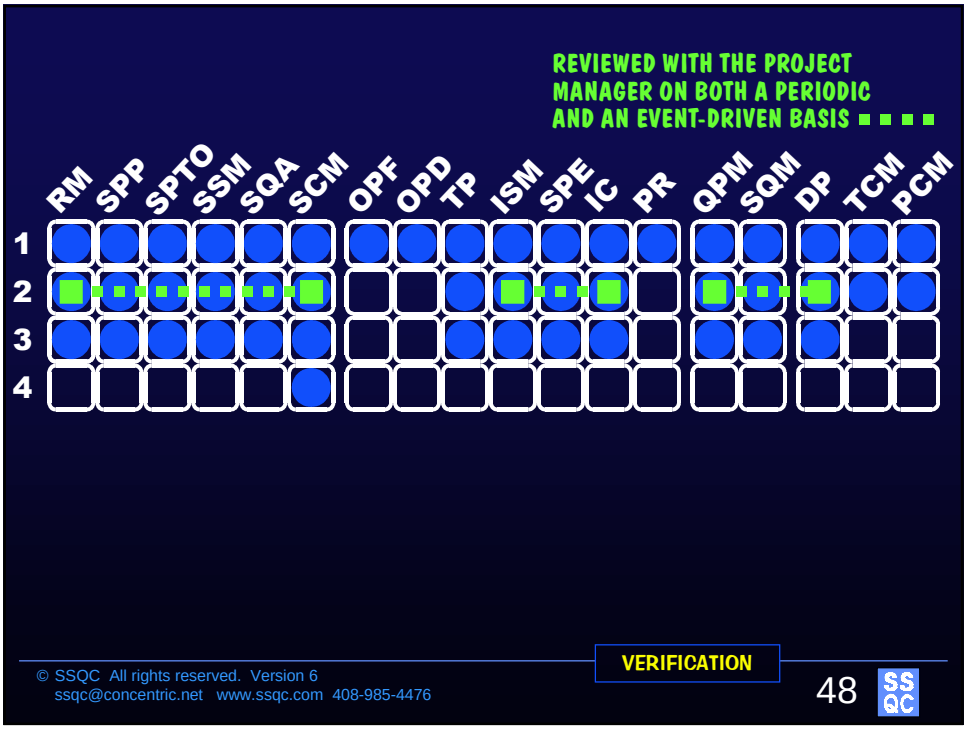


VERIFICATION

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

46


REVIEWED WITH SENIOR MANAGEMENT ON A PERIODIC BASIS
RM SPP SPTO SSM SQA SCM OPF OPD TP ISM SPE IC PR QPM SQM DP TCM PCM
1
2
3
4
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
VERIFICATION
47
SS QC


REVIEWED WITH THE PROJECT MANAGER ON BOTH A PERIODIC AND AN EVENT-DRIVEN BASIS
RM SPP SPTO SSM SQA SCM OPF OPD TP ISM SPE IC PR QPM SQM DP TCM PCM
1
2
3
4
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
VERIFICATION
48
SS QC

Senior management review versus project manager review

Project manager - The role with total business responsibility for an entire project; the individual who directs, controls, administers, and regulates a project building a software or hardware/software system. The project manager is the individual ultimately responsible to the customer.

Senior manager - A management role at a high enough level in an organization that the primary focus is the long-term vitality of the organization, rather than short-term project and contractual concerns and pressures. In general, a senior manager for engineering would have responsibility for multiple projects.

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

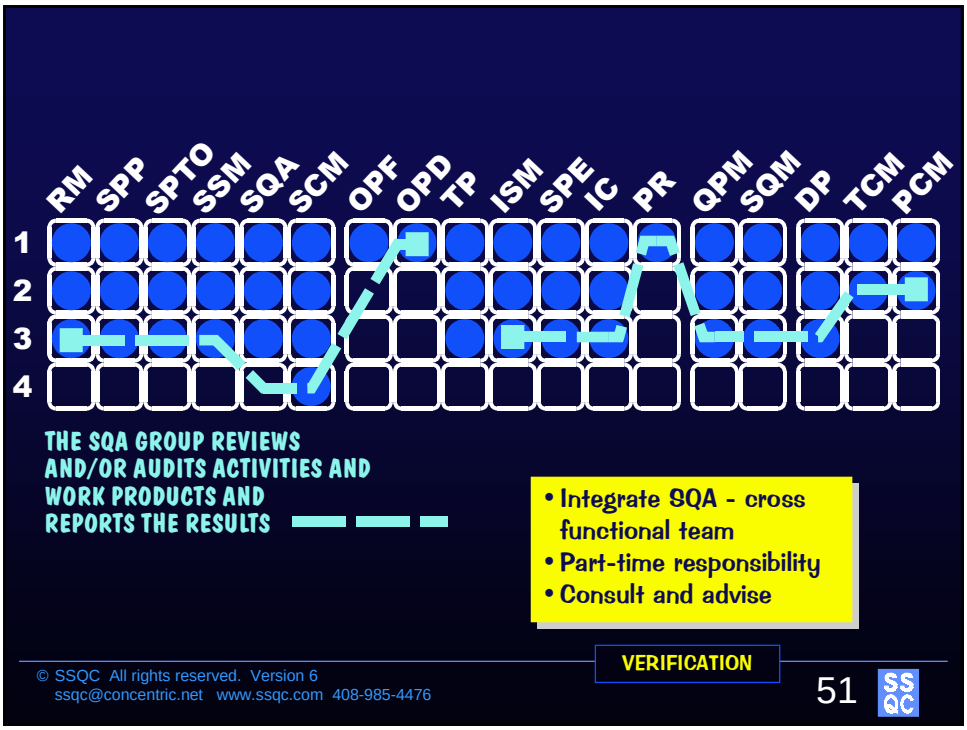
49

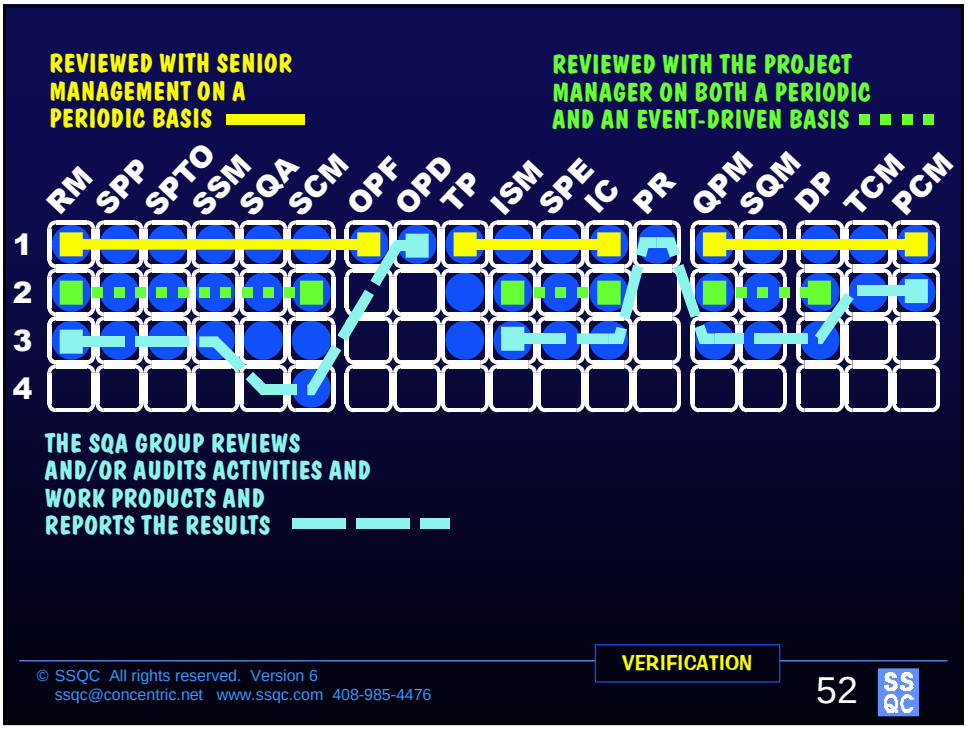
Case Study
❷ Translate definitions and conventions

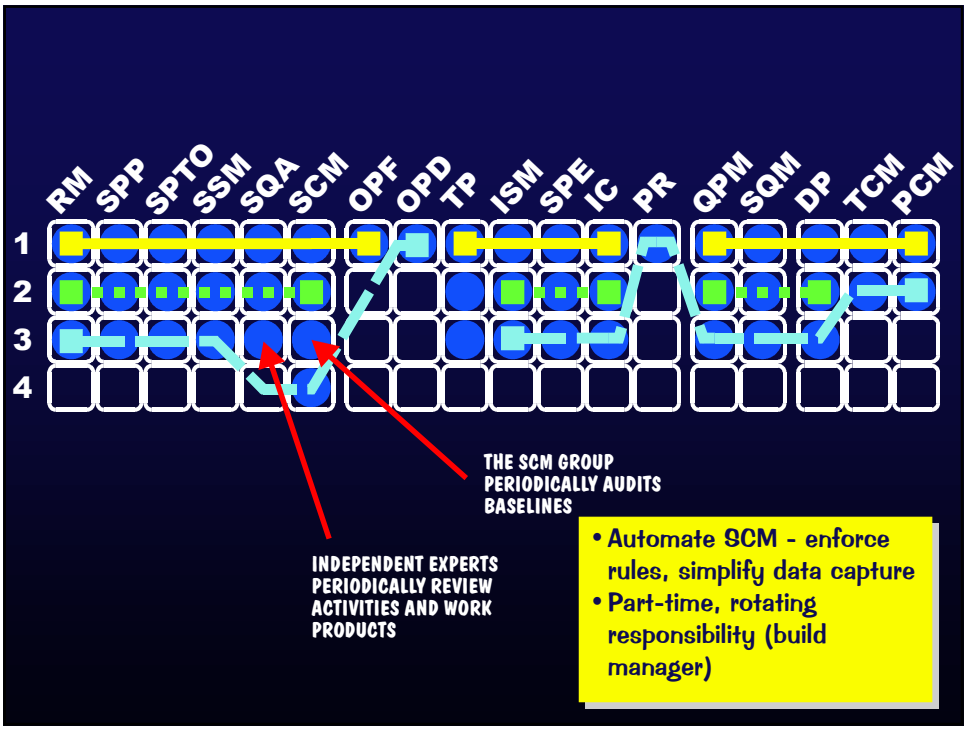
Financial Algorithms, Ltd. (FAL)
Compleat Systems, Inc. (CSI)

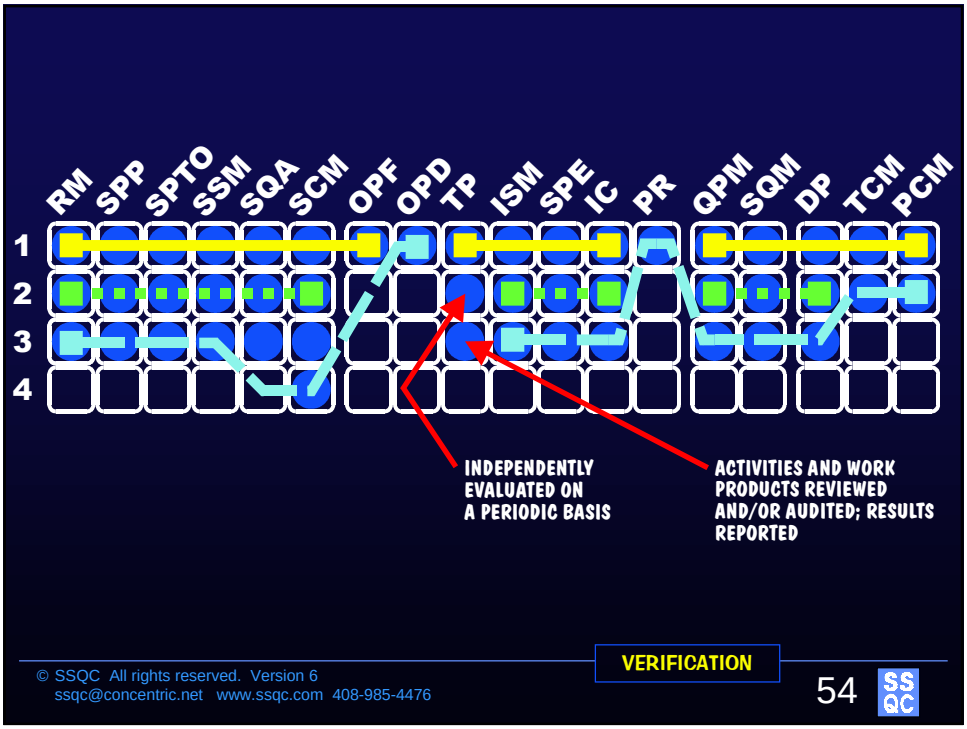
© SSQC All rights reserved. Version 3
ssqc@concentric.net www.ssqc.com 408-985-4476

50


RM SPP SPTO SSM SQA SCM OPF OPD TP ISM SPE IC PR QPM SQM DP TCM PCM
1
2
3
4
THE SQA GROUP REVIEWS AND/OR AUDITS ACTIVITIES AND WORK PRODUCTS AND REPORTS THE RESULTS
• Integrate SQA - cross functional team
• Part-time responsibility
• Consult and advise
VERIFICATION
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
51


REVIEWED WITH SENIOR MANAGEMENT ON A PERIODIC BASIS
REVIEWED WITH THE PROJECT MANAGER ON BOTH A PERIODIC AND AN EVENT-DRIVEN BASIS
RM SPP SPTO SSM SQA SCM OPF OPD TP ISM SPE IC PR QPM SQM DP TCM PCM
1
2
3
4
THE SQA GROUP REVIEWS AND/OR AUDITS ACTIVITIES AND WORK PRODUCTS AND REPORTS THE RESULTS
VERIFICATION
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
52


RM SPP SPTO SSM SQA SCM OPF OPD TP ISM SPE IC PR QPM SQM DP TCM PCM
1
2
3
4
THE SCM GROUP PERIODICALLY AUDITS BASELINES
INDEPENDENT EXPERTS PERIODICALLY REVIEW ACTIVITIES AND WORK PRODUCTS
• Automate SCM - enforce rules, simplify data capture
• Part-time, rotating responsibility (build manager)


RM SPP SPTO SSM SQA SCM OPF OPD TP ISM SPE IC PR QPM SQM DP TCM PCM
1
2
3
4
INDEPENDENTLY EVALUATED ON A PERIODIC BASIS
ACTIVITIES AND WORK PRODUCTS REVIEWED AND/OR AUDITED; RESULTS REPORTED
VERIFICATION
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
54
SS QC

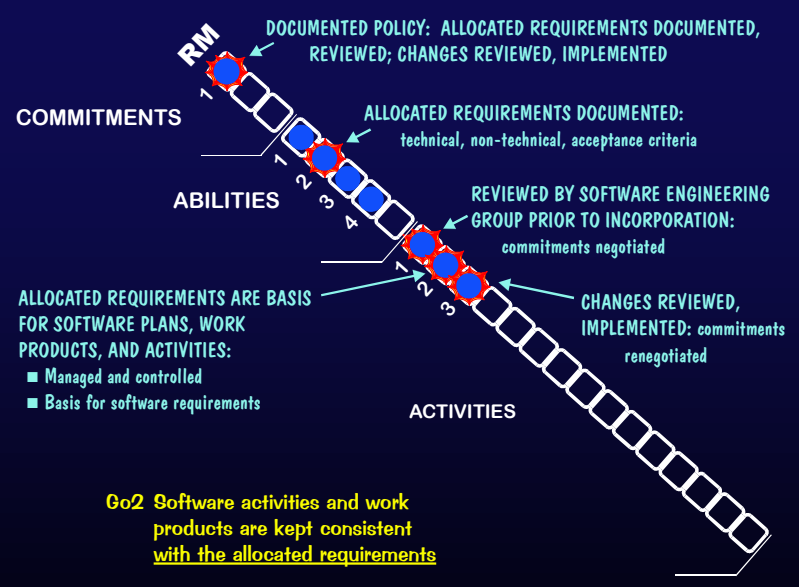
Example: Map activities across the KPAs

❸ Consolidate practices, share responsibilities

→ RM, SPP, SPE, and IC
SPP, SPTO, ISM

© SSQC All rights reserved. Version 3
ssqc@concentric.net www.ssqc.com 408-985-4476

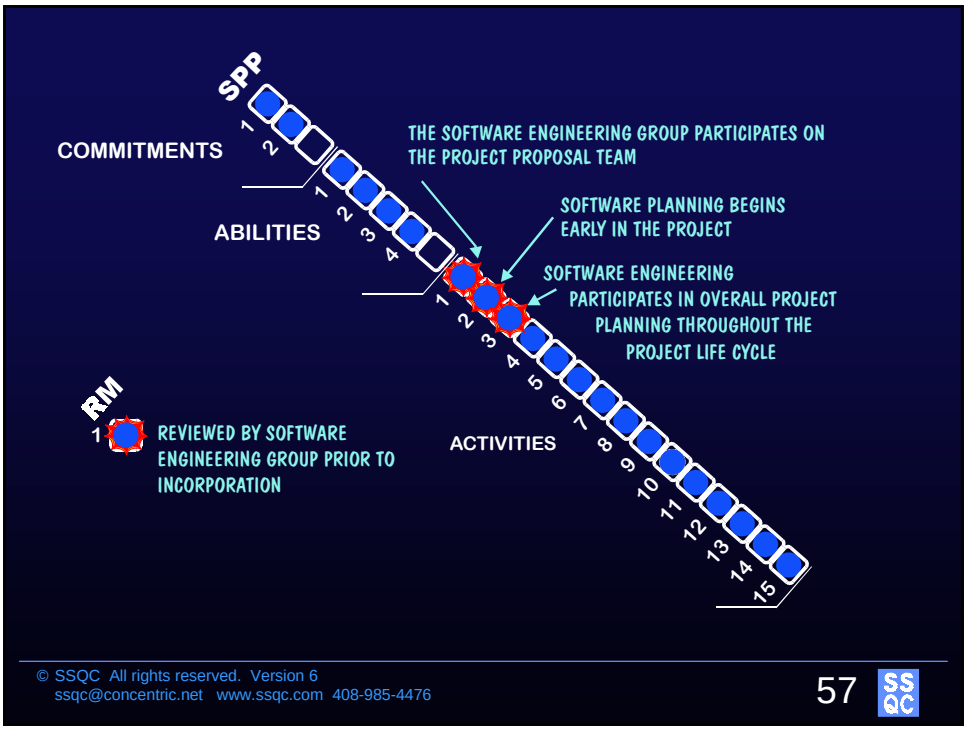
55

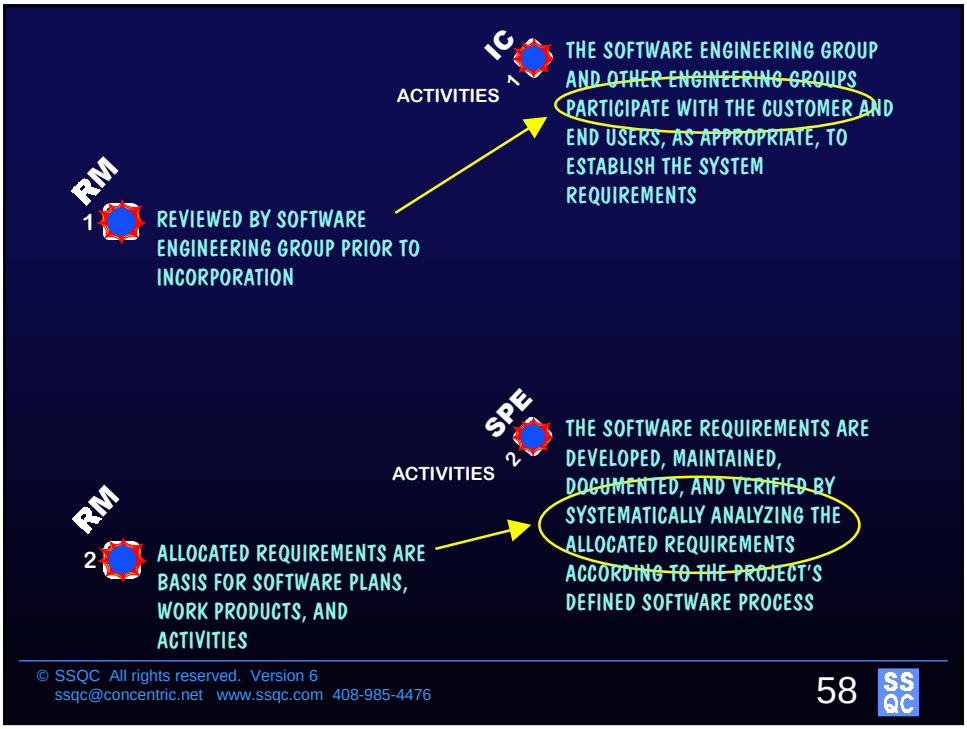


Go2 Software activities and work products are kept consistent with the allocated requirements

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

56


SPP
COMMITMENTS
1
2
ABILITIES
1
2
3
4
THE SOFTWARE ENGINEERING GROUP PARTICIPATES ON THE PROJECT PROPOSAL TEAM
SOFTWARE PLANNING BEGINS EARLY IN THE PROJECT
SOFTWARE ENGINEERING PARTICIPATES IN OVERALL PROJECT PLANNING THROUGHOUT THE PROJECT LIFE CYCLE
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
ACTIVITIES
RM
1
REVIEWED BY SOFTWARE ENGINEERING GROUP PRIOR TO INCORPORATION
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
57
SS QC


IC
1
ACTIVITIES
THE SOFTWARE ENGINEERING GROUP AND OTHER ENGINEERING GROUPS PARTICIPATE WITH THE CUSTOMER AND END USERS, AS APPROPRIATE, TO ESTABLISH THE SYSTEM REQUIREMENTS
RM
1
REVIEWED BY SOFTWARE ENGINEERING GROUP PRIOR TO INCORPORATION
SPE
2
ACTIVITIES
THE SOFTWARE REQUIREMENTS ARE DEVELOPED, MAINTAINED, DOCUMENTED, AND VERIFIED BY SYSTEMATICALLY ANALYZING THE ALLOCATED REQUIREMENTS ACCORDING TO THE PROJECT'S DEFINED SOFTWARE PROCESS
RM
2
ALLOCATED REQUIREMENTS ARE BASIS FOR SOFTWARE PLANS, WORK PRODUCTS, AND ACTIVITIES
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
58
SS QC

Case Study - Implications for software-only projects

© SSQC All rights reserved. Version 3
ssqc@concentric.net www.ssqc.com 408-985-4476

59

Example: Map activities across the KPAs *(cont.)*

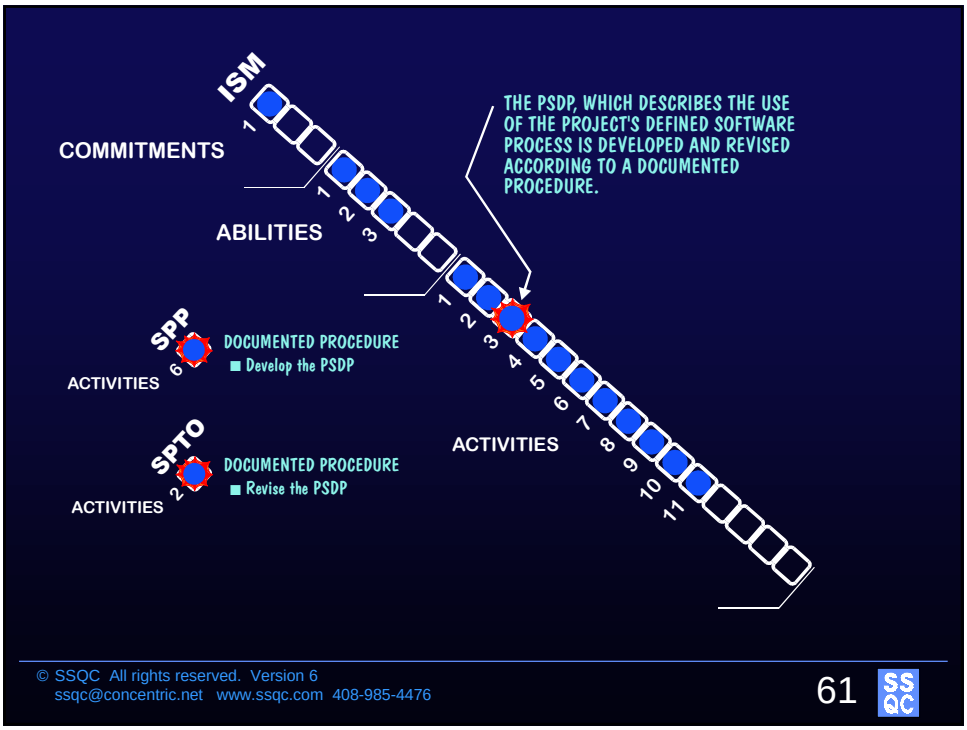
❸ Consolidate practices, share responsibilities

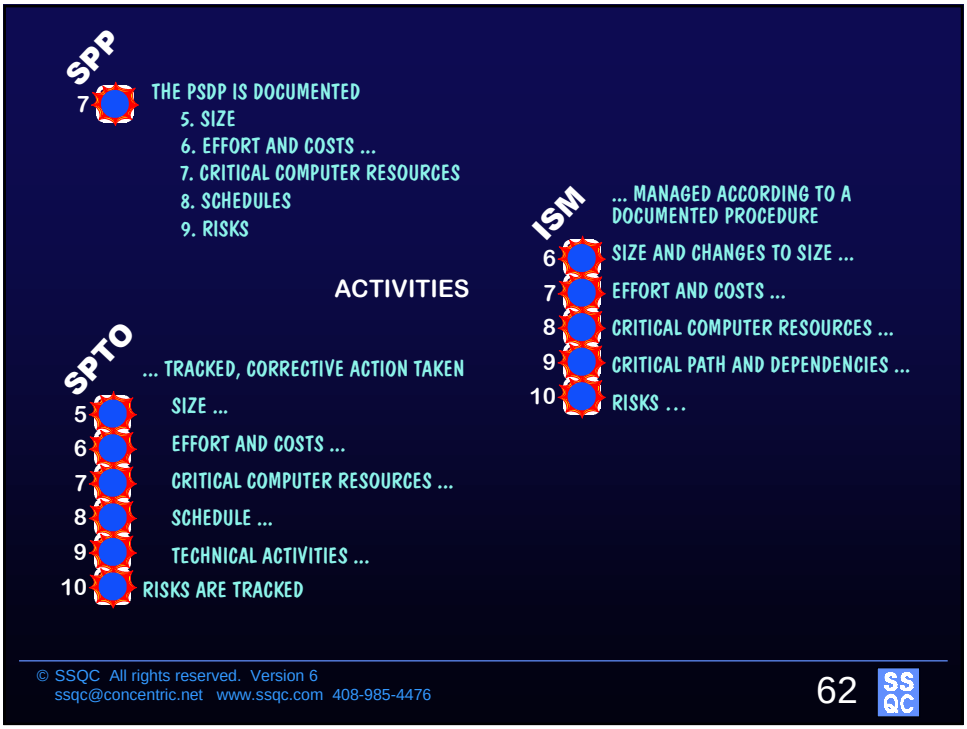
RM, SPP, SPE, and IC

→ SPP, SPTO, ISM

© SSQC All rights reserved. Version 3
ssqc@concentric.net www.ssqc.com 408-985-4476

60


ISM
COMMITMENTS
1
ABILITIES
1
2
3
THE PSDP, WHICH DESCRIBES THE USE OF THE PROJECT'S DEFINED SOFTWARE PROCESS IS DEVELOPED AND REVISED ACCORDING TO A DOCUMENTED PROCEDURE.
SPP
6
DOCUMENTED PROCEDURE
■ Develop the PSDP
ACTIVITIES
SPTO
2
DOCUMENTED PROCEDURE
■ Revise the PSDP
ACTIVITIES
1
2
3
4
5
6
7
8
9
10
11
ACTIVITIES
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
61
SS QC


SPP
7
THE PSDP IS DOCUMENTED
5. SIZE
6. EFFORT AND COSTS ...
7. CRITICAL COMPUTER RESOURCES
8. SCHEDULES
9. RISKS
ACTIVITIES
SPTO
... TRACKED, CORRECTIVE ACTION TAKEN
5 SIZE ...
6 EFFORT AND COSTS ...
7 CRITICAL COMPUTER RESOURCES ...
8 SCHEDULE ...
9 TECHNICAL ACTIVITIES ...
10 RISKS ARE TRACKED
ISM
... MANAGED ACCORDING TO A DOCUMENTED PROCEDURE
6 SIZE AND CHANGES TO SIZE ...
7 EFFORT AND COSTS ...
8 CRITICAL COMPUTER RESOURCES ...
9 CRITICAL PATH AND DEPENDENCIES ...
10 RISKS ...
© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476
62
SS QC

Estimation

- Find methods suitable for small projects
- PROBE - Proxy-based estimation
  - A systematic method for preparing accurate, individual estimates from historical data
    - You're only as good as your data
    - You still need data (Tinstaafl)
    - Requires that new work relate closely to previous work

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

❹ Combine documents

© SSQC All rights reserved. Version 3
ssqc@concentric.net www.ssqc.com 408-985-4476

Combine documents

- Policies, plans, procedures, work products
- Consolidate into a single document
- Joint authorship (e.g., MRD and SRS)
  - Minimize hand offs, intermediaries
  - Offer options based on scope and size of work
- Simplify and automate “managed and controlled” - use an intranet

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

Combine documents (cont.)

- Evolve and reuse
  - Flow forward
    - Proposal to SDP and software requirements
    - Requirements to high-level design(s)
    - Unit test cases to integration test
  - Flow back
    - Acceptance to integration test

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

Plans across the KPAs and levels

Level	KPA	KP	PLANS
2	SPP	AC7	A plan for the software project
	SSM	AC4	A subcontractor's SDP
		AC12.3	An action plan for any software product that does not pass acceptance
	SQA	AC1	SQA plan
	SCM	AC1	An SCM plan
3	OPF	AC1	Action plans to address findings from the periodic assessments
		AC2	A plan for software process development and improvement activities
	TP	AC1	Project training plan
		AC2	Organization's training plan
	ISM	AC3	The project's software development plan
		AC10.1	A software risk management plan
	SPE	AC5.7	Test plans, test procedures, and test cases
		AC6.1	Plans for integration testing
		AC7.2	Plans for system and acceptance testing
	PR	AC1	Plans for Peer Reviews
4	QPM	AC1	[A project] plan for quantitative process management
	SQM	AC1	The project's software quality plan
5	DP	AC1	The software project [defect prevention] plan
	TCM	GO1	Continuous process improvement [plan]
	PCM	AC3	A plan for software process improvement

Example - Consolidate plans

SPP,
SQA,
SCM

© SSQC All rights reserved. Version 3
ssqc@concentric.net www.ssqc.com 408-985-4476

68

Plan contents: SPP and SQA

SPP - Ac 7

- Purpose, scope, goal, objectives
- Selected life cycle
- Identify procedures, methods, standards
- Identify work products
- Size estimates
- Effort and cost estimates
- Critical computer resource estimates
- Schedule (milestones, reviews)
- Risks
- Engineering facilities and support tools

SQA - Ac 2

- Responsibilities and authority
- Evaluations, audits, and reviews to be performed; procedures for tracking non-compliance; feedback method
- Project standards and procedures
- Documentation to be produced
- Resources required
- Schedule and funding

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

69

Plan contents: SPP and SCM

SPP - Ac 7

- Purpose, scope, goal, objectives
- Selected life cycle
- Identify procedures, methods, standards
- Identify work products
- Size estimates
- Effort and cost estimates
- Critical computer resource estimates
- Schedule (milestones, reviews)
- Risks
- Engineering facilities and support tools

SCM - Ac 2

- Activities to be performed
- Resources required (staffing, tools facilities)
- Schedule of activities

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

70

Transition strategy

© SSQC All rights reserved. Version 3
ssqc@concentric.net www.ssqc.com 408-985-4476

71

Where to start

- RM, SPP, and SPTO
  - Integrated product life cycle
    - Activities and responsibilities
    - Create cross-functional teams
- Project planning and management methods
  - Small projects and cross-functional teams
  - Significant impact of individual performance

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

72

Deployment

- Pilot versus roll-out irrelevant
  - Incremental
- Training versus review and participation
  - 100%
  - Development, review, training roll together
- ✗ Assessment
  - CBA-IPI or SCE or ...

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

73

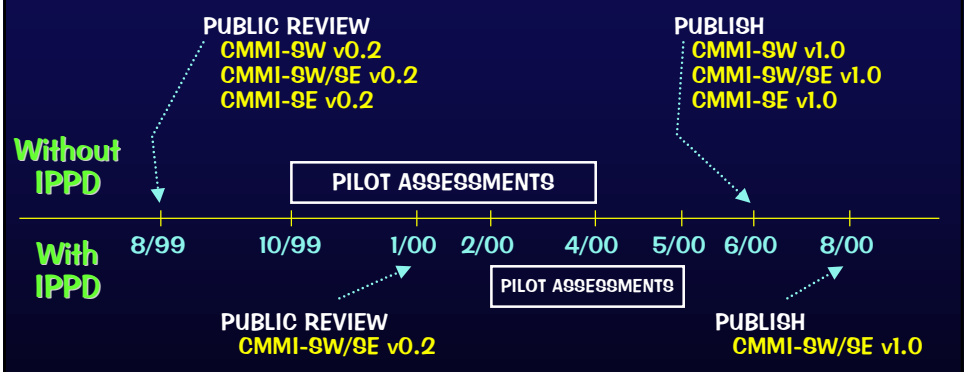
The Future of the CMM

CMMI and small projects

© SSQC All rights reserved. Version 3
ssqc@concentric.net www.ssqc.com 408-985-4476

74

CMMI product release plan



© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

75

IPPD?

- Integrated Product and Process Development

  A management technique that … integrates all essential acquisition activities through the use of multidisciplinary teams … . *"A" Specification for the CMMI Product Suite*, Version 1.3

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

76

CMMI product release plan (cont.)

	CMMI Models	Representation	
		Staged	Continuous
Without IPPD	SW	✘	✘
	SW/SE	✘	✘
	SE	✘	✘
With IPPD	SW/SE	✘	✘

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

77

Staged? Continuous?

- Staged

  Denotes a … structure that includes organizational maturity levels based on groupings of process areas. *"A" Specification for the CMMI Product Suite*, Version 1.3

- Continuous

  Denotes a … structure of processes that provide a continuum of increasing capability for each process area along with a recommended grouping of process areas.

  "A" Specification for the CMMI Product Suite, Version 1.3

	CMMI Models	Representation	
		Staged	Continuous
Without IPPD	SW	✘	✘
	SW/SE	✘	✘
	SE	✘	✘
With IPPD	SW/SE	✘	✘

Topics - review

- Background, guidance, assumptions
- Mapping the CMM
  - Levels, ratings, and other traps
  - Tailoring techniques and examples for small projects
- Strategy for transitioning to the CMM
  - Appraisal methods - return on investment
- The future of the CMM: CMMI

Throughout the session
- At the KPA and key practice levels
- Within and across KPAs and Levels

❶ Select KPAs
❷ Translate definitions and conventions
❸ Consolidate practices (e.g., reviews) and share responsibilities
❹ Combine documents (e.g., plans)

- Lifecycle
- Project management
- Integrated teams

Check out the continuous representations (with or without IPPD) scheduled for availability in 1999 and 2000

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

79

Contact Information

Bill Deibler
Software Systems Quality Consulting
2269 Sunny Vista Drive
San Jose, CA 95128
Phone 408-985-4476 Fax 408-248-7772
deibs@ssqc.com

www.ssqc.com

© SSQC All rights reserved. Version 6
ssqc@concentric.net www.ssqc.com 408-985-4476

80

Thoughts on Oracles & Software Test Automation

Cem Kaner, JD, PhD, ASQ-CQE
Doug Hoffman, BA, MBA, MSEE, ASQ-CSQE

Quality Week. May, 1999.

Acknowledgment

- **Many of the ideas in this presentation were presented and refined at the 5th Los Altos Workshop on Software Testing. LAWST 5 participants included:**
  - **Chris Agruss (Autodesk), James Bach (RST), Jack Falk (softGear technologies), David Gelperin (Software Quality Engineering), Elisabeth Hendrickson (Quality Tree Software), Doug Hoffman (Software Quality Methods), Bob Johnson (Coyote Valley Software), Cem Kaner (KANER.COM), Brian Lawrence (Coyote Valley Software), Noel Nyman (Microsoft), Jeff Payne (RST), Johanna Rothman (Rothman Consulting Group), Melora Svoboda, Loretta Suzuki (Storage Techologies), Ned Young (softGear technologies).**
  - **(Affiliations are presented for identification only)**

Starting Exercise

Picture how automation would look like in your environment . . .

The GUI Regression Automation Problem

- **Dominant paradigm for automated testing.**
- **Prone to failure because of difficult financing, architectural, and maintenance issues.**
- **Low power (in its traditional form) even if successful.**
- **Extremely valuable under <u>some</u> circumstances.**

THERE ARE MANY ALTERNATIVES
THAT CAN BE MORE APPROPRIATE
UNDER OTHER CIRCUMSTANCES.

If your only tool is a hammer, everything looks like a nail.

Capabilities of Automation Tools

- **Automated testing tools provide special capabilities, such as:**
  - **Analyze source code for bugs**
  - **Design test cases**
  - **Automatically create test cases**
  - **Ease manual creation of test cases**
  - **Execute tests**
  - **Record test events**
  - **Measure software responses to tests (Discovery Functions)**
  - **Determine expected results of tests (Reference Functions)**
  - **Evaluate test results (Evaluation Functions)**
  - **Report results**

Examples of Capabilities

- **Analyze source code for bugs**
  - **LINT, complexity analyzers, syntax checkers, etc.**
- **Design test cases**
  - **Softtest, AETG**
- **Automatically create test cases**
  - **Simulators and other random test case creators. Filter checkers, test data generators**
- **Ease manual creation of test cases**
  - **Keyboard capture**
- **Execute tests**
  - **QA Partner, Visual Test, Spiders, TCL**

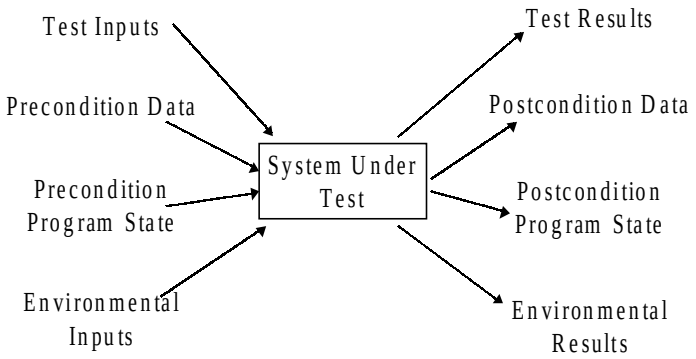
Examples of Capabilities

- **Record test events (flight recorders)**
  - **QA Partner, etc.**
- **Measure software responses to tests (Discovery Functions)**
  - **QA Partner, etc. Also, performance measurement, memory meters, trace tools, event loggers, spiders.**
  - **Note: this includes looking further than superficial results. For example, check the contents of a changed data file, not just the reported, calculated result.**

Capabilities: Examples

- **Determine expected results of tests (Reference Functions)**
  - **Oracles (parallel or inverse functions, simulators, alternate algorithms, or other versions of product).**
  - **Oracles traditionally predict the main result, but expected results might include expected side effects.**
- **Evaluate test results (Evaluation Functions)**
  - **Perform results comparisons**
  - **Analyze differences**
  - **Identify potential faults (e.g.. heuristic evaluators)**
- **Report results**
  - **Report detailed test results and summarize suite results.**

Evaluating Tests: Limited Comparison



A complete evaluation of the system under test would be multi-dimensional

Reference Functions: Examples

- **Parallel function**
  - **previous version**
  - **competitor**
  - **standard function**
  - **custom model**
- **Inverse function**
  - **mathematical inverse**
  - **operational inverse (e.g. split a merged table or round outbound conversion routines back to inbound converters)**
- **Statistical distribution**
  - **test for outliers, means, predicted distribution**
- **Saved result from a previous test. (Consistency test)**

Evaluation Functions: Examples

- Compare (apparently) sufficiently complete attributes
  - compare calculated results of two parallel math functions (but ignore duration, available memory, pointers, display)
- Compare incidental but informative attributes
  - durations
- Check (apparently) insufficiently complete attributes
  - ZIP Code entries are 5 or 9 digits. ZIP codes starting with 9 are not from a NY dataset.
- Check probabilistic attributes
  - X is usually greater than Y. More customers in NY than Wyoming.

Strategies for Automation

What characteristics of the
- goal of testing
- software under test
- environment
- generator
- reference function
- evaluation function
- users
- risks

would support, counter-indicate, or drive you toward a strategy?
- small sample, pre-specified values
- exhaustive sample
- random (aka statistical)
- heuristic analysis of a large set
- consistency evaluation

Strategies for Automation

The following slides are a work in progress. They are not complete.

Please consider them in the context of the questions on Slide 12 and try to list:

- more of the relevant characteristics
- more examples of the strategies (e.g. more heuristic rules, more items for consistency comparison, etc.)

Strategy: Small Sample

- The small sample strategy is about limiting the number of tests used to exercise a product. The most common approach is to divide a large population of possible tests into subsets and to choose a few values that are representative of each set.
- Automated regression testing is a typical case:
  - Execute tests
  - Measure software responses
  - Evaluate by comparison (typically of screen output or screen-related output) against a saved result

Favorable Conditions: Small Sample

- **Goal of Testing**
  -
- **Software Under Test**
  - **Regular function or any other input or output domain that is well-tested by a small group of representative values (such as boundary values)**
  -
- **Environment**
  - **Environment or data cost high (e.g. Beizer's report of costs of Y2K time machine tests)**
  -
- **Generator**
  - **High cost to generate test cases (e.g., no automated generator)**
  -

Favorable Conditions: Small Sample

- **Generator**
  - **High cost to generate test cases (e.g., no automated generator)**
  -
- **Reference Function**
  - **High cost to generate comparison data (e.g., no oracle)**
  -
- **Evaluation Function**
  - **Automated evaluation is slow, expensive**
- **Users**
  - **Tolerant of errors**
- **Risks**
  - **Low risk**

Favorable Conditions: Small Sample

NOTE:

FAVORABLE CONDITIONS ARE NEITHER NECESSARY NOR SUFFICIENT.

Evaluation: Small Sample

- **Advantages**
  - **Identifies results of changes**
  - **Automated comparisons are straightforward**
  - **Product can be oracle for itself**
- **Disadvantages**
  - **Saved results may contain unrecognized errors**
  - **Doesn't necessarily consider specific, key data values, especially special cases not at visible boundaries**
  - **Already-missed errors will remain undetected by repeated regression tests**
  - **If this testing is done before SW is cooked, then the code becomes tailored to the tests**
- **The fundamental problem is, it only checks a few values (we don't know anything about the rest).**

Regression Automation

- **Regression tools dominate the automated testing market.**
  - **Why automate tests that the program has already passed?**
  - **Percentage of bugs found with already-passed tests is about 5-20%**
  - **Efficiency of regression testing shows up primarily in the next version or in a port to another platform.**
- **If the goal is to find new bugs quickly and efficiently, we should use a method based on new test cases.**

Strategy: Exhaustive

Exhaustive testing involves testing all values within a given domain, such as:

- **all valid inputs to a function**
- **compatibility tests across all relevant equipment configurations.**

Example: Testing the MASPAR square root function.

Favorable Conditions: Exhaustive

- **Goal of Testing**
  -
- **Software Under Test**
  - **Limited input domain**
  -
- **Environment**
  - **The range of environments is limited: embedded software or system configuration that is fully controlled by vendor.**
  - **The important parameters (key elements of the environment) can be identified and are known.**
  -
- **Generator**
  - **Easy to create tests**
  -

Favorable Conditions: Exhaustive

- **Reference Function**
  - **Oracle available**
  -
- **Evaluation Function**
  - **Evaluation function available**
  -
- **Users**
  - **?**
  -
- **Risks**
  - **Safety-critical or business-critical**
  -

Evaluation: Exhaustive

- **Advantages**
  - **Complete management of certain risks**
  - **Discover special case failures that are not visible at boundaries or suggested by traditional test design approaches.**
  -
- **Disadvantages**
  - **Expensive**
  - **Often impossible**
  -
- **Bottom line**

Strategy: Random

- **Random (or statistical or stochastic) testing involves generating test cases using a random number generator. Because they are random, the individual test cases are not optimized against any particular risk. The power of the method comes from running large samples of test cases.**
- **Typical examples:**
  - **<u>Random</u>: Function equivalence testing**
  - **<u>Stochastic</u>: Random transition from state to state. Complex simulations, involving long series of events or combinations of many variables.**
  - **<u>Statistical</u>: Generate many tests in order to estimate reliability (e.g. clean room)**

Favorable Conditions: Random

- **Goal of Testing**
  -
- **Software Under Test**
  - **knockoff of a successful competitor**
  - **upgrade from a working program**
  - **conditions under test are very complex**
  -
- **Environment**
  -
- **Generator**
  - **Random inputs through a generator function, such as creating random formulas for a spreadsheet**
  -

Favorable Conditions: Random

- **Reference Function**
  - **Need *some* way to evaluate pass or fail. For example, compute the value of a formula from a reference spreadsheet.**
  -
- **Evaluation Function**
  - **Must be available.**
  -
- **Users**
  -
- **Risks**
  - **Significant errors that involve complex sequences of states or combinations of many inputs**
  -

Evaluation: Random

- **Advantages**
  - **Can run a huge number of test cases**
  - **Few or no evaluation errors**
- **Disadvantages**
  - **Doesn't consider specific, key data values (no special allowance for boundaries, for example).**
- **Risks**
  - **People sometimes underestimate the need for a good oracle. They run so many tests that they think they are doing powerful work even though they are merely testing for crashes.**
  - **Risk of false negatives (oracle has same errors as software under test, so no bug is discovered)(see Leveson's work).**
  - **Risk of overestimating coverage--miss need for *other* types of tests to check for risks not tested for by this series of tests. E.G., might test individual functions but miss need to check combinations.**

Strategy: Heuristic

- **Heuristics are rules of thumb that support but do not mandate a given conclusion. We have partial information that will support a probabilistic evaluation. This won't tell you that the program works correctly but it can tell you that the program is broken. This can be a cheap way to spot errors early in testing.**
- **Typical case:**
  - **Predict a characteristic (such as ZIP code is 5 or 9 digits, or X is usually less than Y) and check it against a large random sample or a complete input or output domain.**
- ***See Hoffman's papers on these for more details and examples.***

Favorable Conditions: Heuristic

- **Goal of Testing**
  - **Early testing for plausibility of results**
- **Software Under Test**
- **Environment**
- **Generator**
- **Reference Function**
  - **Usually there are multiple choices for oracles (can select "best" for the circumstances).**
  -

Favorable Conditions: Heuristic

- **Evaluation Function**
  -
- **Users**
  -
- **Risks**
  - **The risks that you manage by this type of testing are based on your knowledge of any testable fact about code or data that might be proved false by testing across a large set of data.**
  -

Evaluation: Heuristic

- **Advantages**
  - **May allow exhaustive testing of values of inputs (or results).**
  - **Allows discovery of problems early in testing**
  - **Heuristic oracles are often reusable.**
- **Disadvantages**
  - **The results are not definitive.**
    - **Some tests can tell you that the program is broken but not that the program is correct.**
    - **Some tests will indicate failure when the program is actually correct.**
- **Bottom line**
  - **Handy, powerful for early detection, but should not be the only test type that you use.**

Strategy: Consistency

- **Consistency-based testing involves comparing the results of today's test with a prior result. If the results match (are consistent), the program has "passed" the test.**
- **More generally, A/B comparison where the set {B} is a finite set of reference data, not a program that generates results. (Example, A/B PDF file tests.)**
- **Typical case: Traditional automated regression test.**
  - **Run a test manually. If the program passes the test, automate it. (Create a script that can replay the test procedure, create a reference file containing screen output or result data). Then rerun the script, and compare the results to the reference file.**

Favorable Conditions: Consistency

- **Goal of Testing**
  -
- **Software Under Test**
  - **For a GUI-based test, uses standard controls, not custom controls.**
  - **Hooks provided (e.g. API) for testing below the UI level**
  -
- **Environment**
  -
- **Generator**
  - **Expensive to run tests in order to create reference data. Therefore it is valuable to generate test results once and use them from archives.**
  -

Favorable Conditions: Consistency

- **Reference Function**
  - **captured screen, captured state, captured binary output file, saved database**
  - **duration of operation, amount of memory used, exiting state of registers, or other incidental results.**
  - **finite set of reference data against which we can compare current behavior.**
  -
- **Evaluation Function**
  -
- **Users**
  -
- **Risks**
  -

Evaluation: Consistency

- **Advantages**
  - **The program can serves as its own oracle**
  - **Effective when test cases are very expensive or when the software design is very stable**
- **Disadvantages**
  - **Every time the software changes, tests that relied on that characteristic of the software must change. Unless the test code is carefully architected, the maintenance cost is impossible.**
  - **Common mode of failure errors won't be detected. Legacy errors won't be detected.**
- **Bottom line**
  - **Potentially, a good tool for regression testing.**

Confounding Factors

- **The displayed (or printed) value may not be the same as that generated by the SUT. [Interface defects]**
- **Assumptions made may not be valid and need to be reconfirmed during and after testing.**
- **Smart tools limit visibility into actual SUT behaviors (smart tools –> less tester control).**

About Cem Kaner

Cem Kaner tests software and software-related legislation.

The senior author of ***Testing Computer Software***, Kaner has worked with computers since 1976, doing and managing programming, user interface design, testing, and user documentation. Through his consulting firm, KANER.COM, he teaches courses on black box software testing and consults to software publishers on software testing, documentation, and development management. Kaner is also the founder and co-host of the Los Altos Workshop on Software Testing. He is writing a new book, ***Good Enough Testing***, with James Bach and Brian Marick.

An attorney whose practice is focused on the law of software quality, Kaner usually represents customers and individual developers or small consulting firms. He is active (as an advocate for customers, authors, and small development shops) in several legislative drafting efforts involving software licensing, software quality regulation, and electronic commerce. He has recently published a new book, ***Bad Software: What To Do When Software Fails*** (with David Pels. John Wiley & Sons, 1998).

Kaner holds a B.A. in Arts & Sciences (Math, Philosophy), a Ph.D. in Experimental Psychology (Human Perception & Performance: Psychophysics), and a J.D. (law degree). He is Certified in Quality Engineering by the American Society for Quality.

About Doug Hoffman

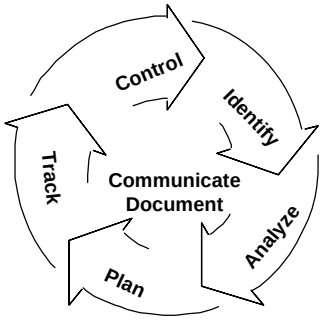
Douglas Hoffman has fifteen years experience in creating and transforming software quality and development groups, and twenty years of management experience. He has been a participant at dozens of software quality conferences and has been Program Chairman for several international conferences on software quality. He has architected test automation environments for several commercial systems and software companies, and has been an active participant in the Los Altos Workshops on Software Testing (LAWST).

He is an independent consultant with Software Quality Methods, LLC. He has been in the software engineering and quality assurance fields for over 25 years and now teaches courses and consults with Silicon Valley companies in strategic and tactical planning for software quality. He has been elected Chairman of the Santa Clara Valley Software Quality Association (SSQA) five. He is active as a Senior Member in the ASQ, participating in the Software division, the Software Quality Task Group, and the ISO 9000 Task Group, and is also a member of the ACM and IEEE. He has earned an MBA as well as an MS in Electrical Engineering and BA in Computer Science, was among the first to earn a Certificate from ASQ in Software Quality Engineering, and has been a registered ISO 9000 Lead Auditor.

Continuous Risk Management at NASA

Dr. Linda Rosenberg
NASA GSFC
301-286-0087
Linda.Rosenberg@gsfc.nasa.gov

Ted Hammer
NASA GSFC
301-286-7123
thammer@pop300.gsfc.nasa.gov



http://satc.gsfc.nasa.gov

Overview

- Background
  - Risk Management Guidance
  - Course Objectives
  - Course Outline
- NASA's Continuous Risk Management Course
  - Risk Management Functions
  - Course Evaluations
- Implementation

Background

NASA Risk Management Guidance

- NPG 7120.5A, “NASA Program and Project Management Processes and Requirements,” April 3, 1998
  - RM is one of 7 major themes;
  - Section 4.2 establishes RM requirements;
  - Stresses RM as an integral part of program/project management.
- NPD 2820, “Software Policy”
- SMA independent assessments of programs and projects are RM-based.

Continuous Risk Management (CRM) Course

Taught by Software Assurance Technology Center, Goddard Space Flight Center

Course #NASA-FSFC-SATC-98-001

Text: Software Engineering Institute at Carnegie Mellon University, *Continuos Risk Management Guidebook*, 1996.

NTIS#: AD-A319533KKG

DTIC#: AD-A319533\6\XAB

NASA Risk Management Course Objectives

⇒ Understand the concepts and principles of Continuous Risk Management and how to apply them

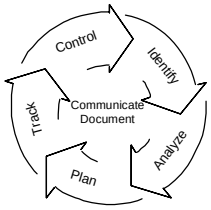
⇒ Develop basic risk management skills for each function of Continuous Risk Management

⇒ Be able to use key methods and tools

⇒ Be able to tailor Continuous Risk Management to a project or organization

CRM Course Outline- Day 1

1 Introduction
2 Continuous Risk Management Paradigm
   - 2-1 Overview
   - 2-2 Identify
   - 2-3 Analyze
   - 2-4 Plan
   - 2-5 Track
   - 2-6 Control
   - 2-7 Communicate and Document
3 Risk Management Example Implementation
4 Getting Started in Continuous Risk Management



CRM Workshop Outline - Day 2

Objectives: 1 - Partial completion of risk management plan
2 - Initial risk identification and mitigation

Steps:

Organizational chart	Risk identification
Roles and responsibilities	Writing risk statements
Meeting schedule	Mitigation strategies
Tools and methods	Tracking data
Attribute definitions	
timeframe, probability, impact	

NASA's Continuous Risk Management Course

Definitions of Risk

Risk involves the <u>likelihood</u> that an undesired event will occur.

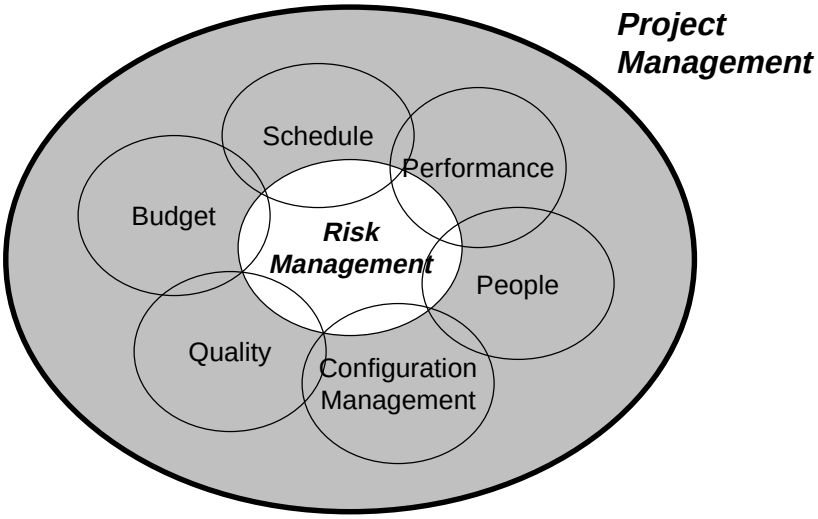
Qualitative or Quantitative

Risk involves the <u>severity of consequence</u> of the event should it occur

Qualitative or Quantitative

Risk = Likelihood * Severity

Risk Management & Project Management



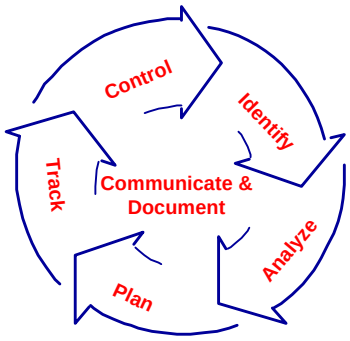
Why Do Risk Management?

- Early identification of potential problems
- Increase chances of project success
- Enable more efficient use of resources
- Promote teamwork by involving personnel at all levels of the project
- Information for tradeoffs based on priorities and quantified assessment

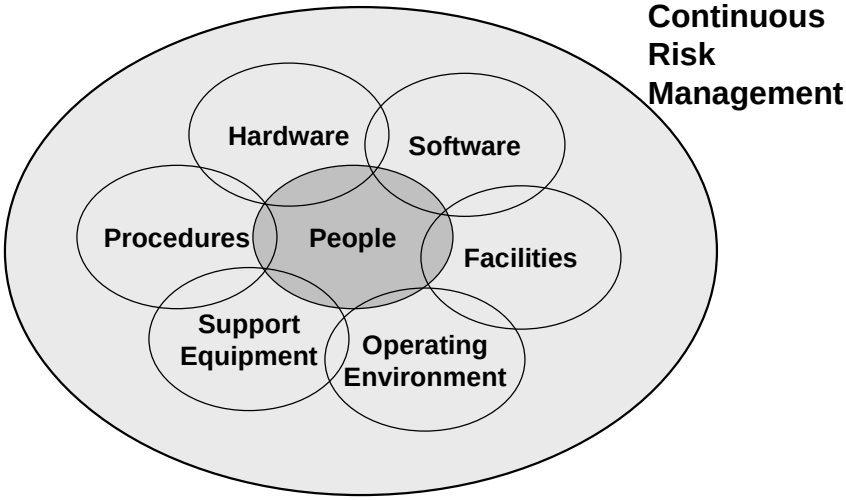
Relationship Among Functions

Throughout the project life cycle, risk components evolve

- **continuously**
- **concurrently**
- **iteratively**



Where is Continuous Risk Management Applied?



Relationship to Everyday Practice

Learning
Continuous Risk Management
is similar to incorporating
any new habit
into your daily life.

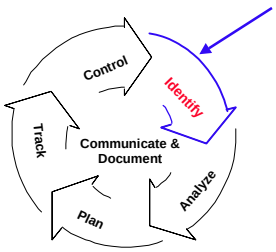
1 - Identify

Purpose

Search for and locate risks before they become problems

Description

The process of transforming uncertainties and issues about a project into distinct (tangible) risks that can be described and measured



Components of a Risk Statement

Given the Condition ; *there is a possibility that* → Consequence *will occur*

Risk Statement

Condition: a single phrase briefly describing current key circumstances, situations, etc. that are causing concern, doubt, anxiety, or uncertainty

Consequence: a single phrase or sentence that describes the key, negative outcome(s) of the current conditions

Example Context

Risk statement:

This is the first time that the software staff will use OOD; the staff may have a lower-than-expected productivity rate and schedules may slip because of the associated learning curve.

Context:

Object oriented development is a very different approach that requires special training. There will be a learning curve until the staff is up to speed. The time and resources must be built in for this or the schedule and budget will overrun.

Risk Statement & Context

Condition → Consequence
Risk Statement

A good risk statement:

- Contains at least one condition
- Contains at least one consequence
- Is clear and concise

Good context:

- Provides additional information not in the risk statement
- Ensures that the original intent of the risk can be understood, particularly after time has passed

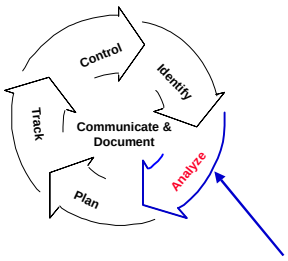
2 - Analyze

Purpose

- Convert risk data into decision-making information

Description

- The process of examining the risks in detail to determine the extent of the risks, how they relate to each other, and which ones are the most important



Evaluating Attributes of Risk

Purpose:

to gain a better understanding of the risk by determining the expected impact, probability, and timeframe of a risk

Description - involves establishing values for:

Impact: the loss or effect on the project if the risk occurs

Probability: the likelihood the risk will occur

Timeframe: the period when you must take action to mitigate the risk

Levels of Analysis

Level	*Impact*	*Probability*	*Timeframe*
binary level	significant insignificant	likely not likely	near far
tri-level	high moderate low	high moderate low	near mid far
5-level	very high high moderate low very low	very high high moderate low very low	imminent near mid far very far
n-level	n levels of impact	n levels of probability	n levels of timeframe

Classification Perspectives

By Source: Risks are grouped based on the same source or root cause. This will show the major sources of risk to the project.

By Impact: Risks are grouped based on where or how the impact will be felt by the project. This shows the major aspects of the project that are most at risk.

Analysis Activities

Evaluate:
- impact (I)
- probability (P)
- timeframe (T)

Classify:
- identify duplicates
- consolidate risks to sets

Prioritize:
- identify Pareto top N
- rank top N

Risk	I	P	T
Risk a	M	M	F
Risk b	M	L	N
Risk c	L	H	N
. . .			

Consolidate risks

Risk	I	P	T
Risk set A ----- -----	H	M	F
Risk b	M	L	N
Risk c	L	H	N
. . .			

Sort by evaluation results

Pareto top N

Risk	I	P	T
Risk n	H	H	N
Risk s	H	M	N
Risk set A ----- -----	H	M	F
Risk c	L	H	N

Rank order the Pareto top N

Top N
1.
2.
3.
. . .

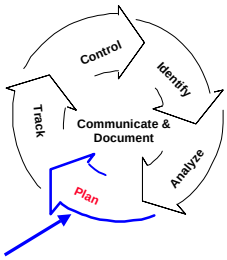
3 - Plan

Purpose

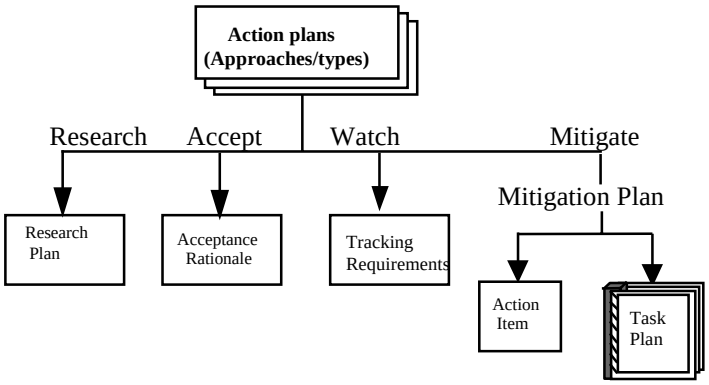
Translate risk information into decisions and mitigating actions (both present and future), and implement those actions

Description

The process of deciding what, if anything, should be done about a risk or set of related risks



Action Plan Approaches



Mitigation Goals and Success Measures

Set a realistic, measurable (or verifiable) goal for mitigating the risk, for example

- avoid any changes to scheduled milestones
- eliminate change requests unsupported by funding to implement the change

Define success criteria— you need to know when you've succeeded or failed

For example

- all current change requests implemented by 3/1/96 with no change to scheduled milestones

Contingency Plans

Not all mitigation plans can or should be carried out immediately, for example:

- there may not be sufficient funding at this time
- other circumstances (such as having the right personnel) may not be right
- it may be a low probability, catastrophic impact risk with an expensive mitigation plan

May be used as Plan B if Plan A fails

Contingency plans are held in reserve until specific conditions are true or certain events occur

- watch for the conditions and events!

4 - Track

Purpose

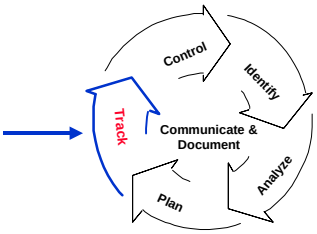
Monitor risk indicators and mitigation actions

Is the plan followed?

Is the risk reduced?

Description

The process in which risk status data are acquired, compiled, and reported



Tracking Risks and Plans

1. Tracking the <u>mitigation plan</u> will indicate whether the plan is being executed correctly if the plan is on schedule

2. Tracking the <u>risk attributes</u> will indicate mitigation plan effectiveness

Risk Metrics

Measure attributes of a risk

- impact, probability, and timeframe
- other risk-specific attributes

Assess the impact or success of a mitigation plan

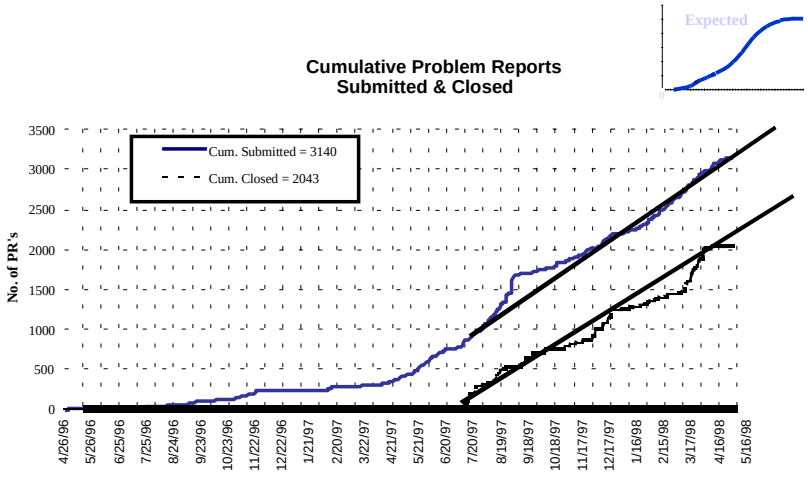
Chosen during planning

Provide meaningful information to enable more informed control decisions

Triggers

- provide early warning of an impending critical event
- indicate the need to implement a contingency plan to preempt a problem

Testing Metrics Example - Tracking Errors/Faults/Changes



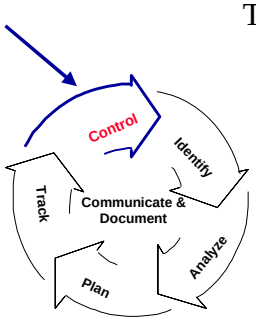
5 - Control

Purpose

To make informed, timely, and effective decisions regarding risks and their mitigation plans

Description

The process that takes the tracking status reports for the project risks and decides what to do with the risks based on the reported data



Decide

Purpose:

ensure that project risks continue to be managed effectively

Description:

uses tracking data to determine how to proceed with project risks

- close
- continue tracking and executing the current plan
- replan
- invoke a contingency plan

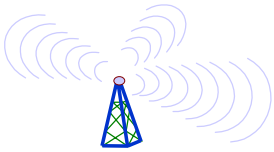
6 - Communicate & Document

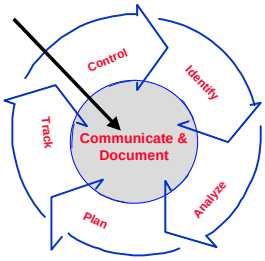
Purpose

Provide information and feedback to the project on the risk activities, current risks, and emerging risks

Description

A process in which risk information is conveyed between all levels of a project team

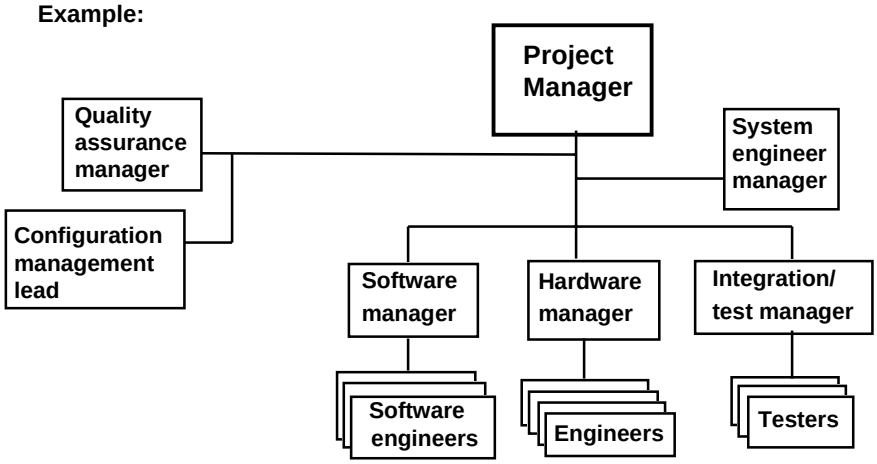




Implementation

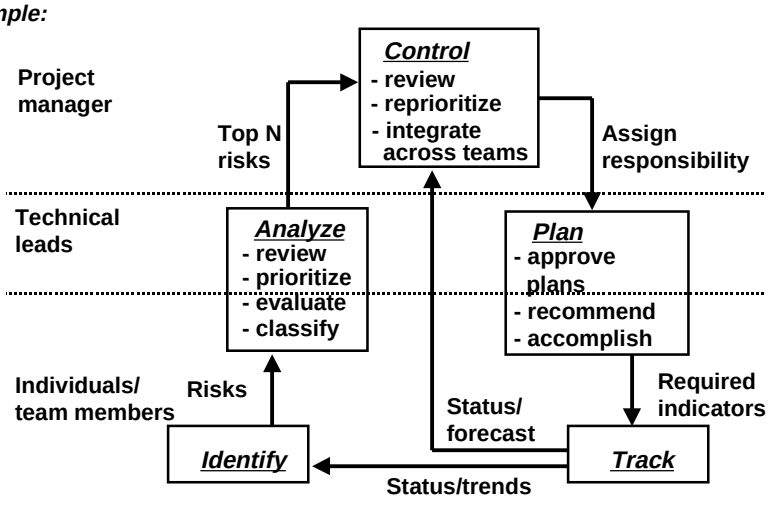
You need an . . .
Organization Structure

Example:



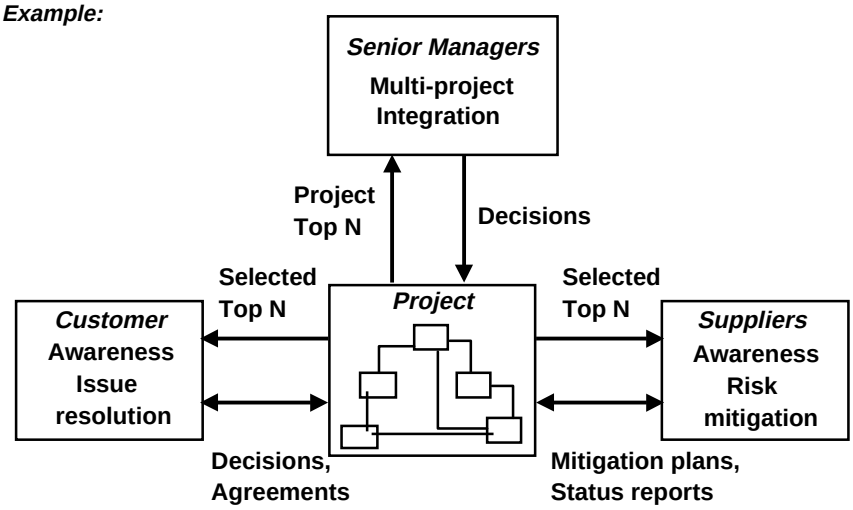
You need . . .
Internal Communication

Example:



You need . . .
External Communication

Example:



You need . . .
Assigned Roles & Responsibilities

Example:
IR-SIP Personnel

Who	Responsibilities
Individuals	Software engineers, testers, leads, and project manager • identify new risks • estimate probability, impact, and timeframe • classify risks • recommend approach and actions • track risks and mitigation plans (acquire, compile, and report)
S/W CSCI, CM, and Test Managers	Software engineering leads for each CSCI • ensure accuracy of probability/impact/timeframe estimates and the classification • review recommendations on approach and actions • build action plans (determine approach, define scope & actions) • report their top N risks and issues to the project manager • collect and report general risk management measures/metrics
Software Project Manager, Hardware Project Manager, etc.	• integrates risk information from all technical leads • reprioritizes all risks to determine top N risks in each area (software, hardware, etc.) • makes control decisions (analyze, decide, execute) for risks (e.g., Software Project Manager controls software risks) • authorizes expenditure of resources for mitigation • assigns or changes responsibility for risks and mitigation plans within the CSCI, CM, and test areas • handles communication IR-SIP project manager
IR-SIP Project Manager; IR-SIP Project Systems Engineer	• integrates risk information from all software, hardware, and CM leads • reprioritizes all risks to determine top N project risks • makes control decisions (analyze, decide, execute) for Top N project risks • authorizes expenditure of project resources for mitigation • assigns or changes responsibility for risks and mitigation plans within the project (e.g., moving responsibility for a risk from software to hardware) • handles communication with AA program manager • reviews general risk management measures/metrics with Quality Assurance during each quarter to evaluate effectiveness of risk management

You need . . .
Established Meetings

Weekly Team Meetings
- establish priority of team's risks
- assign responsibility for new risks
- review and approve mitigation plans

Monthly Project Meetings
- Leads present the team's Top N risks (and mitigation plans)
- Project manager Leads decide on appropriate action
- Project manager determines allocation of resources for mitigation discretionary funds for technical leads

You must ...

Document your Risk Management Plan

Baseline a set of risks

Obtain training and project familiarization

You must choose your . . .
Methods and Tools

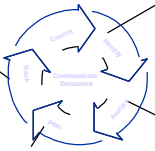
Risk Management Plan

A Risk Management Plan documents how risks will be managed on a project: the process, activities, milestones, and responsibilities associated with risk management. It is a subset of the project plan and is written before the project begins.

Control
- Cause and Effect Analysis
- Closing a Risk
- Cost-Benefit Analysis
- List Reduction
- Mitigation Status Report
- Multivoting
- PERT Chart
- Problem-Solving Planning
- Risk Information Sheet
- Spreadsheet Risk Tracking
- Stoplight Chart
- Project Metrics

Identify
- Baseline Identification and Analysis
- Brainstorming
- Periodic Risk Reporting
- Project Profile Questions
- Risk Information Sheet
- Short TBQ
- Taxonomy-Based Questionnaire (TBQ)
- TBQ Interviews
- Voluntary Reporting
- Project Metrics
- FMEA
- FTA

Track
- Bar Graph
- Mitigation Status Report
- Risk Information Sheet
- Spreadsheet Risk Tracking
- Stoplight Chart
- Time Correlation Chart
- Time Graph
- Project Metrics
- SPC



Plan
- Action Item List
- Baseline Planning
- Planning Decision Flowchart
- Planning Worksheet
- Problem-Solving Planning
  - Affinity Grouping
  - Brainstorming
  - Cause and Effect Analysis
  - Cost-Benefit Analysis
  - Gantt Charts
  - Goal-Question-Measure
  - Interrelationship Digraph
  - List Reduction
  - Multivoting
  - PERT Chart
  - Work Breakdown Structure
- Risk Information Sheet
- WCA

Analyze
- Affinity Grouping
- Baseline Identification and Analysis
- Binary Attribute Evaluation
- Comparison Risk Ranking
- Multivoting
- Pareto Top N
- Potential Top N
- Taxonomy Classification
- Risk Information Sheet
- Top 5
- Tri-level Attribute Evaluation
- FMEA
- FTA

You should carefully . . .
Adapt to Your Project

Purpose:

make maximum use of existing, effective project management processes and methods while integrating a set of proactive risk management activities

document the tailored processes, methods, and tools in a risk management plan

define a schedule for transitioning specific methods, tools, and activities into the project

Description:

tailors risk management processes, methods, and tools for use on the project

You must choose your . . .
Risk Database

A database is the simplest means of retaining and keeping risk information up to date.

Data entry forms and reports can be used as the risk information sheet, spreadsheet, and other templates.

A risk database enables documentation of lessons learned, trend analysis, pattern analysis to support identifying common risks (and solutions) across projects.

Hints and Tips

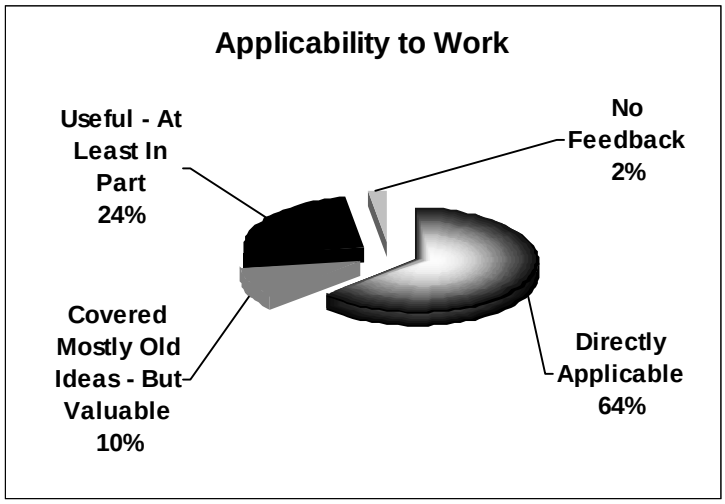
Start simple; learn to “think risk.”

Never throw out or ignore any risk information; scan it once in a while.

Always ask for feedback on how things are going and what works.

Use outside facilitators until you’re comfortable with the processes.

Course Evaluations



References

Ted Hammer, GSFC
301-286-7123
thammer@pop300.gsfc.nasa.gov

Dr. Linda Rosenberg
301-286-0087
Linda.Rosenberg@gsfc.nasa.gov

Continuous Risk Management at NASA

Dr. Linda H. Rosenberg
Unisys @ NASA GSFC SATC
Bld 6 Code 300.1
Greenbelt, MD 20771
301-286-0087
Linda.Rosenberg@gsfc.nasa.gov

Theodore Hammer
NASA GSFC
Bld 6 Code 302
Greenbelt, MD 20771
301-286-7123
Thammer@pop300.gsfc.nasa.gov

Albert Gallo
Unisys @ NASA GSFC SATC
Bld 6 Code 300.1
Greenbelt, MD 20771
301-286-8012
agallo@mail.hst.nasa.gov

Abstract

NPG 7120.5A, "NASA Program and Project Management Processes and Requirements" enacted in April, 1998, requires that "The program or project manager shall apply risk management principles…" The Software Assurance Technology Center (SATC) at NASA GSFC has been tasked with the responsibility for developing and teaching a systems level course for risk management that provides information on how to comply with this edict. This risk management structure of functions has been taught to projects at all NASA Centers and is being successfully implemented on many projects. The course was developed in conjunction with the Software Engineering Institute at Carnegie Mellon University, then tailored to the NASA systems community. This presentation will briefly discuss the six functions for risk management: (1) Identify the risks in a specific format; (2) Analyze the risk probability, impact/severity, and timeframe; (3) Plan the approach; (4) Track the risk through data compilation and analysis; (5) Control and monitor the risk; (6) Communicate and document the process and decisions.

This paper will give project managers the information needed to implement Continuous Risk Management successfully at a cost they can afford.

Introduction

Software risk management is important because it helps avoid disasters, rework, and overkill, but more importantly because it stimulates win-win situations. The objectives of software risk management are to identify, address, and eliminate software risk items before they become threats to success or major sources of rework. In general, good project managers are also good managers of risk. It makes good business sense for all software development projects to incorporate risk management as part of project management. NPG 7120.5A, the NASA guidebook for project managers, requires risk management applications and includes a section briefly discussing what should be included in a risk management plan. A course in continuous risk management was developed by the Software Engineering Institute at Carnegie Mellon University and has been adapted to NASA by the Software Assurance Technology Center (SATC) at NASA GSFC. The course was first taught in January, 1998, and has since been taught to over 300 students at all NASA centers.

There are a number of definitions and uses for the term risk, but there is no universally accepted definition. What all definitions have in common is agreement that risk has two characteristics:

uncertainty: An event may or may not happen.

loss: An event has unwanted consequences or losses.

Therefore, risk involves the likelihood that an undesirable event will occur, and the severity of the consequences of the event, should it occur. Risk management can:

- Identify potential problems and deal with them when it is easier and cheaper to do so—before they are problems and before a crisis exists.
- Focus on the project's objective and consciously look for things that may affect quality throughout the production process.
- Allow the early identification of potential problems (the proactive approach) and provide input into management decisions regarding resource allocation.
- Involve personnel at all levels of the project; focus their attention on a shared product vision, and provide a mechanism for achieving it.
- Increase the chances of project success.

At NASA, we focus on Continuous Risk Management that can be applied to any development process: hardware, software, systems, etc. It provides a disciplined environment for proactive decision making to:

- assess continually what could go wrong (risks)
- determine which risks are important to deal with
- implement strategies to deal with those risks
- assure, measure effectiveness of the implemented strategies

Risk management must not be allowed to become "shelfware". The process must be a part of regularly scheduled periodic product management. It requires identifying and managing risks routinely throughout all phases of the project's life. The paradigm shown in Figure 1 illustrates the set of continuous risk management functions throughout the life cycle of a project. These functions serve as the foundation for the application of continuous risk management. Each risk nominally goes through these functions sequentially, but the activity occurs continuously, concurrently, and iteratively. Risks are usually tracked in parallel while new risks are identified and analyzed, and the mitigation plan for one risk may yield another risk.

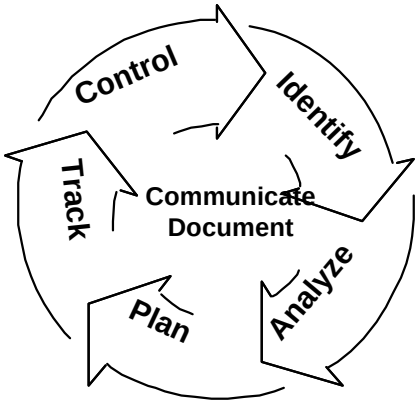


Figure 1: Continuous Risk Management Principle Functions

Continuous Risk Management Principle Functions

1- Identify

The purpose of identification is to consider risks before they become problems and to incorporate this information into the project management process. Anyone in a project can identify risks to the project. Each individual has particular knowledge about various parts of a project. During Identify, uncertainties and issues about the project are transformed into distinct (tangible) risks that can be described and measured.

During this function, all risks are written with the same, two part format. The first part is the risk statement, written as a single statement concisely specifying the cause of the concern as well as its impact. The second part may contain additional supporting details in the form of a context.

The aim for a risk statement is that it be clear, concise, and sufficiently informative that the risk is easily understood. Risk statements in standard format must contain two parts: the condition and the consequence. The condition-consequence format provides a complete picture of the risk, which is critical during mitigation planning. It is read as follows:

*given the <**condition**> there is a possibility that <**consequence**> will occur*

The *condition* component focuses on what is currently causing concern; it must be something that is true or widely perceived to be true. This component provides information that is useful when determining how to mitigate a risk. The *consequence* component focuses on the intermediate and long-term impact of the risk. Understanding the depth and breadth of the impact is useful in determining how much time, resources, and effort should be allocated to the mitigation effort. A well-formed risk statement usually has only one condition, but may have more than one consequence.
Risk statements should avoid:

- abbreviations/acronyms that are not readily understood
- sweeping generalizations
- massive, irrelevant detail

Since the risk statement is to be concise, a context is added to provide enough additional information about the risk to ensure that the original intent of the risk can be understood by other personnel, particularly after time has passed. An effective context captures the what, when, where, how, and why of the risk by describing the circumstances, contributing factors, and related issues (background and additional information that are NOT in the risk statement).

A diagram of the complete risk statement and context are shown in Figure 2.

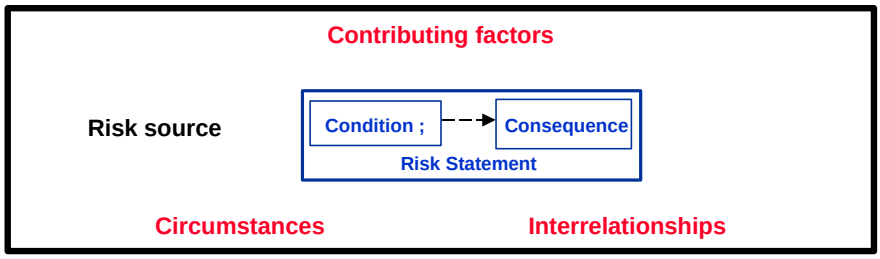


Figure 2: Risk Statement and Context

An example is shown in Figure 3. Note there is one condition and two consequences in the risk statement. The context explains why this is a risk.

<u>Risk statement:</u>

This is the first time that the software staff will use OOD; the staff may have a lower-than-expected productivity rate and schedules may slip because of the associated learning curve.

<u>Context:</u>

Object oriented development is a very different approach that requires special training. There will be a learning curve until the staff is up to speed. The time and resources must be built in for this or the schedule and budget will overrun.

Figure 3: Example Risk Statement and Context

Risk identification depends heavily on both open communication and a forward-looking view to encourage all personnel to bring forward new risks and to plan beyond their immediate problems. Although individual contributions play a role in risk management, teamwork improves the chances of identifying new risks by allowing personnel to combine their knowledge and understanding of the project.

2 - Analyze

The purpose of Analyze is to convert the data into decision-making information. Analysis is a process of examining the risks in detail to determine the extent of the risks, how they relate to each other, and which ones are the most important. Analyzing risks has three basic activities: evaluating the attributes of the risks (impact, probability, and timeframe), classifying the risks, and prioritizing or ranking the risks.

Evaluating - The first step provides better understanding of the risk by qualifying the expected impact, probability, and timeframe of a risk. This involves establishing values for:

Impact: the loss or negative affect on the project should the risk occur
Probability: the likelihood the risk will occur
Timeframe: the period when you must take action in order to mitigate the risk

Figure 4 demonstrates sample values that might be used to evaluate a risk's attributes.

Attribute	Value	Description
Probability	Very Likely (H) Probable (M) Improbable (L)	High chance of this risk occurring, thus becoming a problem > 70% Risk like this may turn into a problem once in a while {30% < x < 70%} Not much chance this will become a problem {0% < x < 30%}
Impact	Catastrophic (H) Critical (M) Marginal (L)	Loss of system; unrecoverable failure of system operations; major damage to system; schedule slip causing launch date to be missed; cost overrun greater than 50% of budget Minor system damage to system with recoverable operational capacity; cost overrun exceeding 10% (but less than 50% of planned cost Minor system damage to project; recoverable loss of operational capacity; internal schedule slip that does not impact launch date cost overrun less than 10% of planned cost
Timeframe	Near-term (N) Mid-term (M) Far-term (F)	Within 30 days 1 to 4 months from now more than 4 months from now *NOTE: refers to* ***when action must be taken***

Figure 4: Sample Attribute Values

Classifying - The next step is to classify risks. There are several ways to classify or group risks. The ultimate purpose of classification is to understand the nature of the risks facing the project and to group any related risks so as to build more cost-effective mitigation plans. The process of classifying risks may reveal that two or more risks are equivalent—the statements of risk and context indicate that the subject of these risks is the same. Equivalent risks are therefore duplicate statements of the same risk and should be combined into one risk.

Prioritize - The final step in the Analysis function is to prioritize the risks. The purpose is to sort through a large number of risks and determine which are most important and to separate out which risks should be dealt with first (the vital few risks) when allocating resources. This involves partitioning risks or groups of risks based on the "vital few" sense and ranking risks or sets of risks based on consistently applying an established set of criteria. No project has unlimited resources with which to mitigate risks. Thus, it is essential to determine consistently and efficiently which risks are most important and then to focus those limited resources on mitigating risks.

Conditions and priorities will change during a project, and this natural evolution can affect the important risks to a project–. ***Risk analysis must be a continual process***. Analysis requires open communication so that prioritization and evaluation is accomplished using all known information. A forward-looking view enables personnel to consider long-range impacts of risks.

3 - Plan

Planning is the function of deciding what, if anything, should be done about a risk or set of related risks. In this function decisions and mitigation strategies are developed based on current knowledge of project risks.

The purpose of plan is to:

- make sure the consequences and the sources of the risk are known

- develop effective plans
- plan efficiently (only as much as needed or will be of benefit)
- produce, over time, the correct set of actions that minimize the impacts of risks (cost and schedule) while maximizing opportunity and value
- plan important risks first

Figure 5 indicates the potential approaches to Risk Planning.

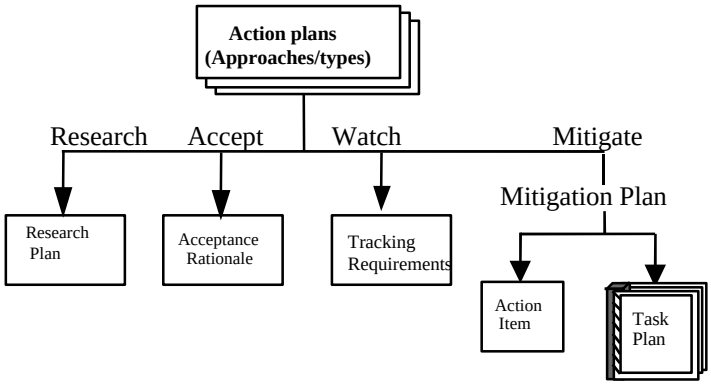


Figure 5: Planning approaches

There are four options to consider when planning for risks:

1. *Research:* establish a plan to research the risk(s)
2. *Accept:* decide to "accept" the risk(s) and document the rationale behind the decision
3. *Watch:* monitor risk conditions for any indications of change in probability or impact (tracking metrics must be established and documented)
4. *Mitigate:* allocate resources and assign actions in order to reduce the probability or potential impact of risks. This can range from simple tasking to sweeping activities:
   Action Items: a series of discrete tasks to mitigate risk
   Task Plan: formal, well-documented and larger in scope

Dealing with risk is a continuous process of determining what to do with new concerns as they are identified and efficiently utilizing project resources. An integrated approach to management is needed to ensure mitigation actions do not conflict with project or team plans and goals. A shared product vision and global perspective are needed to create mitigation actions on the macro level to the benefit the project, customer and organization. The focus of risk planning is to be forward looking, to prevent risks from becoming problems. Teamwork and open communication enhance the planning process by increasing the amount of knowledge and expertise that can be applied to the development of mitigating actions.

4 - Track

Tracking is the process by which risk status data are acquired, compiled, and reported The purpose of Track is to collect accurate, timely, and relevant risk information and to present it in a clear and easily understood manner to the appropriate people/group.

Tracking is done by those responsible for monitoring “watched” or “mitigated” risks. Tracking status information become critical to performing the next function in the Continuous Risk Management paradigm, i.e. Control. Supporting information, such as schedule and budget variances, critical path changes, and project/performance indicators can be used as triggers, thresholds, and risk- or plan-specific measures where appropriate.

When a mitigation plan has been developed for a risk or risk set, both the mitigation plan and the risk attributes are tracked. Tracking the mitigation plan, or even a list of action items, will indicate whether the plan is being executed correctly and/or on schedule. Tracking any changes in the risk attributes will indicate whether the mitigation plan is reducing the impact or probability of the risk. In other words, tracking risk attributes gives an indication of how effective the mitigation plan is.

Program and risk metrics provide decision makers with the information needed for making effective decisions. Normally program metrics are used to assess the cost and schedule of a program as well as the performance and quality of a product. Risk metrics are used to measure a risk’s attributes and assess the progress of a mitigation plan. They can also be used to help identify new risks.

> *Example*: A program metric might look at the rate of module completion. If this metric indicates that the rate of completion is lower than expected, then a schedule risk should be identified.

Open communication regarding risk and mitigation status stimulates the project and risk management process. Tracking is a continuous process - current information about a risk status should be conveyed regularly to the rest of the project. Risk metrics provide decision makers with the information needed for making effective decisions.

5 - Control

The purpose of the Control function is to make informed, timely, and effective decisions regarding risks and their mitigation plans. It is the process that takes in tracking status information and decides exactly what to do based on the reported data. Controlling risks involves analyzing the status reports, deciding how to proceed, and then implementing those decisions.

Decision-makers need to know 1) when or whether there is a significant change in risk attributes and 2) the effectiveness of mitigation plans within the context of project needs and constraints. The goal is to obtain a clear understanding of the current status of each risk and mitigation plan relative to the project and then to make decisions based on that understanding. Tracking data is used to ensure that project risks continue to be managed effectively and to determine how to proceed with project risks. Options include:

- *Replan* - A new or modified plan is required when the threshold value has been exceeded, analysis of the indicators shows that the action plan is not working, or an unexpected adverse trend is discovered.
- *Close the risk* - A closed risk is one that no longer exists or is no longer cost effective to track as a risk. This occurs when: the probability falls below a defined threshold,

impact lies below a defined threshold, or the risk has become a problem and is tracked.

- *Invoke a contingency plan* - A contingency plan is invoked when a trigger has been exceeded or some other related action needs to be taken.
- *Continue tracking and executing the current plan* - No additional action is taken when analysis of the tracking data indicates that all is going as expected or project personnel decide to continue tracking the risk or mitigation plan as before.

Open communication is important for effective feedback and decision-making - a critical aspect of Control. Risk control is also enhanced through integrated management - combining it with routine project management activities enables comprehensive project decision-making.

6 - Communication & Documentation

The purpose of Communicate and Document is for *all* personnel to understand the project's risks and mitigation alternatives as well as risk data and to make effective choices within the constraints of the project. Communication and Documentation are essential to the success of all other functions within the paradigm and is critical for managing risks.

- Identify: In risk identification, risk statements are communicated.
- Analyze: In analysis, project personnel communicate information about impact, probability, and timeframe attributes. Risk classification involves grouping risk information communicated by individuals.
- Plan: During planning, action plans are developed and communicated to project personnel.
- Track: Reports designed to communicate data to decision-makers are compiled during tracking.
- Control: The decisions made during control must be communicated and recorded to project personnel.

For effective risk management, an organization must have open communication and formal documentation. Communication of risk information is often difficult because the concept of risk comprises two subjects that people don't normally deal well with: probability and negative consequences.

Not only Continuous Risk Management, but the project as a whole are in jeopardy when the environment is not based on open communication. No one has better insight into risks than project personnel, and *management needs that input*. Experienced managers know that the free flow of information can make or break any project. Open communication requires:

- Encouraging free-flowing information at and between all project levels
- Enabling forma, informal and impromptu communication

Using consensus-based processes that value the individual voice, bringing unique knowledge and insight to identifying and managing risks.

Implementation

Successful implementation of a Risk Management Plan requires full integration of risk management activities into the existing organizational structure. Risk management is not something that is done off to the side of normal activities, rather it is something that is integrated into the normal business operations of a project and is adapted to the specific project. It should make maximum use of effective project management processes and methods, while integrating a set of proactive risk management activities.

In order to successfully integrate risk management into the existing project infrastructure, internal and external communication must be effective and roles and responsibilities, which identify specific individuals and their tasks, must be assigned. It is essential that every individual working on a project be involved in the risk management process in order to take maximum advantage of existing knowledge and capabilities.

Internally, individual team members should be involved in identifying all possible risks. The risks are then passed to the project technical leads who will analyze them (i.e. review the risks, prioritize them, evaluate and classify them). Once the technical leads have completed analysis, they pass risk information to the Project Manager. The Project Manager will perform Control functions such as reviewing the risks, reprioritizing them, and integrating the risk management process across teams. The Project Manager will assign responsibility for individual risks or groups of risks and the technical leads will begin Plan functions. Technical leads will recommend, approve, and accomplish risk management plans, which will then be tracked and status reports will be generated to monitor progress of the plan. Externally, the Project Manager must establish and maintain effective communication with the customer to ensure that the customer is aware of existing risks and risk management plans and to facilitate the agreement and decision-making processes. The Project Manager needs to establish a solid communication link with suppliers so they are aware of existing and potential risks and so they can work jointly with the Project Manager to mitigate risks and generate status reports. The Project Manager must maintain effective communication with Senior Managers so decisions can be made and to give Senior Managers the most effective tools to effect multi-project integration.

Risk management is a continual process and must be implemented in conjunction with other routine activities. It should be discussed at weekly team meetings, monthly project meetings, and other meetings as appropriate. Weekly team meetings can be used as a forum to establish the priority of a team's risks, to assign responsibility for new risks, and to review and approve risk mitigation plans. Monthly project meetings provide a venue where project leads can present the team's high-priority risks and mitigation plans, where Project Managers and technical leads can decide on appropriate action, and where Project Managers can determine allocation of resources for mitigation discretionary funds for technical leads.

As risk management processes are implemented and decisions are made, the Risk Management Plan must be documented. This ensures the provision of information and

feedback to the project regarding current risks, risk activities, and emerging risks. It also provides a process by which risk information is conveyed between all levels of a project team. A risk database is probably the simplest means of retaining and keeping risk information up-to-date. It enables documentation of lessons learned, trend analysis, and pattern analysis to support identifying common risks and solutions across projects.

Risk management training is possibly the most vital asset to any Risk Management Plan. Most project members are aware of risk management basics, but many are not aware of how to successfully implement a solid Risk Management Plan. Training can help a project clarify the risk management process, develop a Risk Management Plan, establish a baseline set of risks, and act as a team building vehicle. It also provides an awareness of methods and tools that can be utilized in the various risk management phases. Risk management training can provide a project that extra edge needed to succeed.

Conclusion

Most project managers agree that risk management works, but the difficulty lies in actually implementing it, even when required to do so. The risk management plan is often hastily written and then thrown in a corner to gather dust. In addition to the course, one of the steps NASA has taken is to establish a risk management web site (http://satc.gsfc.nasa.gov) that contains sample risk management plans and a schedule of classes. Much time is spent discussing with managers the benefits of taking a formal training course, which is more than recovered by a project when all team members are working toward common goals in a coordinated manner.

Recruiting Software Testers

Cem Kaner, J.D., Ph.D., ASQ-CQE
kaner@kaner.com

Quality Week
May, 1999

For more detail, please see my paper,
"Recruiting Software Testers," on the CD ROM.

Diversity is Essential

- Credentials are not the answer
  - An example: Staffing for a financial application
- Opportunity hires
  - Mutual opportunities: special skills, special deals
    - Temporary assignment to testing
    - Family commitments
    - Training
- The need for consensus to support flexible hiring

Behavioral Interviewing

- The general principle:
  - The goal of the interview is to predict how the candidate will behave if she joins your group. The interview provides a sample of her behavior.
- Questioning
  - Closed vs. open-ended questions
  - Factual vs. opinion questions
  - Hypothetical vs. behavioral questions
- Work samples

Behavioral Interviewing

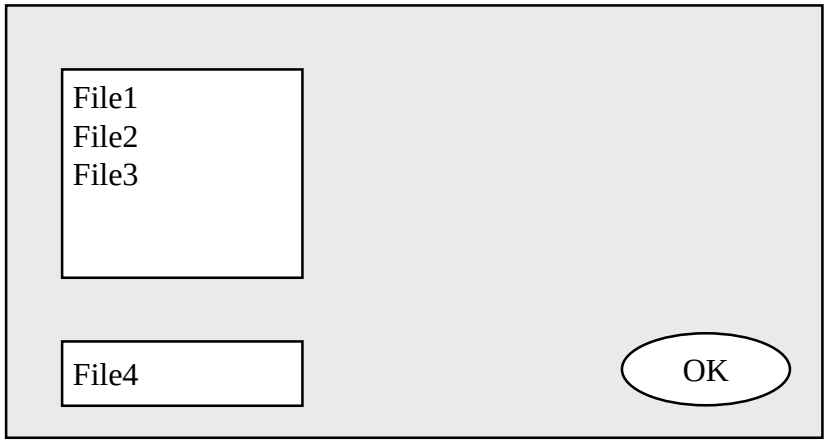
- Testing
  - Puzzles
  - Test cases
  - A testing / training exercise
  - Bug reports
  - The colloquium
- Debating
- Other opportunities to see behavior

The Myers Test Case

- The program reads three integer values from a card. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral.

- *Write a set of test cases that would adequately test this program.*

The Training Exercise



KANER.COM **Cem Kaner, Ph.D., J.D.** **kaner@kaner.com**
P.O. Box 1200 **408-244-7000 (Voice)**
Santa Clara, CA 95052 **www.kaner.com** **408-244-2181 (FAX)**

RECRUITING SOFTWARE TESTERS

INTERNATIONAL QUALITY WEEK
SAN JOSE, CA, JUNE, 1999

This paper is a work in progress. Eventually, it will become a chapter in the 3rd Edition of *Testing Computer Software* (under development by Cem Kaner, Bob Johnson, Brian Lawrence, Jack Falk, & Hung Quoc Nguyen. This is a multi-year project. Please don't ask me when it will be done. It certainly won't be done in 1999.) I'd appreciate comments and suggestions. (Send them to kaner@kaner.com.)

I don't claim to be an expert on recruiting, but I have been reasonably successful at recruiting testers (and other software workers) over the last 15 years, have made a lot of mistakes, and learned a few lessons. This paper collects some of those lessons.

Managers get work done through other people. The people that a manager chooses are the people who will achieve, or fail to achieve, the mission of the group and the tasks assigned by the manager. Recruiting staff is one of the hardest jobs for any manager. Hiring the wrong staff is, I believe, the worst mistake that a manager can make.

A good treatment of recruiting would consider the following issues:

- A behavioral approach to gathering information
- Legal issues
- Consensus-driven hiring
- A strategy for gathering information
- Characteristics of the person you're seeking
- Defining the position and characteristics needed for it
- Who approves the hiring
- How to find candidates
- Sifting through resumes
- Evaluating the candidate's public materials
- The phone screen
- The application form
- Preparing for the interview
- Asking questions during the interview
- Reviewing work samples
- Reviewing sample test documentation (of yours)
- Writing bug reports
- Using tests and puzzles
- Debates and controversial questions

Copyright © Cem Kaner, 1999.

- The problem of free consulting
- The post-interview meeting
- Checking references
- Investigation (such as drug testing, credit checking, etc.)
- Red flags
- Making and closing the offer

I'll include notes on each of these sections, but because this paper is, in effect, an early draft seeking comments, my depth of coverage of these issues will vary more than usual and I will cite fewer references than I would in a final paper.

A BEHAVIORAL APPROACH TO GATHERING INFORMATION

When you interview someone, you interact with them. In the process, they reveal things about themselves that go beyond the tidy sections of their well-formatted resume and the pat answers that they might have prepared for your standard questions. This can give you a better chance of learning how it would be to work with them. I recommend being very conscious about what candidates reveal to you in their behavior, and in designing interviews to elicit behavior samples.

Here are some examples that come almost automatically during the interviewing process:

- Look at the candidate's resume and cover letter (if there is one) for their structure and organization. Are they sloppy? Disorganized? Do they communicate well? Does the candidate appear to be hiding anything?
- Listen to the message on the candidate's answering machine? Clear? Weird? Rambling? Are there oddities in the message that suggest that the candidate didn't listen to/debug the recording?
- Look at the candidate's web site and at the signature section of her e-mail messages. This is information that she has chosen to make public. How interesting is it? What does it tell you about how she organizes information, how carefully she gathers information, and how well she maintains the documents that she posts? You can find a lot more about the candidate on the web, way more than you'd expect. See Lane (1997), for example. I suggest that you avoid looking for credit-related or sexual-activity-related information, and that if you find it, you make a point and a policy of not reading it.
- When the candidate comes for the interview, is he dressed appropriately? Does he show up on time? If he's late, did he call in advance? Did he bring additional information (such as work samples, letters of reference, publications, etc.) with him? If so, were they relevant and well organized?

Along with these, I ask probing questions about the candidate's experiences. My goal is to get the candidate to tell me about something she has already done, rather than make up a hypothetical answer. So I don't ask,

> "What would you do with a product that came to you without specifications?'

Instead I ask,

> "Have you ever worked on a product that came to you without specifications? Tell me about the challenges this raised and how you handled them. (And then, as a follow-up question,...) What do you think you did particularly well in that situation? (And then...) What did you learn that will help you handle this better in the future?"

Behavioral questioning is a standard approach. Rosse and Levin (1997, p. 173) provide more examples of what they call situational and behavior-descriptive questions. Risser (1993, pp. 150-152) provides more examples and useful discussion of the value of behavioral questioning.

I also give tests. For example, when I interview an experienced tester, I want to learn how well he can write a bug report. So I give him a bug and ask for a bug report.

I also do some role playing. For example, when I interview a test manager, I want to learn how effectively she can defend her point of view in a discussion with an authority figure. So I will arrange things so that an authority figure (maybe me, maybe someone else) will challenge her viewpoint on something that she considers important.

In general, to learn how well someone will do something, try to set up a situation that lets you see how well they do it. To learn how someone will respond to something, try to set up a simulation that elicits their response.

LEGAL ISSUES

I'm a lawyer, but I don't know the laws governing interviewing. I cannot give you reliable legal suggestions, but I can point out some (not all) of the issues.

- Several books that I've read tailor their approaches to minimize the chance that the company can be successfully sued for discriminatory hiring practices. These authors would advocate a more rigidly structured approach to interviewing than I do, such as making sure that every candidate is asked the same list of questions. I tailor the details of my approach to the human being that I am interviewing. Different humans, with different interests, aptitudes and backgrounds, get different questions. In a lawsuit alleging discriminatory hiring practices, an approach that is tailored to the individual is harder to defend.
- Independently of the legal issues, in a testing group, discrimination on the basis of race, gender, age, etc., is unwise. Diversity is a key goal for a testing group. Limiting that diversity means limiting your group's effectiveness.
- Privacy concerns implicate another body of law. For example, you probably can't check a candidate's credit rating without her knowledge. You probably can't require an employee to take a pre-hiring polygraph (the alleged but far from reliable "lie detector") test.
- You also have to be careful during the interview process to avoid making unintended promises. For example, suppose that you make a statement to a candidate that "No one here is fired unless they have a substance abuse problem." If you hire this person and she doesn't do any work, she can argue that you can't fire her unless she has a substance abuse problem (which she does not). Similarly, you have to be careful about statements that you make about the nature of the job itself. You can clear all of this up with an appropriately worded offer letter and employment contract (or you can make life worse with a poorly worded one).

Risser (1993) lays out some of the issues, but you should recognize that even when you are dealing with an excellent book, laws differ across states and change over time. For guidance on the legal issues, consult your company's HR department. That's one of the reasons that you have an HR department.

CONSENSUS-DRIVEN HIRING

I follow three simple rules when hiring:

- Anyone in the company who wants to be part of the interview process for a candidate is welcome.
- Of the people who have interviewed the candidate, anyone in the testing group and any senior player from any other group who will work with the candidate can veto the hiring of this person.
- The veto policy must be actively managed so that vetos will not be based on race, religion, family situation, gender, sexual orientation, age, national origin, etc.

There are a few reasons for doing this.

- I believe that it is a more serious mistake to hire badly than to pass up a good candidate. The consensus process will sometimes exclude a good candidate, but it is likely to expose problems. I haved been repeatedly humbled by hiring someone over the objection or strong reservations of one person, only to discover the hard way that the candidate is a jerk, a sexual harrasser, an incompetent, or just not a good fit. I am just not willing to make this mistake any more.

- Different interviewers bring out different answers and different characteristics from candidates. I want a broad process that encourages people with different viewpoints and interests to interview candidates. Encouragement requires letting people know that they're welcome and that their viewpoint will be carefully considered.
- Many of the people who I hire are "opportunity candidates" (see below). We are taking a risk hiring these people, or making some accommodations for them. I don't want to confront staff resentment later that this new person has some privileges that the complaining staff member lacks. For example, if someone can't work beyond a 40-hour week, I raise this during the post-interview evaluation meeting (see below). My question to the staff is direct. I explain the person's situation and ask whether they can live with the fact that this person will work shorter hours than they do. If they say no and I can't talk them around, then I can't hire this person. If they say that they'd resent a situation in which the candidate would get full pay but work fewer hours, I reassure them that I will calculate the person's salary with their limitations on hours of work factored in. And if they say yes, then three months later, when someone says "Gee, how come Jane doesn't work as hard as I have to?", I can say, "You agreed to this. Now we all have to live with it."
- The people who participate in the hiring process become part of the support network for the candidate after he joins the company. Many people new to a company go through a rough period during their first few months as they learn cultural or technological issues the hard way. I want to be in a position to say, *We made the decision to bring this person in. Now let's help him succeed.* This is particularly important when it is the new manager who is having a rough time. He needs a staff who will rally around him and support him.
- On her first day on the job, it is made clear to the candidate that every person who interviewed her voted to hire her. She was carefully considered by all these people, and they all welcome her. Especially in a controversial position (welcome to testing), it is good to remember that the person you're arguing with today was one of the people who liked and respected you enough to vote to hire you. This can go a long way toward lessening the mistrust that sometimes develops between testers and programmers.

Even in a small testing group, I don't require everyone to be part of the interview process. Some people just don't want to do this, or they don't want to do it for every candidate. That's fine. But they can't complain about the hiring decision later.

Junior staff are often hesitant to interview people because they don't know what to ask or because they're afraid to alienate the person they're interviewing. Some people like to interview the person who will become their supervisor. Others are uncomfortable. I encourage juniors to participate in interviews (partially because they have to get training in interviewing sometime, and there are only so many opportunties for this in a year), but I look for ways to make them comfortable. Here are some examples:

- The junior can silently watch an interview conducted by someone more senior. When I do this, as the interviewer, I introduce the observer to the candidate, explain that I am training this tester in interviewing by allowing him to watch some interviews, and ask the candidate's permission. If the candidate balks, I will ask the junior to leave. If the candidate is an individual contributor who is bringing technological skills, I might not hold this against him. Some very competent individual contributors are shy or awkward in groups. On the other hand, if the candidate who rejects the junior is interviewing for a management position, I will probe deeply into her attitude toward training and mentoring staff. She will probably not get the job.
- A small pack (maybe four of them) of juniors can take the candidate to lunch, maybe accompanied by one mid-level member of the staff who is a good observer. The mid-level staffer will observe but not speak beyond the minimum required for politeness. I am particularly likely to do this with management candidates. This group and I might even draft some questions before lunch, that they will ask during lunch. I encourage them to ask questions about the candidate's attitude toward training, education, and working conditions. If the candidate gets huffy ("You can't ask questions like that—I'm the manager, you're just the junior employee"), the candidate gets to go home early.

Note that both of these examples provide an opportunity for the candidate to exhibit behavior (the way they handle the situation, rather than what they say) that gives you insight that you probably can't get by asking questions. You'll see a lot of examples like this running through this paper, though I won't keep drawing your attention to them.

You'll also note that my interpretation and reaction to the behavior might be different from yours. I am a fan of Deming (1982). I believe that in employee relations, management should proceed from leadership and from credibility, not from fear and power. I believe that it is normal for people's behavior to vary over time, for them to screw up sometimes, and for them to be good staff despite that. I believe that many employee-made errors are induced by systematic weaknesses in their working situation (i.e. management-induced problems). I believe that most people want to do a great job. I look for management candidates who try to foster that greatness. You might look for people who have other attitudes. I can't say that you're wrong. This paper isn't to convince you either way on those issues. It is to help you gather data that will help you evaluate whether the candidate meets the vision you have. My interpretations are for illustration, not because they are necessarily the best ones for your situation.

The problem of discrimination is a more difficult one. First, let me stress that it is a real problem. Asians, blacks, hispanics, and women are still finding it hard to get work, to get treated with respect on the job, to get equal pay, and to get promoted. I have been personally reprimanded for hiring a black employee. I have had to deal with repeated and credible complaints from female staff members that they were being harassed by an executive and I have personally witnessed some of that unacceptable behavior. Asian immigrant colleagues of mine have faced offers of as little as half of the going rate paid to whites. All of this is illegal, all of it is unacceptable, and somehow you have to convince a racist not to use a veto for a discriminatory purpose or your consensus-driven process becomes a roadblock (or a steep cliff) instead of a tool. Certainly, I publicly remind the testing group that they cannot discriminate on the basis of certain types of characteristics. Most of my work, though, is private. People make racist or sexist (or etc.) comments in private. Some people will also consistently speak against candidates of a certain kind, gradually revealing their colors. I talk with them privately and will cut off their veto power (and their opportunity to interview) if necessary. (I've also suggested to some people that if they don't like these rules, maybe they should find a more hospitable company.)

One of the ways that I deal with the discrimination problem, if I think that there might be one, is to bring prospective employees to the company as short term contractors. This makes it hard to recruit people who already have a job somewhere else, but if the pay is right, unemployed testers will gladly accept a six-week contract. I'll hire contractors on my own, or in conjunction with just one or two interviewers. The full group process doesn't take place until the contractor has been working for a few weeks. Now we have data on actual performance. A candidate who is doing a good job is harder to reject for spurious reasons.

A STRATEGY FOR GATHERING INFORMATION

You're looking for a candidate who has certain characteristics (knowledge, skills, abilities, and other characteristics--see below). Make a list of the desirable characteristics.

You can gather information about the candidate on these characteristics from several sources:

- Resume
- Phone screen
- Work samples
- Publications
- Interview questions
- Tests
- Puzzles
- Group interview
- Behavior elicited by the interview

- References
- Investigative material (such as drug tests, credit reference checks, etc.)

Different sources are more effective for different characteristics. You can make a matrix to represent this. For example, yours might be structured like this:

Characteristics	Sources								
	Resume	Screen	Samples & Pubs	Intervie w Ques	Tests & Puzzles	Group	Interview Behavior	Refs	Invest.
Knowledge									
▪ **Testing courses**	x	x		x				x	
Skills									
▪ **bug reporting**			x		X			x	
Abilities									
▪ **team building**	x							x	
Other									
▪ **tolerance of ambiguity**				x			x	x	

For more on this type of matrix, see Rosse & Levin (1997), Chapter 6.

Along with determining what method(s) you'll use to gather the information, you have to decide who will gather it. For example, if you and several of your staff are interviewing the candidate, you will probably split up the characteristics. For example, one of you might check bug reporting while another focuses on tolerance of ambiguity.

CHARACTERISTICS OF THE PERSON YOU'RE SEEKING

As in so many aspects of software development, there is great value in thinking about your requirements first rather than wondering why you didn't meet them later.

And, as in so many other aspects of requirements analysis, you can take this too far and get paralyzed by analysis. The process of defining the job and the personal characteristics in advance has struck me as the greatest opportunity for analysis-paralysis in the hiring literature. Please read this through and think on it, but remember the 80/20 rule--80% of the benefit will come from the first 20% of the work that you can do in this area.

Strength in diversity

There is no single profile that fits the ideal software tester. Two testers who work on the same program will find different bugs. Diversity is essential. People with different skills, backgrounds, and sympathies will spot different classes of issues.

Additionally, the testing effort requires several strikingly different skill sets. For example, a few years ago, Jack Falk and I worked with a company that produced software to manage employee stock options. This application area is subject to complex government regulation (taxes, employee compensation, securities law, and sometimes lending law). If I had to build a small testing group for a company like that, I would shoot for a staffing mix like this:

- Senior tester or test manager with business operations or human resources experience.
- Senior tester or test manager with strong test planning skills. If this is the test manager, she needs excellent mentoring skills, because she won't have time to write the test documentation unless she is an individual contributor.
- Test automation hotshot, willing to serve as the group's tool builder.
- Talented exploratory / intuitive tester, someone who is really good at finding bugs by playing with the product.
- Attorney who is willing to wander through the various statutes and regulations looking for rules that the program must cover.

Some of these people have programming skill. Others have special knowledge of the application area. Others have special knowledge of the tools and techniques of black box testing.

I often see ads that require a specific profile of software testers: degree in software engineering or computer science, knowledge of the programming languages and tools in use at the hiring company, and some number of years of experience as a tester or as a programmer. These are nice credentials, but they would only solve some of the problems that must be addressed by this financial application company's testing group. Diversity is essential.

Several of the most effective testers that I've worked with have had no programming experience. On the other hand, most of the effective testing groups that I've worked with or consulted to have had at least one knowledgeable programmer on staff. Diversity is essential.

Repeat until hiring is completed <Diversity is essential.>

Accepting testing as a way-station

The testing group is the easiest entry point to the software development company, and therefore, many people come to testing on their way somewhere else. They want to become programmers, marketers, project managers, technical writers, whatever.

Over the years, a remarkable number of people have come to me with the glitter of somewhere else in their eyes and the willingness to stick it out in testing for an appropriate amount of time. If they offered the right mix of opportunities to me, I hired them even though I knew that they would go away eventually. (Then again, everyone goes away eventually.) A surprising number of those folks stayed in testing for a long time, or have come back since. Testing is a seductive field, once you come to know it.

But even the confirmed transients can add substantial value and substantial diversity to your group. Consider the following examples. These are real people, but the names have been changed:

- Joe was a marketer, who had reached as high as Director of Marketing and Sales. But he wasn't as effective as he could have been, partially because he didn't understand enough about software development. Between jobs, he decided to take a testing position (and so learn a great deal more about the realities of development) rather than moving right back into marketing. The job educated Joe a great deal. And Joe educated the other testers. He had plenty of ideas on how to make bug reports more persuasive, how to spot issues that would make the product harder to sell or support in the market and how to collect data to back up a bug report of the issue, and how to build credibility with groups outside of product development. He was a net gain to the group within weeks of being hired.
- Sam was a former VP of Marketing who heard about Joe and decided, between jobs, to learn more about product development too. He joined testing but only lasted in the group for a month or so before being transferred into a senior position in marketing, eventually becoming the company's VP. The testers didn't learn too much from Sam, but they had a great link into marketing.
- Sandy was a technical support supervisor with a software sales background. When her department was outsourced, she transferred into testing. Along with bringing a customer focus that was always welcome in this department, she brought strong scheduling, budgeting and status reporting skills.

She also had a personable style (salespeople are so valuable in testing groups) and was able to present bad news without creating an interpersonal edge. A short time (perhaps two months) after joining the testing group, she became the supervisor of the company's largest test team, working on its most delivery-date critical project. Her status tracking, early warnings based on status, and her visible but non-threatening had a big effect. The product shipped, with good quality, a day ahead of schedule. About a year after joining the testing group, Sandy moved into a senior tech support role elsewhere. For the year that she was in testing, she was invaluable.

- Tony was a largely self-trained, but talented, programmer who wanted to work with a well-known programming team. He joined the testing group to gain an entrée to that team and he worked in the test group for a year or two before being promoted to a senior individual contributor level in the programming group. Tony was, in some ways, difficult to manage as a tester. He hung around with the programmers too much and with the testers too little. He collected and used a lot of information about the program but kept too much of it to himself. In retrospect, Tony's manager realized that Tony had been mismanaged. Tony would have been a valuable toolsmith, and would have probably been quite happy in that role. Instead, he got the assignments that many other testers enjoyed more than he did: lead the testing of a technically interesting project, map out the test plan and test documentation and find/report lots of bugs.
- Cindy was a librarian who probably wanted to become a technical writer but got into a testing group first. As a tester, she was able to spot communication issues and training-related issues and explain them with clarity. She also excelled at writing testing documentation and she was effective as a supervisor and trainer of new testers. Over subsequent years (and subsequent companies), she moved back and forth between writing and testing assignments.
- Sean had been a project and a product manager, but he burned out on software and took a several year break from software development to manage construction projects. Eventually he was ready to come back to software development but software companies were hesitant to hire him. He didn't have experience in the currently fashionable language and so his skills were seen as outdated. So he joined a test group to update his skills. Years later, he was still in testing, doing a great job, managing the test group of one of the industry's largest publishers.

There are lots of other stories like this--people who came to a testing organization with strong skill sets and a desire for a relatively short term (6 to 18 months) stints in testing, who very quickly started to contribute tremendously to the groups that they joined or who would have contributed tremendously if they had been properly managed.

The mismanagement problem is a real one. Or, at least, it has been for me. On balance, I've been very successful with staff who come to me with skills, on their way somewhere else. But when I've tried to jam them into standard molds, without recognizing and taking advantage of their special strengths, they've been disappointed and disappointing. When they don't fit a near-term or intermediate-term need, I am hesitant to hire them.

Opportunity hires

I want a great staff. This isn't easy. It's harder when the field is booming, as it is today, because there are fewer people than jobs. It's even harder when I work with a company that pays less than the market rate. (I founded the testing group at one company whose products were very exciting, but whose top salary matched the entry level rates of other software companies. I was successful in recruiting a top quality staff, but it took creativity.)

When I can't offer the top money in the field or the most exciting product line or career path in the field, I have to find some other incentives to attract solid talent. And so I look for people who have special needs that I can fill. These people have a strong skill set but they also carry some baggage that make other test managers reluctant to hire them. That leaves proportionally more of these people for me. If I can connect with this market (these markets) of people, and if I have a good job to offer, then I can have my pick of them. I call these people "opportunity hires." For them, the job that I offer is a special opportunity. For me, the skills they come with provide a special opportunity.

The people who I described in the last section, people experienced in some other area who want to spend six months or a year in testing and then move on, are an example of opportunity hires. There are a lot of other people who are relatively unpalatable to traditional hiring managers, who have a hard time finding a position even though they might be brilliant. Here are some examples:

- *Retirees, especially retirees who want to work only 20 or 30 hours per week.* People step down from their fast-paced, full-time jobs, and discover that they need money or that they've become bored. The ones that I'm thinking of are not looking for fast-paced jobs that might require lots of overtime. They're looking for an intellectually stimulating position that offers steady hours and some flexibility to spend time away when they have to deal with personal matters.
- *Pregnant women or single mothers with young children.* It might be illegal to discriminate against these people, but they're discriminated against anyway. I don't ask about candidates' family situations--if you do, the odds are that some day, you will be sued for it. But when someone raises the issue with me because they anticipate needing a leave of absence in the near future, or because they need unusually flexible hours, I'll talk frankly with them about it. The fact is that at most Silicon Valley style software companies, people work long hours. And near the product release date, people work longer hours. Especially testers. I don't feel obliged to hire anyone who can't reliably put in those hours. But when someone with an extraordinary skill set and a positive attitude tells me that she'll gladly take a step down in responsibility and salary in return for strong control over her hours, I'd like to find a way to make this work.
- *Other family commitments:* Many middle-aged workers are facing a new responsibility. Their parents need time-consuming attention. As another example, non-custodial parents will often bargain for guaranteed vacation times. For example, Mom has custody of the kids but Dad has visiting rights and also has the right to a few weeks with them per year, for vacations. Dad has to book these weeks in advance with Mom, and scheduling has to be take into account her convenience. From you, Dad needs some assurance that he can have X weeks per year away from work, that he can book this time in advance, and that you won't demand that he change his plans at the last minute. As with the pregnancy issue, you can't ask about family commitments beyond asking whether there is anything in the candidate's personal situation that would interfere with their regular attendance at work at the normal work pace that you expect of your staff. But if a candidate raises family commitments as a bargaining issue, I think that it is fair game to consider what accommodations can and cannot be made and whether this candidate's talents justify the need for the requested accommodations.
- *Returning to the workforce.* The prototypic case is the mother who comes back to software after investing several years in the care of her children. Sean (the project manager described above) is another example--he stayed commercially employed, but not in software. Similarly for the technical person who spends years trying out sales and marketing and realizes that she's happier in engineering. These people come to your with aptitude and seasoning but without current knowledge of tools, languages, and best practices.
- *Immigrants who speak English with a thick accent.* Testers must be able to fluently understand the language spoken by the product development team. I don't want to hire people who can't understand spoken or written English. But many immigrants understand more than they can say. I am reluctant to consider a candidate whose written grammar and spelling (especially if the mistakes are on the resume) are poor. And I am reluctant to consider a candidate who has no employment references within reasonable telephone distance (North America). But so is every other hiring manager. If a candidate looks as though he might be particularly talented or particularly smart, I'll suspend judgment and try to find out more about him.
- *Career switch.* Some of my most successful hires have been of people who were leaving a mid-level to senior position in a different field in order to come to software. These aren't people who come to you and say, "I want to work in testing for a year before I move on." They're people who say, "I want to switch to software. Are you where I start?" They might well stay in testing. The key for these folks is often not the pay scale. It is the opportunity for training in a new profession. If you can/will offer to spend additional personal time (of yours, as the hiring/training manager) coaching this

person, you can make a competitive offer even if the candidate will have to take a steep pay cut to come work for you.

In sum, opportunity hires are people who come to you with special needs or special circumstances. Their circumstances are odd enough that recruiters will often not help them, perhaps never presenting their resumes to employers or only presenting their resumes for unacceptably junior or low-paying positions. You can be in the position (I have been in the position) of being the only employer that the candidate has talked with, through a significant job search, who is willing to show respect for their talent and experience while making allowances for their special circumstances. That is, you might be able to be the only person who makes a serious job offer to a candidate who would normally appear to be substantially overqualified for the position you have open.

Opportunity as risk

Not every "opportunity" candidate is a good bet. Here are some examples of people who initially look like opportunities but on closer inspection turn out to be potential disasters. Again, the names are changed:

- Joe was an alcoholic. He wanted to shift from technical (non-software) sales management into software. He looked great on paper, increasingly responsible jobs, good performance reviews, decent technical education, MBA, analytical and smart. But his real reason for looking for a new career is that he is screwing up so badly this year that he's just about to lose his current job and he's afraid that no one else in his current field will hire him.
- Sandy wants a programming job today. She'll accept a position in the testing group but starting on her first day on the job, she will do anything she can to wangle a transfer into the programming group.
- Jerry used to have a senior position. He will accept the job with you, but he will still expect to be treated with the kind of deference that he got used to in his last position(s). He will become unhappy when people (such as programmers) don't take his advice and follow his directions and will dedicate his time on the job to political intrigue rather than to finding and reporting bugs.
- Jane did well in her last career (perhaps even a software career, as a programmer or software marketer) but has no aptitude for testing. In your group, she will be bright, articulate, cooperative, hardworking and (unless you manage her very effectively) unproductive.
- Cecil will need a lot of time away from the job and the attitudes within your group are such that this will be resented by the rest of your staff. You will face a revolt if you give him privileges that the rest of your staff don't get.

Minimizing opportunity hiring risk: Short-term contracts

When someone's moving into a new type of position or moving back into the workforce after a long time out, there is just no way to be sure that they and the job will be good for each other. You will discover that many wannabe-opportunities are no good and won't hire them, but for the person you seriously consider, you still face a substantial risk of being wrong.

Whenever possible, I try to hire people for a six-week contract. If the candidate worked well and has the skill set that I want, I can then make a long term employment offer. But if she didn't work out, I don't have to renew the contract, make a new contract, or extend an employment offer. (I make it very, very clear up front that the contract is complete at the end of six weeks, with no promises made as to consideration for future work.)

If the candidate doesn't work out, but is trying hard and gets along reasonably well with the other staff, I'll typically keep her for the full six weeks. This way, she gets to finish out the contract. This helps her in her next interview, when a prospective employer asks, "Why did you leave BugWare, Inc.? You were only there for six weeks." She can truthfully answer "It was only a six week contract. I completed it."

Handled this way, the six week contract limits my risk as an employer and the candidate's risk. Either of us can walk away without complication.

Minimizing opportunity hiring risk: Group buy-in

I discussed this above, in the section on consensus-driven hiring. It is important for your group to accept special arrangements. My position is that everybody has some special arrangements and everybody needs some special accommodations. The issue is to make this clear enough that people are tolerant of each other's situations.

DEFINING THE POSITION AND CHARACTERISTICS NEEDED FOR IT

Any book on recruiting will talk about this. Rothman's (1998) treatment is well tailored to the programming environment and quite thoughtful. She also provides sample descriptions of testing positions. (ST Labs website sometimes posts some excellent job description information too. www.stlabs.com.) Rosse & Levin (1997) provide a detailed generic treatment. Risser (1993) provides a good overview. Kaner, Falk, & Nguyen (1993) discuss some of the attributes of good testing candidates.

Defining tasks you need accomplished today

You're recruiting because there is some work that you want done. You're looking for someone to do it. It makes sense to list the specific tasks and the knowledge, skills or abilities required to do those tasks. You will then evaluate the candidate against the list. If he can't do what you need done today, you need someone else.

Defining tasks you need accomplished later

If you take a primarily task-oriented approach to defining the job, then there will be several tasks that aren't needed for today's product that will be important over the next year. What other things should the tester be able to do, over the next year or two, that aren't on the immediate needs list?

If this person isn't capable of doing some of these tasks now, can you train him? Can you spare the time? Do you have the skill? Does he have the aptitude?

Defining the strategic role of this person

Along with the specific tasks that you are trying to get done now, you should have a sense of what this person will do for you over the longer term. For example, are you trying to find an automation specialist? A test planner? Someone who will become a supervisor or manager after a few months of training? A brilliant opportunity (candidate with strong potential to become a senior manager) might be no opportunity at all if your goal is an automation hotshot.

List of individual characteristics

It can be very useful to list the individual characteristics that you will consider important for the job. You might want all of these for all jobs, but some are more important for some jobs than others. For example, diplomacy is probably a skill more needed by managers than by senior programmers.

- Lists of characteristics are often broken into lists of KSAO (Knowledge, Skills, Abilities, Other). These are defined as follows:
- "Knowledge" is the body of information that the candidate will need in order to perform effectively. For example, a person might know or have training in test case design, maybe even a course in creating test matrices.
- "Skills" involve proficiency at a specific task. For example, a person might be really good at creating test matrices.
- "Abilities" involve potential to do a job. For example, a person might be a very bright, systematic, analytical thinker, who you would expect to be able to quickly become very good at creating test matrices.

- "Other" includes other characteristics that are important to the job, that aren't abilities. For example, a person might have personal integrity.

Here is a list of some of the skills, abilities, or other characteristics that are sometimes mentioned as relevant to success as a tester. Many of these are redundant. You might define them differently. This is a starting point, not the ultimate list for you. Don't look for all of these in one person. Pick a subset, define them the way that works for you, and add your own.

- Alertness
- Attentive to detail
- Analytical problem solver
- Architect (talented at designing systems, in breaking the system into achievable tasks, subtasks, and data)
- Arrogance (usually, less is better)
- Artistic (understands visual or audio presentation issues, can knowledgeably critique the esthetics of a design or product)
- Assertive
- Auditor (compare situations against standards)
- Author (published, impressive, credible)
- Commitment as a person (keep promises, stick around)
- Commitment to a task (do what it takes)
- Commitment to quality
- Coping with difficult circumstances
- Courageous
- Creative
- Credible (people believe what she says)
- Curious (inquisitive, likes to explore and find things out)
- Customer focused
- Decision making and problem solving (judgment, realistic understanding of issues)
- Decisiveness
- Diplomatic (able to convey bad news, criticism, or unreasonable requests in ways that don't offend and do encourage a desired response)
- Effectiveness with junior testers
- Effectiveness with senior testers
- Effectiveness with test managers
- Effectiveness with programmers
- Effectiveness with non-testing managers
- Empathetic (able to appreciate other people's situations and viewpoints)
- Empirical frame of reference (learn by running experiments)
- Empowering (promotes excellence and risk taking in other individuals)
- Energizing
- Fast abstraction skills

- Financially aware and sophisticated (for example, able to make economic arguments)
- Finds bugs (some people have excellent bug-finding intuition)
- Flexible (willing to shift takes or to take on new things)
- Goal setting
- Glue (promotes group cohesiveness)
- Humility
- Integrity (likely to keep commitments, unlikely to engage in dishonest conduct)
- Interpersonally perceptive (reads verbal and nonverbal behavior)
- Interviewer (good at getting information by asking questions)
- Investigative reader (good at pulling information from incomplete specs, from the Net, etc.)
- Leadership
- Long term thinker
- Meeting manager (skillfully facilitate or record (e.g. on flipcharts) other people's meetings)
- Mentor
- Multi-tasking (juggles multiple tasks well and can handle the pressure)
- Organizer and planner
- Persuasive
- Politically perceptive (reads the system)
- Policy and procedure developer
- Pragmatic
- Programmer (able to write good code, command technical respect of other programmers)
- Protective (stands behind, defends his staff, even when they're wrong)
- Punctual
- Scholarly (collects information, and is able to back up or evaluate arguments using data or credible opinions/statements of others)
- Sense of humor
- Spoken communication
- Strength of character (does the right thing even when it's personally costly or inconvenient)
- Subject matter expert, in the area being automated by the software
- Substance abuse (undesirable)
- Team builder
- Tolerant of ambiguity
- Tolerant of other approaches to managing projects, doing tasks and solving problems
- UI design (skilled at designing the appearance of features, etc., and a persuasive knowledgeable critic of the designs of others)
- Versatile (many abilities)
- Warm (interpersonally, makes the human environment more pleasant)
- Written communication

- Zealot (believes in The One True Way and insists that everyone else believe too. Probably not desirable in large quantities.)

I'll come back to the question of how to evaluate a person on these dimensions later.

HOW TO FIND CANDIDATES

You can advertise a particular position or you can advertise the general desirability of working for your company an ongoing basis, whether you have positions open or not. Some people prefer the ongoing approach because it generates a steady stream of resumes from people who pay attention to you or your company.

Ongoing advertising

The goal of ongoing advertising is to present an image of yourself, your group, and your company that makes people want to work with you. Do this by talking or writing about your technical views or your company's technology or management style. Most of these activities (publishing, teaching) are done for other reasons, but they have the effect of attracting people to your company. It should be obvious that it's important to avoid creating a false impression. Even if you succeed in hiring someone this way, it won't be a lot of fun working with them. Some of the usual types on ongoing advertising:

- Advertisements (paid spots) in newspapers, radio, TV, that promote your company generally rather than a specific position
- Books that you or your staff write
- Conference presentations
- Courses that you offer to the public or to the profession
- Newspaper or magazine articles about you or your company
- Newspaper, magazine or technical articles written by you or your staff about technical or management issues

Promoting a position

You can be much more effective at getting a stack of interesting resumes if you use multiple methods for publicizing your company and your currently open position than if you rely on one approach or medium.

You can advertise positions through:

- Announcements at conferences or meetings
- Announcements or ads in professional journals or newsletters
- Announcements (job listings) at employment services (your state unemployment agency, for example). (Yes, this can be useful. When BigCo. lays off 1000 testers, many of them will file an unemployment insurance claim. A notice with that government agency might capture their attention at the start of their job search.)
- Current employees (for example, pay them a bounty for bringing in good people)
- Newspaper advertisements
- Postings on Net-based job boards and job-related newsgroups and e-mail lists
- Radio and TV advertisements
- Recruiters
- Spamming (if you want to find people who are willing to work for a spammer. Blecch.)
- Word of mouth announcements spread as rumors through the community of people you'd like to recruit from

What do you say in an advertisement?

Beyond my suggestions, Whitaker (1994) provides several other useful suggestions for advertisements.

When I advertise:

- I want the right people to send me their resumes.
- I want the wrong people to send their resumes to someone else.
- I don't want people to misunderstand the job or the company.
- I don't want government regulators to tell me (or my company) that I am acting unlawfully or unethically.

Positions are often advertised ineffectively. An advertisement that lays out a generic job description is less likely to capture anyone's imagination, and is very likely to discourage opportunity candidates (people with unusual backgrounds) from even applying.

The following examples were all invented for this article. They might be reminiscent of ads that you have seen, but they were not based on any particular advertisement.

Here is an example of a poor advertisement:

> SOFTWARE TESTER. Hard-working, diplomatic, detail oriented, effective communicator. Great at finding and reporting bugs. Strong test planning skills. Automation experience desirable. UNIX and PC platforms. Must have B.S. in Computer Science or equivalent and five years of related experience.

Why would you want to work for this company? The ad doesn't say. What is special about this company? The ad doesn't say. What is interesting about this position? The ad doesn't say. Who will apply? My bet is that this ad will attract lateral hires (people who want to do the same job as they have today, but for more money, with a different boss). It will attract people who have standard backgrounds and it will attract people who don't worry about background (people who have consciously decided to apply for positions for which they are not qualified, at least on paper).

The next ad is another generic special that any HR Department can put together for you quickly:

> SOFTWARE TESTER. Come join our state of the art company, and define the leading edge in the testing of consumer software. Hard-working, diplomatic, detail oriented, effective communicator. Great at finding and reporting bugs. Strong test planning skills. Automation experience desirable. UNIX and PC platforms. Must have B.S. in Computer Science or equivalent and five years of related experience.
>
> We pay highly competitive salaries and have a superb benefits package. Work in the heart of Silicon Valley. Etc.

This ad defines some of the benefits of the job, but in HR-speak. It promotes the company in general terms and the financial and lifestyle benefits of working for the company. It doesn't promote the career growth or the content of the job.

Here's an ad that could work:

> SOFTWARE TESTER: Bank applications, COBOL, SQL, Visual BASIC, Client/Server, etc. Financial application sophistication required. We are looking for excellent staff and will pay appropriately. Depending on experience and demonstrable skill, this position will pay $75,000 to $120,000 per year.

This is a generic financial position but is specific about its core selling feature. The employer will pay big bucks. (If $75-120K is no longer big bucks for this type of position, reread the ad with a bigger pair of

numbers.) People who want the big bucks will apply for this job, whereas they might well not apply for a job that offers "highly competitive salaries."

Here's another variation. In this case, the staff are the benefit, not the money:

> SOFTWARE TESTER: Bank applications, COBOL, SQL, Visual BASIC, Client/Server, etc. Every member of our group has sophistication in financial applications, and strong technical skills outside of testing (such as programming, data design, etc.) along with solid testing experience. Our department fosters a mutually supportive, growth environment. We work in teams and we make time to educate ourselves and each other.

Now consider this one.

> SOFTWARE TESTER. We test data communications software for the home market. Help us develop software that must be reliable, quick, and easy and fun to use.
>
> To effectively test our products, you will probably have to be able to read and write code. For this position, we are especially interested in people who know about set-top boxes, cable modems, TCP/IP, browser internals, or other current data communications implementation and design issues. We are also willing to meet testers (whether you can program or not) who are skilled in finding OS-level or device-level bugs, or who are skilled in performance measurement, client/server methods, or component level testing. Excellent communication skills are a must.
>
> Pluses include project management skills, experience creating and managing test plans to coordinate the work of several testers, and test automation experience.
>
> The successful candidate will probably have a degree in computer science and five years of software development or testing experience. In your cover letter, please indicate the types of evidence that you can provide that you can excel in a fast paced company as a technology-sophisticated software tester.

The ad conveys interest in technology and in the satisfaction of customers. Evidently, these are values of the group and (presumably) of the company. The ad might attract lateral hires (from other testing groups), or programmers who are will to switch into a testing role in order to learn more about data communications. The ad will also attract a few senior testers who want to brush up on technology. Their letters (the best of them) will admit to a lack of data communications experience, and to rusty programming skills, but will stress their project management and test planning skills. They will express great interest in this opportunity to learn about this new (to them) field, and will stress their willingness to work hard to achieve the learning. In general, this ad tells people what they'll have to be (or become) good at, without demanding any particular credential (such as a degree). It will also attract the same clueless crowd of people who will send their resume to any ad that says "tester" or "programmer" and so you'll still have to throw those applications away.

One last example.

> SOFTWARE TESTER. Be one of the first employees in a software start-up. We offer challenging technology, long hours, and a stock option plan that will let you share in our success. It's too early to publicize the nature of our business, but we will carefully consider every resume that shows at least five years' experience in any aspect of software development, including at least two years' experience in software testing.

Each of these ads stresses what is special about the employer. Every well-marketed product or service carries a "unique selling proposition"--something special that is hard or impossible to find anywhere else. A reason to buy it. I apply that principle to the job advertisement as well. The company might be committed to customer satisfaction, hard driving pursuit of new technology, process management (ISO 9000-3 or CMM

done by a company that believes in it and wants to benefit from it), job stability, family values, whatever. Telling candidates about the corporate mission and values, and the group mission and values helps people decide whether they are excited by you or not. Rosse & Levin (1997, p. 61) talk about this “unique selling proposition” in terms of being an “employer of choice,” a place that people will seek out. Rothman (1998) writes of positive and negative company factors that should be considered for advertising.

The ad will often also say something about what must be special about the candidate. I tend to be flexible on formal "requirements" like a degree in computing or accounting. I take some care to position these requirements as desirables, rather than as rigid requirements, unless I believe that they are absolute minimum requirements for the company at hand. In some other cases, I will be very specific. For example, I worked for an entertainment software company at a time that certain DoD contractors and some other large IT employers were laying off technical staff. Our style of testing was fast-paced, exploratory, without the benefit of a specification. I got tired of talking to people who insisted that their 15 years of experience qualified them for consumer software testing because testing is testing is testing and who then insisted that our development methods were all wrong and we would have to change them. Eventually, my ads read:

> Verifiable experience in development, support, or testing of software that was to be sold to or used by mass-market customers.

I still got the inappropriate resumes (fewer) and the follow-up phone calls, but it was faster and easier for me to say to a candidate that he was unqualified, as demonstrated by the lack of a qualification listed in the advertisement. The specific language in the ad satisfied people that they weren't being singled out when I turned them away. I might be a fool, but I was being consistent, and so they were willing to leave me alone.

I didn't adopt this wording because I don't like DoD or IT training. I adopted it because, at that time, in that market, I was getting flooded with resumes and follow-up calls from people who were not going to get hired for this class of job. At that time, in that company, other more appropriate, enthusiastic candidates were readily available in the pool. The non-consumer candidates were wasting their time and mine.

Should you put your name in the advertisement?

Another issue in the content of the ad involves whether you should list your name as a contact point. I always do list my name. This wastes some of my time, because recruiters and more recruiters and more recruiters call me, and because some candidates call me. But I eliminate most of this issue while keeping the ad personal by saying:

> Send resumes to Cem Kaner, Manager of Software Testing, <<company address, company e-mail address>>. Please send inquiries and resumes by letter or by e-mail. I cannot handle inquiries and applications by telephone.
>
> Principals only please. Materials received from intermediaries, such as recruiters, will not be reviewed.

An advertisement with a human face will attract people who like to work for/with humans. Most ads are impersonal, so a personal touch stands out.

By the way, after you've been in the business for a while, people come to recognize your name. I never associate my name with an employer that I wouldn't commend to a friend. I never associate my name with a description of a job that I wouldn't give to an appropriately qualified friend. I never associate my name with an interviewing process that is designed in a way that it will demean or intimidate the candidates. Over the years, goodwill develops. People will apply for a position just because it's your name on the ad.

If I am interviewing on behalf of an employer who has staff difficulties, I’m honest about that with candidates, and I encourage other interviewers to be honest about it. The goal is to give candidates a “realistic job preview” (Rosse & Levin, 1997, p. 62). That doesn’t mean that we try to advertise the company’s faults or to discourage people from working at the company. And I don’t necessarily put the company’s weaknesses forward in the first phone screen. But it’s important to make sure, before the candidate accepts a position, that he knows what he’s signing up for.

SIFTING THROUGH RESUMES

Most resumes will be rejected. Sort *quickly* through the resume pile to find the people who are worth calling. It's important to call good candidates *quickly* because they will probably be lost (someone else will hire them) if you delay.

I sort resumes into four piles: rejects, reject but keep in an active file, priority 2 and priority 1.

Rejects

Rejects get a form letter right away that says thanks, but no thanks. A common form letter says,

> "Your qualifications are impressive and we appreciate your effort in contacting us, but there is not a match between our requirements and your skills at this time."

I send a slightly-flattering letter like this to every reject, even the hopelessly underqualified and even the ones who have obviously lied on their resume (like the cretin who claimed to have authored a manual that I wrote). I am unfailingly polite. My goal is to spend a minimum of time and emotional energy on the rejects. My rejection is friendly and respectful for a few reasons. First, being told that they're not going to get the job is bad enough. I'm not out to make anyone's life miserable, just to close this relationship. Second, I don't want to make anyone made because that makes them more likely to reappear, threatening a lawsuit or demanding satisfaction in some other way, or just pestering some executive who will make my life miserable for having motivated a person to become a pest.

Here are examples of resumes that I will reject:

- *Inappropriate behavior*, such as foul language, inappropriate gender references, or jokes in poor taste.
- *False statements or exaggerations.* Many resumes, perhaps 25%, contain lies or significant exaggerations. I have no tolerance for these. By the way, I make these judgments quickly, and sometimes I might be wrong. I don't know of any requirement that I make a thorough investigation before privately concluding that someone made a false statement. For example, if a candidate claims to be an expert in Java and then writes about the Net in ways that misuse common phrases and appear to demonstrate fundamental ignorance of the platform, I won't spend investigating further. I'll simply reject the resume (but without sharing my private conclusion that this person is a liar.) On the other hand, I am always conscious of my duty to not discriminate against members of various protected groups and so my dismissal of a resume cannot be based on a stereotypic judgment like, "No woman could have led the testing of WordStar. She must be lying."
- *Clearly insufficient background.* If the ad calls for experience in testing or programming, and the resume doesn't list any, I usually reject it immediately. Not only is the candidate unqualified. He is evidently not reading or responding to the content of the ad. On the other hand, if the candidate writes a cover letter that admits that her experience is too thin, but says that she really wants the job, then I will read the resume more carefully. This tester is paying attention to what was said in the ad and responding to it head on. She is negotiating. I value those behaviors in a tester and so I will tend toward keeping her under consideration for some other position, if not this one.
- *Spelling and obvious grammar errors.* Someone who doesn't take the time to check his own work (or get help from someone else) is unlikely to cut it as a tester. I make three exceptions to this generalization. First, if the candidate is a very recent immigrant, I might read the resume a bit more carefully before rejecting it. Second, if this is a very long resume, I tolerate an error or two as a normal bug rate. And third, if the candidate has clear, verifiably successful experience, then my prediction that he'll never be a good tester has been refuted. But in any of these cases, I'll question him closely and check his references carefully.
- *Insufficient information.* Some resumes convey so little information that I have no idea whether this tester is suitable for the position. If nothing in the resume tells me to be interested in the candidate, I reject it.

Reject but keep in an active file.

Some candidates are not appropriate for the current position but have distinguished themselves in some way that makes me want to keep their resume in an active file. For a different position, I might hire this person. The rejection letter for this person might say something like:

> "Thank you for your application. We have decided to consider other candidates for the position for which you applied, but we are impressed with your qualifications and will keep your application in an active file. We will contact you if a more suitable position becomes available in the near future."

Priority 2

This is a holding pile. I'm not enthusiastic about these candidates, but it's possible that they would be suitable for the job. I won't reject them yet but I won't call them for interviews until I've explored the higher priority candidates.

Examples of candidates in this group:

- *Underqualified but within training distance.* This candidate doesn't meet the position's minimum knowledge / skill requirements, but it might be possible to train her into the role.
- *Recommended by someone significant.* This candidate was recommended by a staff member or a trusted colleague. Before I reject the resume (assuming that it is within the realm of possibility), I will look at it with some care.
- *Insufficient information*. Some resume styles (the very brief functional resume, for example) are uninformative. They hint at information about the candidate but provide little useful detail. I will usually reject such a resume if the candidate is applying for a management-level position (he should have evaluated enough resumes as a hiring manager to know how worthless this one is). I won't automatically reject such a resume from candidates for less senior positions or for technical positions, as long as the resume somehow makes it seem plausible that this candidate might be qualified for the position.
- *Unusual situations.* Most of the "opportunity hires" start out in my second priority group.

First priority

I call first priority candidates (for a phone screen) as soon as possible.

EVALUATING THE CANDIDATE'S PUBLIC MATERIALS

Some candidates are published authors (in magazines, conferences, books). Some have web sites. Some are active on news groups or mailing lists (these are often archived). These writings reveal a lot about the candidate. For example, you are likely to learn:

- about the candidate's knowledge and writing style
- what the candidate claims in public about the nature of his job and his experience (Is it consistent with the resume?)
- what issues seem to draw the attention of the candidate--what he thinks is important
- whether he engages in flame wars. What it takes in a debate to irritate him
- how much time he spends posting to news and (to the extent that you find out) to mailing lists and whether he is posting from a company e-mail address during normal business hours.

I prefer to read a candidate's material before a phone screen, but I often don't have time before the screen. I make the time to read the material before the face-to-face interview.

THE PHONE SCREEN

The usual point of a phone screen is to filter out candidates. The face-to-face interview is expensive and time-consuming for everyone. If this person is obviously a mistake, the sooner you close the call, the less time and money that you waste on them.

I also use the phone screen to learn more about the candidate, in order to better prepare for the face-to-face interview.

I allow ninety minutes per phone screen. The calls actually last between two and ninety minutes.

I have a list of issues/questions that I select from, and I often go through them in the order listed. But I might go directly to an issue if I have specific concerns or interests in that candidate.

For example, if I'm calling someone who submitted a low-information content functional resume (they list types of tasks that they've done, but don't tell me where or when or provide much beyond a buzzwordy list and then list the employers and dates but don't say what they did where), I might start by collecting a chronology. Where did you work? When? What Was your title? Who did you report to? What were your major accomplishments? Why did you leave? Or, I might start by asking about the claim or two that captured my attention. For example, I might say, "In your resume, you said that you have experience with QA Partner. Can you tell me when you use that tool and what you used it for?" If the answer is weak, and this was the key skill that led me to make the call, then I'll move into shutting down the call (see below).

Here's a list of questions that I put together over the course of recruiting a few test managers. I normally bring this list up on-screen, creating a candidate-specific file. I often type while I talk with the candidate. If I don't type, I take detailed written notes.

Several of the issues are well answered in the resume, and I won't spend much (or any) time on them in the phone call.

The questions here are questions that I am asking myself. They tell me what I'm trying to find out. I word the questions to the candidate differently for each candidate, following the flow of the discussion we've had so far and taking into account what I already know.

You might be required to ask all candidates the same questions in the same order. (Some companies that have been sued for discriminatory hiring practices, or that don't want to be, will adopt the rule that everyone during a screening is asked the same question.) If so, I recommend that you start the phone screen (after introductions) by telling the candidate that your practice is to ask everyone the same questions in the same order, apologizing in advance for any questions that overlap or that seem out of context.

My list of questions for a test manager

The resume presentation

- Typos
- Consistency of presentation

If there are weaknesses in the presentation of the resume, I'll ask about them. For example, if there are typos or spelling mistakes, I might note them and ask why. The typo or spelling mistake might not have disqualified the candidate, but the candidate's response to this question might give me a reason to close the interview quickly. A response that no one cares about these minor details, or a response that seems unusually defensive (the candidate asks, "How dare you ask about something like that?") will typically convince me to close the interview quickly.

Educational qualifications

- College or university studies
- Continuing education

- Books and publications read / written
- Conferences attended
- Professional societies--member / activity
- Standards committees--membership / what did they do
- Awards received by the candidate or his staff

How has this person learned about testing and about software development in general? There are very few university courses on software testing. What has the candidate read? Even at the test manager level, the majority of candidates that I screen have never read a book on testing, never taking a university-level or university extension class on testing, and never attended a conference on testing. Their training has been completely in-house. I ask every testing candidate about education. The more senior position that the candidate is applying for, the stronger my feelings are that the candidate should have been actively broadening and deepening her knowledge of the field.

This group of questions also gives the candidate a chance to tell me about his commitment to the profession. I don't expect the candidate to be active in the IEEE, ASQ, ACM, etc.. I don't expect the candidate to work on the development of professional standards. But if the candidate does this, I want to know what he does, what he's been learning from it, and I want a sense of whether this candidate offers too much of a good thing. Some candidates expect to do this work on company time, for many hours per week. That might or might not be acceptable for the open position.

Employment history. For each employer:

- The basic data: company / dates / title / role / supervisor
- What kind of products they worked on?
- What interesting technology they used or developed (that they can talk about)?
- What worked well?
- What didn't work well?
- What they did that was special?
- What approach to continuous improvement?
- Why they left?

The question on continuous improvement is often informative. Sometimes I ask this up front, before asking about specific companies. I want to know how they monitor and improve their own work and the work of their staff (if they are managers).

Approach to testing

- What is SQA?
- What is the value of the testing group? How do you justify your work and your budget?
- What is the proper role of the test group?
- What is the role of the test group vis-à-vis documentation, tech support, etc.?
- How much interaction with end users should testers have, and why? How should you learn about problems discovered in the field, and what should you learn from them?
- What role of glass box / black box / gray tools?
- Automation?
- Development model? What should the programmers use? What should the test group use?
- How did you get programmers to build testability support into the code?

- Role of bug tracking system (track bugs / personnel / design bugs)?

I ask these questions of supervisory candidates or of senior individual contributors. I am not looking for The One Right Answer about How Testing Should Be Done. I primarily want to know if this candidate has thought about these issues in any depth. I might also be trying to learn whether her views are roughly compatible with the company's. For example, throughout this series of questions you see a bias toward testing, with little regard to process standards. I listen to the answers about SQA and role to hear whether this tester will work happily in a group that does not follow a process standard like ISO 9000-3 or CMM. If this tester will work in one of those environments, I would probe their knowledge of those standards and their sympathy with them.

The tool / technology questions here are really asking, what is the role of technology in your group, and what should it be? I'll cover details later.

Knowledge of areas of testing

- What are the key challenges of testing?
- Have you ever completely tested any part of a product? How?
- Have you done exploratory testing effectively?
- Have you done specification-driven testing effectively?
- Should every type of business test its software in the same way?
- Economics of automation?
- Role of metrics in testing?
- Describe components of a typical test plan?
- Tools for interactive products?
- Tools for database products?
- Cause-effect graphs?
- Data flow diagrams?
- When have you had to focus on data integrity?
- Typical bugs at your last company?

This list illustrates the questions that I ask. The actual list that I would use will depend on the company, application areas, etc.

The question, "Should every type of business test its software in the same way?" provides some indication about the candidate's open-mindedness and about the breadth of the candidate's actual education and exposure to the field. I have a Right Answer for this—No, every type of business should not test its software the same way. I expect to hear that life critical applications probably go through more rigorous testing and process management than here-today, new-version-tomorrow web-based applications. I would like to hear that different application issues call for different approaches. For example, the techniques that you apply when the key issue is whether a financial application (written in COBOL, doing fancy stuff with a huge database) is computing the right answers is very different from the techniques you'll use to test the interactive competence of a word processor. If I'm lucky, I'll hear the candidate talk about the different paradigms of software testing (the different ways that people think about the core issues of the field). Within the black box world, for example, James Bach (1997) identifies Domain Testing, Stress Testing, Flow Testing, User Testing, Regression Testing, Risk-Based Testing, and Claim-Based Testing as separate techniques. In my course on testing (Kaner, 1998), I identify 9 paradigms that lead testers to think of different kinds of criteria for what makes up an effective test case or test suite: Domain Testing, Stress Testing, Risk-Based Testing, Random Testing, Specification-Driven Testing, Function Testing, Scenario-Driven Testing, User Testing, and Security Testing. I don't think that there is one right

partitioning of paradigms, but it is a mark of maturity in the field to recognize that two different groups can have substantially different views of what is a good approach to testing, and both can be right (given their context).

For a senior candidate (individual contributor or test supervisor), I want to find out what they think about various common issues in testing. How sophisticated is their thinking? Not whether I agree with them, but whether they have a well developed point of view. I also want to give them a chance to describe and evaluate the tools that they've used.

A highly skilled tester / test manager for interactive applications (games, word processors) might be clueless about high end data storage or financial applications. That data-oriented questions illustrate questions that I'll ask in order to probe sophistication in the testing of an application area. For a different class of applications, I'd ask different questions.

The "typical bugs" question is trying to get at the underlying question--"What kinds of problems with products are you used to dealing with?"

Interest and skill in this company's areas of application

- Product-category specific questions

This section gives the candidate a chance to show me that she is a subject matter expert.

Project Management

- How do you prioritize testing tasks within a project?
- How to develop a test plan and schedule? Tell me about bottom up vs. top down approaches?
- When should you begin test planning?
- When should you begin testing?
- Do you know of metrics that help you estimate the size of the testing effort? How do you scope out the size of the testing effort?
- How many hours per day should a tester work?
- How should staff overtime be managed?
- How should your overtime be managed?
- How to estimate staff requirements?
- What to do (with the project tasks) when the schedule fails?
- Conflict with programmers?
- How do you know when the product is well enough tested?

These questions are primarily for mid-level to senior testers and for supervisors. At some point in seniority in many companies, a tester becomes largely self-managing. For example, the tester is assigned to a fairly large area of work and left pretty much alone to plan the size, type, and sequence of tasks within that area. Drucker (1966) wrote a remarkable book on time management, decision-making, prioritization, and survival skills for managers. Drucker includes any knowledge worker who has to manage her own time and resources within his definition of "executive." One of the successes in the development of my management style comes from learning to see the managerial nature of my mid-level individual contributors.

Staff relations

- What characteristics would you look for in a candidate for test group manager?

- What do you think the role of the test group manager should be? Relative to more senior management? Relative to other technical groups in the company? Relative to your staff?
- How do your characteristics compare to the profile of the ideal manager that you just described?
- How does your preferred style work with the ideal test manager role that you just described? What's different between the way you work and the role you described?
- Who to hire in a testing group & why?
- Role of metrics comparing staff performance in HR management?
- How to estimate staff requirements?
- What to do (with the project staff) when the schedule fails?
- Tell me about staff conflicts that you've handled?

This section is primarily for supervisory staff.

I ask the test group manager questions only of management candidates. These four questions are really enlightening, during the phone screen and during the face-to-face interviews. I think that it's entirely fair to ask these of someone who has management and hiring experience, and I expect thoughtful answers. Here are examples of some of the insights that I can get from these answers:

- *The candidate's picture of an ideal manager is dramatically different from his image of himself, or from the impression of him that you've built up during the interview. This can be a* huge *red flag. Not always. In some cases, for example, this reflects genuine humility. But a significant mismatch should make you think. And it should help you structure questions for the face-to-face interview.*
- *The candidate's description of the ideal manager exactly manages his perception / presentation of himself. This person* might *not be pathologically egotistical. He might just be trying to manage an interview in a way that puts himself in a good light—I think this is OK, as long as he doesn't lie or exaggerate. But again, it gives me a lead on future questions.*
- *The candidate's description of the ideal manager differs strongly from the expectations of your company. I expect to see some differences, but if there are fundamental differences in expected role or in expected relationship with the staff, then I will wonder whether this person can fit with the company. Wonderful, brilliant people might fit perfectly in some companies and poorly in others.*

Knowledge of the company

- What does the candidate know about the company?
- What questions does the candidate have about the position or the group?
- What questions does the candidate have about the company?

For example, if the company has a web site, I'll ask the candidate whether she has looked at it. If we scheduled the interview in advance, then maybe the candidate will have looked at the site. But if I'm simply phoning in response to the resume, the odds are good that the candidate sent the resume without checking the web site.

By asking the candidate during the phone screen whether she has seen the web site, I set up a follow up question for the face-to-face interview. If I told the candidate where to look during the phone screen, and she hasn't looked by the time she comes for the interview, that's not a good sign.

I might also offer to send marketing materials or company profile materials to the candidate before the face-to-face interview. I will certainly agree to this if the candidate asks me for the material. This helps the candidate prepare her questions (she should have some).

THE APPLICATION FORM

When the candidate comes in for an interview (or you can mail or email it to him in advance), he should complete an application form. Your HR people probably have one handy.

The typical form sets the candidate's experience out in chronological order and asks a variety of other standard questions about the candidate's background. It is useful to interviewers to have this information in a standard format.

This form is particularly important when interviewing someone who gave you a functional resume. People often select that format in order to hide problems in their chronology or to make it easier to exaggerate what they have done across companies.

For more on standard application forms, see Rosse & Levin (1997, Chapter 7).

THE INTERVIEW: PREPARING FOR THE INTERVIEW

Over a period of one to three days (perhaps spread over a few weeks), a candidate will meet with up to ten (maybe even more) of your company's staff. This is expensive. It interferes with other work. You should prepare that time in advance, so that you spend it effectively.

Every interviewer needs some basic information, such as a copy of the advertisement, the candidate's cover letter (if there is one), and the resume. If the candidate's resume is light on detail, then your chronological notes from your phone screen are valuable.

There has to be an interview schedule. People have to know when they will meet with the candidate and who they will bring the candidate to. You probably have to book the appropriate conference room in advance. You might want to make restaurant reservations in advance, so that your staff and your candidate don't waste time standing in line waiting for a table.

If you're going to give the candidate a demonstration of your products, you want to have a machine set up and available for the purpose.

You probably also want to review your notes and decide what classes of characteristics are most important to interview for. (See the discussion in the next section.)

There's value in having a brief pre-interview meeting to divide tasks and establish ground rules. For example, how independent do you want peoples' impressions to be? In your process, once someone has interviewed the candidate, can he discuss it with someone else who has interviewed the candidate? Can he discuss it with someone who has not yet interviewed the candidate? Under what circumstances can he discuss the interview with non-interviewers?

It's valuable also to educate your staff so that they recognize some common mistakes in reasoning or in appraising candidates, and so avoid them. For example, I try to help people understand:

- *Someone can be a wonderful person, very bright, and very competent, but still be inappropriate for the position at hand.* A rejection of the candidate's suitability for this position is not a rejection of that person.
- *A candidate can be weak in some areas even though she is superb in others.* (This is the problem of the "halo" effect.) For example, a person can be analytically talented, a solid mathematician, but incapable of putting together even a simple test plan, even a simple sequence of relevant tests. (Hard to believe? It was for me, too. But this is a real case.)
- *A candidate might be weak even though he has great credentials.* It's remarkable what glowing letters of reference a company will give a problem employee as part of the process of convincing the employee to go away. It's remarkable how many great companies some losers can amass on their resume. Once they start at a well-known, high quality (of testing) shop like Adobe or Apple or IBM or HP or Microsoft, other companies will hire them. To give you a sense of how extreme this is, I've seen two companies that hired a senior staff member who turned out not only to be (in my opinion) incompetent but who had also been repeatedly involved in sexual harrassment complaints and/or

litigation. Their credentials appeared to be impeccable but they were disasters. Both of them might have been avoided by more careful interviewing, reference checking, and respect for the rule that allows any interviewer a veto.

- *A candidate might be unacceptable even though he doesn't look unacceptable at first glance and you are desperate or just really sick of interviewing.* At some point, you might feel like you'll hire the next candidate who can prove that they can breathe. Don't do this. If you have to hire in desperation, bring on a short term contractor to fill the seat. Find the best contractor that you can, pay what you have to pay, and buy yourself some breathing room.
- *A candidate who acts oddly during the interview won't somehow improve when you hire him.* If he shows up late for the interview, with no good excuse, why do you think he'll keep these types of commitments later? If he makes promises that he doesn't keep, if he doesn't do his homework, if he fumbles the assignments or snaps at people under pressure, he's showing you what you're going to see later. He's showing you the person that you're going to work with. If you don't like that person,

- *A candidate might be acceptable even though she is not a perfect match for your fantasy of the perfect candidate.* Nobody will perfectly match your ideal candidate. Remember the wisdom of the Rolling Stones: "You can't always get what you want, but if you try sometime, you might just find *Imperfections are normal. We are human. People will come to you with weaknesses as well as with strengths. You want to hire someone with the strengths that you need and the weaknesses that you know how to manage.*

THE INTERVIEW: DIVIDING ISSUES AMONG STAFF

You want to know what this person knows, and how he thinks, and what his skills are, and whether he's a decent human being, and whether you can work with him. That's a lot of questions. You have the huge list of nice-to-have characteristics that I provided above (plus others that you've added). Which of these are important? Make a much shorter list. For each item on the list, ask two questions (and get them both answered) in the pre-interview meeting:

- How are we going to find out about this?
- Who is going to find it out?

For example:

- One interviewer might demonstrate the product to the candidate. This is partially an important courtesy to the candidate, but it also gives the interviewer a chance to watch how observant the candidate is. Does the candidate ask questions? Does he try things? Does he take notes? Is he interested?
- One interviewer might focus on the technical programming and design knowledge of the candidate who claims to be a competent programmer. The same interviewer (perhaps) or someone else might focus on the candidate's knowledge of specific test automation tools.
- One interviewer might focus on the test planning aptitude of the candidate by walking through some test planning exercises.
- One interviewer might focus on the subject matter expertise of the candidate (if he claims to have such knowledge).
- One interviewer might ask questions focusing on how well a test manager candidate will train staff, support growth along their career paths, and provide them with growth opportunities. Maybe this person is also appraising negotiating skills, integrity, management of staff under difficult circumstances.
- One interviewer might serve as a guide, walking the candidate through the building, walking him from interview to interview, and answering any questions that the candidate has. The guide is at a peer level to the candidate and makes it clear that she'll be glad to answer questions. The candidate

might feel more comfortable asking questions of one person who feels more like a host than an interviewer. The guide answers the questions, but also reports the questions back to the group. Perhaps this person asks a few questions of her own, gently probing the inquisitiveness of the candidate.

Unless you explicitly note the issues, you won't even realize that you're missing many of them. Unless you assign them intentionally, you won't cover them all.

(Gosh, it's just like test planning.) (But of course. You are conducting a series of tests of a very complex subject matter. Coverage is an important issue, just as it is for testing software.)

No group is perfect at this the first few times they interview a candidate. But you can get better at it by walking through the issues again in the post-interview meeting.

THE INTERVIEW: QUESTIONS

I can't begin to list all of the interesting questions that you can ask in an interview. Some of my thoughts are reflected in the issues that I suggested for the phone screening. These are all good issues for the main interview. The main interview should also look at skills (by demonstration as well as by discussion) and detailed knowledge.

I refer to "issues" instead of questions because you can ask very different types of questions to get at the same issue.

Here are some of the key types of questions:

- Hypothetical vs. Behavioral
- Factual vs. Opinion
- Closed vs. Open-ended
- Traditional interview questions

Hypothetical questions

The hypothetical question is a what-if question. You describe a situation and ask how the candidate would deal with it. Normally, the candidate can ask you any questions that she considers appropriate, and then she frames her answer. You appraise the answer and, if you're paying attention, you also take note of the kinds of questions she asked. These tell you something about the analytical approach of the candidate.

I like to ask a few hypotheticals, not because I'm necessarily interested in the answer (I ask behavioral questions when I'm really interested in the answer), but because I am interested in seeing how the candidate gathers information. So my hypos lack some critical details.

And, of course, sometimes the answers are informative too. But often, the answers reflect what the candidate thinks you'd like to hear, or reflect an ideal situation rather than anything that the candidate has ever achieved or even attempted.

Behavioral questions

The behavioral question probes the candidate's actual experience.

For example, suppose that you want to appraise a candidate's understanding of bug tracking system design. You might ask:

(Situational)

> "Suppose that someone asked you to design their bug tracking system. What would you suggest as the most important characteristics of the system?"

Or you might ask (behavioral):

"Have you ever had to design a bug tracking system? How did you go about deciding what were the most important characteristics of the system? What were they? Did you actually succeed in building them into the system? How well did the system work? What did you learn that would help you design a new system in the future?

Rosse and Levin (1997, p. 173) provide more examples of what they call situational and behavior-descriptive questions. Risser (1993, pp. 150-152) provides more examples and additional useful discussion.

When interviewing a test manager candidate, I spend time creating some behavioral questions that tell me how this person has handled problem employees. There are several variations, including the employee who makes a political mess by criticizing the product's (or the programmer's) quality at an inopportune time, or the employee who has been a great worker for a long time but has developed a drinking problem, or the employee who is disillusioned with the company and is spending more time visiting with other staff and complaining than on getting work done. I want to be able to predict how committed this manager is to supporting, defending, and growing her staff, and I also want to be able to predict how she will demand discipline when she must.

Factual vs. opinion questions

If I ask someone to tell me what they know about ISO 9000-3, I might be asking for their knowledge ("What is it?") or for their opinion ("Do you like it?") Ideally, I would know which question I'm asking before I ask it. If so, maybe I woud ask the question better: "Can you describe ISO 9000-3?" or "What do you think of

Factual questions are important. At least one interviewer should ask several of them, to test the candidate's detailed knowledge of an area.

- Some candidates for a lab technician's role know a lot about printers, video cards, and other peripherals. If you don't ask, you'll never realize that one person is an expert (even if she is modest) whereas another person is only slightly knowledgeable (even if he is boastful).
- A candidate who claims to know a lot about QA Partner ought to be able to answer questions about its syntax, bugs, special capabilities, and the ways that people use it to create test suites optimized for different characteristics (maintainability, speed, re-use of scripts for foreign-language versions of the software, whatever). Don't just let the candidate tell you the (only) three things that he knows. Ask him questions that he doesn't volunteer the answer to.
- A candidate who claims to be active in the software quality community and interested in promoting the professional development of her staff ought to know who the main professional societies are, what the differences are between the certifications (such as ASQ's CQE or CSQE and QAI's CSTE and CQA), what conferences and courses are available to staff, etc.

Opinion questions are useful too:

- The candidate's opinion might be important. For example, if your company is committed to an ISO 9000-3 program, and the candidate thinks that there's nothing wrong with this standard that you couldn't fix with a shredder and a magnet, then you probably don't need to spend much more time interviewing him.
- Often, the question is not whether the candidate's opinion is right or wrong but whether the candidate forms opinions thoughtfully. For example, suppose that you ask the candidate about ISO, and he gives his negative opinion. Your next question might be, "Why do you think that?" Some people turn out to despise ISO 9000-3 because that attitude is fashionable in some circles or because their manager told them it was stupid or because they think that's what you want them to say. Some other people despise ISO 9000-3 because it was badly applied at a place they worked, and they've seen it badly applied elsewhere. They even read a book about it. Some people have a large set of thoughtless opinions. Others form opinions more carefully. If I'm hiring a lab tech, maybe I don't care. But if I'm hiring a senior tester, I want the one who is committed to knowing what he's talking about.

Closed and open-ended questions

A closed question calls for a yes or no answer, or a very short factual answer. Occasionally they are useful. Usually, they are the product of an untrained questioner. Closed questions often tell the candidate what answer you expect to hear, and so she just agrees with you because that's all you're calling for. Other closed questions are hostile in tone or nature (these are the questions used in cross-examinations in court) and they make people defensive. The candidate feels as though you are trying to put words in her mouth. (You are.)

An open-ended question calls for a broader or more detailed answer. It calls for more input from the candidate and provides very little input from you.

For example, (closed question):

> "You agree, don't you, that maintainability is one of the most important characteristics

For example, (open-ended question):

> "What do you think are the most important characteristics of an automated test suite?" (followed up with) "Why?"

Traditional interview questions

Here are some of the traditional "good" interview questions. Note that they are all open-ended, which is good.

- Tell me a bit about yourself.
- What are your strengths?
- What are your weaknesses?
- How would you feel if one of your subordinates was promoted to a position above yours?
- Why did you leave your last job?
- Why are you interested in joining our company?
- What makes you interested in software testing?
- Where do you see yourself X years from now?

It's probably worth having someone ask them, just to hear the answers, but many people practice canned answers to these things. Most people (of those who practice answers) practice their own answers, with their friends. But some people look for standardized answers that will be socially acceptable. Rothstein (1996, p. 13) (a book of standard answers) suggests the following answer to the question:

> "Q. How would you feel about one of your subordinates being promoted to a position above yours?
>
> "A. I guess it would depend on who it was and the circumstances in which it happened. If I had honestly felt that the person deserved the promotion, I might be a little jealous, but I'd also be among the first to congratulate him or her. But if I had reason to believe that it was due to backroom politics or personal favors, I'd probably be very

By the way, let's turn this into a behavioral question.

> Q. Has a subordinate ever been promoted above you? (If yes) What happened? How did it feel? (If no) Has this happened to a friend of yours? How did they take it? How would you have felt in that situation?

THE INTERVIEW: WORK SAMPLES

When I schedule an interview, I ask the candidate if she has any work samples that she can bring in. I very carefully don't ask the candidate to bring in any secret documents. I ask if she has anything that she's done that she can share with me.

Confidential work samples

When the candidate arrives on interview day, I'll briefly look at what (if anything) she's brought. I won't look at the details yet. If there are any apparently confidential materials, I have to manage this issue with care. The conversation might go like this:

> Q. This is the test plan for your current product? Cool. Have you released the product yet?
>
> A. No, we'll probably release it in a few weeks.
>
> Q. Oh. Wow, I really appreciate your bringing this, but does your company consider these documents confidential? Would they mind if I looked at them?
>
> A. Well, yes, they probably would. But you asked for my work and this is what I have. What was I supposed to do?
>
> Q. I understand. I'm very sorry about confusing you. We never look at confidential documents. Would you mind putting them away and not showing them to my staff during the interviews? I could get into trouble with my management if my staff look at this.

Notice two things:

- My staff aren't going to see these documents and I haven't looked at them carefully enough to learn anything from them (beyond the fact that they are confidential).
- I've avoided blaming the candidate or calling her a dolt for bringing in company confidential documents. I want to minimize the extent to which the candidate feels awkward over this. I don't want her to blow the interview because she feels bad about this.

Now, behind this, I have to decide what to do about this candidate.

- If I decide that this candidate is probably naïve, then I won't hold this against her. We'll have a long talk about trade secrets at hiring time, and more training later.
- If this is a mangement candidate, she should know better than to make personal use of company secrets. The odds are high that I will disqualify this candidate.
- If I don't know what to think about this candidate, I'll find some way to probe further on questions of integrity.

One last thing to notice. I didn't ask for confidential materials, but I haven't made a point of saying "Don't bring confidential materials to the candidate." This is another case of letting the candidate show me who she is and what she does. By the way, if she asks, "Do you want me to bring confidential materials?", then of

Usable work samples

Suppose that the candidate brings stuff that you can look at. Publications, or generic charts, or test documentation that his manager has agreed he can use. Then you want to review this material in detail.

- Read some of it. Ask questions about it. What is special about it? What was challenging?

THE INTERVIEW: SAMPLE TEST PLANS

You might have documentation that is simple enough for the candidate to review. If so, have her inspect your document and criticize it. This is primarily interesting if the candidate claims to be good at auditing or inspecting testing documentation. Let her demonstrate her skill.

THE INTERVIEW: AN AUDITION

DeMarco & Lister (1987) recommend that you hold an "audition" for candidates. The candidate comes to the interview with a prepared 10 or 15 minute presentation on some aspect of past work.

I would include the full group (anyone who wants to interview the candidate, or who wants to see the audition) as attendee and would allow them to ask a few clarification-type questions, but no I-disagree-with-you type of questions. (This can be a very threatening situation for some candidates.)

The more that this candidate will have to present material in public or the more experienced in presentation that this candidate is, the more appropriate I think that this interview style would be. It is another type of sample of the candidate's work. As DeMarco & Lister put it, you want to see a juggler juggle before hiring him.

THE INTERVIEW: BUG REPORTS

It's fun to talk about how to write a bug report. It's interesting to have the candidate actually write one. I think that this is a particularly important test for testers who have a few months or a few years of experience. You'll find a lot of variation in how well they can do this, one of the most important and most basic parts of their job.

Find a reasonably straightforward bug in part of your software that is reasonably easy to understand. If none of your product's bugs fit that bill, get one from www.bugnet.com.

In my course on black box testing, I demonstrate a simple bug in Windows 95 Paint:

(1) Start the program.

(2) Color the background black.

(3) Zoom 200%.

(4) Select an area using the Freehand Select tool.

(5) Hit Del to delete the selected area.

RESULT: Either nothing gets deleted or some other area (lower and to the right) will be deleted instead.

In class, I also show a few additional (irrelevant) steps. I do some moving and deleting at 100% zoom first (nothing bad happens). I draw a circle and run all of my tests by selecting around the circle (even though the circle itself is completely irrelevant.) I show that the bug occurs with deleting, but not with moving (but that there is no bug first if you move an area and then delete it). I draw the circle in the lower right corner, where it appears that nothing gets deleted (rather than the wrong area being deleted). Then I grow the window (so you can see the whole canvas) or select an area up and to the left, and the wrong area gets deleted. I demonstrate the bug, give students screen shots of all the steps (there are 15 screens), and ask them to write a bug report. I walk through the room answering individual questions (including "What would happen if I did

Even for this very simple bug, there are stunning individual differences among the students (who are usually experienced testers).

I've used this bug with perhaps 400 students by now. Some students can write a good bug report in 5 minutes. Others are still struggling after 30 minutes. Some write effective summaries and describe the bug step by step. Others, even some articulate people with 10 years experience, write a long, disjointed paragraph that is hard to understand.

It's easy to recognize a really good bug report, but even at leading software companies, a large percentage of the students (usually 6 months to 20 years of testing experience) don't write really good bug reports. It takes a while before you can tell the difference between a pretty good report, a not-so-bad report, a mediocre report, and a bad one. Try your bug on your staff, so that you have a sample of reports for comparison, before trying it on interview candidates.

THE INTERVIEW: TESTS AND PUZZLES

Several groups do some type of informal aptitude testing, using logic puzzles or numeric puzzles. I don't object to these, but I don't think that they are as informative as some people think they are. Here are some of my concerns:

- There are huge practice effects with logic and number puzzles. I used to do them with my daughter when she was about 12. She got pretty good with them. That didn't mean she was smarter, and it didn't make her a better tester. It meant that she was better at solving puzzles. These practice effects are the basis of the large industry of test preparation for SAT, LSAT, GRE and other standardized college admission tests. Practice effects (previous experience) last quite a long time and they are more pronounced in speeded tests. They are more pronounced in nonverbal tests and performance tests (Jensen, 1980). So, a person who looks really great on these tests might simply be more familiar with them. A person who looks like a dummy might have no experience solving them but (in my experience) be smart and an excellent tester anyway.
- Speed tests select for quick but not necessarily for thorough thinking. Mental rabbits. Tortoises sometimes design better products or better strategies for testing products.

A simple testing puzzle

Another old favorite among commonly used speed tests is Myers' (1979, p. 1) self-assessment. The candidate is given an extremely simple program and asked to generate a list of interesting test cases. The specific program involves an abstraction (a triangle).

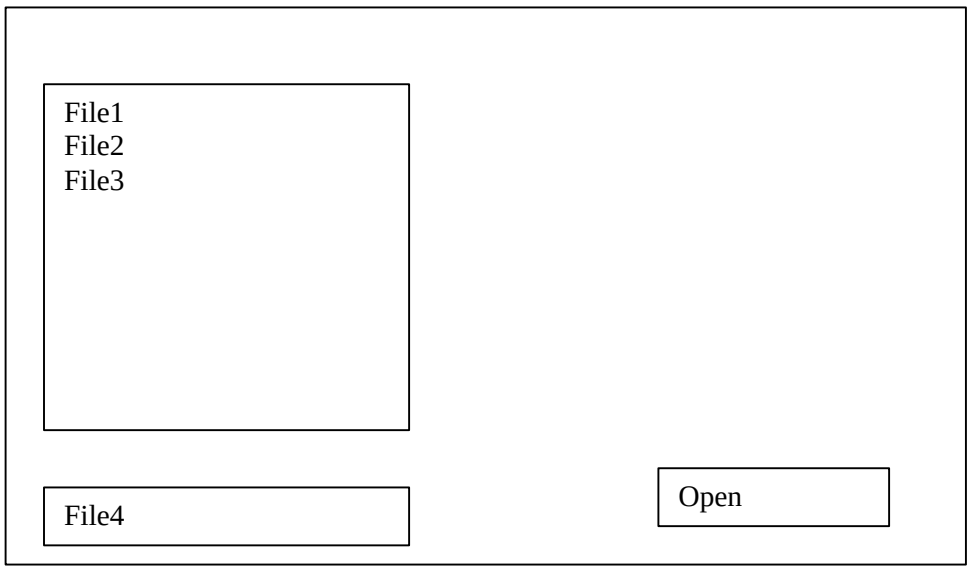
I prefer this because it tests something that testers will actually do (analyze a program and figure out ways to test it). However, there will still be practice effects. Average testers who worked through Myers will probably do better than strong testers who have never seen the puzzle.

Additionally, I suspect that among skilled testers there will still be cultural differences in success with this test. I suspect that someone who is used to dealing with abstractions, such as geometric abstractions, or with logical relationships among numbers, is probably going to do better than someone who tests user interfaces or compatibility with devices.

Another simple testing test

Here's an illustration of a test that I use that allows for cultural variation.

1. I draw a simple Open File dialog on a whiteboard.
2. I explain the dialog. This is an open file dialog. You can type in the file name (where it says File4 at the bottom) or you can click on the file name in the file list. Once you've selected the file, you can click on the Open button to open it. (View this figure in Word in Page Layout mode):



3. I hand the marker to the candidate and ask him to tell me how he would test it. I make it clear that he can have as much time as he wants, and that many candidates take several minutes to think before they say anything.
4. The candidate can make notes on the whiteboard or on paper.
5. The candidate eventually begins presenting his thoughts. I listen, ask questions to clarify, but don't criticize and don't challenge. When the tester pauses, I let him be silent (to think) without saying anything. He can tell me when he's done. If it's ambiguous, I ask him if he has any other thoughts.

This is a remarkable test in the extent to which answers vary.

- One candidate might stay at the surface, pointing out every flaw in the design of the this dialog. (There is no Cancel button. There is no dialog title. There is no obvious way to switch directories. And on and on.)

- Another candidate might skip the UI issues altogether and try testing the opening of big files, little files, remote files (specified by paths that she types into the file name box, such as, d:\user\remote\fubar\File4), corrupt files, files with inappropriate extensions.

There are several other patterns. I think of these as cultural patterns because they reflect a cultural difference across platforms or user communities. For example, back in the days before Windows was a big deal, testers with Mac experience tended to focus on the user interface design, and testers with PC experience tended to focus on reliability of opening different file types from different places but were relatively blind to dialog box design. Testers with Amiga experience tended to focus on getting the thing to work under interesting conditions but they did little intentional testing of error handling, and they were less concerned with the niceties of the dialog design. For example, they would test large, existing files but they wouldn't test files that are no longer there. PC testers were more likely to try to open a non-existent file or a file on an empty (no floppy in the) drive.

These patterns of response are based on dozens of interviews, back in 1987 and 1988: I was the founding manager of the Creativity Division's testing group at Electronic Arts, and then was a software development manager at Power Up Software when it was just founding its testing group. The variation was initially puzzling because candidates who seemed equally strong in other ways gave such wildly different answers. Jack Falk and Hung Quoc Nguyen helped me recognize the patterns, and the extent to which they were predictable from the tester's platform.

I didn't try Myers' triangle puzzle with these candidates, but my bet is that testers who were more concerned with logical data relationships would have done better than testers who were more concerned with UI-driven products. And yet testers with either of these backgrounds might have been equally bright and equally effective with the application that I would ask them to test.

As I came to recognize the variation in responses, I changed how I used this dialog:

1. I presented the dialog, gave the candidate the marker and whatever time he needed, and encouraged him to give me his thoughts.
2. Then I complimented him on his analysis (even if he did badly, I tried to be encouraging) and I showed him some other types of tests that he had missed. I explained that no one got all of the types of tests and that some people missed some issues because they were nervous or they thought they had been rushed. I spent most of the time showing different types of tests and suggesting why they might be interesting (what kinds of bugs they could find).
3. Then I erased the whiteboard, drew a Save File dialog that was just as badly designed (a few of the UI design flaws from before still there, some fixed and some new ones) and asked the tester to try again.

The *real* test was the second test. For this one, everyone had just received an initial practice test and some coaching, so differential practice effects were minimized. Everyone had received feedback, been reassured that they weren't dolts, but had been told that they'd missed some things. Most testers were substantially less nervous the second time through.

My real question was whether this tester was responsive to my style of training. Could the candidate pick up my explanations and do a substantially better job the next time? If yes, and if the second analysis was pretty good, then I had a reasonable candidate (as measured by this test). If the second analysis wasn't much better than the first, then this candidate was unlikely to be hired. This might be a really bright, well intentioned, interesting person, but if he doesn't learn when I teach, he needs a different teacher.

Occasionally, I have dispensed with the second test because the candidate did impossibly badly during the first test, or was extremely defensive or argumentative during my explanation of alternative tests. Both of these have been rare, but they happen. Usually, this means that I'm done with this candidate. I'll spend a little more time looking for a polite way to sew up the interview, but he won't be hired.

More complex performance tests

Sometimes you know exactly what you want the candidate to do, it's a specialized task, and you don't much care if she is weak in other areas. In this case, I want to try to find a way to measure the candidate against the task at hand.

For example (details changed to protect confidentiality), a colleague and I interviewed a candidate for a senior position that involved performance-related testing of a complex product. This candidate had experience modelling complex systems, was very smart, had a solid technical background, had been in the business for years, and was very good at oral presentation. Despite that, I had some reservations. Ultimately, my colleague and I agreed to pose the candidate a puzzle that would be representative of the type of work that she would do.

1. We explained the task first. We would demonstrate the product before lunch and answer any questions that she had. We would run any test that she requested. Then when she was satisfied, we would go to lunch and she would explain her approach to testing any aspect (her choice) of the performance of the system. The candidate said that she understood, and she agreed to do it.
2. We did demonstrate the system and, if she would have asked any questions, we would have answered them.
3. We did go to lunch and we did discuss the performance of the system.

In the particular case, the candidate constantly challenged the design of the product while we demonstrated it. She would explain how this or that was probably slowing up the system. We reminded her that the task was to (a) observe and then (b) figure out how to test, and not yet (c) file bug reports. She persisted, we reminded her again, she persisted, and over the next hour she didn't learn a lot about the system. When we went to lunch, she lectured us on what was wrong with the system, but said that she'd need more information before she could tell us how to test it.

This was a bright candidate, and until this part of the interview, she had a significant chance of being hired. But in this company, this candidate would not have survived.

In short, if you can find a way to present a piece of the job that the tester will actually do, you can see how well the tester does it. You have to make it a fair test, by designing it in such a way that someone who doesn't know your product can still do well at it. That's challenging. But if you come up with a fair test, the behavior that you see can be very informative.

THE INTERVIEW: DEBATES AND CONTROVERSIAL QUESTIONS

When I interview a test manager candidate, either I engage him in a debate or I recruit some other authority figure to engage him in a debate. My expectation is that test managers will have to stand up and be persuasive under difficult circumstances, in the face of contrary pressure from authority figures. I want some indication of how well this person can handle this. It is, in my view, an essential part of the job.

My procedure is simple:

1. Go through the usual interview questions, focusing on the role of the testing group, the use of technology, the importance of specs and test plans, and so on. Encourage the candidate to tell me what he thinks are the most important factors for success, or the things that he has particularly strong opinions about.
2. Eventually, I'll have a sense of what this candidate thinks (a) is really important and (b) he has thought about carefully. There are typically a few possible areas to discuss. I'll choose the one that I can most effectively handle on the other side.
3. For example, suppose that this candidate loves black box GUI automation. I'll comment that in my experience, it has been a waste of time. On the other hand, if he says that it's a waste of time, inefficient, impossible to maintain, then I become a diehard fan of GUI regression tools. In either case, I engage the candidate with questions and politely disagree with his

answers, asking additional questions, often in the form of "But don't you think that . . .?" or

Here's how I appraise the results:

- If the candidate fumbles and stumbles and turns out to not know what he's talking about, I reject him. I only debate on an issue that the candidate has explicitly identified as an area of special knowledge, and I only disagree when he has expressed a strong opinion. If he doesn't have his facts straight, he was feeding me baloney. I have zero tolerance for lies and exaggeration. He's gone. (Of course, I don't call him an exaggerator. I close down the debate, continue the interview with friendly questions, smile, thank him, don't make him an enemy, and veto him in private.)
- If the candidate backs down and adopts my point of view, I get concerned. After all, this is the test manager and this is an opinion that he claims to hold dear. I'm persuasive, but not usually *that* persuasive. If he backs down, it's probably because I am an authority figure in the context of the interview. So what's going to happen in the Real Job when he has a strong opinion, raises it with his boss (or his boss's boss), and encounters some resistance? Will he back down? That's not always the right thing for a *test manager* (or a *software quality assurance manager*) to do.
- If the candidate gets obnoxious (personal attacks on my judgment, calls me stupid or ignorant, raises his voice, treats me without respect) then I predict that he will be ineffective (and maybe quickly fired) when his wisdom is challenged by an executive. Next candidate, please.
- If the candidate listens to what I have to say, acknowledges my points politely, accepts the occasional correction, but sticks to his guns while maintaining his cool, I like him.

Some candidates walk away from this part of the interview feeling that they were unfairly confronted by someone who is closed minded. They might go away and decide not to accept the job if it's offered or they might agree to accept the job but hold a grudge against the interviewer. I'm still learning how to handle this. Probably the best way is directly, to explain at the end of the interview that my style of interviewing is to allow a debate to develop in order to see how the candidate handles it. I then congratulate the candidate, appreciate his attentiveness and his approach to the discussion, and then make my evaluation and decision in private.

Even if the candidate misbehaves, it is important to recognize that this is a high pressure, difficult situation for many people. There is no point attacking this person or insulting this person for responding poorly to pressure. I do my best to smile and show appreciation to the candidate for coming and for working so hard in the interview, even if I consider his performance during the interview completely unacceptable.

I've talked with other test managers who use the debating approach. We stick with it because it is informative, but it is uncomfortable. Some other managers skip that but probe deeply with behavioral questions, like these:

> Q. Tell me about a time that you disagreed with your manager and stood up to her. What was the disagreement about, how did you handle it, and how did it come out? (If the candidate describes a success, follow up with a question asking if he ever stood up, worked hard on an issue, but failed to persuade. How did that feel?) (If the candidate describes an initial failure, follow up with a question asking about success.)
>
> Q. Tell me about a time that a product was shipped over your protest. How did you convey your dissent? What kinds of arguments did you make? Who did you make them to? Why did they fail?
>
> Q. Tell me about a time when you wanted to fire someone but your manager disagreed (or someone senior wanted you to fire one of your staff and you refused).

THE INTERVIEW: FREE CONSULTING?

Some companies use an interview as an opportunity to get free consulting. A few companies have a reputation for this. They have a technical/managerial problem and so they issue some invitations to senior testers to interview with them. The interviewers discuss this current problem and ask for the candidate's opinions. The worst of these companies either drag the "interview" out for several days or cut off communications as soon as they've gotten the advice / opinion / information that they wanted.

Don't do this. It's unethical. It's probably a violation of the minimum wage laws. (After all, these people are doing work for you at this point.) And it's probably fraudulent, if you deliberately interview people with the intent of getting their advice instead of with the hope of hiring them.

There is value in posing realistic puzzles, and your staff will learn a lot about how other people think about testing by participating in interviews. But there's a line between interviewing (giving someone information on which they can base a hiring decision) and consulting (giving someone analysis and/or advice about a current problem).

POST-INTERVIEW MEETING

Suppose that Joe, Sandy, Jane and Ted interview the candidate, in that order.

When Joe is finished, I'll ask for his impressions, but I will ask him not to share them with Sandy until we meet at the end of the day. Similarly for Sandy, Jane, and Ted. Some groups are close-knit, they don't like this, and so we adopt a different rule. Joe can talk to Sandy *after* she has completed her interview. But first, I want Joe and Sandy to both give me, independently, a tentative Yes, No, or Maybe.

If the candidate is clearly failing, then I will send the candidate home early. All of a sudden we will have a rush project that we have to take care of. We are very sorry, but we can't finish this today.

I don't have infinite time to spend on interviews. When a candidate is no longer in the running, I want to stop spending money and time interviewing him. Those of us who interviewed him might briefly meet to discuss it, but I won't spend much time on this.

I've identified the desirability of terminating the interview at several points, because I am conscious that this is an expensive process that can't afford preventable waste. But please don't get the wrong impression. In my experience, most interview candidates make it through the entire day without being sent home early.

The typical candidate has stayed through the day, made good impressions and bad, and now we have to appraise her.

I start by asking for a tentative vote. Do most people like this candidate or not? Yes / No / Not sure. Then we go around the room and trade impressions. Sometimes, this results in a clear, quick decision (No) and so we break quickly and get back to our other work.

My next step is to pull out the list of issues that we were interviewing the candidate against. (See the section on dividing the issues among the staff, above.) We'll work through the list one at a time. For example, suppose that we interviewed a management candidate and we get to the issue of mentoring. Suppose, too, that a pack of juniors was given the task of finding out over lunch how good a mentor this candidate would be, and how helpful this candidate would be in assigning other senior staff as mentors:

- First, the juniors will report on what they asked and what they learned.
- Next (especially if the first feedback came from people who are just learning how to interview), I ask for anyone else's observations. It's often the case that the same issues come up, perhaps as side issues, in several interviews. It's also all too often the case that a candidate will say different things to different people. For example, the candidate might tell the juniors that training is very important, that it will be a priority, and that lots of senior staff time will be spent on coaching. The same candidate might tell the next interviewer (a senior tester) that juniors are a pain in the neck, and that they require too much hand-holding. The candidate might promise to reduce the senior tester's training burden by hiring more senior staff or by subjecting the juniors to sink-or-swim self-training.

Contradictions like these happen. Sometimes they are rooted in a misunderstanding. Other times, they reflect a two-faced candidate.

We walk through the list and by the end realize that we like this candidate a lot and are ready to hire (or to move on to the next stage, perhaps scheduling a final interview with executive staff) or we like this candidate but need more information (which we list, if we can), or that we don't like this candidate.

Sometimes one or two people have reservations that no one else has. We all like the candidate except for this one or two people. This can be difficult for everyone.

- If the interviewer has a firm negative opinion, based on observation and reasonable interpretation of what was said, the candidate is vetoed. Goodbye, too bad, oh well. You might not choose to adopt a consensus model, but I commend it highly.
- The interviewer might have a negative opinion based on a lack of information or on a misunderstanding. This might be dealt with by the in-meeting discussion. It is important that the interviewer be allowed to stick to her guns, and know that she is allowed to stick to her guns. Becoming convinced that she should shift from a veto to an abstention should be the result of "being convinced" and should not be the result of "being intimidated" or "being pressured."
- The interviewer might feel that she would change her mind if she learned certain additional information or if the candidate answered certain additional questions the "right" way. If the candidate is returning for another interview day, she could do her own follow-up interview or she could ask one of the other interviewers to ask the appropriate questions. Alternatively, the dissenting interviewer might agree that the issue can be explored as part of the reference checking process, as long as the right questions are asked and answered during the reference checks. If the answers come out the wrong way, of course, the dissenting interviewer can and (unless she has changed her mind for good reason) probably should veto the candidate.

One piece to keep in mind and to make clear to the group. The meeting doesn't provide the final decision. If everyone agrees to accept the candidate, that is a tentative approval. I still have to check references, and I will probably not broadcast the details of those references to everyone else. I still have to go through the mechanics of developing an acceptable offer. The offer could be blocked for various reasons as we go forward. But the group has spoken, saying that it is OK with them if we hire this person, and that's important.

Another decision that the group might make is that this candidate is acceptable but that the interviewing process is not yet closed. If we have six people scheduled for interviews, we might interview all six before making an offer to the first. The benefit is that you gain perspective when you can compare candidates. The risk is that the first candidate might have a job by the time you get around to offering her a position.

FEEDBACK TO THE CANDIDATE

Throughout this paper, I've suggested that I don't give the unsuccessful candidate much negative feedback, especially when I decide not to hire him. I have several reasons for this:

- Some people find negative feedback insulting, even (especially) when they've asked for it and promised not to be offended. It is too easy to make an enemy through the interview process, and I have no desire to do that.
- Some people become angry and threaten me. I don't like it when people scream at me, or threaten to beat me up. When I was 18, I worked as an assistant manager in a store. Some of my edges were rougher back then than they are now. One person came to the store waving a pistol. He was going to shoot me for insulting his wife. Some people are a little crazy, and if I don't know them, I don't know that I want to criticize them and learn just how crazy they might be.
- Most people that I've given negative feedback to in the past have argued with me. They've tried to convince me that I was wrong, that they deserve the job, that I should extend the interview process and collect some more data. This is not useful to them or to me. It is just difficult.

- Some people argue with me that the basis for my decision to not hire them was inappropriate and in some way discriminatory. Now they're going to complain up and down the corporate chain of command that I am an evil person and they will quote me (or misquote me) to all and sundry. People have threatened me in this way, but it has never gone very far. Still, it is a risk that I would rather consciously manage by minimizing the amount of information that a rejected candidate can misinterpret and misuse.
- If I provide this type of feedback to some candidates but choose not to provide it to others, I am treating people differently in a way that might be characterized as unfair or discriminatory. The people who are most likely to react really badly to criticism seem to be the people who will demand the most forcibly that you give them the feedback if you give anyone the feedback. Adopting and following a minimal-feedback policy for everyone makes it easier to deal with the most troublesome people.
- This is not a culture that gives this kind of feedback. I don't get it when I interview, unless I get it from a friend, or a person who becomes a friend (and then gives me feedback from an interview long, long ago). My friends don't get it. The books that I read don't recommend it. And I don't have a legal duty to provide it. And finally (I'll check with this company's HR to be sure), there is probably a company policy or preference against it.

Instead, I will reject someone by appreciating the time that they spent coming to us (they deserve that, no matter how awful they were after they arrived). And I appreciate their thoughtfulness (they must have done *some* thinking during the interview). And if I can think of anything else that I can honestly praise, then I will. And then I express my regrets, but we found someone else who was a perfect fit. Or I express my regrets but we decided that it just wouldn't work out. Sorry, we can't discuss the reasoning, that's company policy, you know those bureaucrats, but it was really great meeting you. The goal is to cleanly terminate the process, without insulting or hurting the feelings of the candidate.

CHECKING REFERENCES

Always check references. Some of my worst hiring mistakes would have been avoided if I had only been more thorough about checking references.

Despite the fact that most companies have a policy against giving references that contain more than name, dates of employment, and other strictly superficial factual information, many managers will give you additional information if you build some rapport with them and ask polite, direct questions.

To the best of my knowledge, and I am not an expert in this field, the risk to an employer of providing an honest but negative reference has been vastly overblown. Lawsuits over this are, as far as I can tell, extremely rare. There are statutes in several states that make it extremely hard for a former employee to prevail in such a suit and the courts are, as far as I know, pretty unfriendly toward these suits in the other states. Rosse & Levin (1997, especially pages 143-53) have a lot to say about this. Risser (1993, pp. 165-168) is a readable book about the law that provides advice on this. Again, I am not saying that in my opinion as an attorney, Rosse & Levin and Risser are correct. I have not done the level of research necessary to form a lawyer's opinion on this matter.

I can't provide an extended discussion of the reference call—it would take as long as the discussion so far of interviewing. But my basics are pretty straightforward:

- I ask for factual information, checking the candidate's claims. This includes asking for a job description. *After* I get the employer's job description, I ask about specific tasks that the candidate mentioned, and whether these were significant parts of the candidate's job. (Sometimes they were not part of the job at all. Hmmmm.)
- I ask for examples of good performance. What were some of the memorable events that made you happy to have been working with this candidate.
- I ask what training the candidate received. If the candidate claimed that he attended specific courses (or whatever), then I ask whether the manager remembers these. If not, well, sometimes people

forget these things. In my experience, the candidate was not necessarily incorrect in the resume in cases like this. But it's a tiny red flag that might combine with some other red flags.

- I ask questions that came out of the interview. For example, if we had some questions about an automation project that they candidate did, I might ask, "He told me about a product in which he was the lead automated test developer. I think this was BugWare 2000. Do you remember his role on that project?" If the reference-giver says yes (she might well not remember enough details to answer the question fairly), then I ask for a description of the candidate's role and the success of the automation project. If I get pablum (bland reassurance), I Let me tell you my concern. I've heard about a lot of test automation projects like this that failed because the test code wasn't maintainable enough. We didn't ask detailed enough questions to form an opinion about what happened on this project. Can you give me some additional insight?"
- I ask whether the candidate appeared to get along well with the other staff.
- I ask why this person left, or what he said was the reason.
- I ask whether the employer would hire this person again, and why.
- And I ask whether there were any weaknesses in the person's performance.

Beyond these general points, here are a few specific comments.

Asking for negative feedback

When you call for a reference, you can certainly ask whether some aspects of the employee's performance that would make the employer reluctant to hire this person again. You can also ask what aspects of the employee's performance needed improvement. But please, understand that some people will be cautious about answering questions that call for direct criticism of the employee.

- When you call someone for a reference, that person doesn't know you. She doesn't trust you. She has no idea how mature you are, how experienced you are, how likely you are to keep what she says in confidence, and how thoughtfully you will interpret what she says.
- Many hiring managers jump on *any* criticism as the end of the world. I've been astonished by how badly people (managers or HR staff calling for references) have responded to identification of even relatively mild problems. Some very positive references from people who have been very enthusiastic about a former employee have been interpreted as negative because of the answer to a tell-me-about-the-employee's-faults question.
- Even if the current employee is great, and the flaws are weak, many managers have policies against answering this question. Some handle it by providing a virtually meaningless answer, pre-determined Pablum, often saying the same thing about different people. (Maybe you've heard this one? *sometimes deadlines were a challenge, but deadlines are a challenge around here for everybody. He worked very hard."*)
- Others simply refuse to answer the question. That's what I do, and what some of my legal clients do, on my advice. The problem is a complex one because if I give an answer to this question for Joe and refuse to give an answer for Sandy (whose performance was terrible) and Sandy someday sues me and claims, among other things, that my refusal was taken as an unfair criticism of her, then I am in a much simpler defensive position if I can say, "I answer the question the same way for everybody. I tell people that I'm not allowed to answer it, as a matter of company policy." When I've given refusals, the caller sometimes gets very huffy about it. One person interpreted it as a serious negative criticism of a candidate who is, in fact, the single best individual contributor tester that I have ever worked with. (I said that, and also that I had actively recruited this person into two subsequent companies after managing him in a first company, and that I would hire him again, at top wages, any time I had the opportunity. I said lots of other great stuff about this candidate, to no avail.) Please, when someone refuses to answer a question, realize that they are simply refusing to answer a

question that they have probably been told not to answer or not to answer meaningfully. You are not entitled to an answer. You might be entitled to a pattern of answers that does not taken as a whole, intentionally mislead you. But you are not entitled to an answer to a difficult question from a stranger who has no reason to trust or respect you.

Letters of reference

When someone gives you a letter of reference, it often means exactly what it says. But sometimes, it is a negotiated document that carefully expresses everything good that a firing or contract-not-renewing or encouraging-an-employee-to-quit manager can say without lying while carefully avoiding mentioning all of the horrible things that this manager would like to say. If you call that manager for an interview, ask first if he wrote the letter. If he says yes, ask some follow-up questions.

- He might answer them by continuing to paint a glowing picture of the candidate.
- He might answer them by continung to paint a favorable picture of the candidate, but under your smooth questioning, he might reveal some other useful information.
- He might answer them by filling in the gaps, when you ask specific questions. And so you realize that this candidate was not so good an employee after all.
- He might refuse to answer on the ground that company policy forbids it. This is odd, because he did write the letter, so company policy doesn't forbid some level of reference-providing. The underlying problem might be that there is a termination contract that promises that the only thing that the company will say about the former employee is what is in the letter. I'll ask straight out whether this is the problem. Sometimes the manager will tell me (yes, or no).

If someone sends me reference letters with their resume, I feel free to check those references before the face-to-face interview. Sometimes, I'll call one before the phone screen.

If someone gives me a list of references, I feel free to call them after the first face-to-face interview. I am likely to call one after this interview (while making the decision to call back for another interview), and call the others later, when I am making the hire/don't hire decision.

I will also call some other people who are not on the candidate's list. These might be other people who I know, who worked with the candidate. Or it might just be a call to HR at that company, checking employment dates, job description, salary, and asking for any additional information they can give (which, as a matter of company policy, might be nothing).

RED FLAGS

Rosse & Levin (1997) have an excellent discussion of red flag issues (things that come up in the interview, the resume, or the reference checks that should make you think twice about hiring this employee). I don't agree with everything they said—In particular, they raise a red flag when candidates lay out vacation or attendance needs, without, in my opinion, spending enough time on the notion of opportunity hiring. But overall, they provide a good discussion that you might find useful to consider.

Walley & Smith (1998) provide another useful red flag discussion. Deception (lies, exaggeration) is widespread in interview responses and resumes. They focus on the types of ways that candidates mislead potential employers and ways (not all of which I would feel comfortable recommending) to discover this.

REFERENCES

Bach. J. (1997) *Tripos: A Model to Support Heuristic Software Testing*, available at http://www.stlabs.com/testnet/docs/tripover.htm.

DeMarco, T. & Lister, T. (1987) *Peopleware: Productive Projects and Teams.* Dorset House.

Deming, W.E. (1982) *Out of the Crisis*, MIT.

Drucker, P.E. (1966) *The Effective Executive*, HarperCollins.

Kaner, C., J. Falk, & H.Q. Nguyen (1993, 2nd Ed.) *Testing Computer Software*, ITCP/Coriolis currently, but check www.kaner.com for updated publishing information.

Jensen, A.R. (1980) *Bias in Mental Testing*, The Free Press.

Kaner, C. (1998) *Black Box Software Testing*, course available from UC Berkeley Extension, UC Santa Cruz Extension, Software Quality Engineering, or by arrangement with the author.

Lane, C. A. (1997) *Naked in Cyberspace: How to Find Personal Information Online*, Pemberton Press.

Myers, G. J. (1979) *The Art of Software Testing* John Wiley & Sons.

Risser, R. (1993) *Stay Out of Court: The Manager's Guide to Preventing Employee Lawsuits.* Prentice-Hall.

Rosse, J. & R. Levin (1997) *High-Impact Hiring.* Josey-Bass.

Rothman, J.R. (1998) *Hiring Technical People: A Guide to Hiring the Right People for the Job*, Rothman Consulting Group, Inc.

Rothstein, M. (1996) *Ace the Technical Interview*, 2nd Ed., McGraw-Hill.

Walley, L. & Smith, M. (1998) *Deception in Selection*, John Wiley & Sons.

Whitaker, K. (1994) *Managing Software Maniacs*, John Wiley & Sons.

ADDITIONAL REFERENCES

I'm still reading these. They look valuable, and I suggest that you look at them, but I haven't quoted them above. I've noted some of the useful characteristics of these books, but understand that I'm still reading

Beatty, R.H. (1994) *Interviewing and Selecting High Performers.* John Wiley & Sons. *Good examples of questions and questioning styles.*

Constantine, L.L. (1995) *Constantine on Peopleware*, Yourdon Press / Prentice Hall. *Interesting discussions of consensus-based engineering and staff characteristics.*

Cook, M.F. (Ed.) (1992) *The AMA Handbook for Employee Recruitment and Retention*, American Management Association. *There's a lot of policy and procedure material here, including a long chapter on legal issues*

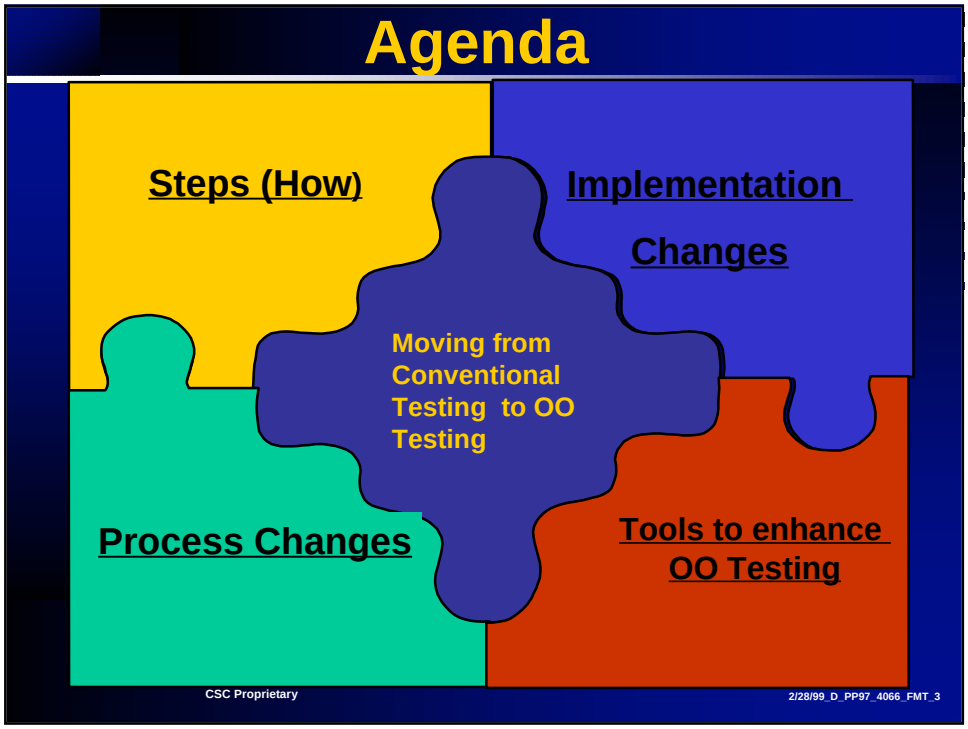
Humphrey, W.S. (1997) *Managing Technical People*, Addison-Wesley. *The discussion of talented people will be useful for enriching the consideration of desirable staff characteristics.*

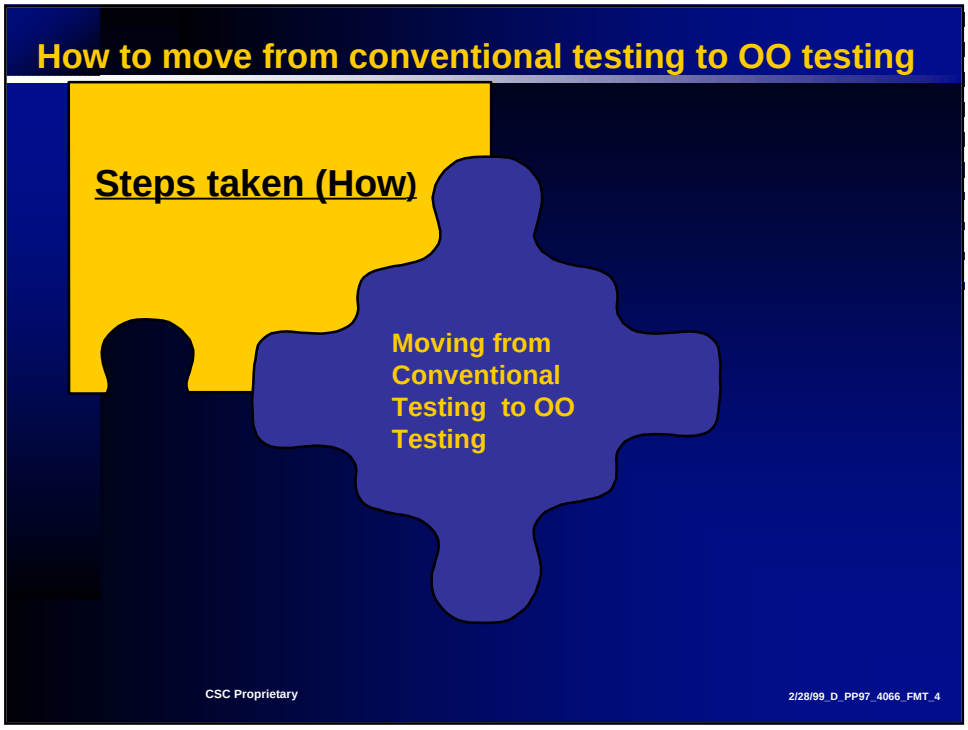
Irish, R.K. (1987, Revised 3rd Ed.) *Go Hire Yourself an Employer*, Doubleday.

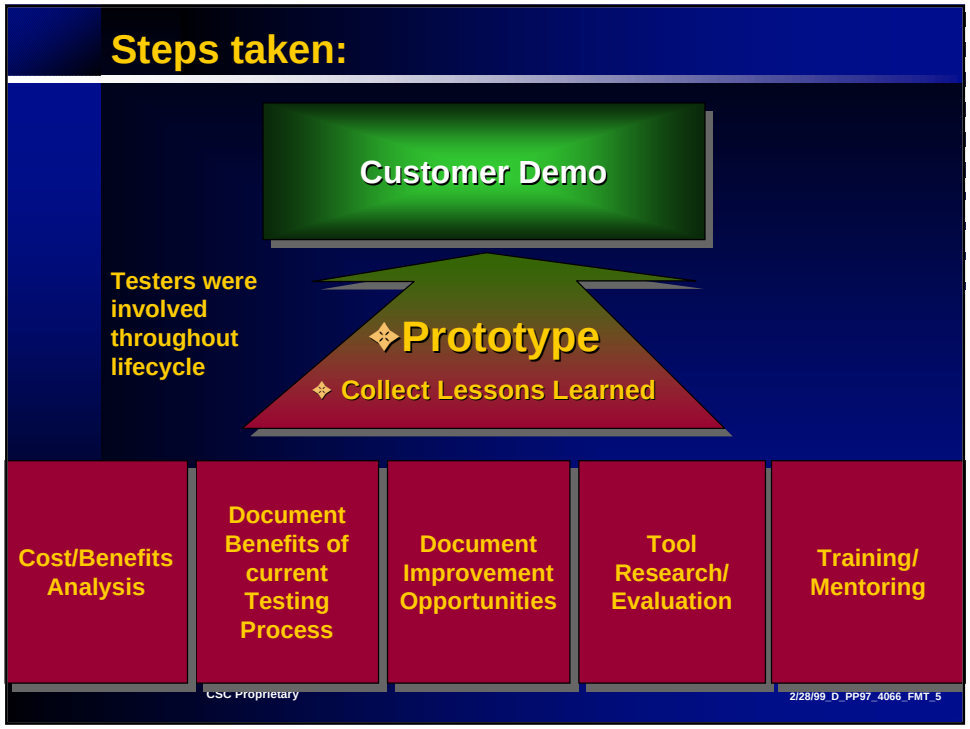
Moving From Conventional Testing to Object Oriented (OO) Testing

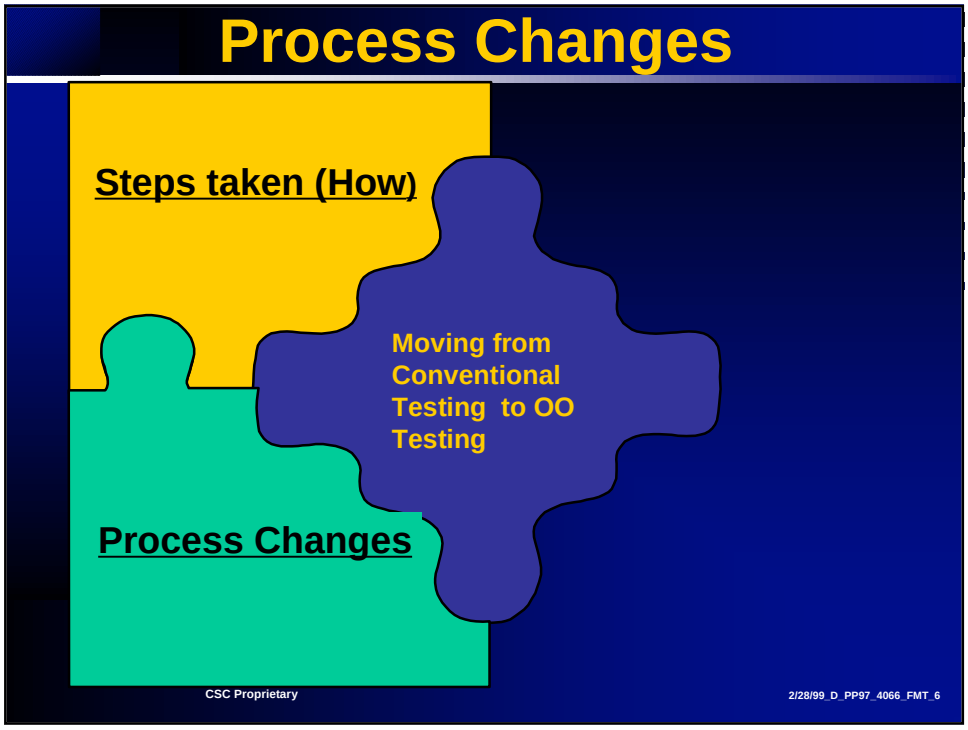
- Elfriede Dustin, *CSC*
- John Paul, *Freddi Mac*

CSC Proprietary


Agenda
Steps (How)
Implementation Changes
Moving from Conventional Testing to OO Testing
Process Changes
Tools to enhance OO Testing
CSC Proprietary
2/28/99_D_PP97_4066_FMT_3


How to move from conventional testing to OO testing
Steps taken (How)
Moving from Conventional Testing to OO Testing
CSC Proprietary
2/28/99_D_PP97_4066_FMT_4


Steps taken:
Customer Demo
Testers were involved throughout lifecycle
Prototype
Collect Lessons Learned
Cost/Benefits Analysis
Document Benefits of current Testing Process
Document Improvement Opportunities
Tool Research/ Evaluation
Training/ Mentoring
CSC Proprietary
2/28/99_D_PP97_4066_FMT_5


Process Changes
Steps taken (How)
Moving from Conventional Testing to OO Testing
Process Changes
CSC Proprietary
2/28/99_D_PP97_4066_FMT_6

Conventional testing to OO testing (cont)

Process Changes:

- Modeling is a vital part of OO (Use Case, Design, etc.)
- e.g. Use Case Model
  - Use Cases and Use Case Diagrams
    - A description of the interaction of a system with the outside world. It is a description of the system.
      - Allows understanding of Business Process

Conventional testing to OO testing (cont)

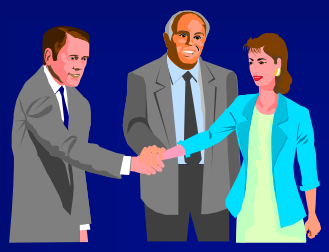
Process Changes (cont):

- Use cases are an effective tool in communicating to users
- Both from a user's, developer's, tester's perspective use cases are simple to model and understand
- System Testing can begin very early as those models are tested
- Criteria for quality models
  - Correctness, Completeness, Consistency

Conventional testing to OO testing (cont)

Process Changes (cont):

- Testing of models and use cases involves a closer interaction between customer, developers and testers



Conventional testing to OO testing (cont)

Process Changes (cont):

- Basis of designing and developing code is now the same as the basis for designing and developing test procedures

Use Cases

- Big OO selling point for users and testers

Conventional testing to OO testing (cont)

Process Changes (cont):

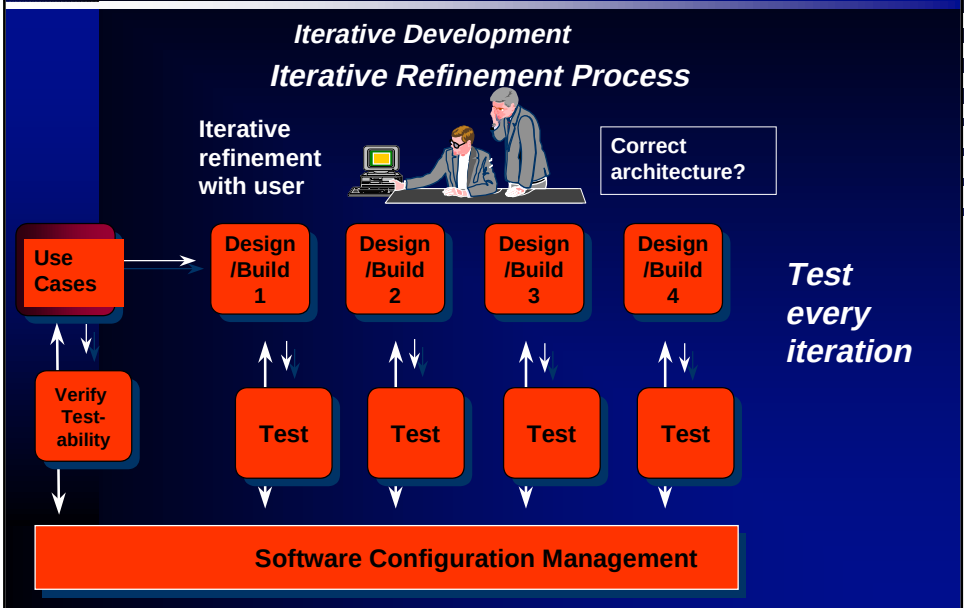
- Work products (use cases, design models, etc) flow across phase boundaries.
  - For example, object models that start during business analysis phase are further refined during requirements analysis

CSC Proprietary

Conventional Testing Process:

CSC Proprietary

Testing Throughout the Lifecycle



Conventional testing to OO testing (cont)

Process Changes (cont):

- Shorter development cycles
- Incremental approach - Iterative cycles
  - Analyze a little, design a little, code a little, test a little
    - » Problems are discovered early
    - » Build on Lessons Learned
    - » Much more Regression testing is needed
- CM

Conventional testing to OO testing (cont)

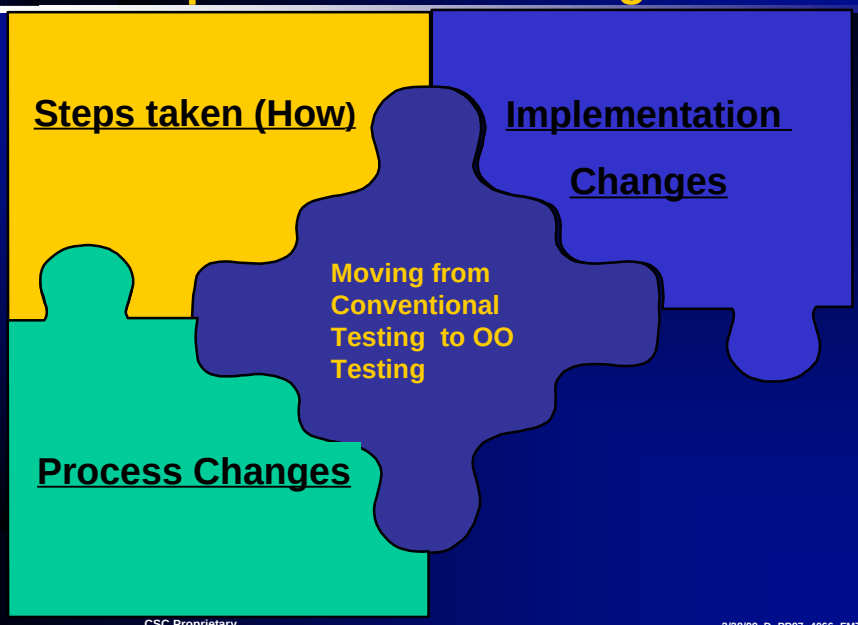
Process Changes (cont):

- Goals of OO development is for code to be reusable, portable, maintainable.
- Same goals apply for creation of OO test procedures
  - Goals of OO testing is for test procedures to be reusable

Process Differences of Conventional testing vs OO testing

Conventional	OO
•Detailed Design Requirements basis for Design and Requirements basis for Test Procedures	•Use Cases basis for Design and Test Procedures
•Various Traceabilities	•Traceability is consistent
•Requirements result in fragmented view of system	•Use Cases - system flow
•Waterfall approach	•Iterative, incremental approach
• Work products are sequential	•Work products flow across boundaries
	• Models are vital
	• Closer interaction between all parties involved
	• Metrics Change

Implementation Changes



Conventional testing to OO testing (cont)

Implementation Changes:

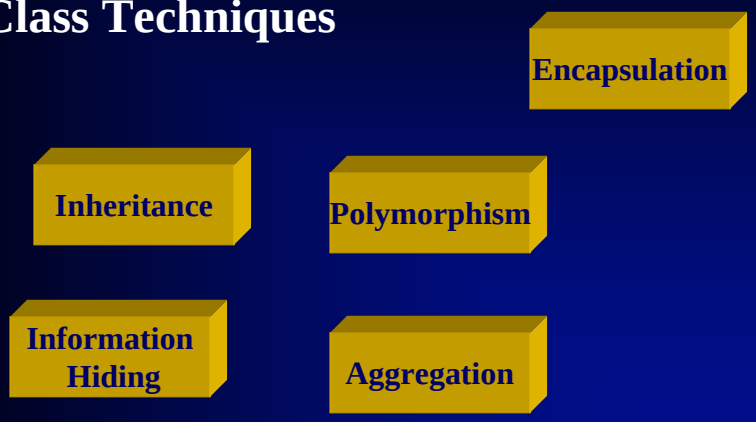
OO Code is developed differently

- Basic unit of testing is a class instead of a subprogram
- Can't test operation of class in isolation
  - Classes interact with each other by modifying the state of the object they are applied to

Conventional testing to OO testing (cont.)

Implementation Changes (cont.):

- Class Techniques



Conventional testing to OO testing (cont)

Implementation Changes (cont):

- Exhaustive testing becomes almost impossible, always very expensive
- Derive Test Cases using specific test coverage techniques
  - Orthogonal Array Testing (OATS)
  - Equivalence Partitioning

Conventional testing to OO testing (cont)

Implementation Changes (cont):

- OO functions are generally smaller
  - More opportunities for integration faults
- Selective regression testing is important
  - Due to progressive changes there may exist test cases which must not be rerun to avoid false alarm

Implementation Differences of Conventional testing vs. OO testing

Conventional

- Smallest software unit: subprogram
- System Testers are usually not involved in unit and integration testing
- Test Procedures are rarely reusable
- Testing Coverage techniques are rarely used

OO

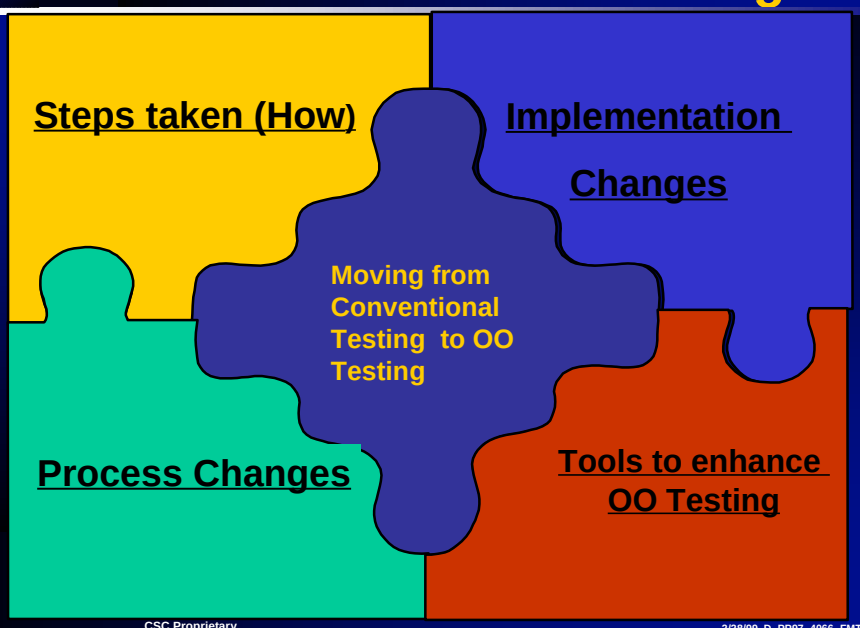
- Smallest software unit: Class
- System Testers audit class (unit) and integration testing
- Test Procedures are reusable
- Integration testing and Selective regression testing is more important - picking correct test cases
- Testing Coverage techniques

Conventional testing and object-oriented testing

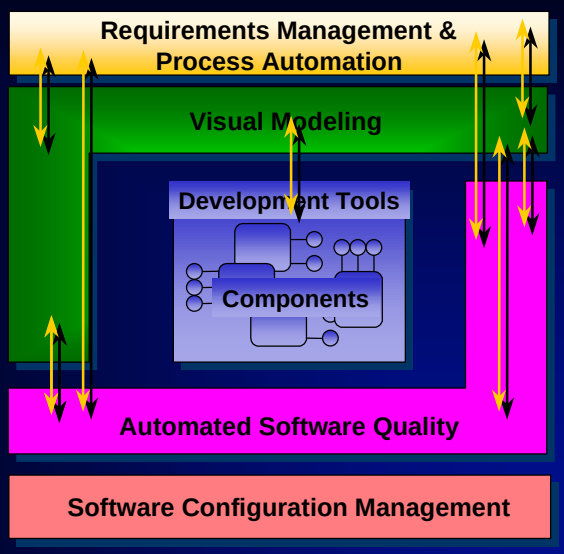
Lessons Learned:

- Testing needs to be involved from beginning of lifecycle
- Manage expectations - Yes OO developed code still needs testing
- Use Cases - major selling point
- Document new standards and procedures to allow for repeatability
- Mentoring and Training are very important

Tools to enhance OO Testing

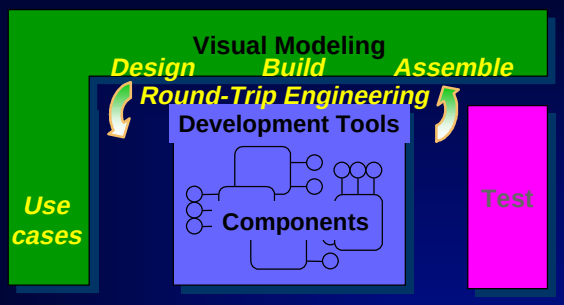


Integrated Suite of Tools



Key: A Well-Designed Architecture

- **Rational Rose**
- Support for ActiveX, Java, Corba
- Support for UML

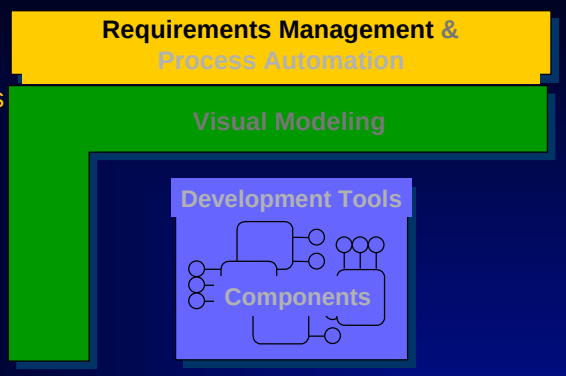


Business Modeling Tools

- Allows for Business Modeling
- Allows for Creation of Use Cases
  - Example: Rational Rose
- Integrated with Requirements tool Requisite Pro

Key: Controlled Iterative Development Process

- **RequisitePro**
  - Organizes, tracks & controls requirements
  - Reduces cost and risk



Requirements Management (RM) Tools

Tool not only used for RM but for TM

- Test Procedures located in one central repository
- Multiple Testers can be assigned a section of functionality of system to write test procedures
- Multiple Testers could access a tool simultaneously without affecting anyone else (allows for locking)
- History of updates is maintained in RM tool (who, what, when)

Requirements Management Tools (cont)

Testers created....

- ~ 5000 Test Procedure Steps

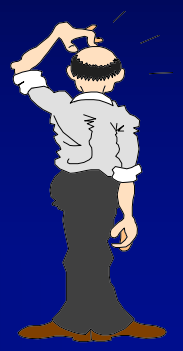
Requirements Management Tools (cont)

- ~ 500 Testable Requirements
- ~ 5000 Test Procedure Steps
- Imagine, creating a traceability matrix manually!

Requirements Management Tools (cont)

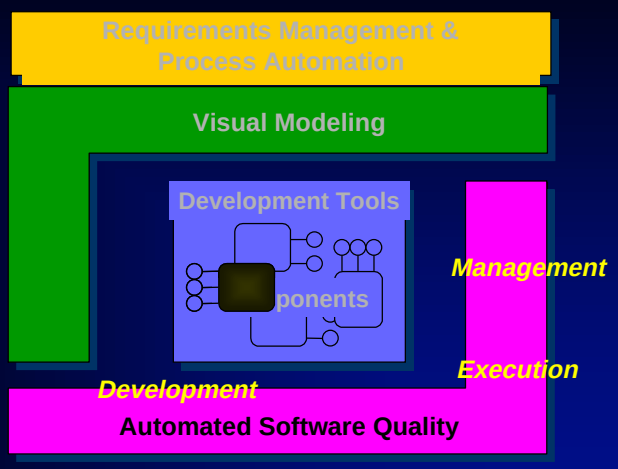
Imagine…

- Monitoring Progress of Test Procedure Execution (5000 Test Procedure Steps)
  - Use RM tool



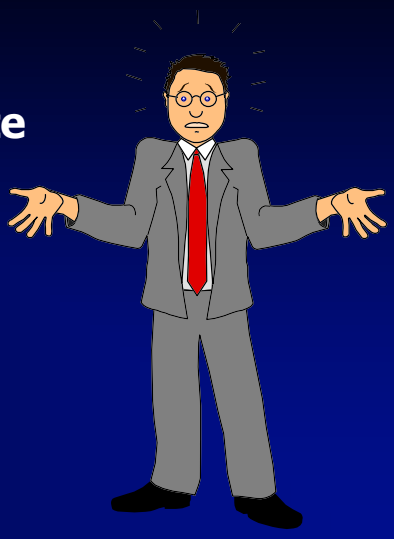
Key: Automated Testing

- **Test Studio**
- Client/server & web functional testing
- Leading ActiveX testing



GUI Capture/Playback Testing Tools

- **Why automate testing?**



GUI Capture/Playback Testing Tools

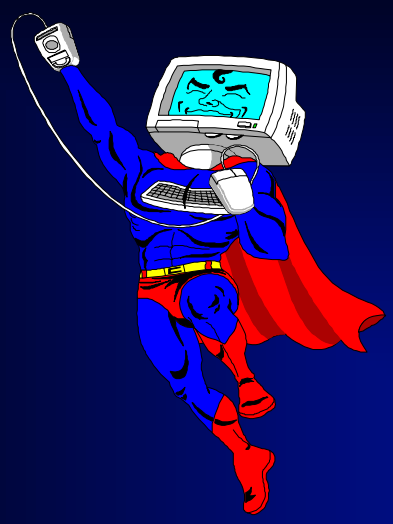
- Augment manual Testing
- Allows for Automation of repetitive, error-prone and time-consuming tasks
  - (i.e. adding a large number of accounts, replicating a transaction, verifying those transactions, and many more)
- Test Development/Execution Time Savings
- Reduce Regression Testing Time and Effort

More Tools....

Imagine....

- **100+ testers needing to hit the start key at the same time to perform stress, load or volume testing**

Load Testing Tools to the Rescue



Load Testing Tools

- Conduct Load Testing using Load Testing Tool (stress testing the application without requiring 100+ testers' time)
- Example: Performance Studio
- Reuse Scripts developed using Rational's TestStudio

Using automated tools to enhance OO-testing

Lessons Learned:

- Don't buy tools before system architecture is in place
- Tool training is very important
- Tool Mentors and Champions are very important
- Many tools lessons learned
  - e.g. Everyone needs to use same tool version
  - straight Capture/Playback waste of time
- Manage Expectations

When To Use Which Tool?

Peering into the Crystal Ball....

Using automated tools for OO-testing

Book "*Automated Software Testing*" will help sort out tool issues (www.autotestco.com)

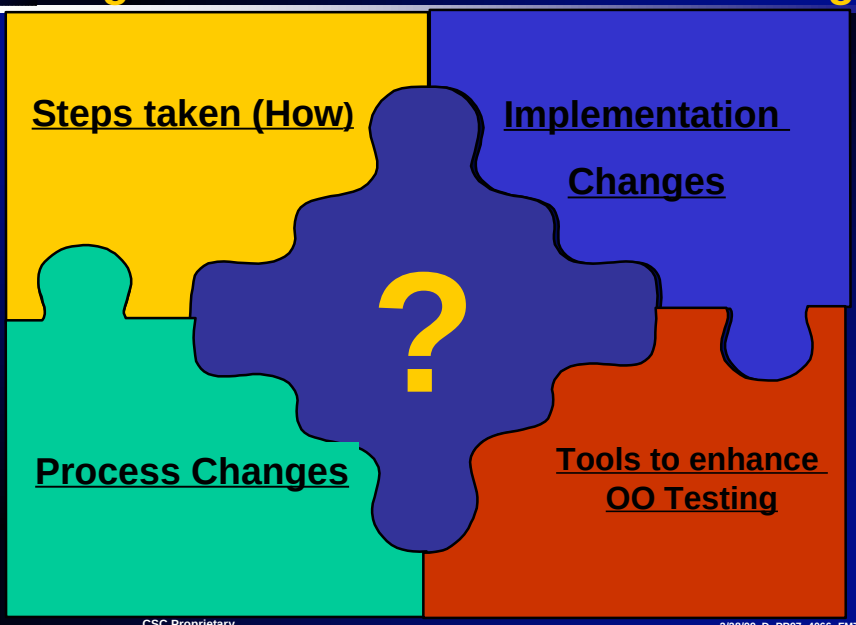
- What tools are available
- How to convince management to buy the tool
- How to evaluate tools
- How to incorporate tools into project and how to manage automated tools
- Lessons Learned

Moving From Conventional Testing to OO Testing

Summary:

- **Conduct Lifecycle Prototype**
  - **Testers need to be involved throughout system lifecycle**
  - **Incorporate Lessons Learned**
- **Document Standards and Procedures**
- **Use Mentors and other experienced people**
- **Training, training, training**
- **A lot of what we know still applies, but much has changed.**

Moving from Conventional to OO Testing



Moving from Conventional to OO Testing

Thank you!

Challenges of Testing Mass Market Software

By Elizabeth Hendrickson
Aveo, Inc

Testing mass market software presents unique challenges. We all work on "Internet time" now: releases are scheduled at an ever-accelerating pace. At the same time, applications are becoming more and more complex. The lack of clear, unambiguous specifications and requirements documents (if any exist at all) is pervasive. Our customers have a wide variety of configurations, making compatibility testing a daunting if not impossible task. And because the software is released to such a large audience, the cost of failure can be very high. Yet somehow companies release software that does well in the consumer marketplace.

Software companies address the challenges of mass market software in different ways. Some test groups fight with development and marketing to get good specifications and requirements documents. Other test groups find alternate ways to capture the same information from email threads, conversations, meeting notes, and hallway interviews. Some organizations outsource portions or all of the testing while other organizations try various hiring strategies. One tactic is to hire large numbers of relatively inexperienced testers to "pound" on the product, another is to hire primarily senior testers with programming experience who can write automated tests using tools or APIs.

Whatever other tactics an organization uses to overcome the challenges of testing mass market software, I believe that it's important that the test group understand the customer. Value, or quality, is in the eye of the beholder, in this case the customer. Unless we make an effort to understand who buys the software, for what purpose, and what their expectations are, it's impossible to know whether we're testing it right. As testers, we make the worst mistakes when we misunderstand the significance of an issue—we might report an issue as a minor annoyance only to find out later that it caused thousands of support calls—or when we misunderstand the users' expectations.

Fortunately we can learn from others successes and failures by sharing experiences and insights with other testers and test managers in similar situations. What do you think makes testing mass market software so difficult? What challenges are other mass market software test groups facing, and are they similar to your own? What solutions have others found that you haven't tried yet? Have others found solutions that failed in your environment—if so, how did the idea succeed in one environment when it failed in another? Do you have experiences or ideas that might help someone else facing challenges that you've faced before?

QW99 BOF Session

Medical and Safety Critical Application Testing

Agenda

- Introductions
- What do you want to cover?
  - Possible topics
- Helpful resources

Introductions

- Name
- Brief description of industry and your role

What do you want to cover?

- QA - the whole validation process
- Testing only
- Attitude required

Helpful Resources

- FDA Guidance for Software Validation
  - www.fda.gov/cdrh/odc/swareval.html
- Others

Medical and Safety Critical Application Testing

This is intended to be a brief overview of possible topics for consideration in the BOF session. It is not intended to be a comprehensive study of the topic.

Most standards governing the validation of safety critical applications require not only appropriate testing strategies but also a well-controlled process for the full development lifecycle from requirement specification through the maintenance and retirement phases. One reason for this is to prevent problems which are harder to detect as the development cycle proceeds. Another is that it is impossible to properly validate a safety critical application without an established software development lifecycle. The standards also strongly recommend an independent validation group – separate from the development team. Know what your standards require and plan the validation accordingly.

The lifecycle should include planning, execution, analysis and documentation of appropriate validation activities and tasks. All of this starts at the requirements specification stage calling for predetermined and documented software requirements that are reviewed and accepted before proceeding with design. During this phase the safety requirements should be identified and a hazard analysis performed. The hazard analysis identifies the system failures that could result from software failures. This allows the design process to consider the measures required to prevent the failures. Based on the hazard analysis the testing team should understand what the safety requirements are, design tests to cover them specifically, and watch for failures related to them in all tests that are executed.

Not all modules require the same level of validation effort. Take into consideration the risk of the device and the module's relationship to the risk. The size and complexity of the module may also play into the decision about how much validation is required. However, a change in any module should call for a regression test of the entire system.

The validation responsibilities include both static and dynamic testing activities. Reviews at every stage are necessary in order to catch all possible defects. The independent testing team should be involved in the static activities as well as the dynamic testing. This implies that they have technical and user level (functional) understanding of the system.

Testing levels should include module, integration (including regression tests as new modules are integrated), system, installation and maintenance test. Test to find defects and test to prove that all requirements have been satisfied. The testing strategy should cover both of these objectives. In order to prove that all requirements have been satisfied, some form of requirements tracing should be implemented in order to show the links from requirement through design, implementation and test. The validation process should also include both structural and functional testing.

With all of this said, we know that it is not impossible to get around the real intent of the standards. When working in circumstances where we sense that people are cutting corners and possibly compromising safety, it is important for us to remember our role – so that we can provide appropriate information and remain credible within the organization. The role of the tester is provide clear information in the form of plans, test results and reports so that those with decision-making responsibility can weigh the risks and make an appropriate determination of the consequences of the release of the software. It is not the testing team's responsibility to stop shipment! Though we play an important part, we are not the only ones responsible for the safety of those who rely on the system.

Other considerations:

- How to fulfill the requirements to validate OTS software which is part of the application or which is used as part of the validation process
- Performing usability (human factor) testing during the design phase to avoid user errors that may cause failures

Questions that may be discussed during the
OS and Embedded System Testing Techniques
Birds of a Feather Session.

1. Would you please briefly describe the system(s) that you test?
2. What are the most important types of problems that you try to find with your testing?
3. Do you use commercial and/or home-grown tools to aid your testing. if so, would you please briefly describe them?
4. What kinds of additional resources would you like in order to improve your testing?
5. What unique testing challenges does testing your software present?
6. Do you measure code coverage as part of your testing? If so, what obstacles did you overcome to make it work?
7. Do you do reliability testing? If so, how do you do it?

Larry Apfelbaum

Position paper for Testing Telecommunication Software BOF on Wednesday, May 26

Customers have grown to expect that the dial tone will be there, that their calls will be completed quickly, and their telephone bills will be accurate. Those famous words, "the software will be fixed in the next release", do not satisfy telecommunications customers. And if their current vendor or service provider can't offer them the quality level they expect, there are several other suppliers queuing up for their business who will.

The communications industry is experiencing unprecedented growth and change with the explosion of the Internet, Client/Server computing, and the Telecommunications Reform Act of 1996. Boundaries are being reset as communications service providers and other companies are entering new markets with revolutionary technology. This boom is fueled by businesses who are becoming increasingly dependent on communications technology as part of their "mission-critical" systems and networks, where failure just isn't an option. At the heart of these systems are sophisticated electronics which handle high-speed communications protocols, and thousands of lines of complex software which control switching, enhanced services, billing, database access- just to name a few. Both hardware and software must be thoroughly tested to ensure that these systems will operate error free and that they withstand the rigors of peak-loading conditions prior to actual deployment.

This session will explore the issues, trends and some of the techniques employed to be successful in the real practice of telecommunications testing.

- - - - - - - - - - - - - -

Larry Apfelbaum (800) 996-8778 x3555
Teradyne Software & Systems Test (603) 879-3555
44 Simon Street (603) 879-3075 FAX
Nashua, NH 03060 larry@sst.teradyne.com
http://www.teradyne.com/sst

Testing for Military and Government Software

Jon Hagar
Lockheed Martin Astronautics Company
Mail Stop H0512
P.O. Box 179
Denver, CO 80201
303-977-1625
fax 303-977-1472
jon.d.hagar@lmco.com

Question and items to consider are as follows:

1. Why are you here and what do you want to learn?
2. What have you done in you testing to address the following big issues:
    a. Y2k
    b. Security of information from hackers, warfare, corruption, etc.
    c. Formal vs. Informal testing, "time crunch" documentation, etc.
3. What problems do you have?
    a. One of kind
    b. Conversions
    c. Life, safety, and mission critical
4. Do you use Commercial off the shelf software (COTS) and should you be using COTS (and how, when, where, why, etc.)?
5. Should you be making decision about Legacy systems?
    a. Keep
    b. Replace
    c. Out source
    d. Buy-Make decisions
6. Is the CMM just a DOD tool? Who should keep and use the CMM? What are the holes in the CMM?
7. Government Reqs, are they good? Is IEEE Better? As the Air Force and other remove standards how do companies, customers, and users cope?
8. What does OO mean to you?
9. What do commercial practices mean?
10. How can we get lean and mean (better faster cheaper) but keep good quality?
11. Who is your customer and what makes them happy when you test?

Life as a New Test Manager

© 1999, Johanna Rothman

Congratulations! You've just been promoted to test manager. Now what do you do?

We all have our own ways of becoming successful test managers. Please attend this BOF to share what's worked for you or your manager. Some of the topics we'll discuss: drawing the line between project management and test management, how to get the rest of the organization to listen to you, how to know that you're managing the right things.

Areas of discussion

What do you do, what does the project manager?

- What are your responsibilities, how do they intersect with the project manager?
- What other matrix management issues do you have?

Influence and power, and how to use them

- How do you get others to listen to you?
- Do you have to be right all the time?
- Where is the power in your organization?

What are the right things to manage?

- What kinds of people have you hired/inherited? What bearing do their skills have on your work?
- Do you have to create vision and mission statements?
- What's your style of management?

CEM KANER, J.D., Ph.D.

Law Office of Cem Kaner **kaner@kaner.com**
P.O. Box 1200 **408-244-7000 (Voice)**
Santa Clara, CA 95052 **www.badsoftware.com** **408-244-2181 (Fax)**

STATUS REPORT ON

PROPOSED LAWS GOVERNING SOFTWARE QUALITY

This session will follow up my keynote talk (and status report on the law of software quality) on Thursday morning.

Thanks in no small part to opposition and criticism by the software development and testing community (ACM, IEEE, ICCA, sw-test-discuss forum), draft Uniform Commercial Code Article 2B is being viewed more skeptically by the legal community.

Article 2B was co-authored by the National Conference of Commissioners on Uniform State Laws (NCCUSL) and by the American Law Institute (ALI). The ALI passed a resolution last year requiring what it called fundamental revision of 2B, but the 2B drafting committee chose to not make those revisions.

In early April, 1999, Article 2B was pulled out of the Uniform Commercial Code. It is being renamed (probably to the Uniform Computer Information Transactions Act) and will be sponsored solely by NCCUSL. NCCUSL will vote on Article 2B / UCITA at its annual meeting in July, in Denver. If the bill passes, then it will be submitted to state legislatures this fall.

There's a lot of work to do on this. I am in this way over my head. Along with reviewing status, I will plead for help (as I have at previous Quality Week sessions).

==========

Article 2B / UCITA is an enormously complex (200+ page) proposed law that will become the main body of US law governing software quality and a significant source of law governing reverse engineering, troubleshooting, and maintaining software. It will probably govern most computer sales and many contracts involving "information" (such as contracts to write or to develop multimedia content).

This proposed law will virtually eliminate liability of vendors for software defects (even serious defects that were known by the vendor but not disclosed to the customer at the time of sale). It will also make it easier for software publishers to limit competition and to impose new restrictions on your use of the software.

For my latest short summary of objections to Article 2B / UCITA (I update this summary fairly frequently as I learn more), send me an email at kaner@kaner.com.

==========

Cem Kaner practices law, focusing on the law of software quality. He is the senior author of a new book on consumer protection, *Bad Software: What to Do When Software Fails.* See http://www.badsoftware.com. Kaner also consults on technical and software development management issues and teaches about software testing at UC Berkeley Extension, UC Santa Cruz Extension, and at several software companies. He founded and hosts the Los Altos Workshops on Software Testing. He is the senior author of *Testing Computer Software*. See http://www.kaner.com. Kaner holds a B.A. (Math, Philosophy), a J.D.(law degree), and a Ph.D. (Psychology) and is Certified in Quality Engineering by the American Society for Quality Control.

“Blue Collar Formal Methods” in Commercial Quality Assurance

Richard Denney
QA Manager
Landmark Graphics Corp.
Rdenney@lgc.com

There are many areas in commercial QA where Formal Methods (FM) could make useful contributions, yet there has not been a lot of focus by the FM community on commercial QA in a “real-world” setting. The use of FM by a software QA group in a commercial setting presents a different set of needs than might be encountered in the context of, say, FM as used by developers working on their own product in a commercial setting, or researchers working in FM [Denney et.al.] For example, QA groups often operate in a "MASH" like mode to support a much larger development group: there is no time for tidy, thorough operations on a single patient; rather surgeons are forced to triage and hastily perform rudimentary operations on a large number of incoming. For this type environment there needs to be increased focus on what I call “Blue Collar FM”, i.e. FM for the working folks. It’s about trying to identify that 20% of FM – as well as those 20% of applications of FM -- that gives developers & QA engineers 80% of the bang for the buck.

Example areas for applying Blue Collar FM are: test specification, project estimation metrics (e.g. function point like metrics based on formal specs), operational profiles and reliability engineering, and formal specification styles (e.g. algebraic and model-based) as a basis for natural language specification templates.

Another potentially fruitful area for the application of Blue Collar FM is in increasing the rigor of technical peer reviews. Formal technical peer reviews (e.g. walkthroughs & inspections) are popular in industry as a way to "test" a product early in the life-cycle before code is even written. Providing training on, and participating in, these peer reviews is “state of the practice” stuff for QA groups. The use of FM as an analysis tool of other peoples work has received some amount of discussion in the FM community, e.g. [Bowen and Hinchey]. Likewise in the QA community the potential leverage of formal methods in the technical review process has been recognized [Britcher], [Dyer 91], [Dyer 92], [Jackson and Hoffman], [Parnass and Weiss], [Van Emden].

But using FM to increase the rigor of analysis in the context of technical peer reviews adds some real world challenges:

- Reviewers are often times domain experts, rather than software engineers, with little or no experience with requirements specification, much less formal methods
- Reviewers are in some cases developers from other projects brought in to get an “outside view”, and hence are more or less looking at the documents for the first time; a practice encouraged by the technical peer review literature
- Preparation times by reviewers prior to the meeting is usually only 1 to 2 hours

Analysis under such constraints is a real challenge, but this is "state of the practice QA" and systematic analysis tools are needed for these types of reviews.

What would Blue Collar FM as applied to technical peer reviews look like? Perhaps the design of better inspection checklists that can be used by reviewers not trained in formal methods. If a reviewer doesn't understand what a good specification is, how will they know a bad one when they see it? The structure of formal specification methods like the model-based and algebraic can contribute to knowing what information a good natural language specification should contain without necessarily delving into all the mathematics.

The author conducts a style of inspection that borrows from the depositions of [Votta] and the active design reviews of [Parnass and Weiss]. They are "depositions" in that the inspection minimally involves two people: the interviewer and the interviewee. They are "active" in that the interviewer, who must be versed in formal methods and has pre-reviewed the functional spec, walks the interviewee (usually a domain expert) through a light-weight modeling session of the functional specification (or some part thereof) under review as a way to generate questions that are then posed to the interviewee; this puts the interviewee in "active" review mode as opposed to "passive". The advantages of this approach are:

- One gains independent (from the person who wrote the functional spec) review from a domain expert or experts
- One gains a more rigorous review with a minimal investment in formal methods training
- The cost efficiency of depositions (see [Votta] for details) offsets what some might complain is the added expense of using formal methods
- In-depth pre-meeting preparation time is incurred by the interviewer, then leveraged across the many interviews, with minimal preparation time by the interviewees.

As has been said [Kac], models aren't so much to explain and predict, as to pose sharp questions, and that's exactly the role that formal methods based modeling plays in this type of inspection.

In conclusion, I believe commercial QA groups as a consumer of FM is a fruitful yet largely unexplored area, with potential benefits for both QA and FM. For QA groups to leverage off FM however, a body of Blue Collar FM techniques needs to be identified which scales-up to the hectic MASH style operation of QA groups of the real world.

References

Bowen, Jonathan P. and Michael G. Hinchey, "Ten Commandments of Formal Methods", Computer, Vol. 28, No. 4, April 1995

Britcher, Robert N., "Using Inspections to Investigate Program Correctness," Computer, Vol. 21, No. 11, Nov. 1988, pp. 38-44.

Denney, Richard, Dick Kemmerer, Mark Ardis (sitting in for Nancy Leveson), Alberto Savoia, Joint ISSTA / FMSP Panel: “What State-of-the-Art is not State-of-the-Practice”, Proceedings of the 1996 Int’l Symposium on Software Testing and Analysis (ISSTA), ACM Press, 1996

Dyer, Michael, "Verification Based Inspection," Proc. 25th Hawaii Int'l Conf. System Sciences, Vol. 2, IEEE CS Press, Los Alamitos, Calif., 1991, pp. 418-427.

Dyer, Michael, The Cleanroom Approach to Quality Software Development, John Wiley & Sons, New York, N.Y., 1992, pp. 96-99.

Jackson, Ann and Daniel Hoffman, "Inspecting Module Interface Specifications," Software Testing, Verification, & Reliability, Vol. 4, No. 2, June 1994, pp. 101-117.

Kac, Marc, “Some Mathematical Models in Science”, Science, 166, No 3906 695, 1969

Parnas, David L. and David M. Weiss, "Active Design Reviews: Principles and Practices," J. Systems and Software, No. 7, Elsevier, New York, N.Y., 1987, pp. 259-265.

Van Emden, Maarten H., "Structured Inspections of Code," Software Testing, Verification and Reliability, Vol. 2, No. 3, Sept. 1992, pp. 133-153.

Votta, Lawrence G. Jr., "Does Every Inspection Need a Meeting?," Proc. 1st ACM SIGSOFT Symp Foundations of Software Eng., ACM Press, New York, N.Y., 1993. Published in ACM Software Eng. Notes, Vol. 18, No. 5, Dec. 1993, pp. 107-114.

Rodney Wilson

"Care and Feeding of a Testing Career"

Abstract

Questions and discussion for this Birds Of A Feather will center around the following three topic areas.

1. What are core competencies for software test and quality engineering professionals?
    - o Product technology and domain knowledge (databases, operating systems, networking, languages, business or scientific applications, etc.)
    - o Verification and Validation (methods, tools, metrics, processes, training, etc.)
    - o Certified Software Quality Engineer (CQE) body of knowledge
    - o Certified Quality Manager (CQM) body of knowledge
    - o Interpersonal skills (cross-functional teamwork, leadership, change management, conflict resolution, positive attitude to creative destruction, oral and written communication, etc.)
    - o Positive attitude toward creative destruction.
    - o Interest and ability to successfully obtain broad product and complete solution knowledge. As knowledgeable as the companies best application and field support engineers.
    - o Desire to be the customer advocate. Success is customer satisfaction and delight.
    - o Ability to successfully multi-task several assignments.
    - o There are many others. What are some more examples?

2. What are potential career destinations for software test and quality engineers?
    - o Sr. Test Engineer, Test Project lead, Test Architect,
    - o Quality Manager, Director or VP
    - o Technical Support Manager, Director or VP
    - o Engineering Manager, Director or VP
    - o Chief Information Officer
    - o Contractor, Consultant or Company Owner
    - o Test execution to test development
    - o Testing to Quality Assurance and Software Engineering Process Group
    - o There are many others. What are some more examples?

3. How can you keep from falling into a career rut?
    - o Stay accountable, don't get caught in the victim loop. Have clear deliverables. Strive to always ensure the Test and Quality function is viewed as an important corporate asset (e.g., testware library, metrics, defect prevention and containment). Respect is earned. Image is nothing, contribution is everything.
    - o Strive for balance (work, home and play). It's easy to burnout in the high-tech industry.
    - o Continue to learn and explore (your responsible for managing your own career, not your manager or company HR representative).
    - o Develop a personal vision/mission statement. Define your short- and long-term career goals.
    - o Identify the important factors in a new career opportunity (management team, marketplace and technology, money and equity, location, company culture, your responsibilities and opportunities for advancement, etc.).
    - o There are many others. What are some more examples?

Increasing productivity through small integrated development/QA teams

Mark Johnson, OrCAD
mark.johnson@orcad.com

A Quote:

In Peopleware[1], DeMarco and Lister reported an 11 to 1 difference between best and worst participants in their coding war games. But the average difference between people from same organization is only 21%. They found that "Two people from the same organization tend to perform alike. That means the best performers are clustering in some organizations while the worst performers are clustering in others."

An Observation:
The most enjoyable, motivating, and productive work experiences I have ever had have been with small teams where we really shared a common mind-set about what we were doing and had the skills and confidence to deliver what we really believed in.

The Quest:
How do we create one of the 10x productivity organizations? The title of this session and the abstract expose some of my opinions on what can make for a productive team.

What are your experiences?
This is an interactive session where we want to hear what you have experienced and learned!

1. Have you ever been part of a team that you considered to be at the 10x-productivity level?
2. What factors do you believe allowed the team to operate at a this productivity level?
   a) Were there special factors about the skills or mix of people?
   b) Were there special factors about the way the team was organized, the team's goals and objectives, or the ways the team members interacted?
   c) Were there other special factors about the team?
3. Were there environmental factors in the organization that were critical to the team's productivity?
   a) Were there special factors about the structure between teams, and communication in the organization?
   b) Were there special factors in the organization's 'systems' that encouraged 10x productivity?
   c) Were there other special factors about the organization?
4. Do you believe that creating 10x productivity teams is a repeatable process that is a matter of following a specific set of steps? Or does it require 'special' people? Or both?
5. Do you feel there is a relationship between high productivity and work being 'fun?'

---

1. DeMarco, Tom and Timothy Lister, Peopleware, Productive Projects and Teams, Dorset House Publishing Co. New York, 1987.

Mark D. Anderson

Load testing practice today is hampered by several factors:

- immature tools
- the scarcity of those skilled in the art
- the lack of a recognized position in the product lifecycle

This condition has arisen because in the past, large multi-user systems have usually had sizeable budgets, and so they fostered a high-end market for load testing. Furthermore, there just haven't been that many such systems. Both of those factors have been eliminated by the web.

The sorry state of the art has also been exacerbated by the proximity of load testing to two other markets: GUI functional test tools (from Mercury, SQA, Segue, etc.), and canned benchmarks (TPC-C, specweb96, etc.). This tends to obfuscate the technique and even the goals of load testing, and undermines the appropriateness of the tools available. It is dubious enough whether the tools in either of those adjacent markets are valuable even in their core areas, without the added problem of extending them beyond their supposed area of competence.

Indeed, a useful segmentation of test tools is the following:

	Errors	**Performance**
single-user	Purify, functional test tools, coverage	Quantify, gprof
multi-user	load testing tools, performance monitoring	load testing tools, performance monitoring

Load testing can and should be done long before a system has a stable GUI.

Attend this BOF session to commiserate on this lamentable state of affairs, or to participate in a ritual stoning of me for my incendiary and heretical comments.

A useful link farm of QA tools, including load testing, may be found at http://www.charm.net/~dmg/qatest/index.html.

Running the Nightlies

- What is a Nightly ?
  - A nightly is the process in which a piece of software is built daily and a suite of sanity tests run against it to verify that it is at least as sane as it was yesterday.
- Benefits
  - Minimizes integration risks
  - Reduces bug isolation time
  - Immediate feedback to developers
- Drawbacks
  - Overhead of running and maintaining nightlies
- Requirements
  - Automated build and test process
  - Sane tests
  - Must happen daily

Now You See It.™

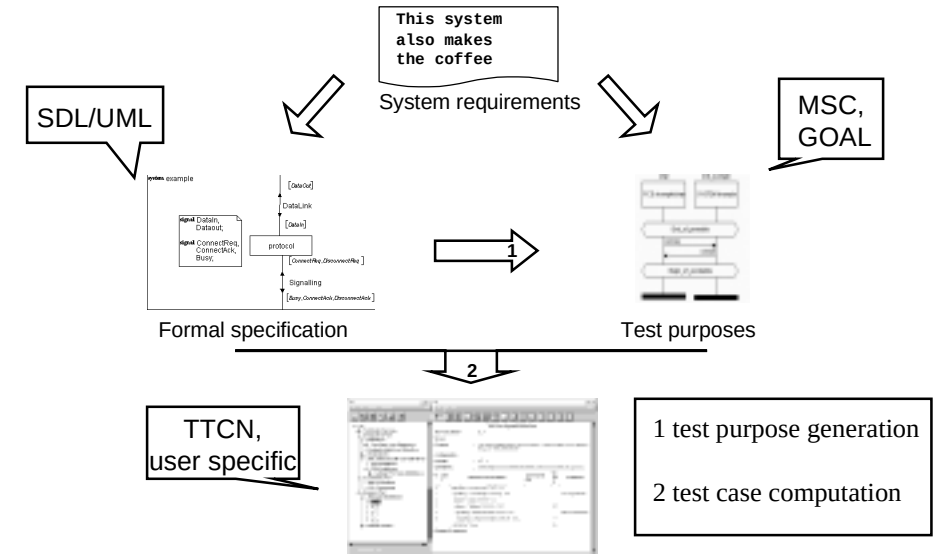
Automated test generation from SDL/UML specification

Alain Kerbrat, Iulian Ober

Automated test generation

- **Benefits**
  - **correct, reproducible tests**
  - **cost effective**
- **Requirements**
  - **unambiguous, tractable description**
  - **clear definition of the test semantics**

TestComposer principles



1 test purpose generation

2 test case computation

TestComposer basics

- **ISO/IEC 9646: « conformance testing »**
- **based on two research prototypes :**
  - **Tveda from France -Telecom/CNET**
    pragmatic test purposes generation
  - **TGV from Irisa/Verimag**
    efficient test cases generation

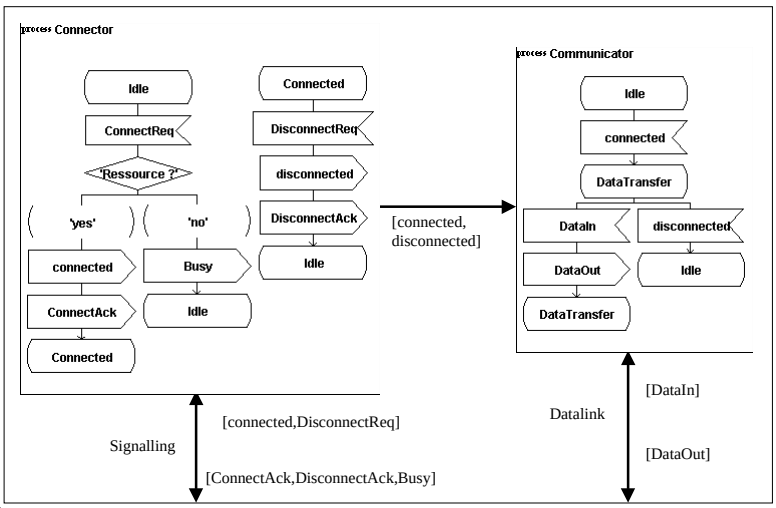
TestComposer objects

- **SDL specification :** ***expected system's behaviour***
  - ***must be validated***
- **Test environment specification**
- ***Test purpose* : *functional view of the test***

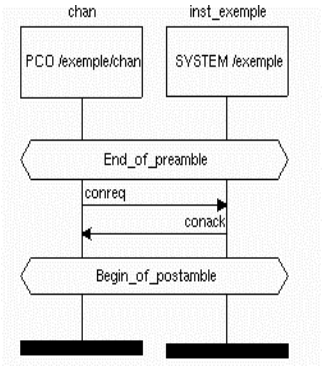
  "What should I test ?"
- ***Test case : operational view of the test***

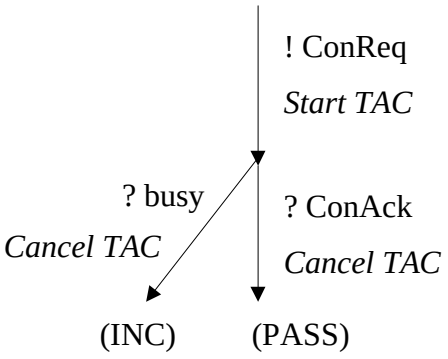
  "How should I test ?"

SDL specification



From test purpose to test case





Main test sequence

A tree with all possible outcomes

Test suite production

- **Test cases database**
  - **build in a multi-session mode**
  - **Application Programming Interface**
- **TTCN production module**
  - **ISO/IEC 9646-3 compliant**
  - **SDL type definitions translated in ASN.1**

TestComposer prototype usages

- **France Telecom/CNET :** ***relevancy for the end user***

  Experimentation on several low level protocols

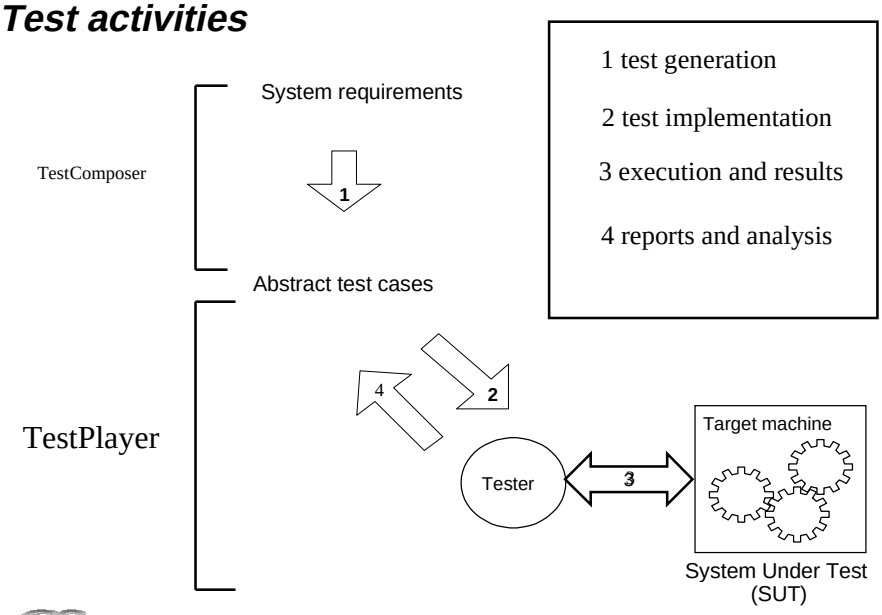
- **CS-Telecom :** ***down to the test execution***

  Two ISDN complementary services (MCID, Call Waiting)

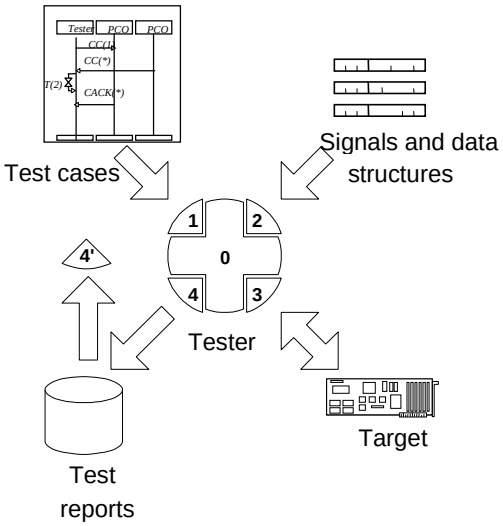
- **Danet, Sema Group :** ***integrated TTCN environment***

  Edition, generation, cosimulation and execution

Test activities



TestPlayer principles



1 test cases interpreter

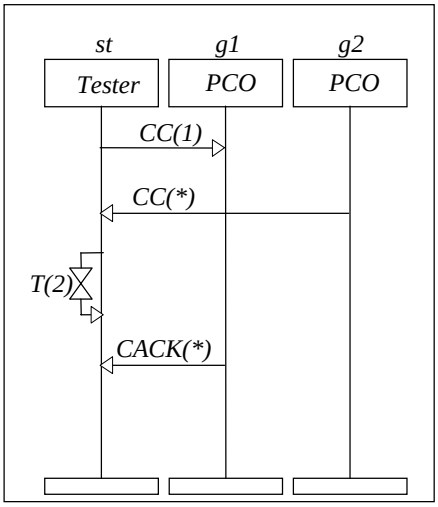
2 declarations handler

3 communication, encoding/decoding

4 results storage

4' results retrieval

MSC and Test Description Language example




```
Output CC with 1 via g1
TimerSet U 5
Expect CC  with star via g2
TimerSet T 2
TimerWait T
Expect CACK  with star via g1
Verdict PASS
```

TestComposer and TestPlayer summary

- **standardized formal notations (SDL, UML, MSC)**
  unambiguous and tractable descriptions
- **interactive or automated test purposes generation**
  covers both functional and structural coverage concerns
- ***sound* and *efficient* test cases generation**
  no ambiguous interpretation of the test results
- **APIs for user specific test language and environment**
  seamless integration within customer testbed

Software Quality

Copyright McCabe & Associates 1996

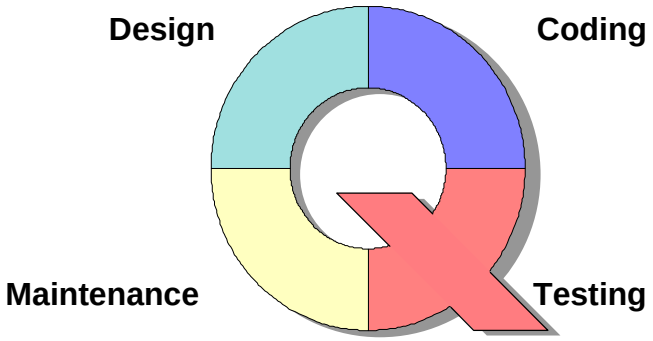
Why does Quality Suffer?

- Shortened Timescales
- Reduced Budgets
- Changes in Personnel
- Changes in User Requirements
- Enhancements
- Bug Fixing
- Ineffective Testing

"Quality does not happen by mistake!"

Copyright McCabe & Associates 1996

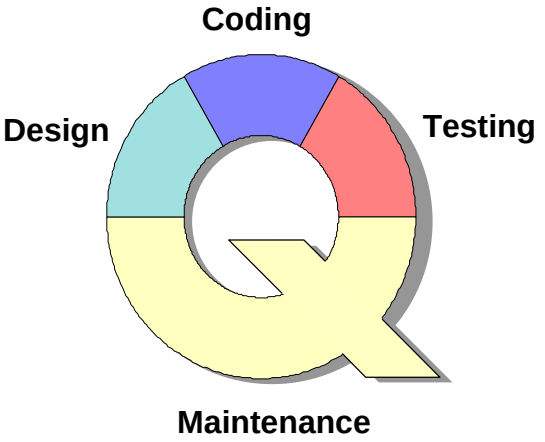
Software Development



3.2m-2t-2s

Copyright McCabe & Associates 1996

Software Development Effort ?

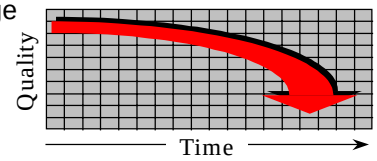


3.3m-3t-3s

Copyright McCabe & Associates 1996

Improving Quality

- Measurement
  - Determine Quality
  - Prevent Degradation
  - Enable Informed Decision Making
- Comprehension
  - Improve Productivity
  - Determine Cost of Change
  - Analyze Risk
- Testing
  - Ensure Quality
  - Improve Test Effectiveness



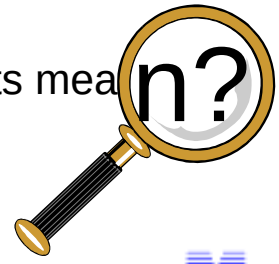
4.1s

Copyright McCabe & Associates 1996

Measurement

Measurement is useful but...

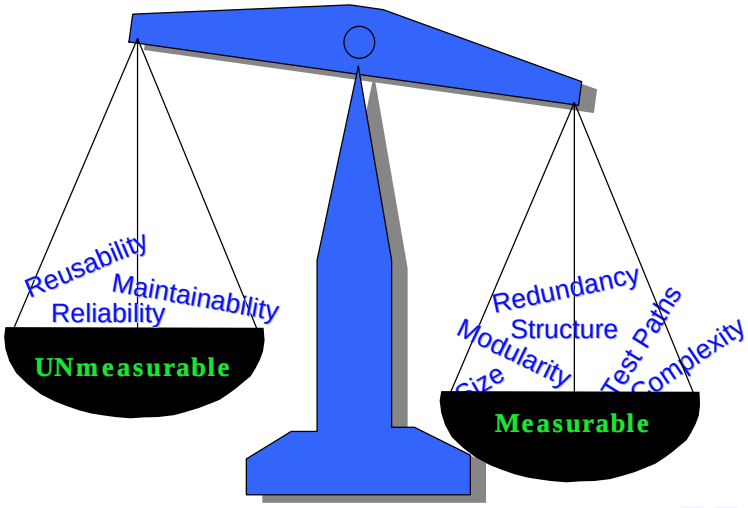
- What do we want to measure?
- How do we measure?
- What do the measurements mean?



5.1o-1s

Copyright McCabe & Associates 1996

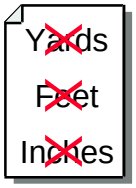
What to Measure?



5.2o-2s

Copyright McCabe & Associates 1996

What is a Metric ?



- Any unit of measurement
  - e.g. Cm, Litre, Ohm, Second, Colour
- Describes an attribute of an entity
- Software metrics indirectly describe:
  - Maintainability
  - Reliability
  - Costs
  - Timescales

5.3s

Copyright McCabe & Associates 1996

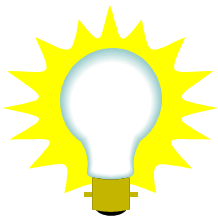
Good Metrics

Good Metrics must:

- Be Intuitive
- Be Objective
- Be Language Independent

Good Metrics should:

- Directly Correlate with Errors
- Directly Correlate with Testing Effort
- Be Automated
- Be Easy !

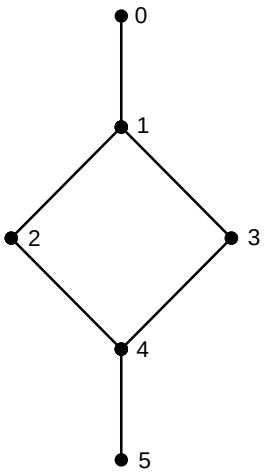


5.4s

Copyright McCabe & Associates 1996

Flowgraphs

```
    function_test(y)
0   {
        x=3;
1       if ( y < 4 )
2           x=sin(y);
        else
3           x=cos(y);
4       x=x*x;
5   }
```

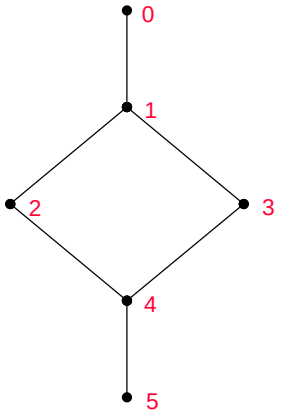


5.3o-5s

Copyright McCabe & Associates 1996

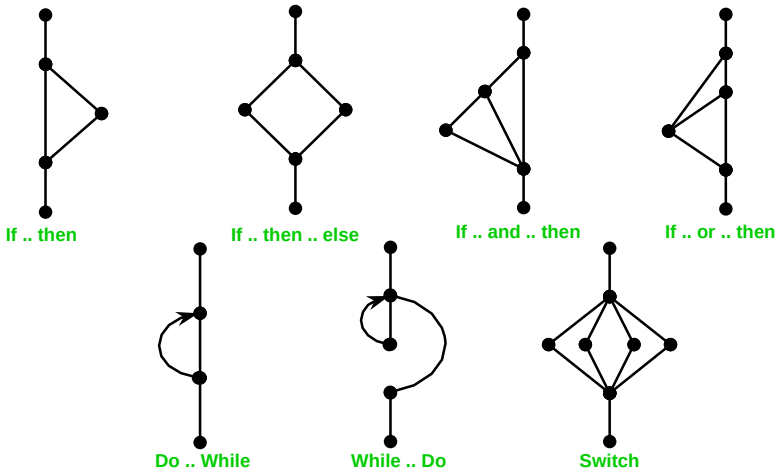
Flowgraphs

```
0 SUB-PROC.
     DISPLAY "SUB-PROC".
     MOVE A TO B.
1    IF SOME-NUMBER > OTHER-NUMBER
2        DISPLAY "IF - True"
     ELSE
3        MOVE A TO B.
4    PERFORM DISPLAY-PROC.
5 EXIT.
```



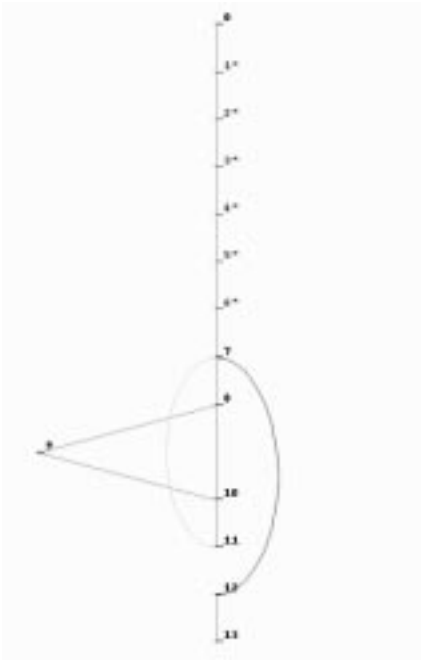
Copyright McCabe & Associates 1996

Flowgraph Notation

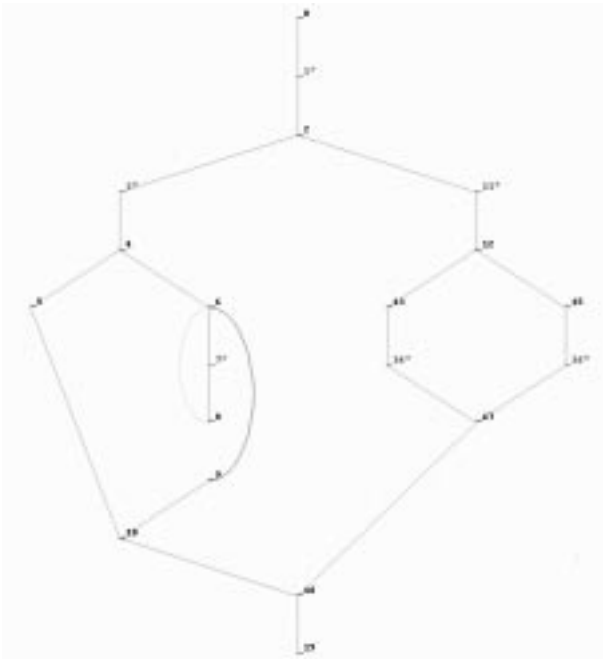


5.6s

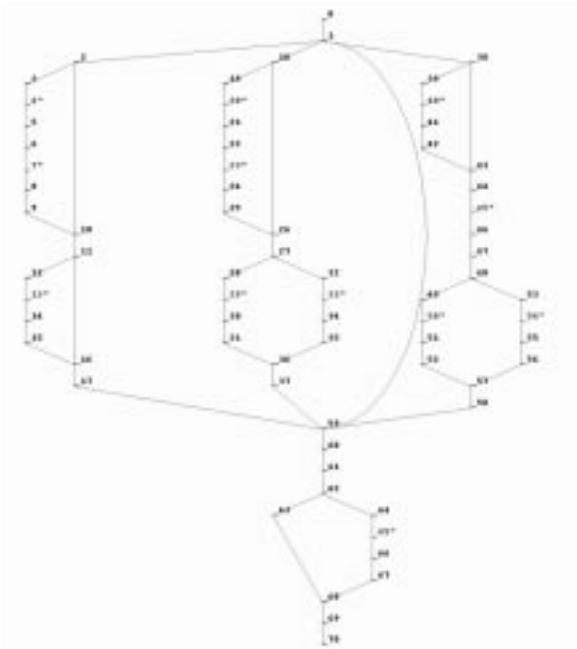
Copyright McCabe & Associates 1996



Copyright McCabe & Associates 1996



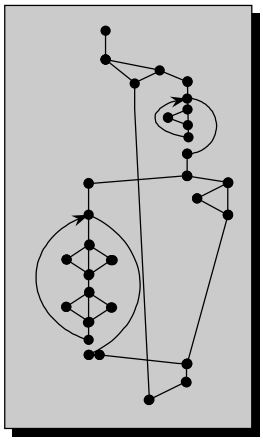
Copyright McCabe & Associates 1996



Copyright McCabe & Associates 1996

Copyright McCabe & Associates 1996

Control Flowgraph



- Flow Graphs Give Intuitive View of Complexity
- Requirement for Absolute Measure

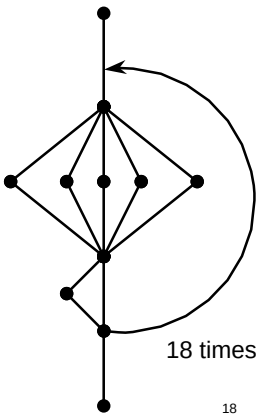
"Intellectual Content" = Complexity

5.4o-7s

Copyright McCabe & Associates 1996

Complete Testing

Why Measure Complexity?



Statistical Paths = 10^{18}

If each test requires 1 nanosecond then the total testing time is :

$$T = \frac{10^{18}}{10^{9} \times 3600 \times 24 \times 365}$$

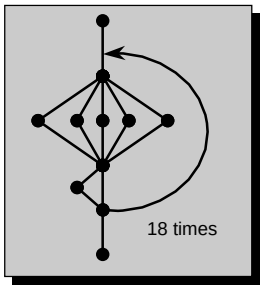
T = **317 Years**

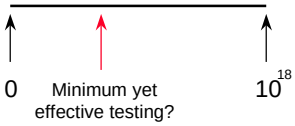
5.9s

Copyright McCabe & Associates 1996

Testing Effort

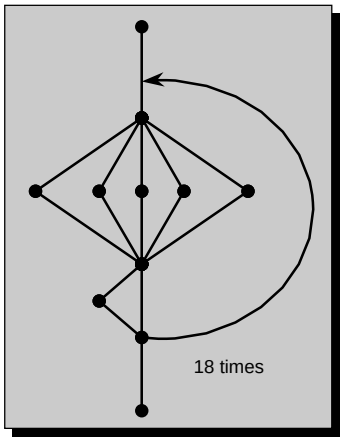
- Not Possible to Test All Statistical Paths
- Requirement to Determine what is Minimum Yet Effective Testing





Copyright McCabe & Associates 1996

Testing

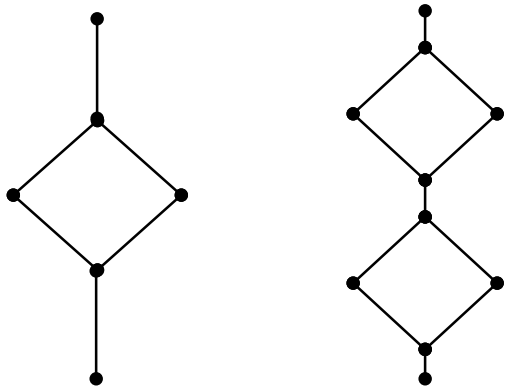


How many tests are needed to exercise every line of code at least once?

Code Coverage Testing Technique

Copyright McCabe & Associates 1996

Code Coverage



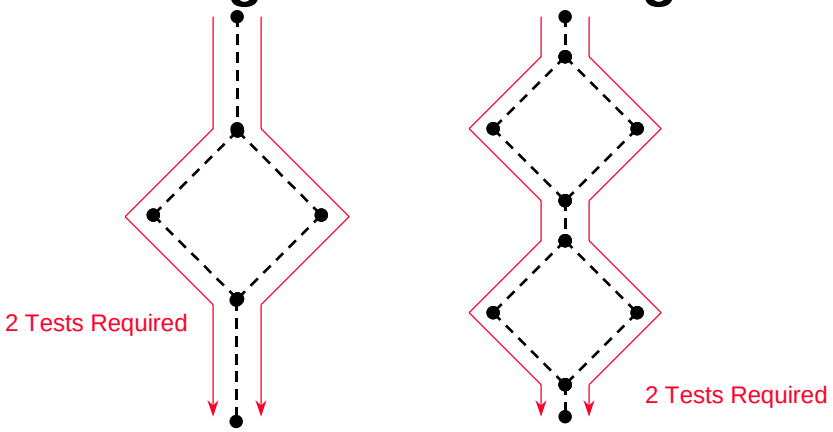
Example ‘A’ Example ‘B’

Which function is more complex?

5.12s

Copyright McCabe & Associates 1996

Using Code Coverage



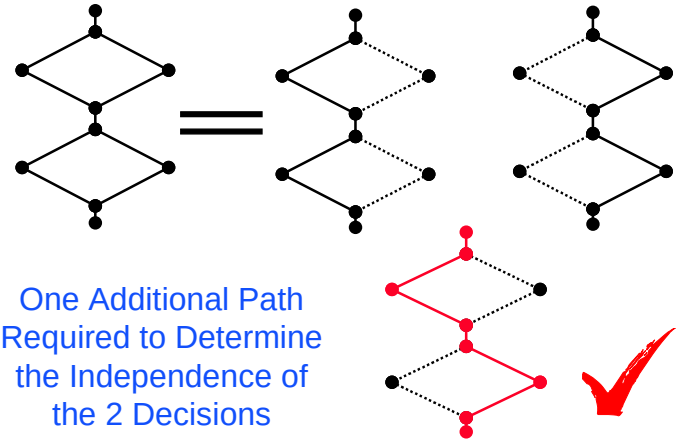
Example ‘A’ Example ‘B’

Code Coverage is <u>not</u> proportional to complexity

5.13s

Copyright McCabe & Associates 1996

McCabe's Cyclomatic Complexity Metric



One Additional Path Required to Determine the Independence of the 2 Decisions

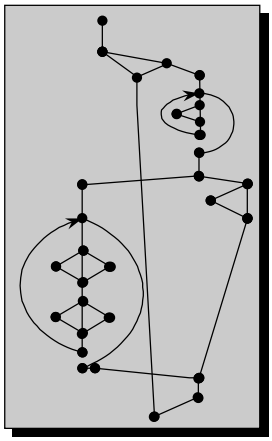
McCabe's cyclomatic complexity ***v(G)***

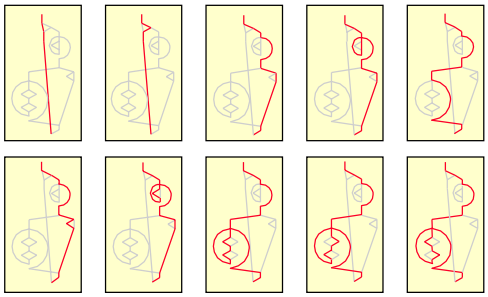
Number of linearly independent paths

5.14s

Copyright McCabe & Associates 1996

McCabe's Cyclomatic Complexity





Cyclomatic Complexity v(G) = 10

10 tests will:

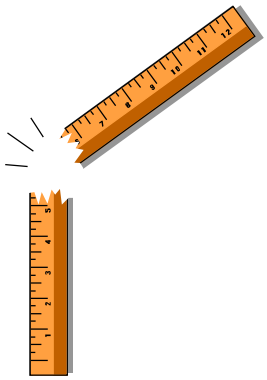
- Ensure Code Coverage
- Test independence of decisions

5.15s

Copyright McCabe & Associates 1996

Complexity

- Complexity Correlates with:
  - ✓ Probability of Errors
  - ✓ Reliability
  - ✓ Testing
  - ✓ Comprehension
- Complexity does not Correlate with:
  - ✗ Maintenance Effort

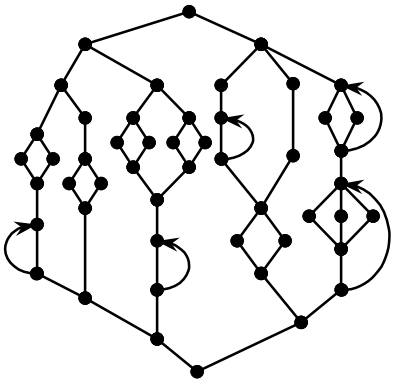


5.6o-16s

Copyright McCabe & Associates 1996

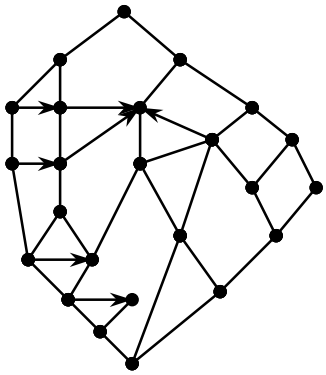
Visualization

v(G) = 20



Function A

v(G) = 18



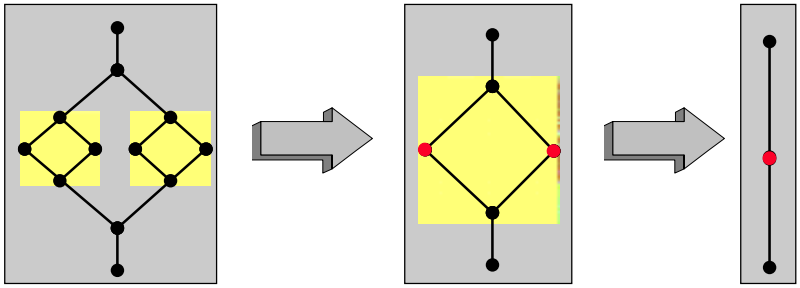
Function B

Which function do you prefer?

5.17s

Copyright McCabe & Associates 1996

Flow graph reduction



Cyclomatic
Complexity = 4

Essential
Complexity = 1

McCabe's Essential Complexity ***ev(G)***

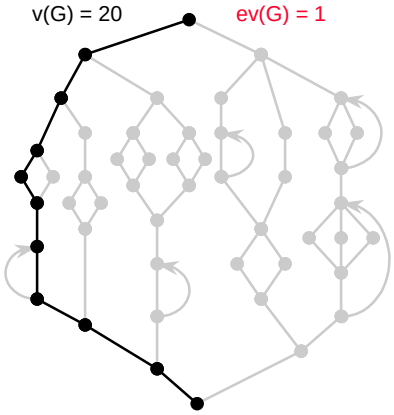
Remove structured elements and recalculate the complexity

5.7o-18s

Copyright McCabe & Associates 1996

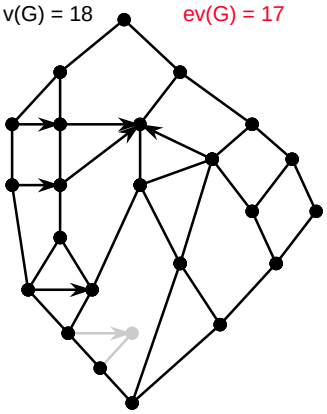
Structure

v(G) = 20 ev(G) = 1



Function A

v(G) = 18 ev(G) = 17



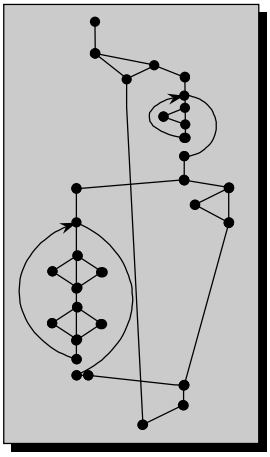
Function B

5.20s

Copyright McCabe & Associates 1996

McCabe Unit Level Metrics

- Complexity v(G)
  - Comprehensibility
  - Probability of Errors
  - Reliability
  - Testing Effort
- Essential Complexity ev(G)
  - Structure
  - Maintainability
  - Re-engineering Effort

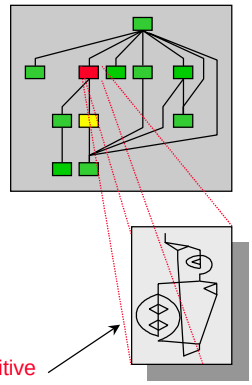


5.9o-22s

Copyright McCabe & Associates 1996

McCabe Visualization

- System Level Views
  - Complete System Architecture
  - Overlay of Quality Metrics
  - Interaction of Modules
- Module Level Views
  - Flowgraphs & Code Listings
  - Detailed Metrics Analysis
  - Summary Reports



Context Sensitive
Measurement and Visualization

6.4o-4s

Copyright McCabe & Associates 1996

Q-Labs acts as change facilitators, providing industry with state-of-the-art solutions designed to keep our customers steps ahead in software engineering

Experience Using Statistical Usage Testing at the FAA

Ara Kouchakdjian
Ara.Kouchakdjian@q-labs.com

Q-Labs Group

Sweden, Germany, Norway, U.S.A.

© Q-Labs 1999
All Rights Reserved.

Agenda

- The Project
- Statistical Usage Testing
- Timeline
- Technology Acquisition
- Usage Model Development
- Test Generation and Execution
- Lessons Learned
- Broader Insights
- Next Steps

© Q-Labs 1999
All Rights Reserved.

Q-Labs

The Project

- Support System for Air Traffic Controllers
  - Information system to support controllers in making longer terms decisions
  - Not a control system
  - >>300 KSLOC
- FAA had responsibility for test/acceptance and subsequent fielding of this as a part of a
- Decided to use a usage=based testing
  - Focus on field reliability
  - Typical 'acceptance' test felt to be insufficient

© Q-Labs 1999
All Rights Reserved.

Q-Labs

Statistical Usage Testing

- Test the software the way users use it
  - Develop test without code knowledge
  - Provide maximum field reliability gain
  - Provide user focus for quality
- Testing as a statistical experiment
  - Statistical inference
  - Testing decisions based on objective data

© Q-Labs 1999
All Rights Reserved.

Q-Labs

Why SUT?

- To maximize field reliability
- To maximize the potential for objective
  - Measurement for test planning
  - Measurement for test execution
  - Measurement for product certification
  - Measurement for process assessment
- To remove human bias from testing
- To test consistently with field usage
- To make software quality assessment more

© Q-Labs 1999
All Rights Reserved.

Q-Labs

SUT Process

Test Planning	Define Test Goals / Stratify Define Boundary and Stimuli Define a Use Allocate Test Budget to Goals Allocate Test Approaches to Goals
Usage Model Development	Build Model Structure Assign Probabilities Analyze Model
Generate Tests	Generate Scripts Process into Test Cases Study Coverage Compute Expected Results
Test Execution	Run Tests Determine Success/Failure Record Pass/Fail
Certification	Compute Quality Measures Compute Stoppage Criteria Make Release Decisions

© Q-Labs 1999
All Rights Reserved.

Q-Labs

Timeline

- Initial Training: Summer 1997
- Initial Modeling: Summer 1997-Summer 1998
- Testing: Spring 1998-Present
- Additional Training: Spring 1998
- Extensions/Modifications to Models: Summer
- Additional Systems to Model: Summer 1998 -

© Q-Labs 1999
All Rights Reserved.

Q-Labs

Technology Acquisition

- Initial training - 3-4 day course to FAA staff
- ToolSET_*Certify*® delivered to FAA
- Q-Labs did initial modeling
- FAA focused on testing
- FAA took on modeling responsibilities with minor amounts of coaching

© Q-Labs 1999
All Rights Reserved.

Q-Labs

Usage Model Development

- 7 major models (state transition diagrams)
  - Corresponded to major menu pull-downs
  - Composed from 50+ smaller models
  - Major models had >350 states
- Models were at a mouse click level of granularity (arcs values)
- Represented all possible sequences of inputs
- Customer defined probabilities on arcs exiting each state

© Q-Labs 1999
All Rights Reserved.

Q-Labs

Test Generation and Execution

- Approximately 20 tests automatically generated per model
  - All executed, failures recorded
  - Looked at arc and state coverage based on
- Generated and executed additional tests for models with low coverage
- Sufficiency of test based on state and arc
- Failures recorded by type, severity and where in test they occurred

© Q-Labs 1999
All Rights Reserved.

Lessons Learned (1)

- Test cases appeared sufficient
  - Very few other tests run
  - Approach works for this type of software
- Test quite efficiently executed
  - Tests on one screen
  - Target software on another
  - Software easy for testers to directly validate
- Elimination of paper
  - Tests viewed/executed electronically
  - Infinite numbers generated at any time
  - No need for paper test instructions

© Q-Labs 1999
All Rights Reserved.

Lessons Learned (2)

- Testing effective
  - Non-modeled components were more reliable prior to FAA testing (used SUT on the
  - Modeled components more reliable after FAA test
- Found large numbers of requirements and
  - Result of usage models being precise descriptions
  - Allowed questions to be concrete

© Q-Labs 1999
All Rights Reserved.

Broader Insights

- Model based approach eliminated human
  - Tested what one would not have otherwise tested
- Can test 'forever'
  - Always generate 'fresh' tests
- Change in testing paradigm
  - Go away from waiting until end to think about test and creating detailed procedures
  - Shift to initial investment which pays dividends
-

© Q-Labs 1999
All Rights Reserved.

Next Steps

- Have modeled another subsystem
- Will model remainder of system
- Convinced that this is the way to test
  - Cannot imagine writing test procedures
  - Continue to use model generated tests to this day
- Gathering up quantitative results (currently doing installations)

© Q-Labs 1999
All Rights Reserved.

Q-Labs acts as change facilitators, providing industry with state-of-the-art solutions designed to keep our customers steps ahead in software engineering

toolSET_Certify®

A CASE tool for Statistical Usage Testing

Peggy Gilbert

Peggy.Gilbert@q-labs.com

(423) 450-5151 ext. 237

Q-Labs Group
Sweden, Germany, Norway, U.S.A.

© Q-Labs 1998-1999
All rights reserved.

Agenda

- Test Challenges
- Testing As Sampling
- Statistical Usage Testing (SUT)
- Process and Techniques
- Project Experiences

© Q-Labs 1998-1999
All Rights Reserved.

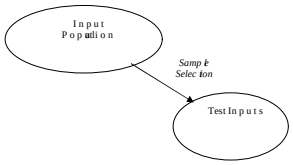
Actual Test Challenges

- Not testing the 'right' thing
- Seem to test the same things
- Quality goes down in each phase
- Cannot measure product 'reliability'

- Low field reliability
- Too many field reports

© Q-Labs 1998-1999
All Rights Reserved.

The Reality of the Sample



- Exhaustive testing is not practical due to variable inputs and infinite languages
- ALL SOFTWARE TESTING IS SAMPLING
- How to choose the best sample?

© Q-Labs 1998-1999
All Rights Reserved.

The Implications of Sampling

- Cannot ensure the software contains no
- Cannot ensure the software will never fail
- Two kinds of uses/tests
  - Those executed during test
  - Those left for the user to execute
- How to predict what the user should expect

© Q-Labs 1998-1999
All Rights Reserved.

Statistical Usage Testing

- Test the software the way users use it
  - Develop test without code knowledge
  - Provide maximum field reliability gain
  - Provide user focus for quality
- Testing as a statistical experiment
  - Statistical inference
  - Testing decisions based on objective data

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

Why SUT?

- To maximize field reliability
- To maximize the potential for objective
  - Measurement for test planning
  - Measurement for test execution
  - Measurement for product certification
  - Measurement for process assessment
- To remove human bias from testing
- To test consistently with field usage
- To make software quality assessment more

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

'Inside the Project' Goals

- Our primary goals are:
  - quantitative decision that software (intensive) product is fit for its intended use
  - quantitative decision to stop testing
    - fitness has been statistically demonstrated
    - fitness cannot be demonstrated within budget of time and money
  - traceability to testing requirements
  - optimal use of testing schedule and resources
- toolSET_*Certify*® supports this approach to software quality certification.

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

Phases of Statistical Testing

- Usage Model Development
- Usage Model Analysis and Test Planning
- Construction and Execution of Crafted Test
- Automatic Generation of Random Test
- Product Certification Testing
- toolSET_*Certify*® is a test tool that supports all

© Q-Labs 1998-1999
All Rights Reserved.

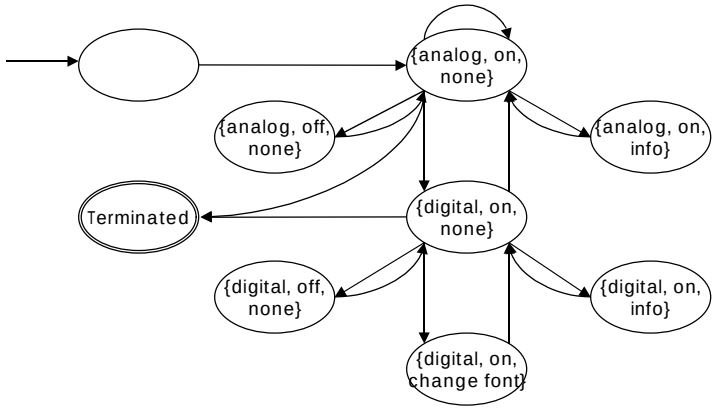
Q-Labs

Operational Profiles

- Divide usage to achieve sampling efficiency
  - identify classes of users (beginner, expert, home,
  - identify environments of use (normal, emergency light load, heavy load)
  - identify management test objectives (safety critical
- Create one or more usage models

© Q-Labs 1998-1999
All Rights Reserved.

Usage Model Structure



© Q-Labs 1998-1999
All Rights Reserved.

Determine Transition Probabilities

- Estimate probabilities on arcs based on:
  - existing data or user surveys
  - analysis of prototype or prior system
  - management objectives

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

Leveraging the Usage Model

- The usage model is a precise specification of how the system can be used
  - At a given level of abstraction
- Usage model facilitates discussion of
  - Precise artifact for discussion
  - Common forum
- Usage model development often results in requirements validation/clarification

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

Usage Model Analysis and Validation

- Analyze the model using results computed *Certify*®
  - Structural error check
  - Model statistics:
    - identify heavy use states and paths
    - expectation of testing required to experience
    - expectation of the average test case length
    - measure usage path complexity of the model
- Validate the estimates with respect to:
  - known usage information, common sense, feasible
  - testing requirements

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

Test Planning

- Plan testing based on:
  - time and resources available
  - crafted testing that must be done
  - random testing to support reliability goals
- Set stopping criteria for failures that might be
  - stop testing if failure renders further testing futile
  - stop testing when reliability or coverage goals are
  - stop testing when goals cannot be reached within

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

Crafted Test Cases

- Using the model as a reference guide, non-random, crafted test cases should be selected and executed for the following
  - to assure understanding is sufficient to address all
  - to test sub-paths of special concern (contractual requirement, management requirement, customer request, experimental interest) not expected in the amount of random testing planned
  - to achieve model coverage, if practical
    - toolSET_*Certify*® generates the set of test sequences that will cover all arcs with the minimal number of transitions.

© Q-Labs 1998-1999
All Rights Reserved.

Automatic Test Script Generation

- Generate a set of random test scripts using *Certify*®
  - provide human readable test scripts
  - interact with automated test tools
  - generate test cases
  - analyze test cases
  - assess model coverage
  - forecast possible test plan outcomes

© Q-Labs 1998-1999
All Rights Reserved.

Quality Measures

- Analyze product and process quality data *Certify*®:
  - Reliability
    - bounded by 95% and 99% confidence intervals
  - Mean Time To Failure

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

Stopping Criteria

- Monitor testing sequence-by-sequence and relative to the stopping criteria established in the test plan and data computed by *Certify*®:
  - Structural coverage
    - percent of states traversed
    - percent of arcs traversed
  - Sample sufficiency
    - Discrimination measure
    - Distance measure

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

Project Experiences

- Primarily Embedded Systems
- Domains include Telecommunications, Defense, Aerospace, Medical Devices, CASE Tools, Mass Storage Devices

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

Alcatel

- Evaluate feasibility of Usage Specification and Functional Test
- Modeled three ISDN switch features
- Results
  - 30% more effective (defects per test case)
  - Large number of requirements issues noted and
  - Usage modeling more interesting than creating test
  - Shift in thought process for testing
  - Reduced time to market due to earlier defect

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

IBM-SSD Tucson

- Mass Storage Solutions - Devices, controllers
- Using approach since 1992
- Many products released, usage and
- Tests correctness and performance
- High degree of automation - full process + >>90% of tests evaluated automatically

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

Summary

- Decide operational profile
- Build model(s) structure of nodes and arcs*
- Determine probabilities that characterize the environment of use*
- Validate model(s)*
- Develop test plan
- Generate test cases*
- Execute test cases
- Monitor quality measures and stopping

  - *Steps supported directly by toolSET_*Certify*®

© Q-Labs 1998-1999
All Rights Reserved.

Q-Labs

What's Next!

- New version of tool available 4th quarter
  - Model composition
  - Enhanced data handling / testing support
  - Java
- Additional research in
  - Reliability
  - Efficient model construction
  - Leveraging test results

© Q-Labs 1998-1999
All Rights Reserved.

Push the Limits of Your Applications and Deploy With Confidence!

A Successful Deployment Story

CYRANO

1

Test Objective

• Test the applications scalability & performance.

• The current number or users was 250 and the goal was to add 750 users to reach 1000 concurrent users.

• CYRANO Impact was first used to simulate the 250 users, and re-create the usual application load, then additional users were added.

• Four tests were run to check system performance and capacity prior to roll-out.

•Each test simulating the activity of 250, 500, 750 and 1000 concurrent on-line users respectively.

2

CYRANO

Test Configuration

Item	Description
Application	Major ERP application
Database	Oracle 7.3
Server Type	Digital Alpha 4100
Server OS	Digital UNIX
CPU	One Alpha 533 MHz
Memory	512 MB
Disk IO Channel Number/Type and Speed	2 Fast and Wide
Number of Disks	7
Disk capacity and performance characteristics	Five 4 GB disks Two 8 GB disks
LAN Connection	One FDDI 100 MB

CYRANO

3

Testing Software

CYRANO Impact Enterprise Edition

- Optimize user response times by tuning SQL statements
- Predict system behavior under heavy load
- Appropriately size and configure the server

CYRANO

4

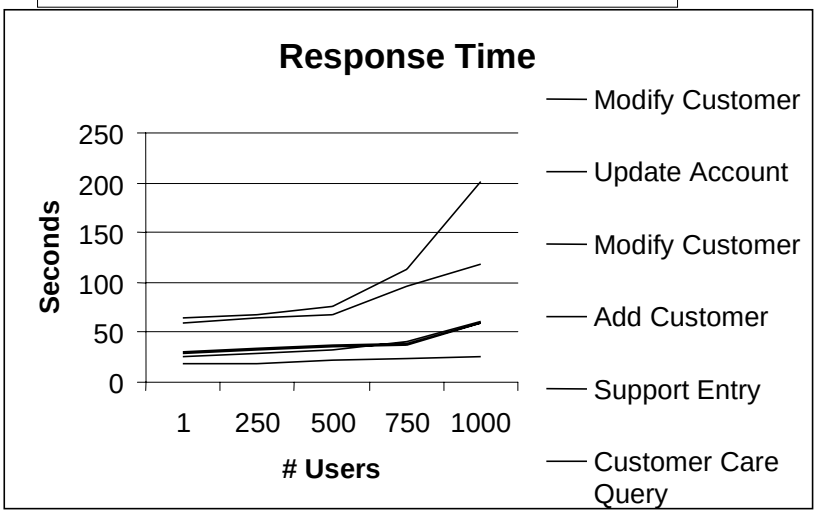
Test Results

Business Transaction Per Day per User	90th Percentile Response Time per number of user						
	#	%	1	250	500	750	1000
Modify Customer	50	23%	59	64	200	1650	2700
Update Account	31	14%	25	29	93	341	992
Modify Customer	29	13%	29	32	87	145	783
Add Customer	58	27%	65	68	464	2610	7714
Support Entry	21	10%	18	19	63	84	147
Customer Care Query	28	13%	30	34	84	140	700
TOTAL	**217**	**100%**	**227**	**496**	**991**	**4970**	**13036**
Time Lost/Day/Emp.			**3'47''**	**8' 16"**	**16' 31"**	**1h22'**	**3h37'**

CYRANO

5

Response Time Without Tuning



CYRANO

6

Tuning Information

	Number of Users 250	500	750	1000
CPU Utilization	36.80%	66.97%	97.13%	95.71%
IO/s Per Second	64	89	161	144
Disk Utilization	38%	54%	97%	86%
IO Response Time (S)	0.614	1.158	28.66	6.580

CYRANO

7

First Results, using CYRANO Tuning Capabilities

- **CPU on the test server is 100% utilized when running between 500 and 750 users.**

- **Utilization of one disk is 100% when running between 500 and 750 users. This disk is the only disk that is doing any significant amounts of physical IO for ORACLE.**

- **One transaction (add Customer) is generating a table scan of a large table every 32 seconds in the 25-user test and every 8 seconds in the 100-user test. This generates physical IO, consumes additional CPU, and forces other data blocks which could have been reused out of buffer cache (more more physical IO).**

CYRANO

8

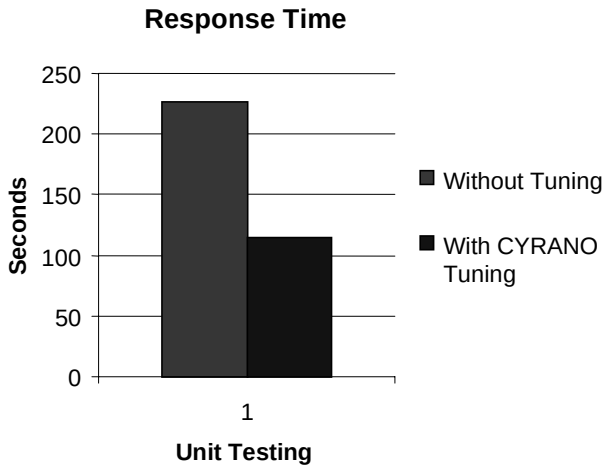
Unit Testing Results

	No Tuning 1 user	Unit Tuning 1 user	Improvement
Modify Customer	59	**12**	**+ 80%**
Update Account	25	**13**	**+ 48%**
Modify Customer	29	29	0%
Add Customer	65	**11**	**+ 83%**
Support Entry	18	18	0%
Cust. Care Query	30	30	0%

CYRANO

9

Unit Tuning With CYRANO Impact Enterprise Edition

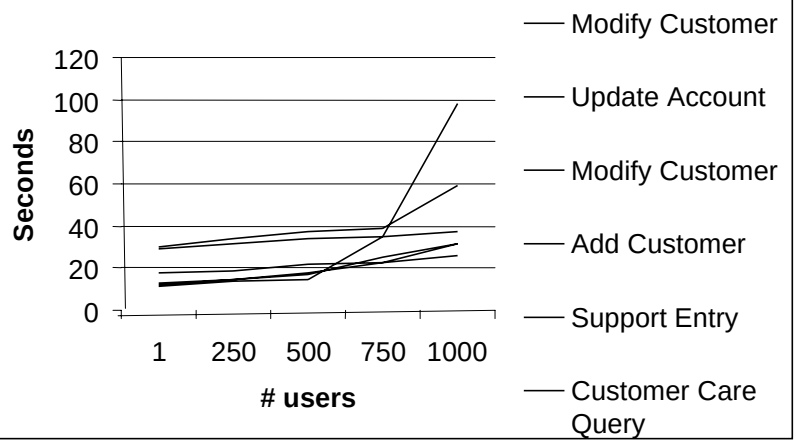


Global Response Time Divided by 2

CYRANO

10

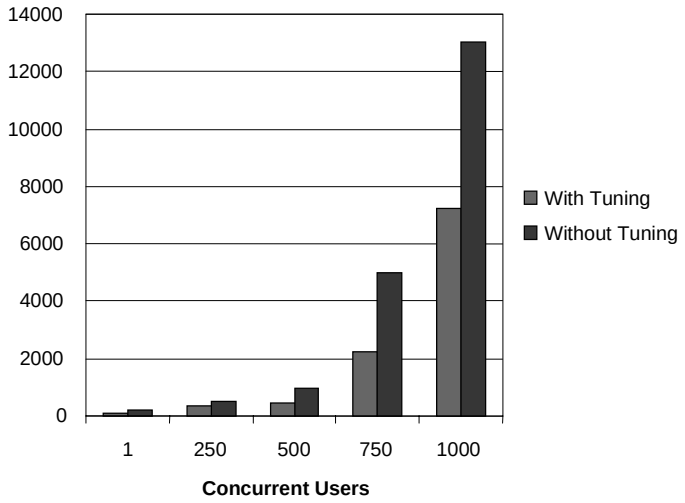
Response Time With Tuning



CYRANO

11

Response Time Comparison



CYRANO

12

General Results

Capacity Planning
CYRANO Impact used as a standard load test tool determined that 500 users was the maximum concurrent on-line users the system could handle before it crashed.

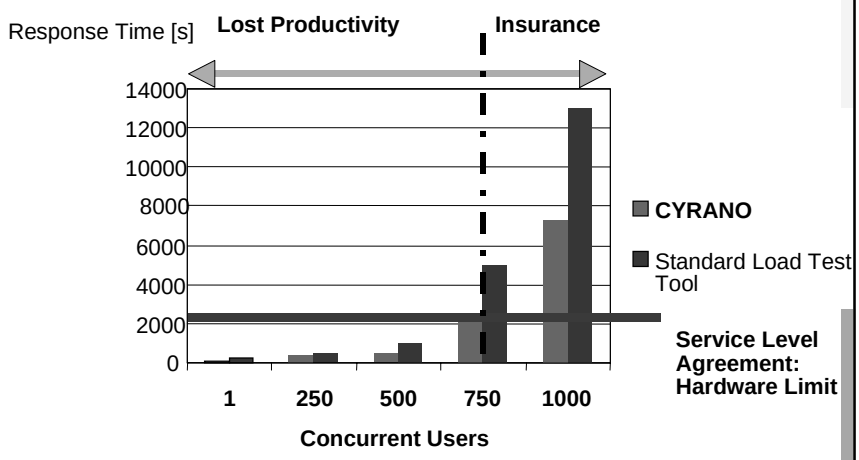
Optimization
With the tuning capability of CYRANO Impact, this number has been increased to 750 concurrent on-line user with the same hardware platform.

Without adding new users, under a 250 user load, the application response time has been improved by 23% giving an immediate return on investment even before deployment. The application performance has been increased by 44% under a 1000 concurrent user load.

13

CYRANO

Capacity Planning Diagram



14

CYRANO

Customer's Choice

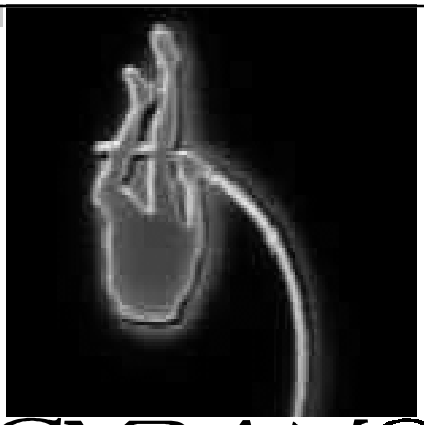
Decided to deploy the first 750 users within the next 4 months. Then buy more hardware to deploy with increased application functionality.

Decided to systematically use CYRANO Impact Enterprise Edition for their next deployment & development efforts.

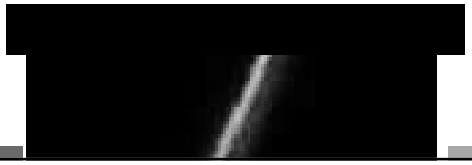
- **Test component Reusability**
- **Ease of Use**
- **Repeatable test results**
- **Increased performance**

CYRANO

15



CYRANO



16

Shari Turney's Technical Presentation:

Title: Push the Limits of Your Applications and Deploy with Confidence

Abstract: A description of how CYRANO solutions helped a worldwide travel agency to properly size their ERP implementation and reduce their costs. By optimally tuning their system and testing the entire application, this CYRANO customer increased their ROI, productivity and quality of service. After 16 months of performance issues, CYRANO solutions solved the problem within 4 weeks, allowing the customer to have a smooth deployment.

Model-based performance analysis of an EDP/ERP-oriented wide area network

Eugenio Cervetto

Performance Research, S.r.l.
Via Jervis, 22
10015 Ivrea, Italy
+39 0125 45 698

ecervetto@predicta.it

ABSTRACT

In this paper the operational phases of a model-based performance analysis activity performed on the EDP/ERP WAN deployed by FIAT Auto, S.p.a. in Argentina are described.

Keywords

Performance Modeling, Performance Prediction, Wide-Area Network, EDP, ERP.

1. INTRODUCTION

The Problem is to optimize the performance of a large network composed by several processing nodes and geographic links. The network is devoted to provide remote processing and data to financial, commercial and production environments operating according to an integrated corporate strategy. The computational nodes consist of multiprocessor, tightly coupled processing units. Inside each node, one or more servers are devoted to handle the applications and the database management, or are configured as redundant resources to support high-availability requirements. Servers are connected by a Local Area Network, which also connects the user community to the application and database servers. Commercially available routers are deployed to support communication between nodes.

2. MODELING

On board of each modeled node several application environments were defined, hence implying a typical multi-class scenario. Application environments originate both local requests addressed to processing units belonging to the same LAN and remote requests addressed to processing units allocated on different nodes, hence competing for system resources with the local activities.

2.1 Modeling Approach

The whole model configuration can be considered as a "puzzle" in which the visible pattern is originated by a replication of archetypal objects such as processors, memories, I/O ports, physical links, protocols, routers, O.S. kernels, access methods, data bases, etc… The first modeling step consists in the achievement of a *SIMPLE* archetypal model for each of the above components.

2.1.1 Model Validation

The above archetypal models are made available as component-level model libraries. All library models already passed through a validation step to check whether the predicted figures for each of them were consistent with the related "real world" performance behaviors.

Model library items that are used in this system model are:

- IBM RS6000 servers (models J40 and J50) including the PowerPC processors, multi-processor board, memory and I/O components.
- SCSI RAID 5 controller
- IBM SSA controller
- SCSI disk subsystems
- LAN adapters
- Ethernet & Fast Ethernet links
- IBM HACMP
- IBM AIX O.S.
- CISCO 3620 and 3640 routers
- FDDI, radio and VSAT links
- Oracle database.
- Triton by BAAN.

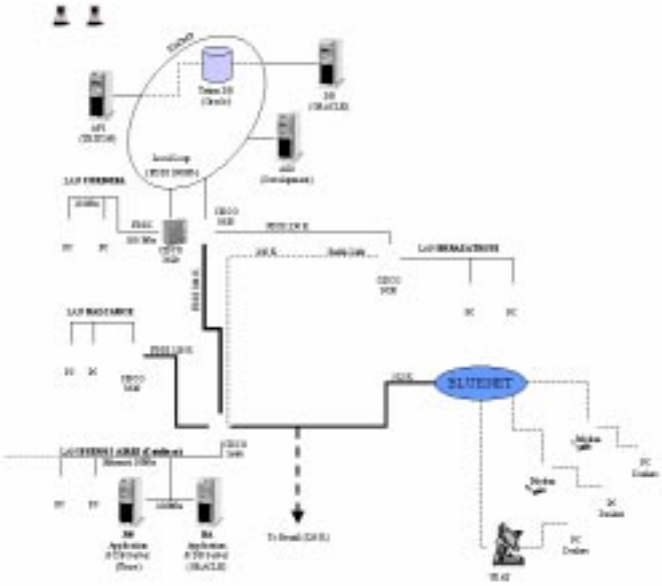


Fig. 1 The Network Architecture under analysis

2.1.2 System Modeling

The previous steps were devoted to provide libraries describing the components required for the model definition. The next one is addressed to build the system-level model. A large network model is an aggregation of occurrences of many archetypal objects generated by properly setting for each of them the global configuration variables used to parameterize the object. The concept of global configuration variable is general: all architectural and implementation options can be represented in this way. Relevant parameters such as pool sizes, buffer sizes, policies, strategies, disciplines and so on can be assigned at configuration time without modifying the model libraries.

2.1.3 Advantages

This method supports a structured approach to the problem: while modeling a large network, many of the archetypal models can be instanced more than once. This results in advantages in the model readability, re-usability and maintainability. A relevant aspect is represented by the reduction of the model code, hence reflecting in the time/cost paradigm of the modeling activity. Using this approach, the above described WAN was modeled with a 2m/m effort.

3. THE TOOL

Following several tests, it was agreed with the customer that a simulation tool was not suitable to guarantee the achievement of *system-level* results. The instantiation process underlying this approach requires more sophisticated algorithms. After the model has been generated, a visiting process of the defined components becomes necessary to collect system-level resource demand profiles. Considering the complexity of the real system, this visiting process can become long and somewhat cumbersome. To overcome this problem, a performance modeling and prediction tool grounded on symbolic execution principles has been selected. The main advantage of this technology proved to be its efficiency: the visiting process related to the whole system model was accomplished with a ten-minute run on a standard Pentium™ PC. The tool is *PREDICTA 2.1*, developed by Performance Research (http:/www.predicta.it or http:/www.pr-usa.com). It provides a solution allowing the model designer to build up a system-level model from a set of model libraries describing market-standard and Commercial-Off-The-Shelf (COTS) components. All components but the characterization of the application-level user classes were taken from available model libraries.

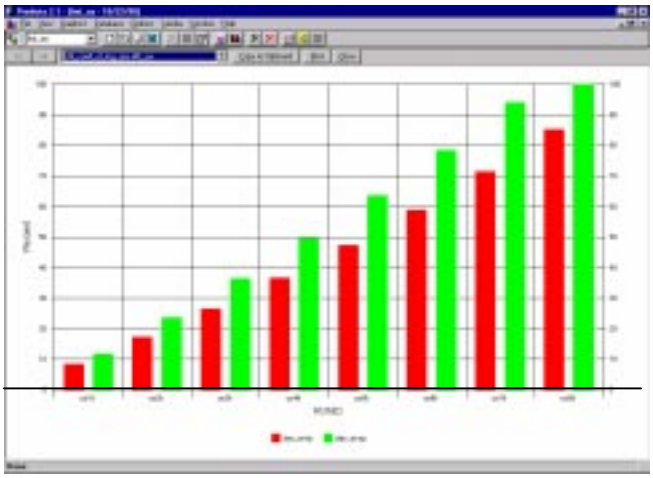
3.1 The Computing Process

After the symbolic execution process has been accomplished, the tool produces the model queuing network script by inspecting the collection of the resource demand profiles. This phase is devoted to build up a hierarchic model suitable to be solved producing useful performance parameters such as arbitration times, semaphore wait times and coefficients of utilization. To achieve this, the queuing network script is compiled and linked with a queuing network analysis library and the resulting task is executed.

3.2 The Results

The performance figures produced by *PREDICTA* individuate bottlenecks in the modeled system. Inside a hierarchic model, the starvation of an innermost component can imply that some of the outermost ones may appear as bottlenecks. The figures produced by *PREDICTA* have been conceived to individuate the primary bottleneck. A failure in the bottleneck analysis leading to a corrective action addressed to enhance the performance of a secondary bottleneck can slow down the system despite of the related costs.

Fig. 2 "What-if" output charts.



4. THE PROBLEM SOLVING ACTIVITY

A performance prediction activity addressed to a large network requires peculiar considerations: as a primary point we have the widespread use of market standard components. It may appear that a performance analyst, even when the existence of a problem inside a COTS component has been demonstrated, has no chance of recommending any modification and/or improvement in its internal design. The outcomes of this activity prove that much more can be done.

4.1 How to Use the Results

The outcomes of this performance analysis activity provide a relevant support to the customer in checking its proposed solutions. In this case, the designer's opportunities can be summarized as follows:

- the redefinition of some aspects of the application architecture;
- the utilization of different HW and/or SW components;
- a tuning action on the configuration (e.g. the adoption of larger memories, faster disks, more CPUs, etc.)

Among the above options, the first one proved to be the most fruitful. It is clear that, when a corrective action is necessary, the modeling effort to describe the action and predict its results plays a fundamental role. Using *PREDICTA*, the analysis of design, tuning and upgrade alternative can easily executed by means of a low-cost re-configuration activity.

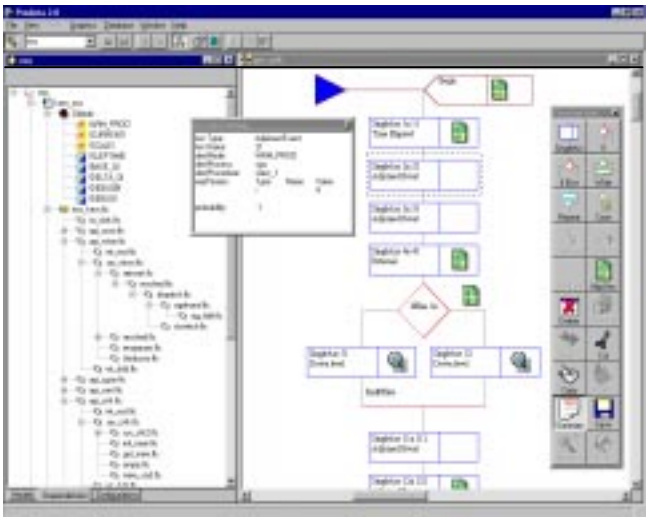


Fig. 3 *PREDICTA 2.1* user interface.

4.2 Improving Response Times

A large network should give answer to the expectations of different classes of users, and, in many circumstances, the application environment requires to access data allocated in remote nodes. In this case the response time, as experimented by the users, is composed of contributions originated by the residence time on several system components. The time required to get service by I/O ports, routers, physical links, protocols, transportation layers and interrupt handlers represents a non-negligible fraction of the whole response time. The results, in terms of response time reduction, obtained by enhancing a single component are in general poor. The only effective option is to enhance the whole chain of components involved in the operations. This action always represents an expensive approach to the problem. A large network represents a constrained environment in which progresses are slow and expensive and where common sense-driven actions can be dangerous.

4.2.1 Tuning Actions

The user perception of tuning results was in this case poor. We can distinguish among the following possible actions:

- tuning actions related to the kernel configuration;
- tuning actions related to the database management;
- tuning actions related to the buffering conditions.

About the first point it appears that the kernel tuning, as proposed by the system manufacturer, is generally behaving in a satisfactorily way and just in peculiar situations a tuning action could result in advantages. In this specific case, the tuning actions related to the database management (e.g. reducing the number of tables requiring a scansion, replacing the selection sequence with a more efficient one, compacting data, etc…) prove to be more effective in achieving advantages. About the buffering condition tuning actions, in most cases expectations are not confirmed by the field results: in particular a very large buffer pool size, when used to tune databases, implies the consumption of more computing resources.

4.2.2 Multiprocessor Environments

Attention has to be paid to the symmetric, multiprocessor environment. This analysis demonstrates that processing units with a large number of processors can behave in two main ways. When the real time issues are low, the kernel overhead and the related spin loop synchronization between processes do not represent an unacceptable loss. When the system is involved in severe real time issues, the number of concurrent processing increases and the synchronization activities represent a significant loss. It appears that almost no advantage can be achieved by upgrading servers from six to eight processor. The output data from the modeling activity demonstrates that the increase in computing power achieved by means of this upgrade equates the increase of the synchronization losses originated by the larger number of processors competing for the inner kernel services.

4.2.3 Memory Size

Memory starvation for a processing unit involved in severe real time activities is, in general, a disaster. For a large processing unit with severe real time issues, the number of the active processes can reach very high values. The presence of paging activities can increase the inner kernel service requests per unit time. Experiments executed with smaller memory sizes on application servers show that a collapse of the system performance becomes possible in this case.

5. ACKNOWLEDGMENTS

Many thanks to Mr. Gianluigi Castelli, Chief IT Officier of FIAT Auto, S.p.A. for the high-profile collaboration offered by its group and for allowing us to report about this challenging activity.

Eugenio Cervetto
Performance Research

Title:
"Model-based performance analysis of an EDP/ERP-oriented wide area network"

Abstract:
"Many people think that the system-level performance modeling of a wide-area network is such a complex and time-consuming issue to discourage its application when a short time optimization is forced by the user dissatisfaction. Those people usually manage the efficiency issue at component level and, eventually, manage the unavoidable system level problems as a matter of experience and chance. Experience shows that, as the system complexity increases, this way of thinking fails in providing predictable and satisfactory solutions and, sometimes, even drives the “performance risk” to become unmanageable. PREDICTA by Performance Research is a performance modeling tool that has been specifically designed to face the system-level complexity of a large network and provide an analitic view of its performance behavior.
In this paper we report about a recent industrial application of PREDICTA by FIAT Auto in Italy. PREDICTA was used to investigate a severe efficiency problem on its ERP/EDP WAN serving all the production, commercial and financial departments in Argentina and, therefore, deeply impacting the performance of the organization. The modeling activity was performed at system level using model libraries describing the hardware layers (mainly IBM RS6000 servers, CISCO routers, physical links, controllers, protocols, etc.) and the IBM AIX operating system. The software layer models represent the most relevant exposure of this activity:
- BAAN Triton functions including Product Data Management, Work-item tracking, Enterprise Resource Planning and Computer Aided Process Planning.
- Data Base Management functions using Oracle.
Finally, special emphasis is given to the performance oriented "what-if" analysis and to the obtained results."

Year 2000: Lessons Learned

Presented by
Steve Goodhall
Managing QA Architect
Compuware Corporation

Compuware Corporation

It's January, 2000

- We just dealt with the biggest software quality problem in history.
- What did we learn?
- How can we do better?

Compuware Corporation

Requirement defects are the worst kind

- We never expected this stuff would still be running after 30 years.
- Did we ask?
- Who remembers?

Compuware Corporation

Testing is repetitive

- 3-5 tries to get the test right.
- Test to decide what to fix.
- Regression test before implementing.
- At least 2 actual executions for time dimensional test.
- Retest after correcting defects.

Compuware Corporation

Spend your effort close to the ground

- Touch-point testing checks your work.
- Unit testing finds the most defects.
- Integration testing finds interface problems.
- Stop when you stop finding errors.

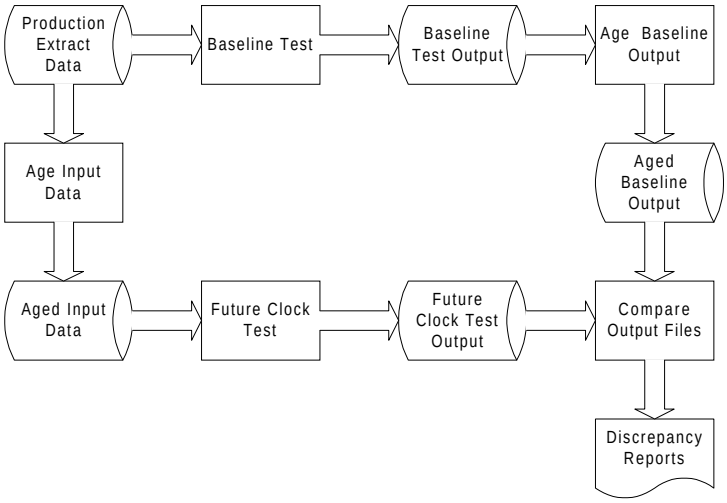
Compuware Corporation

Test what you didn’t change

- Most remediation errors are omissions.
- Touch-point testing doesn’t find these.
- Time-dimensional testing does.

Compuware Corporation

Time dimensional testing



Compuware Corporation

Test your test

- Didn't find any defects?
- You probably didn't try hard enough.
- Base tests on business functions.
- Measure coverage.
- Improve tests.

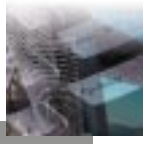
Compuware Corporation

Data-aging - not just for breakfast anymore

- Developed for Y2K testing.
- Useful for many things.
  - Shift test data to current time.
  - Test future events.
- Related tools
  - Data extraction and generation
  - File comparison

Make the tools better

- Clock emulation tools.
- Data aging and other data manipulation.
- Better test management.
- Better defect tracking.



Automation is a wonderful thing

TESTING SOLUTIONS	MAINFRAME	CLIENT/SERVER
Test Management	*QA*Director / *QA*Batch	*QA*Director
Capture and Replay	*QA*Hiperstation	*QA*Run
File and Data Management	File-AID	File-AID/CS
Problem Reporting/Tracking	*QA*TrackRecord	*QA*TrackRecord
Date Simulation/Validation	XPEDITER/*Xchange*	*QA*Run
Date Aging	File-AID/Data Ager	File-AID/CS
Unit Testing and Debugging	XPEDITER	*QA*Run, NuMega, XPEDITER/SQL
Performance/Regression	*QA*Hiperstation	*QA*Load
Problem ID/Resolution	CICS Abend-AID /Abend-AID	EcoSYSTEMS

Compuware Corporation

Defect Tracking and Analysis, Software Quality, and the Internet

By

Manu Das
SOFFRONT Software, Inc.

Abstract

Tuesday, April 27, 1999

It wasn't so long ago that defect tracking and analysis was not thought of as separate disciplines worthy of their own software tools. Feature requests and defect reports were kept manually on scraps of paper if at all. With the advent of the PC and the LAN, more and more development groups began using generic tools such as word processors, spreadsheets, and databases to record defects and requirements. These later unfolded into home-built applications but remained relatively basic and primitive. More recently, a number of specialized tools targeted at addressing the specific needs of development and program maintenance and support organizations in this area.

While the PC and LAN have made this evolution possible, it would not have taken place without the increase in complexity and size of the average programming projects that has taken place over the past ten years. This, combined with an amazing growth in popularity of object oriented technology and component-based development has led to an explosion of geographically dispersed development efforts. These factors have all resulted in a greater need for timely communication of defect and requirements information among development groups.

Current products address this need by incorporating a host of features including automatic e-mail notification to key personnel of changes to defect records, customizable databases, status transition matrix, scripting, integration with version control programs, advanced reporting and analysis capabilities, attachment of documents and test files to defect records and more.

The need for greater communication among widely dispersed development groups on complex projects and the advent of the Internet and intranets have led to the emergence of the Web enabled defect tracking systems. These systems allow users interaction to the defect information and users can download maintenance updates from a web site.

Now that the internet door has been opened on the support organization, users are bound to start asking some obvious questions. "If my computer is already connected to support, why do I need to call them on the phone when I find a bug?" "Why can't reporting a defect or requesting an enhancement be as easy as using help?" "Why can't the system simply capture configuration and program status information when a defect is encountered, send it to support with a problem report, and automatically download a fix if one is available?" "Why can't users be automatically notified about maintenance updates?"

This technical seminar will explore these issues in more depth and attempt to identify facilitating technologies and obstacles to implementation of systems to meet these emerging requirements. The target audience will be development, quality assurance and support engineers trying to understand the future direction of their discipline and anyone interested in helping to stimulate an examination of this subject. Topics to be discussed will include the following;

1. The Evolution of Defect and Request Tracking Systems
2. Current State of the Art in Tracking Systems
3. Impact of the Internet and Other Technologies
4. Future of the Tracking systems

Presented by:

Peter Szirmak and Scott Gardiner

Y2K

The remaining tasks and the challenge of leveraging for the future

Information Balance

Copyright Information Balance Inc.

Agenda

- Year 2000
  - Remaining tasks
  - Quality Assurance techniques
- Beyond Y2K
  - Challenges observed during Y2K Projects
  - Leveraging Y2K experience

Copyright Information Balance Inc.

Y2K Methodology

- Stages:
  - Planning
  - Inventory
  - Assessment
  - Renovation
  - Testing
  - Implementation
  - Continued Readiness
  - Contingency Planning

Complete

In Progress

Copyright Information Balance Inc.

What about Quality?

- Has quality been considered?
  - Quality of Code Renovations
    - High error rates of some renovations
  - Quality of Testing
    - Coverage and documentation of results
  - Quality of Overall Program
    - Have best industry practices been followed?
- Has quality been verified independently?

Copyright Information Balance Inc.

Independent Code Verification

- The problem
  - Some vendor or internal assessment methods are error-prone
  - 10-40% error rates found in some locations
- Multi-level Quality Assurance
  - Use a separate approach to confirm quality and / or identify errors
  - Catch what has been missed in renovations and testing

Copyright Information Balance Inc.

Y2K Testing Methodology

- The tasks
  - Test entire application inventory
  - Test several times (baseline, several future dates, infrastructure upgrades)
  - Provide clear audit trail to prove due diligence
- Year 2000 Testing Methodology
  - Templates define deliverables
  - Clear trail from Test Plans through Test Conditions to Test Results
  - For all applications, infrastructure and desktop software

Copyright Information Balance Inc.

Program Review

- The task
  - To confirm completeness and accuracy of the Y2K program
  - Provide a confidence factor to management of the organization's Y2K readiness
- Independent Program Review
  - Measure program against industry best practices
  - Determine actions, if necessary, to fully complete the program

Copyright Information Balance Inc.

Leveraging for the Future

- What have we learned from Y2K?
  - Need better control of Inventory
  - Need for improved Project Management
  - Ability to reuse and extend Legacy Systems
  - Improve Testing and QA processes
  - Invest in Training

Copyright Information Balance Inc.

Project Outsourcing

- Many companies have outsourced Year 2000 renovations
- Why not outsource development?
- Advantages:
  - Professional Project Management
  - Results oriented
  - Delivery assured
  - Compressed timeframes

Copyright Information Balance Inc.

Legacy Extensions

- Legacy systems are now Y2K-ready
- Why not extend to handle business requirements of the next century?
- Advantages:
  - Retain the investment in the Legacy
  - Web-based extensions
  - Integration / data synchronization extensions
  - Retain quality / testing knowledge

Copyright Information Balance Inc.

Testing Methodology

- The Challenges
  - Project, not process, focus
  - Testing viewed as a phase at end of development
  - Lack of enthusiasm for Testing and QA
  - Testing tools still on the shelf
  - Lack of Testing Environments
  - New technologies - Web, others

Copyright Information Balance Inc.

Testing Methodology

- Requirements for the New Millennium
  - Overall focus on quality processes featuring inspections and measurement
  - QA and Testing are performed throughout each stage of development
  - Recognize Testing / QA as a career path with appropriate training and rewards
  - Create Testing Environments using appropriate tools to support new applications and legacy applications

Copyright Information Balance Inc.

Training

- The Challenge
  - Must continue to invest in staff education
  - Training in new / revised processes
- Focussed Training
  - Testing and QA Techniques
  - Development Processes
    - Data, Process, Object Modeling
  - Specific Technologies
    - Web-based, Client/Server, Mainframe

Copyright Information Balance Inc.

Information Balance

- **Y2K Services**
  - Y2K Renovations
  - Testing Methodology
  - Multi-Level Quality Assurance

  (Independent CodeReview)
  - Y2K Program Review
- **Millennium Services**
  - Project Outsourcing
  - Legacy Extensions
  - Testing Methodology
  - Training

Copyright Information Balance Inc.

Information Balance

Questions?

Copyright Information Balance Inc.

Y2K: The remaining tasks and the challenge of leveraging for the future

Paper for Quality Week '99

Date Printed:
April 29, 1999

Author:
Peter Szirmak
Scott Gardiner
Information Balance Inc.

Background

Most North American companies are well on their way to becoming Year 2000 compliant. Having completed their renovations of business critical systems, they are now involved in testing those systems. Some lower criticality components such as desktop software (e.g. spreadsheets) and applications (e.g. desktop databases) are still being renovated.

During the remainder of 1999, some key areas remain to be addressed including:
• Testing, including testing of applications, infrastructure, embedded systems and desktop systems,
• Quality Assurance (or Independent Code Verification) and.
• Independent Program Reviews.
•These tasks will ensure the quality and integrity of Year 2000 programs by providing an independent assessment of processes and results, ensuring that no details fall between the cracks. The various tasks involved follow.

Independent Code Verification

During the assessment and renovation of the application code, various techniques are employed by the internal IT staff or by an external vendor to find and fix the Year 2000 issues. The applications are then tested to ensure the quality of the assessments and renovations.

Our experience has shown that some assessment and renovation techniques don't necessarily find all of the code problems. In addition, Year 2000 renovations, particularly manual renovations, can introduce new defects into the code. Testing is often difficult due to a lack of application knowledge, lack of test scripts and little systems documentation.

Our Quality Assurance projects have detected an extremely high incidence of errors in some systems that have been renovated and tested!
Independent Code Verification, or what we call Multi-level Quality Assurance, assesses the quality of your code renovations and testing by providing an independent assessment of your code using unique assessment techniques. This approach can catch many bugs that were undetected in the initial pass of assessment and testing, and provides an extra assurance of the quality of Year 2000 compliance projects.

Year 2000 Testing Methodology

Year 2000 Testing is a huge chunk of the overall Year 2000 project. It is estimated that anywhere from 40-60% of the total compliance project effort may be required for the Testing portion.

Why is this necessary effort so large?

Essentially, every application in your inventory must be tested. And not just once. Year 2000 testing typically involves several passes including: a baseline test (for current dates), future date testing for several dates and application testing for upgrades to the underlying infrastructure (hardware, operating systems, etc.).

The Methodology used should provide clear steps, well-defined deliverables and templates to get the testing up and running quickly with the least amount of experimentation and rework.

Desktop software testing requires a slightly modified approach. The methodology is downsized to fit the smaller size and scope of end user-supported software applications. This smaller methodology is intended to handle spreadsheets, databases and other end user-supported desktop applications.

Year 2000 Program Review

Has anyone stepped back and taken a big picture view of your entire Year 2000 program? Are you following best practices as defined for your industry? Has any key item been missed or been performed at a sub-par level?

The purpose of an independent program review is to verify the quality of the entire Y2K program and provide a confidence level to management that the organization will continue to operate into the next century.

The review should check an organization's methodologies and practices against industry best practices, confirm the claims of compliance from the various departments or divisions within the organization, and determine an action plan, if necessary, to complete any items that are not up the accepted industry level of completeness.

Lessons Learned

We have all learned a lot from the Year 2000 Compliance experience. Some of these lessons were predictable, while others were not.

One component of these projects that proved to be a most difficult task for many organizations was the creation of an inventory of applications, infrastructure components and embedded systems. Many did not realize or predict the effort involved, which has resulted in the realization of the value of asset management tools and techniques.

The need for good Project Management was a predictable outcome. Good project managers have always been a prized asset; this position requires a skill-set that is quite rare. Year 2000 compliance is the largest project that many companies have ever faced, so the fact that some have been delayed or late is not a surprise.

After years of verbally trashing their legacy applications, companies are realizing that these systems are a huge corporate asset that should be valued. Now, these applications have been expanded, extended and have been made Year 2000 compliant.

 Copyright Information Balance Inc.

We must learn to love these systems and learn how to extend them to solve current and future business requirements.

The lack of enthusiasm for Testing is not new to Testing and QA professionals. Our hope is that the large effort required for Year 2000 testing will serve as a catalyst for organizations to improve their testing processes.

The IT staff turnover of the last couple of years and the need to test all of our applications at once has exposed a knowledge issue: most organizations have a low level of business, applications and technology knowledge.

The next section address these issues and offer some options for solving the issues.

Project Outsourcing

After the panic of the rollover to the year 2000 subsides, many organizations will be faced with a backlog of development activity that has been delayed by the Year 2000 projects. How does a company make significant headway into that backlog?

One way is through project outsourcing. Companies that have proven to be able to successfully manage large, complex Year 2000 renovations are a natural fit to manage outsourced projects and to ensure that the functionality and quality is delivered as required by the client.

We feel the key to such outsourcing is proven project management capability. Remember: good programmers and technicians are important, but it is the project manager that ensures these specialists retain their focus and bring their projects in on target and on budget.

Legacy Extensions

How do we retain the value in our Legacy applications?

Extending applications for new input mediums is an ever-growing area of development. Organizations are web-enabling their applications for e-commerce, as well as using many other innovative input mediums such as IVR, Call Centers and PDA's to improve customer service and productivity.

As well, many of the existing applications are isolated from each other. It has been estimated that approximately 40% of development effort involves integrating these islands of automation. Integration efforts will involve middleware, message queuing and database technologies to integrate legacy systems.

From a QA / Testing viewpoint, Integration testing will become a more important aspect. Integration Testing will need to verify the correct interactions between many components deployed on multiple platforms and to also confirm the performance and scalability of such solutions.

Many exciting challenges lie ahead in this area.

Testing Methodology

There are a number of challenges in how testing is currently performed in most organizations. Some of the challenges we have experienced include:

• Testing is viewed from a project viewpoint, where the goal is delivery of a specific product, rather than from an organizational process viewpoint, where the goal is to improve the overall development process.

• Testing is viewed as a phase squished between the coding and the project target date.

• Our IT organizations are comprised of developers who want to do modeling or coding, but not testing. Testing is viewed as a chore or a necessary evil by the majority of developers.

• Many testing tools are still shelfware. Despite the fact that Year 2000 testing typically required applications two, three or more times, automated test tools have not caught the interest of most developers.

• A significant challenge is the lack of testing environments in which applications can be tested in isolation from the production environment. This is most acute in the client/server environment.

The following features must be included in improved Testing and QA programs:

• A focus on the quality of every deliverable. Inspections can provide a quick start to this quality focus, with additional automation being provided as required.

• Testing is not a phase, but a continuous process that is performed throughout the development life cycle. The development life cycle should contain steps that prevent defects from travelling further into the project. In this way, testing is a method of confirming quality rather than the primary method of preventing defects.

• The lack of enthusiasm for testing and QA must be met by a commitment to recognize and reward QA professionals. Testing and QA should be recognized career paths. We should also recognize that the way we recruit / assess developers and testers may need to be quite different: development managers should recruit developers and QA managers should recruit testers.

• Testing environments, especially client/server test labs, containing appropriate automated test tools are required to make the QA / testing professionals as efficient as possible. Consider retaining your Y2K Test Labs for ongoing use in development activities.

Training

To ensure the ongoing quality of your development efforts, training plays a significant part. Training over the last 18 months has typically been directed at Year 2000 efforts or put on hold completely.

Staff training programs must be re-established with the goal of creating quality deliverables the first time from the development efforts. No company has time for re-work and trial-and-error training. Training programs should include hands-on training courses and follow-up guidance and tutoring on the most appropriate usage of tools and techniques.

Information Balance Y2K and Post-Y2K Services

Information Balance is uniquely positioned to assist your company in completing your Year 2000 projects and in leveraging your experience for future development efforts.

Our Year 2000 services have been developed, focussed and sharpened through our work with many large- and medium-sized companies worldwide. In 1999, our clients are particularly interested in Quality Assurance code reviews, testing of desktop applications and independent reviews of Y2K Program processes.

Into the new millennium, Information Balance will use our proven project management and testing expertise to continue to deliver solid, practical services to our clients. **Project outsourcing** allows companies to off-load development to a company with a proven track record of successful project deliveries. **Legacy Extensions** recognizes the need to extend the existing code base in new directions to solve new business challenges. **Testing Methodologies** are utilized to entrench a quality mindset throughout all phase of development. **Training** brings all of the development, testing, quality assurance and tools components together for your staff.

About Information Balance

Information Balance Inc. offers Information Technology consulting and technical training services, encompassing solutions for both public and private sectors in Information Systems Planning, Project Management, Application Development/Project Outsourcing, Systems Integration, Software Testing and Quality Assurance.

Building on years of training and consulting experience, and expertise with some of the leading Year 2000 tools, we also offer a complete cycle of Year 2000 services, from Assessment through Renovation to Quality Assurance, Testing, and Project Review.

 Copyright Information Balance Inc.

intrinsa

Concurrent Software Development and Quality Assurance:

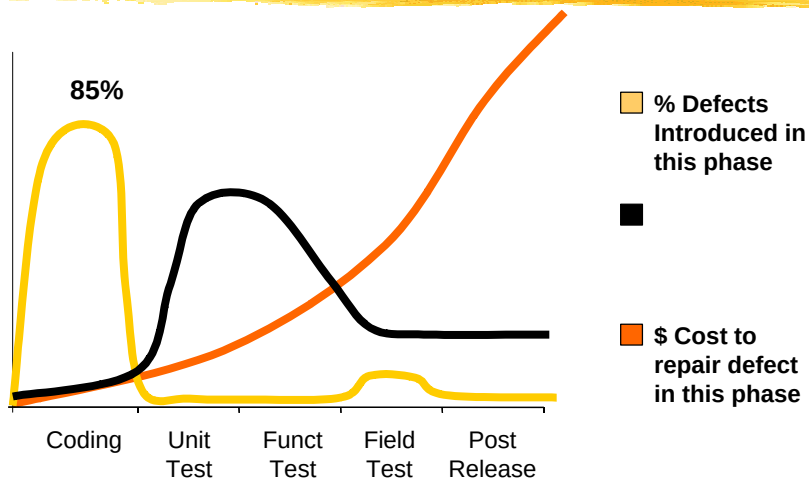
New Tools and Techniques

Outline of Presentation

- Why Undertake Concurrent Software Development and Quality Assurance?
- Exploring How To Improve Quality During
- Concurrent Development and Quality Assurance is Possible

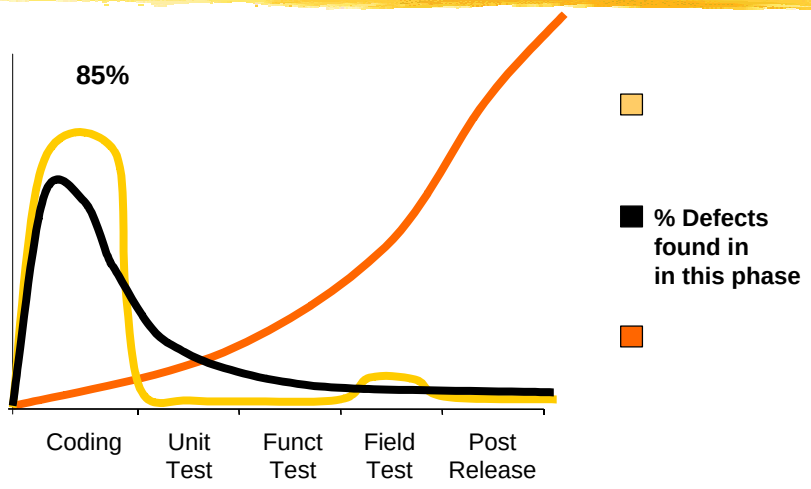
intrinsa

Why Undertake Concurrent Software Development and Quality Assurance?



intrinsa

Ideal Defect Creation, Detection and Repair



intrinsa

Reviewing the Benefits of Concurrent Development and QA

Major Benefits

- Allows bugs to be fixed in the same phase they
- Finds bugs when they are the least costly to fix

Additional Benefits

- Allows QA to focus on *intent* errors, not *effect* errors
- Can shorten time-to-market

intrinsa

intrinsa

Exploring How To Improve Quality During Development

The Ideal Solution

- Available in Coding Phase
- Identifies the Bug's Source
- Finds Significant Bugs
- Performs Comprehensive Testing
- Usable in Embedded Systems
- Low Development Impact
- No Impact on Code
- No Test Cases
- Low Spurious Rate

Current Solutions

- Manual Code Review
- Software Testing Tools

intrinsa

Manual Code Review

- Pros
  - Still considered one of the best methods of early bug detection
  - Can be performed at any time
  - Does not require test cases
  - Lower cost than other methods
- Cons
  - Extremely time-consuming (100 LOC/hr per reviewer)
  - Heavy impact on development resources
  - Low frequency
  - Inefficient use of development resources
  - Non-comprehensive review of code
  - Difficult to find very complex / random defects

intrinsa

Existing Software Testing Tools

- Automated Code Reviewer
  - PREfix
- Execution Monitors
  - Purify, Memory Advisor, BoundsChecker
- Static Analyzers
  - lint, Code Advisor, PC-lint, Flexelint
- Automated Test Management
  - Xrunner, Silk, Hindsight
- Code Comprehension
  - Discover, iSight++

intrinsa

How Do Current Software Testing Tools Compare to the Ideal Solution

Ideal	Auto Code Review (PREfix)	Execution Monitors (Purify)	Static Analysis (Lint)	Auto Test Managemt (Xrunner)	Code Comp. (Discover)
Available in Coding Phase	✓		✓		✓
Identifies Source of Bug	✓	✓	✓		
Finds Significant Bugs	✓	✓		✓	
Comprehensive Testing	✓	✓		✓	
Usable in Embedded Systems	✓		✓		✓
Low Development Impact	✓		✓		✓
No Impact on Code	✓		✓	✓	✓
No Test Cases	✓		✓		✓
Low Spurious Rate	✓	✓		✓	**N/A**

intrinsa

Concurrent QA and Development Is Possible Via PREfix Automated Code Reviewer

PREfix Automated Code Reviewer:

- All the benefits of Manual Code Review without the
- Patented technology that simulates program execution in source code *without test cases*
- Works on C/C++ source code under Windows NT and UNIX platforms

intrinsa

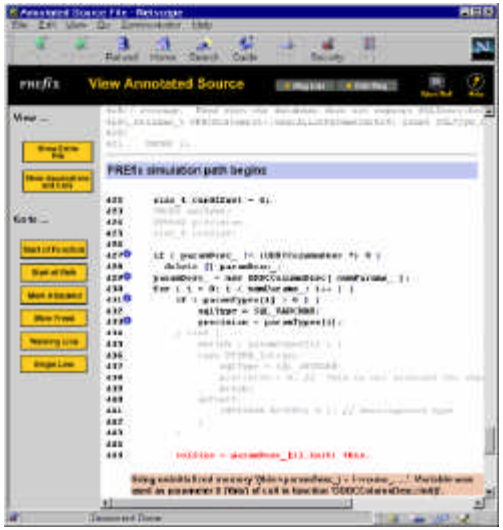
PREfix Benefits

- Useful from the beginning of the coding phase
  - Finds bugs immediately after they are coded
  - Works on incomplete (non-executable) code
  - Does not require test cases

intrinsa

PREfix--Benefits (Cont.)

- PREfix identifies where the bug is in the source code
  - Indicates under what conditions the bug occurs



intrinsa

PREfix--Benefits (Cont.)

- PREfix finds significant defects

Memory Management
- ✓ Freeing pointer to already freed memory
- ✓ Freeing pointer to global memory
- ✓ Freeing pointer to static memory
- ✓ Freeing pointer to stack memory
- ✓ Freeing pointer in middle of memory block
- ✓ Freeing NULL pointer
- ✓ Freeing invalid pointer

Initialization
- ✓ Using uninitialized memory
- ✓ Freeing uninitialized pointer
- ✓ Dereferencing uninitialized pointer

Resource Exhaustion
- ✓ Leaking Resource
- ✓ Leaking memory

Pointer Management
- ✓ Dereferencing NULL pointer
- ✓ Dereferencing invalid pointer
- ✓ Dereferencing pointer to freed memory
- ✓ Exports pointer to local
- ✓ Returns pointer to freed memory

Illegal State
- ✓ Resource in illegal state
- ✓ Illegal value
- ✓ Divide by zero

intrinsa

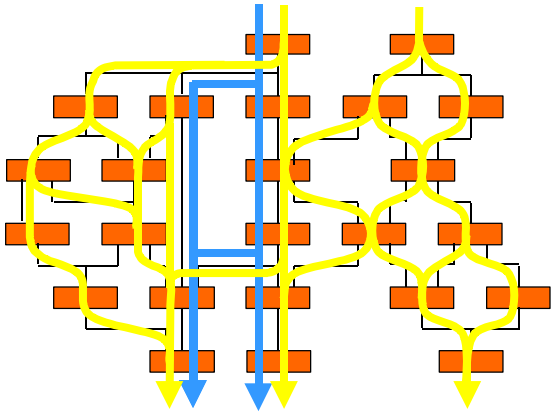
Prefix finds ~ 1 Bug/2000 LOC

PREfix--Benefits (Cont.)

Comprehensive Testing

Traditional Test Path Coverage

- Traditional Test Path Coverage
- PREfix Test Path Coverage



intrinsa

PREfix--Benefits (Cont.)

- Usable in Embedded Systems
  - Does not require target environment to analyze source code (e.g., does not use operating system services to check the code)
- No Impact on Code
  - Does not expand code size during testing
  - No code instrumentation
- Does Not Require Test Cases
  - Code execution is simulated dynamically
  - Not limited by the scope of test data

intrinsa

PREfix--Benefits (Cont.)

- Low Development Impact
  - Seamless integration into regular software build process
  - Browser-based review of error report
  - Most significant bugs are prioritized for more effective review
- Low Spurious Error Rate
  - Defects are automatically sorted, prioritized and ranked according
  - Warnings marked “unimportant” disappear

intrinsa

Summary

- You can perform concurrent Quality Assurance and Development -- with *PREfix!*

intrinsa

Quality Week '99

Fred Scheck, CQA
Vice President

Interim TECHNOLOGY
TheConsultingGROUP

Interim: How the World is Working

Beyond Y2K & The *New* Software Quality Professional

Interim TECHNOLOGY
TheConsultingGROUP

Interim: How the World is Working

Beyond Y2K

What have we learned ???

- 3 Major Disciplines Needing Most Attention

Interim TECHNOLOGY
TheConsultingGROUP

Interim: How the World is Working

Beyond Y2K

Project Management

- Planning - “Touch Point” All Dependencies
- Reliable Estimates - Metrics and Measures
- Business Continuity -- Mission Critical Processes

Interim TECHNOLOGY
TheConsultingGROUP

Interim: How the World is Working

Beyond Y2K

Software Quality Management

- Software Quality Plan
  - ✓ Overview
  - ✓ Quality Assurance and Quality Control
  - ✓ Methodology
  - ✓ Organizational Requirements
  - ✓ Templates
  - ✓ Glossary

Interim TECHNOLOGY TheConsultingGROUP

Interim: How the World is Working

Beyond Y2K

Tool Support

- Requirements Manager
- Test Management
- Configuration Management
- Issue Management

Interim TECHNOLOGY TheConsultingGROUP

Interim: How the World is Working

The *New* Software Quality Professional

- 4 Major Roles to Play

Interim TECHNOLOGY
TheConsultingGROUP

Interim: How the World is Working

The *New* Software Quality Professional

Product Business Analyst

- Business User Communication & Facilitation Skills
- Product & Subject Matter Expertise

Interim TECHNOLOGY
TheConsultingGROUP

Interim: How the World is Working

The *New* Software Quality Professional

Software Technical Analyst

- Multi-Tier Environment Knowledge
- Database & Communication Experience

Interim TECHNOLOGY
TheConsultingGROUP

Interim: How the World is Working

The *New* Software Quality Professional

Industry Solutions Analyst

- Internet
- Web
- E-Commerce

Interim TECHNOLOGY
TheConsultingGROUP

Interim: How the World is Working

The *New* Software Quality Professional

Tool Specialist

- Comparative Assessment Knowledge
- Tool Certified
- Training & Mentoring Experience

Interim TECHNOLOGY
TheConsultingGROUP

Interim: How the World is Working

Quality Week 1999
Facing the Future

Beyond Y2K and the *New* Software Quality Professional

The Global Y2K correction will continue into 2000 as companies transition their efforts from failure prevention to failure management. What is on the horizon beyond Y2K? New products, projects, and technology upgrades that have been deferred due to millennium priorities will be launched in record speed as companies catch up to meet their business goals. Will we have learned enough from the enterprise Y2K project to help us avoid the pitfalls of our next mission critical projects? How will the software quality professional need to evolve to meet the challenges of the new platforms and market driven product delivery timeframes? Let us explore these questions through observations of the Y2K process.

Beyond Y2K

What has Y2K taught us to date? First we were not prepared. Surprised? You shouldn't be. The management processes that we have been promoting and partially implementing are the key items that were not in place for most companies when they began the Y2K assessment process. Second, the best practices that were implemented to assist in Y2K will not be carried forward unless industries invest in the roll out and maintenance of these processes for new projects. The lessons we are still learning from Y2K must be carried forward to mitigate the issues and risks for new projects beyond Y2K.

Three major areas have emerged as needing the most attention: project management, software quality management, and automated tool support.

Project Management

Y2K risk, budget and time constraints will continue dominating the remainder of this year. New and innovative industry initiatives to support business needs will begin their launch in 2000. As we proceed into the new millennium, deferred projects for enhanced products and associated processes will need to be integrated into the new corporate solutions. Managing these complex systems integration, business process reengineering, and platform migration endeavors will require proven large project disciplines, some of which has been successfully demonstrated during the Y2K corrective measures.

What has Y2K taught us thus far for managing complex integrated and dependent projects? First and foremost, planning all activities for these projects must cover all of the touch points that are to be affected. Y2K has made us acutely aware that any corporate global project requires considerable thought as to how the solution will be implemented. We must take into account all of the dependencies, lead time preparation, and proper personnel to support these activities. This planning process was generally demonstrated to be inadequate during the early stages of Y2K corrective activities and still remains inadequate today in many corporations.

Second, Y2K has confirmed that most organizations do not prepare realistic estimates for major projects. One local joke for estimating "rule of thumb" has been to provide a corporate best analysis guess for Y2K, then multiply that number by a factor of 2 or 3. This joke has become a reality, with to-date time and costs grossly exceeding most original corporate budget estimates. Improved metrics and measures must be implemented to properly forecast the next projects, without which corporations will have limited confidence in the reliability of future budget requests.

Finally, Y2K contingency planning, commonly referred to as business continuity, has been seriously pushed back by many companies until this year. The problem is not only how IT will recover their applications, but also how the business functions including IT will continue to be operational should an interruption or failure occur within a critical business process. IT disaster/recovery plans typically addressed hardware and software failures, not disruption at the business units nor the receiving/distribution legs of the business product. Business continuity plans will need to accompany all mission critical processes as part of the implementation plan for these new solutions in the next millennium.

Software Quality Management

A Software Quality Assurance Plan (or SQAP) is fundamental in managing the development, enhancement and maintenance of software products. This plan describes the procedures required to apply Quality Assurance and Quality Control standards to each step of the product lifecycle. Many corporations have a quality assurance entity that may follow such a plan, but it was found that in most Y2K projects these key concepts were not consistently integrated in the corporate Y2K strategy. Suggested components of a SQAP are detailed in a sample that follows.

1) Plan Overview

- Purpose of Document
  Process quality is the measure of the efficiency and effectiveness of the execution of defined processes. It results from the correct application of skills, tools, procedures and standards to a process, and from the continued monitoring of the process to manage deviations. Key process quality requirements include:
  (a) Development, integration, and testing processes are being conducted in a consistent manner across organizational units
  (b) Training on all processes is provided to all personnel engaged in these activities
  (c) Metrics are collected to allow analysis of process efficiencies and effectiveness
  (d) Process reviews and random audits are in place to spot process difficulties and to initiate corrective actions, and identify best practices for publication and adoption across the organization
- Objectives of Plan
  (a) QA/QC processes which will ensure a quality product
  (b) Quality criteria to demonstrate that the projects and products meet their requirements
  (c) Plans and activities necessary to ensure that QA/QC criteria are met
  (d) Reference techniques and standards used to perform the specified QA/QC activities
- Scope of Plan
  (a) Quality Assurance Processes
  (b) Quality Assurance Organization
  (c) Test Procedures
  (d) Change Control
  (e) Defect Tracking
  (f) Test Data Management
  (g) Quality Metrics
- Metrics Monitoring
  The ongoing monitoring of process metrics is achieved through the use of standard work product status reporting and through the use of process audits. Measures of process quality include:
  (a) Process Reviews
  (b) Number and type of significant variations from defined procedures, where "significant" is defined as omitting a process step or failure to use standard templates or forms.
  (c) Technical Reviews
  (d) Number of issued identified and logged
  (e) Number of reviews completed relative to the number of items subject to review
  (f) Defects
  (g) Number of defect reports logged
  (h) Number of defect reports resolved
  (i) Number of open defects
  (j) Average time expended to close a defect
  (k) Product Build or Component Performance
  (l) Product Build or Component Development/Integration Phase Elapsed Time
  (m) Test Execution Elapsed Time
  (n) Product Build or Component Test Phase Reiterations

2) **Quality Assurance and Quality Control**
   - Quality Assurance
     (a) Quality Requirements for product implementation
       - Product functions to business and technical requirements
       - Product meets performance and load requirements
       - Code complies with established coding standards
     (b) Quality Assurance Strategy
       The QA strategy establishes QA procedures to monitor the full renovation life cycle. Quality assurance reviews are of two types:
       - Reviews and Walkthroughs - by someone other than the deliverable producer
       - Audits - Periodic reviews of deliverables and/or walkthroughs by a QA representative
     (c) Procedure and Template References
       - Test Plan Template
       - Test Summary Report
       - Problem Report Template
       - Problem Report Log Template
       - Test Progress Template
       - Test Case Specification Template
       - Test Case Log Template
     (d) External documents
       - Coding Standards
       - Change Control Procedures
       - Review Procedures
       - Defect Resolution Procedures
       - Installation Procedures
       - Configuration Management Procedures
   - Quality Control
     (a) Quality Control Strategy
     (b) Quality Control Process (mirrors development lifecycle model - hierarchical, iterative, etc.)
       - Unit Test
       - Integration Test
       - Systems (or Release) Test
       - Performance and Load Test
       - Business Acceptance Test
3) **Methodology**
   For each quality assurance identified control point (milestone, major activity, etc.) identify the following:
   - Project process (milestone/activity name)
   - QA control point name
   - QA defined purpose
   - Exit Criteria for QA completion
   - QA task list (Task, QA Requirement, Role Responsible)
4) **Organizational Requirements**
   - Project resource profiles
   - Project training requirements
   - Project approved tools
5) **Templates (repository reference)**
6) **Glossary of Terms**

Tool Support

Most companies would have been in a more serious Y2K time constraint if it were not for the deployment of automated and auto assisted tools. Inventory, impact assessment, remediation, testing, and verification tools have been instrumental in reducing the time and improving the accuracy of the Y2K corrective measures. Although they were not the silver bullet, as many companies had envisioned and some vendors had proposed, they have been key in the implementation of the Y2K Compliance Plan. Y2K has radically demonstrated that these tools, and derivations yet to come, will be important in helping to accelerate the applications project life cycle and continue the maintenance and enhancement management throughout the product life. A key point to emphasize is a tool is only as good as the disciplined process that it supports. Four key application processes have benefited from tool support in Y2K.

Requirement Management tools help to document and manage the business and technical requirements, and the releases for change throughout entire product life. Just as we coordinated the Y2K remediation requirements for future test and verification criteria, documented business requirements, product reviews, and a change management process will depend heavily on tool support to reduce the overhead of creation, while reinforcing consistency and continuity of the information.

Test Management tools have already provided a repository for managing application regression tests as well as Y2K compliance tests. As Y2K has demonstrated, quality control has represented forty to sixty percent of the overall effort, with tools providing demonstrative reduction in time for regression testing of the remediated code. These tools will continue to evolve with the expectation that one can not envision having regression and release tests scripts without this support.

Performance and load testing are categories we have typically limited to new applications, purchased packages, and environment upgrades. Y2K has added to performance considerations, as most applications were not designed to accommodate data windows and bridges. The load/performance test tools have significantly enhanced the ability to simulate a business environment to validate that the compliant code will actually function within mandated response time and production cycle operation windows. Internet and Web applications are forecasted to dominate new business solutions and load/performance test tools will be essential in the business acceptance test.

Configuration Management support has been available for years in almost all platforms, with tools that coordinate the promotion and demotion of software, documentation, and execution procedures. Y2K has added to the complexity of managing production environments with the need of fully operational test environments and, in addition, "clean environments" that are Y2K compliant. Although most companies have some level of configuration management processes in place, they were not prepared for the millennium. The mainframe rich tool investment that Y2K had reinforced will now have to be supplemented with the emerging technology (client/server, e-based, web) configuration tools of equal or greater caliber to adequately manage this growth.

Issue management has been an "issue" forever. The management of issues is fairly straightforward once everyone understands and uses the process. Y2K issue management has been chaos for many companies – primarily due to the lack of any consistent process and the poor communication of any process, should it exist. Y2K has clearly demonstrated that contingencies, dependencies, errors, and other items that could have knowledge share benefit across the organization were not coordinated at an organizational level, resulting in considerable time and cost overruns that could have been mitigated if the information was at hand for others to review. Tools will make it easier to document, distribute, track, record resolution, and close issues, and ultimately improve the return on investment beyond Y2K when an issues management <u>process</u> is defined, communicated, and supported in the organization infrastructure.

The *New* Software Quality Professional

Prepared or not, we will begin 2000 facing some of the same needs and experiencing a variety of issues that have been demonstrated for more than the past thirty years. Yet, industry will require a new software quality professional to embrace and be skilled in the products and technologies that will be more prevalent in the next millennium, and have more business savvy. What this professional should have in their tool kit is best defined by the roles that will need to be fulfilled in the business solutions forecasted. We see four roles that the new software quality professional should prepared for.

Product Business Analyst

In addition to being a solid quality assurance and quality control skilled engineer, the software quality professional will need to be a skilled business analyst, as most immediate new projects will address the backlog of business requirements put on hold until the millennium correction completed. This will include platform migrations, system replacements, and new technology solutions. The professional will be working more with business users in understanding the functional requirements and as more technology solutions deploy emerging technology object based design, creating plans and scripts that focus on business objects and less on legacy code testing concepts.

Working with the business units will also require subject matter knowledge in the key industries, such as financial, manufacturing and insurance. It will be important to share the vocabulary and concepts of the business to effectively develop the test processes, without being heavily dependent on the business subject matter experts. Excellent communication skills are a key attribute of an analyst, and the ability to interact at most levels of the business community will be required to be effective in confirming quality requirements.

Software Technical Analyst

The business analytical skills will need to be complemented with new technology skills. Emerging technology solutions including Web based design, distributive solutions and multi-tier architecture such as client, server and data hosts require a fundamental technical knowledge to properly define the appropriate test strategies. Integration tests will be more network centric, while systems tests will mandate complete platform knowledge for defining and staging entry/exit criteria. A background in database standards and communication protocols will be necessary to properly set up the test data and access the correct version, as release testing will be the prevalent choice for iterative development and system integration projects.

Industry Solutions Analyst

Combining the business and technical analyst expertise will not be enough to satisfy the software quality requirements for a number of the new solutions. Internet, web based, and electronic commerce (E-Based) industry products will be dramatically expanding in the immediate future. The quality professional will require a conceptual understanding of these business solutions to prepare the proper strategy, plan, and templates. Functional and performance/load testing will be the dominant strategies to certify this architecture.

Tool Specialist

Aggressive schedules and conservative budgets require automated assistance at any level to improve productivity. The new software quality professional will need experience in test productivity tools to meet the growing competitive demand of product to market time constraints. Familiarity with tools including configuration management, capture/playback, performance, load, data aging, and data comparison will be critical in developing the test strategy and deploying the test packages.

Handouts to be given at the time of presentation!

Handouts to be given at the time of presentation!

Tools for usability validation

- Need for usability testing
- Methods for usability testing
- Need for automation
- *ErgoLight* implementation
- Features and limitations
- Getting started

1

The need

Developer of
MS-Windows
Desktop Applications

- Need to redesign the GUI?
- Need to improve the On line Help?
- Need to enhance the User Documentation?

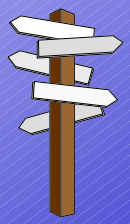
2

Don't ...

- Don't rely on standards
  - Standards were arranged by the programmers ... for the programmers
- Don't rely on style guides
  - Many users are not familiar with the guides
- Don't rely on design rules
  - Many users do not obey the rules
- Don't educate the user
  - Users might prefer to change vendors

3

Do ...



Adapt your product to the user's needs!

How?

Test the product usability, with real users:

- During prototyping and alpha testing stages
  - Do laboratory testing
- During beta testing and product deployment stages
  - Do remote testing

4

Laboratory testing

Regular testing

- Facilitation
- Video recording
- Thinking aloud
- Observer's logs

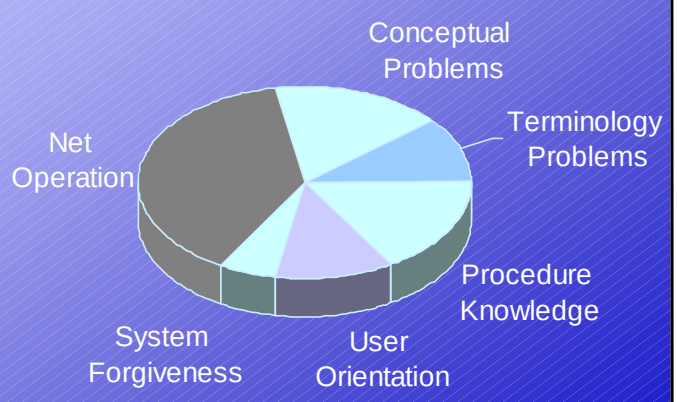
Automation -
Understand the user's confusion

- User actions => error prone controls
- Product modes => error prone modes

5

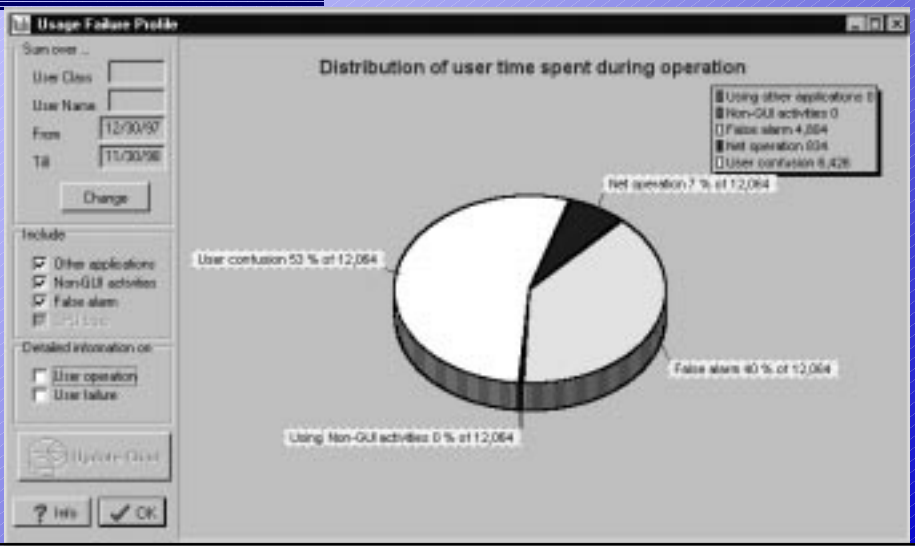
Education Systems

- CAI
- CBT
- CAE
- Guides
- Tutorials



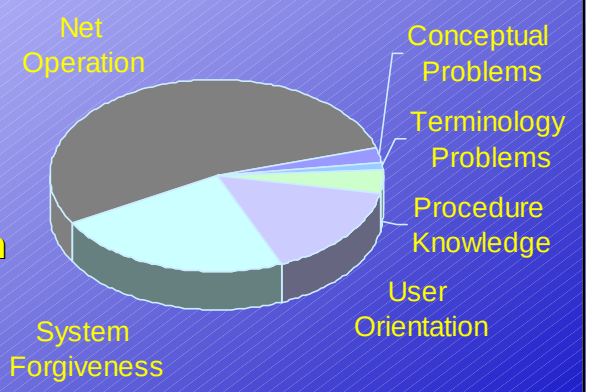
6

User's time



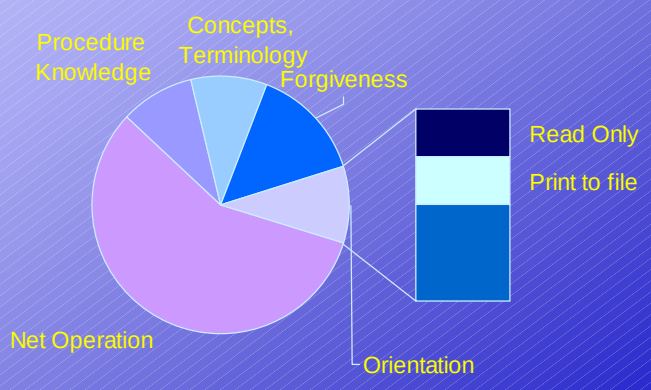
Production Systems

- Data entry
- Data mining
- Office
- CAD/CAM
- Administration
- CASE



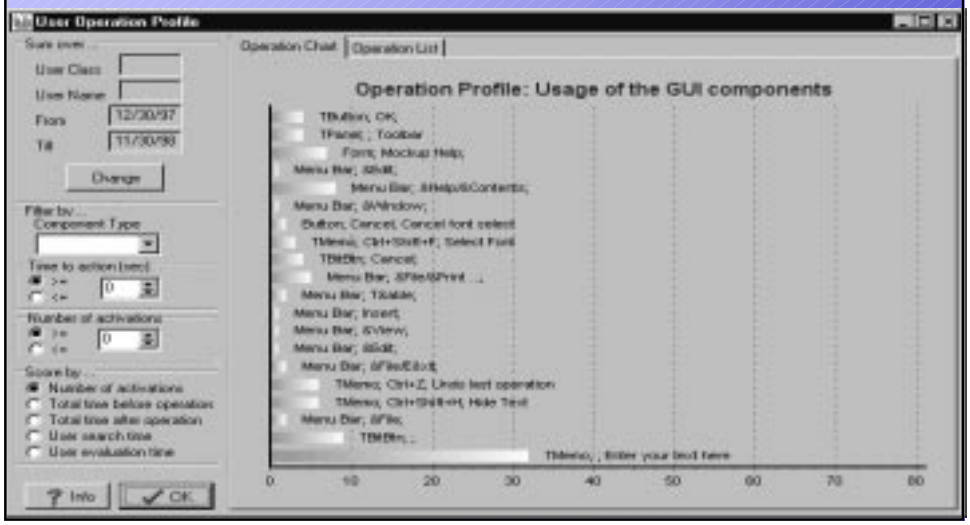
8

Time Spent (example)



9

Operation profiles



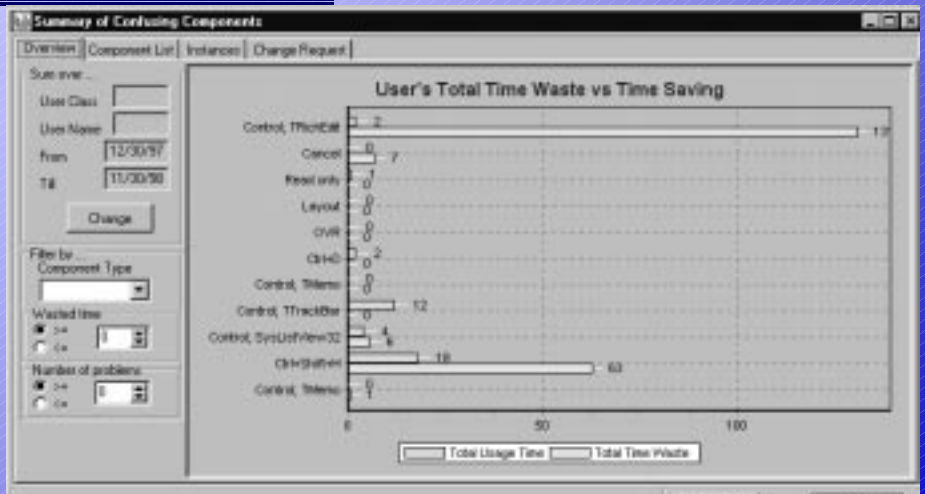
Barriers to high performance

In production systems users waste time due to:

- Unintentional actions
- Inadvertent actions
- Working in the wrong mode.

11

Evaulation
(example)



Integration in usability labs

Record

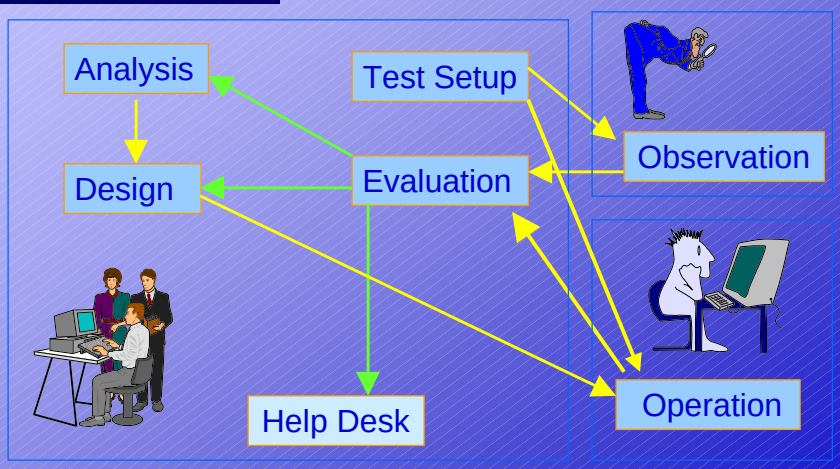
- User: video recording, think aloud
- Facilitator: identify user confusion
- Observer: log user intention
- *ErgoLight*: capture user actions
- *ErgoLight*: capture product modes

Backtrack

- Sync by time stamp

13

Teamwork support

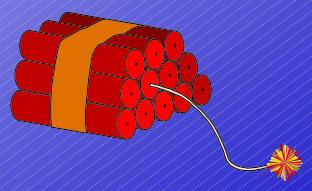


14

Understanding user errors

Safety critical systems

- Air traffic
- Nuclear
- Safety
- Health care
- Military



15

Remote Testing

- User difficulties
- Reasons
- Severity/ cost

16

Computer Aided Usability Testing CAUT

- Identify user confusion
  - User delay
  - Help, Cancel, Undo activation
- Identify user intention
  - User selects from task breakdown
  - Incentive - resolve problems

17

Getting started

- Download
  - Evaluation versions, documentation
- Evaluation version
  - Instrumentation, local recording
- Express testing
  - Backtrack and operation profiles
- Remote testing
  - Identify recurring user errors

18

ErgoLight

Thank you for thinking

USABILITY

19

Measuring The Quality of GUI Usability

Avi Harel

***ErgoLight*™** Ltd.

Traditionally, Graphical User Interfaces (GUI) are designed based on style guides and validated with reference to the GUI specification, which formalizes the GUI design. Often, end users experience difficulties in using GUIs that are derived from specification. The reason for this is that design based on style guides is often not suited for a particular application and that GUI designers often fail to anticipate the user's problems. GUI developers are challenged by the need to understand the ways end users actually operate the application. Accordingly, GUI validation should rely on record of the Computer Human Interaction (CHI) rather on the GUI specification.

Common practices of usability validations are by manual procedures, using questionnaires, video recording and observations. The measures of operation obtained by these techniques are subjective and they suffer from low reliability and low validity. The procedures involved in manual validation are lengthy and expensive.

In order to decide that an operational procedure should be changed, the measurement of operability should be objective, valid and reliable. Objectivity may be obtained by using a tool that collects data consistently, independent of the tester's observations. Validity may be obtained by observing real end users while performing real tasks in their real working environments, using measures that express organizational needs, such as operation costs. Reliability may be obtained by statistics of the measurements of the operational procedures and of the user's failure mode. Obviously, automation is required for achieving objectivity and reliability. Nevertheless, methodology is essential for obtaining face validity.

A common measure of GUI operability is the "user profile", typically consisting of an array of frequencies of operation of either GUI components or operational procedures. User profiles of these types allow the developer to identify those components and procedures that end-users prefer over their alternatives. An improvement of the user profile may be obtained by changing the measure, by summing up the time that users spend while operating each of the components or procedures. This measure is preferred over mere counting, because it expresses the real costs of operation.

A user's profile that describes the user's operation is not sufficient for GUI validation. About 50% of typical data entry procedures are wasted because of user's errors. The situation is even worse for education and utility applications, which are not operated frequently. Typically, users of such an application waste most of the operation time trying to figure out the basic functions, the concepts underlying the application, the meanings of terms used in the application and the procedures required to actuate the user's tasks. For this reason, the user's profile should include not only information about the operational procedures that users perform, but also information about the procedures that user's fail to perform.

While the user's operational procedures may be measured transparently, based on the user's actions, measuring the user's failure modes require elicitation of the user's intention, through dialog with the end user. Besides the user's intention, the dialog with the end-user provides information about deficiencies in the user's documentation, on-line help, the training program and also about error prone GUI components and operation modes. Statistics of repeating failure modes may provide measures of the costs of design deficiencies.

ErgoLight Ltd., 6 Giv'on St., Haifa 34335, Israel

Tel: +972-4-826-3012 Email: info@ergolight.co.il
Fax: +972-4-825-8199 HTTP://www.ergolight.co.il

Outsourced or Out of Luck

By Eric J. Bowden
Systems Engineer, KeyLabs Inc.
Phone: (801) 226-8200
Fax: (801) 226-8205
eric@keylabs.com
www.keylabs.com

Planning and Communication are key to a successful outsource testing relationship

Outline:

- When is outsourcing software testing appropriate?
- Test plan development
- Picking an outsource partner
- Pitfalls in outsourcing

When is outsourcing appropriate?

- Resource Constraints
- Specialized Expertise
- Independent Results
- Temporary (non-permanent) Need
- Cost Containment
- Strategic Direction

Test Plan Development

- Must be objective-based
- Test tool and methods evaluation
- Cutting up the pie

Picking a Test Lab Partner

- Skill set and training
- "Try before you buy"
- Relationship, relationship, relationship

Common Pitfalls

Without a clear vision, measurable objective and continuous involvement, outsourcing will produce less-then-desirable results.

Common Pitfalls

1) Waiting until forced to outsource
2) Assuming shared vision
3) "Manure Principle"
4) Lack of commitment
5) Clear, singular objective
6) Minimal communication

Outsourced or Out of Luck

By Eric J. Bowden
Systems Engineer, KeyLabs Inc.
Phone: (801) 226-8200
Fax: (801) 226-8205
eric@keylabs.com
www.keylabs.com

Outsourced or Out of Luck

Consigning an independent testing partner can help you attain your QA goals when budgets are tight and resources are scarce.

By Eric J. Bowden
Systems Engineer at KeyLabs Inc.
ebowden@keylabs.com
801.226.8200

Abstract

With so much work to be done and not enough hours in a day, hardware and software companies are deciding that it is more cost effective for them to focus their resources on the things that they do best. This sometimes means finding someone else to make sure that the product meets design requirements. Outsourcing your QA testing can be advantageous, but simply having good intentions will not guarantee that your outsourced relationship will be a successful one. This paper will help you decide when outsourcing your testing needs is appropriate and will also help you understand the common pitfalls when dealing with an outsourced partner.

"A man's got to know his limitations." This trite sound bite from a *Dirty Harry* movie can apply to both police work and software development. Good intentions alone are not enough to ensure the quality requirements for your software projects. It takes planning and continuous involvement. Understanding your own limitations will inspire you to search for alternatives. For many companies--tight on resources--this means consigning a partner to help in the QA process. KeyLabs has spent years helping ISV/IHV's with their testing and QA requirements. [And while it may appear self-serving to write a self-promoting article on outsourcing, the real] intent is to educate potential customers on how to make an outsource software testing relationship successful. I have worked with many customers as Lab Director at KeyLabs. I have also found that the better an outsourcing customer understands what they are getting themselves into, the more likely they are going to have a successful and productive experience.

The two areas that I will cover in this article include:

- When is outsourcing software testing appropriate?
- Common mistakes in dealing with outsource labs:

When is outsourcing software testing appropriate?

Resource Constraints:

By definition, dealing will scares resources is what business is all about. If you find yourself in an organization with copious resources, then count your blessings, but don't plan on it lasting forever. That kind of bounty is an open invitation for competitors, which have the redoubtable tendency of reducing your margins. When this happens, companies are forced to "make due or

Resource constraints can take many forms:

Lack of Hardware--The first, and one of the more obvious forms, is a lack of hardware and infrastructure. Some companies simply do not have enough hardware to adequately test their products. For them test methodologies are tailored to existing equipment—not to a requirements document. In these situations, a quality assurance engineer takes the list of "all things we might

test" and limits it to "all things we can test based on the hardware we have available." Pruning the list of test cases in this way can be dangerous when the undiscovered defects are only exposed on large installations or under heavy load. This is especially important for client/server applications and applications that are to support many users.

In some cases, you may have plenty of hardware to do your job, but lack a key piece of equipment to adequately test your product. This can be anything from an esoteric test tool to a common, but expensive piece of the test bed infrastructure. For example, over a year ago, a customer came to KeyLabs wanting to create a test network with 100 users dialing into a server. Not many companies have the 100 computers required to run a test like this. And even if they did, they probably would not have the phone simulator, remote access server and modems to support this kind of network. Outsourcing that portion of the testing that requires specialized equipment makes particularly good sense if the limited amount of testing you will be doing does not justify the cost of purchasing that piece of equipment. An outsource testing house is more likely to amortize the cost of expensive test equipment over many customers than you will be. It all boils down to do you rent or buy a water-ski boat if you know you will only be taking it out once or twice a year. Sure the rental price of the same boat might be exorbitant, but it is a lot cheaper than buying it your self and having it sit idle most of the year. And when you factor in what your money could be earning if it weren't tied up in capital assets.

Lack of People—Streamlining. Downsizing. Restructuring. Call it what you want, it usually means fewer people doing the work that many were doing before. However, many companies are taking this opportunity to re-focus on their core competencies—committing corporate positions only to those key technologies that define a company and then outsourcing the rest. Projects deemed "not part of the core business" are not getting funding for in-house execution. What this means is a software developer will focus on making software. Everything else, including testing and QA, will be outsourced. Contracting with a testing house can pick up the slack created by downsizing.

Deadlines—We've all heard the term "moving at the speed of business." This rapid movement means that deadlines and delivery dates might not afford you the time required to outfit a lab and ramp up. In these situations, an outsource testing partner can help smooth out the peaks, making your job more manageable.

Specialized Expertise

Another reason you may decide to outsource your software testing is to benefit from the specialized expertise of another organization. This may seem obvious, but let's spend a little time here. Hopefully, nobody understands your product as well as you do. However, what happens when your product interacts with other products that you have no control over? In cases like these, it pays to find somebody that already has the needed expertise. For example, we had a customer that made a product that was to run and be certified on every server platform available. These server platforms included many flavors of UNIX, Windows NT, and NetWare. This company had limited NetWare experience. Because of KeyLabs extensive background in NetWare, KeyLabs was able to provide both developmental and certification help to this customer in a much more timely manner than if they tried to do it themselves. Building specialized expertise in house can be costly and time consuming, especially when that expertise accounts for only a small portion of your overall revenue.

Independent Results

[Twelve years ago, I worked for a university doing departmental IS/IT. The scope of my job ranged from fixing PCs to hiring part-time dataentry personnel. Most of my time was spent programming and maintaining the 300,000-entity database on a mainframe. Since I was the only database administrator for this department, I had to write and test my own code. While I know

that I never intentionally wrote bad code, it still would have been valuable having an independent 3rd party make sure that my programs were doing what they were supposed to.]

Many software engineers and, indeed, many customers at KeyLabs believe that their code "doesn't need to be tested." However, the perspicacious QA manager realizes that their products not only need to be tested, but in many cases, need to be tested by an impartial third party—whether that party is inside or outside the company. An outsource test lab offers instant impartiality, primarily because they are not ego involved with the code they are testing. Their sole job is to expose defects, and in many cases they are rewarded for each bug they find. An outsource test lab comes to the testing event with no preconceptions. In other words, they aren't saddled with notions of what the product is or is not. They simply follow the test cases and report inconsistencies. I have seen junior technicians out perform their mentors in shear number of bugs found, simply because the technician did not try to second guess the application or to devise a workaround to a problem. This independence is critical for quality software testing.

Another reason independent results may prompt you to consider outsourcing is that benchmark and performance numbers are only as good as the "perceived credibility" of the source. Understandably, most companies have a "Six Million Dollar Man" complex about their products—better, stronger, faster. This leaves marketing departments with the responsibility of proving to potential customers that their product is "better, stronger, faster" than the competitions'. Performance testing executed by the vendor will always be under suspicion. Outsourcing to an independent lab lends credibility to the report.

Finally, sometimes your customers demand independent validation of your technology. For example, we had a firewall customer bring their firewall vendor into our lab to prove that it could support 70,000 simultaneous connections. Fortunately for the firewall developer, we were able to provide 3rd party validation that the firewall was performing as expected.

Temporary (non-permanent) Need

Even though you might be adequately equipped for your on-going, day-to-day testing, you may not be prepared for the occasional peaks and unexpected developments. For instance, recently many corporate testing organizations have received the additional responsibility of testing for Y2K compliance. This is in addition to their normal responsibilities. Outsourcing is a viable alternative to hiring for a temporary need.

Cost Containment

When you outsource, your costs become very visible. And because of this visibility, they are much easier to budget for. How much does it cost to hire, set up, and train a new employee? At a previous company I was allocating 36% over and above a persons salary. And this didn't include things like heating/air conditioning, office space rental, and equipment depreciation. As a QA manager, you might be tempted to say "I can hire a temp for $20 an hour." However, when you add all the hidden costs (i.e. a PC, time spent training and managing, floor space, etc.) of that worker, you will probably find that you are not any better off. Additionally, you managers will agree, it is often easier to expense a project to an outside contractor than it is to add head count to your department.

Strategic Direction

Finally, you may find that outsourcing is part of your company's strategic direction. There are companies that outsource everything that is not part of their "core competencies." What this means is, if company *X* is in the business of making software widgets, then only internal corporate resources are dedicated to developing the best software widgets on the market.

Everything else is contracted out including support, order fulfillment, and yes, even QA testing. We have seen many companies that outsource all their product testing. These companies typically have a clear strategic vision of what they want to accomplish and are committed to the outsource partnering relationship. Some of the value in partnering with an outsource lab is that it allows them to focus their engineers on developing new technologies instead of using them to test and maintain legacy applications.

The first half of this article talks about why you would want to outsource your software testing. The remainder of this document will focus on common pitfalls in dealing with an outsource software testing house. In reality, the points that follow are the most important things to keep in mind when considering a move to outsourcing.

Common Mistakes in Dealing with Outsource Labs

Too many times developers see outsource testing as a magic bullet that is viewed as a way of reducing pain or increasing bandwidth of already tight resources. Without a clear vision, measurable objectives, and continuous involvement outsource testing will produce less-then-desirable results. The following are pitfalls that I have seen that have resulted in an unsuccessful outsourced relationship.

1) **Assuming the outsourced lab shares the same vision you do.** Common vision is shared only after time and involvement. No contract test lab will intentionally try to mess up your test. The more you can communication your desires and objective with the lab the more their results will be in line with your expectations. When testing with an outsource partner, evaluate them on process performance as well as testing performance. This means looking beyond the technical qualifications of the test engineer. Do they have processes and procedures for ensuring that you will be satisfied with the final results? Do they have a mechanism for reporting on the status of a project? Do they communicate information in a way that is beneficial to you? Are they capable of adjusting to your changing requirements without re-bidding the entire project? Finally, when a screw up occurs, did they take ownership, fix it, and move on or did they try to pass blame?
2) **The Manure Principle** comes from the idea of taking all the horse manure, throwing it over the fence, and expecting to return in a week and find a beautiful rose garden. Throwing a disorganized project over the fence with no involvement and expecting successful results is bound to result in frustration and disappointments for both you and your outsource partner. True, contract labs can help ease your pain by sorting through the chaos that you call your job. Just make sure that they know what is expected of them. If you expect them to sort through the incomplete test plans and make recommendations, then say so. Don't just give them some test cases and tell them to execute them, because that is exactly what they will do. The customer statement that sends up a red flag at KeyLabs is "I work like the military. I give an order and expect it to be obeyed." Typically, this phrase is spoken by someone whose only experience with the military was when they were giving orders to their GI-Joe with the Kung Fu Grip. Even Napoleon would run his instructions by the dumbest soldier. Only after that conscript understood the order completely would Napoleon issue the instructions to the rank and file.
3) **Lack of commitment to outsourcing.** Outsourcing needs both upper and lower level support. To many times I've seen customer technicians try to sabotage the process because they felt that outsourcing was going to cost them their job. If you sense or suspect reticence, try assuaging their doubts before the process starts. As a manager, this type of reluctance on the part of your test engineers does not help your situation, because you still have work that needs to be done and uncooperative employees.
4) **Setting departments at odds with each other.** Just as it is important to communicate your objectives with your partners outside the company, you need to make sure that the different departments inside the company share the same vision. There was an occasion where a customer came to KeyLabs to develop a benchmark. We worked closely with their appointed

contact to get the benchmark just right—making requested adjustments to the original test plan as we went along. When their contact was satisfied with the results, he presented them to his VP, who promptly notified us that the work did not meet the VP's expectations of the assignment. It is important to make sure that all internal departments share the same vision for each project otherwise support for outsourcing will erode and frustration will abound.

Finally, minimal planning and no communication are key to an *unsuccessful* outsource testing relationship. Most of the problems I have encountered have been due to breakdowns in communication. This was so important that at KeyLabs, we require, at a minimum, daily communication—even if nothing has happened that day. This level of communication eliminates surprises and uncertainty.

With all the things you have to worry about, it is nice to be able to share the workload with someone else. However, good intentions are not enough to ensure success. Partnering requires open communication and continuous involvement before expectations can be met.

KeyLabs invite questions or comments regarding this article. Please email your comments to ebowden@keylabs.com.

Software Emancipation Technology

presents

DISCOVER®

Quality Process-Control and Management

presented by:
Bruce Boes

Agenda

- The Problem
- DISCOVER
- Software Quality Process-Control and Management
- About SET

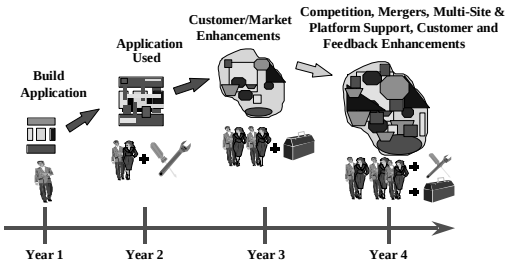
IDC States:

"The fact that applications are rapidly becoming larger, distributed and more complex is indisputable. There is also pressure to deliver applications far more quickly and with higher quality than ever before."

"The result is the 'Software Complexity Crisis' which suggests that the increasing size and complexity of applications will require a far more rigorous approach to application development"

Jeff Kinz
IDC #15523, March, 1998

Software System Evolution



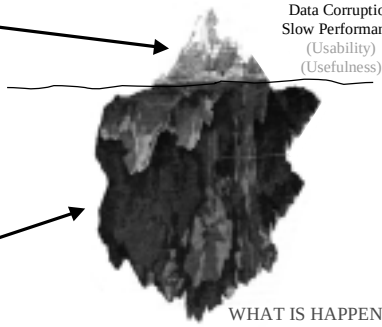
Change of computing paradigms and
Increasing Complexity of the product;
Result in a loss of Control

Software Quality

- 2 Types of Quality

External Quality
That Which Can be Seen by Customers
Traditionally Tested

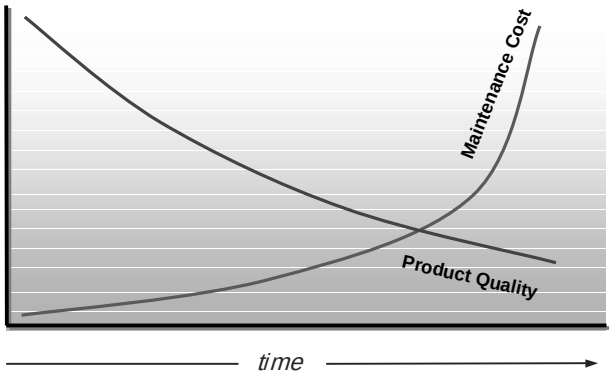
WHAT YOU SEE
System Crashes
Unexpected Behavior
Data Corruption
Slow Performance
(Usability)
(Usefulness)



Internal Quality
Program Structure
Coding Practices
Maintainability

WHAT IS HAPPENING
Lost Development Time
Fixes introduce new Problems
Lengthy Retesting
Loss of Competitiveness
Loss of Reputation

Costs are Out of Control



"As Software ages and complexity increases - quality deteriorates while maintenance cost spiral out of control"

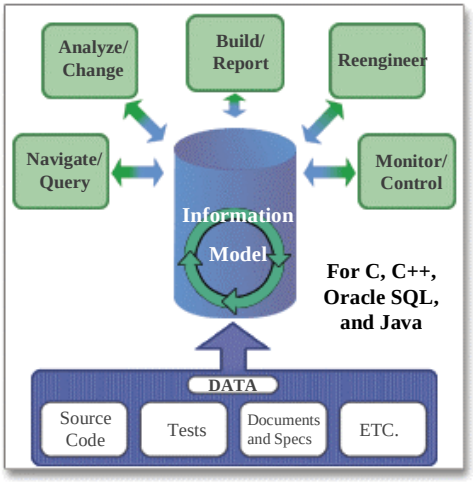
Software Quality

- Assertions!
  - External Quality is a Symptom
  - Internal Quality is the Root Problem
  - Poor Internal Quality = High Maintenance
  - In Order to Improve SW Quality:
    - Internal Quality Must be Addressed

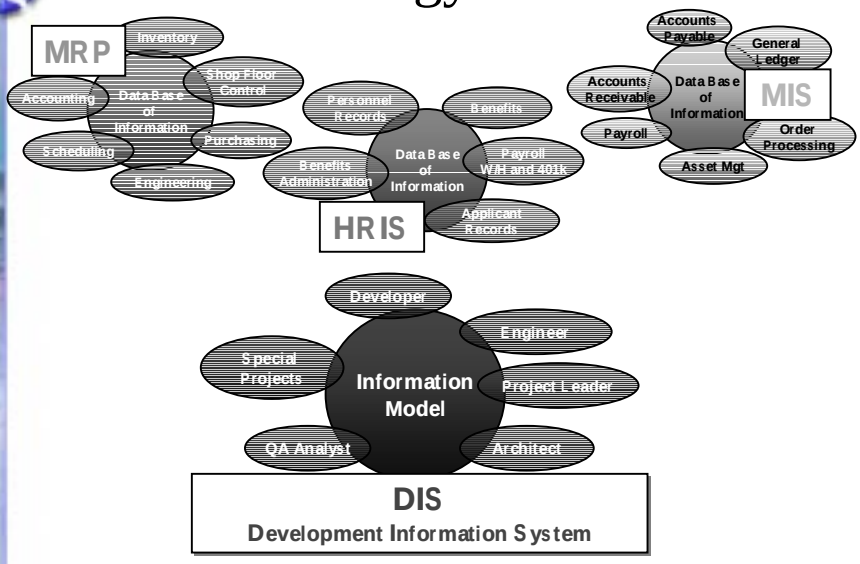
DISCOVER

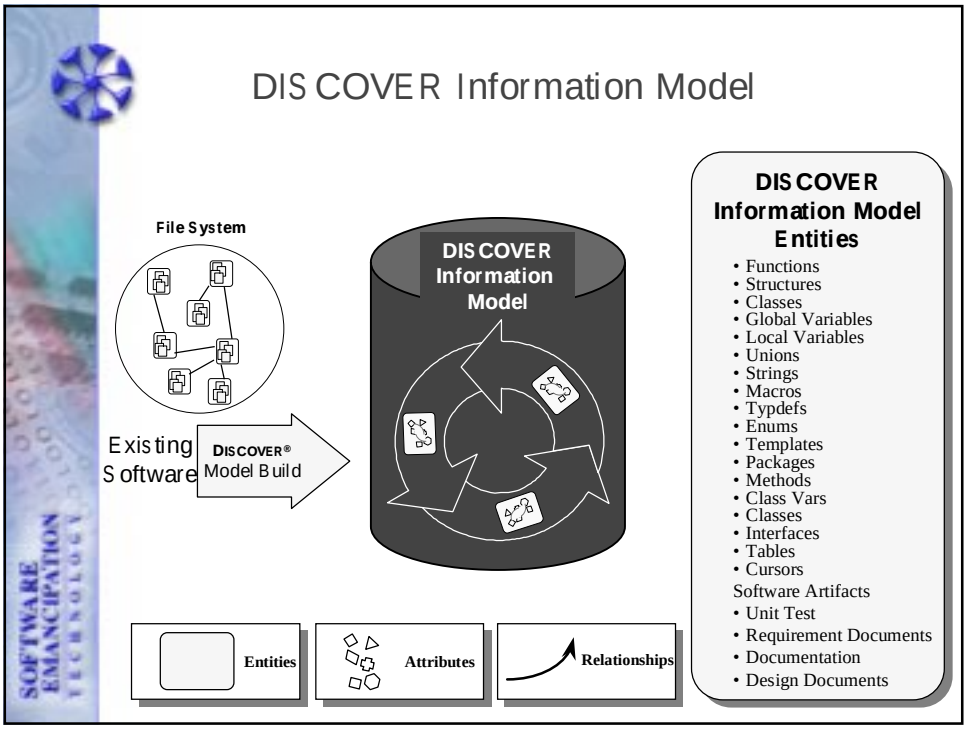
- DISCOVER is a Software Development Information System based on a single, comprehensive database of all the objects used throughout the software design, implementation, test, and maintenance process

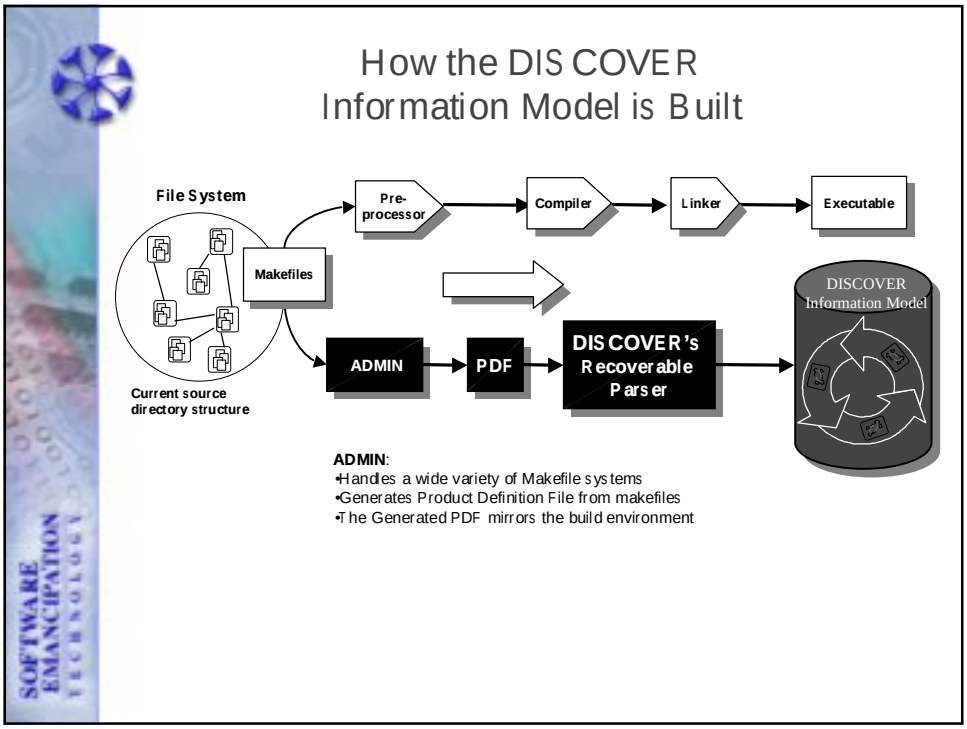
Information System Approach



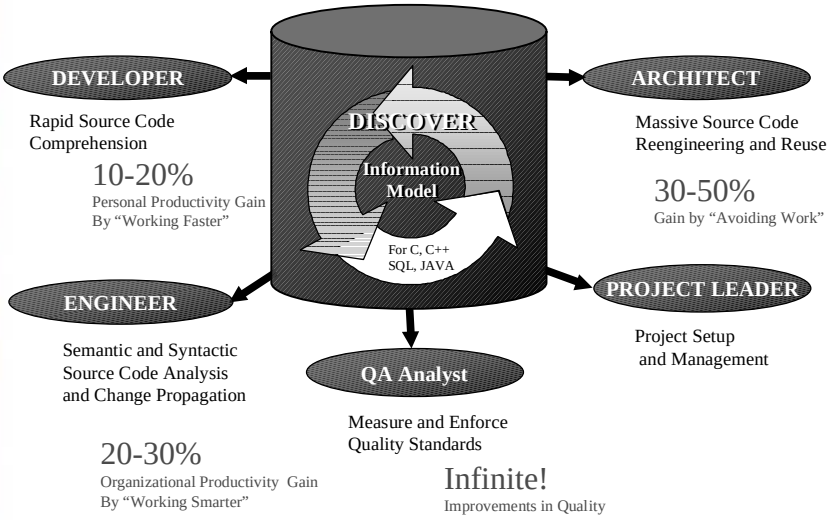
Technology Parallels




DISCOVER Information Model
File System
Existing Software
DISCOVER® Model Build
DISCOVER Information Model
DISCOVER Information Model Entities
• Functions
• Structures
• Classes
• Global Variables
• Local Variables
• Unions
• Strings
• Macros
• Typdefs
• Enums
• Templates
• Packages
• Methods
• Class Vars
• Classes
• Interfaces
• Tables
• Cursors
Software Artifacts
• Unit Test
• Requirement Documents
• Documentation
• Design Documents
Entities
Attributes
Relationships
SOFTWARE EMANCIPATION TECHNOLOGY


How the DISCOVER Information Model is Built
File System
Current source directory structure
Makefiles
Pre-processor
Compiler
Linker
Executable
ADMIN
PDF
DISCOVER's Recoverable Parser
DISCOVER Information Model
ADMIN:
•Handles a wide variety of Makefile systems
•Generates Product Definition File from makefiles
•The Generated PDF mirrors the build environment
SOFTWARE EMANCIPATION TECHNOLOGY

DISCOVER
Software Development Information System



SOFTWARE EMANCIPATION TECHNOLOGY

DISCOVER Focus

- Quality Process Control and Management
  - Capture Company Standards for Software Development
  - Attack Internal Quality
    - Root Cause Problem
  - Provide a means to monitor and measure quality complementary to testing
    - Entire base
    - Each new submission
  - Transform Quality Control to Quality Assurance
  - Provide Access for all Developers

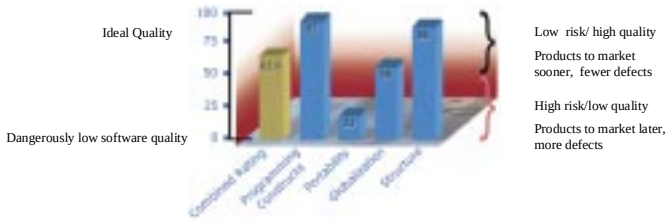
SOFTWARE EMANCIPATION TECHNOLOGY

Quality Process Control and Management

- Capture Company Standards for Software Development
  - Quality Filter Sets (QFS)
    - Document Company Coding Standards for SW Quality
    - Identifies Targeted Programming Constructs
    - Provides Pre-Defined Filter Sets with Domain Expertise
      - Programming Standards
      - Portability
      - Globalization
      - Structure
      - Complexity Metrics
      - Statistical Measurements
      - Y2K
    - Customizable and Expandable with QFS Master
    - Multiple Uses

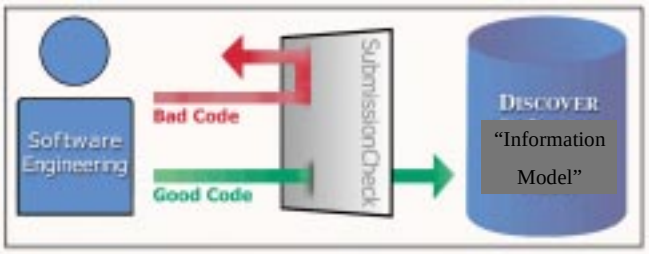
DISCOVER Quality Improvement Solutions

- Provide a Means to Measure and Assess Initial Quality
  - QAR
    - Establishes a Quantitative Baseline to Measure Improvements in Product Quality
    - Generates a QFS Based Quality Assessment Report (QAR)
      - » Graphical Charts
      - » List of Viewable Suspect Instances



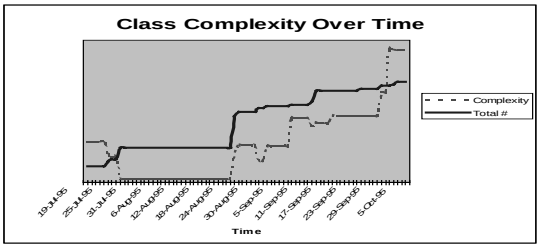
Quality Process Control and Management

- Provide a means to monitor Quality
  - Submission Check
    - Monitors Compliance with Established Standards via QFS *Before* Submission
    - Allows Process Repeatability, Scalability



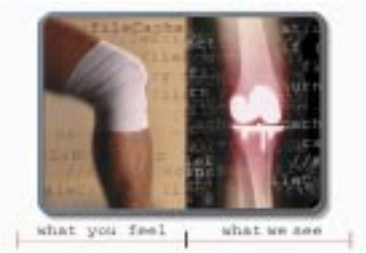
DISCOVER Quality Improvement Solutions

- Provide a Means to Monitor Quality
  - QA Cockpit
    - Regularly Generated QFS Based Quality Assessment Reports (QAR)
    - Provides Assessment of Trends Over Time
      - Graphical Charts



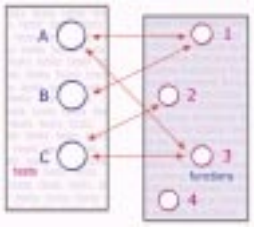
Quality Process Control and Management

- Software Quality Assessment Diagnostic Service
  - Software Quality Assessment Diagnostic
  - One Time Quality Assessment Report
  - For Use In:
    - Management Audits
    - Software Acquisitions
    - Proof of Concepts



Quality Process Control and Management

- QA Analyst
  - QFS Master Helps Establish Company Standards
  - TestCoverage Assures all Application Features have Tests
  - Turns Quality Control Departments into Proactive Quality Assurance Departments

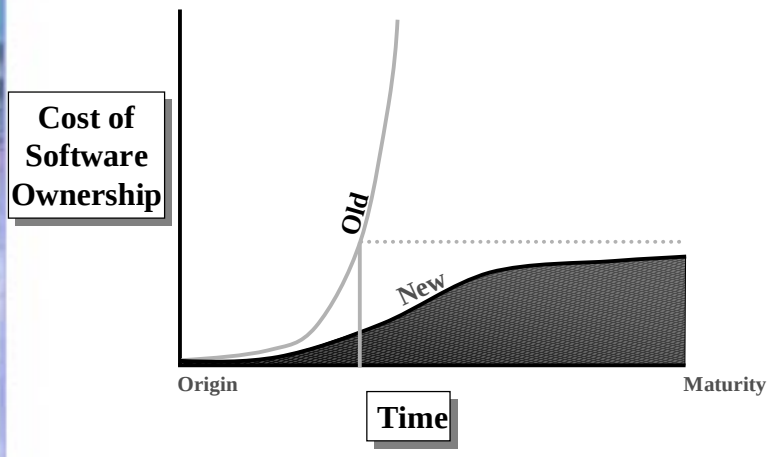


Accelerated Deployment and Ease of Use

- Developer Xpress
  - Concentrates on Comprehension
  - Provides Fast Ramp Up
  - Requires Only1/2 Hour Training
  - $3000 Per Seat

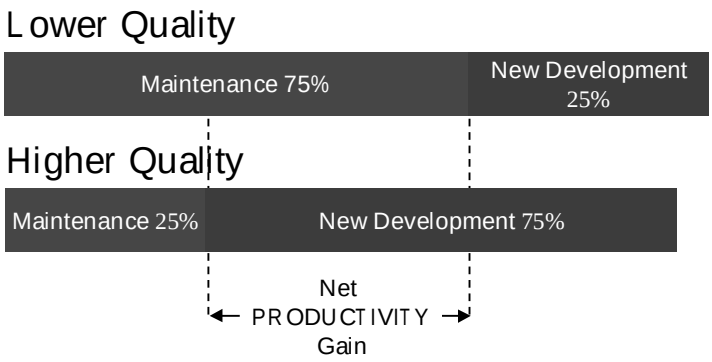
SOFTWARE EMANCIPATION TECHNOLOGY

Cost of Ownership is Stabilized



SOFTWARE EMANCIPATION TECHNOLOGY

Quality = Productivity



Improvement of Software Quality, Liberates Organizational Productivity

Summary

The new Quality Process Control and Management Solutions empower management with unprecedented control, giving them precise, predictable, and measurable information about the quality of their software as it is being developed.

Progress Software States:

"Progress Software is always searching for new technologies to gain competitive advantage. We recognize the benefit of DISCOVER's Quality Assessment Report (QAR) in providing us with a mechanism to measure and track the quality of our software on an ongoing basis, and to deliver high quality product to market on time."

Peter Sliwkowski
Vice President, Engineering
Progress Software
1/99

SET Confirms:

"For the first time in history, Software Emancipation is placing in the hands of software management, quality analysts, and engineers the capability for quantitative, unbiased measurement of software. This mechanism will provide the means to measure the quality of software during the development process, as opposed to merely testing it after integration."

Software Emancipation Technology

Recent Company Developments

- #2 New England Fast 50
- #11 National Fast 500
- 140 Customers
- 2600 End Users
- 50% Growth Forecast FY1999

SOFTWARE EMANCIPATION TECHNOLOGY

Automating Cleanroom Management

John Okanishi
Technical Consultant

Automating Cleanroom Management

- What is Clean Room Management
- Change and Configuration Management Benefits
- Endevor: The Solution

What is Cleanroom Management?

Cleanroom, as defined by IBM, is a "managerial and technical process for the development of software with ultra-high quality with certified reliability"

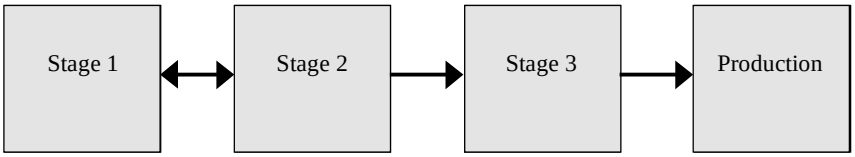
"*Management* of the Cleanroom process is based on an incremental life cycle in which development and certification are conducted in a pipeline of user-function increments."

Incremental Life Cycle

- Composed of site-defined requirements executed in the system environment, designed to accumulate changes into a final product.
  - Integration is top-down and continuous
  - Implementation of process key to ensure change process is error free
  - Each change increment implemented onto of the last, following the same procedures

Incremental Life Cycle

Automation, the Key to Success



- Components of change increment move together
- Required processes and tests can be automated -- ensuring accuracy
- Audit trail of all activities

Change and Configuration Management

The ability to manage, control and improve the process of building, changing and operating software systems.

CCM Benefits

- Desired business standards are applied…everytime
- Audit requirements are achieved
- Concurrent development is under control
- Prior executables recreated quickly and correctly
- Accurate source to executables trace

CCM Benefits

- Track all components of a change
- All required components are included
- Appropriate approvals performed in a timely fashion
- Reduced downtime
- Improve time to market

The Solution: Endevor

Endevor features:

- Version control
- Change control
- Configuration control
- Life-cycle modeling
- Concurrency protection
- Auditing and reports

Version and Change Control

Standards Enforcement

- Know who, when, why change made
- Ensure the right people get involved

Configuration Management

Standards Enforcement

- Creates executables when source is changed
  - Standards defined and enforced
  - Standard processes implemented
  - Repeatable procedures established

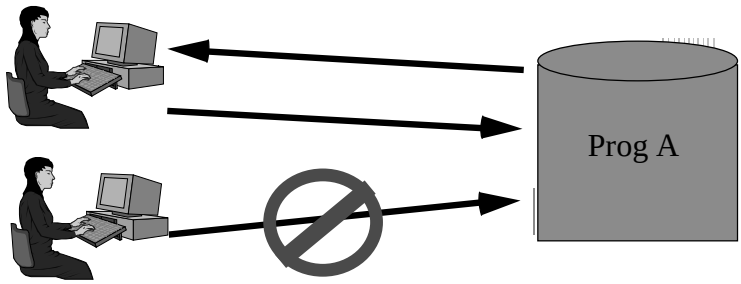
Life-cycle Modeling

Standards Enforcement

- Establish and enforce site-specific standards for life cycle administration
- For specific change increments, individual applications or across the organization
- Ensures process control

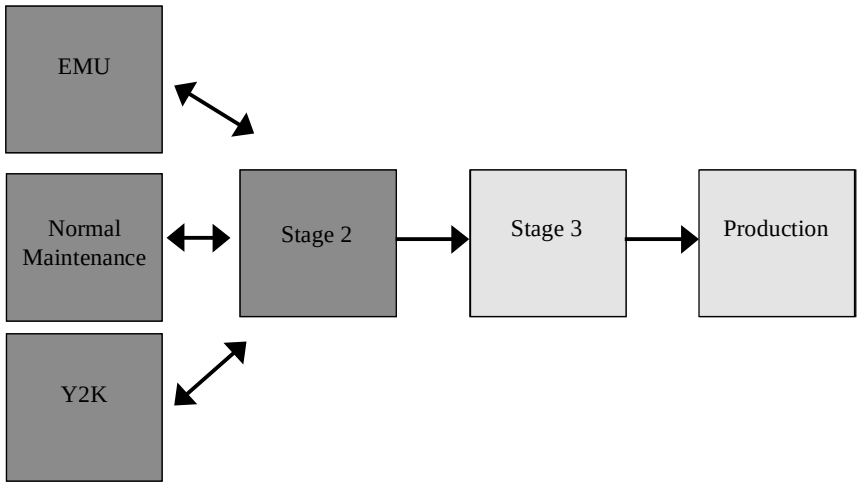
Concurrency Protection

Parallel Development Management



A signin-signout facility prevents unintentional overlaying of code

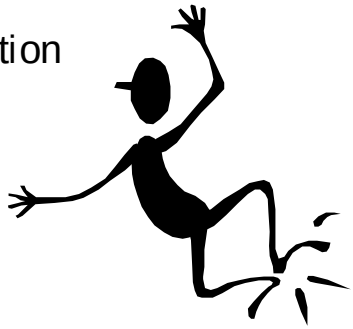
Concurrency Protection



Auditing and Reporting

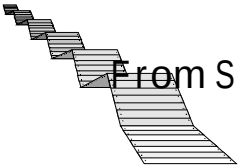
Use ad hoc or predefined reports with selection criteria

- Application logical classification
- Activity
- Application changes

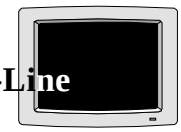


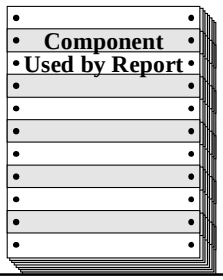
Auditing and Reporting

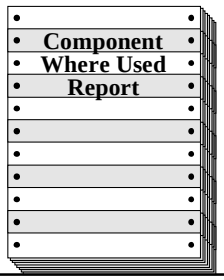
Impact Analysis

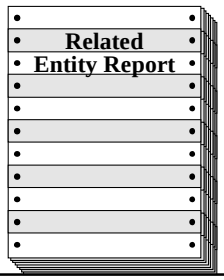


From Synchronize Reports or **On-Line**









What is Cleanroom Management?

Cleanroom, as defined by IBM, is a "managerial and technical process for the development of software with ultra-high quality with certified reliability"

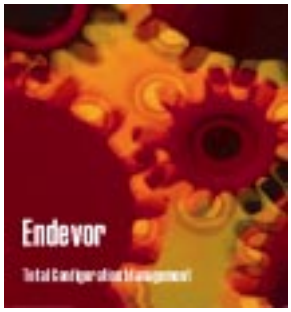
"*Management* of the Cleanroom process is based on an incremental life cycle in which development and certification are conducted in a pipeline of user-function increments."

Endevor Solution Suite

Endevor for MVS

Endevor for NT

Endevor Workstation



Endevor for UNIX

Unicenter TNG Change and Configuration Management Option

TESTMASTERS, Inc.

Y2K INDEPENDENT VERIFICATION & VALIDATION

YEAR 2000 READINESS DISCLOSURE NOT UNIVERSALLY

What is Y2K IV&V?

TESTMASTERS, Inc.

- INDEPENDENT
  - Conduct project review outside authority of established project management
  - Ensure project team reviews and understands all findings and recommendations prior to reporting
  - Preliminary and final written reports for project and senior management

What is Y2K IV&V? TESTMASTERS, Inc.

- VERIFICATION
  - Study the project management and testing methods to determine the effectiveness of the processes
  - Ensure appropriate measures are being taken to minimize the Y2K risks while staying on schedule and within the budget
  - Recommend process improvements, if necessary, to find an approach that strikes a balance between risk mitigation and cost

What is Y2K IV&V? TESTMASTERS, Inc.

- VALIDATION
  - Authenticate the correctness of the currently used methods to achieve the expected benefits
  - Review of the year 2000 test plans, testing procedures and test results, including specified date validations
  - Validation efforts include independent testing of the methods and techniques used by the project

Usefulness of the Y2K IV&V TESTMASTERS, Inc.

- Recommended by attorneys
- Required by some companies and state agencies
- Verify the exercise of "Due Diligence"
- Increase the comfort level of the project team, management and customers that the computer dependent processes will perform satisfactorily after December 31,
- Maximize Y2K project performance by providing unbiased advice on methods and review of results

The Y2K IV&V Process TESTMASTERS, Inc.

1. Planning and Initial Assessment
2. Evaluation
3. Collection and Analysis
4. Remediation and Test Review / Test Execution
5. Reporting

The Y2K IV&V Process — TESTMASTERS, Inc.

- Planning and Initial Assessment
  - Develop the scope and details of the IV&V effort including project duration & deliverables, reporting responsibilities and structure, client point(s) of contact,
  - Collect basic project materials, such as project plans, program inventories, coding/testing standards, Y2K program change standards, test strategies and plans, test results, status reports

The Y2K IV&V Process — TESTMASTERS, Inc.

- Evaluation
  - Review all project materials collected during Start-up & Initial Assessment with particular attention to missing or incomplete materials
  - Use standard checklists and the expert's understanding of "good standard practices" to determine, if or where, project practices, including the program strategy, are not adequate to accomplish project
  - Prepare preliminary findings on the effectiveness of the project management, program correction and testing methodologies in use

- Collection and Analysis
  - Collect and review detailed project plans, strategy and test plans for completeness
  - Ensure test plans contain all applicable test plan
  - Examine test conditions and test data to determine if they provide for adequate test coverage of the key test dates and exercise changes called for in the
  - Review standard policies and Y2K project practices to determine if they are applicable and consistently used

- Remediation and Test Review / Test Execution
  - Verify the program remediation by selecting a sample system and reviewing the sample for adherence to established guidelines
  - Report any problems found with, or enhancements process to project
  - Verify program testing by reviewing test plans and test deliverables to ensure that testing is following established guidelines and achieving adequate levels of
  - Conduct or witness test execution

- Reporting
  - Report preliminary findings after planning and initial
  - Log, track and report all project issues
  - Report the results of schedule monitoring including any variances and slippage
  - Provide periodic IV&V reports to project management and the project oversight authority
  - Produce Final “letter of certification” for the system(s)

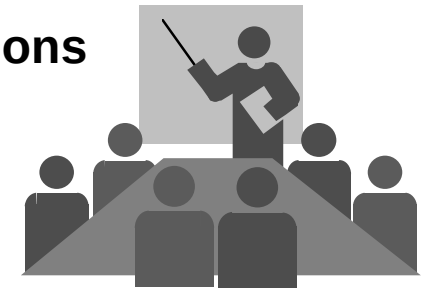
Testing Considerations for Web-Enabled Applications

Gerry Ocampo

Data Dimensions

Agenda

- Web Overview
- Test Considerations
- Test Types
- Conclusion



Data Dimensions

Types of Web Applications

- Business-to-Consumer
- Business-to-Business
- Catalog Applications
- Publishing Applications
- Transactional Applications

Data Dimensions

Web Architecture & Technology

- INTRANET
- INTERNET
- EXTRANET

Data Dimensions

Web Components

- Storefront
- Backoffice
- Security
- Database
- Other integrated systems

Data Dimensions

Test Considerations

- Quality Criteria
- Product Elements
- Project Factors
- Risk Assessment & Contingency Planning

Data Dimensions

Test Considerations

- Project Planning
- People and Skills
- Test Environment
- Tools
- Test Outsourcing
- Testing Types

Data Dimensions

Compatibility Testing

Determines how well an application works in conjunction with a variety of other products, on certain operating systems, across a broad range of hardware and component configurations, and when exposed to earlier versions of the product.

Examples and Approach

Data Dimensions

Functionality Testing

Determines the extent to which an application meets expected functional benchmarks through systematic validation of product features with respect to representative input data. This process can be as simple as checking to make sure that each piece works, or as detailed as checking a variety of scenarios and validating that all output is correct.

Examples and Approach

Data Dimensions

Functional Globalization Testing

Determines how well an application functions across a range of languages. Once functionality has been verified in the English language, localized versions are checked to determine whether particular language translations create failures specific to that language version.

Examples and Approach

Data Dimensions

Interoperability Testing

Determines, to a deeper extent than compatibility testing, how well an application works with a specific cross-section of external components such as hardware, device drivers, second-party software, and even specific operating systems and factory delivered computer systems.

Examples and Approach

Data Dimensions

Performance Testing

Determines how quickly an application executes a variety of events. This type of testing sometimes includes reports on response time to a user's command, system throughput, or latency. The word, "performance" has various meanings. Data Dimensions uses the word to refer to speed.

Examples and Approach

Data Dimensions

Load Testing

Determines how well an application functions when it is in competition for system resources. This competition most commonly comes from active processes, CPU utilization, I/O activity, network traffic, database storage, or memory allocation.

Examples and Approach

Data Dimensions

Stress Testing

Determines, to a deeper extent than load testing, how well an application functions when a load is placed on the system resources that exceeds their capacity. Stress testing can also determine the capacity of a system by increasing the load placed on the resources until either a failure or unacceptable product behavior occurs.

Examples and Approach

Data Dimensions

Scalability Testing & Capacity Planning

Predicts when future load levels will saturate the system. Performing load/stress testing will assist in accomplishing this but usually is supplemented by other techniques such as performance modeling techniques.

Examples and Approach

Data Dimensions

Installation Testing

Determines how well and how easily an application or component installs on a variety of platform configurations.

Examples and Approach

Data Dimensions

System Integration Testing

Determines, through isolation, which component of an application is the roadblock in the development process. This testing is beneficial to products that come together through a series of builds where each step in the development process has the potential to introduce a problem.

Examples and Approach

Data Dimensions

Test Automation

Determines how well an application functions through a series of automated tasks, using a variety of tools to simulate complex test data.

Examples and Approach

Data Dimensions

Conclusion

- Address the complexity
- Identify key areas by components
- Analyze test considerations
- Implement test strategy

Data Dimensions

Data Dimensions, Inc.

Gerry Ocampo
Gerry-Ocampo@Data-Dimensions.com
http://www.data-dimensions.com

Data Dimensions

Disciplined Software Testing Practices
&
Software Testing Certification

Dr. Magdy Hanna
International Institute for Software Testing

www.softdim.com

How Are We Doing??

- Testing is <u>the only</u> defect removal method practiced in 99% of software organizations
- Testing is the only activity that gets shortened or almost eliminated to meet budget and time constraints
- Testing has much defect removal efficiency than formal inspections.
- Most testing efforts focus on testing the functionality of the software

Practice #1:
Maintain User's Requirements

- You can't test what you do not know
- Requirement precession is a must
- The use of prototyping is NOT an excuse for not documenting requirements
- Use Case analysis is always works
- GUI models only functional requirements

Functional Vs Quality Requirements

- Although functional requirements seem to be most important to the user, most software disasters result from poor quality software.
- Quality requirements are the least understood requirements by both customers and developers.
- Although never talked about, quality requirements seem to be an "assumed" customer expectation.
- You can't achieve quality unless you define it.
- You can't evaluate quality unless you measure it.

The Sixteen Fitness-for-Use Factors

1. Configurability

the ability to configure the software for user's convenience. Examples are changing the system-user interface to use certain graphical symbols or changing the default use of directories.

2. Correctness

the degree to which the software conforms to the user's functional requirements.

3. Efficiency

the amount of resources required by the system to perform its intended functionality. This includes processing time, memory, disks, or communication lines.

4. Expandability

the ability to change the software to add more functionality or to improve its performance. This deals with the perfective maintenance of the software

The Sixteen Fitness-for-Use Factors

5. Flexibility

the ability to change the software to function in a different environment. This includes working under different database structure or to execute in a context different from that considered in the original implementation. This deals with the adoptive maintenance of the software.

6. Integrity

the ability of the software to protect itself and its data from overt and covert access.

*7. **Interoperability***

the ability of the software to exchange data with other software

8. Maintainability

the ability to change the software to fix errors. This deals with the corrective maintenance of the software.

The Sixteen Fitness-for-Use Factors

9. Manageability
the ability to properly manage the administrative aspects of the software. This includes resource allocation, configuration management, etc. It also deals with the availability of tools to support manageability.

10. Portability
the ability of the software to function on different platforms.

11. Reliability
the rate of failures in the software that make it unable to deliver its intended functionality.

12. Reusability
the ability to reuse portions of the software in other applications.

The Sixteen Fitness-for-Use Factors

13. Safety
the ability of the software to perform its functionality without causing any unsafe conditions.

14. Survivability
the ability of the software to continue execution, even with degraded functionality, after a software or hardware failure.

15. Usability
the ease by which the software can be learned and used.

16. Verifiability
the ease by which it can be verified that the software is working correctly.

Practice #2: Trace Requirements to GUI Components and Database Components

	Req 1	Req 2	Req 3	Req 4	Req n
Component 1	X		X		
Component 2			X	X	
Component 3	X	X	X		
Component 4					
Component n					

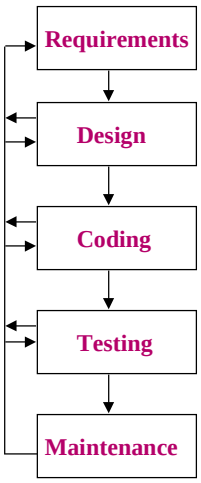
Practice #3: Trace Requirements to Test Objects

	Req 1	Req 2	Req 3	Req 4
Test Case 1	Passed/ Failed	Passed/ Failed		X
Test Case 2		Passed/ Failed	Passed/ Failed	
Test Case 3			Passed/ Failed	X
Test Case 4	Passed/ Failed			X
Test Case n				

Practice #4: Clearly Understand and Communicate The Difference Between Testing and Debugging

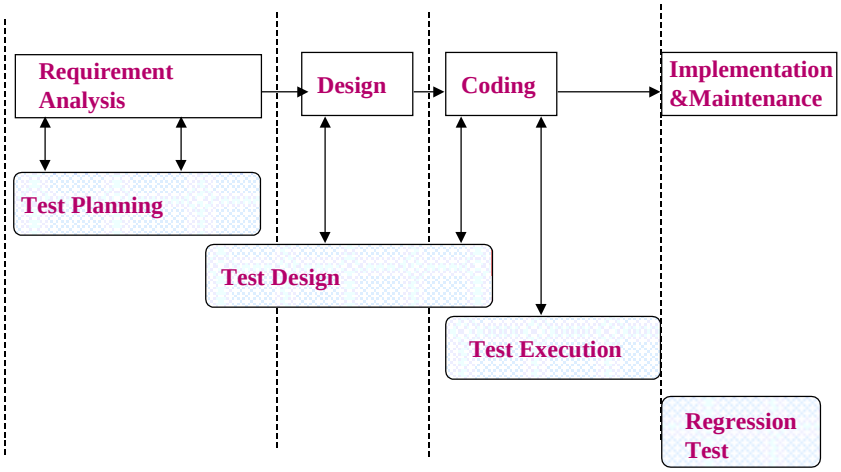
- Debugging is an ad hoc activity performed by individual developers to find and remove bugs from a program.
- Testing is a planned activity

Practice #5: Treat Testing As a Process NOT As a Lifecycle Phase

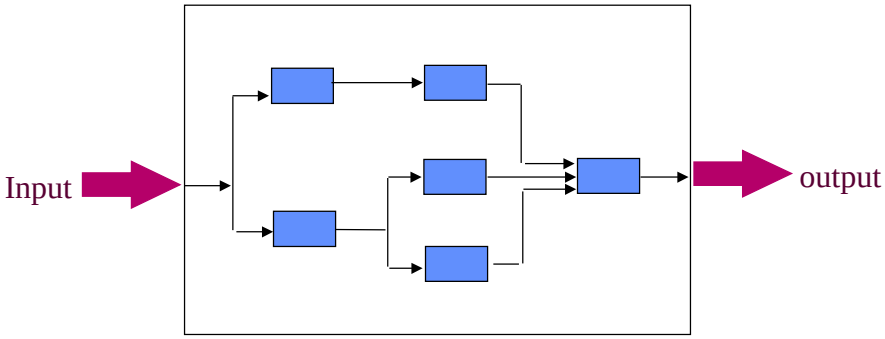


????

Testing As A Process



Practice # 5: Formally Define a Testing Process



Practice # 6:
Select Tools to Support Your Process

- Test planning tools
- Test management tools
- CM tools
- Test design tools
- Test coverage tools
- GUI testing tools
- Static analysis tools
- Test execution tools

Practice #7:
Establish a Mechanism For Improving The Process

- Measure, analyze and publicize success stories
- Revise process based on observations
- Re-train
- Re-measure
- Continuously improve the process

Practice #8: Develop A Test Plan

- Start planning for testing during the software requirement phase or once it is complete.
- Never combine planning, designing, and executing test in one phase.
- Test planning is the only way to perform successful testing and to deliver reliable systems.
- Test planning is about developing a test strategy.

Components of The Systems Test Plan

- Test Environment for systems test
- Objectives and scope of systems test
- Approach to systems test
- Staffing requirements and responsibilities for systems test
- Hardware, software, and network resources required for systems test
- Tools
- Tasks and their sequence
- Deliverables and documentation of systems test

A Test Plan Template

1. Identification
- 1.1 Systems Test Plan title
- 1.2 Application
- 1.3 Document locator
- 1.4 Other documents

2. Systems Test Environment
- 2.1 Site
- 2.2 Security
- 2.3 Submitting test items
- 2.4 Software and central repository
- 2.5 Supporting the testing environment
- 2.6 Backup/Restore procedures

3. Systems Test Objectives and Scope

4. Systems Test Approach
- 4.1 Test design
- 4.2 Cycle and sequence
- 4.3 Acceptance/Rejection criteria
- 4.4 Suspending/Resuming systems test
- 4.5 Reporting changes and corrections
- 4.6 Support
- 4.7 Order of systems testing

5. Staffing and Responsibilities

6. Hardware, Software, Network Requirements

7. Testing tools

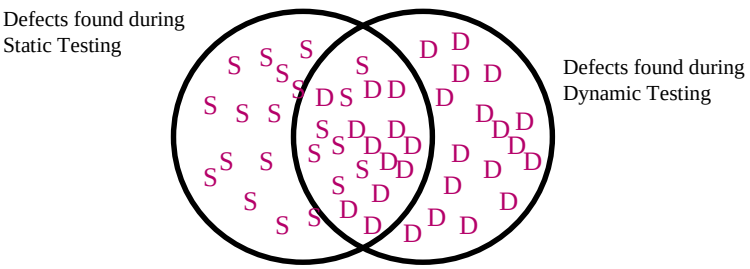
8. Systems Test Deliverables

9. Systems Test Tasks

Practice #9: Balance Between Dynamic and Static Testing

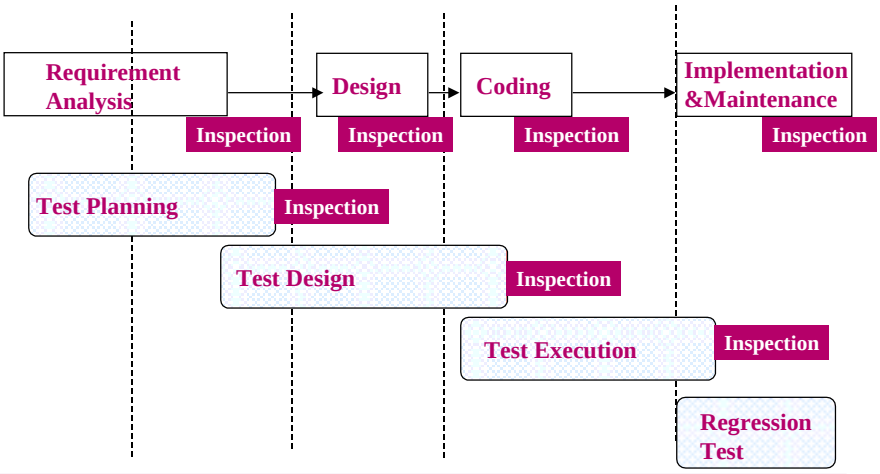
- Dynamic testing is the process of executing a program or system with the intent of finding error (*Glen Myers' definition*)
- Static testing is any activity that aims at finding defects by inspecting, reviewing and analyzing any static component of the software (code, documents, and models)

Balance Between Dynamic and Static Testing



Many defects that are normally left for dynamic testing can be found early through static testing

Where Does Static Testing Fit?



Practice #10: Acquire Formal Training and Certification

- Principles of Software Testing and Test Case Design Techniques
  - Requirement-based testing
  - Test design specification
- Test management
  - Planning
  - Scheduling
  - Resources
  - Risk Management

Acquire Formal Training and Certification

- Testing process
  - Defining the test process
  - Measuring the testing process
  - Improving the testing process
  - Testing mainframe applications
  - Testing client/server applications
  - Testing Internet and web applications
  - Testing object-oriented applications
  - Testing embedded systems

Acquire Formal Training and Certification

- Test Execution and defect tracking
  - Test scripting
  - Reporting
  - Defect tracking
- Test automation
  - Tool evaluation and selection
  - Scripting

Acquire Formal Training and Certification

- Requirement definition and refinement
  - Writing testable requirements
  - Requirement validation
- Static Testing (Inspections, Reviews)
  - Defining the process
  - Defining standards
  - Measuring the process
  - Improving the process
  - Defect data analysis

Why Get Certified?

- Testing has become a career; You can't have a career without certification
- Broaden your skills and improve your marketability
- Improve the image of software testing professionals
- Professional recognition of skills and experiences
- Formal training can only be acquired through Education-based certifications
- Be prepared to meet the today's testing challenges

Charting the Progress of System Development Using Defect Data

Copyright © 1999 Rex Black. All Rights Reserved.

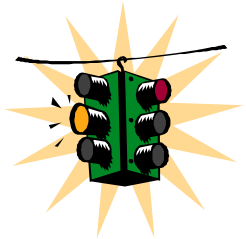
Rex Black
President and Principal Consultant

7310 Beartrap Lane

www.rexblackconsulting.com
Rex_Black@rexblackconsulting.com

Benefits of Charting Bug Data

- Assess Product, Process, and Project
  - Product stability
  - Defect removal trends
  - Root cause analysis
  - Bug management
  - Hot spots
- Communicate Status to Peers and
  - Summarize key facts and underlying trends
    Get the point across quickly.
    Manage key indicators, not the crisis du jour.



Case Study: SpeedyWriter

Phase	*Cycle*	*Start Date*	*End Date*	*Bugs Found*
Component test	1	7/19/99	7/25/99	25
	2	7/26/99	8/1/99	20
	3	8/2/99	8/8/99	5
Integration test	1	8/2/99	8/8/99	20
	2	8/9/99	8/15/99	15
	3	8/16/99	8/22/99	5
System test	1	8/16/99	8/22/99	10
	2	8/23/99	8/29/99	5
	3	8/30/99	9/5/99	0
First customer ship (FCS)	—	9/13/99	—	—

Key fields in bug tracking database for metrics

Bug report opened date — Bug root cause

Bug report closed date — Affected subsystem

Export the bug reports to a spreadsheet to prepare charts

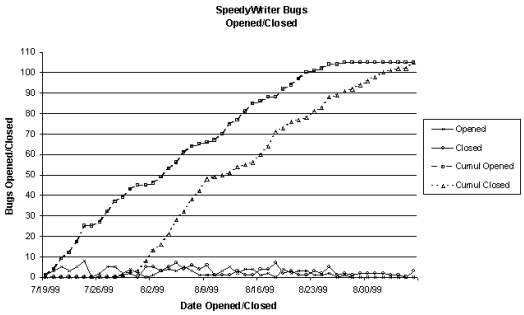
Bug reports approximate (linearly) underlying bugs (about 27% high)

Bugs Opened/Bugs Closed

- Cumulative opened curve becomes flat as system under test stabilizes and test system finds all the bugs it can identify.

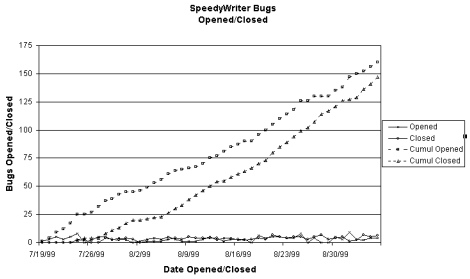
  Cumulative closed curve converges to cumulative open curve, closing the quality gap, as system under test approaches “customer ready” state.

  A cumulative closed curve that tracks the cumulative open curve



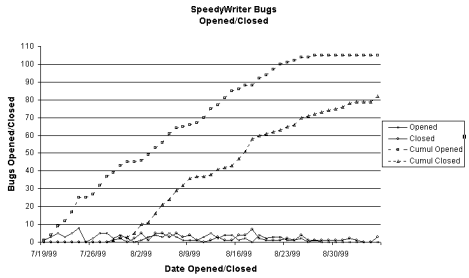
- Milestones in the project show up as changes in the shape of the curves.
- In a worksheet, use Excel `COUNTIF()` function for opened and closed bug reports by date and two running totals to create the chart.

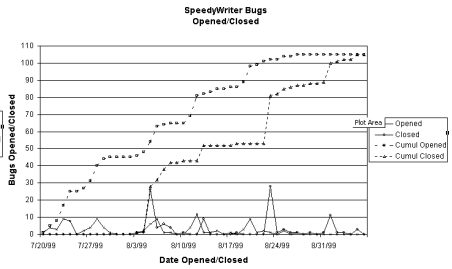
Opened/Closed:Trouble Indicators



Counterclockwise from left:

- Endless bug discovery (product never stabilizes)
- Ignored bug reports (persistent quality gap)
- Poor report management (sloppy open/close process)

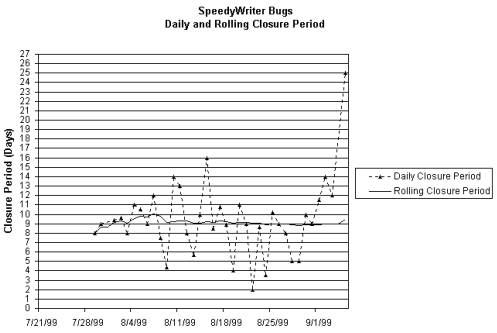




24-28 May 1999 **12th International Software Quality Week**

Closure Period

- Measures turnaround time on reported bugs (developer responsiveness):
  - Daily closure period is average turnaround for all bugs closed on a given day
    Rolling closure period is average turnaround for all project bugs closed to date
- A stable closure period implies a smooth rolling closure period and daily closure periods that are bounded at the top end by two or three times the test cycle duration (in this example, each cycle is one week).
- An acceptable closure period shows bugs moving to resolution in the time specified by project management.
- This chart measures the quality of the overall process, including development and test, for bug fixing.



Calculating Closure Period

1. Find the closure period for each closed bug report.

Microsoft Excel - Good SW OC RC SS CG Analysis.xls

File Edit View Insert Format Tools Data Window Help

D2 = =IF(B2>0,IF(C2>0,IF(ISNUMBER(C2),C2-B2,NA()),NA()),NA())

	A	B	C	D	E	F	G	H	I	J
1	Bug ID	Date Opened	Close Date	Closure Gap						
2	1	7/21/99	8/3/99	13						
3	2	7/21/99	7/30/99	9						
4	3	7/24/99	8/1/99	8						
5	4	7/23/99	8/2/99	10						
6	5	7/24/99	8/1/99	8						

2. Calculate the total closure period for all bug reports by date.

Microsoft Excel - Good SW OC RC SS CG Analysis.xls

File Edit View Insert Format Tools Data Window Help

B2 = =SUMIF('Closure Gap-Calc'!$C$2:$C$500,$A2,'Closure Gap-Calc'!$D$2:$D$500)

	A	B	C	D	E	F	G	H
1	Date	Closure Period	Cum Closure Period	Inactive	Cum Inacti	Daily Closure Perioc	Rolling Closure Period	
2	7/21/99	0	0	0	0	#N/A	#N/A	
3	7/22/99	0	0	0	0	#N/A	#N/A	
4	7/23/99	0	0	0	0	#N/A	#N/A	
5	7/24/99	0	0	0	0	#N/A	#N/A	
6	7/25/99	0	0	0	0	#N/A	#N/A	

Calculating Closure Period (cont)

3. Count the number of bugs closed on each date.

Microsoft Excel - Good SW OC RC SS CG Analysis.xls

File Edit View Insert Format Tools Data Window Help

D2 = =COUNTIF('Closure Gap-Calc'!$C$2:$C$500,$A2)

	A	B	C	D	E	F	G	H
1	Date	Closure Period	Cum Closure Period	Inactive	Cum Inacti	Daily Closure Perioc	Rolling Closure Period	
2	7/21/99	0	0	0	0	#N/A	#N/A	
3	7/22/99	0	0	0	0	#N/A	#N/A	
4	7/23/99	0	0	0	0	#N/A	#N/A	
5	7/24/99	0	0	0	0	#N/A	#N/A	
6	7/25/99	0	0	0	0	#N/A	#N/A	

4. Determine the daily and rolling closure periods.

Microsoft Excel - Good SW OC RC SS CG Analysis.xls

File Edit View Insert Format Tools Data Window Help

G2 = =IF(C2>0,IF(E2>0,C2/E2,NA()),NA())

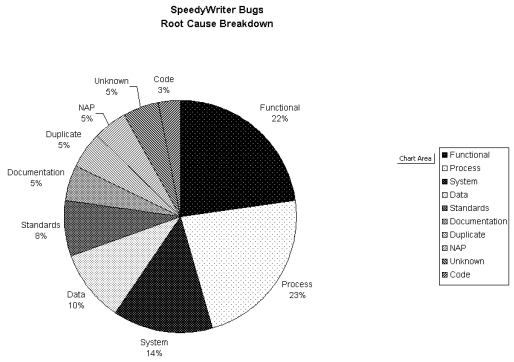
	A	B	C	D	E	F	G	H
1	Date	Closure Period	Cum Closure Period	Inactive	Cum Inacti	Daily Closure Perioc	Rolling Closure Period	
2	7/21/99	0	0	0	0	#N/A	#N/A	
3	7/22/99	0	0	0	0	#N/A	#N/A	
4	7/23/99	0	0	0	0	#N/A	#N/A	
5	7/24/99	0	0	0	0	#N/A	#N/A	
6	7/25/99	0	0	0	0	#N/A	#N/A	

Root Cause Breakdown

- Understand the underlying mistakes, miscommunications, and other errors that lead to bugs and bug reports.

  Allow for course-correction during the project.

  Provide information for long-term development process improvement.

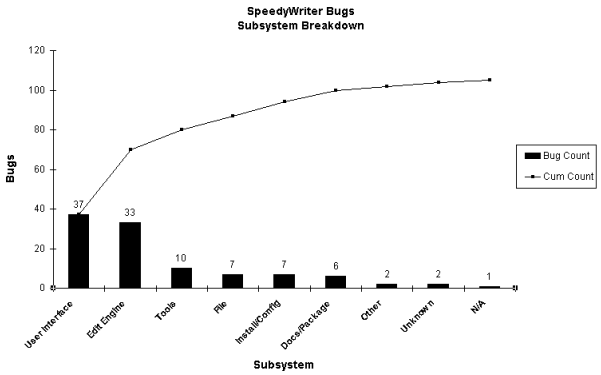


Subsystem Breakdown

- Analyze which portions of the system experience the most problems.

  Increase testing of the subsystems that produce the most bug reports (where there's one bug, there's another).

  Improve the development processes (including internal testing) for the subsystems that are most error-prone (preventing the additional bugs is



Charting the Progress of System Development Using Defect Data

Rex Black; President and Principal Consultant,
RBCS, San Antonio, TX

Key Words: Test management, development management, project management, bugs, defects, bug reports, metrics, development project, opened/closed, found/fixed, defect removal, closure period, root cause, subsystem.

Abstract

Aggregate defect data (bug reports), presented graphically, allow testers to give development, project, and executive management "dashboard" information they need to drive development projects to successful conclusions. I discuss four charts that distill meaning from test findings in terms of product stability, quality, bug repair, root cause, and affected subsystems.

Introduction

While bug reports are useful by themselves, you can discover some interesting facts by analyzing them in aggregate. Defect data form the basis of various quality control charts in a number of industries, and computer software and hardware are no exception. In this paper, I will introduc you to four simple yet powerful, sophisticated defect analysis charts. The charts provide information on project, process, and product quality. Using bug data, you can generate defect analysis charts that show patterns in the process of defect removal, the root causes of bugs, th effectiveness of bug management, and the parts of the product that cause the most problems.

These charts are also excellent for communicating with management for three reasons. First, test metrics, graphically presented, highlight and illustrate test results that are difficult to explain using the raw data, which usually consist of a thick stack of bug reports. Second, getting peopl to focus for an hour-long bug review is hard, but presenting and explaining four charts fits into the available attention span. Third, the perspective of these charts encourages the viewers to manage the project and process according to key indicators, rather than to resolve the crisis du jour.

Case Study

In the following paper, I use a case study approach to introduce these charts. The case study is a hypothetical software development project to implement a word processor called SpeedyWriter. I assume that the test team will execute a three-phased test effort as outlined in Table 1. The cas study is presented from the perspective of a person looking back on the project on September 5, 1999, which gives us the benefit of hindsight. However, these charts are at their most useful *during* a project; with practice, you'll be able to understand the trends when you see them developing. I've assigned pertinent data points somewhat randomly but to illustrate specific circumstances. The charts are representative, though cleaner and clearer than those from a real project.

Copyright © 1999 Rex Black. All Rights Reserved.

Phase	Cycle	Start Date	End Date	Bugs Found
Component test	1	7/19/99	7/25/99	25
	2	7/26/99	8/1/99	20
	3	8/2/99	8/8/99	5
Integration test	1	8/2/99	8/8/99	20
	2	8/9/99	8/15/99	15
	3	8/16/99	8/22/99	5
System test	1	8/16/99	8/22/99	10
	2	8/23/99	8/29/99	5
	3	8/30/99	9/5/99	0
First customer ship	—	9/13/99	—	—

Table 1: The Bug Location and Test Schedule for SpeedyWriter

The SpeedyWriter team uses a basic bug tracking database. To prepare the defect analysis charts discussed in this paper, the following fields are necessary:

- Report opened date
- Report closed date
- Root caus
- Affected subsyste

To prepare these charts, you can use any bug tracking system capable of collecting these four data points for each bug report, and exporting at least these four data fields into a file format readabl by a spreadsheet program. (I used Microsoft ® Excel to create the charts in this paper.) Of course, a useful bug tracking database must capture and report a lot more information than these four fields, but that's a topic for another conference.

In the following sections, I use bug reports-documented symptoms of problems observed by testers-as a numerical proxy for actual bugs-defects present in software or hardware. Because of duplicate bug reports, bug reports that turn out to be the result of test system or tester failure, and other noise in the bug tracking database, this is a liberal approximation. On five projects consisting of both hardware and software, I have observed defect overestimation by bug report counts between 26 and 32 percent, with 27 percent the most frequent value. Therefore, th approximation is defensible, and, assuming your bug reporting database captures root cause data, you can correct your estimates dynamically.

Opened/Closed

The first and most important chart shows daily and cumulativ bug opened and bug closed counts. It illustrates interesting trends in both defect location and repair, which often correspond to important events in the project timeline. It also shows the stability of the product, and th quality gap. Figure 1 provides an example of an opened/closed chart.

Copyright © 1999 Rex Black. All Rights Reserved.

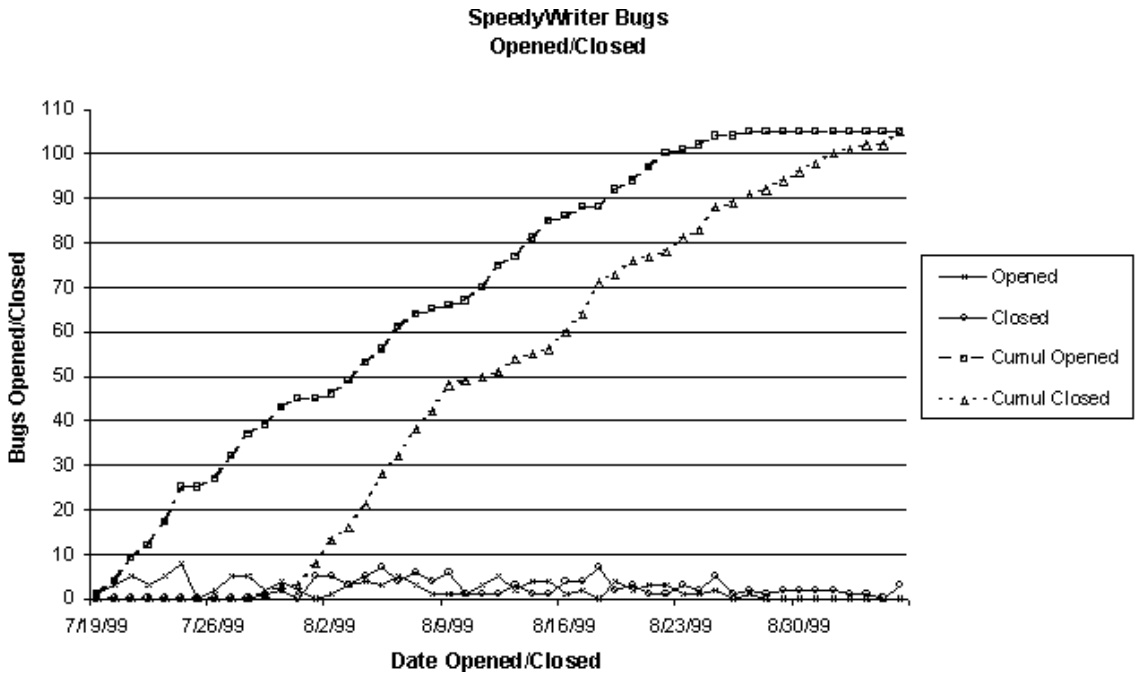


Figure 1: Ideal SpeedyWriter opened/closed chart

This information-rich chart provides answers to a number of questions. First, are you ready to ship the product? Since the number of bugs in a given program is unknown but is essentially a fixed number once development ends, test projects eventually hit a point at which further testing produces diminishing returns. When the cumulative opened curve—the top curve on the chart in Figure 1—levels off at an asymptotic limit, testing is usually considered complete, at least for th phase currently under way. (The asymptote actually indicates the fading of the test system's ability to find bugs in the current phase. Given a good test system, the bugs found represent th bugs most likely to torment the customer, and the fading of the test system is consistent with customer indifference.) Conversely, a cumulative opened curve that refuses to flatten indicates that you have plenty of problems left to find. On the chart in Figure 1, the limit was hit around August 23: the second cycle of system testing revealed few additional bugs, and then none were found in the final round of system testing.

Next, have you finished fixing bugs? Once development winds down, developers usually start to catch up with the problems. At about the same time, the cumulative opened curve starts to flatten. Consequently, the cumulative closed curve—the lower curve on the chart in Figure 1—begins to converge with the cumulative opened curve. Given the latency period that arises from the release management process, the closed date for a given bug does lag the developer fix date. If you want to add this, you will need a field to in your bug tracking database that tracks when bugs are fixed.

At a more general level, is the bug management process working? It is working well in this example: the closed curve follows right behind the opened curve, indicating that the project team is moving the bugs quickly toward resolution.

Finally, how do milestones in the project relate to inflection points, changes in the slope of th cumulative opened or cumulative closed curves? The overlapping of the phases in the exampl obscures this relationship somewhat, but often when you move from one test phase to the next, you will see a spike in the cumulative opened curve. Such rises are gentle in our idealized

Copyright © 1999 Rex Black. All Rights Reserved.

example, but these transitions can be dramatic and even downright scary on some projects. As developers move from writing new code or engineering new hardware to fixing bugs, you should see an upward turn in the cumulative closed curve. A "bug scrub" meeting, where the technical and management leaders of a project gather to decide the fate of all known bug reports, can result in a discontinuous jump in the cumulative closed curve.

Such a chart is easily prepared using the `COUNTIF()` function in Excel. Create a worksheet that counts, for each date during the test phases, how many bug reports were opened and closed. Then accumulate these numbers day-by-day. Finally, use the Excel chart wizard to create the chart.

To explore the use of opened/closed charts, let's look at three troublesome scenarios that represent those unpleasant projects in which all test managers eventually participate. All of these examples assume the SpeedyWriter testing schedule outlined in Table 1. For the first, imagin that during the system test phase the bug find rate remains high and refuses to level off. The opened/closed chart that results appears in Figure 2. Notice the deceptive leveling in the second cycle of system test (8/23/99 through 8/29/99), where the opened curve appears to flatten, only to leap up sharply in the third cycle (8/30/99 through 9/5/99). If the project team ships the product on September 13 as scheduled, they can expect many failures in the field.

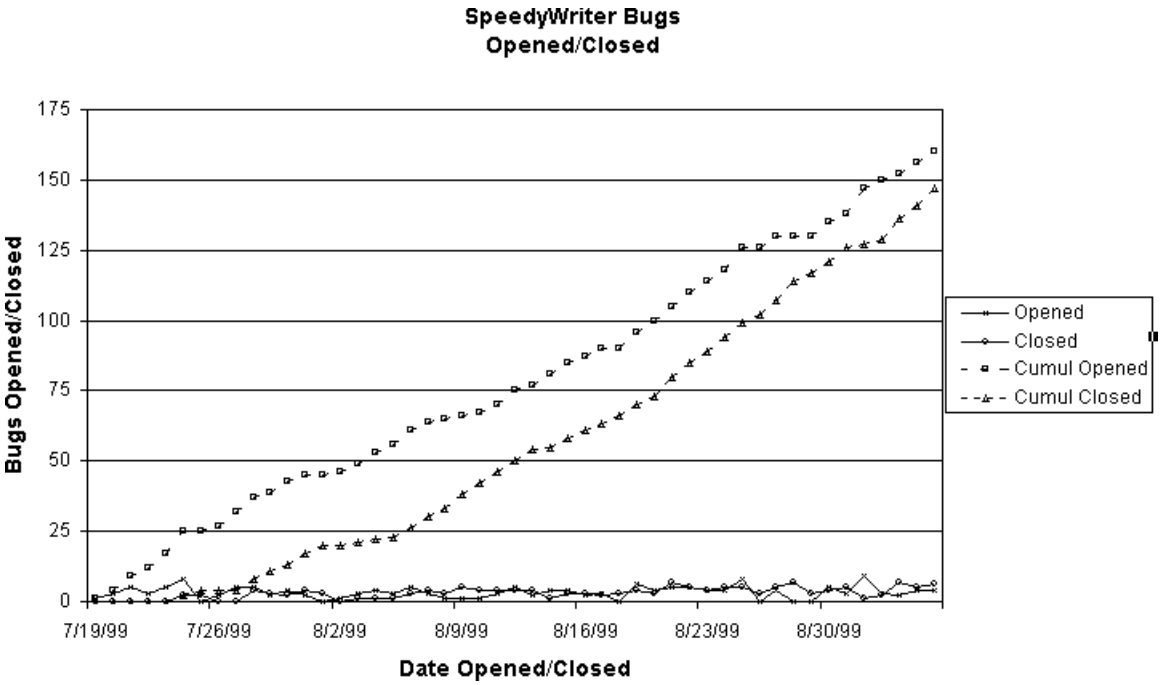


Figure 2: Endless bug discovery

For the second scenario, let's assume that development is ignoring some of the bugs. Th developers have convinced themselves that about 25 of the bugs that test has reported are beneath contempt and can be disregarded. Figure 3 shows the opened/closed chart for this scenario. Until about August 20, the chart in Figure 1 (the idealized example) and the chart in Figure 3 are not radically different: on that date, the quality gap is about 20 in the former chart, whereas it is about 30 in the latter. Ten bugs out of 100 one way or the other three weeks before the first customer ship is not a catastrophe. But as the second and third cycles of system testing unfold (8/23/99 through 8/29/99, and 8/30/99 through 9/5/99, respectively), it becomes clear that the gap is not narrowing. Unless you bring the pernicious bugs to project management's attention around

Copyright © 1999 Rex Black. All Rights Reserved.

August 23, the development team will deliver a *fait accompli*. Overturning their decision to ignore these bugs even a week later will require a slip in the delivery date.

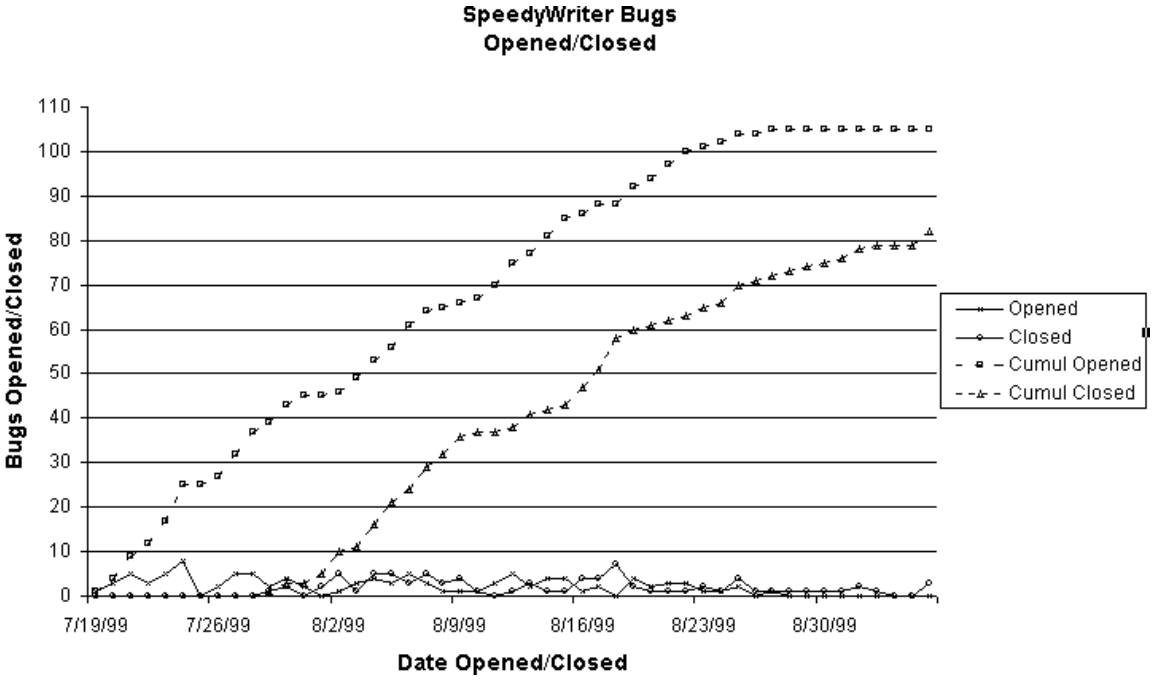


Figure 3: Ignored bug reports

Finally, suppose that the developers and the testers are both doing the right things at a technical level, but the bug management process isn't working. The development manager doesn't notify you when bugs are fixed and ready for confirmation testing, and you don't follow up with your testers to make sure they close bugs that pass confirmation testing. Also, testers don't bother to report bugs when they find them but instead wait until Thursday or Friday each week. Then they enter their findings, some dimly remembered, into the bug tracking database. Figure 4 shows how the opened/closed chart looks in this example. The trouble here is that you can't tell whether a jump in a curve represents a significant event or simply indicates that some people are getting around to doing what they should have done earlier. If you use this chart as part of your project dashboard, your gauge is jumpy.

Closure Period

While we're on the topic of bug report management, let's proceed next to the closure period chart, which shows the daily and rolling (project to date) closure period for bug reports. Closur period (a.k.a. closure gap) is complicated to calculate, but it has a simple intuitive meaning: th closure period measures development's responsiveness to test's bug reports. Daily closure period refers to the average number of days between the opening of a bug report and its resolution for all bug reports closed on the same day. Rolling closure period is the average for all closed bugs, including the current day and all previous days. Figure 5 shows the closure period chart for th SpeedyWriter project. As you can see, the daily plot tends to "pull" the rolling plot toward it, although the ever-increasing inertia of the rolling average makes it harder to influence as th project proceeds.

Copyright © 1999 Rex Black. All Rights Reserved.

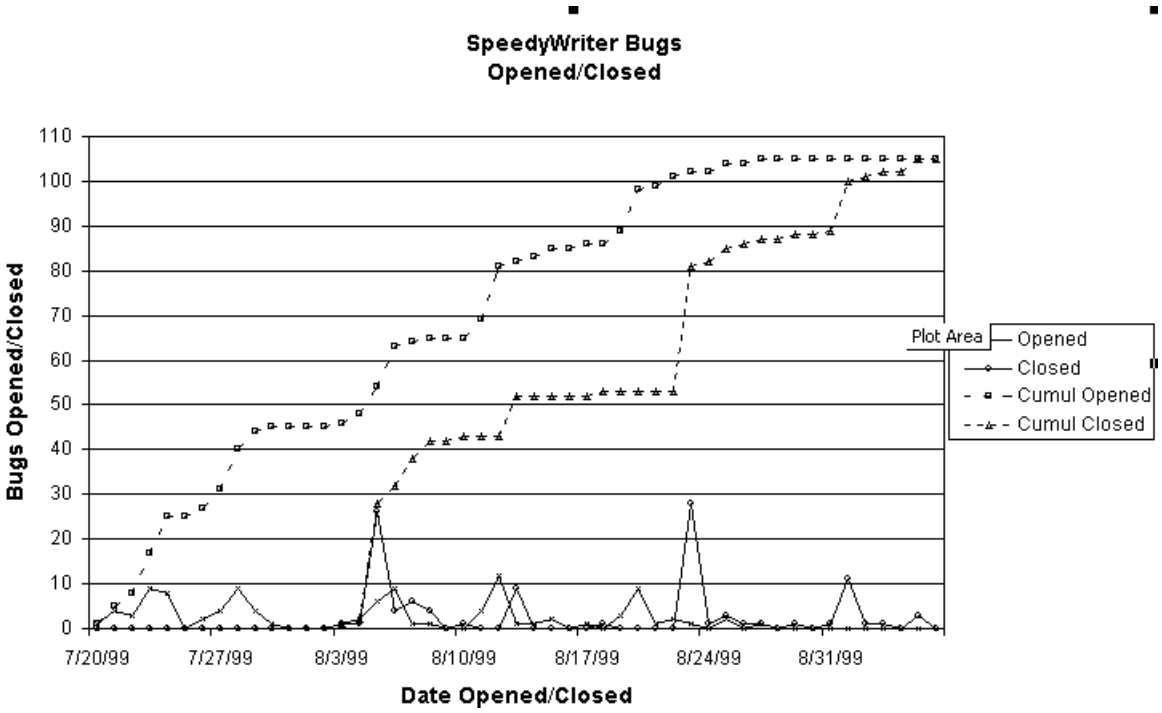


Figure 4: Poor report management

It's useful to look at closure period in terms of stability and acceptability. A stable closure period chart shows a relatively low variance from one day to another, with the slope of the rolling closure curve remaining almost constant and close to 0. In addition, the daily closure period fluctuates randomly around the rolling closure curve, staying within a few days in either direction.

On an acceptable closure period chart, both the daily and rolling closure period curves fall within the upper and lower limits set in the project or test plan for bug turnaround time. Although th pressures of the typical project make it hard to believe, there is indeed a lower limit for an acceptable closure period. Bugs deferred the day they are opened pull the daily closure curv toward 0, but the bug remains in the product. Furthermore, an acceptable daily closure curve does not exhibit a significant trend toward either boundary.

A closure period chart that is both stable and acceptable indicates a well-understood, smoothly functioning bug management process. The ideal is a low number with a downward or level trend since an efficient bug management process drives bugs through their state transitions to closure with all deliberate speed. The closure period in Figure 5 is stable and, if management is realistic in its expectations, acceptable. Bugs tend to get fixed in about a week and a half, which is a good pace if you assume one-week test cycles and formal release management.

Unlike opened/closed charts, there are not archetypal "bad" closure period charts that indicate particular types of failures in the development process. If the daily closure period shows too much variability, then that might indicate sporadic bug fixing, but it could also mean that a mix of simple functionality and complicated stability bugs are in the bug fix queue. An upward trend in the rolling closure period can indicate a slowing of the fix process, but it may also imply that development is focusing on the difficult bugs.

Copyright © 1999 Rex Black. All Rights Reserved.

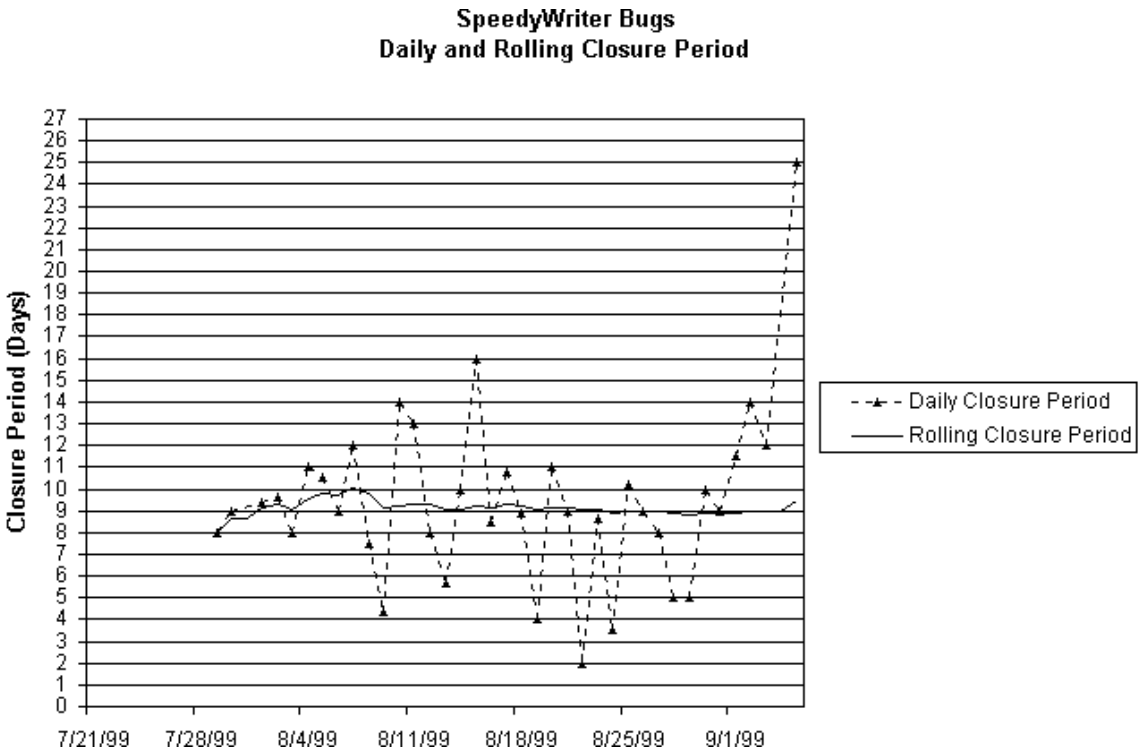
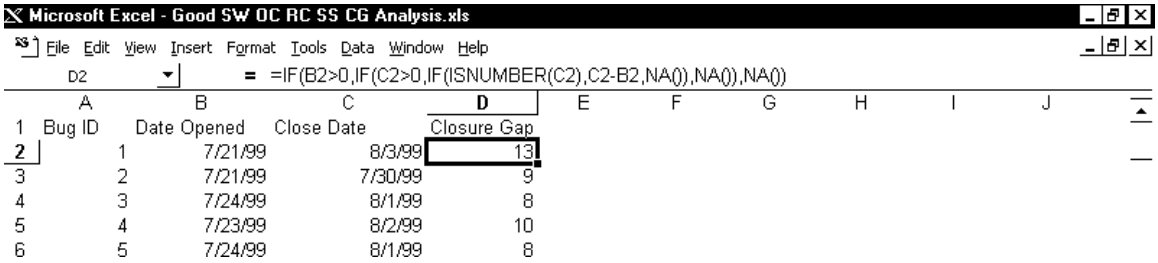


Figure 5: SpeedyWriter closure period

Calculating the closure periods, as I mentioned above, is a little tricky. The process involves two separate worksheets in Excel. In the first worksheet, calculate the closure period for each bug report. The first step is to put all the bug report IDs in the left-most column on the worksheet. The closure period for a given bug is `N/A` if the report is not closed. Figure 6 shows the upper third of the Excel worksheet that calculates this number for each bug, including the formula (in the formula bar, preceded with an "=" sign, directly above the worksheet).



	A	B	C	D
1	Bug ID	Date Opened	Close Date	Closure Gap
2	1	7/21/99	8/3/99	13
3	2	7/21/99	7/30/99	9
4	3	7/24/99	8/1/99	8
5	4	7/23/99	8/2/99	10
6	5	7/24/99	8/1/99	8

Figure 6: Calculating closure period for each bug report.

On the second worksheet, calculate the daily and rolling closure periods. First, set up the left-most column with all the dates from the start of test execution. Then, using th `SUMIF()` formula, add the closure periods (from the previous worksheet) that correspond to a particular closure date. (This number is zero if no bug report closed on that day.) Then, using the `COUNTIF()` formula, find out how many bug reports closed on that day. The remaining calculations entail totaling and dividing the numbers calculated by these two formulas. (Se Figures 7 and 8 for the Excel worksheet, with the two formulas appearing in the respectiv

Copyright © 1999 Rex Black. All Rights Reserved.

formula fields.) The closure period chart is obtained by plotting the daily closure period and rolling closure period columns against the dates in the first column.

Microsoft Excel - Good SW OC RC SS CG Analysis.xls

File Edit View Insert Format Tools Data Window Help

B2 = =SUMIF('Closure Gap-Calc'!$C$2:$C$500,$A2,'Closure Gap-Calc'!$D$2:$D$500)

	A	B	C	D	E	F	G	H
1	Date	Closure Period	Cum Closure Period	Inactive	Cum Inacti	Daily Closure Perioc	Rolling Closure Period	
2	7/21/99	0	0	0	0	#N/A	#N/A	
3	7/22/99	0	0	0	0	#N/A	#N/A	
4	7/23/99	0	0	0	0	#N/A	#N/A	
5	7/24/99	0	0	0	0	#N/A	#N/A	
6	7/25/99	0	0	0	0	#N/A	#N/A	

Figure 7: Determining total closure periods by date

Microsoft Excel - Good SW OC RC SS CG Analysis.xls

File Edit View Insert Format Tools Data Window Help

D2 = =COUNTIF('Closure Gap-Calc'!$C$2:$C$500,$A2)

	A	B	C	D	E	F	G	H
1	Date	Closure Period	Cum Closure Period	Inactive	Cum Inacti	Daily Closure Perioc	Rolling Closure Period	
2	7/21/99	0	0	0	0	#N/A	#N/A	
3	7/22/99	0	0	0	0	#N/A	#N/A	
4	7/23/99	0	0	0	0	#N/A	#N/A	
5	7/24/99	0	0	0	0	#N/A	#N/A	
6	7/25/99	0	0	0	0	#N/A	#N/A	

Figure 8: Determining bug reports closed by date

Root Cause

The third chart breaks down the root causes for all closed bugs. Root cause data is most interesting in the aggregate. Listing the closure of one bug with a specific root cause may not mean much, but seeing the breakdown for a hundred—or a thousand—bugs can tell an engaging story. Figure 9 presents a root cause breakdown for SpeedyWriter, showing the contribution of each type of error to the total number of bugs found and fixed so far. As you might imagine, a chart such as the one in Figure 9 grabs management's attention more effectively than a table.

Capturing root cause data has both short-term and long-term benefits. In the short-term, developers and project management can use the root cause data to course-correct. If many bugs arise from the lack of intelligible specifications, clarifying what the system should do may prevent some of these problems as the project moves forward. In the longer term, for each subsequent project, the project team can use the root cause information from previous projects to improve their process.

The chart shown uses industry-standard categories to group the data. You can choose different categories, you can use these, or you may have to accept the classifications offered by your bug tracking system. The advantage of using industry-standard classes is that you can compare your project and company results with those obtained by others in the software industry.

You can easily prepare the chart shown in figure 9. Th `COUNTIF()` formula allows you to tally the bug reports whose root causes fit into each category. The Excel chart wizard converts this into a pie chart automatically.

Copyright © 1999 Rex Black. All Rights Reserved.

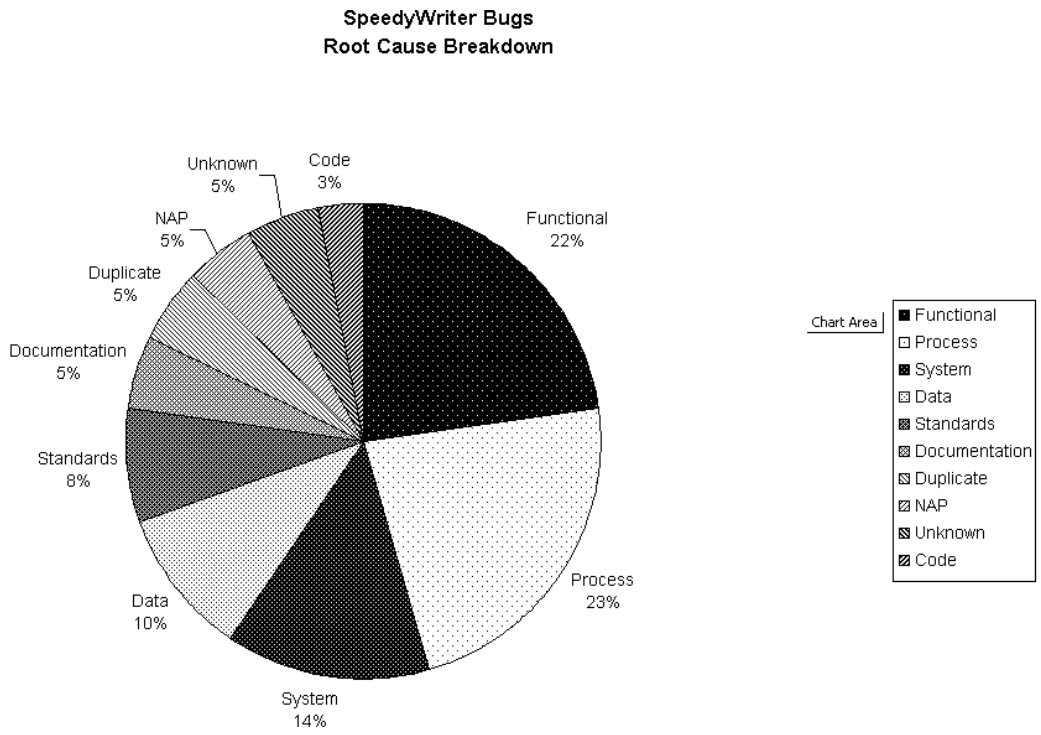


Figure 9: SpeedyWriter root cause breakdown

Subsystem

Like the root cause breakdown, the subsystem breakdown is a simple chart with an important message: it tells you which subsystems experience the most bugs. Since this is usually closely correlated with the subsystem in which the bug exists, you can draw conclusions about which subsystems have the most bugs. This allows the project manager to focus process and product improvement. It's useful to format this as a Pareto chart, as shown in Figure 10, becaus generally two or three subsystems suffer the most problems.

You can use a subsystem chart in the same way you use a root cause chart, to focus process and product improvement efforts. The fact that the user interface and the edit engine account for two out of every three bugs found in SpeedyWriter, for instance, indicates that an effort to make fewe mistakes in these areas would pay off handsomely.

In addition, if you are dividing your test effort evenly among the six subsystems, you should consider spending most of your testers' time on these two problem areas. This conclusion might seem counterintuitive—after all, if you didn't find many bugs in the other four subsystems, maybe you should have spent more time looking in those four categories. And you certainly should do this if field-reporting problems indicate a disproportionate number of test escapes in these four areas. However, it is usually the case that where you find many bugs, you will find more bugs. On the five projects mentioned, the top two subsystems accounted for more than half of the bugs every time, in two cases almost two-thirds, and once almost three-quarters.

Like the root cause chart, the subsystem breakdown is easy to create. On a separate worksheet, use the `COUNTIF()` formula to compute the number of bugs in each subsystem. Sort the list by count so that the most frequently reported subsystem comes first. Then, use another column in

Copyright © 1999 Rex Black. All Rights Reserved.

the same worksheet to accumulate an overall count for the upper curve on the chart. The Excel chart wizard will do the rest.

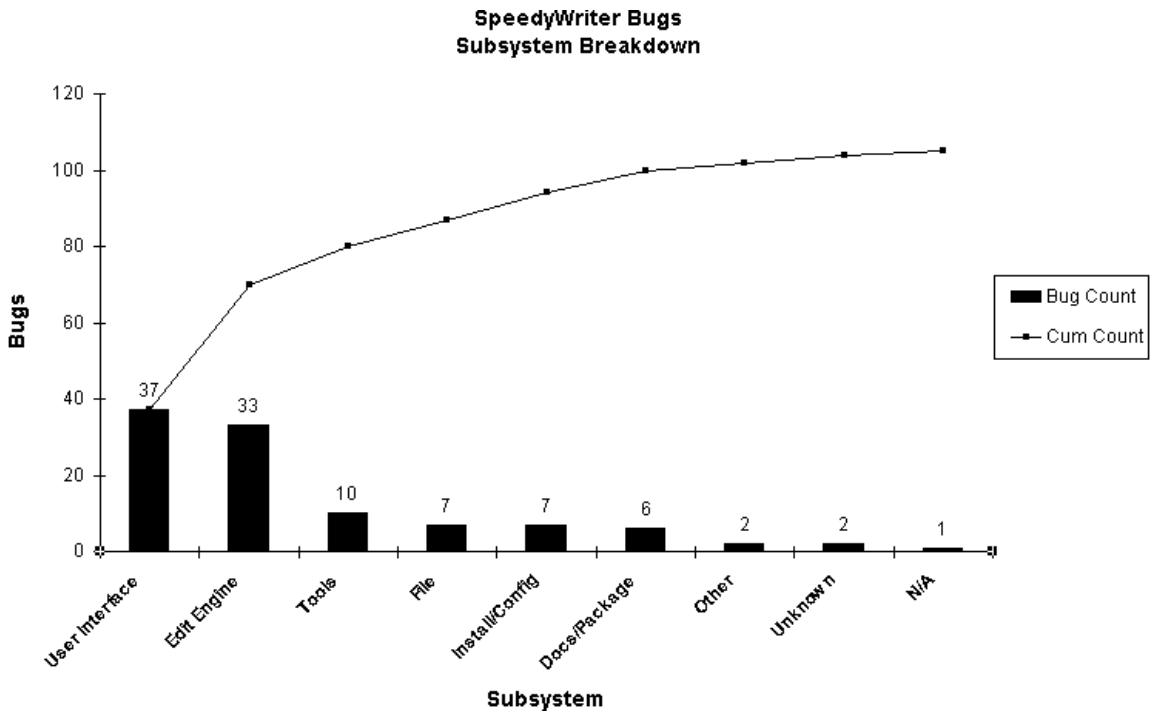


Figure 10: SpeedyWriter subsystem breakdown

Points and Pitfalls

As you accumulate historical bug report data—on both good and bad projects—you can compar charts from your current projects with charts from previous projects. Such comparisons can b enlightening. Even within a single class of projects such as laptop computer development, I hav seen variances of 600 percent in the total number of bugs found. Beyond the simple totals, th shapes of the curves can also differ. If you use these four charts consistently across a fe projects, you will soon recognize "virtuous" and "evil" curves.

Avoid blind faith in your charts, though. One key assumption of the opened/closed chart is *ceteris paribus* (all things held equal). You can arbitrarily flatten any cumulative opened curve by stopping the test effort. You can cause the opened and closed curves to converge by deferring bugs rather than fixing them. You can spoof opened and closed dates in the database to make th opened/closed chart fit any profile you choose. You can easily rig the closure period chart, too. Mass deferral of stale bug reports, recording phony opened and closed dates, opening new bug reports rather than reopening existing ones when fixes fail in confirmation testing, and other manipulation of the opened and closed dates will defeat your purpose. Finally, carelessness when assigning subsystems and root causes to bugs renders these charts worse than pointless. Nothing good can come of making decisions based on phony data.

Similar cautions apply to any other analysis of defect or test data. For your analyses to hav meaning, the underlying data must be accurate, complete, and free from gerrymandering. Only honest data yields worthwhile information.

Copyright © 1999 Rex Black. All Rights Reserved.

Finally, for the closure gap and root cause charts, keep in mind that the charts represent a subset of the overall bug data available. Only closed bug reports count in these charts. In the case of th closure period, that means that the chart probably understates the true daily and rolling turnaround metrics. The extent of the likely error in this chart depends on the proportion of th bug reports that remain open at the moment of concern. The same argument applies to the root cause chart. If fifty or more percent of the bug reports remain open, the root causes reported may not represent accurately the root causes of all the bugs in the product. The opened/closed and subsystem charts do not suffer from this problem to the same extent, being based on all the bugs reported to date, but you are still managing from an incomplete data set until the project is over. Be careful when using incomplete measurements to fine-tune processes in the midst of development or test execution.

Conclusion

In this paper, I have introduced (or re-introduced) you to four charts I find valuable for managing test projects. The perspectives offered by each chart are impossible to obtain by looking through a stack of bug reports. The higher level of abstraction is enabled by using analysis and charting tools to view certain key variables graphically. This abstraction along carefully chosen data dimensions is what makes these charts good "dashboard" indicators of test and development project performance.

The reports, charts, tables, and forms presented in this chapter are just starters. With a littl imagination, you can extract all sorts of useful data and reports from a bug tracking database. Start with simple tasks, learn the tool, and then expand it to meet your needs. For day-to-day management, however, these four charts—or your own enhancements of them—may well suffice. A cluttered dashboard makes it hard to focus on any one key indicator.

Recommended Readings

Rex Black: *Managing the Testing Process* (1999), Microsoft Press, Seattle, WA.

Boris Beizer: *Software System Testing and Quality Assurance* (1996), International Thomson Computer Press, Boston, MA.

Kaoru Ishikawa: *Guide to Quality Control* (1986), Asian Productivity Organization, Tokyo.

Stephen Kan: *Metrics and Models in Software Quality Engineering* (1995), Addison-Wesley, Reading, MA.

Biograph

Rex Black has spent almost seventeen years in the computer industry, with fourteen years in testing and quality assurance. He is the President and Principal Consultant of Rex Black Consulting Services, Inc., an international software and hardware testing and quality assuranc consultancy (www.rexblackconsulting.com). His clients include Dell, SunSoft, Hitachi, Motorola, Pacific Bell, GE Capital, Tatung, IMG, Renaissance Worldwide, DataRace, Omegabyte Omnipoint, TeleSource, Strategic Forecasting, and Clarion. He recently completed *Managing the Testing Process*, published by Microsoft Press in its Best Practices series. Mr. Black holds a BS Degree in Computer Science and Engineering from UCLA, and belongs to th Association for Computer Machinery and the American Society for Quality.

Model-based testscript generation for a C++ Object

J. P. Schroeder

TERADYNE

Software & Systems Test

44 Simon Street

Nashua, NH 03060-3094

800-996-8778

Table of Contents

Table of Contents i

Table of Figures ii

Table of Tables ii

Purpose 1

Introduction 2

Objects and Classes 2

- Object Oriented Programming 2
- Object Testing 3
- C++ Classes 3

Example Class 4

- List Class 4
- Stack Behavior 4
- Queue Behavior 5
- Common Methods 6

State Machine Modeling 7

- Black Box Testing 7
- Behavioral Models 8
- State Machine Models 8
- Attributes and Context 9
- Stimulus/Response Pairs 9
- Styles of State Machine Models 9
- Pure Behavioral Models vs. Testing Models 10

Example Class Modeling 10

- Object Analysis 10
- State Transition Tables 10

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

Example Class Events 11
Example Class States 11
Example Class STT 12
Example Class State Machine Model 13
Exploring Model Behavior 15
Model Validation 15
Test Design 16
Visual Documentation 16
Test Design Methodology 16
Tests 18
Design Implementation 18
Model Annotation 18
Test Code 18
Automated Test Generation 19
Constraint Strategies 21
Path Proliferation 21
Constraints 21
Coverage Schemes 22
Path Control 22
Example Class Model Constraints 23
Summary 24
Questions or Comments? 24

Table of Figures

- Figure 1 C++ Class Definition 4
- Figure 2: List with Stack Behavior 5
- Figure 3: List with Queue Behavior 6

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

- Figure 4: Example Class 7
- Figure 5: State Machine Model of a "Body in Motion" using classic notation 8
- Figure 6: Moore-style Stimulus/Response Pairs 10
- Figure 7: Example State Transition Table (STT) 11
- Figure 8: Example Class STT 13
- Figure 9: TestMaster Example Class State Machine Model 14
- Figure 10: Stack Scenario 16
- Figure 11: Breadth-First Generated Stack Scenario 18
- Figure 12: Push Function Stimulus and Response Verification Code 19
- Figure 13: Stack Scenario Test Driver 20
- Figure 14: Example Class Model with Constraints Shown 24

Table of Tables

- Table 1: Stack Example Test Creation Comparison 17

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

Purpose

This paper describes how to approach software object testing using a model-based testing technique. Specifically, it demonstrates how to create function-type tests for C++ classes.

This paper has been written for Testers with experience testing software applications. A familiarity with state machine models and TestMaster™ would be helpful, although is not necessary to understand this document. Readers are also assumed to have a working knowledge of at least the C programming language and some experience using it.

Finally, the use of state machine models to test software objects can be extended to any object-oriented program language or technology. Readers should be prepared to generalize the concepts and extend the simple examples shown in this paper to meet their own software testing needs.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

Introduction

This application note illustrates how to generate tests for a C++ class using a state-machine-based modeling approach.

C++ classes enable software engineers to design objects that have attributes (typically data) and behavior (typically implemented as functions). The member functions of a class are commonly called methods. Testing a class involves exercising its methods to ensure that they are internally and externally consistent. Internal consistency means the methods work correctly with all other methods of the class. External consistency involves correct operation with objects in the larger software application. The following discussion only includes testing for internal consistency.

A model-based approach to automated test design and generation involves creating "black box, state machine models of a system's behavior". In this case the System Under Test is a C++ class. The behavior of a class is defined by its public functions. Black box testing meshes neatly with the information hiding inherent in class definitions. The implementation details of the methods are purposely ignored. The focus of the testing is (again) the class's public functions. State machine notation is used to describe and partition the class's external behavior. The invocation of a method typically produces very different results depending on the class object's state. The state machine notation allows you to set-up and invoke the methods under these differing conditions.

Finally, the tests being generated are each a small, compilable C++ Test Driver.

All the examples shown use the TestMaster automated test design tool. The document also provides a step-by-step method for analyzing the components of a class and modeling these components. Given these examples, Testers can create their own state machine models manually or using TestMaster.

Objects and Classes

This section discusses Object Oriented Programming (OOP) and presents the C++ class used as an example throughout the paper. Readers familiar with OOP and C++ can skip this section.

Object Oriented Programming

OOP is a software technique for creating models of "things" in the real-world. It attempts to create these models by grouping the software (data and functions) needed to describe a "thing" into an *object*.

Objects are defined to have attributes and behavior (data members and methods). Attributes describe the condition of an object. For example, empty or full. Attributes map directly to data. Behavior describes the actions the object can perform. For example, object empties or object fills. Behavior maps into functions (called methods).

Ideally, an object can be used by only knowing "what" it does without knowing "how" it works. This is the programming technique called information hiding. Information hiding provides an unambiguous interface to an object, but "hides" the details of the object's implementation. An object's user can change its behavior and query its status through the public interface, but not be aware of the code that is effecting this behavior and these

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

attributes. The interface should provide the information to use the object, without the user having to know the internal details of the algorithms the object uses.

Using OOP, applications are programmed by combining objects to create the larger system. Typically a system is analyzed to determine its set of constituent objects. Then the objects are analyzed to determine their attributes and behavior. The interaction of the objects is also defined within the system. Finally, the objects are programmed individually and then together to create the overall system's software. Useful objects are intended to be "re-used" in subsequent systems, saving time and labor.

Object Testing

It should be noted that the use of previously created objects implies a certain amount of faith on the part of the person using the object. The user of an object may not be able to work back through its implementation to ensure it is producing the correct result. Without this faith the use of objects lose their advantages as software components for reuse. It is important that objects be coded in a robust fashion and be well tested to maintain to promote their re-use.

C++ Classes

In the C++ programming language, objects are created as classes. A class defines all the data associated with an object as well as the functions needed to manipulate the data. Classes are used in conjunction with each other to create the C++ programs for larger systems.

A C++ class definition begins with the language keyword **`class`**. This is followed by the class's label (or name). The body of the class object is defined with open and closed curly brackets (**`{}`**) and is terminated with a semicolon. Within the body of the class is the public member access specification (**`public:`**). This defines the external interface to the object. Client objects will use the class through the member functions described in this interface. One (or more) of the public functions has the same name as the class. This function is called the *Constructor* . A constructor is used to initialize an instance of a class. After the public functions may appear a private member access specification (**`private:`**). Private data and functions are only accessible through public member functions. The example below shows a generic C++ class definition.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

```
class name {
public:
     name();    // Constructor
     method 1   // Member function
     .                   .
     .                   .
     .                   .
     method n   // Member function
private:
     .          // Private functions
     .          // and data.
     .
};
```

- Figure 1 C++ Class Definition

Optionally, a function with the same name as the class, but preceded with a tilde (~) may appear in the list of public functions. This function is called a *Destructor*. The Destructor does termination cleanup for each class instance.

Example Class

This section introduces the C++ class used as the example in this paper.

List Class

The example is a List class. The List can be configured to handle any data type. The behavior of the List allows it to be used as either a Stack or a Queue.

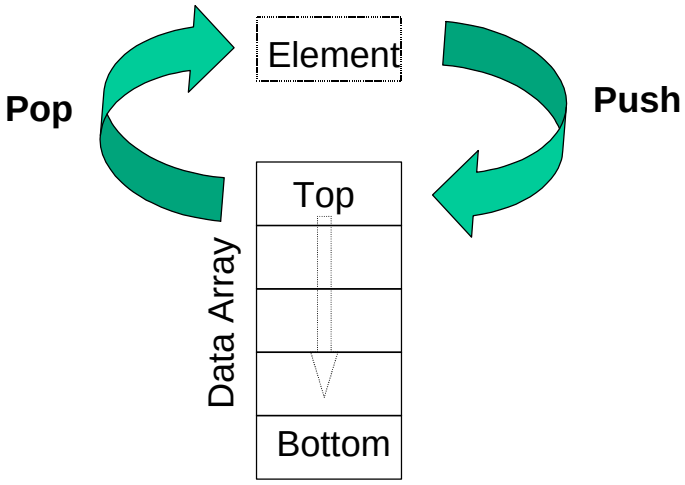
Stack Behavior

When behaving as a Stack, elements can only be added to the array from the top and removed from the top, i.e. it performs in a Last-In-First-Out (LIFO) fashion. There are two Stack methods: Pop and Push.

- **Pop**: Places an array element on the top of the array.
- **Push**: Removes an array element from the top of the array.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

Both member functions return TRUE when successful, otherwise FALSE. The figure below shows the behavior of Pop and Push on the data array.



- Figure 2: List with Stack Behavior

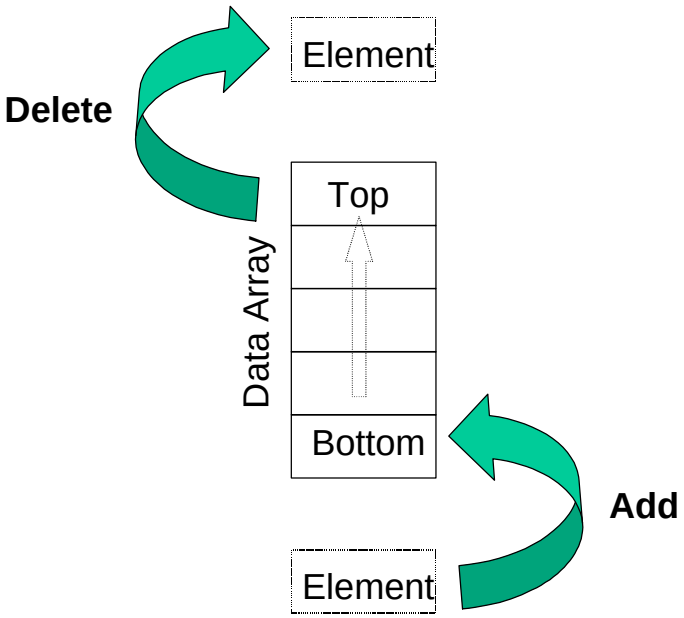
Queue Behavior

When behaving as a Queue, elements can only be added to the array from the bottom and removed from the top, i.e. it performs in a First-In-First-Out (FIFO) fashion. There are two Queue methods: Add and Delete.

- **Add**: Places an array element on the bottom of the array
- **Delete**: Removes an array element from the top of the array.

Both member functions return TRUE when successful, otherwise FALSE. The figure below shows the behavior of Add and Delete on the data array.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED



- Figure 3: List with Queue Behavior

Common Methods

In addition to the member functions affecting the list's storage behavior there are two functions used to query the current condition of the list: Empty, and Full.

- **Empty**: Returns TRUE when no data elements are stored in the list. FALSE is returned when the list is in use.
- **Full**: Returns TRUE when no data elements are available for data storage. FALSE is returned when storage is available.

Finally, both a Constructor and Destructor are included in the list of member functions.

The figure below shows the class template for the List object.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

```
// tstqu.h
// Simple template class for Stack/Queue data storage object
#ifndef TSTQU_H
#define TSTQU_H

#include <iostream.h>

template<class T>
class Stack_Queue {
public:
        // Alpha and Omega
        Stack_Queue(int = 5);                   // Constructor (Default Size is 5)

        ~Stack_Queue() { delete [] storePtr; }  // Destructor

        // Stack Methods
        int Push(const T&);                     // Push an element onto Stack (top)

        int Pop(T&);                            // Pop an element off of Stack (top)

        // Queue Methods
        int Add(const T&);                      // Add an element onto Queue (bottom)

        int Delete( T&);                        // Delete (remove) and element from Queue (top)

        // Common Methods
        int Empty() const { return top == -1; } // > 0  when no elements stored

        int Full() const { return top == size - 1 ; }       // > 0 when full
private:
        int size;                                           // Number of stored elements

        int top;                                            // Location of top element

        T *storePtr;                                        // Pointer to storage
};
```

- Figure 4: Example Class

State Machine Modeling

This section discusses creating “black box, state machine models of a system’s behavior”.

Black Box Testing

Typical testing strategies are either structural or behavioral in nature. Structural testing is also called “glass-box” or “white box” testing. Behavioral or functional testing is called

Testers perform glass-box testing by having complete access to the System-Under-Test’s design. In the case of software testing, this is the source code. This approach allows Testers to ensure that every source code statement, branch of a conditional check, and loop iteration executes correctly. Glass-box testing is most commonly employed by software developers when unit testing their software.

Black-box testing, is based upon the functional, performance, and interface requirements of the System-Under-Test. In the case of software testing, this is performed at the program or system level. A system is defined as multiple software programs (and possibly hardware assemblies) being employed together to perform an overall function. Knowledge of the internal workings of the system’s components is not needed and is even avoided. This approach encourages the Tester to concentrate on ensuring all the systems requirements at a defined interface are met. Tests are constructed without being influenced by the details of the system implementation. Functional testing is commonly employed by independent test organizations performing system integration as part of customer acceptance testing.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

Often, a product is tested using a hybrid test strategy, with unit level testing performed using the glass-box approach and higher system level testing performed using the black-box approach.

Behavioral Models

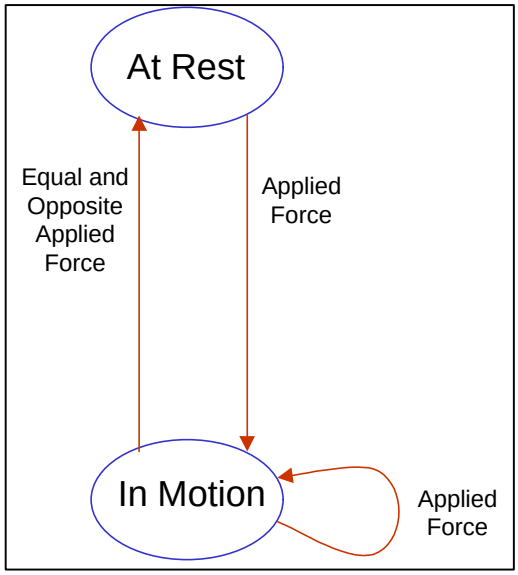
A model is an analogue of a thing or an object. It provides a tentative image of how its subject will look or perform. Models can be rendered mathematically, textually, in software, or graphically.

A behavioral model shows the lifecycle of an object. Lifecycle is defined as the object's behavior over time.

State Machine Models

State machine notation is a graphic technique for describing the behavior of an object over time. Classic state machine notation is made-up of only states and events.

A state represents the object under defined conditions. Objects typically require more than one state to describe all of their behavior. An event is an action causing the object to change from one of its states to another. A change in an object's state (or the movement from state to state) is called a transition. State machine notation is usually shown using a combination of labeled circles (or boxes) for states connected by labeled arrows describing possible transitions. The simplicity of state machine notation makes it very precise.



- Figure 5: State Machine Model of a "Body in Motion" using classic notation

There are two types of transition shown in the example model above: Simple and Looping. A simple transition is a single event (lone transition) that can cause a state change. For example: the In Motion state is a direct result of the Applied Force event. A Looping transition has the same state at its head and tail. This indicates an event having no effect on the object's state. For example: the Applied Force event re-applied to a body "In Motion" does not cause a state change.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

Attributes and Context

Not shown in the model above are associated attributes and context. Attributes and context are both maintained outside of state machine models.

Attributes are information associated with Events and States. There may or may not be dependencies between a model's events and states and its attributes. Examples of attributes in the model above might include: the body's mass and shape, and the magnitude, direction, and duration of the Applied Forces.

Context is a point in time within the model's lifecycle. Sometimes context is referred to as model history. Knowing the past and current context of a model can be useful. The model above has an infinite duration—there is no beginning or end. However, it may be important to specify the initial state (initial context) of the body to be At Rest. The model's terminal state and context might also be At Rest. Further, it might be convenient to know how many additional Applied Forces (and their direction, magnitude and duration) have occurred while the body is In Motion to calculate the Equal and Opposite Applied Force. This would require knowing the past context of the Applied Forces event.

Stimulus/Response Pairs

When using state machine models in testing, it's important to understand the concept of a stimulus/response pair.

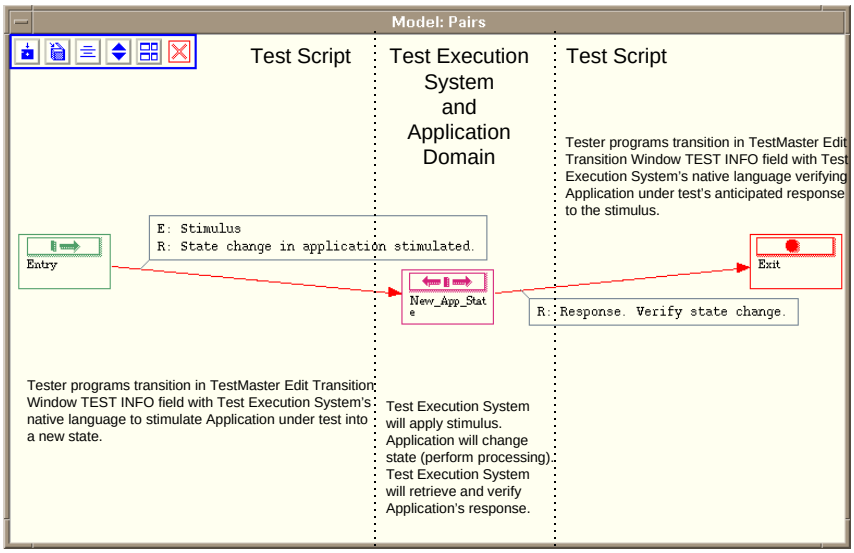
Tests typically are made-up from individual test steps. Every test step is made-up of a stimulus and a verifiable response. When you create models of objects you need to know the events that stimulate the object to change state and the corresponding characteristics of the new state. The characteristics of an individual state are used to verify the object's response. Generally, there is a one-to-one correspondence between stimuli and their corresponding verifiable response. There can also be a many-to-one correspondence between stimuli and a response transition. Your testing will be seriously complicated (or your System Under Test has serious design problems), if a single stimulus can result in any of several responses. Occasionally an event will have no effect on the state of an object within the context of the model. The Tester must determine if this event is important in the testing of the object and either include or exclude it from the model.

Styles of State Machine Models

The TestMaster tool models a stimulus as an event. Events are shown as transitions. Modelers can embed information into a TestMaster model's transition record (shown as an Edit Transition window by the tool). The response verification is co-located in the stimulus transition. Occasionally, the stimulus and response verification are located in separate transitions. Generally, when modeling results in the stimulus and response being located in the same transition, this results in Mealy-style state machine models. When they are located in separate transitions this is Moore-style state machine models. The style chosen is not important, as long as it is applied consistently throughout the model.

The diagram below shows a transition object (arrow) dedicated to stimulating the System-Under-Test, the System Under Test's change of state (a state object), and a transition object dedicated to verifying the response of the System Under Test. Each test step in your model needs to include these three elements (although not necessarily in three separate objects).

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED



- Figure 6: Moore-style Stimulus/Response Pairs

Pure Behavioral Models vs. Testing Models

A purely behavioral model may be very different from a model used for testing purposes. A "pure" behavioral model may be an academic description of its subject's operation. A model used for testing purposes will necessarily include details of the implementation or the (typically limiting) capabilities of the Testers Test Execution System. A compromise is struck.

In the example class used in this paper note the state of "Full" is not inherent in the definition of a Stack or Queue. Full is an artifact of the Stack or Queue's implementation. Likewise, the use of the C++ programming language will impose further limitations upon a purely behavioral model.

The purpose of this paper is to demonstrate model usage for software testing. The example model will make several compromises with regard to States and Events to test the C++ implementation of the example class. Practitioners of OOD wanting to explore model usage for design purposes may want to create slightly different models that exclude these compromises.

Example Class Modeling

This section discusses creating a state machine model of the Example class.

Object Analysis

Modeling begins by listing the events and states in an object's lifecycle. The actual model is a description of the relationships between these states and events.

State Transition Tables

Describing all the behavior of an object is not always easy. Frequently Testers new to modeling have trouble defining an object's states. An effective tool for exploring the

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

behavior of an object is the State Transition Table (STT). A STT is also a type of model. However, we will use it as the input to a graphic state machine model.

A STT is a matrix with states on columns and events on rows. The intersection of a row and column (cell) is the new state resulting from an event occurring when the object is in the column's state. Not all events are applicable in all states. It is not uncommon to place *Not Applicable* (N/A) in a STT cell. A STT provides a comprehensive listing of the state event relationship and can be used by Testers to create their state machine models. The figure below shows an example STT.

Example State Transition Table				
	State 1	State 2	…	State *n*
Event 1	*new State*	*N/A*		*new State*
Event 2	*N/A*	*new State*		*new State*
.				
.				
.				
Event *m*	*new State*	*N/A*		*new State*

- Figure 7: Example State Transition Table (STT)

Example Class Events

Events are actions that may cause state change. The events of a class object are already defined for us. They are the methods listed in the class's public member access specification. The events in our class example include:

- Push
- Pop
- Add
- Delete
- Empty
- Full

Behavioral modelers should note that we are making a compromise in pure behavioral modeling by listing the Empty and Full methods as events. However, from a testing perspective we want to include the Empty and Full methods in our model (and our tests). The only available mechanism is to consider them as events.

Example Class States

A state is a representation of the object under defined conditions. Deriving all your modeling subject's states is not always intuitive. It helps to abstract the nature of subject as much as possible. For example, a car is a vehicle, a ball is a sphere, and a bottle is a container.

In the case of our example class, it helps to consider it as a "container of data". This perspective is hinted at by Empty and Full methods in the class's declaration. The "state" of our class is defined by the value of its data members.

It helps to verify your choice of states by creating one or more scenarios involving your subject's behavior. These scenarios should include the object's common usage. For example, a program incorporating the example class might use it in the following way:

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

The Constructor is called. Data is Pushed or Added to the Stack or Queue. Added or Pushed data is Popped or Deleted. If enough data to fill the Stack or Queue is Pushed or Added, no more Pushes or Adds can be performed until data is Popped or Deleted. The Stack or Queue is checked to be Empty or Full. The Destructor is called.

Use your scenario to determine your subject's Initial and Terminal States. The Initial State is the state of your subject when the scenario begins. The Terminal State is the state of your subject when the scenario stops. Typically it is easiest to determine your subject's Initial State. It is not uncommon to find your subject has multiple Initial and Terminal states.

Examining the scenario created above using the container abstraction you could derive the following states:

1. When the Constructor is called, the Stack or Queue state is Empty.
2. When data is initially Pushed or Added, the Stack or Queue state is not Empty (or not Full).
3. When the maximum amount of data is Pushed or Added, the Stack or Queue state is Full.
4. When less data is Popped or Deleted then is Pushed or Added, the Stack or Queue state is not Full.
5. When all the Pushed or Added data is Popped or Deleted, the Stack or Queue state is Empty.
6. When the method Empty or Full is called, the Stack or Queue state is: Empty, not Full (not Empty), or Full.
7. When the Destructor is called, the Stack or Queue state is: Empty, not Full (not Empty), or Full.

Notice that when the Destructor is called the Stack or Queue can be in multiple states. Also, that "not Full" and "not Empty" are equivalent states. Throughout this example model, we'll use the "not Full" state. The final list of possible states includes:

- Empty
- Not Full
- Full

Behavioral modeler's should note the inclusion of the "Full" state. There is no "Full" state inherent in the definition of a Stack or Queue. Full is an artifact of the class's implementation. This artifact can provide many interesting tests.

Example Class STT

Creating an STT from the list of Events and States results in the following:

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

Example Class State Transition			
	Empty	**Not Full**	**Full**
Push	*Not Full*	*Not Full* *Full*	*N/A*
Pop	*N/A*	*Not Full* *Empty*	*Not Full*
Add	*Not Full*	*Not Full* *Full*	*N/A*
Delete	*N/A*	*Not Full* *Empty*	*Not Full*
Empty	*Empty*	*Not Full*	*Full*
Full	*Empty*	*Not Full*	*Full*
Constructor	*Empty*	*N/A*	*N/A*
Destructor	*Empty*	*Not Full*	*Full*

- Figure 8: Example Class STT

The table is read in the following fashion: “When the Stack is Empty and the Push method is exercised, the new state is Not Full”.

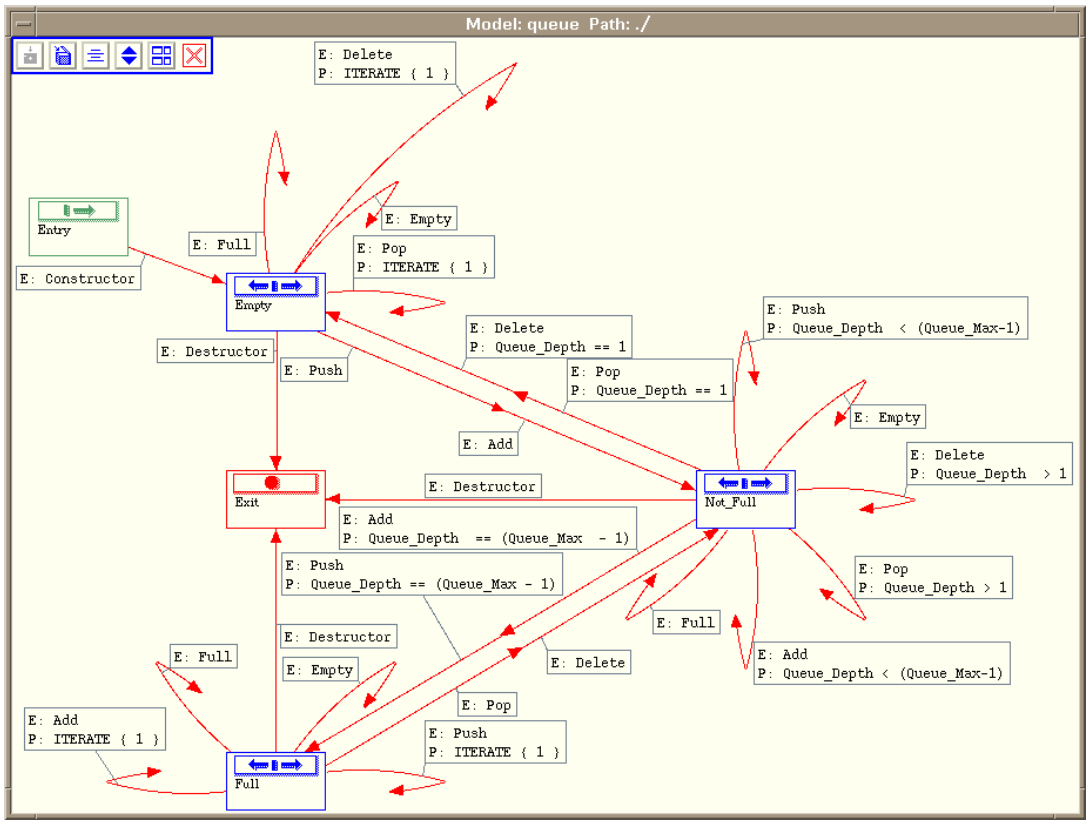
A few things must be noted about the table. The number of data elements contained by the Stack or Queue effects the subsequent state. For example, when the Stack is Not Full and a Push operation is exercised; if this last Push will fill the Stack., the new state is Full, otherwise it is Not Full. The example table shows this as two states in one table cell. An event that may lead to two different states is shown graphically using two separate transitions.

Certain methods are prohibited on the current state. These are shown in the example’s STT with an N/A. An example of this is trying to Pop an Empty Stack. Testers need to decide if these events should be included in their model. Finally, some methods have no effect on the current state. No state change is shown by listing the current state in the cell. An example of this is the Empty method which does not change the state of a Stack or Queue.

Example Class State Machine Model

The following figure is the state machine model created (using TestMaster) from the Example Class STT.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED



- Figure 9: TestMaster Example Class State Machine Model

The Example class's state machine model uses Mealy-style state machine notation. Stimulus (method) and response verification are presumed to be located together in the same transition.

All states in this model are shown as boxes. This model includes explicit Entry (Initial) and Exit (Terminal) states describing the beginning and end of the object's lifecycle. These states are not shown in the STT.

Each method is a separate Event with its own transition. All of the methods have been applied to each of the states for testing completeness. Mutually exclusive methods are shown as two head-to-tail transitions between states. Mutually exclusive methods have the same state changing effect on the List object, but are separate operations. For example: the Push method when the Stack state is Empty results in the Not Full state, the Add method also has the same effect on a Queue.

In the example, model transitions are documented with a comment box. A comment box includes the Event name and optionally a Predicate. Event's ("E:" in the box) are always the method being documented by the transition. The optional predicate ("P:" in the box) describes when the particular transition is applicable within the context of the model. For example, checking to see if the current Push operation will fill the List (and change the state) is an attribute check. Predicates in the model are implemented using the TestMaster's Path Flow Language (PFL). PFL has a programming language-like syntax and can be used to create, check and maintain model attributes and model context.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

TestMaster also uses PFL to automatically design and code tests directly from models created with the tool.

Note this model was constructed to be complete. From the STT we saw that several events were N/A in certain List states. In the “pure discipline” of behavioral modeling this may be true. However, in the context of software testing it would be prudent to verify these events truly have no effect.

Exploring Model Behavior

This section discusses using the example state machine model to design tests. To simplify the narrative, only Stack behavior with be explored for the remainder of the document.

Model Validation

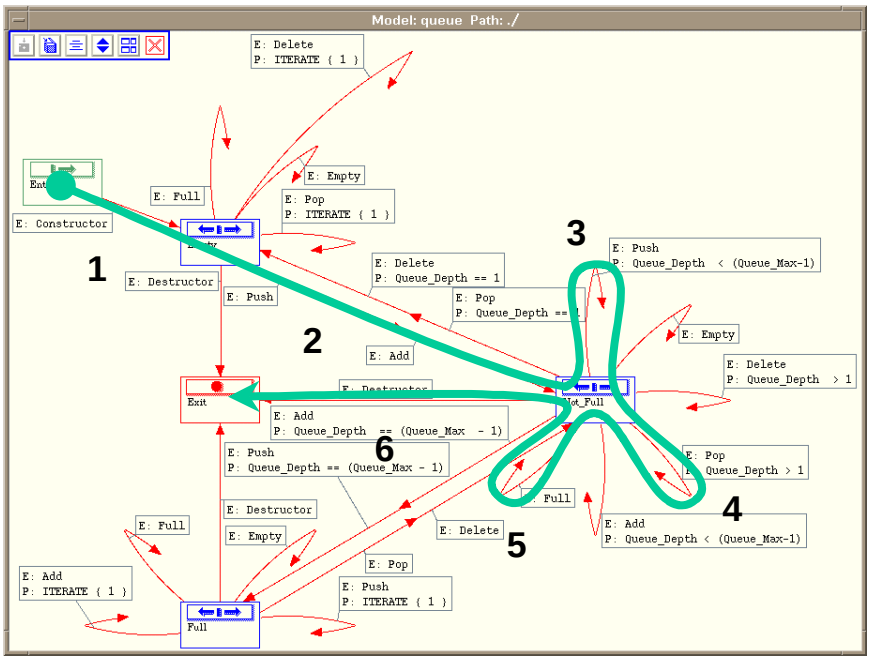
The TestMaster state machine model previously shown was created directly from the STT. The intersection of each State and Event is included in the model.

It would be prudent to verify the model with regard to typical List behavior to ensure its accuracy. Below is a scenario for using a Stack. The Stack methods (Push and Pop) are used exclusively along with the Constructor, Destructor and the Full method.

1. The Constructor is called.
2. A data item is initially Pushed onto the Stack.
3. An additional data item is Pushed onto the Stack.
4. The previous data item is Popped off the Stack.
5. The List is checked to be Full.
6. The Destructor is called.

We can show this scenario graphically on the model. We call this sequence of model objects a Path. Model objects include state and transition icons. A Path is a complete, valid sequence of objects from the model’s Entry state to its Exit state. Below is a diagram of the Path described in the enumerated scenario.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED



- Figure 10: Stack Scenario

Additional Stack-type scenarios, or Queue-type scenarios (using Queue methods Add and Delete) can be constructed to further test the accuracy of the model. It should be noted that there is no prohibition from using Stack methods together with Queue methods in a scenario.

Test Design

It follows, that if you can map usage scenarios into the model, you can use the model to create new scenarios. The creation of new scenarios is behavioral test design. There are two advantages to using models to create tests: it's visual i.e. it's easy to understand, and it provides an easy to follow methodology toward test design.

Visual Documentation

Diagrams are a very compact method of communication. The state machine model created for the Example class tells a lot about the function of the class's methods and their relationship to each other. For example, we can easily see the Pop and Delete methods are equivalent. Also, the Full and Empty methods have no effect on the object's state whether it's used as a Stack or a Queue. Creating the state machine diagram teaches the Modeler-cum-Tester about its subject and provides useful documentation for people who might want to learn about the object.

Test Design Methodology

The state machine model can be used to create new scenarios. The creation of new scenarios can be approached in a purely heuristic way or in a more methodical fashion.

If a heuristic approach to test design is taken, the diagrams act just like a map would be used to plan travel. By looking at the map, the scenario creator can judge the total number

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

of possible Paths and select the important or representative Paths based on their previous experience and best judgment.

If a methodical approach to test design is merited, search algorithms can be borrowed from directed graph theory to examine all possible scenarios. The two applicable techniques are Breadth-First and Depth-First searching. Either of these algorithms will provide results. They will provide roughly equivalent results with simple state machine models, and more divergent results as model complexity increases. Model complexity increases as the number of states and transitions increase. Complexity is also increased by the number of branching transitions occurring in the model.

Selecting either search methodology and manually applying it methodically will result in a comprehensive list of all possible tests. The TestMaster tool supports the use of both search algorithms. Testers can direct the tool to search for and document Paths using either algorithm. In addition, it allows you to set criteria to condition the search. An example of conditioning is: "Produce the minimum number of scenarios, where every method is exercised appearing at least once in a test".

Using TestMaster to automatically apply both of these algorithms to the Stack example seeking the minimum number of tests to exercise all the methods together produces the following results:

Search Algorithm Applied	Number of Tests Found
Breadth-First Methodology	5
Depth-First Methodology	3

- Table 1: Stack Example Test Creation Comparison

The different tests produced are roughly equivalent in length. Below is an example of a Breadth-first search test produced. The Stack depth has been initialized to five elements deep. Note that each transition is one of the class's function calls.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

```
Constructor /* Call the Constructor*/
Push        /* Push the initial data element on the Stack
Full        /* Is the Stack full?
Empty       /* Is the Stack empty?
Push        /* Push a data element on the Stack
Pop         /* Pop a data element from the Stack
Push        /* Push a data element on the Stack
Push        /* Push a data element on the Stack
Push        /* Push a data element on the Stack
Push        /* Attempt to Push a data element on the Stack
Pop         /* Pop a data element from the Stack
Push        /* Push a data element on the Stack */
Push        /* Attempt to Push a data element on the Stack
Full        /* Is the Stack full?
Empty       /* Is the Stack empty?
Destructor  /* Call the Destructor */
```

- Figure 11: Breadth-First Generated Stack Scenario

Tests

This section describes how to create tests from state machine model scenarios

Design Implementation

Designed tests can be easily converted into executable tests for software objects. The Tester needs to rewrite the scenarios created from the model to be executable in a Test Execution Environment as a series of stimulus and response pairs (test steps).

Test Execution Environments can be commercially available products or applications you write yourself. In its simplest form, a Test Execution Environment is a test driver. Since the example class is written in the C++ language, our Test Execution Environment will be C++ and we will write a C++ test driver.

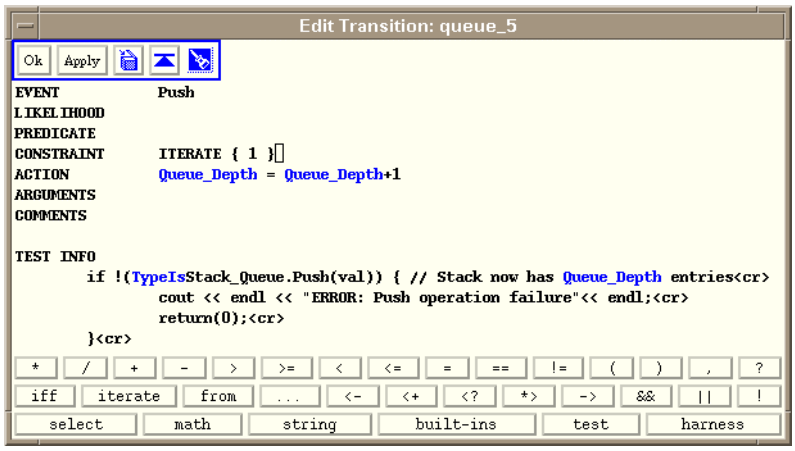
Model Annotation

Going back to test design, we found that each transition was one of the class's function calls. Because we have created a Mealy-style state machine model each transition must include both stimulus and response. So each transition must include both the function call and verification of success or failure of the call. Because several of the functions return success or failure as part of the call, code for simple stimulus and response verification can be written very quickly. Below is an example of the code needed to verify a Push Function call written using TestMaster. The code is found in the window field labeled Test Info.

Test Code

Since a test is a Path, it's made up of a series of transitions, it is easy to see how we can substitute these code fragments for each of the transitions to incrementally build-up programs that exercise the behavior of the software object.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED



- Figure 12: Push Function Stimulus and Response Verification Code

The construction of the test code requires the use of "regular" code fragments. The model's construction supports this. For example: the Push operation that occurs between the class's Empty and Not Full states always has the same syntax, whenever the initial class state is Empty. In a behavioral test, the Push function may be exercised several times between these Stack states. The same code fragment would be used for each of these executions. Note any Push operation executed when the Stack's state is Full would be expected to be a different expression. In this case, we would expect the Push to fail (the Stack is Full!), while a Push operation initiated when the Stack was Empty is expected to succeed.

In the following figure, the Breadth-first generated Stack test (previously shown) is implemented as executable code. The code fragments to generate the appropriate stimuli and responses have been substituted for the transition events. The result is a Test Driver for exercising the Example class's performance as a Stack using floating point data type elements. A C++ program header has been added to make the result compilable.

Note the example Test Driver instantiates a Stack with a capacity of five (5) float data values. This capacity was purely a testing-based decision. In some circumstances the selection of the appropriate capacity may be a seriously considered modeling decision.

The code in the Test Driver only takes advantage of each function's return a status code. More elaborate testing can easily be performed to verify the contents of the Stack by varying the value of the Stack's contents, checking stack order, and verifying the depth. The degree and sophistication of the code to create the stimulus and verify the response is only limited by the imagination of the Modeler and their ability to write code.

Automated Test Generation

The manual creation of a large number of tests can be time consuming and tedious. The Test Driver shown as an example was automatically coded using TestMaster.

TestMaster is a graphic design tool used to create state machine models. It also has the facility to design tests, and then code them automatically from the models. The coding of the tests requires the Tester to provide code fragments for each transition in the model. Several algorithms for creating tests from a model are available in the tool. In addition, there is facility for formatting the tests produced directly from the model.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

```
// Test #3

#include <iostream.h>
#include "test_object.h"

main ()
{


Stack_Queue<float> floatStack_Queue(5);
float val = 1.1;

        if !(floatStack_Queue.Push(val)) { // Stack now has 1 entries
                cout << endl << "ERROR: Push operation failure"<< endl;
                return(0);
        }

        if (floatStack_Queue.Full()) {// Stack/Queue value is: 1
                cout << endl << "ERROR: full stack/queue reported when not full"<< endl;
                return(0);
        }

        if (floatStack_Queue.Empty()) {         // Stack/Queue value is: 1
                cout << endl << "ERROR: empty stack/queue reported when not full"<< endl;
                return(0);
        }

        if !(floatStack_Queue.Push(val)) { // Stack now has 2 entries
                cout << endl << "ERROR: Push operation failure"<< endl;
                return(0);
        }

        if !(floatStack_Queue.Pop(val)) { // Stack now has 1 entries
                cout << endl << "ERROR: Pop operation failure"<< endl;
                return(0);
        }

        if !(floatStack_Queue.Push(val)) { // Stack now has 2 entries
                cout << endl << "ERROR: Push operation failure"<< endl;
                return(0);
        }

        if !(floatStack_Queue.Push(val)) { // Stack now has 3 entries
                cout << endl << "ERROR: Push operation failure"<< endl;
                return(0);
        }

        if !(floatStack_Queue.Push(val)) { // Stack now has 4 entries
                cout << endl << "ERROR: Push operation failure"<< endl;
                return(0);
        }

        if !(floatStack_Queue.Push(val)) { // Stack now has 5 entries
                cout << endl << "ERROR: Push operation failure"<< endl;
                return(0);
        }

        if !(floatStack_Queue.Pop(val)) { // Stack now has 4 entries
                cout << endl << "ERROR: Pop operation failure" << endl;
        }

        if !(floatStack_Queue.Push(val)) { // Stack now has 5 entries
                cout << endl << "ERROR: Push operation failure"<< endl;
                return(0);
        }

        if (floatStack_Queue.Push(val)) { // Stack better have 5 entries
                cout << endl << "ERROR: Push operation failure"<< endl;
                return(0);
        }

        if !(floatStack_Queue.Full()) {// Stack/Queue value is: 5
                cout << endl << "ERROR: not full stack/queue reported when full"<< endl;
                return(0);
        }

        if (floatStack_Queue.Empty()) {         // Stack/Queue value is: 5
                cout << endl << "ERROR: empty stack/queue reported when full"<< endl;
                return(0);
        }


        return 0;
}
```

- Figure 13: Stack Scenario Test Driver

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

In a small model like the example shown, automation is not needed. However with larger models or several models used in combination with each other, using a tool can save a lot of time creating tests.

Typically, Tester fatigue or lack of time to design and execute tests lead to incomplete product testing. The best manually designed and coded test in a test suite is the first test. Subsequent tests, especially when large numbers of tests are needed tend to be variations of this initial test. Automatically designed and coded tests have several advantages over manually generated tests. There are generally more of them, they are longer containing more test steps, they can be more varied in behavior, and (once debugged) contain less coding errors. The variation in the test's behavior is a function of the algorithm used to generate the tests.

Constraint Strategies

This section discusses how to constrain the model described in the previous section. It applies to Testers using TestMaster to automatically design and code their tests.

Path Proliferation

In the context of state machine modeling, constraining is the used to reduce the number of Paths (and hence tests) produced.

Carefully crafted models faithfully reproduce the lifecycle behavior of their subject. However, it's not uncommon for the simplest models to have a limitless variety of behavior. It's simply not practical to reproduce all of this behavior as tests. Since TestMaster initially produces Paths, (which can be coded into tests) we call this the Path Proliferation problem.

For example, in the automatically generated Stack Test Driver presented in a previous section, the Stack is filled once and partially emptied. This provides an interesting and meaningful test. Another meaningful test might be to fill and empty the Stack once, twice, and maybe ten times. However, what is the value of filling and emptying the Stack 11 times? Probably none. If it works correctly 10 times, it will most likely work correctly the 11th (barring any "boundary" type software defects).

Hence constraining. In automated test generation, 11 Stack fills and empties has the same importance as 10 fills and empties since it is valid Stack behavior. The Tester has to constrain the model to produce interesting tests that can be completed in a reasonable amount of time.

Testers should not confuse test constraint strategy with system behavior and the use of predicates. Models should exhibit the behavior of the System Under Test. PFL statements in transition Predicate fields ensure the model behaves like the System Under Test. Predicates are control applied to the model called-out by the System Under Test's specification. Constraints are control applied to limit the quantity and direct the type of tests generated by a correctly behaving model.

Constraints

The two basic methods for constraining TestMaster Models are Coverage Schemes and Path Control.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

Coverage Schemes

Coverage Schemes are a TestMaster term for the implementation of different Path search algorithms. They are the implementation of the previously mentioned search methods. Coverage schemes are the easiest method for controlling the number of tests being automatically produced by a model. By choosing the appropriate coverage scheme Testers can vary the number of tests, and the content of the tests being generated. TestMaster supports the following coverage schemes:

- Quick
- Transition
- Full
- N-Switch
- Profile

Quick Cover is the Breadth-first search that includes every transition in a test at least once. Transition Cover is the Depth-first search producing tests that include every transition in a test at least once. Quick or Transition coverage can produce a modest number of tests that can satisfy most testing requirements.

Full Cover is a Depth-first search. It produces tests that include every transition in all possible combinations with each other transition. A higher degree of constraining needs to be applied to reduce the number of tests produced using Full coverage schemes. Full coverage generally produces a large amount of tests for even simple models. N-Switch is a coverage scheme producing results between Transition and Full coverage. It uses a Depth-first search. Profile cover allows Testers to “weight” the likelihood of individual transitions in a model being incorporated into a Path. It uses a Depth-first search.

Table 1 previously showed that Quick Cover produced three tests and Transition Cover produced five on our Stack example. When Full Cover was attempted, it produced greater then 5000 tests.

Path Control

Path Control is the application of constraints to individual model paths. These path constraints provide a finer degree of control over model test generation than coverage schemes.

Path controlling constraints are PFL statements applied to a transition in the CONSTRAINT field of a transition’s Edit Transition window.

There are two types of Path controlling constraint applied in TestMaster models:

- **In-Path** – Constraints limiting the number of times a transition occurs in a current path.
- **Across-Path** – Constraints limiting the number of times a transition occurs in all paths.

Note that a single transition may include both In-Path and Across-Path constraints.

In-Path Constraints

In-Path constraints limit the number of times a transition occurs in the current path. The primary use of In-Path constraining is to control circular behavior (sometimes called Loop Control) and access control.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

Circular paths frequently occur in models and represents valid behavior. However, in excess it typically does not lead to interesting or practical tests. Software engineers are familiar with this as a problem known as “infinite loops”. Circular behavior can occur in the use of Looping transitions and models where behavior can oscillate between one or more states. The PFL ITERATE statement is a loop counting mechanism used to control the number of times a transition is repeated in a Path.

Examples of Loop control can be seen on the Loop transitions used to describe the example’s Empty and Full functions.

Access control involves setting-up logical conditions to ensure “interesting” or correct behavior. Interesting behavior is test paths the Tester finds non-trivial, or test paths the Tester believes are more likely to find errors in the System Under Test, or to ensure the model reproduces its subject’s behavior correctly. Access control is accomplished through PFL statements controlling and testing model context. This can be done by creating variables maintained in the model or by testing model context.

Examples of Access control can be seen in the mutually exclusive transitions used to describe the state change from Not Full to Full in the class. In this case the last Push or Add of a data element needed to fill the Stack or Queue triggers the state change.

Both types of In-Path constraining have been applied to the model.

Across-Path Constraints

Across-Path constraints limit the number of times a transition occurs in all paths. The primary Across-Path constraint types are the PFL PRUNEAFTER and SELECT Statements.

PRUNEAFTER removes a transition from inclusion in subsequent Paths after it has appeared a specified number of times. It can be used to delete transitions leading to Paths which include marginal behavior, or that might produce unwanted multiple tests.

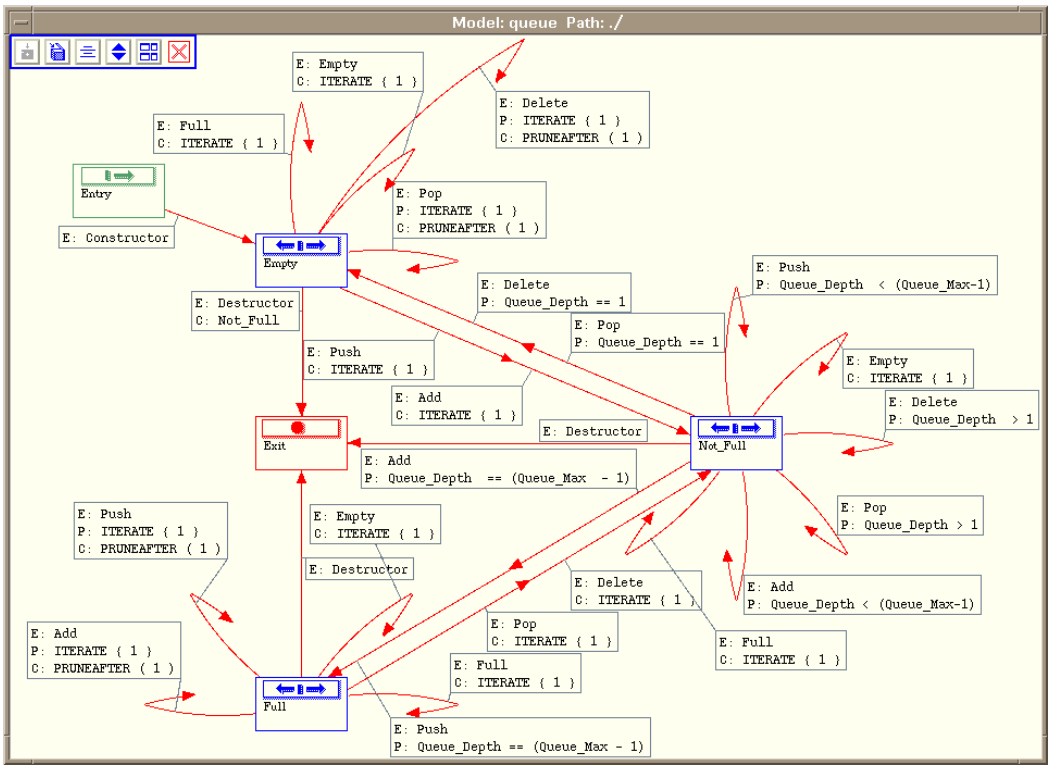
Examples of PRUNEAFTERs are used in the model to limit of the appearance of transitions testing the Not Applicable Event and State combinations. For example, after trying to Push or Add to a Full Stack or or Queue once this test step probably does not need to be repeated.

Selection statements limit the sets of path objects appearing in a path. They also control the sequence of appearance in a path. Selection statements were not use in the model.

Example Class Model Constraints

The figure below is the model with all constraints shown.

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED



- Figure 14: Example Class Model with Constraints Shown

In the above example, transitions are documented with a comment box. The box includes the Event Predicate and Constraint being applied. In this model, the Event ("E:" in the box) is always the method being documented by the model. The optional predicate ("P:" in the box) describes when the particular Event is applicable within the context of the model. The optional constraint ("C:" in the box) describes the applied control to the automatic generation of tests.

Summary

State machines can be used to model software objects, in particular C++ classes. Note that state machine models can be used with any object oriented programming language.

By analyzing class methods a state machine model can be constructed. An example class with both Stack and Queue behavior was modeled to show a class object state machine model. Once a model is created, it can be used to explore the behavior of the modeled object. The model also becomes a graphic test artifact. Tests can be designed directly from the model. With the use of TestMaster these tests can also be automatically designed and coded.

Given the example in this paper, Testers can create their own state machine models of software objects. If they are using TestMaster they can use the product to design and code the tests for their software objects automatically.

Questions or Comments?

Please direct any questions or comments to Applications Engineering at:

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

TERADYNE
Software & Systems Test
44 Simon Street
Nashua, NH 03060-3094
VOX: 800-996-8778
FAX: 603-791-3075
e-mail: support@sst.teradyne.com

© COPYRIGHT 1998 BY TERADYNE CORP. ALL RIGHTS RESERVED

AutoTester Inc.

View ad

AutoTester provides software quality assurance experience, expertise, and technology to more than 1,600 Fortune 3000 customers worldwide. Founded in 1986, AutoTester specializes in developing and delivering software quality methodologies, technologies, services, and support for mainframe, midrange, and client/server environments. Using a defined quality process, experienced professional service, and a comprehensive suite of quality assurance software, AutoTester helps customers worldwide with Year 2000, SAP™ R/3™ internet/intranet, Windows ®, and OS/2® software quality initiatives. AutoTester is based in Dallas, Texas with a network of sales offices and distributors throughout North America and Europe.

AutoTester takes a holistic approach to software quality assurance by providing solutions that include processes, services, technology, and support.

Quality Assurance System- AutoTester has developed a unique quality assurance system that ensures the integrity of your software applications during every step of the implementation process. Our system includes a framework that integrates your existing development and QA processes and provides a structured, repeatable process to help ensure your applications meet your quality requirements.

Comprehensive Professional Services - AutoTester's software quality professionals provide consulting, training, and implementation support to customers throughout the software deployment process.

Proven Technology - AutoTester provides project managers, business users, and software developers with a suite of software products designed to test a wide range of applications. AutoTester’s integrated testing, test management, and task automation software supports a wide range of development projects and environments.

Complete Customer Support - AutoTester’s customer and technical support teams are committed to providing you with the highest level of support possible.

SURGEON GENERAL'S WARNING:

Clinical trials have proven that you can't test quality into software.

Visit booth #308 and find out how to improve the quality of your enterprise software.

AutoTester
Your Partner in Software Quality™

6688 N. Central Expressway
Suite 600
Dallas, Texas 75206
www.autotester.com

View ad

"Ferreting out bugs"

1032 Elwell Ct.
Suite 240
Palo Alto, Ca. 94303

AZOR, Inc. is committed to helping high technology companies improve the quality of their software based products.

Outsource Testing:, Test Automation, Contract Testing™

Your product can be independently tested at our facility. AZOR, Inc. can generate the test plan, test design, test cases, test data and scripts and then execute tests, automate tests (using WinRunner®, QA Partner®, Ferret®, etc.) , and report defects to your developers. AZOR can also supplement your own in-house testers with test execution or on-site contractors.

Training: AZOR provides world class two and three day seminars on:

- Best Practice Software Testing™ and Value Added Inspections™
- Software Program Management and Software Test Management
- Software Quality Assurance

Test Execution Product: Ferret® Patented Automated Software Testing System

FERRET is an automated capture/playback system testing tool useful in situations where traditional invasive GUI test tools will not perform adequately. Such as embedded devices, medical systems, real time, legacy, or mixed operating systems, graphics intensive or 3D systems, multi-media, IR devices, set top units, television based web products, factory automation, AC power cycling, etc. FERRET performs automated regression and system testing to verify that code/firmware repairs or software enhancements have not caused additional software bugs.

Test Planning Product: CodePlan™ Project Estimation Tool

CodePlan is a development estimation tool that calculates development effort, staffing needs, and timing for each phase of your project. You provide a few simple inputs on the expected project size (either in lines of code or function points), the organization of your development team, and your project's risk factors.

Consulting:

- Software Development Process Assessment
- Software Project Management
- Software Testing and Quality Assurance
- Software Test Tool Selection and Jump Starts
- Y2K Project Management and Testing

Outsource Testing, Training, Tools, and Consulting

…. Azor helps its customers by providing a complete solution

Omega™ Outsource Testing:

- ✓ ***Faster Time to Market, Reduced Risk***
- ✓ ***Test Plan, Design, Execution, Reporting, and Automation***

Principle Based Software Test Seminar:

- ✓ ***Courses for Developers, QA Engineers, Test Engineers, Management***
- ✓ ***What to do, how to do it, tools to use. Can be tailored for in-house.***

Ferret® Automated Test Execution Tool:

- ✓ ***Works with most all platforms, tool kits, browsers, and O/S Types***
- ✓ ***Completely non-intrusive, easy to learn and use.***

CodePlan™ Estimation Tool:

- ✓ ***Tells you how long the development and testing will take.***
- ✓ ***Helps you get more time to do your job***

Consulting:

- ✓ ***Assessments, Process Improvements, Hands On Projects***
- ✓ ***On-site Testing, QA, Program Management***

For additional information, or to receive a proposal, please contact:

1032 Elwell Court, Suite 240
Palo Alto, CA 94303
Phone: (650) 934-2869
Fax: (650) 934-2860
Email: azor@azor.com
www.azor.com

V1.0 Copyright AZOR, Inc. 1/99

Capital One Description

View ad

What do you mean that you haven't heard of Capital*One*?!

We have a long history and are a leader in the financial services industry. In fact, we are a top ten issuer of the Mastercard and Visa credit cards with over $16 billion in managed loans and more than 16 million customers! Despite our success in the credit card industry, we actually consider ourselves to be an international information-based marketing company. Our success is based upon our proprietary information-based strategy. This strategy involves a partnership between Marketing, Operations, Customer Relations and Information Technology to integrate systems to ultimately "mass customize" the right product to the right customer at the right time and at the right price. Our IBS strategy has contributed to our 20% earnings per share growth over the past three years—and 41% earnings growth for 1998!

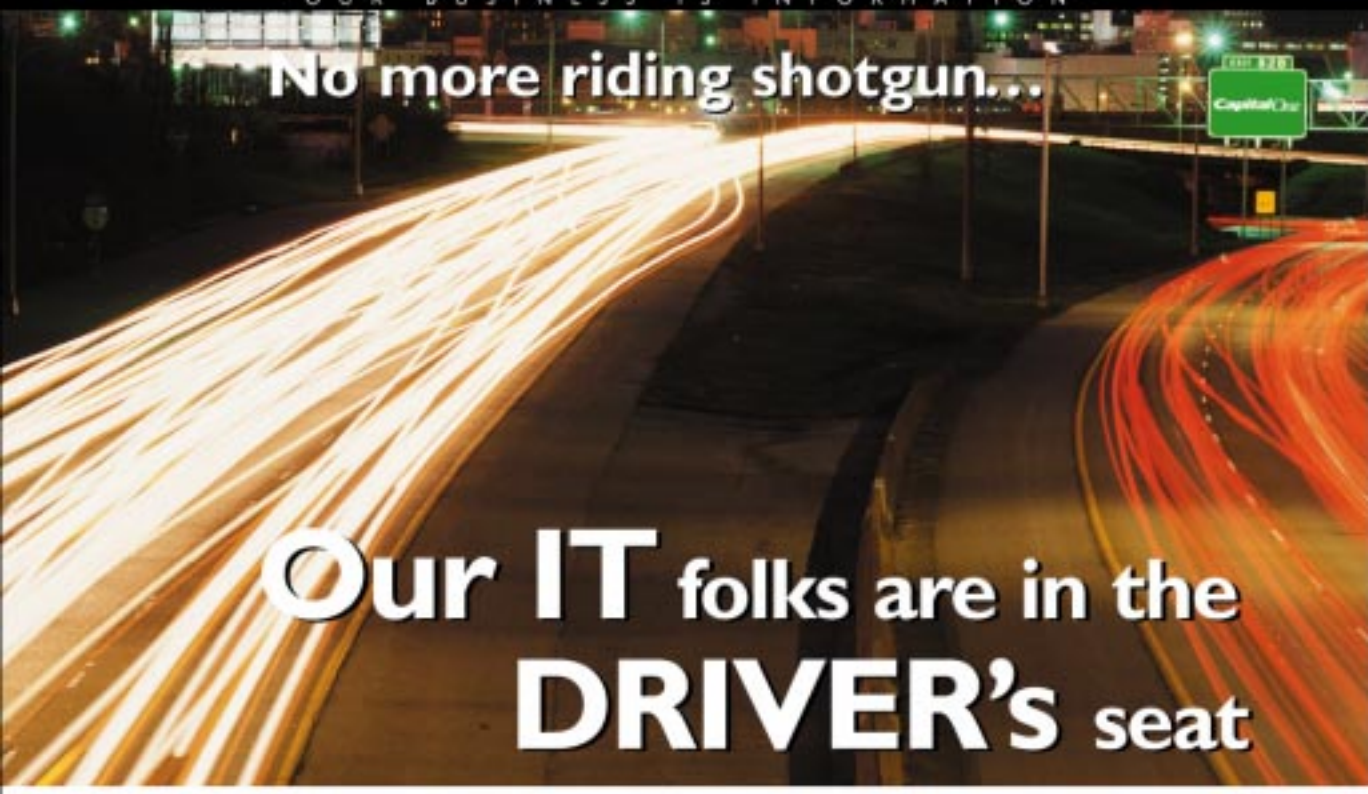
We have now applied this same strategy to the telecommunications industy in our subsidiary – America One. America One is a provider of wireless phones and services, as well as long distance telephone service. The IT group within America One integrates off-the-shelf applications along with software we write ourselves. This software is written with Java in the middle layer, an n-tier architecture and use cases. Both Capital One and America One have cutting edge test, QA and configuration management opportunities! **We are looking for testers, test leads, test harness developers, test database specialists, quality assurance analysts and configuration management specialists to work on a variety of projects**.

Only *One* Company…

- adds 20,000 new customers a day
- has had 20% earnings growth for the last three years and 41% earnings growth in 1998
  has partnered with World Championship Wrestling (a subsidiary of Turner Sports and Time Warner Company) to offer a credit card to its more than 32 million fans
- is the largest (and only) direct marketer of cellular service through its subsidiary, America*One*
- has 16 Terabytes of information total in its data warehouse and is one of the largest users of Oracle databases in the nation
- has benefits from day one -- including three weeks vacation in your first full calendar year, 401K, stock purchase plan, flexible spending accounts, tuition reimbursement and much, much more!

All of this growth has created opportunities organization-wide to join the over 10,000 associates already working at Capital*One*! If you're looking for a unique work environment where career potential is limited by only individual desire…it's time you discovered Capital *One*! A big part of our success is our people. That's why we believe in rewarding them…with an exciting, passionate culture, an inspiring entrepreneurial spirit, and one of the best compensation and benefits packages in the business! If you can't find all this at your current job then it's time for your to check out our website (www.capitalone.com) by clicking below.

"OUR BUSINESS IS INFORMATION"



The right product, to the right person, at the right price, at the right time. It's what we do. But without accurate, clear information, it falls apart. You see, it's not enough simply to provide data—we use our information proactively to *drive* our business strategies.

That's why we're currently seeking experienced, qualified **IT professionals** who are up to the challenge of helping plot the course of an S&P 500 company.

At Capital One, IT professionals contribute to the business ***every single day*** by working with our Marketing and Analysis department to set strategy, identify opportunities and, ultimately, increase the bottom line. We're redefining the way IT contributes to the success of our company.

And best of all, you'll be rewarded with a competitive salary and incentive bonuses. You'll also enjoy an exciting, action-packed company culture and excellent benefits, including medical, dental and vision coverage from day one, stock purchase and 401(k) plans, and three weeks' vacation in the first full calendar year.

So, if you're an IT professional who enjoys the freedom to develop new and innovative models, drive strategy and work in a team environment with a fast-growing company named "one of the top 100 companies to work for" by *Fortune* magazine, then perhaps it's time you considered Capital One.

Capital One®

Candidates interested in the exciting opportunities at Capital One are invited to send a letter of interest and their résumés to **804-967-1483** (*fax*) or **itrecruiting@capitalone.com** (*e-mail*).

For more information, visit our website at **www.capitalone.com**

Capital One is an equal opportunity employer committed to diversity in the workplace. We promote a drug-free work environment.

Computer Associates International, Inc.

View ad

One Computer Associates Plaza
Islandia, NY 11788-7000
Phone: (516) 342-5224
Fax: (516) 342-5329
Web: http://www.cai.com

Computer Associates International, Inc. (NYSE: CA), with headquarters in Islandia, N.Y., is the world leader in mission critical business software. The company develops, licenses and supports more than 500 integrated solutions that leverage technology for a competitive business advantage in an emerging global market. These solutions include enterprise computing and information management, database and application development, Internet/Intranet, manufacturing and financial applications. CA has nearly 13,000 people in 160 offices in 43 countries and had revenue of $5.1 billion in fiscal year 1998.

CA solutions featured at Quality Week '99 include Endevor, CA-Fix/2000 for IV&V, and the CA testing suite, including CA-Verify, CA-Datamacs/II, and CA-TestCoverage/2000.

The Endevor family of products automates change and configuration management throughout the enterprise on MVS, NT and UNIX platforms, helping you maintain the integrity of your organization's software assets and create the highest quality systems.

CA-Fix/2000 performs Independent Verification and Validation (IV&V) of COBOL code for Year 2000 compliance, identifying at-risk year fields that may have been missed by manual remediation or less sophisticated tools.

CA-Verify is an automated regression testing tool for mainframe online applications, automatically detecting differences that the human eye may miss. Its Rules Function streamlines testing by allowing users to predefine expected changes. CA-Datamacs/II automates test data generation and includes date aging capabilities.
CA-TestCoverage/2000 for COBOL batch programs help testers ensure that test data is adequate and that key execution paths have been exercised.

CHOOSE THE WRONG VENDOR, AND IT COULD DESTROY YOUR ENTIRE ENTERPRISE

NOW is the time to find out if you have errors in your renovated code!
NOW it is less expensive to find and fix any problems!
NOW you need Independent Verification and Validation (IV&V)!
NOW you need best-of-breed automated testing tools!

Computer Associates recognizes the necessity for thorough Year 2000 conversions. Using powerful data flow analysis engines, you can validate that code has been changed correctly, identify year fields that were missed (false negatives), and pinpoint bugs before testing, when they are easier and less expensive to correct.

For ANSI standard COBOL (IBM 68, 74, and 85), MVS, VSE, Fujitsu MSP, and Data General:
CA-Fix/2000™

For CA-IDMS® COBOL and Unisys COBOL:
UNICAST/2000

Computer Associates also offers a complete suite of testing tools for faster and more accurate Year 2000 testing:

CA-Verify®	Automated regression testing; migration, stress, and concurrency testing
CA-Datamacs™/II	Test data generation; date aging
CA-TestCoverage/2000™	Automated test coverage analysis
CA-InterTest®	CICS source-level debugging
CA-InterTest®/Batch	Batch source-level debugging
CA-Realia® II Workbench™	CICS and Batch source-level debugging
CA-SymDump®	CICS abend resolution
CA-Optimizer®/II	Batch abend resolution; test coverage analysis; optimization
CA-Accuchek™	File comparison

For more information, call 1-888-4-2000YR, or visit our website at www.cai.com/y2k

CA DISCOVERY 2000™

COMPUTER ASSOCIATES®
Software superior by design.

©1999 Computer Associates International, Inc. All trademarks used herein belong to their respective companies.

Compuware Corporation

Compuware Corporation is a leading worldwide provider of software products and professional services.Compuware productivity solutions help 14,000 of the world's largest corporations more efficiently maintain and enhance their most critical business applications. Providing immediate and measurable return on information technology investments, Compuware products and services improve quality, lower costs and increase the speed at which systems can be developed, implemented and supported. Compuware employs more than 11,000 information technology professionals worldwide, including more than 6,600 in its professional services organization. With calendar 998 revenues in excess of $1.5 billion, Compuware is the world leader in client/server development technology.

The Compuware QACenter product family offers the best in automated testing technology for today's enterprise needs. From mainframe to client/server to web, from test management through validation, from unit through load testing, for Year 2000 and euro conversions-QACenter tools are helping companies achieve consistent, dependable application performance

The Products:

QACenter products are available separately or as a part of an integrated family of products that includes:

QARun- comprehensive capabilities for thorough, accurate client/server testing.

QALoad- Load testing for client/server applications

QADirector- managing the testing process across the enterprise.

QAHiperstation- broad capabilities for testing VTAM-based applications.

QABatch- regression testing for MVS batch applications.

QAPlayback- advanced capabilities for testing CICS-based applications.

QAHiperstation+ - testing mainframe applications from the desktop.

QAPlayback+- windows-based interface for testing mainframe-based applications from the desktop.

QASolutions- process, people and products for quality testing.

View ad

ready or not?

A better way to work
www.cisco.com/jobs

You can stop wondering. The Internet is definitely here to stay. And if you think it's big now, just wait awhile and see. Better yet, why wait? Join Cisco Systems and get a head start on the future. Here's what we bring to the table-a majority share of the networked world, mind-boggling success, and a highly evolved working culture. The Internet is a revolution that is changing the world. And no matter who you are, it has touched your life already. Which means that the question ready or not is about more than just asking if you're ready to start. It's also about reminding you that you already have.

Ready or not? Now is the time to act. Visit us online at www.cisco.com/jobs for detailed listings of hundreds of current job opportunities. It's quick, easy and well worth it.

ready *or* not?

A better way to work.

www.cisco.com/jobs

You can stop wondering. The Internet is definitely here to stay. And if you think it's big now, just wait awhile and see. Better yet, why wait? Join Cisco Systems and get a head start on the future. Here's what we bring to the table—a majority share of the networked world, mind-boggling success, and a highly evolved working culture. The Internet is a revolution that is changing the world. And no matter who you are, it has touched your life already. Which means that the question *ready or not?* is about more than just asking if you're ready to start. It's also about reminding you that you already have.

- **Account Managers**
- **Customer Relationship Managers**
- **Customer Support Engineers**
- **Financial Analysts**
- **Hardware Engineers**
- **Human Resource Professionals**
- **IS Project Managers**
- **IT Analysts/Engineers**
- **Manufacturing Test Development Engineers**
- **Marcom Specialists**
- **Network Consulting Engineers**
- **New Product Introduction Engineers**
- **Product Marketing Engineers/Managers**
- **Professional Services (multiple locations)**
- **Project Engineers/Managers**
- **Quality Assurance Engineers**
- **Software Engineers**
- **Support Engineers/Managers**
- **Systems Administrators/ Managers**
- **Systems Engineers**
- **Test Development Engineers**
- **Web Developers**
- **Managers & Directors (various departments)**

Ready or not? Now is the time to act. To apply directly, please submit your resume along with ad code: **QW99** by E-MAIL: **jobs@cisco.com; (ASCII text only)**; by MAIL: **Cisco Systems, Inc., Human Resources, P.O. Box 640730, San Jose, CA 95164-0730**; or by FAX: **(800) 818-9201**. And don't forget to visit us online at **www.cisco.com/jobs** for detailed listings of hundreds of current job opportunities. It's quick, easy and well worth it.

CISCO SYSTEMS

EMPOWERING THE INTERNET GENERATION™

EOE

CS VERILOG

View ad

3010 LBJ Freeway, Suite 900
Dallas, TX 75234

Phone: 972-241-6595
800-424-3095
Fax: 972-241-6594
Email: info@verilogusa.com
Web: www.verilogusa.com

CS VERILOG is a world leader in software development solutions for real-time, distributed, embedded, and critical systems. CS VERILOG solutions encompass technologies such as UML, SDL, MSC, software metrics, test coverage, and graphical reverse engineering to generate products that cover the complete development phase: system modeling, simulation, code generation, testing, and quality assessment.

CS VERILOG is a subsidiary of the CS Communications & Systems Group, a telecommunica-tions equipment and systems integration company listed on the Paris Stock Exchange. The company's US headquarters are in Dallas, TX, with international headquarters in Toulouse, France, and distributors worldwide. VERILOG has become one of the leading software companies providing technical CASE solutions and related services to Fortune 500 corporations in the aerospace, defense, telecommunications, transportation, medical and energy industries.
Visit CS VERILOG's Web site at **www.verilogusa.com**.

Products:

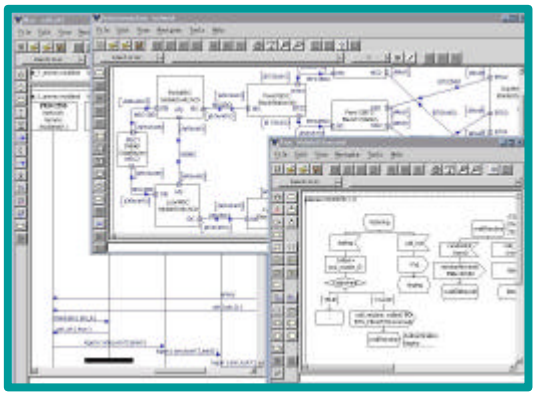
ObjectGEODE is a complete integrated development environment for system modeling, verification and validation, code generation, and testing of real-time and distributed applications. ObjectGEODE supports a coherent integration of complementary object-oriented and real-time approaches based on UML, SDL and MSC standard notations.

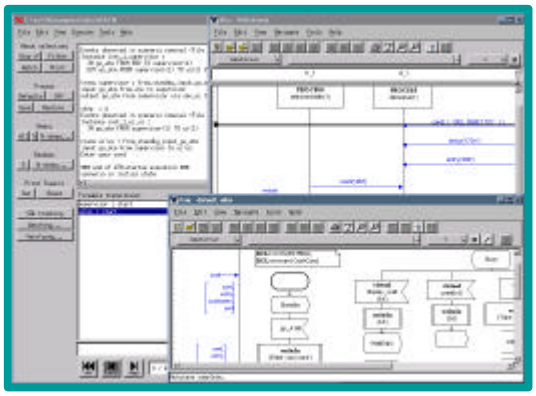
LOGISCOPE is a set of tools providing quality and test analysis through software metrics, test coverage, and reverse engineering, enabling effective software production and maintenance.

SCADE is a solution for developing safety critical real time systems. SCADE is based on a formal technology developed in cooperation with Aerospatiale and Schneider Electric, and supports software design using a block diagram approach, reuse of control engineering libraries, 100% code generation, and testing.

*Object*GEODE

Integrated Development Environment for Embedded, Real-Time, and Distributed Applications





•System or Software Modeling
- Scalable to Handle Small to Very

Used by System and Software Engineers

• Based on International Standards
- UML from OMG
- SDL and MSC from ITU
- ASN.1 and TTCN from ISO

• Comprehensive Graphical Modeling

- Fully Object-oriented
- Use Cases
- Architectural Patterns
- Complete Real-time Behavioral Modeling

• Full Blown Simulation
- Interactive, Random, and Exhaustive

Verification of Deadlock, Race Conditions, Unreachable Parts, etc.
Validation of Design
Performance Evaluation

• Production-Quality Code Generation
- Readable, Small, Efficient, and Versatile

, Unix, Win32, Various Micro-Controllers, etc.
On-target Back-to-Design Debugging

• Strong Testing Capabilities
- Automatic Test Case Generation

System Test Case Execution

• Integrated with the Best-in-Class
- Requirements: QSS DOORS
- General-purpose: Rational ROSE

Others: Please ask

VERILOG

www.verilogusa.com
info@verilogusa.com
1-800-424-3095

CYRANO, Inc.

View ad

26 Parker Street
Newburyport, MA 01950
Phone: (978) 462-0737
Fax: (978) 462-4755
Email: info@cyrano.com
Web: www.cyrano.com

CYRANO is a leading provider of products and services designed to help IT organizations deploy high quality/high performance Web and client/server applications.

CYRANO provides end to end automated testing for client or web browsers through any middle tiers and on to the database server.

The products range from web based functional and availability testing to load and stress testing of web servers. Tools addressing performance analysis of SQL transactions, database server stress testing, automated coding standardization and network bandwith improvement are also available.

With a focus towards e-business and proof of concept planning and execution, CYRANO solutions and services have helped many IT departments with system deployment, capacity planning, performance and availability testing and more.

To learn more about CYRANO, Inc., call us at 800.714.4900 or visit our web site at **www.cyrano.com**

Ready to Deploy? Relieve the Stress.

CYRANO provides the most powerful solution to test your applications deployment and properly size your hardware.

CYRANO gives you the confidence you need to **guarantee a smooth and controlled deployment.** Our solutions help you to test scalability, functionality and performance. Your hardware will be properly sized. Your project will meet roll out dates.

In a world with no time for mistakes, **CYRANO is your insurance.** Relieve the Stress. Enjoy a fully controlled deployment.

CYRANO has helped companies to accelerate their ERP or in-house applications deployment by successfully sizing their hardware.

Call us to learn how at
800.714.4900

CYRANO

USA: 978.462.0737 · UK: +44 1635 876 876 · France: +33 1 56 33 40 00 · Australia: +612 99 56 38 09 · Germany: +49 6196 400 827
Singapore: +65 870 0663 · Spain: +39 091 524 74 00
www.cyrano.com

Data Dimensions, Inc.

View ad

The Data Dimensions' Test Centers provide comprehensive Quality Assurance and Test services including Testing Methodology, Best Practices, Project Management, Year 2000 work, and Training.

The Test Centers offer heterogeneous testing environments from the PC to mainframe, providing both LAN and WAN testing in a non-production, risk-free environment.

Data Dimensions, Inc. provides Enterprise Integration Solutions (EIS) to companies committed to aligning their Information Technology (IT) infrastructure and business strategies. Since 1968, Data Dimensions has provided IT leadership to national and international companies, including many among the Fortune 500. For more information about Data Dimensions, visit the company's Web site at http://www.data-dimensions.com

Data Dimensions, Inc.™ (DDI) Offers Comprehensive QA and Test Services

The Data Dimensions' Test Centers provide comprehensive Quality Assurance and Test services include Testing Methodology, Best Practices, Project Management, Year 2000 work, and Training.

The Test Centers offer heterogeneous testing environments from the PC to mainframe, providing both LAN and WAN testing in a non-production, risk-free environment.

We offer complete outsourcing solutions. Our facilities include both onshore and offshore solutions centers, a comprehensive lab environment for testing client software, and a high capacity multiplatform system environment.

Our clients have included Microsoft, Hewlett Packard, Texas Instrument, Xerox, Intel, Kodak, GTE Intelligent Networks.

Data Dimensions, Inc. provides Enterprise Integration Solutions (EIS) to companies committed to aligning their Information Technology (IT) infrastructure and business strategies. Data Dimensions leverages a company's IT investment by applying in-depth expertise through a comprehensive suite of knowledge-based services and products. Since 1968, Data Dimensions has provided IT leadership to national and international companies, including many among the Fortune 500.

877.DIAL.DDI (877.342.5334) • www.data-dimensions.com

Data Dimensions®

©1998 Data Dimensions, Inc. All Rights Reserved.
Data Dimensions is a registered trademark of
Data Dimensions, Inc. All Rights Reserved.

ErgoLight

View ad

***ErgoLight*™** Ltd. develops and markets solutions for enhancing the usability of Windows applications. The term "**usability**" is typically interpreted in various ways, to mean friendly appearance, ease of use, ease of learning, user productivity and resisting user's errors. The solutions offer **usability profiles**, including operation profiles and failure profiles that are based on human factors. The solutions allow automated capturing of the user operation, as well as of the user's confusion, useful for **usability lab**s and for **remote testing**. The technology is based on a unique, patent pending, invention, which merges psychological considerations in software tools.

ErgoLight addresses the full development cycle, from product specification to deployment. The procedures that ***ErgoLight*** automates are mainly of usability validation, but also of design, on-line assistance and Help Desk. ***ErgoLight*** offers solutions for software developers, system integrators, testers and technical support personnel. The first solutions run on Windows 95, 98 and NT.

ErgoLight professional solutions addresses the whole range of interactive Windows applications, including education, mission critical and safety critical systems and production applications. Particularly challenging applications of using ***ErgoLight*** are for validating the forgiveness to user's errors of "safety critical" systems, such as traffic control, nuclear control, military, health and emergency systems. The capability of ***ErgoLight*** to identify mode errors is most useful for validating the operation of modal systems, such as electrical and electronic equipment.

ErgoLight express version is an easy-to-use version of the professional version that allows QA engineers to identify error prone controls by statistics, even when running in silent mode.

ErgoLight Ltd., 6 Giv'on St.,
Haifa 34335, Israel
Email: info@ergolight-sw.com
Web: www.ergolight-sw.com

Tel: +972-4-826-3012
Fax: +972-4-825-8199
USA: 1-877-Use-Ergo
1-877-873-3746

Usability Software

Need to Redesign Your GUI?
Need to Improve the On line Help?
Need to Enhance the User Documentation?

Don't rely on standards, style guides or design rules!

Standards were arranged by programmers . . .for programmers!
Most users aren't familiar with the guides and don't obey the rules.

Don't risk trying to educate the users!

Most users prefer to change vendors rather than be educated.

Test your product's usability with real users.

At the prototyping and alpha testing stages:

Add ***ErgoLight*** to your laboratory testing procedures,
to capture the user's actions and the operation modes.

At the beta testing and product deployment stages:

Install ***ErgoLight Agent*** on the user's PC as part of your product installation, to capture the users' intentions when they encounter difficulties using your product.

Get usability data that you never had before:

- •0 Statistics about your product's usefulness
- •1 Objective measures of user operation.

Understand the user's errors:

- •2 Backtrack the user's actions including the product's operating mode at the time of the action
- •3 Use integrated analysis tools based on human factors.

ErgoLight Ltd., 6 Giv'on St.,
Haifa 34335, Israel
Email: info@ergolight-sw.com
Web: www.ergolight-sw.com

Tel: +972-4-826-3012
Fax: +972-4-825-8199
USA: 1-877-Use-Ergo
1-877-873-3746

Hall Kinion

View ad

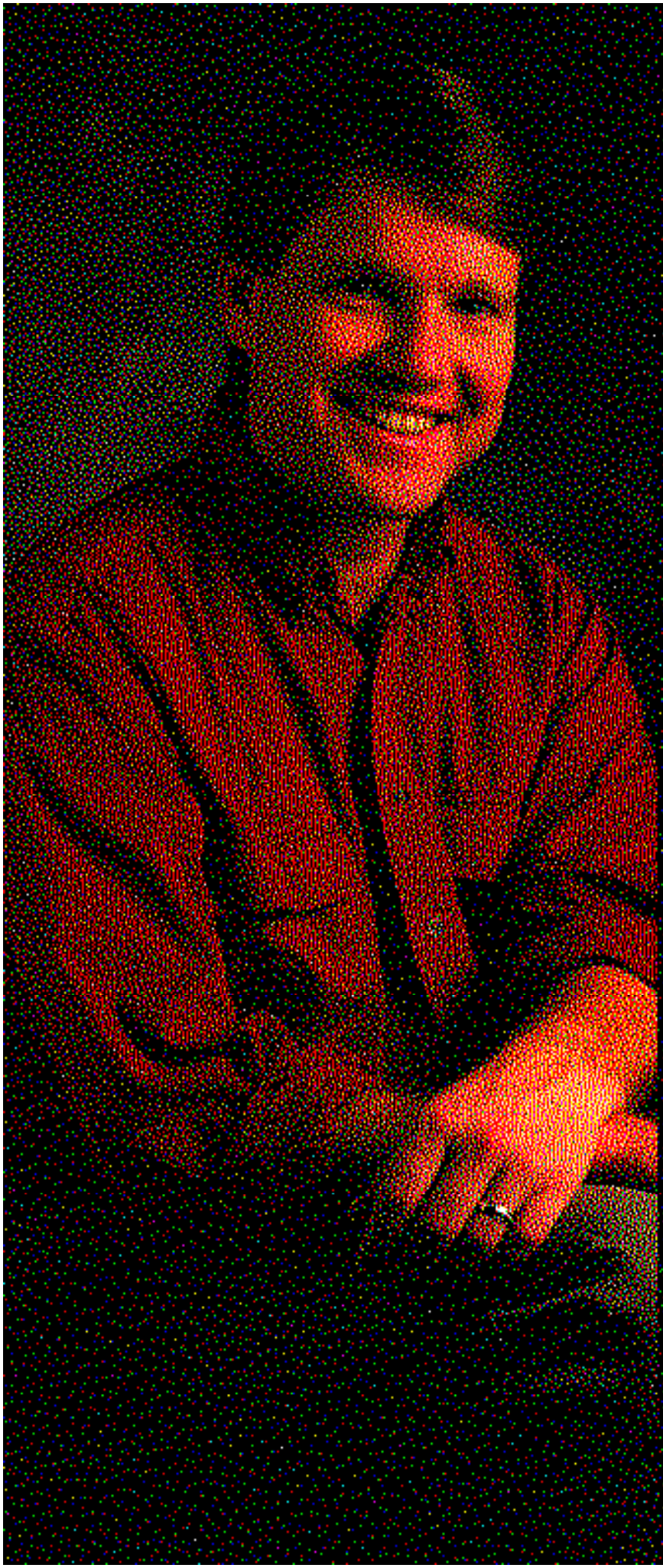
Hall Kinion's objective is to provide efficient and high quality contract and permanent IT professionals to high technology clients and to become the "agent of choice" for IT professionals.

The Contract Services Division **is divided into nine types of Practice Groups. These groups focus on those technologies widely used by Hall Kinion clients in the development of their products *(Windows, UNIX, QA, Hardware, Writers, Internet, NET, IS and Technology Support)*.**

Technical Recruiting Agents **(TRAs) are responsible for recruiting IT professionals in their particular fields of specialization. TRAs build long-term relationships to understand their IT professional's preferences and capabilities and monitor their availability, progress and job satisfaction. *Account Managers* are responsible for relationships with Hall Kinion's clients in different departments and at different management levels to provide quicker ,more accurate placements.**

The Permanent Placement Division **includes permanent placement to R&D and IS departments of high technology firms; executive searches on a retainer basis at a national level; and international recruiting from India, Japan, China and Korea, to name a few. *Recruiters* are primarily responsible for establishing relationships with clients needing permanent IT professionals and matching the needs of the client.**

Hall Kinion's goal is to attract and retain the highest quality IT professionals available. These IT professionals are provided with, among other things, advance notice of state-of-the-art assignments and prequisites, including participation in the Company's 401(k) plan, access to medical and dental benefits and inclusion in the Company's stock options plan to become the "agent of choice" for IT professionals.



"WHY DO I USE HALL KINION? BECAUSE OF THE VALUE THEY ADD TO MY CAREER."

BILL CLARK, Star Contractor
Represented by Hall Kinion

BILL CLARK IS PRETTY UP-FRONT about one of the things he likes most about contracting. "It's the assignments," he says cheerfully. And as a contractor, he knows challenging assignments in emerging technologies broaden both his career and his market value. Today he's helping to develop intelligent agents for database management applications – an "adventure in networking" arranged by Hall Kinion.

• We connect the stars of technical contracting with the best contracts in the Silicon Valleys of the world. Call one of our personal recruiting agents today.

1.888.246.9551
Résumés: stars@hallkinion.com
http://www.hallkinion.com

HALL KINION

Information Balance Inc.

View ad

Information Balance offers Information Technology consulting and technical training services, encompassing solutions in Information Systems Planning, Project Management, Application Development/Project Outsourcing, Systems Integration, Software Testing and Quality Assurance.

Building on years of training and consulting experience, and expertise with some of the leading Year 2000 tools, we also offer a complete cycle of Year 2000 services, from Assessment through Renovation to Quality Assurance, Testing, and Project Review.

Our Year 2000 facilities are equipped with state-of-the-art hardware, software, and tools, high-speed communication lines, and highly skilled professionals. Senior staff have managed and implemented multiple large-scale development and Year 2000 projects for Fortune 500 companies.

Information Balance is uniquely positioned to assist organizations as they complete Year 2000 programs, and to help leverage their experiences for future development. Services include:

IBI*TEST2000, a turnkey testing methodology package, is a refinement of the best practices and repeatable processes that our experts have developed during the testing phase of Year 2000 projects. Comprised of test plans, templates, boilerplates, mentoring and training, this testing pathway will speed up and streamline this crucial phase.
Multi-Level Quality Assurance Services will catch missed or erroneous renovations before testing, saving both valuable time and effort.
Y2K Program Review to independently assess program processes, ensuring projects are on-track, on-budget, and on-time.
A highly specialized **Mobile Factory**, designed for clients whose geographic location, time limitations, or security needs necessitate on-site delivery. Able to assess, renovate, test, and perform Quality Assurance at the client site, the Mobile Factory is particularly valuable as the 2000 deadline approaches.

Getting you through to the Year 2000....
And into the new millennium

For Y2K and post-Y2K QA and Testing,
We are Your Information Technology Solutions Partner

Information Balance is uniquely positioned to assist your company in completing your Year 2000 projects and in leveraging your hard-earned experience for future development efforts.

We provide consulting and technical training solutions for both the public and private sectors in Software Testing, Testing Methodologies, Quality Assurance, Information Systems Planning, Project Management, Application Development/Project Outsourcing and Systems Integration. Quality Assurance techniques and Testing Methodologies are firmly entrenched in all our projects to ensure a quality mindset throughout all phases of development.

Building on years of training and consulting experience, and expertise with some of the leading Year 2000 tools, we also offer a complete cycle of Year 2000 services, from Assessment through Renovation to Quality Assurance (IV&V), Testing, and Project Review.

Quality focused. Guaranteed delivery.

Information Balance

For more information please contact our International Headquarters at:

416-962-5235 Fax: **416-962-8020**

E-mail: **infobal@infobal.com**

www.infobal.com

View ad

TheConsultingGROUP

823 Commerce Drive Oak Brook, IL 60523 877-9IT-NSQM
peggythomas@interim.com www.interim.com/technology

Interim Technology, The Consulting Group is a global leader in providing Software Quality Management solutions to premier companies around the world. As a founding member of the Quality Assurance Institute, more than 30 years experience, more than 5,000 full-time salaried consultants working from an international network of offices, and more Certified Quality Analysts (CQAs) on staff than any other consulting firm; Interim Technology offers its client base expertise that stands alone in the industry.

Our world-class expertise along with our unique approach to service delivery provides an unbeatable combination in exceeding our clients' expectations with each and every engagement. Our rigorous performance standards, commitment to staff development and dedication to continuous improvement ensure satisfaction for our clients.

Our Software Quality Management (SQM) practice provides the best guarantee of delivering quality the first time, every time. Our services include: Software Validation and Testing, Tool Automation Evaluation and Implementation, Quality Process Assessment and Implementation, Quality and Productivity Metrics, as well as Quality Process training for both users and IT professionals. These services are supported by a proprietary set of tools and methodologies, which include our state-of-the-art hypertext VALI/TEST Pro (Validation and Testing methodology) and Interim Technology/SQM Tool Suite$^{sm.}$. The Interim Technology/SQM Tool Suite^{sm} provides comprehensive planning and management capabilities throughout the software development process. Additional components help you size software development activities, as well as build and maintain a comprehensive metrics repository for software engineering projects.

In addition, Interim Technology offers a full range of educational programs that can be customized for your organization's needs, as well as for individual mentorship and coaching. All are designed to instill and reinforce the understanding of quality principles. The programs address management and implementation issues, and fully support all elements of our services and products.

In case you were wondering, here is our resume.

Interim Technology, The Consulting Group with over 30 years experience, is a world leader for Software Quality Management solutions.

With offices around the world, and a founding member of the Quality Assurance Institute, our scope of operations encompasses over 5,000 full-time salaried consultants, and more CQAs than any other consulting firm on this planet.

Now, can we hear from you?

We are currently seeking qualified professionals in the following areas:

Test Automation Architects
Test Automation Analysts
Test Automation Engineers

The skill levels required:
•Minimum of 3 years experience with structured testing
•Automated test tools experience such as Rational SQA, Mercury Interactive •Segue and Compuware

We also require a 1 year minimum in programming, with Visual Basic, Java or C language preferred.

Please forward your detailed resume to The Director of National Recruiting at the address below. Or e-mail: interview@interim.com.
It just might be the best career move of your life, in case you were wondering.

Interim®
TECHNOLOGY
THE CONSULTING GROUP
INFORMATION'S POWERFUL MINDS

Interim Technology, 823 Commerce Drive, Oak Brook, IL 60521 • Phone (630) 645-8800 Fax (630) 645-8801
E-mail: www.interim.com/technology

EOE

International Institute for Software Testing

View ad

IIST is an educational and professional development organization that has been founded to meet the following goal:

To promote a disciplined approach to software testing and to caution against ad hoc testing by non-qualified individuals and groups.

To achieve this goal, IIST will operate according to the following charter:

- To promote and provide education and certification of software testing professionals around the world.
- To create a pool of qualified software testing professionals to meet the needs of testing organizations.
- To provide assistance and guidance to members, both corporate and individuals, performing testing of all types of software systems.
- To provide a framework for assessing organizational testing practices and procedures.
- To partner with different software quality groups around the world to assure continuity of information flow among testing professionals.
- To serve as a clearinghouse of services, methods, and tools supporting the discipline of software testing.
- To provide an open forum for discussing different testing issues.

IIST offers a comprehensive training program for software testing and quality professionals. IIST offers a certification of for software testing professionals that is based on formal training as well as job experience requirement. IIST sponsors the International Conference on Practical Software Quality Techniques (www.softdim.com/psqt)

Details for the institutes programs and offerings including the Certification for software testing professionals are at www.softdim.com. For information, call (651) 306-1387.

International Institute for Software Testing

IIST is an educational and professional development organization that has been founded to meet the following goal:

To promote a disciplined approach to software testing and to caution against ad hoc testing by non-qualified individuals and groups.

IIST offers a comprehensive training program for software testing and quality professionals.

Details for the institutes programs and offerings including the Certification for software testing professionals are at www.softdim.com. For information, call (651) 306-1387.

International Institute for Software Testing

Promoting Disciplined Software Testing Practices

IIST is an educational and professional development organization that has been founded to meet the following goal:

To promote a disciplined approach to software testing and to caution against ad hoc testing by non-qualified individuals and groups.

To achieve this goal, IIST operates according to the following charter:

- To promote and provide education and certification of software testing professionals around the world.
- To create a pool of qualified software testing professionals to meet the needs of testing organizations.
- To provide assistance and guidance to members, both corporate and individuals, performing testing of all types of software systems.
- To provide a framework for assessing organizational testing practices and procedures.
- To partner with different software quality groups around the world to assure continuity of information flow among testing professionals.
- To serve as a clearinghouse of services, methods, and tools supporting the discipline of software testing.
- To provide an open forum for discussing different testing issues.

IIST Offers Certification for Software Testing Professionals

The purpose of the certification program is to teach individuals from different disciplines sound and effective testing techniques and methods and to certify them as software testing professionals

Objectives of the Certified Software Testing Professional Program (CSTP):

- Help individuals develop their software testing skills through education.
- Provide member organizations access to a pool of software test professionals
- Provide education that addresses the curriculum requirements of certification.
- Prepare candidates for a wider range of software testing assignments.
- Complement company in-house and on-the-job training programs.
- Provide a record of courses completed
- Provide professional recognition and career enhancement.

For more information, contact IIST at (651) 306-1387 or e-mail at iist@softdim.com.
Visit our website at www.softdim.com

Intrinsa Corporation

View ad

444 Castro Street, Suite 130
Mountain View, CA 94041
650-526-8965 (Direct)
650-254-0350 (Fax)

Visit Intrinsa's at: **http://www.intrinsa.com** and discover PREfix Automated Code Reviewer software, the new standard for improving software quality from Intrinsa Corporation. PREfix uses patented technology to identify crash-causing defects in C/C++ source code without the need for test cases or instrumentation of the code, and it works on Windows and UNIX platforms. Find out why Fortune 500 companies such as AT&T, Autodesk, Cisco, HP, Microsoft, and Sun have committed to improving their products through PREfix.

Why is PREfix Better?
Catching defects sooner can significantly reduce the cost of error correction. PREfix enables the earliest identification of defects – as soon as the code is created -- providing substantial savings over run-time analysis or traditional debug tools.

Using Intrinsa's patented technology, PREfix can uncover crash causing defects that are virtually impossible to find using other software testing tools. Even better, PREfix operates without test cases and works on components as well as incomplete (non-executable) or complete applications.

Rather than simply pointing to where the code failed, PREfix actually tracks and displays the execution path that leads to the defect. Defect reports are also prioritized automatically, making it easier to target the most significant defects first. Best of all, PREfix can be integrated into a project's build process for automatic "round the clock" code analysis and improved project management.

Contact Intrinsa at 1-888-CODEFIX and learn more about how PREfix can benefit you.

intrinsa

Looking for Bugs? Why Wait for QA?

Test Software During Development with PREfix® Automated Code Reviewer™

NEW version 3.0

Discover PREfix Automated Code Reviewer software, the new standard for improving software quality from Intrinsa Corporation. PREfix uses patented technology to identify crash-causing defects in C/C++ source code without the need for test cases or instrumentation of the code, and it works on Windows and UNIX platforms.

Find out why Fortune 500 companies such as AT&T, Autodesk, Cisco, HP, Microsoft, and Sun have committed to improving their products by incorporating PREfix into their software development process. By finding significant defects earlier, PREfix can help you

- improve your software quality
- make your developers more productive
- fix bugs in minutes, not hours
- reduce the cost of fixing errors/bugs
- accelerate your time to market

Explore the pages that follow and see the many ways that PREfix can benefit you. And when you are ready to transform your software development process with the new standard for improving software quality, contact Intrinsa at 1-888-CODEFIX.

USE PREfix TO:

- **Identify Bugs Early**
- **Find Significant, Difficult-to-Locate Bugs**
- **Determine Which Bugs to Fix First**
- **Fix Bugs Faster**
- **Validate Software Component Integration**

Find Intrinsa At Software Quality Week Booth #201

intrinsa

Intrinsa Corporation • 444 Castro Street, Suite 130 • Mountain View, CA 94041
650.390.8600 • fax 650.254.0350 • www.intrinsa.com

KeyLabs, Inc.

View ad

KeyLabs has the world's largest independent, professional testing lab dedicated to testing hardware and software products in network environments. With its state of the art testing facility, KeyLabs delivers results quickly and accurately, allowing you to keep up with the pace of technology while maintaining costs and productivity.

KeyLabs' expert personnel have years of experience in test planning, design, and implementation. We provide innovative solutions to your testing needs and back that up with reliable and competent test results.

KeyLabs provides a full range of quality industry-standard tests and testing tools. Whether testing for Year 2000 or your own specialized need, KeyLabs can provide a solution for you.

* **Custom Testing Services**
* **Load/Stress Testing**
* **Y2K Testing**
* **Benchmark Testing**
* **Performance & Scalability Testing**
* **Configuration Validation**
* **Interoperability Testing**
* **E-Commerce Testing**
* **Certification Programs**
* **QA Testing**
* **Functionality Testing**
* **Localization Testing**
* **Bug Reporting**

KeyLabs has a flexible lab with more than 800 networked PC's that can be configured to simulate from one to tens of thousands of users, multiple network platforms, and a modem pool that boasts over 100 true dial-up connections. With its unequaled test lab and experts with years of experience, KeyLabs can simulate your network environment, your Internet/Intranet and test your E-Commerce business model.

Your full-service testing solution partners

Eric Bowden, Account Mgr.; Rana Cook and John Conarroe, Technical Staff; Pete Nelson, Program Manager; and Shelly Linschoten, MarCom (right sidebar)

KeyLabs' expertise and scalable lab provide today's technologically advanced companies with unequaled testing services.

The experts in network testing

Today's customers are more knowledgeable than ever before. They require products that not only solve business needs but also work as advertised. This is why it makes sense to call in the experts at KeyLabs. Whether you need to supplement your own in-house lab or you'd like to create a testing partnership to handle the entire process, KeyLabs puts your hardware and software to the test. We offer independent verification for your customers' satisfaction and your peace of mind.

As the world's largest independent testing lab dedicated to testing hardware and software products in network environments, KeyLabs delivers results quickly, accurately and cost-effectively. Choose KeyLabs and keep pace with your product launch schedules, reduce your lab expenses and eliminate engineer burnout.

KEYLABS DELIVERS SOLUTIONS

At KeyLabs, we've moved beyond task-oriented testing. We're a full-service solutions provider. Testing Managers choose KeyLabs to be an extension of their own in-house test lab. We reduce expenses and stress by providing you with the technical staff, the environment and the testing expertise needed during peak times. Product Managers use us to verify internal results, to provide competitive analysis and to run industry certifications. In fact, we become your company's one-stop shop for quality testing services and solutions.

From the time your product is born on a white board until the time it's retired, KeyLabs is there every step of the way, evaluating technologies, streamlining the product life cycle and regressing bug fixes.

"KeyLabs by far had the greatest degree of customer-service excellence and a we'll-make-it-work attitude."

James Leonard
Sun Microsystems

KeyLabs™
www.keylabs.com

McCabe & Associates, Inc.

View ad

9861 Broken Land Parkway
Fourth Floor
Columbia, Maryland 21046
Toll Free: 1-800-638-6316
Phone: 410-381-3710
Fax: 410-995-1528
Web: **http://www.mccabe.com**

McCabe & Associates, Inc. is an international leader in software solutions for improving the quality and reliability of enterprise software applications. Based on over twenty years of research and experience in software quality, testing, and reengineering, McCabe IQ is an integrated approach to building quality into your software development lifecycle. Using Thomas McCabe's groundbreaking research in software testing and complexity, it combines a strong theoretical foundation with practical, visual tools to help organizations:

* Accurately assess software quality, complexity, and testing requirements
* Pinpoint potential problems and high-risk areas
* Eliminate redundant code
* Thoroughly tests applications
* Validate coverage of high-risk areas

Since 1977, we have been working closely with our customers to improve the quality of large-scale, mission-critical software. Many of the world's most influential corporations and government organizations have used our products successfully to test, reengineer, and verify the quality of over 30 billion lines of products for source code analysis and testing. And our structured testing methodology has been adopted and published by the National Institute of Standards and Technology (NIST).

Solidly based on source code analysis technology, McCabe IQ integrates the build, test, and change phases of software development. By linking these processes through advanced visualization, industry standard metrics, and dynamic monitoring, McCabe IQ raises the quality standards of your deliverables, reduces your overall development costs, decreases your time to market, and lets you focus your resources where they will have the greatest impact, today and tomorrow.

Find out how to deploy flawless applications in Internet time...

Obtain a **FREE** copy of the META Group research note entitled ***"Is the Cost of Application Quality Worth It?"***

Learn how cost of quality principles and McCabe & Associate's Integrated Quality™ framework can improve your organization's application quality and reduce costs by:

- **Using software quality to promote customer loyalty.**
- **Decreasing the risk of failure due to software defects.**
- **Decreasing the time between new software releases.**
- **Improving software test effectiveness prior to deployment.**

McCabe & Associates has been improving ***quality and test effectiveness*** *of Fortune 1000 organizations with McCabe IQ™ -- building customer loyalty through software quality assurance, testing and reengineering. Respond today and learn how McCabe IQ can help your organization!*

Visit us at Booth 102

Visit our **Web site @ www.mccabe.com/solutions**

Contact us at **@1-800-638-6316**

View ad

MicroCrafts Inc
A Metamor Software Solutions Company
8700 148th Avenue NE
Redmond, WA 98052
Tele: (425) 250-0000
Fax: (425) 250-0100
Email: info@microcrafts.com
Web: www.microcrafts.com

MicroCrafts is a leading software engineering firm specializing in the design, development, quality assurance and testing of Windows software applications, components and business solutions. We are experts in technology, project management, software process improvement, and quality assurance engineering. Since 1989, we have focused exclusively on the development and testing of Windows-based solutions, resulting in the delivery of truly innovative products for some of the most demanding and successful companies in the world. MicroCrafts is constantly striving to create new and more effective processes to help deliver higher quality software products and solutions to our clients. MicroCrafts is a Metamor Software Solutions company, a division of Metamor Worldwide (Nasdaq: MMWW), a leading information technology consulting solutions organization.

Experts in Software and Quality Assurance

To be successful in the software industry, your business must produce high quality, reliable, stable and bug free software. This means, for every software project, premium software design, development and quality assurance testing is vital.

Thousands of projects have been entrusted to MicroCrafts for our proven software development, quality assurance methodology, and testing expertise. A proven track record, backed by over a decade of successful business relationships with leading high tech multinationals, assures your projects' success.

Trust our experience to work for you.

(425) 250-0000
info@microcrafts.com
www.microcrafts.com

MicroCrafts

A Metamor Software Solutions Company

Microsoft®

View ad

Microsoft® Project 98 is a powerful application you can use to efficiently plan, manage, and communicate project information. If you need software to handle dynamic scheduling, manage multiple projects and resources, and support better communication, Microsoft Project 98 is the tool for you. It is designed for anyone who oversees a team, plans a budget, juggles schedules, or has deadlines to meet. In short, anyone who manages a project as part of their job in today's busy working world.

No matter how large your project, or how many people on your team, Microsoft Project 98 helps you to transform your organization into a more agile, responsive, customer-oriented company by focusing on project needs. Because Microsoft Project 98 is part of the Microsoft Office family, it's as easy to use as your other favorite Office Programs.

Benefits

Control Your Projects with Confidence - With the flexible new scheduling engine in Microsoft Project 98, you can plan and track your projects more effectively and identify and respond to conflicts before they happen.

Communicate your Plans with Ease - Microsoft Project 98 has rich presentation features including the ability to publish information to the intranet and internet, plus workgroup features that let you communicate with team members via e-mail or the web.

Works the way that you do - Full compatibility with Microsoft Office applications, as well as other databases and systems, means that users learn Microsoft Project 98 faster and can share information easier with others in the company.

Your old planning tool won't be completely obsolete.

Developing software is a complicated task. In most cases, too complicated to update and track on a whiteboard. Microsoft® Project 98 helps you make your deadlines by giving you the power to plan each step, make changes, and keep everyone informed along the way.

You can update an entire plan with just a couple of clicks. You can integrate Microsoft Project 98 data with applications you already use. And everyone on your team can stay up-to-date via e-mail or the Web. Try doing any of these tasks with your whiteboard. Visit our Web site for a free* trial at www.microsoft.com/project/tryit98 or call 1-888-877-9092.

Microsoft Where do you want to go today?®

*Connect time or shipping and handling charges may apply.

© 1999 Microsoft Corporation. All rights reserved. Microsoft and *Where do you want to go today?* are either registered trademarks or trademarks of Microsoft Corporation in the U.S. and/or other countries.

Performance Research, S.r.l.,

View ad

Performance Research, S.r.l., specializes in analysis and prediction of the performance of complex integrated software/hardware systems. The company provides software and services based on its technology.

PREDICTA is a comprehensive software tool that helps designers at solving development issues of complex computing network systems by modeling, simulating and predicting system interactions and performances.

PREDICTA features a performance-oriented modeling environment providing engineering value for even complex, multi-layered systems. It answers performance-related feasibility, configuration and optimization questions quickly and can give objective guidance to the systems development team and the system deployment team.

PREDICTA highlights many performance defects that cannot be identified either by static analysis or by black-box testing techniques. Introducing a performance-oriented modeling and prediction activity into the system design process allows the design-team to perform what-if analyses on the proposed designs from the start. With this information, the whole development process can be guided by the performance requirements.

Using ***PREDICTA***, system engineers can:

- Optimize a deployed system by performing the tuning or re-configuration actions that resulted to be the most effective in the model-based analysis.
- Optimize a system design with respect to the specific service requirements of a specific customer before deploying the real system.
- Discover bottlenecks and evaluate (what-if) the consequences of making a particular choice of system components and/or system architectures.
- Have evidence of the sustainable service request increase curve for a given system implementation/configuration.
- In general, benefit from knowledge of the projected performance for the design-in-place in contrast to the service level agreements made on performance requirements.

THE STATE OF THE ART IN PERFORMANCE PREDICTION

PREDICTA

PREDICTA is a comprehensive software tool that helps designers at solving development issues of complex computing network systems by modeling, simulating and predicting system interactions and performances.

www.pr-usa.com

PERFORMANCE PR RESEARCH

PR-USA Sales:
Toll Free: 888 607 6267
www.pr-usa.com
info@pr-usa.com

Performance Research, S.r.l.
Phone: +39 0125 45 698
Fax: +39 0125 63 30 76
www.predicta.it
info@predicta.it

Q-Labs, Inc.

View ad

Q-Labs is dedicated to providing software engineering solutions to the industry, enhancing software development capabilities. Q-Labs' expertise is as a change facilitator, which includes providing 'state-of-the-art solutions' as well as supporting dissemination of 'industry standard best practices.' Our approach is to keep our customers steps ahead in software engineering. Q-Labs products and services include:

- Software Management
  (Business Analysis, Software Process Improvement, ROI Calculations, ...)
- Software Process Improvement
  (SW-CMM-based Software Process Improvement, SE-CMM, Measurements, ...)
- Technology Transfer and Deployment services (Inspections, UML, SDL, ...)
- Cleanroom Software Engineering
  (Statistical Usage Testing, Sequence Based Specifications, Incremental Development)
- Humanics (Teamwork, P-CMM)
- Software Acquisition services (SA-CMM, Supplier Evaluations, External QA, ...)
- Test Support (Daily Builds, Automated test)
- Contract Development Services
  (software specification, development and test for customer projects)

Our goal is customer success. For example, Q-Labs in its role as the primary provider of SPI support for the Ericsson System Software Initiative, helped Ericsson save 60 million USD in 1997 due to improved software quality. Q-Labs works with a number of customers around the world, including Alcatel, Bosch, CTI, Ericsson, IBM, Siemens, and the U.S. Army. Q-Labs was founded in 1989 and has over 60 employees at offices located in Sweden, Germany, US, Ireland, and Norway. We have a very strong international network and a strong capability to support global customers. Q-Labs is a joint venture between DNV and Ericsson.
For more details, visit us at **www.q-labs.com** or send write us at **info@q-labs.com**.

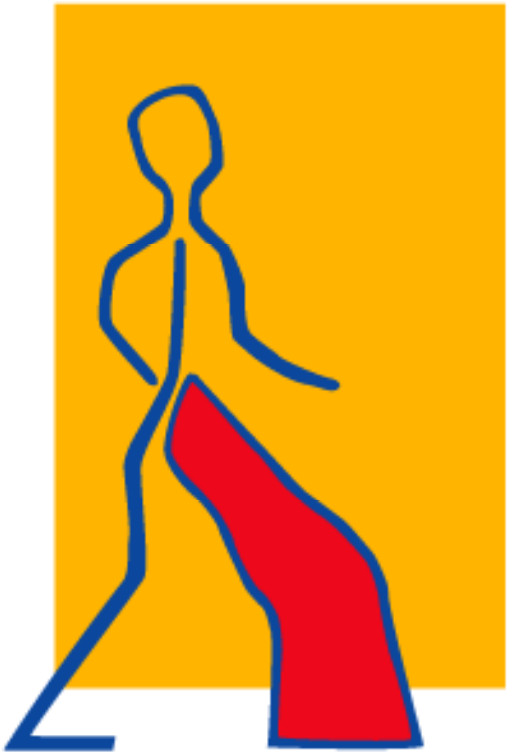
Q-Labs

Your Partner in

Software Engineering

Total savings in 1997 due to improved quality is at least 60 million USD

– Results from the Ericsson System Software Initiative with Q-Labs as the primary provider of the SPI support



We are dedicated to providing software engineering solutions to the industry, enhancing software development capabilities.

- Our expertise includes providing state-of-the-art solutions as well as supporting dissemination of industry standard best practices.
- Services include
  - Software Management (Business Analysis, Software Process Improvement ROI Calculations, ...)
  - Software Process Improvement (CMM-based Software Process Improvement, Measurements, ...)
  - Technology transfer and Deployment services (Inspections, UML, SDL, ...)
  - Cleanroom Software Engineering (Statistical Usage Testing, Incremental Development, ...)
  - Humanics (Teamwork, P-CMM, ...)
  - Software Acquisition services (SA-CMM, Supplier Evaluations, External QA, ...)

Q-Labs was founded in 1989 and today has over 70 employees at offices located in Sweden, Germany, US, Ireland and Norway. We have a very strong international network and a strong capability to support global customers. Q-Labs is a joint venture between DNV and Ericsson and serves several software-intensive organizations with software engineering solutions. For more details and employment opportunities, visit us at **www.q-labs.com** or send your questions to **info@q-labs.com**.

Quantitative Software Management

Quantitative Software Management (QSM) is a leading software project management organization specializing in software metrics products and services. We offer our clients **Software Lifecycle Management (SLIM)** processes and tools for software cost estimating, reliability modeling, schedule estimating, planning, tracking, and benchmarking. We have been helping our clients develop quality software on time and within budget for over 20 years.

Products:

- **SLIM-Estimate** is a project estimation and planning tool which implements a structured method for determining the amount of function required to satisfy a given set of software requirements and then allows managers to identify the best strategy for building a corresponding product, shortening cycle time, reducing cost, improving quality, and minimizing risk.

- **SLIM-Control** is a tracking and oversight tool which implements Statistical Process Control techniques for assessing the status (plan versus actual with forecast to completion) of built-in and user-defined measures and metrics.

- **SLIM-Metrics** is a metrics repository tool which allows the user to assess competitive position, identify development bottlenecks, quantify the benefits of improvement and support planning for future projects.

View ad

Rational Software Corporation

Address: Corporate Headquarters
18880 Homestead Road
Cupertino, CA 95014
Phone: 408-863-9900
Fax: 408-863-4120
Email: info@rational.com <**mailto:info@rational.com**>
WWW: www.rational.com <**http://www.rational.com**>

Rational Software (NASDAQ: RATL), creator of the Unified Modeling Language (UML), is the leading provider of a solution that unifies proven software development principles, tools, and services to improve the productivity of project teams and individuals. Rational's products span the critical activities of requirements management, visual modeling, testing, and configuration and change management.

Rational's mission is to ensure the success of customers who depend on their ability to develop the software upon which their businesses depend. Rational enables its customers to achieve business objectives by turning software into a source of competitive advantage, decreasing time-to-market, reducing the risk of failure, and improving software quality.

Rational's comprehensive solution unifies proven principles of software development, an integrated family of market-leading tools, and technical consulting services into a set of software Best Practices applicable through the development lifecycle by all members of a development team. Rational's products can be purchased and used individually or integrated with other Rational products, leveraging the power of each individual product.

Rational has more than 1900 employees worldwide, with corporate headquarters in Cupertino, California. Major development centers are located in California, Massachusetts, Oregon, Colorado, North Carolina, Washington, Pennsylvania, Sweden, and India. For more information, visit Rational's Website at **www.rational.com**.

We offer competitive salaries and full benefits.

now lives at the office searching for "bugs". Excellent salary, all meals included, great benefits (including two children).
Route 23, Box 89

immediately, please send your resume.

CRUMB HOARDER

Wanted: Snacker with ability to remain at desk for light years at a time. Preferably someone with a penchant for flaky, crunchy food. Hunger for donuts, potato chips and fast-food a plus. Must be able to eat all meals at desk AND be willing to stockpile crumbs in uniform fashion around chair. Those who fear rats, mice, bosses and other office rodents need not apply. Unhealthy opportunity employer. www.Dingbat.com

INSOMNIAC

WANTED: Loner to work through the night comprehensively testing every new application bell and whistle, followed by writing detailed reports. Must be able to repeat tests on every build without delay. Seven days per week, no vacations, holidays.

SOFTWARE APPLICATION ENGINEER

Responsibilities include designing, coding, and testing new and existing software for multi-service access platforms. Related experience in embedded software, communications systems, device drivers, or Ethernet a plus. Also qualified candidates must have Master's degree in Electronic Engineering, Applied Math or a related area. Some course work will be required. Must be a citizen or legal resident of the United States. Apply by resume only.

PROGRAMMER / ANALYST

This position is for an experienced, ambitious individual who can work on a broad range of systems development projects and commercial applications. You must have experience working with Internet/Web development, client/server application systems, and mainframe environments. Knowledge of HTML, Visual Basic, Java, C/C++, and UNIX desired. Also must be team-oriented and have excellent communication skills. For more information, contact us 555-1435

EDITOR/SPELL-CHECKER/BUG-CHECKER

The ideal candidate will have proven experience writing brochures, follow up letters and code. Also must be an excellent proofreader and have a strong command of the English language ... testing

MIRACLE WORKER

Must be able to write difficult programs and be able to detect leaks and errors in the application quickly and easily. Also must be able to make the blind see, the deaf hear and the mute speak. For more information, please contact Joe Black.

WHITE HOUSE DAMAGE CONT... SPECIALIST NEE...

Desperately seeking somec... the leaks! Everyday, more... stories are being "leaked" t... Its about time we put an en... need someone to dedicate ... of every day to finding and ... the leaks. Qualified candidates ... willing to work extremely hard a... able to chase wild geese to no end... this job intrigues you, then send us your resume. But please, dont tell the press. www.anyssite.com.

...TWARE PROJECT LEADER

...ecurities Corporation is ... for an experienced ... Project Leader. The ...didate will have experi... project management, as ... software development ...sses. Knowledge of net...k protocols and Internet ...applicatio... So if you woul... for a dynam... any and yo... of software ... experience with ... year of experience leading software engineers and software projects, then send us your resume and a cover letter. We also require strong written and oral commu-

SOFTWARE TESTERS

This is not another ordinary testing job. We're seeking an extremely experienced software tester to work with a host of talented engineers that have bee... testing software their entire career. Knowledge of testing

SOFTWARE APPLICATION EN...ER

Responsibilities include designing, coding, and testing new and existing software for multi-service access platforms. Related experience in embedded software, communications sys-

RATIONAL SUITE TESTSTUDIO WILL TAKE THE JOB.

Rational Suite TestStudio's comprehensive suite of testing tools includes breakthrough technology that automatically exercises your application, measures code coverage, and creates reusable test scripts for regression testing. It liberates your team from mundane testing chores, making them more productive and improving overall quality. No wonder Rational tools are #1 worldwide. Get some sleep by visiting our website to order a demo CD-ROM.

RATIONAL SOFTWARE

See Us At Booth #309

UNIFYING SOFTWARE TEAMS

www.rational.com • 1-800-728-1212

The word "Rational" and Rational's products are registered trademarks of Rational Software Corporation. All other names and trademarks are the property of their respective owners.

View ad

Solutions for Automated Software Inspection

Reasoning is a leading provider of solutions and services that lower the cost and accelerate the speed of delivering reliable software and Web-enabled applications. The company's solutions are based on the *Reasoning5 Code-base Management System* (CBMS)—an advanced software analysis and transformation engine. Reasoning's solutions for mainframe and client/server environments feature extraordinary automation and accuracy providing benefits of high productivity, low cost fast turnaround and high quality.

Inspector II™ for Automated Software Inspection
Inspector II is a software quality toolset and process that helps organizations produce high-quality software and lowers the cost and time of testing. Inspector II automatically inspects mission-critical applications and pinpoints and predicts software defects that cause application failures—it also measures the fragility of software.

Inspector 2000™ for Year 2000 IV&V
Inspector 2000 is acknowledged by Gartner Group and META Group as a superior solution for Independent Verification and Validation (IV&V) of Y2K software repairs. Inspector 2000 audits of the thousands of remediated applications show that virtually all repaired code bases still contain hundreds to thousands of Y2K defects. Inspector 2000 can audit internally or externally remediated code, audit commercial software vendor and supply chain partner applications, as well as help with Y2K contingency planning.

enVision for Internet Transformation
enVision solutions combine CBMS technology and innovative services that accelerate the transformation of client/server applications to the Oracle Internet Platform and the reengineering of IBM CICS applications to support e-business. enVision is a superior solution for application migrations and Web-enablement because it is more accurate than code translations and more time and cost-efficient than code re-writes.

Enterprise Application Auditing Consulting Services
Reasoning offers consulting services to analyze and assess the quality, maintainability and Internet adaptability of existing application software. These unique audits produce detailed information and recommendations for improving the reliability of business-critical applications, lowering the costs of software development and maintenance and streamlining ERP and EAI initiatives. Reasoning's e-business feasibility audits are designed to help organizations develop rapid implementation plans to support strategic e-business initiatives.

For more information, visit the Reasoning website at http://www .reasoning.com

Reasoning, Inc.
700 East El Camino Real
Mountain View, California 94040
+1 (650) 429-0350 tele
+1 (650) 429-0222 fax
www.reasoning.com

Before taking another step, contact us to ensure your software is fault-free.

Automated inspections of mission-critical applications performed by Reasoning, Inc., have revealed serious defects and fatal errors. Whether you are concerned with ensuring the reliability of key applications or validating Y2K software repairs, Reasoning's highly automated services and tools can inspect and blueprint your software fast and with unparalleled accuracy--reducing the risk of software failures and the cost of testing and maintenance. Our inspections find defects in software prior to testing, at 1/10th the cost; inspect applications that you don't have time to adequately test; and assure due diligence in the event of liability claims. Let Reasoning be your software safety net.

Call Reasoning today at **1-888 FIX-4-Y2K** (349-4925) or visit our website at **www.reasoning.com.**

REASONING

© 1999 Reasoning, Inc. Reasoning and the Reasoning logo are trademarks of Reasoning, Inc.

Soffront Software Inc.

View ad

830 Hillview Ct. Ste 140
Milpitas, CA 95035 USA
http://www.soffront.com

Soffront Software Inc. is a leading supplier of enterprise-wide solutions designed to track all aspects of development, sales and support. Founded in 1992, the company designs, manufactures, and markets TRACK family of products for Defect Tracking, Help Desk, Sales Force Automation, and Knowledge Management. These products are used by organizations worldwide to solve their enterprise wide tracking and analysis needs. More than 40,000 licenses have been sold to organizations worldwide.

TRACK 5.1™ is an easy-to-use client server based system that provides tools for designing, creating and customizing databases based on many popular SQL backend databases. The database is then used to keep accurate, detailed records of projects, bug reports, code changes, test cycles, sales leads, sales forecast, account details, support calls, product releases, documentation, system configurations and more. TRACK version 1.0 was first introduced in January 1993.

TRACKWeb™, the industry's first customizable web-based tracking system, was introduced by Soffront in December of 1996. TRACKWeb lets developers, vendors, customers, and users easily access, submit and report necessary information through their web browsers, such as Netscape Navigator™ or Microsoft Explorer™. TRACKWeb supports applications such as Help Desk, Sales Force Automation, and Defect Tracking. TRACKWeb enhances communication between vendors and their customers by providing a way to get answers to questions in real-time.

TRACKRules™, a flexible rule definition system, extends the power of TRACK and TRACKWeb. Introduced in October of 1997, TRACKRules allows a system's administrator to define complex conditions, which activate mail notifications and field updates, thereby ensuring proper maintenance and execution of business rules, process flow, and communication.

TRACK KB™, a knowledge base tool which "learns" a company's product and service information from expert support personnel, thus creating an available pool of wisdom for users who need to quickly find answers to their questions. TRACK Kb was introduced in August of 1998. TRACK Kb, used in conjunction with TRACK™ and TRACKWeb™, utilizes rules, questions and solutions to automate locating answers to users' questions.

Soffront sells TRACK family of products for Help Desk, Defect Tracking, and Sales Force Automation through a direct sales force and distributors in the United States and Canada and through an international network of value-added resellers. Soffront is funded by private investors.

SOFFRONT®
Enterprise Solutions
WWW.SOFFRONT.COM
DEFECT TRACKING
Lowers development costs and increases productivity
Enhances communication and process flow among groups
Notifies specified individuals automatically
Displays reports to measure performance, show status...
Integrates with popular Version Control Systems
Supports SQL databases and multi-level security model
Controls work assignments and estimates future releases
Supports routing, work flow and business rules
HELP DESK
Increases support staff's productivity
Fully customizable environment to suit your needs
Supports local and globally distributed installations
Supports QBE, SQL, keyword & full-text search capabilities
Customize user interface, security and workflow
Initiates requests based on customer-defined requirements
Supports escalations based on customizable criteria
SALES AUTOMATION
Manages contacts, accounts, and pipeline of opportunities
Improves the sales process and customer service
Reduces Selling and Administrative Costs
Associates related accounts
Maintains history of accounts, contacts and correspondence
Customizable security and workflow by account or individual
Automatic territory assignment and mass realignment
Trigger events based on action items
Easy access to actions and relationship hi
Dynamic security based on role, membership, contact type
Supports user-defined work flow and escalation rules
KNOWLEDGE MANAGEMENT
100% SATISFACTION GUARANTEED
"The abilty to get users up and running with TRACK in a short time is invaluable."
Greg Bryant- Principal Software Engineer, NEC Technologies, Inc.
"It has also improved our customer because we have a more accurate way of tracking customer concerns."
Ed Mischkot- Software Engineering Mgr., Atlantis Aerospace Corp.
"TRACK automates interactions with related tools."
Peter Coffee- PC Week, March 4, 1996
"Wins hands down in power, flexibility and ease-of-use."
Data Management Review, January 1996
"You can create custom screens and views to tailor Track to your environment."
Michael Deignan- Windows NT, October 1998
Provides better and quicker customer service
Readily retrieves answers to commonly asked questions
Accumulates "wisdom" for future usage
Identifies solution for users through structured guidance
Converts existing support database to knowledge elements
Soffront Software, Inc. Phone:1-800-763-3766
Outside USA: 408-263-2703 Fax: 408-263-7452
E-Mail: info@soffront.com

Software Development Technologies (SDT)

View ad

SDT's mission is to assist organizations throughout the world with improving their software quality through training and consulting. SOFTWARE DEVELOPMENT TECHNOLOGIES is the only company to offer a fully-integrated solution to Technical Reviews and Inspections, from methodology to automated implementation.

TECHNOLOGY ASSESSMENTS. SDT's senior consultants identify and prioritize the areas in greatest need of improvement and recommend a step-by-step plan to most effectively address the goals and actions for optimal improvement. SDT uses a pragmatic, hands-on approach.

TRAINING SERVICES. SDT's expert methodologists are constantly developing and evolving our comprehensive public and on-site training courses in the areas of Software Testing, Testing Tools and Technical Reviews and Inspections. Our train-the-trainer program can certify your instructors to teach courses that have been customized and licensed for use within your company.

CONSULTING, QUALITY TEAM COACHING/MENTORING. SDT consultants are experts in software technologies that apply universally to all types of software. They meet the same high standards as our course instructors and include prominent technology leaders, authors, conference speakers and leaders of national software standards efforts.

For companies serious about significantly improving their software quality practices, SDT offers coaching for senior Quality, Development and Test managers and their teams.

PRODUCTS**.** SDT provides specialized software testing products that complement our service offerings. **ReviewPro™**, SDT's new web-based, automation tool for Technical Reviews and Inspections, prevents the migration of defects to later phases of software development. **ReviewPro** is integrated with your existing IT infrastructure and easily accommodates your existing Inspection process.

Headquarters: San Jose, California. www.sdtcorp.com. 408.297.1911

Software Development Technologies

Providing practical, proven and effective software quality products and services

Training Services

- Public and onsite courses in:
  - Software Testing
  - Testing Tools
  - Technical Review and Inspection
- Onsite courses customized for your test environment
- Course licensing available
- Train-the-trainer program offered

Technology Assessments

- Evaluation of your company's test and development methods, tools, practices, organizational structure, and environment
- Identification of areas for improvement
- Recommendation of a step-by-step action plan
- Scope can include verification and/or validation as well as entire life cycle

Consulting Services

- Expertise in all areas of software quality improvement:
  - Test practices
  - Automation
  - Inspection
  - TestFrame design and automation methodology
- Senior consultants take a practical, hands-on approach

ReviewPro™

- Brings automation to the Technical Review and Inspection process
- Web-based collaboration application
- Open architecture
- Project management for geographically dispersed teams
- Data collection and metrics reporting capabilities

ReviewPro™

Bringing Automation to Software Review and Inspection

ReviewPro is an enterprise software application which facilitates and simplifies the most powerful defect detection and removal method—the Review and Inspection process. A web-based collaboration application, ReviewPro allows you to harness all the information created and decisions made during your product development cycle.

- Simplifies and facilitates communication
- Eliminates needless paperwork and meetings
- Enables you to measure and manage analysis results
- Serves as a central, accessible repository for collection and reporting of Inspection metrics
- Leverages industry acknowledged best practices
- Increases willingness to participate in the review process
- Compatible with your existing IT infrastructure
- Easily customized to accommodate any industry standard or custom in-house Inspection process
- Gives management visibility to the status of software deliverables

"ReviewPro is the most interesting piece of software for giving support to the Inspections process I have seen. I am recommending that my clients evaluate it." Tom Gilb, www.Result-Planning.com

"The biggest complaint we've had with our current inspections process was getting data into the system; this problem will be solved with ReviewPro. Its collaborative feature allows defects to be documented and visible before our meeting so productivity is greatly improved." Edward Weller, Fellow, Bull Information System

Software Development Technologies (408) 297-1911 · *sdt@sdtcorp.com* · *www.sdtcorp.com*

Software Emancipation Technology, Inc.

View ad

15 Third Avenue, Burlington, MA 01803
Tel: 781-359-3300 or 1-888-9-DISCOVER
Fax: 781-359-3399
Web: www.setech.com
E-mail: info@setech.com

Software Emancipation develops, markets, and supports DISCOVER®, a flexible and scalable development information system designed to help enterprises to evolve their mission-critical software development processes, and to more effectively manage their software systems as strategic assets. By using DISCOVER, enterprises achieve measurable improvements in software quality and development processes, reduction in development and maintenance costs, and predictability in time to market.

DISCOVER analyzes source code and creates a database of information (the Information Model) that captures the interrelationships between all entities in the code base, resulting in a detailed view and high-level architectural perspective of the entire application. This Information Model provides critical information for both management and the development team.

DISCOVER captures detailed knowledge about the structure and operation of an organization's software - that might otherwise reside only in the minds of individual programmers - and brings precision, manageability, and predictability to the software development process. DISCOVER enables software professionals to more thoroughly understand their large software systems, to more efficiently and accurately effect changes to a large body of source code, and to more easily reengineer or reorganize a complex software system.

With DISCOVER 7.0 Software Emancipation introduces premier solutions for improving software quality through quality process-control and management. It includes a mechanism for quantitative unbiased measurement of software and the means to analyze, monitor, control, and improve software quality early in the development process. DISCOVER 7.0 includes:
SubmissionCheck for early detection of violations in coding procedures before submission.
QA Analyst enables to set up, monitor, and enforce quality compliance standards.
QA Cockpit provides a quantitative reporting tool:
QAR (Quality Assessment Report) for monitoring and supervising software quality as it is being developed.
Quality Assessment and Diagnostic Service

DISCOVER supports C and C++ source code, Oracle embedded SQL and PL/SQL, and Java, and runs on SunOS, Solaris, HP-UX, SGI IRIX, and Windows NT.

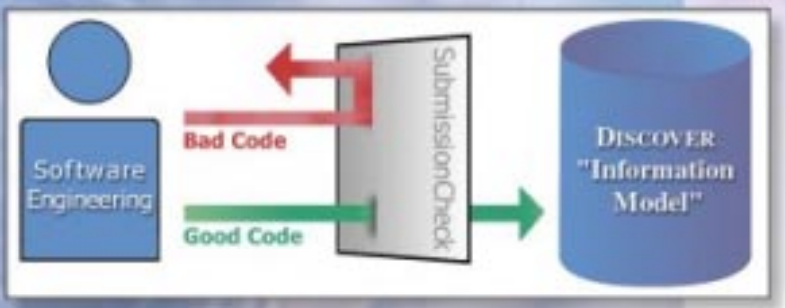
Software Emancipation

Attacks Industry Software Quality Problem

DISCOVER® is a software development information system consisting of a comprehensive, integrated set of applications and tools. DISCOVER analyzes source code and creates a database (the Information Model™) that captures the interrelationships between all entities in the code base.

DISCOVER offers the means to analyze, enforce, control, monitor, and improve software quality early in the development cycle.

DISCOVER's new quality process-control and management solutions empower management with unprecedented control, giving them precise, predictable, and measurable information about the quality of their software as it is being developed, enabling them to make business decisions about the development project on the basis of objective data, rather than intuitive assessment.



DISCOVER®'s SubmissionCheck checks compliance with predefined quality standards **before** submission.

"Improve your Software Quality through Quality Process-Control and Management"

Presented by Bruce Boes

Please Join us for

QW '99 Technical Session
Thursday, May 27 at 12:00 noon

QW '99 Demo Session
Thursday, May 27 at 1:00 pm

DISCOVER®
Development Information System

SOFTWARE EMANCIPATION TECHNOLOGY

Phone: (781)359-3300 **Toll Free:** (888)9-DISCOVER
Headquarters: 15 Third Ave. Burlington, MA 01803 USA
Fax: (781)359-3399 **Email:** info@setech.com
Website url: http://www.setech.com

Software Quality Engineering

View ad

330 Corporate Way
Suite 300
Orange Park, FL 32073
(904) 278-0707
(904) 278-4380 (fax)
sqeinfo@sqe.com
www.sqe.com

Company Description for CD-ROM Proceedings:

Founded in 1986, Software Quality Engineering (SQE) assists software professionals and organizations throughout the world with improving their software testing and quality engineering practices. The company's hands-on experience and training expertise help companies – large and small – to improve testing practices, gain measurable control over software projects, and ultimately deliver better software.

SQE can assist your software organization through the following services:

Professional Services – SQE understands that a one-size-fits-all approach to software improvement is rarely effective. Our professional support process begins with assessment and analysis of the unique needs of your organization, followed by hands-on, targeted action. From on-site training to in-depth project implementation support, SQE has the technical experience and resources to address the special needs of your organization.

Training Seminars – SQE delivers specialized training seminars on systematic testing, test automation, Year 2000 testing, software metrics, requirements, software management, and more. These high-leverage courses are presented publicly throughout the year and provide verifiable results in improving productivity and software quality.

International Conferences – SQE organizes international conferences, including ASM Software Measurement Conference, SM Software Management Conference, and the STAR Software Testing Conferences**.**

Publications & Research – In an effort to make good software testing and development information more readily available, SQE *publishes Software Testing & Quality Engineering* magazine and operates the online *SingleSource Guide to the Best Software Quality Books*.

Software
Quality
Engineering

904-278-0707
800-423-8378

sqeinfo@sqe.com
www.sqe.com

SQE Provides Information, Training, and Support to Help You Achieve Greater Software Quality

Field-Proven Software Testing and Development Seminars

SQE runs over fifteen specialized software testing, development, and management seminars. Learn systematic, field-proven techniques for software testing, test automation, client/server and object-oriented testing, requirements modeling, software project management, and more.

International Conferences with Real-World Experiences

SQE organizes the STAR testing conferences, ASM metrics conference, and SM software management conference. These top events present the real-world testing, metrics, and management strategies of leading software organizations worldwide.

Professional Support for Your Software Efforts

SQE understands real-world software challenges and uses high-leverage consulting to support your unique software needs. Increase software quality & productivity with SQE on-site team training, hands-on assessments and benchmarking, and project implementation support.

PLUS!

Now SQE employs its knowledge and expertise to publish ***Software Testing & Quality Engineering*** magazine - your information source for developing and delivering better software.

Software Research, Inc.

View ad

901 Minnesota Street
San Francisco, CA 94107
Phone: (415) 957-1441
Fax: (415) 957-0730
Email: info@soft.com
WWW: http://www.soft.com

Software Research, Inc. created the field of Automated Software Testing and, since 1987, has been providing the QA community with testing tools such as CAPBAK, the capture/playback system, SMARTS, the corresponding test manager, and the first commercial test coverage analyzer, TCAT.

Since 1995, TestWorks has been the original and only suite of Integrated Testing Tools that includes both test regression and test coverage support for Embedded, GUI, Client/Server and Web applications, on UNIX and Windows platforms. The tools can be used separately or as a family integrated with the other TestWorks tools, enhancing the thoroughness of the testing process.

More recently, TestWorks has expanded with TCAT for Java and CAPBAK/Web. TCAT for Java is the first coverage analyzer for Java applets and Remote Testing Technology (RTT) with local, Email and over-the-Web collection of snapshot user interactions and refined test coverage data from Java applets. CAPBAK/Web is a "test enabled Web browser" that incorporates all of the required features for a CAPture/playBAcK system for testing websites.

Software Research, Inc, the company that pioneered end-to-end testing solutions, looks forward to helping improve the quality of your applications and achieving 100% return on your IT tools and technology investment.

To learn more about TestWorks from Software Research, Inc., visit our website at: **http://www.soft.com**, or send email to: **info@soft.com**.

TOTAL QUALITY MANAGEMENT

Test Tools	TestWorks	Your Process
Capture/Playback	✔	?
Test Management	✔	?
Test Data Generation	✔	?
Static Analysis	✔	?
Metrics	✔	?
Path & Branch Coverage	✔	?
Call-Pair Coverage	✔	?
Web Applications	✔	?

All the tools you need for rock-solid code

Software Research has created a suite of fully integrated and automated testing tools to help you ensure the quality of your software products. TestWorks offers an end-to-end solution that covers all aspects of your process life cycle: test design and development, test data generation, test execution and evaluation, reporting and test management, code comprehension, coverage analysis, metrics and maintenance. Perfect Tools for the Quality Architect.

TestWorks improves the quality of your products, whether in an embedded or distributed client/server system, in GUI desktop or web applications. Its open architecture empowers your work across the major UNIX and all windows platforms and OSs, in C, C++, Ada, Fortran and Java.

You need to deliver high-quality software on time, under budget?

We have TestWorks! Call the company that created the field .

SR Software Research

1 (800) 942-SOFT
email:info@soft.com
http://www.soft.com

Software SETT Corporation

View ad

Software SETT Corporation is a SQA consulting group that specializes in systems evaluation, software testing and tester training services. We have been a Quality Assurance organization for over a decade – creating test plans, recommending testing strategies, managing and executing tests, evaluating defect tracking and automation tools. Our clients include start-ups, the Fortune 50, and many prestigious Silicon Valley companies.

Repeat business and referrals have given Software SETT Corporation solid expertise in the following markets/applications:

- PeopleSoft-based client/server applications for the corporate IT market
- Internet/Intranet applications
- Data warehousing software
- Medical device products
- Embedded systems
- Communications/networking hardware and software

Software SETT Corporation also offers a training curriculum tailored to fit the needs, budgets and timeframes of our clients.

We believe that one of our most important assets is our people. We are dedicated to building the best QA team in the industry. This goal ensures that we have a highly cohesive team of SQA professionals to meet the needs of our clients.

Our commitment to total quality, coupled with our ability to respond quickly to our clients' needs, has resulted in consistently high levels of customer satisfaction. At Software SETT Corporation, customer satisfaction is our number one priority – every employee is dedicated to providing quality service. To maintain this important aspect of our business, customers are invited to confer with us regularly.

You can count on Software SETT Corporation to be your reliable SQA provider.

Software SETT Corporation

YOUR SQA SERVICE PROVIDER

Software SETT Corporation is an SQA consulting group that specializes in systems evaluations, software testing and tester training services. For over a decade, we have acted as the Quality Assurance department - creating test plans, recommending testing strategies, managing and executing tests, evaluation tools for a number of prestigious Silicon Valley clients, ranging from start-ups to the Fortune 50.

Word-of-mouth referrals and repeat business has given Software SETT a vast array of expertise in many market segments. Our knowledge base encompasses, but is not limited to:

- PeopleSoft-based client/server applications for the corporate IT market.
- Internet/Intranet applications.
- Data warehousing software.
- Financial investment software.
- Communications/networking hardware and software solutions.
- Embedded systems software.
- Medical device products.

Software SETT also offers a comprehensive training curriculum tailored to fit the needs, budgets, and timeframes of of your test team.

Software SETT Corporation

233 Oak Meadow Drive
Los Gatos, California 95032
Phone: 408-395-9376
Fax: 408-354-6477
http://www.softsett.com

Sun Microsystems, Inc.

www.sun.com/jobs
www.sunboston.com
jobs@East.Sun.COM

A $10 billion company with offices in 150 countries, Sun Microsystems provides end-to-end solutions for doing business in the network age.

We provide high-speed microprocessors, scalable systems, robust software, network storage, mission-critical support, and comprehensive professional services. Not to mention innovative, platform-independent Java and Jini technologies. In short, all the key components that drive the Internet as well as corporate intranets and business-to-business extranets.

Which explains why we're the dot in .com.

Headquartered in Palo Alto, CA, Sun recently opening a 170 acre East Coast Headquarters in Burlington, MA. As well as a new facility in Broomfield, CO. For information on job opportunities at Sun, please visit us at www.sun.com/jobs or www.sunboston.com or send email to jobs@east.sun.com

Powerlab™

View ad

Powerlab is designed as quality assurance testing hardware that dramatically reduces floor space, cooling and power consumption. It is ideal anywhere where multiple computers or servers are needed.
The Powerlab provides 20-24 computers in a cabinet that uses only 10 square feet of floor space, and offers electronic switching.

... for Software Developers
The Powerlab model for software testing runs on a standard 20 amp circuit, uses 10 square feet of floor space and produces no more heat than a typical bread rack populated with 15 PCs. It allows the operators to have all the components at their fingertips - in drawers that each hold 4 computers.

... for Drive Manufacturers
The Powerlab model for drive testing increases testing capacity by 30% over conventional bread rack testing configurations. Only 12.25 square feet of floor space is required for 21 standard PCs. Powerlab provides a convenient test shelf with connections for power and all drive types, including USB, PCMCIA, IDE, SCSI, floppy, tape, cd-rom, dvd, and removable media drives.

... for Peripheral Manufacturers
Powerlab provides a space saving, energy efficient method for testing almost every computer peripheral on the market. It is the ideal solution for testing modems, network cards, video cards, sound cards, scsi controller cards, and much more.

... for Testing Laboratories
There are many reasons for testing labs to choose Powerlab over conventional bread rack systems. Powerlab provides up to twice the testing power per square foot as compared to it's bread rack counterpart, works on conventional 20 amp circuits, produces less heat.

Sunpower
COMPUTING

In 1986 Sunpower built it's first computers in order to standardize the quality and consistency of the hardware used to run its accounting systems. By 1993 hardware and network contracts replaced software as Sunpower's primary products.
In 1997 Sunpower started the design and engineering of a consolidated test rack, now known as the Powerlab™. These racks incorporate up to 20 computers in a compact design and are controlled by a single monitor and keyboard, using either an electronic or software switch to access the individual PC's.
Sunpower's markets are now firmly planted in the powerlab.

Technology Search International

1737 North First Street, Suite 600
San Jose, CA 95112
408-437-9500
www.tsearch.com

Technology Search offers over 15 years of experience in all aspects of technical search. Our national clientele range from some of the most prestigious computer systems / software companies to the hottest venture funded start-ups in the Silicon Valley. Our detailed understanding of the industry and of our client companies is unparalleled, as is our reputation and the results that ensue. Industry leaders provide TSI direct onsite access to the hiring managers that should be reviewing your resume. The close relationship we share with these clients has earned us an unparalleled reputation as the logical place to go for top software talent. Often times the best opportunity is the one that finds you. Our purpose is to open a dialog and keep you informed on what is happening outside of your present situation. Supply your background information in advance and let us learn about your sphere, focus and goals. Then we can act as your agent and bring suitable and attractive opportunities to your attention as they arise.

Montage Solutions

Montage Solutions was created to offer clients a new model for addressing their temporary technical staffing needs, and consultants an open and more inclusive way to work with agencies. The cornerstone to our approach is an open and fixed pricing model designed to remove a common stumbling block to a smooth process, and to allow each of our clients to feel comfortable that our rates are consistent and fair. We've taken this dramatic departure from the standard approach to our business for one reason - to offer you a better service. Contact us today to find out more about our services in Software Development, Hardware Development and Information Technology.

Teradyne SST

View ad

Teradyne Software and Systems Test (SST) is the leading supplier of solutions for automating the system definition and test generation process for software systems. Major telecom suppliers currently use SST's products across a wide range of telecom systems to achieve 80-90% improvement in test development time and productivity.

Unlike manual script generation or capture/replay tools which require test engineers to spend days or weeks interactively generating software tests, SST products generate tests in minutes using a graphical description of the software's behavior. This description is developed from a functional software description. It can easily be updated or expanded as specifications/requirements evolve, and new tests can be generated in minutes.

Teradyne SST offers two product lines. TestMaster is a general-purpose solution that generates tests for virtually any application running under any test execution environment or combination of test execution environments. Hammer CallMaster is a new optimized test generation solution for computer telephony applications that are tested using the Hammer Integrated Telecommunications (IT) Test system.

The new TestMaster Version 1.9 release extends its industry leading test generation capability to include the first integrated solution for automating the QA process from requirements definition through testing, including test documentation, test program generation and requirements tracking. TestMaster is well proven across a wide variety of software systems, driving a variety of commercial and custom execution environments.

The NEW Hammer CallMaster is the first product to automatically generate computer telephony tests from a call flow diagram. Instead of analyzing dozens of pages of call flow diagrams to determine the best test strategy, test developers simply enter the call flow diagram in Hammer CallMaster by clicking and placing call flow icons from a palette of standard and/or custom icons. A test generation algorithm then finds an optimal set of test paths that cover all data and all call flow icons at lease once, outputting the result in Hammer Visual Basic (HVB) code. New tests for feature changes or additions are created in seconds by editing the call flow diagram and re-running the test generation algorithm.

Teradyne SST is a division of Teradyne, a billion-dollar corporation and the world's largest supplier of automated testing solutions.

TERADYNE SST

Mastering Software Quality

Teradyne Software and Systems Test (SST) is the leading supplier of solutions for automating the system definition and test generation process for software systems. Major telecom suppliers currently use Teradyne SST's products across a wide range of telecom systems to achieve 80-90% improvement in test development time and productivity.

Testing is the Bottleneck

Software development generally represents the largest portion of telecom projects, and verification of the software has become the bottleneck in the delivery of most telecom systems. That's because software testing for most telecom companies is still largely a manual process that is very dependent on individual skills and yields highly variable results.

The Toughest Job in Testing is Generating the Tests

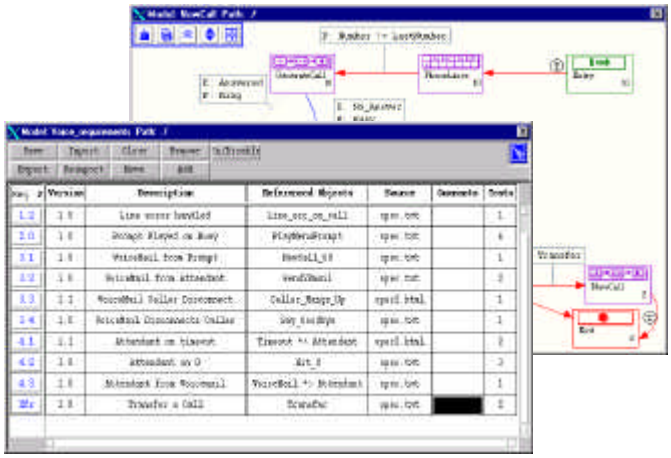
Telecom software testing can be thought of in two major parts:
1) creating the tests
2) executing the tests
The industry has made significant progress in recent years in automating the execution of tests with such tools as GUI recorders and telephony test systems. However, the toughest job in software testing is creating and maintaining comprehensive tests. That's why companies who adopt a systematic automated approach to test generation will have a significant competitive advantage. That's where Teradyne SST comes in.

TestMaster

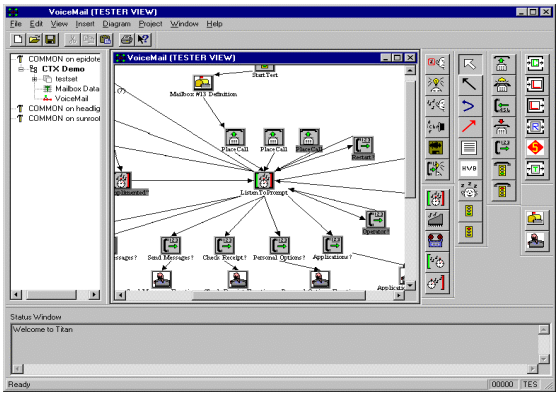
TestMaster provides an integrated solution for automating the QA process from requirements definition through testing, including test documentation, test program generation and requirements tracking. TestMaster is well proven across a wide variety of software systems, driving a variety of commercial and custom execution environments.

Hammer CallMaster

Hammer CallMaster automatically generates computer telephony tests for the Hammer IT Test System from a call flow diagram. Test developers simply enter the call flow diagram in Hammer CallMaster by clicking and placing call flow icons from a palette of standard and/or custom icons. A test generation algorithm then finds an optimal set of test paths that cover all data and all call flow icons at lease once, outputting the result in Hammer Visual Basic (HVB) code.



TestMaster automates the QA process from requirements definition through test



Hammer CallMaster automatically generates Hammer IT test scripts from a call flow diagram

Key Features

TestMaster

- Powerful graphical editor quickly captures software requirements
- Controllable tes coverage optimizes testing resource
- Flexible output formats adapt to any test execution environment
- Automated requirements tracking

Hammer CallMaster

- Develop tests graphically with call flow objects instead of scripts
- Automatically generate that exercise all call flow objects and data
- Modify tests for feature changes or additions in seconds

TESTMASTERS, INC.

Test Management Tools & Consulting

View ad

TESTMASTERS, INC. specializes in functional test management. Armed with extensive experience in software systems testing, TESTMASTERS' consultants assist in creating efficient and cost effective structured test environments. TESTMASTERS' consulting services and product suite, Test Management System™, bolsters our clients' risk mitigation efforts supplementing, current practices or introducing new structured methodologies that enable the implementation of repeatable testing processes. We work closely with both user and technical staff to formulate an integrated team focused on timely completion of an organizations' test objectives.

TESTMASTERS' offers structured test methodologies based on risk assessment, test planning, generic test data development, strong test execution and tracking control methods, and the philosophy of establishing test strategies that build equity in every test effort. TESTMASTERS' test methodologies can be applied to reduce on-going testing costs and improve the overall quality of all development projects.

TESTMASTERS' products and services assist clients in overcoming current testing obstacles and establishing a foundation that will produce cost effective returns well throughout the next millennium.

TESTMASTERS' SERVICES:

- Test Management & Execution
- Test Strategy Development & Planning
- Test Data Development
- Integration of Automated Tools
- Training Programs
- Year 2000 Testing

TEST TOOLS:

TEST MANAGEMENT SYSTEM™

- Test Control System, TCS™
- Problem Reporting System, PRS™

TESTMASTERS, INC.
4055 Oceanside Blvd.-Suite M, Oceanside, CA 92056
(760) 724-2720 or (800) 499-3811
(760) 630-8139 FAX
testmasters@testtools.com
http://www.testtools.com

TESTMASTERS

Test Management Tools & Consulting

- Test Management
- Test Strategy Design
- Test Planning
- Test Execution
- Year 2000 Testing
- Test Date Development
- Integration of Automated Tools
- Training Programs
- Test Management Tools

TESTMASTERS, INC.
4055 Oceanside Blvd., Suite M, Oceanside, CA 92056
Phone (760) 724-2720 or (800) 499-3811 Fax (760) 630-8139
testmasters@testtools.com
http://www.testtools.com

Go back to 1st page

WebSite Validation Technology

To request a copy of the viewgraphs for this presentation, please contact:

Dr. Edward Miller
miller@soft.com
Software Research, Inc.
901 Minnesota St.
San Francisco, CA 94107
(415)957-1441
(800)942-SOFT