The Need for Speed:
Automating Acceptance Testing in an Extreme Programming Environment

Lisa Crispin, Senior Test Engineer, iFactor-e
Contributors: Carol Wade, TRIP.com; Tip House, OCLC.org

"Extreme Programming, or XP, is a lightweight discipline of software development based on principles of simplicity, communication, feedback, and courage. XP is designed for use with small teams who need to develop software quickly in an environment of rapidly changing requirements." Ron Jeffries, http://www.xprogramming.com.

What makes XP Extreme?

As Kent Beck says in Extreme Programming Explained, XP takes commonsense principles and practices to extreme levels. For example: if testing is good, everybody will test all the time (unit testing), even the customers (acceptance testing). Taking anything to extremes can feel scary. While you or I might happily go skiing, we're not likely to ski off the side of a cliff in the manner of Warren Miller. Extreme Programming isn't about taking risks - it's about reducing risks and having fun. It takes courage, but the rewards are immediate.

The XP practices we follow at iFactor-e include:

- pair programming
- test first, then code
- do the simplest thing that works (NOT the *coolest* thing that works!)
- 40-hour week
- refactoring
- coding standards
- small releases
- play the planning game

How is Testing in XP Different?

How does acceptance testing in an XP environment deviate from traditional software testing? First of all, let's look at acceptance testing. XP authors prefer this term to 'functional' testing, as it better reflects how they are written and is more approachable to customers. Acceptance tests prove that the application works as the customer wishes. Acceptance tests give customers, managers and developers confidence that the whole product is progressing in the right direction. Acceptance tests check each increment in the XP cycle to verify that business value is present. Acceptance tests, the responsibility of the tester and the customer, are end-to-end tests from the *customer* perspective, not trying to test every possible path through the code (the unit tests take care of that), but demonstrating the business value of the application.

Should I strap on a helmet and elbow pads?

Testing in an XP environment feels like a run through a half-pipe when you first try it, turning the software development model on its head. The customer is allowed to change her mind anytime. The XP techniques make sure the cost of making changes remain constant throughout the life of project.

Testers may be dismayed at first by the lack of formal written requirements and specifications. To produce small releases very quickly, XP minimizes written documentation. The system is documented through the unit tests, acceptance tests and the code itself. Customers may create mockups of screens and sample reports, but no traditional specifications are written. Design is done primarily with a whiteboard. Collective ownership, promoted by pair programming, reduces the need for written documentation (Which usually is immediately out of date anyway!)

Question: How do you write acceptance test cases without documents?
Answer: You don't. This is the most dramatic way that XP acceptance testing varies from the traditional software development process: In XP, the customer writes the acceptance tests, assisted by the tester.

Other differences between traditional and XP development are more subtle. It's really a matter of degree. XP projects move fast even when compared with the pace at the Web startup where I used to work. It's like running a motocross race when you're accustomed to a street bike. A new iteration of the software, implementing new customer "stories", is released every one to three weeks. The customer must start writing acceptance tests at the beginning of each iteration, as these are the only written "specifications" available. Acceptance tests should run along with unit tests after each integration - which could be several times a day.

From a tester's point of view, the developer to tester ratio in XP looks about as comfortable as street luge. According to Kent Beck, there should be one tester for each eight-developer team. At iFactor-e, the ratio is even higher.

Eeek! Are you SURE protective armor is not required?

Fear not! XP builds in checks and balances that enable a small percentage of test specialists to do an adequate job of controlling quality.

- Becuse the developers write so many unit tests , which they must write before they begin coding - the tester doesn't need to verify every possible path through the code.
- The developers are responsible for integration testing and must run every unit test each time they check in code. Integration problems are manifested before acceptance tests are run.
- The customer is responsible for writing and performing acceptance tests. Naturally, the tester will need to guide the customer in this effort and just as naturally, the customer will soon tire of manual testing and beg for help in the form of test automation.
- The entire development team, not just the tester, is responsible for automating acceptance tests. Developers also help the tester produce reports of test results so that

everyone feels confident about the way the project is progressing.

The roles of the players on an XP team are quite blurred compared with those in a traditional software development process. Thus our iFactor-e XP ("iXP") philosophy is "*specialization is for bugs"*. Here are some of the tasks I perform as a tester:

- Help the customer write stories
- Help break stories into tasks and estimate time needed to complete them
- Help clarify issues for design
- Team with the customer to write acceptance tests
- Pair with the developers to code the application and the test tools
- Pair with the developers to code automated test scripts

Question: Wait a minute. The whole concept of pair programming sounds weird enough. How can a tester pair with a programmer?
Answer: I'm not a Java programmer and our developers don't know the WebART scripting language, but we still pair program. The partner who is not doing the actual typing contributes by thinking strategically, spotting typos and even serving as a sounding board for the coder. This is a fabulous way for developers and testers to understand and work together better. It also gives the tester *much* more insight into the system being coded.

Once you've mustered the courage to jump in to XP, the water's great.

How do I Educate Myself About XP?

Just as you wouldn't attempt to climb Mount Everest without preparing yourself with months of intense training. the XP team needs good training to start off on the right path and stay on it.

Start by reading the XP books. The first book on to be written on XP is Extreme Programming Explained, by Kent Beck. It's a fascinating and quick read. Two new books will be published in the fall of 2000, Extreme Programming Installed, by Ron Jeffries, Ann Anderson, and Chet Hendrickson; and Planning Extreme Programming, by Kent Beck and Martin Fowler.

You can get an overview and extra insight into XP and similar lightweight disciplines from the many XP-related websites, including:
http://www.xprogramming.com
http://www.extremeprogramming.org
http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap
http://www.martinfowler.com

When we at iFactor-e had assembled our first team of eight developers and a tester, we got together and went through Extreme Programming Explained and Extreme Programming Installed as a group, discussing each XP principle, recording our questions (many of them on testing) and deciding how we thought we would implement each

principle. This took several hours but put us all on common ground and made us feel more secure in our understanding of the concepts.

Once your team has read and discussed the XP literature, it's time to get professional training. We hired Bob Martin of ObjectMentor, a consulting company with much XP expertise, for two days of intense training (see www.objectmentor.com for more information). After Bob answered all our questions, we felt much more confident about areas that had previously been difficult for us to understand, such as the planning game, automated unit testing and acceptance testing.

Don't stop there. Talk to XP experts. Look at the Wiki pages and sign up for the egroups. If no XP user group has been formed in your city, start one.

Automating Acceptance Tests

What can you automate?

According to Ron Jeffries, author of XP Installed, successful acceptance tests are customer-owned, comprehensive, repeatable, automatic, timely and produce results that are known to everyone. The "automatic" criterion has given us trouble in some cases, although our goal is to automate whenever it makes sense. Sometimes a mountain bike is the best way up the hill; other times it's easier to get off and walk your bike. For example, we haven't found a cost-effective way to automate Javascript testing. Also we're struggling with how to automate non-Web GUI testing in an acceptable timeframe. Even if we can't automate a test right away, we *can* make it comprehensive, repeatable and timely, and we can publish our results.

Principles of iXP Test Automation

Automated acceptance tests at iFactor-e must meet the following criteria:

- **Modular and self-verifying** to keep up with the pace of development.
- **Verify the minimum criteria for success.** Because the unit tests are comprehensive, we don't need to duplicate it in QA.
- **Perform each function in one and only one place** to minimize maintenance time.
- **Contain modules that can be reused,** even for unrelated projects
- **Do the simplest thing that works. T**his XP value applies as much to testing as to coding.

In addition, the developers try to design the software with testability in mind. This might mean building hooks into the application to help automate acceptance tests. Push as much functionality as possible to the backend, because it is much easier to automate tests against a backend than through a user interface.

iXP Automated Test Design

Appendix A includes a diagram of the design I am using for testing Web applications.

I'm using WebART (see the Tools section below) to create and run the scripts. However, this design should work with any method of automation that permits modularization of scripts. Please see Appendix A for the details of this test design.

Who automates the acceptance tests?

Some sports appear to be individual, when in actuality, they involve a team. Winners of the Tour de France get all the glory, but their victory represents a team effort. Similarly, the XP team may have only one tester, but the entire team contributes to automating acceptance tests. If tools are needed to help with acceptance testing in an XP project, write stories for those tools and include them in the planning game with all the other stories. You'll probably need to budget at least a couple of weeks for creating test tools for a moderately size project.

In the early days of iFactor-e, we initiated a project for the specific purpose of developing automated test tools. This had several advantages, in addition actually producing the tools:

- **Practice with XP** writing stories, playing the planning game, estimating. This gave us confidence in our XP skills that served us future projects.
- **Practice with development technologies.** Developers could experiment with different approaches and get experience with new tools. For example, the developers investigated in advance the advantages of using a dom versus a sax parser on the XML files containing customer test data. Doing this in advance gave us more time to experiment and research technologies than we might have had later with a client project.
- **Mutual understanding.** The team tasked with producing an acceptance test driver consisted of only four members and me, so I was called on to pair program. This exercise gave me insight into how tough it is to write unit tests, write code and refactor the code. The developers gave a lot of thought to acceptance testing and we had long discussions about what the best practices would be. This is a great foundation for any XP team.

Tools

Sky surfers don't leap out of the plane wearing any old parachutes purchased from a discount store. They look for state-of-the-art harness and container systems, main and reserve canopies, helmets and goggles, even altimeters, all designed with their particular needs in mind. XP testers need a good toolbox too, one containing tools designed specifically for speed, flexibility and low overhead.

I've asked several XP gurus, including Kent Beck, Ward Cunningham and Bob Martin, the following question: "What commercial tools do you use to automate acceptance testing?" Their answers were uniform: "Grow your own". Our team extensively researched this area. Our experience has been that we are able to use a third-party tool for Web application test automation, but we need homegrown tools for other purposes.

For **unit testing**, we use a framework called jUnit, which is available free from http://www.junit.org. It does an outstanding job with unit tests. Even though I am not a Java programmer, I can run the tests with jUnit's TestRunner and can even understand the test code well enough to add tests of my own. It's possible to do some functional tests with jUnit. Some XP teams use this tool for automating acceptance tests, but it cannot test the user interface. We didn't find it to be a good choice for end-to-end acceptance testing.

Tools for Creating Acceptance Tests

Some XP pros such as Ward Cunningham advocate the use of spreadsheets for driving acceptance tests. This isn't a new idea. We want to make it easy for the customer to write the tests, and most are comfortable with entering data in a spreadsheet. Spreadsheets can be exported to text format, so that you and/or your development team can write scripts or programs to read the spreadsheet data and feed it into the objects in the application. In the case of financial applications, the calculations and formulas your customer puts into the spreadsheet communicate to the developers how the code they produce should work.

At iFactor-e, we provide a couple of ways for the customer to enter acceptance test cases. Sometimes they work with me to enter test data and test case actions directly into an XML format that is used by our acceptance test driver. If they prefer to use a spreadsheet, we simply convert the spreadsheet into XML format later. We're currently working to make these methods more user-friendly. See Appendix B for a sample acceptance test spreadsheet template.

Appendix C shows a ***partial*** excerpt of a sample XML file used for acceptance test cases. The customer enters a description of the test, data and expected output, steps with actions to be performed and expected results.

Automated Testing for Web Applications

Test automation is relatively straightforward for Web applications. The challenge is creating the automated scripts quickly enough to keep pace with the rapid iterations in an XP project. Like a motocross racer, I'm zipping down hills and slogging through mud, trying to keep up with the pack of developers. For that extra burst of speed, I use WebART (www.oclc.org/webart), an inexpensive HTTP-based tool with a powerful scripting language. WebART enables me to create modularized test scripts, creating many reusable parts in a short enough timeframe to keep up with the pace of development. Javascript testing presents a bigger obstacle. We test it manually and carefully control our Javascript libraries to minimize changes and thus the required retesting. Meanwhile, we continue to research ways of automating Javascript testing.

Our developers wrote a tool to convert test data provided by the customers in spreadsheet or XML format into a format that can be read by WebART test scripts so that we can

automate Web application testing. Even small efforts like this can help you gain that competitive edge in the speedy XP environment.

Automated Testing for GUI Applications

Test automation for non-HTTP GUI applications has been more of an uphill climb. You can travel faster in a helicopter than a mountain bike, but it takes a long time to learn to fly a helicopter; they cost a lot more than a bicycle and ou may not find a place to land. Similarly, the commercial GUI automated test tools we've seen require a lot of resources to learn and implement. They're budget breakers for a small shop such as ours. We searched far and wide but could not come up with a WebART equivalent in the GUI test world. JDK 1.3 comes with a robot that lets you automate testing of GUI events with Java, but it's based on the actual position of components on the screen. Scripts based on screen content and location are inflexible and expensive to maintain. We need tests that give the developers confidence to change the application, knowing that the tests will find any problems they introduce. Tests that need updating after each application change could cause us to lose the race.

We felt that the most important criteria for acceptance tests is that they be repeatable, because they have to be run for each integration. We decided to start by developing our own tool, "TestFactor-e", that will help customers and testers run manual tests consistently. It will also record the results. We're now enhancing this tool to feed the test data and actions directly into application backends in order to automate the tests.

Reports

Getting feedback is one of the four XP values. Beck says that concrete feedback about the current state of the system is priceless. An extreme skier constantly monitors snow conditions, the course, his speed, the state of his equipment, all while keeping an ear out for the avalanche that may be coming along behind him. He accommodates these factors with changes in speed, trajectory and position. The XP team needs a constant flow of information to steer the project, making corrections in mid-course just as the skier would. The team's continual small adjustments keep the project on course, on time and on budget. Unit tests give programmers minute-by-minute feedback. Acceptance test results provide feedback about the "Big Picture" for the customer and the development team.

Reports don't need to be fancy, just easy to read at a glance. A graph showing the number of acceptance tests written, the number currently running and the number currently succeeding should be prominently posted on the wall. You can find examples of these in the XP books. Our development team wrote tools to read result logs from both automated tests and manual tests run with "TestFactor-e". These tools produce easy-to-read detail and summary reports in HTML and chart format.

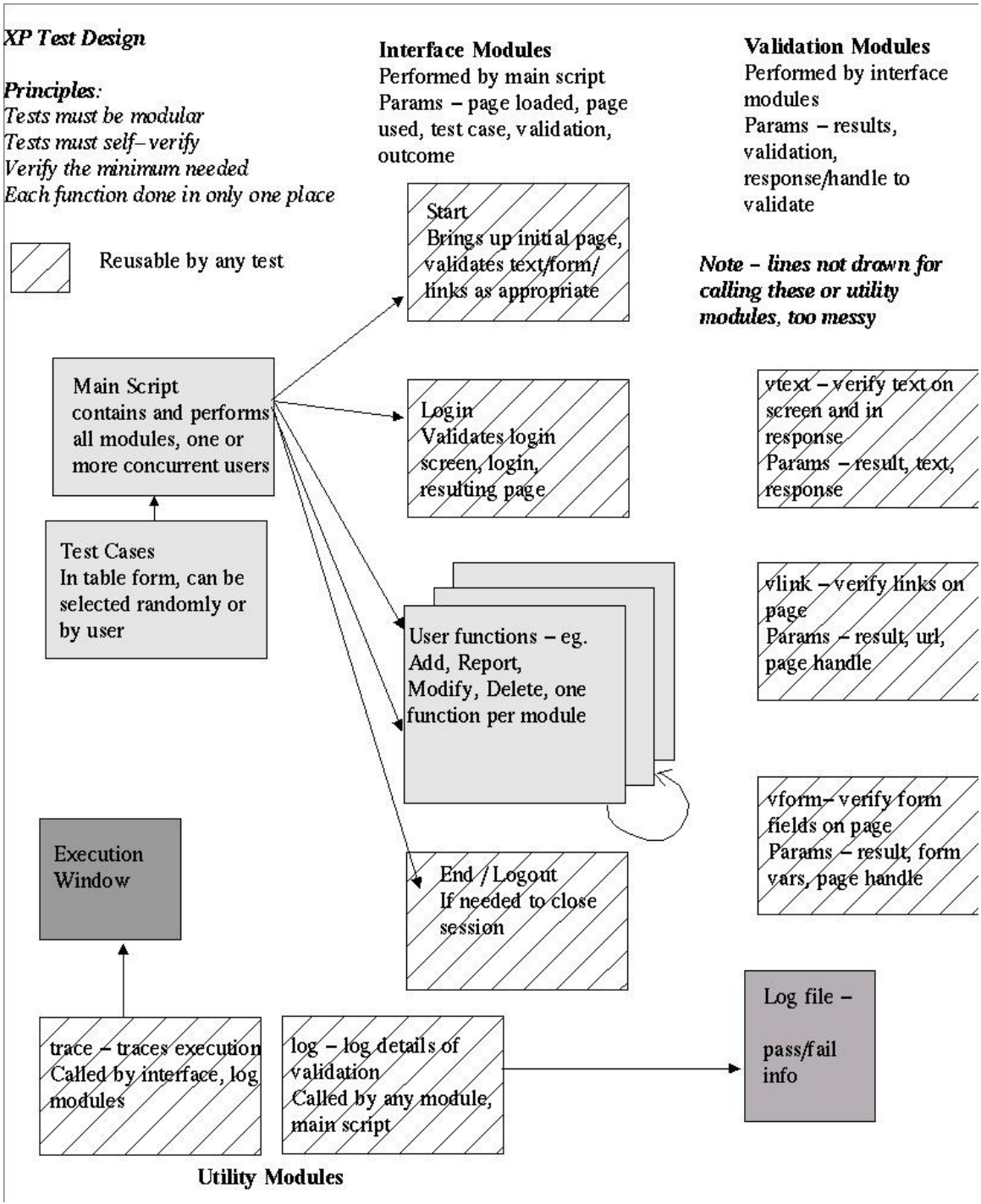
With all this feedback, you'll confidently deliver high-quality software in time to beat your competition. You'll meet the challenges of 21st century software development!

Appendix A: Test Design Description

Because this paper is so long I am not going to include code examples here. Contact me at lisa.crispin@ifactor-e.com if you would like some sample WebART scripts.

Main Script

The **main module** calls the supporting module to perform a typical user scenario and to validate the system responses at strategic points along the way. A basic user scenario is created for each customer story. The **supporting modules** are divided into several groups based on their function. These modules are described below.



The main module passes to the supporting modules test cases from tables of test data along with other parameters.

Sample basic user scenario:

Action	***Minimum Passing Criteria***
1. Go to Add Entry page.	Browser challenges for authentication.
2. Login.	Valid userid/password does not get error.
3. Add a time entry using selected test case (may be done with multiple users).	Form fields are on page at start. After submit, still on Add screen (checks by text and forms.
4. Go to Generate Reports page.	Browser challenges for authentication (Currently, because these are two separate scripts, could be combined).
5. Login .	Valid userID/password does not get error.
6. Generate a report for the date range specified for the test case added in Action 3.	Form fields are on generate reports page After submitting, report contains correct text, correct links back to generate page, and contains the userID, release and iteration from the test case added in Action 3.
7. Log out.	Log out successful message displayed.

Interface Modules

Called by the main script module (and possibly other interface modules) to perform user functions and to validate the correct system response. Some of these modules such as start, login and exit can be used by multiple tests for the same application. The main module passes **parameters** to the interface modules as follows:

page loaded - An *output* parameter; it receives the value of the page loaded by the module. For example, the start module loads the addEntry.jsp page.

page used - An *input* parameter; it is the handle of the page used by the interface module to know what page to load.

test case data - An *input* parameter; it is data to be parsed by the interface module and used for input or validation. For example, for a login module, the test case consists of an *userid* and *password* .

validation clause - Tells the interface module how much validation to perform. A value of STD indicates to do only the minimum validation; extended validation options may be specified and added to STD.

outcome - An *output* parameter that receives a value of **PASS** or **FAIL** indicating the overall outcome of the call.

Additional parameters may be used if needed (eg., more than one page needs to be loaded).

Validation Modules

Called by interface modules to check for specific conditions in a system response and return a pass or fail condition. The validation modules in turn call **utility** modules to record the results. Parameters are:
results - An *output* parameter which returns **PASS** or **FAIL** to the calling module

controls - An *input* containing values that control how the validation is done, specific to each validation module.

response or handle - An *input* parameter containing either the system response to be validated or the handle of the page into which the response was loaded.

Currently, there are three validation modules that can be used by any test:
vtext validates that a response contains specified text. Parameters are:

result - PASS or FAIL

text - the text that must be present for the validation to pass

response - the system response to check for the text string.

vlink validates that a page contains a specific link. Parameters are:

result , *urlMatch* - the value which must exist in a link in the page whose handle is in pPage

pPage - the **page handle** of the page being validated.

vform validates that a page contains a specified form. Parameters are:

result , *formvars* - one or more required variables for the form - if any are missing, the validation fails.

pPage - the **page handle** of the page being validated.

Utility Modules

Currently there are two utility modules which can be used by any test:
trace - Displays execution tracing information in the WebART execution window. Called by interface and log modules. Without going into detail of the many parameters, they reveal who called it and what happened.
log - Records validation outcomes in a log file. Parameters are:

type - detail or summary, *outcome* - PASS or FAIL

validation - describes the validation performed.

Appendix B: Sample Acceptance Test Spreadsheet Template

	A	B	C	D	E
1		**Ref #:**	Template for acceptance test: Timecard/QA/AcceptanceTest.sdc		
2		**Iteration:**			
3		**Functionality proven by this test case * is / is not * critical**	**What does this test?** *Example: Tests to make sure that when a time entry record is input, it is saved to the database and the report generated correctly.*		
4		**What to do:**	*Examples given in italics*		
5	**Step**	**Command/URL**	**Action**	**Input Data**	**Expected Output**
6	1	*Access wwwltimecardladdEntry in browser*	*Select a project, release, iteration, task, and enter duration, date and comments*	*Row 1, columns Project, Release, Iteration, Task, Duration, Date, Comments*	*Selections displaying on screen*
7	2	*Still on wwwltimecardladdEntry*	*Click the save button*		*Message that data was saved to database*
8	3	*Access wwwltimecardlgenerateReports in browser*	*Select a project, start date and end date*	*Row 1, columns Project, Start Date, End Date*	*Report generated – data matches row 1 columns Project, Release, Iteration, Duration, Cost and Total Cost*
9	4	*Repeat steps 1 – 3 with each row in the test case spreadsheet*			

Appendix C: Partial Excerpt of XML Template for Acceptance Test Cases

<?xml version="1.0" encoding="UTF-8" standalone="no" ?>

```
<!DOCTYPE at-test SYSTEM "at-test.dtd" [
  <!ELEMENT input ANY >
  <!ELEMENT loan-amount ANY >
  <!ELEMENT interest-rate ANY >
  <!ELEMENT term-of-loan ANY >
  <!ELEMENT output ANY >
  <!ELEMENT monthly-payment ANY >
]>

<at-test name="calc-monthly-payment" version="1.0" severity="CRITICAL">

  <at-project>mortgage-calc</at-project>

  <at-description>
    Enter loan amount, interest rate, term of loan (in months)
    to calculate monthly payment.
  </at-description>

  <at-data-sets>
    <at-struct id="values">
        <input>
        <loan-amount>1000000000.00</loan-amount>
        <interest-rate>0.5</interest-rate>
        <term-of-loan>1200</term-of-loan>
      </input>
      <output>
        <monthly-payment>A big, fat wad of dough!</monthly-payment>
      </output>
        </at-struct>
  </at-data-sets>

  <at-plan>

    <at-step name="populate-loan-amount">
      <at-action>
        <at-text>Enter "{0}" in the "Loan Amount field".</at-text>
        <at-value dset="values" select="/input[2]/loan-amount"/>
      </at-action>
      <at-expect>
        <at-text>Cursor moved to "Interest Rate" field for input.</at-text>
      </at-expect>
    </at-step>

   </at-plan>

</at-test>
```