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

Lisa Crispin
Senior Test Engineer
Tensegrent
6100 Greenwood Plaza Blvd
Greenwood Village, CO 80210
USA
1.303.268.4621
lisa@tensegrent.com

Tip House
Chief Systems Analyst
OCLC, Inc.
www.oclc.org
6565 Frantz Rd.
Dublin, OH 43017
USA
1.614.761.5139
tip_house@oclc.org

Contributor: Carol Wade
HealthLanguage, Inc.
3960 Lewiston St.
Aurora, CO 80011
USA
1.303.307.4400
carol_r_wade@hotmail.com

ABSTRACT
In eXtreme Programming Explained, Kent Beck compares eXtreme Programming (XP) to driving a car: the driver needs to steer and make constant corrections to stay on the road. If the XP development team is steering the car, the XP tester is navigating. Someone needs to plot the course, establish the landmarks, keep track of the progress, and perhaps even ask for directions. Acceptance tests must go beyond functionality to determine whether the packages meet goals such as specified performance levels. Automating end-to-end testing from the customer point of view can seem as daunting as driving along the edge of a cliff with no guard rail. At Tensegrent, a software engineering firm in Denver organized around XP practices, the developers and the tester have worked together to design modularized, self-verifying tests that can be quickly developed and easily maintained. This is accomplished through a combination of in-house and vendor-supplied tools. This presentation covers:

- How to focus acceptance testing for XP
- How to design automated tests that are low-maintenance and self-verifying
- How to apply the values of XP to test automation
- Ways to gather metrics and provide useful reports

The Tensegrent lightweight test methodology isn't specific to a particular test tool. It allows acceptance testing to keep pace with the rapid iterations of an XP project.

KEYWORDS
Testing, automated testing, acceptance testing, test scripts, tester, test tools, web testing, GUI testing.

INTRODUCTION
The three XP books give detailed explanations of many aspects of the development side of XP. The test engineer coming from a traditional software development environment may not find enough direction on how to effectively automate acceptance tests while keeping up with the fast pace of an XP project. In an XP team, developers are also likely to find themselves automating acceptance tests – an area where they may have little experience. Automating acceptance testing in an XP project may feel like driving down a 12% grade in a VW bug with a speeding semi in the rear-view mirror. Don't worry – like all of XP, it requires courage, but it can – and should – be fun, not scary.

The XP practices we follow at Tensegrent 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

We apply these same practices to testing – including pair testing.

Do XP teams really need a dedicated tester? It's hard for a tester to answer this in an unbiased manner. In my experience, even senior developers don't have much testing experience, beyond unit and integration tests and perhaps load tests. They tend to write acceptance tests only for "happy paths" and don't think of the nasty evil steps that might break the system. At Tensegrent, we had one project wrapping up while another one was starting, so a decision was made to do the first two-week iteration of the new project with a developer serving as a part-time tester. By their own admission, without an experienced tester to push them, the developers got 90% of all the stories done by the end of

the iteration. To the customer, this looked like nothing at all was done, and they were very unhappy. It took some work to win back the customer's trust.

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. 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. Acceptance tests may also include load, stress and performance tests to demonstrate that the stability of the system meets customer requirements.

Should I strap on a helmet and arm the air bags?

Testing in an XP environment feels like a drive through twisting mountain roads at first. When I first read eXtreme Programming Explained, the very idea of testing without any formal written specifications seemed a bit TOO extreme. It's been difficult learning all the different ways I can contribute to the team's success. My roles can be confusing and conflicting – I'm part of the development team, but I need a more objective viewpoint. I'm a customer advocate, making sure the customer gets what she pays for. At the same time, I need to protect the developers from a customer who wants MORE than they paid for.

While XP is definitely a new way to drive, the road isn't as unfamiliar as some might think. For example, many people new to XP think that XP projects produce very little documentation. This hasn't been our experience. For one thing, the acceptance tests themselves become the main documentation of the customer requirements. They can be quite detailed and extensive. As an XP project progresses, many other documents may be produced: installation instructions, UML documents, Javadocs, developer setup documents, the list goes on. The difference between these and the documents in many traditional projects is, the XP project documents are up to date and accurate

Question: How do you write acceptance test cases without documents?

Answer: You don't need documents, because you have a customer there to tell you what she is looking for. Not that this is always easy. In my experience, it is fairly easy to get a customer to come up with tests for the intended functionality of the system. What is more difficult, and requires a tester's skill, is to make sure the customer thinks about areas such as security, error handling, stability, and performance under load.

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 the fast lane on the Autobahn. A new iteration of the software, implementing new customer "stories", is released every one to three weeks. My goal is always to get acceptance test cases defined within the first day or two of an iteration, as these are the only written "specifications" available. For our projects, the acceptance test definitions have been a joint effort of the team.

From a tester's point of view, the developer to tester ratio in XP looks about as comfortable as driving through the desert in an un-air-conditioned Jeep. According to Kent Beck, there should be one tester for each eight-developer team. At Tensegrent, the ratio gets even higher.

Eeek! Are you SURE protective gear isn't 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.

- Because 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 gives input to the acceptance tests and provides test data.
- 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.

A caveat – if developers aren't diligent in writing and running unit tests and integrating often, you're going to have to hire more testers. A couple of iterations into our first project at Tensegrent, I told my boss I thought we'd have to hire more testers, there was no way I could keep up! The problem was simply that the developers hadn't gotten the hang of "test before code" yet. Once they did a thorough job of unit and integration testing, my job became much more manageable.

The roles of the players on an XP team are quite blurred compared with those in a traditional software development process. Thus our Tensegrent XP ("XP") philosophy is "*specialization is for insects*". 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 develop test tools, automated test scripts, and/or test data.

Question: 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 bad habits, 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.

I was reluctant to pair test at first. If the developers wrote the test scripts, would I be able to understand them and maintain them? The developers weren't anxious to pair with me for testing, either. They felt too busy to spare time for acceptance testing. Then we had a project where I needed very complicated test data loaded into a Poet database for testing a security model. By pairing with a developer, I finished in at least half the time it would have taken to do it alone, and did a better job. Now developers take turns on "test support" to produce test scripts and data needed for automation, sometimes also to help define test cases if I'm having trouble understanding a story.

Once you've mustered the courage to switch to the XP fast lane, it feels fun and safe.

How do I Educate Myself About XP?

Just as you wouldn't attempt to drive a Formula One car without preparing yourself with training and practice, the XP team needs good training to start off on the right road and stay on it.

Start by reading the XP books. The first written about XP is Extreme Programming Explained, by Kent Beck. The other two are also essential: 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 Tensegrent 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, among other things, customer-owned and automatic. However, customer-owned does not necessarily mean customer-written. In fact, as Kent Beck points out in Extreme Programming Explained, customers typically can't write functional tests by themselves, which is why an XP team has a dedicated tester: to translate the customers ideas into automatic tests.

Even with a dedicated tester, though, the "automatic" criterion has given us some trouble. We automate whenever it makes sense, but like most things, it is a trade-off. When you have to climb a steep dirt road every day, a four-wheel drive vehicle is a necessity, but it's overkill if you're just cruising around the block.

For example, we haven't found a cost-effective way to automate Javascript testing (so, we just avoid using Javascript). And we're also struggling with how to automate non-Web GUI testing in an acceptable timeframe.

It costs time and money to automate tests and to maintain them once you've got 'em. Recently we had a contract for three two-week iterations with four developers and myself to develop some components of a system for a customer. While the system involved a user interface, the design of the UI itself was to be done later, outside of our project. We developed a very basic

interface to be able to test the system. The system involved multiple servers, interfaces, monitors and a database. Full test automation would have been a big effort. It didn't make sense to spend the customer's tight resources on scripts that had a short life span. Still, I automated the more tedious parts of the testing so I could get the tests done in time. In addition, I needed scripts for load testing. About 40% of the testing ended up automated. For a longer project, I would prefer to automate more.

Principles of XP Functional Test Automation

To get more automation, you have to make automation pay off in the short term, and this means spending less time developing and maintaining the automated tests. Here are the principles we are using to accomplish this:

- **Drive the test automation design with a "Smoke Test"**, a broad but shallow verification of all the critical functionality.
- **Design the tests like software,** so that the automated tests do not contain any duplicate code and have the fewest possible modules.
- **Separate the test data from the test code**, so that you can deepen test coverage by just adding additional test data .
- **Make the test modules self-verifying** to tell you if they passed or failed of course, but also to incorporate the unit tests for the module.
- **Verify only the function of concern for a particular test**, not every function that may have to be performed to set up the test.
- **Verify the minimum criteria for success.** "Minimum" doesn't mean "insufficient". If it weren't good enough, it wouldn't be the minimum. Demonstrate the business value end-to-end, but don't do more than the customer needs to determine success.
- **Continually refactor the automated tests**, by combining, splitting, or adding modules, or changing module interfaces or behavior whenever it is necessary to avoid duplication, or to make it easier to add new test cases
- **Pair program the tests**, with another tester or a programmer.
- **Design the software for testability**, such as 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. I sit in on the developers' iteration planning and quick whiteboard design sessions. If I perceive business logic getting into the front end, for example in Javascript, I challenge the wisdom of such a move.

An XP Automated Test Design

Appendix A gives an example of a lightweight test design illustrating the application of the principles we have been using successfully at Tensegrent. I'm using WebART (see the Tools section below) to create and run the scripts. However, this design approach should work with any method of automation that permits modularization of scripts. The appendix gives details on downloading both the sample scripts and WebART.

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 Tensegrent, 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

To keep the XP car humming, XP testers need a good toolbox: 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 can't 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. 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 Tensegrent, we provide a couple of ways for documenting acceptance test cases. Usually we use a simple spreadsheet format, separating the test case data itself from the description of the test case steps, actions and expected results. We've also experimented with entering test cases in XML format which is used by an in-house test driver. We're continuing to experiment with the XML idea, but the spreadsheet format has worked well. 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 test case consists of 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. This is always toughest in the early iterations. There are times that I feel like the slow old car blocking the fast lane. For that extra burst of speed, I use WebART (http://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 you 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 plan to enhance this tool to feed the test data and actions directly into application backends in order to automate the tests. As we have only been developing web applications, this effort is on the back burner.

No matter what the system being tested, it takes time to get up to speed with automation. I plan to do manual testing in the first iteration. At the start of the second iteration, I can start automating, using the method described in Appendix A. There are times I run into a roadblock which sets me back a day or two. The solution to that is to find someone to pair with me. As the tester in an XP project, you may feel lonely at times, but remember, you aren't ever alone!

Reports

Getting feedback is one of the four XP values. Beck says that concrete feedback about the current state of the system is priceless. If you're on a long road trip, you check for road signs and landmarks that tell you how far along your route you've come. If you realize you're running behind, you skip the next stop for coffee or

push the speed a bit. If you're ahead of schedule, you might detour to a more scenic road. The XP team needs a constant flow of information to steer the project, making corrections to stay in the lane. 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: LIGHTWEIGHT TEST DESIGN

XP Automated Test Design

The sample scripts used to illustrate the test design are written with a test tool called WebART (http://www.oclc.org/webart/). Any test tool which permits modularization and paramaterization of the scripts should support this design. To download a soft copy of the sample scripts, go to http://www.tensegrent.com and click on the "Sample WebART Scripts" link.

The Sample Application

Our sample application is a telephone directory lookup website, http://www.qwestdex.com. This is certainly not intended as an endorsement of Qwest and we have no connection with them, it was just a handy public application with characteristics that allow us to illustrate the tests.

The Smoke Test

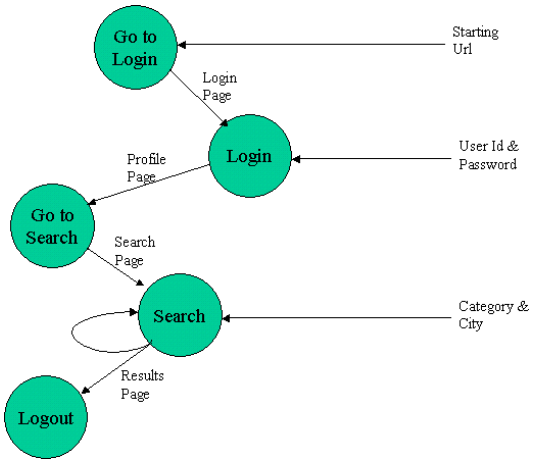
We will consider the critical functionality to be logging into the site and finding the businesses within a certain city and category. Pretend that this is the most important story in the first iteration. Here's the basic scenario we want to test:

Action	***Minimum Passing Criteria***
Go to login page	Page contains the login form
Login	Valid login name and password brings up profile page
Search for valid category in specified city	Valid search retrieves table of businesses
Logout	Page contains link to login page and home page

The Test Design

We know that there will be more functionality to test in subsequent iterations, but we will use the simplest design we can think of to accomplish these tests without duplication. Then we will refactor as necessary to accommodate the additional tests.

The modules will be Go to Login, Login, Go to Search, Search, and Logout. Here is a diagram showing how the modules are parameterized:



Separating the test data from the code

The items on the right side of the diagram represent test data: the URL of the login page, the user id and password to use to login, and the category and city to search. The test data is segregated into a test case file, which is read in by the test when it executes. Here is sample content of that file to run a single test case:

```
smoketest
  [
  :iter1:
        Url <url=http://qwestdex.com>
        UseridPassword
       <uid=bob&psw=bob>
        CatCity <cat=banks&city=dallas>
        ]
```

Verification

The main modules use a set of primitive validation modules to check for the specific conditions required in a system response and determine a pass or fail condition. The validation modules in turn call **utility** modules to record the results.

This example uses the following three validation modules:

vtext validates that a response contains specified text. for the text string.

vlink validates that a page contains a specific link.

vform validates that a page contains a specified HTML form.

Utility Modules

There are also two utility modules which are used by the main modules:

trace - Displays execution tracing information in the WebART execution window, for debugging the tests..

log - Records validation outcomes in a log file. The "zslog" module in the sample scripts writes test results out in XML format. An in-house tool from Tensegrent called TestFactor-e builds an HTML page from this log file showing the results with color-coding for pass, not run and fail. See Appendix B for an example.

Creating the Scripts

Creating the first set of scripts is the hard work. Once you have a working set of modules, you can reuse entire modules in some cases or turn them into templates in other cases. Here are the steps I use (preferably as part of a pair) to create test scripts:

1. Capture a session for the scenario I want to test. See "capqwest" in the sample scripts as an example.

2. Copy "qwmain", "zsqwlogin" and the other supporting modules that I already have to new names. Strip out the code that was specific to that application.

3. Paste in the code specific to the scenario I want to test, copying from the captured script into the newly created "templates". Use XP principles here: work in small increments, make sure your scripts work before you go on. For example, first see if you can get the login to work. Then add the search. Then add the logic for switching depending on the pass/fail outcome. Remember to do the simplest thing that works and add complexity only as you need it.

Appendix B: 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>
```

Appendix C: Sample Acceptance Test Spreadsheet

	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 www/timecard/addEntry 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 www/timecard/addEntry*	*Click the save button*		*Message that data was saved to database*
8	3	*Access www/timecard/generateReports 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*			