

## **Acceptance testing**

Jari Aarniala (jari.aarniala@cs.helsinki.fi)

Helsinki October 30, 2006

UNIVERSITY OF HELSINKI

Department of Computer Science

Tiedekunta/Osasto — Fakultet/Sektion — Faculty		Laitos — Institution — Department	
Tekijä — Författare — Author Jari Aarniala (jari.aarniala@cs.helsinki.fi)			
Työn nimi — Arbetets titel — Title Acceptance testing			
Oppiaine — Läroämne — Subject			
Työn laji — Arbetets art — Level		Aika — Datum — Month and year October 30, 2006	Sivumäärä — Sidoantal — Number of pages 11 pages
Tiivistelmä — Referat — Abstract Acceptance testing is a high-level testing procedure used to verify that a software system behaves as specified by the customer. This paper examines acceptance testing mainly from the viewpoint of the Extreme Programming software development process. In addition to discussing the principles of acceptance testing, some real-world tools are also presented.			
Avainsanat — Nyckelord — Keywords testing, acceptance testing, agile software development, extreme programming			
Säilytyspaikka — Förvaringsställe — Where deposited			
Muita tietoja — övriga uppgifter — Additional information			

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>An overview of acceptance testing</b>	<b>1</b>
2.1	Challenges in acceptance testing . . . . .	2
2.2	Acceptance testing and other types of testing . . . . .	3
<b>3</b>	<b>Tools for acceptance testing</b>	<b>4</b>
3.1	FIT and the table-based approach to acceptance testing . . . . .	5
3.2	Selenium and web-based acceptance testing . . . . .	8
<b>4</b>	<b>Conclusion</b>	<b>9</b>
	<b>References</b>	<b>10</b>

# 1 Introduction

*“Developers write unit tests to determine if their code is doing things right. Customers write acceptance tests to determine if the system is doing the right things.”* [MC01]

*Acceptance testing* is a testing discipline that operates on the highest level of a software application: the customer interface, be it graphical or implemented in some other way. The term *acceptance testing* itself is strongly related to the agile software development method Extreme Programming (or XP for short) [BA04]. In XP, the customer requirements are gathered in the form of *user stories*. Acceptance tests, also called *customer tests*, are tests owned and defined by the customer to verify that a story’s implementation is complete and correct. [Rog04]

In this paper, the principles of acceptance testing are mostly examined from the viewpoint of XP teams. Acceptance testing is discussed on a general level in section 2, with a brief comparison with other types of testing in section 2.2. Section 3 takes a look on some real-life tools for acceptance testing.

## 2 An overview of acceptance testing

Acceptance tests, as the name implies, are mainly a means of *verifying* that a software system behaves as expected under various use cases. However, acceptance testing has many other beneficial aspects, especially if executed in conjunction with other XP practices.

First of all, writing acceptance tests can be a great communication aid within a team. Traditionally, the customer, while always right, doesn’t usually know what she wants. Approaching the requirement specification in a XP project by writing user stories with the customer and then turning them into acceptance tests can help both the developers and customers understand the problem domain and the solution, i.e. the application, better. Acceptance tests can help structure the conversation within the team and also provide a common vocabulary, or a *domain language*, for use in discussions. [Rog04]

One of the most radical and often-critiqued aspects of XP is its inherent lack of documentation: in XP, documents *can* be written, but they are not valued as much as actual working software is. [BA04] Acceptance tests can also be seen as a partial replacement for documentation - especially for requirements documents. Some argue [Rog04] that traditional requirements documents are easy to just skim thru in a matter of minutes without giving them much thought: acceptance tests, on the other hand, can't be easily dismissed. They provide concrete feedback on the implementation status of the stories in the system, and unlike requirements documents, they can actively point out shortcomings in the knowledge of the team and the thinking of the customer. Additionally, acceptance tests, if run regularly, don't go out of sync with the actual application, while a requirements document may easily lag behind. Keeping the acceptance tests up-to-date while the project proceeds helps the team continuously clarify and refine the requirements.

Finally, acceptance tests can be considered as an absolute criteria for deciding when a feature is complete, or "knowing when to stop." If a acceptance test fails, the story's simply not implemented in a way that's acceptable to the customer. The same model can be seen in work on a micro level in the XP practice of Test-First Development, which is usually practiced on the unit test level. By writing a unit test first, the developer knows exactly when he can stop writing new code: once the unit test passes, she's added just enough functionality to the class under test to pass the test.

## 2.1 Challenges in acceptance testing

Acceptance testing is a challenging task because it impacts the software system as a whole, and requires the attention of the whole team behind the product. As the customer is an ever-present part of the project team in a XP project, the testing effort concerns an even larger group of people. Due to its challenging nature, the practice of acceptance testing is often neglected in a project. [Rog04]

While an on-site customer helps with communicating and formulating the user requirements, finally in the form of acceptance tests, a customer's vision of the system isn't comprehensive in terms of non-functional aspects, for example. Thus, while working on acceptance tests with a customer might result in a well-defined test suite

that exercises the functional side of the system thoroughly, difficulties arise when attention needs to be paid to areas such as security, error handling, stability and performance under load. [CHW01]

While there are tools that make it easier for customers to express the expected behaviour of a system (see chapter 3), customers typically aren't cut out for writing functional tests entirely by themselves. For this reason, an XP team should have a dedicated tester. [BA04] "Dedicated" doesn't just refer to a person who commits to testing the application to her best effort, but instead to an actual tester who's trained on the subject. A typical software developer usually has some testing experience with unit and integration tests, but even senior developers aren't usually experienced enough to fill the position of a test expert in a team: they rarely come to think of all the unusual situations that might break the system. [CHW01]

## 2.2 Acceptance testing and other types of testing

Technically speaking, writing acceptance tests is usually simpler than writing unit or integration tests. This is due to the fact that acceptance tests are written, or at least need to be understood, by customers themselves. The underlying machinery of an automated acceptance test framework is probably more complex than that of a unit test driver, but from the test writer's point of view, things are kept simple. This is done with the help of high-level abstractions, such as macros that represent logical actions that can be executed against the system. Comparing acceptance testing with unit and integration testing gives us a better image on where acceptance testing fits in the big picture.

Unit tests are usually the tests with the smallest scope in a project: they test the functionality of a single class, or other small unit in a software system. While acceptance tests are usually written in a custom, domain-specific language that's easy for the customer to comprehend, unit tests are usually written in the implementation language of the software system itself.

Rogers [Rog04] points out a few differences in the execution of unit tests versus acceptance tests. For example, it is fine for an acceptance test to fail until the story for which it was written is implemented. Unit tests, on the other hand, should never fail once integrated into the system.

A great difference can also be observed in the fact that unit tests are, by definition, run in isolation, while acceptance tests have tangible side-effects. In unit tests, any dependencies the class under test might have on the file system, databases etc are removed for the duration of the test by wiring the class with *stubs* or *mock objects* that mimick the behaviour of the actual objects. The isolation guarantees that a failing unit tests shouldn't affect other unit tests, while a failing acceptance test might well cause others to fail, as acceptance test boundaries usually overlap.

Integration tests are closer to acceptance tests in that they work on a higher level than unit tests and verify the behaviour of different parts of the system working together. What's different from acceptance testing, though, is that in integration tests, parts of the system are often replaced with *stubs*, as is the case with unit tests. Integration tests are also strictly developer-tests, written in the native language of the application itself.

### 3 Tools for acceptance testing

As we've seen in previous chapters, acceptance tests capture acceptance criteria on a high level, using different sorts of abstractions to make writing the tests easy for a customer. As with any other type of testing, acceptance testing can be done manually. But in order to get the maximum benefit from the tests, one needs tools to automate the actual execution of the tests. Doing automated regression testing and including the acceptance test suite in an XP-style continuous integration cycle also requires the tests to be implemented using a suitable tool.

On the acceptance testing tool front, most readily available tools are meant for testing web-based applications. Cost-effective solutions for automating the testing of traditional graphical user interfaces are sparse, but web-based applications have their share of problems as well: for example, extensive use of JavaScript can make automating tests harder, and should be avoided altogether, according to some. [CHW01] However, as we'll see in chapter 3.2, there are tools for running web-based acceptance tests within an actual web browser, thus making dealing with script-heavy web user interfaces easy.

Many members of the testing community have stated that developing one's own

testing tool is a viable option as opposed to using a commercial one. [CHW01] This kind of a “do-it-yourself” -mentality can be seen as a sign that the available testing tools are lacking in some way and are definitely not suitable for all purposes. However, there are a number of tools that have gained widespread adoption recently. These will be examined in more detail in the following chapters.

### 3.1 FIT and the table-based approach to acceptance testing

As acceptance tests are meant to test the application’s behavior from the customer’s standpoint, it should be possible for the customers to write the tests themselves, or at least understand the test specifications without any knowledge on programming languages etc. A common approach to writing acceptance tests is the table-based approach, advocated by Ward Cunningham, among others. [CHW01] The rationale behind this approach is that most customers are comfortable with entering data in a spreadsheet application, and having the data they’ve entered automatically processed by different formulas, macros etc. Thus, using a spreadsheet-type of an interface for entering expectations of a program’s behaviour should feel familiar for them.

Perhaps the most well-known implementation of a table-based acceptance testing tool is Fit<sup>1</sup> (or Framework for Integrated Test), introduced by Ward Cunningham in 2002. [WP-FIT] It is an open-source tool that allows customers to provide examples of their software should work by writing them down in a table format. In Fit’s case, the tests are represented as standard HTML tables: the customers are free to use any tool (such as a word processor) for creating the tables, as long as the data can be exported to HTML. An example of a Fit-based test specification for a program calculating the weekly compensation for a worker is shown in figure 1.

Once the examples, or expectations, on the program’s behaviour have been written down, they must be mapped to the application somehow. In Fit, this mapping is done by *fixtures* written by programmers, most often in the actual implementation language of the application. A fixture reads in the data from a table, and exercises the application using the data as the input. Finally, the fixture runner compares the customer-set expectations with the actual results and reports any errors by

---

<sup>1</sup><http://fit.c2.com/>

**Basic Employee Compensation**

For each week, hourly employees are paid a standard wage per hour for the first 40 hours worked, 1.5 times their wage for each hour after the first 40 hours, and 2 times their wage for each hour worked on Sundays and holidays.

Here are some typical examples of this:

<u>Payroll Fixtures</u>	<u>WeeklyCompensation</u>		
<u>StandardHours</u>	<u>HolidayHours</u>	Wage	Pay()
40	0	20	\$800
45	0	20	\$950
48	8	20	\$1360 <i>expected</i> \$1040 <i>actual</i>

Figure 1: An example of a Fit test specification (source: the Fit website)

color-coding the table rows: red for failures, green for passed tests.

A fixture for the weekly compensation test introduced above is depicted in figure 2, written in Java. The fixture example illustrates how Fit fixtures are essentially nothing but the thinnest possible layer of code written in order to map the test specifications to the actual application code. Out of the box, the Fit framework is available for a variety of different programming platforms, such as Java, .NET, Python, Smalltalk and C++.<sup>2</sup>

It should be noted here that while Fit makes specifying the test criteria easy for non-technical persons, there's still a need for programmers when using Fit: the mapping of a specification table to the actual application requires programmer intervention. These mappings, however simple, mean that adding tests for new functionality is not something a customer can do by herself. Additional expectations can naturally

<sup>2</sup><http://fit.c2.com/wiki.cgi?DownloadNow>

```
public class WeeklyCompensation extends ColumnFixture {

    // input values retrieved from the table
    int standardHours;
    int holidayHours;
    Currency wage;

    public Currency pay() {
        WeeklyTimesheet timesheet =
            new WeeklyTimesheet(standardHours, holidayHours);
        // the return value that will be
        // compared against the expectations
        return timesheet.calculatePay(wage);
    }
}
```

Figure 2: A Fit test fixture (in Java, adapted from the Fit website)

be added to existing tests at any time, but implementing new tests depends on the availability of the team's programmers.

Fit-based tests are executed using a command-line tool that accepts a single HTML file as its input. For a more distributed approach to using Fit, there is a tool called FitNesse.<sup>3</sup> Essentially, FitNesse builds on top of Fit, but places the test specifications on a website from which the tests can actually be executed by the click of a button. FitNesse is implemented as a *Wiki*, meaning that the web pages can be freely edited by the team members without any specific tools other than a web browser. As FitNesse uses Fit for running the actual tests, the process for writing test fixtures is the same.

---

<sup>3</sup><http://www.fitnesse.org/>

## 3.2 Selenium and web-based acceptance testing

Fit and FitNesse, presented in the previous chapter, are acceptance testing tools that can be used on almost any piece of software: instead of working automatically against some pre-defined interface, Fit tests must always be mapped to the target system case-by-case using fixtures. By nature, these fixtures are closely related to the internal structures of the program under test, and while they don't operate on the class level as unit tests do, for example, the fixtures do have to know quite a lot about the implementation of system (that is, Fit tests can be considered *white box* tests on some level.)

For an example of a more high-level acceptance testing tool that's geared towards a single category of applications, and that operates on the system under test without knowledge of its inner workings (*black box* testing), we'll examine Selenium,<sup>4</sup> an open-source testing tool for web applications.

Essentially, Selenium follows the same style for specifying test cases as Fit: tests are defined in tables in an HTML page. Test results are also visualized in the same way as in Fit, by coloring the table rows according to their outcome.

The main difference between Selenium and a pure Fit-based approach is that Selenium is bound to the domain of web-based applications. Thus, Selenium is not a general purpose tool for acceptance testing. The actual implementation of Selenium is dramatically simple: instead of requiring a server-side framework for executing tests, the core of Selenium is based on a few HTML and JavaScript files that are placed on the server on which the application under test resides. The user-written HTML pages that contain the tests are also placed on the server, and accessing these static HTML files via a regular internet browser opens a simple user interface through which the tests can be executed. The test execution takes place entirely within the user's browser, with the tests requesting different pages from the originating server.

Due to the fact that Selenium can only be used for testing web-based applications, it comes with a set of pre-defined *commands* that are used to control the execution of the tests (e.g. navigation, data input) and to verify the test results with various

---

<sup>4</sup><http://www.openqa.org/selenium/>

assertions (such as `assertLocation`, `verifyTextPresent` etc.) For an example of a simple test case with a couple of commands, see figure 3. In this example, we utilize four different built-in Selenium commands for opening a URL, typing some text into a form field, clicking the submit button and verifying that the resulting page contains a certain string. The example illustrates how using a domain-specific tool such as Selenium eliminates the need for custom-written fixtures: the mapping between the commands contained in a test case and the web application is fixed. In a way, the set of commands provided by Selenium can be seen as a *domain-specific language* for accessing web applications.

```
MyTest
open      /mypage
type      nameField      John Smith
click     submitButton   True
verifyText name          John Smith
```

Figure 3: A Selenium test case (source: Selenium reference documentation)

Currently, Selenium supports all the major internet browsers on all major operating systems. While being able to run the tests with a click of a button using one's own web browser at any point during the development is beneficial, Selenium's automation goes a bit further. The Selenium *Remote Control*<sup>5</sup> tool provides a way of executing tests written in Selenium without manually opening a browser and running the tests. Instead, the tool enables the tests to be run from a programming language such as Java or Ruby without user intervention but still running inside of an actual browser, thus making Selenium tests candidates for inclusion in a continuous integration cycle.

## 4 Conclusion

As we've seen through this paper, acceptance testing, while often discussed in the context of Extreme Programming and other agile process models, certainly has its uses in any software process model and project. Repeatable, automated acceptance

---

<sup>5</sup><http://www.openqa.org/selenium-rc/>

tests provide tangible results of the application's status and functionality to the customer.

In chapter 2 we saw that acceptance testing fits nicely within the XP process model and its other practices: writing acceptance tests can improve communication between the team and the customers and the tests themselves can serve as a replacement for requirements documents with the added benefit that the tests stay up-to-date with the system more easily. While the on-site customer role of XP helps the customer get the maximum value from the project by specifying, together with the development team, what is expected of the system through user stories and acceptance tests, a dedicated tester should also be brought into the team.

Finally, we presented two lightweight acceptance test frameworks, Fit and Selenium, which can be used to automate the process of acceptance testing in a simple yet powerful way.

## References

- BA04 Beck, K. and Andres, C., *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.
- CHW01 Crispin, L., House, T. and Wade, C., The Need for Speed: Automating Acceptance Testing in an Extreme Programming Environment. *Second International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pages 96–104.
- Fit Fit documentation (accessed october 22nd, 2006). URL <http://fit.c2.com/wiki.cgi?FitDocumentation>.
- Fitnessse Fitnessse user's guide (accessed october 22nd, 2006). URL <http://www.fitnessse.org/FitNesse.UserGuide>.
- MC01 Miller, R. W. and Collins, C. T., Acceptance testing. *XP Universe*, 2001, URL <http://www.xpuniverse.com/2001/pdfs/Testing05.pdf>.

- Rog04 Rogers, R. O., Acceptance testing vs. unit testing: A developer's perspective. *XP/Agile Universe*, 2004, pages 22–31.
- Selenium Selenium reference documentation (accessed october 22nd, 2006). URL <http://www.openqa.org/selenium-core/documentation.html>.
- WP-FIT Wikipedia article on fit (accessed october 25th, 2006). URL <http://en.wikipedia.org/wiki/FrameworkForIntegratedTest>.