XUnit Testing – A plea for assertEquals

Tim Mackinnon
Connextra Ltd.
Studio 312, Highgate Studios
53-79 Highgate Road
London, England, NW5 1TL
+44 (0)20 7692 9898
tim.mackinnon@pobox.com

ABSTRACT
There are too many examples of bad unit tests in the XP literature. Many claim to use XUnit but in reality they have misunderstood the features that it provides and so write bad tests. By properly using features like assertEquals, developers write better tests that fail cleanly and provide enough information for others to identify a failure and rapidly fix it.

Keywords
XP, XUnit, JUnit, assertEquals

1 Introduction

One of the core practices of eXtreme Programming[2] is Unit Testing. In XP, unit tests are written by pairs of developers to help understand a problem and identify code that needs to be written to solve it (i.e. test first design). The tests also help to document the production code as well as identifying failures when they are executed. XUnit[6] is a term that represents the language independent set of testing frameworks that are used to structure and execute these unit tests. The XUnit frameworks provide a consistent and useful set of test objects that help achieve these goals. Unfortunately, although XUnit frameworks have been around for a long time, developers still continue to use them inefficiently and publish articles that do not give others a good example of best practice.

2 Writing the first test

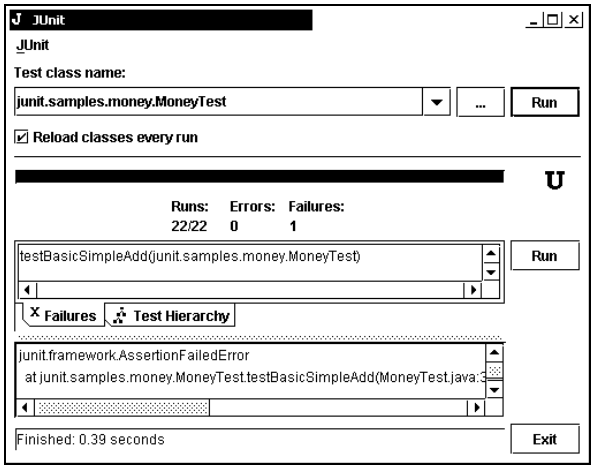
There are many variations of XUnit but the original incarnation was SUnit[3] for Smalltalk, which was later followed by JUnit for Java[4]. JUnit was popularized by the Java Report article, "Test Infected - Programmers Love Writing Tests"[5] and is noted for making unit testing popular in the programming community at large. In this article the basic building blocks of unit testing are introduced, namely the concept of a test fixture which looks something like:

```
public void testSimpleAdd() {
  Money m12CHF= new Money(12, "CHF"); // (1)
  Money m14CHF= new Money(14, "CHF");
  Money expected= new Money(126, "CHF");
  IMoney result= m12CHF.add(m14CHF);  // (2)
  // (3)
  Assert.assertTrue(expected.equals(result));
 }
```

As indicated in the article, the code fragment shows three important parts of the test fixture, Code which:

1. creates objects to interact with in the fixture
2. exercises the objects in the fixture.
3. verifies the result.

The last item is particularly important, as when combined with a TestRunner object it gives a visual indication of the problem in a window as shown below:



Importantly, this example is just an introduction to the features provided by JUnit, however many developers don't seem to notice this. The article goes on to further describe some of the richer functionality of the framework, of which the method Assert.assertEquals(…) is probably the most important. Unfortunately it is simply not used enough in test fixtures and published examples of them.

3 Introducing Assert-Equals

While it is useful to have a test indicate that it has failed, it is even more useful to have it also indicate why it failed, as well as giving the values that it failed with. This may seem like a small point, but having worked on an XP team for 3 years and having encountered many types of tests – we have consistently noted that the tests that produce an informative output are the tests that are the simplest to fix, or the quickest to show an error that was recently introduced.

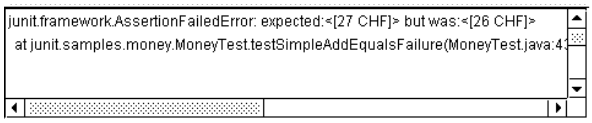
It is for this reason, that JUnit provides an assertEquals test method. This method requires that you give an expected value and an actual value, which are then com-

pared for equality. If a failure occurs, the method will fail showing both values so that you can visually see the difference. This visual indicator gives you the clue to understanding what went wrong.

In the previous example, replacing //3 with:

```
Assert.assertEquals(expected,result);
```

Gives a much clearer result as shown below:



4 Indicating the Intent

While showing both the expected and actual value is a good step in writing a great test, there is another obvious feature that JUnit provides that is often overlooked, namely the "message" parameter. It may seem like overkill adding an additional comment to your test, however our experience shows that what is obvious to you and your partner when you wrote the test, may not be so obvious to a pair that later sees the failure when they have refactored some code and inadvertently broken that test. In fact, the message parameter is more than a comment – on failure it gives a just in time indication of what the original developers were thinking, before you even have to look at the test fixture.

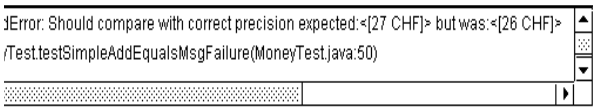
While the idea is simple, this message parameter has a subtle twist that often stalls its usage. The message displays itself on a failure, whereas the test is written from a successful point of view. In our experience a useful starting point to overcome this reversal, is to start the message with the text "Should <some verb>", where "some verb" is an action like get, see, calculate etc. This makes the test read in an English like manner, and helps make it feel more like documentation. Thus:

```
Assert.assertEquals(expected,result);
```

Can be improved by writing something like:

```
Assert.assertEquals("Should compare with correct
precision", expected,result);
```

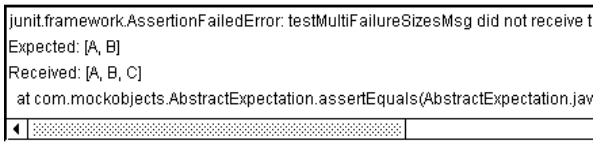
The figure below shows this improvement:



The message parameter is also useful when a test method has several asserts in it. If any one of them fails, it gives an indication of which assert caused the problem (an obvious point, but one often overlooked). Finally, it is important to note that the message parameter should not be used as an excuse for covering up a badly named test.

5 Word wrapped failures

While a simple message failure is great for simple tests, in more complicated multi-valued tests (i.e. Collections) – it becomes important to have multi line failure messages that allow you to visually compare the results next to each other. Again this is a simple point, but it is again one that often gets overlooked in some of the XUnit implementations (e.g. VBUnit). The JUnit UI allows for this possibility making it even more obvious why the test has failed:



6 The Missing Assertions

Finally, while assertEquals is an important method in the Assert utility class, there are a few useful assertions that JUnit has left out. Following the same principal that a test should clearly indicate its intent and fail with a useful message, we have found the following assertions are particularly handy:

```
assertExcludes(msg, excludeString, target-
String);
assertIncludes(msg, includeString, target-
String);
assertStartsWith(msg, startString, target-
String);
assertEndsWith(msg, startString, targetString);
```

And the strangely missing:

```
assertNotEquals(msg, expected, actual);
```

The MockObjects[5] library provides these additions, however they really should be consolidated back into JUnit and its derivatives.

REFERENCES

1. Erich Gamma, Kent Beck, "Test Infected - Programmers Love Writing Tests" online at http://junit.sourceforge.net/doc/testinfected/testing.htm
2. Kent Beck. *Extreme programming explained:embrace change.* Reading Mass.: Addison-Wesley, 1999.
3. Kent Beck, SUnit paper online at http://www.xprogramming.com/testfram.htm
4. Junit.org website, online at http://www.junit.org
5. MockObjects website, online at http://www.mockobjects.com
6. XUnit online at http://www.xprogramming.com/software.htm