**COMP33711: Agile Software Engineering**

# What's in a Test? A Brief Introduction

Pre-Reading for Agile Testing #1 Session (Week 10)

Suzanne M. Embury
September 2012

In the sessions in weeks 10 and 11, we will be looking at the topic of testing, and how the role of testing in software development is transformed by the application of agile principles. In order to get what you need out of these sessions, it is important that you have a clear idea of what is meant by a "test", and in particular what information is needed to make up a useful test case. This short document is intended to help you get up to speed with these concepts.

You will hopefully remember from your second year software engineering that an individual test aimed at revealing the presence of a particular kind of error is called a "test case", and that test cases tend to be grouped into collections, called "test suites", that can be run as a whole. You will hopefully also remember that test cases can focus on a number of different aspects of the software (e.g. functional vs non-functional) and at different levels of granularity. Thus, we have:

- Unit tests, which test single elements of behaviour, as implemented by individual units of software. Typically, we would unit test the behaviour of individual methods or classes, although the notion of a "unit" is intentionally more vague than this, to allow it to encompass elements in a variety of different programming approaches (e.g. a Web page, a sub-model, a database query, …).
- Integration tests, which verify the behaviour of several components (units) working together to solve a problem. For example, we might combine a component which knows about sales with a geographical component, to produce a new component that can tell managers how many of a particular product have been sold within a particular radius of a given point. Individually, each component might work well, but when put together like this, additional errors can be revealed. So, we need to test the integration, even if we have thoroughly unit tested the individual components.
- System tests, which verify the behaviour of the system as a whole. These are the ultimate kind of integration test, because they test the integration of all components. This kind of test is also sometimes called a functional test.

All these different types of test have the same basic form. For example, all test cases result in one of two outcomes: either the test case passes, indicating that the software implements the behaviour described by the test case; or it fails, indicating that the behaviour described by the test case is not correctly implemented. It is not possible for a test case to "partly pass" or "nearly pass". We want the outcome to be clear and unambiguous.

Test cases at all levels can be either manual (a human sits in front of the software and performs the operations described by the test case document) or automated (the test case is a program that describes how the test should be executed and, most importantly, the outcome discovered).

We also place some standard requirements on all these kinds of test, which further govern the form they take. A good quality test suite should:

- Contain only *independent* test cases. An independent test case is one that produces the same result, regardless of the context in which it is executed. Most importantly, this property means that we can run the tests in the test suite in any order, without affecting the outcomes of the individual test cases.
- Contain only *repeatable* test cases. A repeatable test case is one that gives the same outcome every time it is run. This is most important when tests are defined for manual execution. It is essential that when test T is run by Fred on Monday, we get the same outcome (pass or fail), as when the test was run by Jane on the same code base on Tuesday. Only changes to the code base should cause changes to test case outcomes.
- Contain no *redundant* test cases. A test case is redundant if it tests exactly the same behaviour as is tested by another test case in the suite. Since running tests is expensive (whether automated or manual), we can't afford to waste any of our scarce testing resources testing the same behaviour more than once.

So, what does all this mean for people whose job it is to write test cases? It means that a test case (whether automated or manual) must contain the following elements:

- Some means of automatically identifying the test case. When we need to report that a test case has failed, we need to be able to unambiguously communicate which test we are talking about. With automated test cases, this will usually be the full path of the method that implements the test: for instance, org.apache.tapestry5.ioc.internal.util.URLChangeTrackerTest.non_file_URLs_are_ignored() is the identifier of one of the tests for the open source Apache Tapestry project. Manual tests need some organisation-wide identification scheme to be designed to provide identifiers.

- A description of how to get the software into the necessary starting state for the test. For example, suppose we wish to test a simple address book manager, with a capacity of 20 addresses. To test that attempts to add a 21st address are rejected, we must first set up an address book with 20 entries in it, and load it up in the address book manager This is the required starting state for the test. To give another example from the same application, suppose we wish to unit test the component that checks phone numbers for validity, relative to the user's locale. In this case, it is necessary to create a Locale object for a user in memory, before the validPhoneNumber(String number) method on that class can be executed and tested. The data/objects that need to be set up prior to the execution of a test are often called the *fixture* for the test.

- A description of the steps required to execute the test itself. These will either be written instructions to a human tester in the case of manual tests, or a set of program statements in the case of automated tests. For example, a manual test for the "add contact" functionality of the address book manager would state something like:

  1. Enter the string 'Alfie' in the Name field
  2. Press the 'New Contact' button.

  An automated test of the phone number validation code would execute the test as:

  ```
  boolean result = locale.validPhoneNumber("0161 333 4444");
  ```

  Note that in both cases all input values are specified as exact values. We do not say "enter a name into the Name field" but "Enter the string 'Alfie' in the Name field". This is to ensure repeatability of test cases. If ambiguous phrases like "enter a name" are used, then running a test case twice on the same code base can result in two different outcomes, because of the different names chosen by different human testers, meaning that the test case is not repeatable.

- A statement of the result expected from the test, if the system under test correctly implements the behaviour being tested for. Again, this should be a statement of the actual values that should result, when the test is executed relative to the actual values specified for the inputs. When testing a software unit that returns a result, this is typically straight-forward. For example, we expect the result true from the test boolean result = locale.validPhoneNumber("0161 333 4444"); with a UK locale; and false from the test boolean result = locale.validPhoneNumber("abc");

  However, some software units do not return any value that can be checked to verify correct behaviour. In these cases, we must examine the way the unit changes the state of the system as a whole, and describe as our expected result those changes. The "Add Contact" functionality may not return any result, for example, so we must instead describe the expected result in terms of the changes made to the contacts list itself. (It should contain all the entries present before the execution of the test case, with the addition of an empty entry with the name 'Alfie', in the correct position if contacts are ordered by name.)

- A description of the steps needed to remove any of the effects of the test execution that may survive beyond the end of the test itself. This (along with the description of the fixture of the test case) is to ensure independence of test cases. Objects created in memory will be destroyed when the thread in which the test case is executed ends. However, changes to the file store, messages sent to other systems, updates to the database, and other side effects, must be explicitly undone at the end of each test case.