COMP33711: Agile Software Engineering

A Short Introduction to JUnit

Pre-Reading for Agile Testing #2 Session (Week 11)

Suzanne M. Embury
September 2012

"Never in the field of software development have so many owed so much to so few lines of code" - Martin Fowler on JUnit

Introduction

In the second of our two sessions on testing in agile projects, we will be looking at how automated tests written using tools like JUnit can support agile approaches to design and implementation that are quite different from the way these tasks are approached in conventional development processes. We'll be spending the time in the lab writing JUnit tests pretty intensively, so it is important that you are familiar with the basic concepts behind JUnit before you come into the session. This short document will remind you of what you need to know, building on your brief exposure to JUnit in your second year software engineering project.

You may also be asked to read and write (roughly correct) JUnit test cases in the exam for this course unit, this document doubles as a revision aid.

If all that isn't motivation enough for getting up to speed with JUnit, here is the indeed.com job trend graph for JUnit jobs up to September 2011. To get the up-to-date version of this graph, go to: www.indeed.com/jobtrends?q=junit

Or, for UK-specific information on salaries/demand for JUnit skills at the moment, you can check out: http://www.itjobswatch.co.uk/jobs/uk/junit.do

Before you continue with this introduction to JUnit, please remind yourself of the main components of a test case, from *What's in a Test: a Brief Introduction*, the pre-reading for the Agile Testing #1 session in week 10.

Basic JUnit Pattern: Testing the Result of a Method

Suppose we are writing software to support the running of student societies. When implementing the Member class, we might wish to automate the following test:

> After Fred Smith has joined the Team Tricycling society and paid the membership fees in full, the amount he owes to the society is £0.

In JUnit, this test is expressed as follows:

```
1  package comp33711.societymanager.tests;
2
3  import static org.junit.Assert.*;
4  import org.junit.Test;
5
6  import comp33711.societymanager.Member;
7  import comp33711.societymanager.Society;
8
```

```
9   public class SocietyTest {
10
11      @Test
12      public void testNewMemberPaysFullFees() throws Exception {
13          // Set up the fixture
14          Society society = new Society("TTSoc");
15          Member member = society.join("12345", "Fred Smith", society.fees());
16
17          // Create input values for test
18          //     No input parameters needed for this test
19
20          // Set up expected result of test
21          int expectedAmountOwing = 0;
22
23          // Execute the test
24          int actualAmountOwing = member.amountOwing();
25
26          // Check the result: pass or fail?
27          assertEquals("unexpected amount owed", expectedAmountOwing,
28                                                 actualAmountOwing);
29      }
30  }
```

There are a number of points to notice about this automated unit test. All JUnit test cases are expressed as individual methods on a normal class. It is the convention to use the suffix Test for the names of classes which contain unit tests, so the SocietyTest class contains unit test cases for the Society class. But this is just a convention; the class can be called anything you like. It is good practice, though, to write test classes that only contain test cases, and auxiliary methods to support the test cases. Don't mix up your test code and your production code in the same class! It's important to keep these two kinds of code clearly separated.

This class contains one method, and therefore one test case. JUnit test case methods are always public, have a `void` return type and (if necessary) throw `Exception`. In earlier versions of JUnit, test case methods were recognised by having names that begin with the word “test”. In JUnit 4, test case methods are recognised by the `@Test` annotation (see line 11).

The body of the method describes how to carry out the automated test and, most importantly, how to tell whether the test case has passed or failed. To help you get the hang of this approach to testing, I've separated out the main components of the test case (see pre-reading for Agile Testing #1), and indicated the role of each component in the comments. The method is coded in ordinary Java, until we get to the final part: checking whether the test has passed or failed, on lines 27/28. A key idea underlying JUnit is that the programmer needs detailed information about test cases that fail, but test cases that pass should do so silently, without bothering the programmer. So, JUnit provides a family of methods for determining the outcome of a test. Here, we use the `assertEquals()` method, which takes 3 parameters: an (optional) string for reporting failure to the programmer, the expected result value (that is, the value that should be returned by the software if it correctly implements its specification) and the value that the software actually returns. If the actual value is equal to the expected one, then `assertEquals()` succeeds silently and the test method ends. But, if there is a discrepancy, `assertEquals()` throws an exception to indicate that the test case has failed, and the error message is reported to the programmer.

In practice, we wouldn't bother to separate out all the components like this. An experienced JUnit tester would instead just write:

```
1       @Test
2       public void testNewMemberPaysFullFees() throws Exception {
3           Society society = new Society("TTSoc");
4           Member member = society.join("12345", "Fred Smith", society.fees());
5           assertEquals("unexpected amount owed", 0, member.amountOwing());
6       }
```

This automated test does not require much more typing than the English description we started with, expresses the same idea unambiguously and can be executed very quickly, as often as we need.

Basic JUnit Pattern: Testing a Change to State

The test case we have just written illustrates a very common unit testing pattern, where we wish to test that a unit of code (in this case, a method) returns the expected result. However, a lot of software behaviour is not about computing a return result but about changing the state of the system under test. In this case, we use a variant on the above pattern. Consider the following example test:

> Once Fred Smith successfully joins the Team Tricycling Society, he should be recognised by the society as a member.

In JUnit, this test case is expressed with the following method (defined on the `SocietyTest` class as before):

```
1     @Test
2     public void testNewMemberRecognisedAfterJoiningSociety() throws Exception {
3         // Set up the Fixture
4         Society society = new Society("TTSoc");
5
6         // Set up the input values for the test
7         String regNum = "12345";
8         String name = "Fred Smith";
9
10        // Execute the test
11        society.join(regNum, name, society.membershipFees());
12
13        // Check the results - pass or fail?
14        assertTrue("member not recognised by society",
15                                        society.hasMemberWithRegNum(regNum));
16    }
```

Here, we create the fixture and set up the input values for the test execution as before. We execute the method under test (in this case, `Society.join()`) throwing away its result value. We need to test that the internal state of our society instance has been updated correctly, to account for the new member. We cannot access the state directly from our JUnit test case (that would require a violation of the encapsulation of the class under test), so instead we test that other methods that can access that state return the expected results. In this case, we test that the join operation has been executed correctly by asking the society whether it recognises the registration number of the person who has just joined as that of a current member. The `Society.hasMemberWithRegNum()` method already exists and has this behaviour, so this is what we use in the assertion. In this case, we wish to check that the method returns the expected value of `true`, so we use the `assertTrue` JUnit method.

Without the breakdown into individual test components, this test would be written as:

```
1     @Test
2     public void testNewMemberJoinsSociety() throws Exception {
3         Society society = new Society("TTSoc");
4         String regNum = "12345";
5         society.join(regNum, "Fred Smith", society.membershipFees());
6         assertTrue("member not recognised",society.hasMemberWithRegNum(regNum));
7     }
```

Of course, to fully test the behaviour of the `Society.join()` method, we'd have to write more than just this one test. But it illustrates the general test pattern for units that make state changes: execute the test to make the state change, and then use the existing interface of the object under test to check that the publicly visible behaviour of the changed state is what we would expect of a correct implementation.

Basic JUnit Pattern: Testing that Exceptions are Thrown When Expected

A third kind of behaviour that commonly needs to be tested is the throwing of exceptions. How can we write automated test cases that verify that exceptions are thrown in the expected circumstances, and only in those circumstances. For example, suppose we wish to automate the following test:

> If Fred Smith attempts to join the Team Tricycling Society when he is already a member, the AlreadyMemberException should be thrown.

In JUnit 4, we have two options for automating this test case. I'll give the longest but clearest first:

```
1     @Test
2     public void testMemberJoinsSocietyTwice() throws Exception {
3         Society society = new Society("TTSoc");
4         String regNum = "12345";
5         society.join(regNum, "Fred Smith", society.membershipFees());
6
7         try {
8            society.join(regNum, "Frederick Smith", society.membershipFees());
9            fail("AlreadyMemberException not thrown on second join attempt");
10        } catch (AlreadyMemberException ex) {
11           // Test succeeds - do nothing
12        }
13    }
```

I haven't separated out the test components here, in order to make the difference with the previous patterns stand out more clearly. Here, the fixture is set up on lines 3-5 – we create a society that has Fred Smith as a member. Next, we enter a `try-catch` block. In the body of the `try`, we execute the test (line 8). If the method does *not* throw an exception then execution of the test will proceed to line 9, and the special JUnit method `fail()` will be executed. This method forces the test case to fail, with the given text being used in the (JUnit-generated) report on the failure.

If, however, the `Society.join()` method behaves according to its specification, it will throw an exception when Fred attempts to join for a second time. We drop through to the catch block, where we do … nothing! If we reach this point (line 11) the test has passed, so all we need to do is exit the test case method silently.

Note that if `Society.join()` throws an exception other than `AlreadyMemberException`, the test case as a whole will exit abnormally with an error. This is why it is important to catch the specific exception being tested for in the test case, and not to be lazy and just catch `Exception` every time.

The shorter way of automating this test is to use the `expected` parameter of the `@Test` annotation to tell JUnit which exception you expect the test case to throw. This is illustrated below.

```
1     @Test(expected = AlreadyMemberException.class)
2     public void testMemberJoinsSocietyTwiceAnnotation() throws Exception {
3         Society society = new Society("TTSoc");
4         String regNum = "12345";
5         society.join(regNum, "Fred Smith", society.fees());
6         society.join(regNum, "Frederick Smith", society.fees());
7     }
```

Here, the test fails if the method exits normally, without having thrown this named exception, or if any other exception is thrown. Personally, I prefer the longer form for general use, as it allows you to place the try-catch block precisely around the code under test, rather than placing it around the test as whole. In this case, for example, if the code setting up the fixture triggers this exception, I want that to be shown as an error in the test and to be flagged to the programmer. With this form of the test, that will not happen. But, this

shorter form is certainly much quicker to write and works just as well as the longer form when test case bodies are simple and contain no fixture code.

Shared Fixtures

Speaking of fixtures, if you look at the suite of automated test cases we have created for SocietyTest so far, you'll notice that there is quite a lot of duplicated fixture code. Most of the test cases need the same basic fixture: a Society object. JUnit provides a facility for us to pull this repeated code into a single shared "set up" method. To illustrate how this works, look at our complete set of test cases for SocietyTest, implemented with the shared fixture made explicit.

```
1   public class SocietyTestWithFixture {
2
3      String regNum = "12345";
4      String name = "Fred Smith";
5      Society society;
6
7      @Before
8      public void setUpSociety() throws Exception {
9          society = new Society("TTSoc");
10     }
11
12     @Test
13     public void testNewMemberPaysFullFees() throws Exception {
14         Member member = society.join(regNum, name, society.fees());
15         assertEquals("unexpected amount owed", 0, member.amountOwing());
16     }
17
18     @Test
19     public void testNewMemberJoinsSociety() throws Exception {
20         society.join(regNum, name, society.fees());
21         assertTrue("member not recognised",society.hasMemberWithRegNum(regNum));
22     }
23
24     @Test
25     public void testMemberJoinsSocietyTwice() throws Exception {
26         society.join(regNum, name, society.fees());
27         try {
28            society.join(regNum, "Frederick Smith", society.fees());
29            fail("AlreadyMemberException not thrown");
30         } catch (AlreadyMemberException ex) {
31            // Test succeeds - do nothing
32         }
33     }
34  }
```

We have added some fields to the test class, for storing the shared fixture object (`society`) and some scalar values that are used frequently across the tests ("12345" and "Fred Smith"). We have also added a new method, called `setUpSociety`, that has a different annotation from our standard test cases. This is the method that creates the shared fixture. The annotation `@Before` indicates to JUnit that this method should be run before each of the test cases is executed.

The test cases themselves are as before, except that the shared fixture code has been removed, as it is now implemented by the `setUpSociety` method.

JUnit also provides an `@After` annotation, which indicates that the method so annotated should be executed after each test case. It is useful for tidying up any side-effects of test cases (e.g. any files written to, database

records created, connections opened and not closed). Recall that we want all our test cases to be independent: that is, it should be possible to execute them in any order, not just the order in which they are written in the file. If side-effecting test cases are to be independent, we need to write `@After` methods to remove their effects before the next test case is run.

In fact, JUnit provides several more annotations, but we will mention only two more in this introduction: `@BeforeClass` and `@AfterClass`. These annotations are similar to `@Before` and `@After`, except that methods with this annotation are run only once, before all the test cases are executed in the case of `@BeforeClass` and after all the test cases are executed in the case of `@AfterClass`. They are useful for performing set-up/teardown operations that are expensive and only need to be performed once for all test cases. A classic example is setting up/closing a database connection, when testing code that accesses a database.

JUnit 4 Quick Reference

Here is a quick reminder of the main JUnit components, for you to keep handy during the lab.

Annotations: @Test, @Before, @After, @BeforeClass, @AfterClass

Assertions:
assertEqual(String failureReport, Object expectedValue, Object actualValue)
assertTrue(String failureReport, boolean actualValue)
assertFalse(String failureReport, boolean actualValue)
assertNull(String failureReport, Object actualValue)
assertNotNull(String failureReport, Object actualValue)
assertArrayEqual(String failureReport, Object[] expectedArray, Object[] actualArray)

Note: for equality assertions involving doubles, an extra parameter is needed to state the tolerance of the equality check (since testing for exact equality of doubles is problematic due to the error that is intrinsic in floating point calculation).

fail(String failureReport)