**COMP23420: Software Engineering**

# A Brief Introduction to Testing in JUnit

Team Study Session #1 (Week 2)

Suzanne M. Embury
February 2016

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

## What is Automated Testing and Why Do We Need It?

Testing a piece of software is the process of running it to determine how closely its actual behaviour matches the requirements set for it. As you learnt in the COMP16121/COMP16212 lectures and labs, this means deciding on a selection of *input values* the code will be run with, and working out in advance of running (and even writing) the code what the *expected output* should be for each selected input if the code is behaving as we wish it to. When we run the code with the selected inputs, we check the *actual output* produced by the code, and compare it to the expected output. If the actual output matches the expected output, then we say that the test *passes*. If it differs in some way, then we say that the test *fails*[1].

A failing test is evidence that the software we are building does *not* correctly implement the behaviour we require of it. It tells us that we have more coding work to do before we are done, and gives us some information about what that work is.

By contrast, we can't learn much from the fact that an individual test passes, since bugs may still exist in parts of the code not covered by the test. But if we have a comprehensive test suite, covering all the key cases, then we can start to have some confidence that we might have implemented it correctly once all the tests in the suite pass.

In COMP16121/COMP16212, you were asked to test your software manually. That is, you (a human) operated the software directly, entered the carefully chosen input values and painstakingly observed the result you got, to check whether it was what you were expecting or not. In the early days of software engineering, all testing was done like this (i.e., manually, by a human operating the software directly and eyeballing the results). Humans are flexible and creative, but they are also slow and unreliable. But thorough manual testing requires a lot of effort and is very boring and repetitive. It is easy for a human tester to miss out key cases, to mistype a selected input or to misread an output.

Computers, on the other hand, are excellent at repeating the same action over and over again, and they can do this very quickly and with perfect reliability. In theory, they ought to be much better at systematic testing than humans, and in fact this turns out to be largely (though not completely) the case. You have experienced some of the benefits of this approach in your COMP16121/COMP16212 work so far, through the run-test scripts you've been creating. These scripts automate the process of running the tests with your selected input values—and in doing so save

[1] Note the binary outcome here. A test either passes or fails. It is important to stick to this, and not to allow yourself to think of tests as "partially passing" or "nearly passing". These halfway house concepts aren't helpful to us in achieving high code quality.

you a lot of time. But, they are not fully automated tests, since they do not check for themselves whether the actual output matches the expected output. You still had to do this work for yourselves.

You probably noticed that this “semi-automated” approach becomes a bit of a nuisance over time, especially as the outputs from the programs you were writing became larger and more complicated. You may have found yourself quickly scanning over the output of the code, rather than checking that every letter was in the right place, or maybe even not bothering to run the tests at all, since checking the outputs was so much work.

It turns out that computers can do this part of the testing process for us too, and can do it much faster and more precisely than we could hope to (even if we could be bothered). By fully automating our testing, we get a test suite that takes a little bit more effort to set up in the first place, but which we can run many times over very cheaply. This simple idea has revolutionised the way we develop software over the course of the last two decades.

Let’s look at one way in which an automated test suite can save us time when coding. Suppose we have a comprehensive, semi-automated test suite for some code we are about to write, perhaps in the form of the “run_tests” scripts you created in first year Java. Running the script and checking the results takes a good 10 minutes of concentrated effort, and so we normally only run it a couple of times a day, sometimes only running the suite at the very end of a day of coding. One afternoon, when we run the tests, we notice that some of the tests that used to pass now fail. Something we have added to the code that day has broken functionality that we thought was working.

This is called “regression”, since the behaviour of the software system has “regressed” away from the requirements which were previously met. When we cause a regression, it should be fixed before we try to add more functionality or fix other bugs. More often than not, the cause of the regression was something we have changed recently. So, we have to look through the 50 or so new lines we added that day and the 100 lines of code we changed (and maybe all the lines of code that they interact with), to find the source of the regression, and fix it. That is going to take some time!

Imagine instead that we have a fully automated test suite that takes just seconds to run, and which works out which tests have failed for us. Instead of running this suite just once or twice each day, we run the test suite after making every small code change. Now, when we notice a newly failing test, we only have to look at the last 5 or 6 lines of code that we changed since we last ran the tests (and the related lines of code) to find the source of the problem. This reduces the scope of the debugging task, and makes bugs much cheaper and simpler to find. We also find bugs earlier, when they are easier to correct because our attention is already focussed on the area of code they are hidden in. We never end up in the situation where we have a large body of code with several (or many) bugs all mixed up together, requiring a marathon debugging session of goodness knows how long to fix.

The cost-savings from the frequent use of a comprehensive automated test suite can be significant—so much so that many organisations now make use of continuous integration and test systems, which automatically build and test each new piece of code that is checked into the version control system, reporting back to the developers if and when problems are discovered.

## Automated Testing in JUnit: a Simple Example

The most commonly used testing harness for Java code is JUnit and that is the main testing tool you will learn to use in this course unit. The principle behind JUnit is very simple: an automated test case in JUnit is merely a Java method (the test method) that invokes another Java method (the code under test) with some selected inputs, and compares the output against the expected result. If the actual output matches what is expected, then the test has passed and JUnit exits quietly. But if there is a discrepancy, then JUnit reports the test as having failed, and gives the programmer some information about the discrepancy it observed.

We'll introduce these ideas by looking at how we would use JUnit to test a very simple Java method. In Figure 1, below, you'll find the code for some JUnit tests for a method that calculates the largest square that is less than or equal to its parameter. (You may remember working on a very similar program back in the early weeks of COMP16121.) We'll go through this test class line by line.

On line 6, after some import statements to pull in the JUnit classes we need, you'll see what should be a very familiar class definition, for a class called LargestSquareTest. JUnit classes are ordinary Java classes, defined in the usual way. This JUnit class is going to contain tests for a solution class called LargestSquare, so we will call it LargestSquareTest. In fact, the class could have any legal name. But, here, we're following a common convention that JUnit test classes are named after the class they test, with the word "Test" stuck on the end. This is useful because, as a reader of code, we can see instantly which classes are test classes and which not, and also which class is being tested by which test class.

Inside the class, there are four method definitions. Each of these methods describes a separate test case for the class under test. These too are ordinary Java instance methods, of the kind you have met before, with the exception of the fact that they are each annotated with @Test. You have not encountered annotations in COMP16121/COMP212, but they are very simple to understand. They allow us to annotate code with information that is useful to the compiler and other language processors, but which will be ignored during ordinary execution. In this case, the purpose of the

```
1   import static org.hamcrest.MatcherAssert.assertThat;
2   import static org.hamcrest.core.Is.is;
3
4   import org.junit.Test;
5
6   public class LargestSquareTest {
7
8       @Test
9       public void shouldReturn0AsLargestSquareLessThanOrEqualTo0() {
10          assertThat(LargestSquare.lessThanOrEq(0), is(0));
11      }
12
13      @Test
14      public void shouldReturn1AsLargestSquareLessThanOrEqualTo1() {
15          assertThat(LargestSquare.lessThanOrEq(1), is(1));
16      }
17
18      @Test
19      public void shouldReturn1AsLargestSquareLessThanOrEqualTo3() {
20          assertThat(LargestSquare.lessThanOrEq(3), is(1));
21      }
22
23      @Test
24      public void shouldReturn4AsLargestSquareLessThanOrEqualTo4() {
25          assertThat(LargestSquare.lessThanOrEq(4), is(4));
26      }
27  }
```

Figure 1: an Example of A Simple JUnit Test Class

annotations is purely to tell the JUnit test runner which of the methods defined on the class should be considered to be a test case, and which should not.

There are a couple of other JUnit annotations that we'll encounter later in the course. For now, the important thing to note is that we must put the @Test annotation at the start of every test case method we write. If we don't, the test case described by the un-annotated method won't get executed when we run the test suite.

Now lets look at the test methods themselves. Every JUnit test method should be public (to allow the JUnit runner to call it) and should have a void return type. JUnit test methods must have no input parameters. They can be called any legal Java method name, but (just as with JUnit class names) it is usual to follow some naming conventions. Some people (for historical reasons) begin every test method with the word "test". I prefer to follow the convention of beginning the test names with the word "should", and making the name describe the behaviour that the test is testing. The idea is that the names of the tests, when viewed in isolation (as is possible in some IDEs) should read like a specification for the code under test.

You might be a bit surprised by how long these method names are. You may even be wondering if such long method names can possibly be best practice. It's true that these names would be too long and cumbersome for ordinary code. But we are not writing ordinary code here. JUnit methods are called by the JUnit runner, which uses reflection to identify the methods tagged with the @Test annotation, and then run them. No human will ever have to write code which calls these JUnit methods; no human will ever have to type these long names in. So, the only role they play is one of documentation; the method names should tell us what the intent of the test case is. That usually means writing quite a long method name, but it is also useful, as it forces us to think what the test we are going to write is for, before we get into the nitty gritty of coding it up.

Next, we can look at what is happening inside the test case methods themselves. Let's focus on the example on line 10. Here, we are calling the code under test (a static method on the LargestSquare class called lessThanOrEq()) with a specific input value (in this case, 0). Then, we are using a method provided by the test harness (assertThat()) to state what we expect the result of this to be, when the method is called with the specified input value. In this case, we expect the output to be 0.

assertThat() is a matcher method that is provided as part of the Hamcrest matching library. It comes in several forms (some of which are quite complicated), but all you need to understand at the moment is that it takes the value given as its first parameter, and matches it with the expression in its second. If the values match, then the assertion exits quietly. If there is a mismatch then the assertion flags this up as a failing test, along with some information about the exact form of the mismatch detected. The assertions in the test class in Figure 1 all use the pre-provided matcher is(value), that checks whether the provided value is equal to value or not. So, the assertThat() statement on line 10 checks that LargestSquare.lessThanOrEq(0) returns a value that is equal to 0, and lets the programmer know about it if it doesn't.

Notice how the whole test case reads quite like an English sentence describing how we want the code to behave. We want to "assert that the largest square less than or equal to 0 is 0". Well-written test cases should have this property. They should (as far as practicable) read like natural, readable statements of the behaviour we are trying to implement.

There is a lot more to learn about JUnit than the very simple tests we have described so far. But, this brief introduction should be enough to help you get started on working with tests in the workshops this week.