Tutorial 1: Simple Testing at the Code Level

Key:

The exercises in this tutorial are divided into three types: normal, EXTENSION and HARD. If it doesn't say otherwise, the exercise is normal.

- Normal exercises should be completed by everyone. You'll need to master this material to do well in the course!
- EXTENSION exercises cover the same topics as the normal exercises, but will give you extra practice, or introduce some concepts from later lectures. Do these if you have time.
- HARD exercises cover more advanced material. Do these only if you're confident with everything else, and if you want to stretch your understanding.

The point of this tutorial is to get you writing automated tests for simple code that is quite amenable to them. (The next two tutorials will give you some uglier code to work on, where these techniques are only partly applicable)

Time budget: I'd recommended taking around two tutorials of time, plus some time for private study. If you've not made much progress by the end of that, consider spending some more – this is the most important tutorial in the course as it covers the basic skills you'll need for assignments and the tests.

Exercise 0 – Getting Eclipse installed

Take a bit of time and get Eclipse installed on your laptop or desktop. You'll need to make sure JUnit is installed, and later you will need EclEmma installed too.

Exercise 1 – Using JUnit

New Challenge: use JUnit to create simple tests

If you've not used JUnit before, read the JUnit refresher slides available on Avenue and work through the examples and sample exercise. There are numerous examples of JUnit available on the web that you can also use.

You may also find the tutorial at http://www.vogella.com/tutorials/JUnit/article.html useful.

Exercise 2 – Finding and Capturing Faults in Existing Code

New Challenges: find faults in some existing code that you didn't write yourself (given a fairly clear specification of correct behaviour);

Download the "Library System" code from Avenue. This is the (unfinished) code for a library book management system. The code is just that, code – some interacting classes with no UI of any kind. If completed, it could be used as the backend of a GUI application for use by library desk staff. The code offers a range of methods, most of which are fairly well described by their Javadoc. (If you find that something *isn't* well described, let Richard know – figuring what the main public methods are meant to do is *not* meant to be part of this exercise!)

The code has faults, of several kinds and in several places.

Your task is as follows:

- Find as many faults as you can
- Write JUnit test cases that capture those faults (i.e. that fail because of those faults)

I would suggest that you do the above in parallel – when you find a fault, write a test case to capture it. Indeed, one way to find faults is to write JUnit test cases that check the expected behaviour for a range of inputs.

To help guide you, here are some types of fault that definitely exist in the code:

- Input validation code that accepts some inputs it shouldn't
- A string formatting error that has a significant consequence (i.e. it does more than lead to a bad string being printed)
- A sequence of actions that can lead to a system state that doesn't make sense
- An exception is thrown when it shouldn't be
- An exception is not thrown when it should be
- An exception is thrown when it should be, but it's of the wrong type
- A fault that will lead to failure only when a new derived class is implemented that has a certain type of behaviour
- A performance issue where an overridden method meets the letter of its contract but not the spirit of its purpose (you may struggle to create an actual test for this, but you should at least think about how it could be done)

The above is not an exhaustive list, and there may be more than one instance of each type.

In the process of doing the above, you will no doubt generate a number of test cases that pass; tests that do not reveal any faults. That's fine — keep them around, as they will help you meet the coverage goals set out in the next exercise.

Hint —Start by focussing on the Library class. It uses all the other classes, so can potentially reveal faults anywhere in the code.

Exercise 3 — Measuring code coverage with EclEmma

1. Quickly read the EclEmma user guide at http://www.eclemma.org/userdoc/index.html
2. Then, run EclEmma on your test code. You can do this from Eclipse by right-clicking on a JUnit test class and selecting "Coverage as -> JUnit Test Case" from the context menu that appears.

   Once you've done this, summary coverage information will appear in a "Coverage" pane (in the same part of the display that the "Console" pane appears) – more detailed information can be found by right-clicking on a class in the Package Explorer, selecting Properties, and going to the "Coverage" property page.
3. Have a look at how well your test code has covered the LibrarySystem classes - the classes in the SMATLibrarySystem package (don't worry about how well the test code itself has been covered). If the coverage is less than 100% for any of line, branch or method, see if you can improve your test suite to get to 100%. At least do this for the Library class.

Beware of some limitations of EclEmma which may cause it to report less coverage than you actually achieved — see http://www.eclemma.org/faq.html, specifically the heading *"Code with exceptions shows no coverage. Why?"* You may need to manually confirm that some elements were in fact covered. The Library class does not have this problem, however – you can get EclEmma to report 100% coverage.

Exercise 4 – Test Some of Your Own Code (EXTENSION)

Take some code you've written in the past, and write some JUnit test cases for it. Start by writing tests for faults you know are there, then have a go at hunting for some new faults.

If the code you want to test isn't in Java, have a look for a JUnit equivalent for that language – most languages have something.