INB370 / INN370, *Software Development*, Semester 1, 2010

Practical 3 — Automated Unit Testing

This week's practical exercises aim to get you familiar with designing and implementing automated unit tests using the JUnit plug-in to the Eclipse integrated development environment.

Setting up

For this practical session you can use the JUnit implementation provided with Eclipse, or a downloaded copy of the JUnit Java archive. If the former, you may be asked which version of JUnit to add to your path when you first try to create a JUnit Test Class ⊑, in which case you should select the most recent one. If you decide to use a downloaded version you will need to add the JUnit '`.jar`' file to your project's build path by adding it as an 'external archive'.

Group exercise: Test case brainstorming (Totaliser example)

This exercise will give you practice at devising black-box test cases for a simple class.

In the materials supplied for this practical session you will find API documentation for three classes, including one called `Totaliser`. Read the documentation for the `Totaliser` class and then, as a group, identify a small, but sufficient, set of test cases that will exercise the `Totaliser` implementation's functionality. Your tutor will write the test cases selected on the whiteboard.

Individual exercise: Testing with JUnit (Totaliser example)

This exercise will give you practice at using JUnit to test a simple class.

In the materials supplied for this practical session you will find an implementation of the `Totaliser` class. In Eclipse create a new package and install the `Totaliser.java` file *but do not edit or examine it!* The purpose of the exercise is to test the supplied program as a black box, i.e., without looking at the source code.

Now create a JUnit Test Class called `TotaliserTest.java` and implement the various tests identified in the previous exercise. Run your tests and see if you can identify any problems with the supplied program.

Hint: There are (at least) two errors in the code, one obvious and one a bit harder to detect.

When you think your tests have identified the problems with the program examine the `Totaliser.java` source code file to confirm your suspicions. Then correct the errors in the code and rerun your tests to ensure that the problems have been fixed.

Group exercise: Test case brainstorming (AllSame example)

This exercise will give you practice at devising black-box test cases for a more complicated, but stateless, class.

Read the API documentation for the `AllSame` class and, as a group, identify a sufficient set of tests for it. Note that the method provided by this class is a pure function, i.e., it does not update any fields within the class and thus has no 'memory' from one invocation to the next, which simplifies testing.

Individual exercise: Testing with JUnit (AllSame example)

This exercise will give you practice at using JUnit to test a complicated, stateless class.

In the materials supplied for this practical session you will find three different implementations of the `AllSame` class, in the files `AllSameA.java`, `AllSameB.java` and `AllSameC.java`. Using the test cases identified in the previous exercise, your challenge is to identify which, if any, of these implementations is correct, and what the problems with the incorrect implementations are. Again, you are required to do so *without examining the source code*. Look at the code only when you have completed the exercise, in order to confirm your findings.

Hint: In the three implementations there are (at least) three distinct errors in total.

Note: Testing three different programs with one set of test cases is an unusual situation and is a little awkward, no matter how it's done. One approach is to put all three programs under test in the same package, create your JUnit test class in that package, and manually edit which of the three programs is imported into the test class to change from one to the other. Another approach is to put each of the three programs in its own package and have a copy of the test case class in each one. (There is a much more elegant solution which works by creating a superclass for the three programs under test, but it is felt to be a little complex for this one-off exercise.)

Group exercise: Test case brainstorming (JourneyPlanner example)

This exercise will give you practice at devising black-box test cases for a complicated, stateful class.

Read the API documentation for the `JourneyPlanner` class and, as a group, identify a sufficient set of tests for it. Note that this class updates private fields and thus retains a 'memory' from one method invocation to the next, making it hard to devise test cases with good 'coverage'.
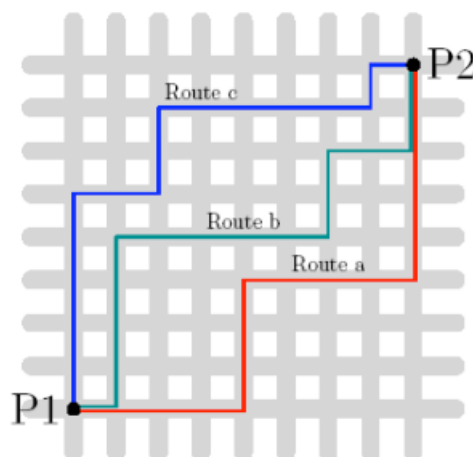
Background: The class calculates journey times based on 'Manhattan distances', i.e., the distance you must travel in a grid-like environment by following the grid lines only (with no diagonal steps). The name derives from the exceptionally regular layout of streets in New York city, which forces people to travel in this way. Notice in the diagram above that the journey from point P1 to P2 is of the same length, no matter which route is taken.

Individual exercise: Testing with JUnit (JourneyPlanner example)

This exercise will give you practice at using JUnit to test a complicated, stateful class.

In the materials supplied for this practical session you will find two different implementations of the `JourneyPlanner` class, in the files `Journey-PlannerA.java` and `JourneyPlannerB.java`. Using the test cases identified in the previous exercise, your challenge is to identify which, if any, of these implementations is correct, and what the problems are in the incorrect class or classes. Again, you are required to do so *without examining the source code*. Look at the code only when you have completed the tests, in order to confirm your findings.

Hint: In the two implementations there are (at least) two distinct errors in total.