Comp47480: JUnit Tutorial

This practical involves an introduction to Test-Driven Development (TDD). We'll be using JUnit 4.x with Eclipse. Eclipse comes with JUnit4, so there's no installation required. This JUnit With Eclipse Tutorial may be useful reading.

Test-driven development is a method of developing software, not merely a method of testing. The technique involves the following three steps being repeatedly applied:

Step 1: Write a test case and try to run it. It probably won't even compile, so write just enough code to make it compile. Now run it. It will of course fail, so you'll get a **red** bar.

Step 2: Now you implement only the code necessary to make the test pass, nothing more. This code can be as quick-and-dirty as you like, once it makes the testcases pass. You'll know it's right when you run your testcases and you see the **green** bar.

Step 3: Now you tidy up your code by **refactoring**. You don't change what the code does, but you make it easier to understand and tighten the implementation wherever necessary. Avoid commenting the code. If you feel it needs a comment, refactor it so the comment becomes unnecessary. Run your testcases at any stage to confirm that behaviour hasn't changed. Do this again when you're finished refactoring to make sure that you haven't accidentally introduced any bugs.

Remember the mantra! **red** **green** **refactor** If a demonstrator asks you, you should immediately know what state you are in.

TDD is often formulated as three simple rules:
1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Preliminaries

Ensure that you have Eclipse installed along with the JUnit plugin. Any **JUnit 4.x** will work fine -- we're just using the very basics of what JUnit provides. Also install the EclEmma coverage plug-in. Create a new workspace project by clicking File -> New -> Java Project and click Next. Type in a project name -- for example, `ProjectWithJUnit`. Click Finish. The new project will be generated in your IDE.

Part 1: A Simple Example

Using TDD, develop a class called `Weather` that has a single public method called `isFreezing` that returns `true` if the temperature is <= 0.0, and `false` otherwise. Its constructor takes the current temperature as an argument. This class is trivial to implement of course, but **apply TDD carefully and precisely** to make sure you understand the process.

The first thing you do is write a testcase that exercises the `Weather` class. To create such a testcase, right-click on the `ProjectWithJUnit` title, select New -> JUnitTestCase. (This may also prompt you to add junit.jar to the project path).

Choose a name for your **testcase** -- for example, `WeatherTest`. You can leave the "Class under test:" box empty as the class you're to test doesn't exist yet. Click on **Finish**.

A possible first testcase for the `WeatherTest` class is as follows:

```
import org.junit.*;

public class WeatherTest {
  @Test  // this annotation tells JUnit that this is a test case
  public void testIfFreezingAtZeroDegrees(){
    Weather w = new Weather(0.0);
    assertTrue("temp is zero, but weather not freezing", w.isFreezing());
  }
}
```

Study this code and make sure you understand it.

In keeping with the TDD approach, we now run our test to confirm that it fails. Highlight *TestWeather.java* in **Package Explorer**. Now click **Run -> Run as -> JUnit Test**. In the left window, instead of **Package Explorer**, you will see the **JUnit** window, which shows a red bar, the failed tests, and details of those failures.

Now implement just enough of the `Weather` class to make this testcase pass. You will probably be inclined to add too much implementation, breaking Rule 3 above. All that is required to make this testcase pass is:

```
public class Weather {
  public Weather(double initTemp){
  }
  public boolean isFreezing() {
    return true;
  }
}
```

Re-run the test and it should pass. Experiment with this, e.g., return `false` and see what happen when the test is run.

Run the EclEmma tool to check what coverage you're getting. Using TDD, it should ways be 100%. If you add untested functionality to the `Weather` class, EclEmma will report it like this:

```
public boolean isFreezing() {
  if (temp <= 0.0)
    return true;
  else
    return false;
}
```

Continue applying the TDD process to this example, until you have a complete implementation of the `Weather` class.

Reflect on what's been achieved. The implementation is very simple, so what has the use of TDD given us?

- Developing testcases first made us define the problem clearly.
- Testcases makes it clear when we were finished.
- When we change our code to add a new requirement, we can rerun the testcases to make sure that existing functionality hasn't been affected. This gives great freedom in refactoring.
- We needn't document for a maintenance programmer what the required behaviour of `isFreezing` is -- the testcase does this and is an executable test to boot.

Part 2: A More Challenging Example

BeeBop is a child's game where numbers divisible by 3 are replaced with "Bee", numbers divisible by 5 a replaced by "Bop" and numbers divisible by both are replaced with "BeeBop". Use TDD to write a method that plays BeeBop, i.e. a method called `processNumber` (in a class `BeeBop`) that takes a single integer argument and returns the appropriate string. Write your code to handle these cases in sequence:

Input (int)	Desired Output (string)
1	1
2	2
3	Bee
4	4
5	Bop
6	Bee
7	7
8	8
9	Bee
10	Bop
12	Bee
15	BeeBop
18	Bee
20	Bop
30	BeeBop

Apply TDD step-by-step taking each test case in sequence, and do a Git commit after each test case has been handled. To get you started, your application code to implement your first test case should be this:

```
public class BeeBop {
  public String processNumber(int number) {
    return "1";
  }
}
```

Of course as you extend `processNumber` more test cases, think how to improve your code (in the refactoring step of the TDD cycle), but never implement functionality you have no testcase for. The code in your final solution should be of excellent quality.

Part 3: Code Coverage

Use the EclEmma tool to determine the level of code coverage achieved by your testcases. If it's less than 100%, you definitely took shortcuts in applying TDD. Work out what went wrong.

If you achieved 100% coverage, you may have used TDD correctly, but it is no guarantee! To illustrate this, see if you can remove a testcase and still have 100% coverage. If you can do this, it means one of two things: (1) The test case was redundant and you didn't need it, or (2) The test case tested an important feature of the code, but the code was covered by other test cases. In this case a bug could be introduced into the `processNumber` method without the test cases reporting any problem, even though coverage is at 100%. Thus regression could occur, even though all testcases are passing. So you see that code coverage is never fully sufficient to prevent bugs.

Another way to assess your testcases, apart from coverage, is to use **mutation testing**. Make a small change (a mutation) to the `processNumber` method and run the testcases. One or more should fail. If they don't, there is a feature of the `processNumber` method that they are not testing. If they pass, undo the change you made and try another one.

Submission and Lab Journal entry

The artefact for this practical is your solution to Part 2. Part 1 is a warm-up exercise, and Part 3 comprises exercises you can write about in your lab journal.

Submit your lab journal entry, following the usual guidelines.
