Entry Three*:* JUnit Tutorial

Student Name: Gregory Sloggett

Student Number: 14522247

Work Done:

The first task at hand was to ensure that I had Eclipse set up correctly, with JUnit installed and the EclEmma plugin correctly setup in the IDE. This was already configured properly in Eclipse, so nothing needed to be adjusted. I then set about doing the basic weather test class as practice. I followed closely the three steps of the TDD process to ensure I was taking the right approach to the assignment. I was rusty in Java as I've spent most of this year developing in Python, so I needed to refresh my memory and this practice weather part was useful for that.

Approaching the artefact and main part of the TDD assignment, I again followed closely the TDD process. Firstly, I made the TestTriangle.java JUnit Test Case Class, and set up a basic test to ensure the Triangle had three sides given to it in the constructor. I later removed this test case as it was clearly redundant given that if the constructor takes three arguments, an error will be provoked, and the class won't compile (I did mention I was rusty in Java!).

As this provoked an error given I had no Triangle class, nor a method called hasThreeSides(), the next step was to set up such a class with such a method. Triangle.java was set up and hasThreeSides() was a simple Boolean function that just returned true, simply so that the test case would pass, as requested by the TDD process. When this basic test case was satisfied, I refactored to add the functionality to confirm whether the triangle had three sides (here was where I made the realisation that this was surplus to requirements).

From here I added another test case in TestTriangle.java, called testIfEquilateral(). This instantiated a new Triangle object with three integer arguments for the sides and asserted that the resulting string given from the method typeOf() was "Equilateral". Again, this came up as an error given that the method typeOf() had not been created yet, and thus I went about achieving this with it simply returning true. As I added the test cases for the isosceles, scalene and invalid triangles, I refactored the code in the typeOf() method, to ensure that the assertions were kept true for each triangle, until no more refactoring was required.

Reflection:

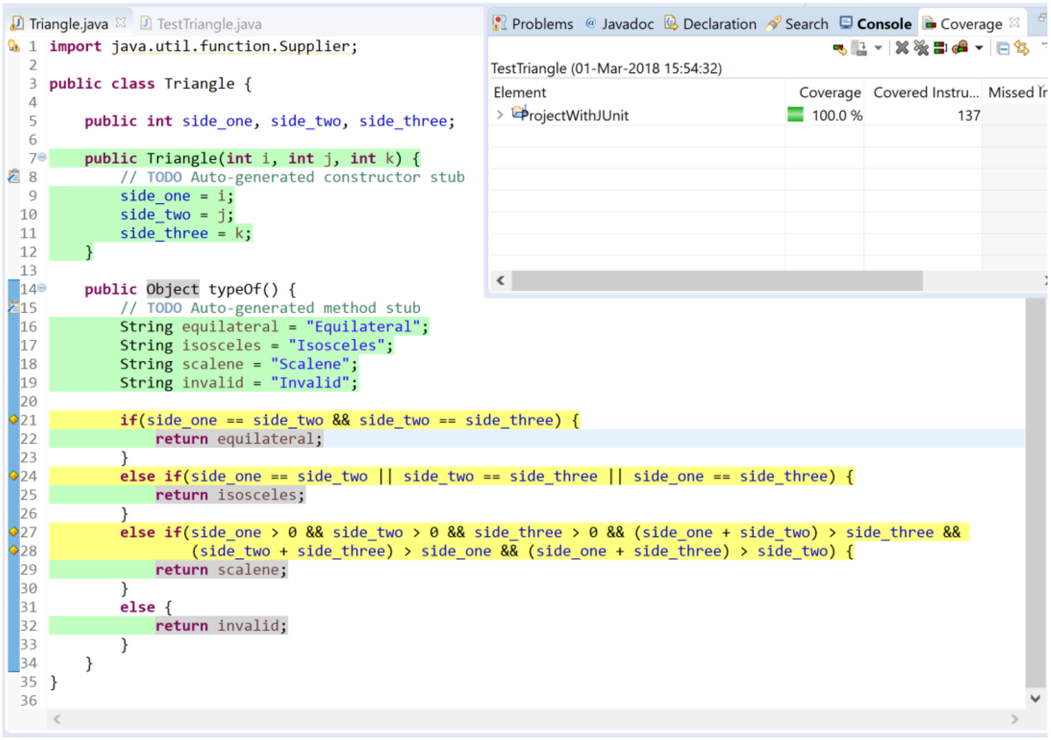
Last year, when I took Mel's module on object-oriented programming, and we developed using ruby, we touched on test-driven development and how that works, so in a sense, this assignment was a refresher of that module. To be quite honest, I don't think I really understood how effective TDD was back then and how widely used it is in the workplace. We recently had a talk with IBM and one of their Chief Architects who informed us how they expect all of their developers to provide unit testing. While I realise now that it is imperative, I don't think that I realised eighteen months ago when I took Mel's previous module just how imperative testing is. In that sense, I really took more value out of this practical than I did back then.

Admittedly, when I have performed testing in the past they've all been on relatively straightforward and simple logic, that you would process in your head far quicker than you might do producing a test case for each scenario first, so there have been times where I've questioned why it might be used. In truth, part of me felt this way during today’s practical, however, I understand that in larger systems, spanning thousands of lines of code, that has been in use over a number of years, testing has it’s place in a massive way.

I still don’t know how accurate or effective my test cases are. I got 100% coverage in this assignment according to the EclEmma plugin, but at the same time I only used four test cases and I’m not sure whether I should have included more or not. My method typeOf() in the Triangle.java class simply returns a string depending on which type of triangle it was, and the four test cases I used were simply assertions for the resulting string produced from this method and checking that it confirmed to the triangle type. Should I have provided more test cases? For example, an equilateral triangle has sides that are equal in length and in my test class, I just assert that the resulting string of the method typeOf(), for a triangle that has three equal sides, comes out as “Equilateral”. But, should I confirm in the test case that the first side equals the length of the second side, and that the second side equals the length of the third side etc.? I don’t believe that this is necessary given that if the sides were not of equal length then this assertion would provoke and error, but for some reason I’m unconvinced of that.

Stemming from what I mentioned in the last paragraph, I found that the Eclipse plugin, EclEmma, was a very useful tool to reassure yourself that you have made coverage on all of the test cases. It’s a simple tool to use and it’s very clear where there is coverage and where there is not. However, the question I posed in the last paragraph also bears fruit here in the use of EclEmma.

The following execution of EclEmma produces a few yellow warnings:



TestTriangle (01-Mar-2018 15:54:32)

Element	Coverage	Covered Instru...	Missed Ir
ProjectWithJUnit	🟩 100.0 %	137	

```java
import java.util.function.Supplier;

public class Triangle {

    public int side_one, side_two, side_three;

    public Triangle(int i, int j, int k) {
        // TODO Auto-generated constructor stub
        side_one = i;
        side_two = j;
        side_three = k;
    }

    public Object typeOf() {
        // TODO Auto-generated method stub
        String equilateral = "Equilateral";
        String isosceles = "Isosceles";
        String scalene = "Scalene";
        String invalid = "Invalid";

        if(side_one == side_two && side_two == side_three) {
            return equilateral;
        }
        else if(side_one == side_two || side_two == side_three || side_one == side_three) {
            return isosceles;
        }
        else if(side_one > 0 && side_two > 0 && side_three > 0 && (side_one + side_two) > side_three &&
                (side_two + side_three) > side_one && (side_one + side_three) > side_two) {
            return scalene;
        }
        else {
            return invalid;
        }
    }
}
```

As you can see, I’ve got 100% coverage across the classes, but I still receive the yellow warnings that tell me something like “1 of 4 branches missed”. This was a factor in my curiosity of whether I needed to add more test cases or not, but I remain with the thought that my test cases are sufficient as it stands.

I performed some mutation testing on the code produced through using the TDD process and produced errors. I changed some of the operators in the conditional statements checking the state of the triangles to determine their shape, and by doing so received errors. This seems like a tedious but resolute way to ensure how you've tested is thorough, as if an error does not occur when you mutate your code, then there is a feature of your code that you are not testing.

In conclusion, this was a relatively short and simple refresher of the Test-Driven Development process, but an informative and well needed reminder of how it works, and how effective and important it can be. I have not used this in practice in any larger projects, but I do intend to do so soon, as I see it as a core principle I should include in my code moving forward in my career.