Tutorial

1

SOFTWARE QUALITY

Ontario Tech University

TDD Techniques

CRN 74663 | SECTION 005

Date : January 29nd , 2024

Mostafa Abedi (100787034)
mostafa.abedi@ontariotechu.net

Table of Contents

Tutorial 1

Introduction

A brief description of the tutorial

This tutorial takes a look at the software development techniques and focus on the Test-Driven Development (TDD) method. TDD is defined and its core concepts of Red-Green-Refactor and its benefits are explained. Finally, in this tutorial we take a look at JUnit with Java.

Learning Objectives

Our goals for this tutorial

1. Understanding Test-Driven Development and how to use this technique

2. Gain Experience writing JUnit Tests with Java

Tasks

Task 1

We first create a test for the add method. First test checks to add two numbers; this test should pass since the expected output should be equal to the actual output. The second test checks the adding of two numbers to be wrong, this test should always fail. Figure 1.1.1.

```java
package com.wordz.domain;

import com.wordz.Sum;
import org.junit.jupiter.api.Test;

import static com.wordz.domain.Letter.*;
import static org.assertj.core.api.Assertions.assertThat;
import static org.assertj.core.api.Assertions.assertThat;
import static org.junit.jupiter.api.Assertions.assertEquals;

public class SumTest {

    @Test
    public void testAddTwoNumbers() {
        Sum calculator = new Sum();
        assertEquals(5, calculator.add(2,3));
    }

    @Test
    public void testADDTwoNumbersWrong(){
        Sum calculator = new Sum();
        assertEquals(6, calculator.add(2,3));
    }
}
```

Figure 1.1.1 – Two testcases created for the method add

The test fails initially as the method has yet to be implemented. Figure 1.1.2.

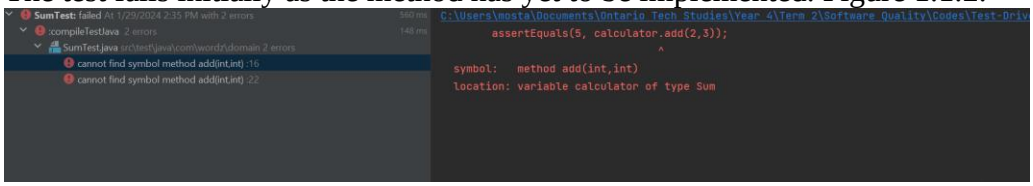

Figure 1.1.2 – The test fails as theres not “Add” method

The add method is implemented to the sum class.Figure 1.1.3.

```java
package com.wordz;

public class Sum {

    public int add(int a, int b) {
        int res = a+b;
        return res;
    }
}
```

Figure 1.1.3 – The new method after the tests

After running the test we get the results as expected with the first test passing and the second test failing. Figure 1.1.4.

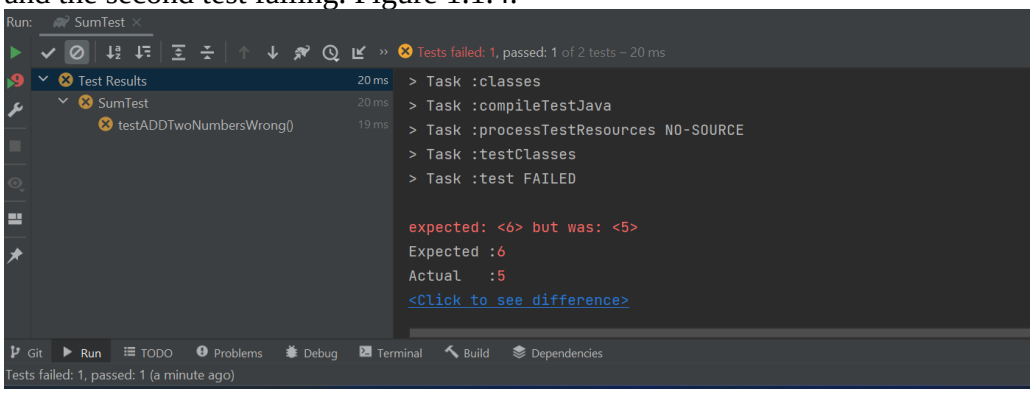

Figure 1.1.4 – The result of running the test code

Task 2

Chapter 5 goes through the wordz example. By starting with the creation of the test codes. Figure 1.2.1.

```
package com.wordz.domain;

import org.junit.jupiter.api.Test;

import static com.wordz.domain.Letter.*;
import static org.assertj.core.api.Assertions.assertThat;

public class WordTest {

    @Test
    public void oneIncorrectLetter() {
        var word = new Word( correctWord: "A");

        var score : Score = word.guess( attempt: "Z");

        assertThat( score.letter( position: 0) ).isEqualTo(Letter.INCORRECT);
    }
}
```

Figure 1.2.1 – The test code we create

We follow this by implementing every method that we need. We need to implement word, score, guess, and Letter. Figure 1.2.2 to Figure 1.2.4 show this respectively.

```java
package com.wordz.domain;

public class Word {
    public Word(String correctWord) {
        // Not Implemented
    }

    public Score guess(String attempt) {
        var score = new Score();
        return score;
    }
}
```

Figure 1.2.2 – Implementation of "word" and "Guess method

```java
package com.wordz.domain;

public class Score {
    public Letter letter(int position) { return Letter.INCORRECT; }
}
```

Figure 1.2.3 - Implementation of "Score" method

Letter here is used in the testcase to emulate an incorect response.

```java
package com.wordz.domain;

public enum Letter {
    INCORRECT
}
```

Figure 1.2.4 - Implementation of "letter" enum

Questions and Answers

Question 1

Waterfall development sounds as though it should work well – why doesn't it?

Waterfall development would work well if we knew about every missing requirement, every change request from the users, every bad design decision, and every coding error at the start of the project. But humans have limited foresight, and it is impossible to know these things in advance. So, waterfall projects never work smoothly. Expensive changes crop up at a later stage of the project – just when you don't have the time to address them.

Question 2

Can we do agile development without TDD?

Yes, although that way, we miss out on the advantages of TDD that we've covered in previous chapters. We also make our job harder. An important part of Agile development is always demonstrating the latest working code. Without TDD, we need to add a large manual test cycle into our process. This slows us down significantly.

Question 3

How do we know what test to write if we have no code to test?

We reframe this thinking. Tests help us design a small section of code upfront. We decide what interface we want for this code and then capture these decisions in the AAA steps of a unit test. We write just enough code to make the test compile, and then just enough to make the test run and fail. At this point, we have an executable specification for our code to guide us as we go on to write the production code.

Question 4

Must we stick to one test class per production class?

No, and this is a common misunderstanding when using unit tests. The goal of each test is to specify and run a behavior. This behavior will be implemented in some way using code – functions, classes, objects, library calls, and the like – but this test in no way constrains how the behavior is implemented. Some unit tests test only one function. Some have one test per public method per class. Others, like in our worked example, give rise to more than one class to satisfy the test.

Question 5

Do we always use the AAA structure?

It's a useful recommendation to start out that way but we sometimes find that we can omit or collapse a step and improve the readability of a test. We might omit the Arrange step, if we had nothing to create for, say, a static method. We may collapse the Act step into the Assert step for a simple method call to make the test more readable. We can factor our common Arrange step code into a JUnit @BeforeEach annotate method.

Question 6

Are tests throwaway code?

No. They are treated with the same importance and care as production code. The test code is kept clean just as the production code is kept clean. The readability of our test code is paramount. We must be able to skim-read a test and quickly see why it exists and what it does. The test code is not deployed in production but that does not make it any less important.

Links

Github

The code from each tutorial can be found here:

https://github.com/Mostafa-Abedi/SOFE3980.git

Conclusion

A summary of the tutorial

In conclusion, this tutorial provided a comprehensive overview of Test-Driven Development (TDD) techniques, focusing on the core concepts of Red-Green-Refactor. The tutorial began with an introduction to TDD, emphasizing its importance in software development. The learning objectives were clearly defined, including understanding TDD and gaining experience in writing JUnit tests with Java.