SENG202 – Software Engineering Project Workshop

2020

Tutorial 3 – Unit testing and Test-Driven Development (TDD)

1. Introduction

Software testing is a very important part of quality control in software development. Testing involves validating and verifying that a software application meets the business and technical requirements that guide system design and development. For a long time we considered software testing to be equivalent to what is considered to be manual testing today. Only recently (in the last 15 years or so) the idea of continuous testing throughout development has started to become more prominent.

Objectives:

- Understand good testing practices;
- Complete the unit testing and TDD quiz on the Quiz Server. The quiz will be open in the second half of the tutorial time.

2. Unit Testing Overview

This tutorial will focus on JUnit, which we expect you to use for your SENG202 project. (It should also be noted that there are *other* testing frameworks available, but we expect you to use JUnit).

What is a unit test

A unit test is a piece of code that invokes a unit of work (e.g. a class or method) in the system, and then checks if the logic of the system behaves as expected. This helps a developer to ensure that expected behaviour corresponds with *actual* behaviour.

Running tests automatically helps to identify software regressions introduced by changes in the source code.

Where should the tests be located?

Unit tests should be kept separate from the source code of your project. In the maven project we created in the first tutorial, you will find a test class named "`AppTest`". Test classes should be placed within the same directory, although they can also be in different packages within the "test" directory.

Which parts of the codebase should be unit tested?

This is a highly controversial issue, but a rule of thumb is that you should test the critical and complex parts of your application. This means that it is generally safe to ignore trivial areas of the codebase, such as getters and setters – it's safe to assume these work just fine (unless your setters/getters are doing more than just setting or getting a field value, in this case they should be unit tested).

Don't worry about testing the GUI code – unless you feel the need (Note that this is not advised due to the rapidly evolving nature of your interface). Moreover, if you find yourself writing a lot of logic in your Controller classes (e.g. input validation), consider moving this logic out (closer to the model) to allow for easier unit testing.

Measuring Testing Coverage

There are three distinct metrics for measuring code coverage: Line Coverage/Statement Coverage, Method Coverage & Class Coverage. Their names are self-explanatory, however where they are applicable, may not be.

There is no golden rule with regards to which method of measuring code coverage is better. 100% class coverage may in fact not be as good as it sounds – it is possible to achieve this by covering just one method in every class, leaving important edge cases untested.

In an ideal world, everything would be as close to 100% line coverage as possible, but this is not always practical – if you don't write unit tests for your getters and setters, then this figure becomes impossible! Furthermore, 100% line coverage does not guarantee 100% **code** coverage. See if you can think of a situation where this applies.

Boundary Value Analysis and Equivalence Partitioning

Boundary value analysis and equivalence partitioning are testing techniques that can be used to improve scenario coverage when unit testing a program. Using these techniques as guidelines when testing ensures that unit tests cover the majority of possible scenarios that can occur in terms of valid and invalid input. These strategies also ensure that the potentially very large range of values that could be used for testing is cut down to a manageable size through partitioning of input and focusing on partition boundaries when testing.

Equivalence Partitioning

Equivalence Partitioning is where input values for a condition are partitioned into multiple “classes” of input. This technique allows for the creation of the minimum number of test cases to cover the maximum number of requirements, and reduces the number of redundant test cases in a project.

For example, an order form for loaves of bread from your local bakery might have a field to input how many loaves of bread you want. The order size is limited from one loaf to one hundred loaves of bread. Performing equivalence partitioning on this problem gives us three distinct categories of order size input:

- Between 1 and 100
- Less than 1
- More than 100

Therefore, this particular scenario has three equivalent partitions of input. Any value in each of these ranges can be tested, and so we end with three test cases for this scenario that cover all possible ranges of valid and invalid input.

Boundary Value Analysis

Boundary Value Analysis in testing is where the boundary conditions of a problem are tested to ensure edge cases are met. Boundary conditions are usually on the edge of equivalence partitions, and so the two techniques are often used together in practice.

Returning to the previous example, we will use boundary value analysis to derive test cases for the bread order form. We would test three values for each boundary:

- The boundary value (i.e. 1 and 100)
- The value one below the boundary (i.e. 0 and 99)
- The value one above the boundary (i.e. 2 and 101)

This time, we have six test cases for the same scenario. Testing above, below and at the boundary means we can ensure that behaviour of input around the boundary values is as we would expect it.

3. Exercise 1: Writing Unit Tests

Let's begin by writing some simple unit tests. For this exercise, you will use an existing Java project.

Clone the following repo and open it up in your IDE of choice:

```
`git clone https://eng-git.canterbury.ac.nz/lwa383/seng202_lab3_part1_2020.git`
```

You will find four classes:

1. App
2. Country
3. AppTest
4. CountryTest

The test classes contain the method signatures for 5 tests – your job is to implement the method body for each of these test cases.

How should you write a unit test?

You will notice that all of the methods outlined in the test classes are annotated with `"@Test"`. This is what tells the JUnit framework which methods should be run.

The fundamentals of a JUnit test involve asserting that an actual outcome is the same as the expected outcome. There are a variety of ways to do this, most revolving around the **`assertEquals()`** and **`assertTrue`**() methods in JUnit. Have a play around and see if you can write tests that pass, and fail.

An **important point** to note is that you should keep the number of asserts in your test methods to a minimum (ideally no more than 1), since unit, by definition, means one. Furthermore, a single assert allows developers to easily identify where errors have occurred in the codebase. If you feel like you need multiple assert statements, then feel free to create more test methods.

4. Test Driven Development

Test-driven development (TDD) is a development process that implements a test-first mindset where the developer writes a unit test prior to writing just enough production code to fulfil that test. The primary goal of TDD is to have you thinking of the requirements or design of your program before you begin writing your functional code. This process promotes clean, modular code that is reusable, extendable and works.

The process of TDD is outlined in Figure 1 below. The developer begins by writing a specific unit test that will test the behaviour that they require their program to perform. This unit test should be independent from the functional code and unrelated to how it will be implemented.

The next step is to write the bare minimum functional code to satisfy the constraints of the unit test. If the test fails, the developer will refactor the functional code (**NOT THE UNIT TEST**) until the aforementioned constraints are satisfied and the test passes. This process is then repeated until the desired functionality has been implemented.

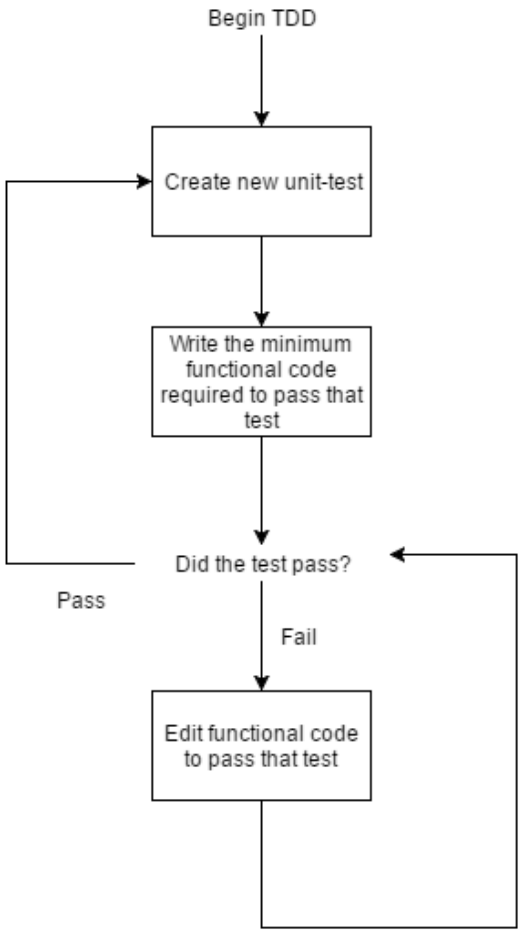

Figure 1: TDD process flow chart

This part of the tutorial will address unit testing best practices and discuss those as we work through the exercises. We will see how production code changes with each new test that we write and how this improves the design of our production code. The experience you gain through this tutorial should be practiced through your project work.

5. Exercise 2: Monopoly Game

We will be writing a simplified Monopoly game (we will not actually be playing it, just performing TDD on the functions below). We will start with some basic requirements. The game is played by two people, Player1 and Player2. A monopoly board has several different outcomes depending on what position you land on. Each position can either be a plain field (no charge or income) or can result in a fee/penalty or income of some sort. A field can also demand a change of player's location on the board.

Begin by navigating to your preferred working directory in the terminal and download the project folder via the following command.

```
`git clone https://eng-git.canterbury.ac.nz/lwa383/seng202_lab3_part2_2020.git`
```

Open your IDE of choice, and select import project and select the root directory (lab3_tdd). Select Import project from external model and choose Maven. Follow through the dialogs as you have done in previous tutorials. Once the project has been imported correctly, navigate through the source code and try to understand what each file is doing.

To start with we will consider the following fields on the board:

- Free_Parking – no charge or income
- Go – offers income of 200
- Go To Jail – no charge, but changes player location to “Jail” field on board
- Income Tax – if player’s balance is less than 2000 reduce the balance by 10%, otherwise a fixed fee of 200 applies
- Jail – no charge
- Luxury Tax – incur fee of 75
- Pay Owner – if player is owner no charge, otherwise a player pay a fee of 100 to the owner

Currently there are method stubs for these different conditions. It is your job to write tests for each of these functions and then implement the desired functionality. Remember, each unit test should only perform one assertion and therefore some functionality may require multiple unit tests to ensure it is working correctly.