# Software Quality Assurance and Testing

# Workbook for Laboratory on Test-Driven Development

|  |  |  |  |
| --- | --- | --- | --- |
| Name: | Nora Yang (Peiyi Yang) | Student Number: | 20165164 |

This notebook should be completed after you have done the development work in the second Laboratory at the end of the course.

Answer the following questions. Then, submit this with the code you wrote. Place the document (as a PDF) and the code in a Zip file. Then, submit to Blackboard.

This notebook and your code are marked. They are worth 20% of your course.

**1. Which stories did you plan to work on in the available time? This includes the time spent on the task, both in the laboratory session and in your own time.**(explain using a few short paragraphs)

I worked on all stories and made sure those that were already implanted still work. Descriptions are in question 2.

**2. Which stories did you complete?**

(explain using a short paragraph)

All of them.

Story 1:

Given the position a player takes,

to check if he wins vertically, check if the three squares below that position are all his moves;

to check if he wins horizontally, first locate the most left one of these consecutive squares, then check if the three squares on the right side are all his moves;

to check if he wins diagonally, first locate the most left one of these consecutive squares, then check if the next three squares on the right side are his moves;

Story 3&4:

I used a list (List<GridPosition> *moves*) to record moves and another (List<GridPosition> undone*Moves*) to record moves that are undone. They are all empty at first.

To undo a move, remove the last one item from *moves* and add it to *undoneMoves*. Then assign EMPTY to the corresponding position. If it’s a single player game, repeat the above steps once.

To redo a move, remove the last one item form *undoneMoves* and add it to *moves*. Then set the corresponding position to be PLAYER\_ONE or PLAYER\_TWO. If it’s a single player game, repeat the above steps once.

After a player takes a move, empty *undoneMoves* so that he can not redo yet.

Story 6:

In a single player game, the computer takes a move at a randomly generated empty position.

Story 7:

When a new game is started, board are emptied and redisplayed, *moves* and *undoneMoves* are emptied, buttons are enabled(may be disabled in the last game if a player has won), and game mode is set to single or two player correspondingly.

Story 9:

With the list moves, this will be easy to implement by printing out its items.

**3. Were there any problems working on the stories?**

(explain using a few short paragraphs)

When I was writing functions to check if a player has won, the IndexOutOfBoundsException was often thrown. Then I checked my while loop and found out that I forgot to make sure the GridPotion object’s column is between 0 and 6, row 0 and 5.

**4. Thinking about the lectures - what is Refactoring? How it can be used with Tests in a project using an Agile methodology?**

(explain using a few short paragraphs)

Refactoring is to change the internal code without effecting the external behaviour of the program in order to increase code reuse, code readability, maintainability, etc. For example, break a large, existing function into several smaller ones to improve the code.

In TDD, refactoring is part of the development. An iteration consists of Red, Green and Refactor. First, we write a test, then run all tests to see if the test fails (Red). Then we write some code to let the new test past (Green). In this step we write code that works but may be very complicated, long or reduplicated. Finally, we refactor the code (Refactor) without altering external behaviour.

**5. What refactoring steps did you use during this exercise?**

(explain using a few short paragraphs)

1. extract a function from a large one to break it into two smaller functions (Extract Method)

I turned handleButtonAction() into handleButtonAction() and takeAMove() to shorten handleButtonAction() and separate different logic.

2. move a function to a new class (Move Method)

I moved the initialise() function to the game class and call game.initialiseBoard() in handleStartTwoPlayerGameAction().

3. reuse code

Initially I wrote enableButtons() to enable buttons at the beginning of a game. Then I found out that enableButtons() and disableButtons() look very much alike so I turned two functions into one disableButtons(Boolean bool) to disable or enable buttons.

4. rename variables and functions

I made the names more descriptive instead of using a, b, etc.

5. remove local variables (Replace Temp with Query)

In a single player game, I need to generate a random position (stored in local variable randPosition) for the computer player then pass it to takeAMove(randPosition). When refactoring, I extract the part of logic that generates the position and add a new function GridPosition getRandomPosition(). So now I can remove the local variable randPosition and just do takeAMove(game.getRandomPosition()).

6. make the code shorter by adding helper functions

Before refactoring, if I want to pass a GridPostion object I must instantiate one first, set its column and row value and then pass it to the function (4 lines of code at least). When refactoring, I add a new constructor GridPosition(int row, int column). Now I can pass an [anonymous object](http://www.baidu.com/link?url=xYo7dwzGVwIaDSD0BFMz73h3AyDgfT0EUoTwuwnMWlXiAOej7hQCCbVpmkD-FiM2jkvd96NEsJUSGw-F5Zra0tNlS7fdNxn-jx-rLlgsdxYvSxjuN_o_KdOlLPYKA69_)(1 line of code).

**6. Based on your experience in these workshops, discuss the advantages and disadvantages of using Test-Driven Development and Refactoring to develop software and tests.**

(explain using approximately 300 words)

Advantage:

1. Code is more reliable.

Because it passes all tests designed, we have the confidence that it is reliable.

2. Time of debugging and maintaining is less.

Because when the new code fails to pass the tests, we know that the old code is still correct and don’t have to look for bugs from the beginning. Also, from different test cases we know where exactly the problem is. Also, the code will be easier to maintain.

3. TDD ensures all required functionalities are implemented.

In TDD, we write tests before implementing code based on required functionalities. This helps ensure that we are not missing a required functionality. Also, when code changes, tests don’t have to change as long as functionalities don’t.

4. TDD ensures safe refactoring.

With tests already written and serve as a safety net, we can refactor a lot without having to figure out if the changes are correct or not – just run the tests to see.

5. Test cases can be used as documentation

When we are writing tests, we would like to write descriptive function names and cover enough test cases, so these tests can be seen as an informal documentation that explain the code well and list all possible input and output of a function.

6. TDD helps make better design.

With safe refactoring, we can improve code readability, reuse, maintainability, etc, and, increase cohesion and decrease coupling.

Disadvantage:

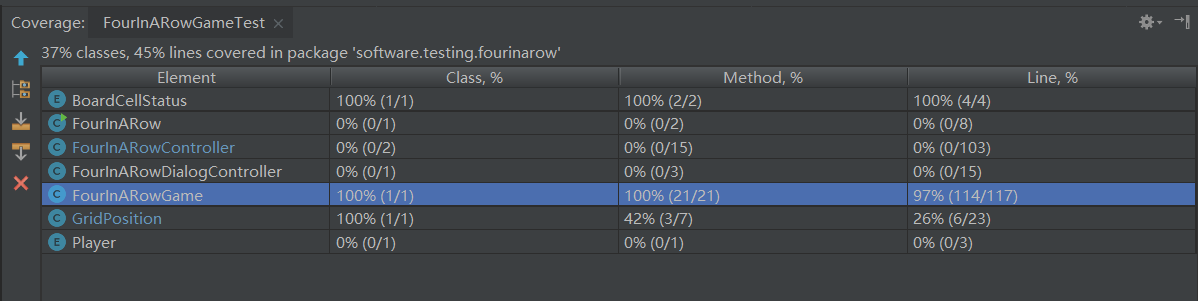
1. TDD does not ensure fault-free software.

When test cases are not enough to cover all cases or they are badly-designed and logically wrong, code that passes all tests might still have fault.

2. TDD does not help save time in development.

Because designing and implementing test code still takes a lot of time and experience for programmers. This increases the complexity and cost of development, and risk of running out of time.

**7. What code coverage did your tests achieve?** Look at the JaCoCo HTML reports. These are generated when you run the maven task. Include a screenshot of the overall coverage.



**Optional questions**

Is there any specific part of your Workbook or your Code that you would like me to comment on?

No.