

SOFTWARE QUALITY

Ontario Tech University

The Rhythms of TDD



Date : February 5th , 2024

Mostafa Abedi (100787034)

mostafa.abedi@ontariotechu.net

Table of Contents

Introduction	1
Learning Objectives	1
Tasks	2
Task 1	2
Task 2	4
Task 3	5
Questions and Answers	6
Question 1.....	6
Question 2.....	7
Question 3.....	7
Links	7
Github	7
Conclusion	7

Introduction

A brief description of the tutorial

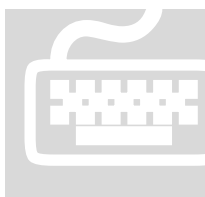
This tutorial takes a further look at the Test Driven Development. Working more on the Wordz Application we understand readability and how to improve it.

Learning Objectives

Our goals for this tutorial



1. Understanding Test-Driven Development and the RGR cycle



2. Gain Experience writing JUnit Tests with Java. Writing tests, making implementation and refactoring.

Tasks

Task 1

Test Driven Development starts with creating a test. In this example we need to create a word game. This games requirements are as follows:

- The correct letter in the correct position has a black background
- The correct letter in the wrong position has aa gray background
- Incorrect letter not present in the word have a white background

First we create a test to see if a single letter is correct or not. Figure 2.1.1 shows this test.

```
package com.wordz.domain;

import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.assertThat;

public class WordzTest {

    @Test
    public void oneIncorrectLetter(){

        var word = new Word("A");

        var score = word.guess("z");

        var result = score.letter(0);

        assertThat(result).isEqualTo(Letter.INCORRECT);
    }
}
```

Figure 2.1.1 – Tests to check for a correct single letter

Our test allows us to implement each class we are missing. We can use the quick implementation option provided by IntelliJ. Figure 2.1.2 shows this feature. Figure 2.1.3 and 2.1.4 shows these class implementaions.

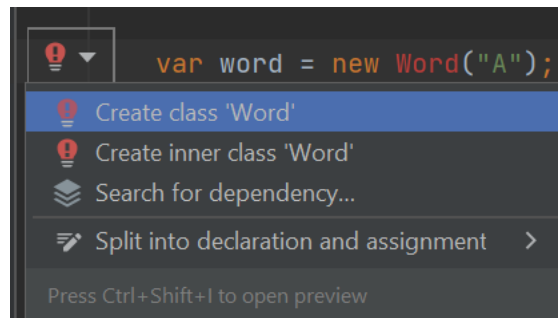


Figure 2.1.2 – Quick implementation of a class feature

```
1 public class word {  
2     1 related problem  
3     public Word(String correctWord){  
4         // Not implemented  
5         return ;  
6     }  
7     1 related problem  
8     public Score guess(String attempt){  
9         var score = new Score();  
10        return score;  
11    }
```

Figure 2.1.3 – Implementation of Word and Score class

```
1 package com.wordz.domain;  
2  
3 public class Score {  
4     1 related problem  
5     public Letter letter(int position){  
6  
7         return Letter.INCORRECT;  
8     }  
9 }
```

Figure 2.1.4 - Implementation of Score class

Task 2

Following the first task, we move from red to green by adding a test for a single correct letter to the class.

```
3      public class Word {
4          private final String word;
5
6          public Word(String correctWord) {
7              this.word = correctWord;
8          }
9
10         public Score guess(String attempt) {
11             var score = new Score(word);
12
13             score.assess(attempt);
14             return score;
15         }
16     }
17 }
```

Figure 2.2.1 – Full implementation of the word class

```
public class Score {
    private final String correct;
    private final List<Letter> results = new ArrayList<>();
    private int position;

    public Score(String correct) {
        this.correct = correct;
    }

    public Letter letter(int position) {
        return results.get(position);
    }

    public void assess(String attempt) {
        for (char current: attempt.toCharArray()) {
            results.add( scoreFor(current) );
            position++;
        }
    }
}
```

Figure 2.2.2 – Full implementation of Score class

```
@Test
public void oneIncorrectLetter() {
    var word = new Word( correctWord: "A");
    var score : Score = word.guess( attempt: "Z");
    assertScoreForGuess(score, INCORRECT);
}

private void assertScoreForGuess(Score score, Letter... expectedScores) {
    for (int position = 0; position < expectedScores.Length; position++) {
        Letter expected = expectedScores[position];
        assertThat(score.letter(position))
            .isEqualTo(expected);
    }
}
```

Figure 2.2.3 – Improving the test by considering readability

Task 3

We will be creating a test for the second letter being in the wrong position. Figure 2.2.1 shows the test and the classes that need to be implemented. This was the red phase.

```
@Test
public void secondLetterWrongPosition() {
    var word = new Word( correctWord: "AR");
    var score : Score = word.guess( attempt: "ZA");
    assertScoreForGuess(score, INCORRECT,
        PART_CORRECT);
}
```

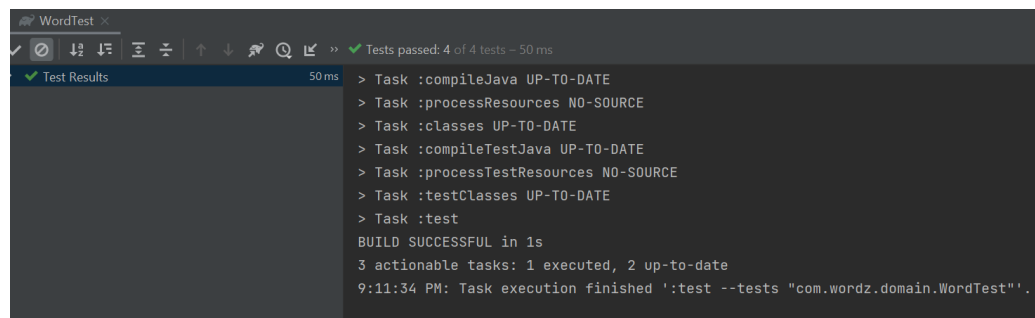
Figure 2.3.1 – Adding a new test for second wrong letter

Next we are at the green phase where we add all the classes required for the test to the main class. Figure 2.3.2 Shows the code to run a successful test.


```
22 @ public void assess(String attempt) {  
23     for (char current: attempt.toCharArray()) {  
24         results.add( scoreFor(current) );  
25         position++;  
26     }  
27 }  
28  
29 private Letter scoreFor(char current) {  
30     if (isCorrectLetter(current)) {  
31         return Letter.CORRECT;  
32     }  
33  
34     if (occursInWord(current)) {  
35         return Letter.PART_CORRECT;  
36     }  
37  
38     return Letter.INCORRECT;  
39 }  
40  
41 private boolean occursInWord(char current) { return correct.contains(String.valueOf(current)); }  
42  
43 private boolean isCorrectLetter(char currentLetter) { return correct.charAt(position) == currentLetter; }  
44  
45 }  
46 }
```

Figure 2.3.2 – New implementation

Finally by executing the test we get successful runs. Figure 2.3.3.



The screenshot shows an IDE window titled 'WordTest'. The top bar indicates 'Tests passed: 4 of 4 tests - 50 ms'. The main area displays a list of tasks: 'Task :compileJava UP-TO-DATE', 'Task :processResources NO-SOURCE', 'Task :classes UP-TO-DATE', 'Task :compileTestJava UP-TO-DATE', 'Task :processTestResources NO-SOURCE', 'Task :testClasses UP-TO-DATE', and 'Task :test'. Below these tasks, it says 'BUILD SUCCESSFUL in 1s' and '3 actionable tasks: 1 executed, 2 up-to-date'. At the bottom, it shows the command '9:11:34 PM: Task execution finished ':test --tests "com.wordz.domain.WordTest"'.

Figure 2.3.3 – test Success

Questions and Answers

Question 1

What TDD cycle is illustrated by writing a failing test, making it pass with production code, and then refactoring the code?

The TDD (Test-Driven Development) cycle illustrated by writing a failing test, making it pass with production code, and then refactoring the code is commonly known as the Red-Green-Refactor cycle. In this cycle:

Red: Write a failing test to specify new functionality.

Green: Write the minimum amount of code to make the test pass.

Refactor: Improve the code without changing its behavior.

Question 2

Which TDD cycle step involves reviewing the code for 'code smells'?

The TDD cycle step that involves reviewing the code for 'code smells' is the Refactor step. During this step, after making the test pass, developers examine the code for any potential improvements in terms of readability, maintainability, or efficiency. 'Code smells' refer to signs that the code might need refactoring to enhance its overall quality.

Question 3

What does the term 'refactoring' imply in the context of TDD?

In the context of TDD, the term 'refactoring' implies making changes to the existing code to improve its structure, design, and readability without altering its external behavior. Refactoring is an essential step in the TDD cycle, performed after the test has been successfully passed with production code. The goal of refactoring is to enhance the code's maintainability, reduce duplication, and address any 'code smells' identified during the development process.

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 provides a hands-on exploration of Test-Driven Development (TDD) using the example of a Wordz Application. The tutorial focuses on the Red-Green-Refactor (RGR) cycle, emphasizing the importance of writing tests first, implementing code to make the tests pass, and then refactoring for improved code quality.