COMP3111: Software Engineering

Unit Testing

Learning Outcomes

- Be able to write unit tests
- Be able to generate and understand coverage reports

An example on how to conduct unit testing in Eclipse will be given, together with the related reference links (Exercise 1 & 2). You are going to apply what you have learned to design and implement appropriate unit tests for your team project (Lab Activity).

Exercise 1: Writing Unit Tests

1.1: Create a new Gradle Project "UnitTestDemo" in Eclipse IDE

1.2: Edit the content of "Library.java"

Library.java

```
1  package UnitTestDemo;
2
3  public class Library {
4      public static void main(String arg[]) {
5          System.out.println("Hello, JUnit 4!");
6      }
7
8      public static int computeOne() {
9          return 1;
10     }
11
12     public static boolean isItTrue() {
13         return true;
14     }
15
16     public static boolean isEvenNumber(int num) {
17         if ((num % 2) == 0)
18             return true;
19         else
20             return false;
21     }
22
23     public static int[] sort(int[] anyArray) {
24         return new int[] {1, 2, 3, 4, 5};
25     }
26 }
27
```

1.3: Right-click at "UnitTestDemo" to create a new JUnit Test Case "LibraryTester.java"

LibraryTester.java

```
1  package UnitTestDemo;
2
3  import static org.junit.Assert.*;
4
5
6
7  public class LibraryTester {
8      int[] arrAscending, arrDescending;
9
10     @Before
11     public void setUp() throws Exception {
12         arrAscending = new int[] {1, 2, 3, 4, 5};
13         arrDescending = new int[] {5, 4, 3, 2, 1};
14     }
15
16     @Test
17     public void computeOneWithValidInput() {
18         assertEquals(1, Library.computeOne());
19     }
20
21     @Test
22     public void isItTrueWithValidInput() {
23         assertTrue(Library.isItTrue());
24     }
25
26     @Test
27     public void sortDescendingOrder() {
28         assertArrayEquals(arrAscending, Library.sort(arrDescending));
29     }
30
31     @Test
32     public void sortAscendingOrder() {
33         assertArrayEquals(arrAscending, Library.sort(arrAscending));
34     }
35
36     @Test
37     public void isEvenNumReturnTrue() {
38         assertTrue(Library.isEvenNumber(2));
39     }
40 }
41
```

1.6: Edit the content of “build.gradle”

build.gradle

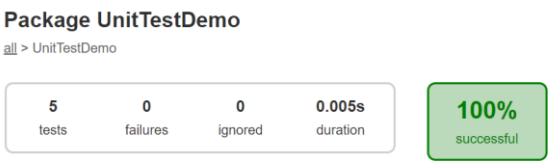
```
plugins {
    id 'java'
    id 'application'
    id 'jacoco'
}

repositories {
    jcenter()
}

dependencies {
testImplementation 'junit:junit:4.12'
compile group: 'net.sourceforge.htmlunit', name: 'htmlunit', version: '2.31'
}

jacocoTestReport {
    reports {
        xml.enabled false
        csv.enabled false
        html.destination file("${buildDir}/jacocoHTML")
    }
}

mainClassName = 'UnitTestDemo.Library'

```

1.7: Go to Gradle Tasks Tab; Click at “Refresh Tasks for All Projects”

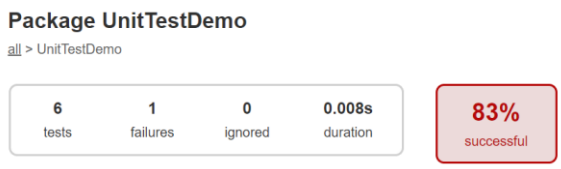
1.8: Run the Gradle Task “test” under the category “verification”

1.9: Result of each test could either be pass or failure. The test report generated can be found under the project workspace folder at: ‘build/reports/tests/test/index.html’

(sample report of 5 tests with 0 failure)



(sample report of 6 tests with 1 failure)



1.10: References

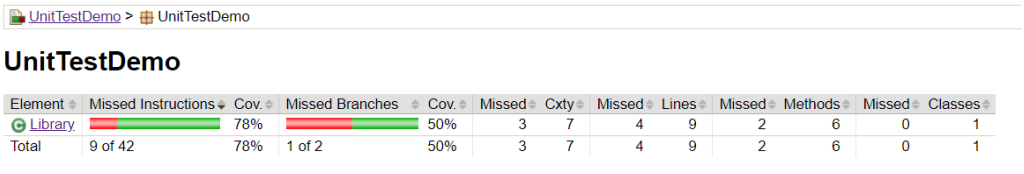
- Video: JUnit 4 and Eclipse: The Basics (36:20)
- JavaDoc: List of Assertion Methods
- FAQ on JUnit 4

Exercise 2: Generating Coverage Reports

2.1: Run the Gradle Task “jacocoTestReport” under the category “verification” to generate the coverage report, after all the tests have been passed (100% successful).

2.2: The coverage report generated can be found under the project workspace folder at: ‘build/jacocoHTML/index.html’

(A sample report with instruction coverage and branch coverage at 78% and 50%, respectively)



So, what do these rates of "coverage" indicate? These measures reflect on how good your test cases have been written or how many different branches of code that your tests have visited. There are two major types of coverage: statement coverage and branch coverage.

- Focus of statement coverage, or instruction coverage, is on whether every statement in the code is executed at least once.
- Focus of branch coverage, or decision coverage, is on whether each branch of a decision (in selection or looping statements) is tested, i.e. all statements in the conditional branches to be executed at least once.

Note that 100% coverage does not mean that your code is bug-free! For example, your test cases may only cover a small range of values. To make sure your code is bug-free, you should always consider testing a wide range of values even though that might not increase the rate of coverage. However, if the coverage is low (less than 50%), it implies there are not enough effort in testing.

Lab Activity

- You are advised to switch to a new local branch (for example, lab_UnitTest) while working on this individual lab assignment.
- By now, you might have already developed a few new classes for your team project other than those given in the skeleton codebase. If not, you could add a new one for the purpose of this lab assignment.
- Design and implement a JUnit Test Case (with at least 2 tests) for one or more selected class in your team project
- Evaluate the effectiveness of your test case with reference to the generated test report and Jacoco coverage report
- Easter Egg: We provide you with a TestFX example in the project skeleton codebase. It is not a mandatory task to use TestFX in your project (or this lab). It might however help you to pull up the rate of coverage. Enjoy and have fun!

Assessment

Make an individual submission of a PDF file through Canvas with the following information included:

- Source code for a JUnit Test Case (with at least 2 tests)
- Test report showing all the tests have been passed (100% successful)
- Jacoco coverage report showing the branch coverage (10% or higher)