SENG 637 - Dependability and Reliability of Software Systems

Lab. Report #2 – Requirements-Based Test Generation

Group #:	4
Student Names:	1. Mahdi Jaberzadeh Ansari
	2. Aadharsh Hariharan
	3. Shirin Yamani
	4. Rahul Ravi

File name	Link
Demo	https://github.com/seng637-winter-2022/seng637-a2-mjza/blob/main/Demo.mp4

1 Introduction

In this assignment we have been introduced with the fundamentals of automated unit testing using jUnit in Eclipse IDE. We actually developed some unit tests for ten functions of two classes of the JFreeChart library. In this work, we tested the following functions of `Range` and `DataUtilities` classes:

Here are the functions from the class `DataUtilities` that we wrote test cases for them:

1. calculateColumnTotal
2. calculateRowTotal
3. createNumberArray
4. createNumberArray2D
5. getCumulativePercentages

Here are the functions from the class `Range` that we wrote test cases for them:

1. isNaNRange
2. shift
3. expandToInclude
4. combineIgnoringNaN
5. intersects

We use the following libraries for testing:

1. JUnit for Java testing
2. JMock for mocking dependencies

We started with the warm up scenarios in the lab instructions. Installed Eclipse and related libraries, made a new Java project and tried to make it runnable. After playing around with the test unit samples provided by the instruction, we started to analyze the selected functions for writing the test scenarios.

We started from the modified JavaDocs that also provided in the artifacts file, and tried to discover more about the expected behavior of those functions. Then we wrote the test scenarios on the paper to cover all possible test cases. Later, we used those scenarios on the paper and implemented them in different test classes. We made one test class for each function and put all the test scenarios of the function in the same class.

Finally, we executed the test suite we created on `jfreechart-1.0.19.zip` . However, we couldn't find any test case that fails due to the intentional modifications. We even created a thread in the discussion section of D2L and sent emails to instructors, unfortunately, haven't received any answers.

2 Detailed description of unit test strategy

2.1 How you used the black-box test-case design techniques equivalence classes, and boundary value analysis.

We followed the black-box test case design strategies to test different functions of the two classes.

Equivalence class technique was used for all range methods to group similar group of values considered for lower bounds and upper bounds in the example ranges to test. They primarily involved combinations of positive and negative values. Here are more details regarding the test cases of each function:

1. **isNaNRange:** This function was the initial test case for us and we easily put the NaN one zero, one and both boundaries of a range.
2. **shift:** For this function function we defined a base range (-1,1). And then we considered shifting the range to left or right by 2. The value 2 selected to cover crossing the origin point (zero) in the axel.
3. **expandToInclude:** We set an example range to which we applied the expand functions that would include positive (greater than the upper bound) value, negative (lower than the lower bound) value and to check with a value already present in the range.
4. **combineIgnoringNaN:**
   We initialized 2 base ranges to combine by checking for different occurrences of NaN values in upper and lower bounds of both the ranges respectively.
5. **intersects:** A base example range has been defined and checked with a variety of range values that had intersecting elements in upper bound, lower bound, both bounds as well as non-intersecting bounds.
6. **calculateColumnTotal:** This function also consider for us as a base for the class `DataUtilities` especially that you provided a sample code for this one. In our test scenarios we had to play with different double values. Therefore we considered an empty array, and some arrays with one column having two negative double values (i.e., -7.5 and -2.5), two positive double values(i.e., 7.5 and 2.5), one negative and one positive double values (i.e., -7.5 and 2.5), two opposite polarity same magnitude values (i.e., -7.5 and -7.5), two zero values, one double value and finally two double values to test the precision (i.e., 5.6 and 5.8)!

7. **calculateRowTotal:** The same scenario and values like step 6 used, except that this time we used a single row array!

8. **createNumberArray:** For the number array, we used all the special double values that are (Double.MAX_VALUE, 0.0d, Double.MIN_NORMAL, Double.MIN_VALUE, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY, Double.MAX_EXPONENT, Double.MIN_EXPONENT, 1.23456e300d, -1.23456e-300d, 1e1d).

9. **createNumberArray2D:** For the 2D array we used same values in step 8, plus a random double value that is generated in runtime. However, we tried to arrange them in different format of 2D arrays. An empty array, an array with single value, an array with single column and 11 rows, and array with single row and 11 columns, a 2D array with 4 rows and 3 columns were our test cases.

10. **getCumulativePercentages:** For testing cumulative percentages we used the following lists {2,2,2,2}, {2,0,0,0}, {0,0,0,2}, {0,0,0,0}, and empty list which we could imagine that they can cover all test cases for this function.

2.2 List the name of the test cases you have designed and identify which one covers which parts of the strategy (which partition, which class, etc.)

The naming conventions for our test classes is as follows:

```
1. The name starts with the name of the class that the method under test belongs
to.
2. Then the name of the function under testing
3. Finally, the post fix `Test.java` has been stuck to the name
```

For the function names we tried to provide a meaningful name. For example the function name `calculateColumnTotalForTwoPositiveValues` means it is a function for testing the function `calculateColumnTotal` when the both values are positive.

Here are the list of test classes and the function names:

1. Function `DataUtilities.CalculateColumnTotal` has been tested under the `DataUtilitiesCalculateColumnTotalTest.java` class, contains the following test scenarios:

    - **calculateColumnTotalForTwoPositiveValues:** Equivalence strategy testing for positive values implemented.

- **calculateColumnTotalForTwoNegativeValues:** Equivalence strategy testing for negative values implemented.
- **calculateColumnTotalForTwoNegativePositiveValues:** Equivalence strategy testing for negative and positive values implemented.
- **calculateColumnTotalForTwoPositiveNegativeValues:** Equivalence strategy testing for positive and negative values implemented.
- **calculateColumnTotalForTwoOppositePolaritySameMagnitudeValues:** Equivalence strategy testing for same magnitude but opposite polarity values implemented.
- **calculateColumnTotalForTwoZeroValues:** Boundary strategy testing for zero values implemented.
- **calculateColumnTotalForSingleValue:** Equivalence strategy testing for single value implemented.
- **calculateColumnTotalForEmptyValues:** Boundary strategy testing for null values implemented.
- **calculateColumnTotalForPrecision:** Equivalence strategy testing for precision implemented.

2. Function `DataUtilities.CalculateRowTotal` has been tested under the `DataUtilitiesCalculateRowTotalTest.java` class, contains the following test scenarios:

   - **calculateRowTotalForTwoPositiveValues:** Equivalence strategy testing for positive values implemented.
   - **calculateRowTotalForTwoNegativeValues:** Equivalence strategy testing for negative values implemented.
   - **calculateRowTotalForTwoNegativePositiveValues:** Equivalence strategy testing for negative and positive values implemented.
   - **calculateRowTotalForTwoPositiveNegativeValues:**: Equivalence strategy testing for positive and negative values implemented.
   - **calculateRowTotalForTwoOppositePolaritySameMagnitudeValues:** Equivalence strategy testing for same magnitude but opposite polarity values implemented.
   - **calculateRowTotalForTwoZeroValues:** Boundary strategy testing for zero values implemented.
   - **calculateRowTotalForSingleValue:** Equivalence strategy testing for single value implemented.
   - **calculateRowTotalForEmptyValues:** Boundary strategy testing for null values implemented.
   - **calculateRowTotalForPrecision:** Equivalence strategy testing for precision implemented.

3. Function `DataUtilities.CreateNumberArray2D` has been tested under the `DataUtilitiesCreateNumberArray2DTest.java` class, contains the following test scenarios:

    - **createNumberArray2DForEmptyDoubleArray:** Boundary strategy testing for null values implemented.
    - **createNumberArrayForSingleDoubleArray:** Equivalence strategy testing for single value implemented.
    - **createNumberArray2DForSingleRowDoubleArray:** Equivalence strategy testing for single row implemented.
    - **createNumberArray2DForSingleColumnDoubleArray:** Equivalence strategy testing for single column implemented.
    - **createNumberArray2DForMxNDoubleArray:** Equivalence strategy testing for multiple rows and columns implemented.

4. Function `DataUtilities.CreateNumberArray` has been tested under the `DataUtilitiesCreateNumberArrayTest.java` class, contains the following test scenarios:

    - **createNumberArrayForMultiDoubleArray:** Equivalence strategy testing for multiple values implemented.
    - **createNumberArrayForEmptyDoubleArray:** Equivalence strategy testing for null values implemented.
    - **createNumberArrayForSingleDoubleArray:** Equivalence strategy testing for single values implemented.

5. Function `DataUtilities.GetCumulativePercentages` has been tested under the `DataUtilitiesGetCumulativePercentagesTest.java` class, contains the following test scenarios:

    - **getCumulativePercentagesForFourPositiveValues:** Equivalence strategy testing for 4 positive values implemented.
    - **getCumulativePercentagesForSinglePositiveValues:** Equivalence strategy testing for single values implemented.
    - **getCumulativePercentagesForEmptyValues:** Equivalence strategy testing for null values implemented.
    - **getCumulativePercentagesForFirstThreeRowsAreZero:** Equivalence strategy testing for zero values of first three rows implemented.
    - **getCumulativePercentagesForLastThreeRowsAreZero:** Equivalence strategy testing for zero values of Last three rows implemented.

- **getCumulativePercentagesForAllRowsAreZero:** Equivalence strategy testing for zero values of all rows implemented.
-

6. Function `Range.ExpandToInclude` has been tested under the `RangeExpandToIncludeTest.java` class, contains the following test scenarios:

   - **expandedRangeIncludingTwo:** Equivalence class strategy for all sets of positive values to include
   - **expandedRangeIncludingNegativeTwo:** Equivalence class strategy for all sets of negative values to include
   - **expandedRangeIncludingExistingValueZero**

7. Function `Range.intersects(Range)` has been tested under the `RangeIntersectionTest.java` class, contains the following test scenarios:

   - **rangeLowerBoundIntersectionShouldBeTrue:** Equivalence class strategy for all sets of intersecting lower bounds
   - **rangeUpperBoundIntersectionShouldBeTrue:** Equivalence class strategy for all sets of intersecting upper bounds
   - **rangeIntersectionShouldBeFalse:** Equivalence class strategy for all sets of non-intersecting upper and lower bounds
   - **rangeIntersectionShouldBeTrue:** Equivalence class strategy for all sets of intersecting both upper and lower bounds

8. Function `Range.combineIgnoringNaN(Range,Range)` has been tested under the `RangeCombineIgnoringNaNTest.java` class, contains the following test scenarios:

   - **shouldIgnoreLowerBoundNaNInRange1:** Equivalence class strategy for all sets of valid upper bounds in range1
   - **shouldIgnoreUpperBoundNaNInRange1:** Equivalence class strategy for all sets of valid lower bounds in range1
   - **shouldIgnoreLowerBoundNaNInRange2:** Equivalence class strategy for all sets of valid lower bounds in range2
   - **shouldIgnoreUpperBoundNaNInRange2:** Equivalence class strategy for all sets of valid upper bounds in range2
   - **shouldIgnoreNaNInRange2:** Boundary value analysis with NaN values for all upper and lower bounds in range2
   - **shouldIgnoreNaNInRange1:** Boundary value analysis with NaN values for all upper and lower bounds in range1

- **shouldReturnNull:** Boundary value analysis with NaN values for all upper and lower bounds

9. Function `Range.shift(Range,Value,shouldCrossZero)` has been tested under the `RangeShiftTest.java` class, contains the following test scenarios:
   - **shiftTowardsPositiveScaleByCrossingZero:** Equivalence class strategy for all positive shift values
   - **shiftTowardsNegativeScaleByCrossingZero:** Equivalence class strategy for all negative shift values
   - **shiftTowardsNegativeScaleWithoutCrossingZero:** Equivalence class strategy for all positive shift values
   - **shiftTowardsPositiveScaleWithoutCrossingZero:** Equivalence class strategy for all negative shift values
10. Function `Range.isNaN(Range)` has been tested under the `RangeIsNaNTest.java` class, contains the following test scenarios:
    - **noNaNValuesInRange:** Equivalence class strategy all non-NaN values
    - **bothNaNValuesInRange:** Boundary value analysis all NaN values
    - **lowerBoundNaNValueInRange:** Equivalence class strategy for all non-NaN upper bound values and NaN-LB
    - **upperBoundNaNValueInRange:** Equivalence class strategy for all non-NaN lower bound values and NaN-UB

2.3 Include a discussion about what you feel are the benefits and drawbacks about using mocking.

Benefits of using mocks:

- *Bug Localization:* We know that if a test is failed it is because of the class under test not its dependencies. So, we can accurately find the responsible method, understand what is not working then solve the issue!
- *Cost of Calling External Methods:* Often calling dependent methods is not possible. For example, they return a value which is not in the given range so we will face a Unexpected Return Values which basically means that it is not proper for our testing strategy. Or it may be difficult to cause them to return the exact value that we want. So here Mocking helps us to solve this issue pretty smooth and simple.

Disadvantages of Mocking:

- *Untested Interactions:* the main drawback is that we cannot test the interaction between methods. In other words, if we are calling a method from a mocked method in a wrong way, our tests don't cover them. So, we need an additional integration testing which can handle them simultaneously.
- *Development Efforts:* Mocking adds lots of additional effort in the work. This is because one should define the return value for each and every defined method which cause the development process gets much longer.

3 Test cases developed

Regarding the development, as mentioned in the section 2.2, we made separate test class for each function. We used an special naming convention for test classes and test functions to make it clear what they do.

We also tried to have suitable messages in assert functions and the whole code is accessible under the following address:

https://github.com/seng637-winter-2022/seng637-a2-mjza/tree/main/JFreeChart/src/org/jfree/data/test

For two functions we used mockery. (i.e., GetCumulativePercentages and CalculateColumnTotal). The reason was these two functions receive some objects as the input argument. Therefore, we needed to use mockery to mimic the functionality of those objects.

We explained more in our demo.

4 How the team work/effort was divided and managed.

Regarding the workload, we worked on the test cases together and finished the entire assignment in two sessions. Prior to our first session each of us installed eclipse and the necessary libraries to get familiarized with the testing process using JUnit.

5 Difficulties encountered, challenges overcome, and lessons learned

First, we faced some difficulties with the java version that needed to be used with respect to the project documentation! At first it is needed to select JRE8 (JSE 1.8), however, if someone changes the folder of the project from the default path in workspace to a custom path, then he or she still needs to set the JRE version again in the Libraries tab!

Second, we are different time zones that makes it a little harder. But we managed to code online via Zoom all together then push the result to GitHub.

Third, we so much learned mocking syntax for Jmock. As we didn't have the relevant experience before, it was a little difficult for us at the beginning since the syntaxes were pretty different from other libraries/frameworks. However, we manage this by by looking at some code examples and tutorials online.

6 Comments/feedback on the lab itself

Here we listed possible problems and issues that we faced during our development:

1. In section 2.1.1 under the third line there is a link to a picture (i.e. media/creatingProject.png) that is missing:

   > iii. Under the folder Java, ensure that Java Project is selected and in Use and execution environment JRE click on vesrsion8 and then click Next (media/creatingProject.png).

2. Figure 5 and Figure 6 are missing.

3. **"2.2.2 Add the Necessary Java Libraries"** must be "*2.1.2 Add the Necessary Java Libraries*"

4. Javadoc picture (i.e. Figure under **"2.1.4 Navigate Javadoc API Specifications"**) is not similar to the version of Javadoc that you provided.

5. **org.jfree.data.Range (in the package org.jfree.data): Has 19 methods.** --> (It said it has 15 methods).

6. **org.jfree.data.DataUtilities (in the package org.jfree.data): Has 9 methods.** --> (It said it has 5 methods).

7. In section 2.2.2 one line must be uncommented:

```
    // exercise        double result =
DataUtilities.calculateColumnTotal(values, 0);
```

```
Must change to:



// exercise
double result = DataUtilities.calculateColumnTotal(values, 0);
```

8. For making the example in 2.2.2 we needed to update the **hamcrest** library to its latest version.