**SENG 637 - Dependability and Reliability of Software Systems**

**Lab. Report #2 – Requirements-Based Test Generation**

|  |  |
| --- | --- |
| Group #: | 18 |
| Student Names: | Ahmed Iqbal |
| Aidan Forester |
|  |
|  |

# Introduction

This lab explores software testing via automated unit testing based on requirements for each unit. Using the JUnit framework, multiple test functions were developed to test the *Range* and *DataUtilities* class. The objectives of this lab are to develop automated test code using JUnit, and to utilize/develop mock objects for efficient testing, the team was successful in achieving these objectives. Prior to this lab, the team was not familiar with JUnit and unit testing, but upon completion of this lab is now familiar with the framework and the uses of unit testing.

Unit testing is a method used to automate the testing process while ensuring each unit – such as a class, function, or subsystem of a program – is performing correctly. The general approach is to create a testing class that contains multiple test case methods that test the units in question. It is important that a unit test verifies one specific unit of functionality and is very specific. A common framework for implementing these unit tests is JUnit – a java testing framework used in writing and running tests. There are other testing frameworks available, but for the scope of this lab JUnit was the established testing framework. Another component to this lab was the element of black box testing. Black box testing allows access to specifications of a program, but not detailed look at the implementation of the source code. That adds a level of uncertainty towards how the unit performs its function, and its partitions are slices of requirements combined with the unit’s expected behavior/output.

The scope of this lab was to create two testing classes using JUnit for the provided packages under the data package *org.free.data* which specifically are *DataUtilities* and *Range*. Two testing class called *RangeTest* and *DataUtilitiesTest* were developed that contain multiple test cases designed for testing five functions in each class. Mock objects are to be implemented to testing of certain functions that require a unit to be tested independently.

# Detailed description of unit test strategy

The strategy for developing our unit test involved analyzing the specifications of each function in question to understand what value ranges needed to be developed. It was also key to determine when and where a mock object was needed for a specific unit test function, as there are scenarios for when and where these mock objects are needed.

Due to the nature of this lab, we were able to split our tests between only two classes since there were no functions that depended on each other.

The following five functions were tested for the *Range* class:

* Contains(double value)
* Combine(Range range1, Range range2)
* Expand(Range range, double lowerMargin, double upperMargin)
* getLength()
* constrain(double value)

The following five functions were tested for the *DataUtilities* class:

* calculateColumnTotal(Values2D data, int column)
* calculateRowTotal(Values2D data, int row)
* createNumberArray(double[] data)
* createNumberArray2D(double[][] data)
* getCumulativePercentage(KeyedValues data)

# Test cases developed

Since the *Range* class was built around involving ranges, the team felt it was appropriate to use the boundary class testing methodology for three of the 5 methods, constrain, expand, and contains. These were dealt with in a way that involved checking the edge cases using such value points as the AUB, UB, LB, ALB, and more. This testing was done because these three functions operated in a way where the outcome depended on the values of the range. For the other two functions, equivalence class testing was used. Since these two functions, getLength(), and contains() did not involve a range, it was decided that the best way to test it would be to split it into tests based on different input values. As an example, for getlength(), this function returns a length based on the range it has. Since there are only so many different equivalent cases that can exist a test was done for each of them, i.e. positive range, negative range, null range, just zero, zero range, and a decimal range. The test cases for this class, including their function names are shown in Figure 1.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Testcase Number | Function Name | Testcase Function Name | Bound Name | Inputs | Expected Output | Actual Output | Bug? (Y/N) |
| 1 | Contains(Double value) | containsTestBLB() | BLB | -11 | False | False | No |
| 2 | Contains(Double value) | containsTestLB() | LB | -10 | True | True | No |
| 3 | Contains(Double value) | containsTestALB() | ALB | -9.9 | True | True | No |
| 4 | Contains(Double value) | containsTestNOM() | NOM | 0 | True | True | No |
| 5 | Contains(Double value) | containsTestBUB() | BUB | 9.9 | True | True | No |
| 6 | Contains(Double value) | containsTestUB() | UB | 10 | True | True | No |
| 7 | Contains(Double value) | containsTestAUB() | AUB | 11 | False | False | No |
| 1 | Combine(Range range1, Range range2) | combineRangeTwoNUll() | second range null | (-5, 5), null | (-5, 5) | (-5, 5) | No |
| 2 | Combine(Range range1, Range range2) | combineRangeOneNull() | first range null | Null, (-5, 5) | (-5, 5) | (-5, 5) | No |
| 3 | Combine(Range range1, Range range2) | combineBothNullRange() | both range null | Null, null | Null | Null | No |
| 4 | Combine(Range range1, Range range2) | combineSameRange() | same range for both | (-5, 5), (-5, 5) | (-5, 5) | (-5, 5) | No |
| 5 | Combine(Range range1, Range range2) | combineRangeWithOneSimilarPoint() | having the same point | (0, 5), (-5, 0) | (-5, 5) | (-5, 5) | No |
| 6 | Combine(Range range1, Range range2) | combineRangeWithNestedRangeInside() | one range inside the other range | (-2, 2), (-5, 5) | (-5, 5) | (-5, 5) | No |
| 7 | Combine(Range range1, Range range2) | combineRangeWithNoMutualPoints() | not any mutual points | (2, 5), (-5,-2) | (-5, 5) | (-5, 5) | No |
| 8 | Combine(Range range1, Range range2) | combineRangeOverlap() | overlap | (-2,5), (-5,2) | (-5, 5) | (-5, 5) | No |
| 1 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForNullRange() | Null range | Range = null, Upper margin = 0.5, Lower margin = 0.5 | Exception Throw | Exception Throw | No |
| 2 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForNullLowerMargin() | Null lower | Range = (-5, 5), Lower margin = null, Upper margin = 0.5 | Exception Throw | Fail | Yes |
| 3 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForNullUpperMargin() | Null upper | Range = (-5, 5), Lower margin = 0.5, Upper margin = null | Exception Throw | Fail | Yes |
| 4 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForLowermargin\_BLB() | Lower margin BLB | Range = (-5, 5), Lower margin = -0.1, Upper margin = 0.5 | Exception Throw | Fail | Yes |
| 5 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForLowerMargin\_LB() | Lower margin LB | Range = (-5, 5), Lower margin = 0, Upper margin = 0.5 | (-5, 10) | Pass | No |
| 6 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForLowerMargin\_ALB() | Lower margin ALB | Range = (-5, 5), Lower margin = 0.1, Upper margin = 0.5 | (-6, 10) | Pass | No |
| 7 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForLowerMargin\_NOM() | Lover margin NOM | Range = (-5, 5), Lower margin = 0.5, Upper margin = 0.5 | (-10, 10) | Pass | No |
| 8 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForLowerMargin\_BUB() | Lower margin BUB | Range = (-5, 5), Lower margin = 0.9, Upper margin = 0.5 | (-14, 10) | Pass | No |
| 9 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForLowerMargin\_UB() | Lower margin UB | Range = (-5, 5), Lower margin = 1, Upper margin = 1 | (-15, 15) | Pass | No |
| 10 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForLowerMargin\_AUB() | Lower margin AUB | Range = (-5, 5), Lower margin = 1.1, Upper margin = 0.5 | Exception Throw | Fail | Yes |
| 11 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForUpperMargin\_BLB() | Upper margin BLB | Range = (-5, 5), Lower margin = 0.1, Upper margin =-0.1 | Exception Throw | Fail | Yes |
| 12 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForUpperMargin\_LB() | Upper margin LB | Range = (-5, 5), Lower margin = 0.5, Upper margin = 0 | (-10, 5) | Pass | No |
| 13 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForUpperMargin\_ALB() | Upper margin ALB | Range = (-5, 5), Lower margin = 0.5, Upper margin = 0.1 | (-10, 6) | Pass | No |
| 14 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForUpperMargin\_NOM() | Upper margin NOM | Range = (-5, 5), Lower margin = 0.5, Upper margin = 0.9 | (-10, 14) | Pass | No |
| 15 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForUpperMargin\_BUB() | Upper margin BUB | Range = (-5, 5), Lower margin = 0.5, Upper margin = 1 | (-10, 15) | Pass | No |
| 16 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestForUpperMargin\_UB() | Upper margin UB | Range = (-5, 5), Lower margin = 0.5, Upper margin = 1.1 | Exception Throw | Fail | Yes |
| 17 | Expand(Range range, double lowerMargin, double upperMargin) | expandTestUpperMargin\_AUB() | Upper margin AUB | Range = (-5, 5), Lower margin = 0, Upper margin = 0 | (-5, 5) | Pass | No |
| 1 | getLength() | getLengthOfPositiveRange() | Positive Length | Range = (0, 8) | 8 | 8 | No |
| 2 | getLength() | getLengthWhenDecimalsInvolved() | Decimal Length | Range = (1.5, 11.5) | 10 | 10 | No |
| 3 | getLength() | getLengthWithNegativeNumbers() | Negative Length | Range = (-5, 10) | 15 | 15 | No |
| 4 | getLength() | getLengthOfZero() | Zero Length | Range = (0, 0) | 0 | 0 | No |
| 5 | getLength() | getLengthWithNUllValues() | Null Input | Range = null | Throw exception | Fail | Yes |
| 1 | constrain(double value) | constrain\_BLB() | BLB | Range = (-5,5), input = -5.1 | -5 | Fail | Yes |
| 2 | constrain(double value) | constrain\_LB() | LB | Range = (-5,5), input = -5 | -5 | -5 | No |
| 3 | constrain(double value) | constrain\_ALB() | ALB | Range = (-5,5), input = -4.9 | -4.9 | Fail | Yes |
| 4 | constrain(double value) | constrain\_NOM() | NOM | Range = (-5,5), input = 0 | 0 | 0 | No |
| 5 | constrain(double value) | constrain\_BUB() | BUB | Range = (-5,5), input = 4.9 | 4.9 | 4.9 | No |
| 6 | constrain(double value) | constrain\_UB() | UB | Range = (-5,5), input = 5 | 5 | Fail | Yes |
| 7 | constrain(double value) | constrain\_AUB() | AUB | Range = (-5,5), input = 5.1 | 5 | 5 | No |

Figure 1: Test suites for Range Class

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Testcase Number | Function Name | Testcase Function Name | Bound Names | Inputs | Expected Output | Actual Output | Bug? (Y/N) |
| 1 | calculateColumnTotal(Values2D data, int column) | calculateColumnTotalForTwoPositiveValues() | All positive numbers | [2.0, 8.0], col number = 0 | 10 | Pass | No |
| 2 | calculateColumnTotal(Values2D data, int column) | calculateColumnTotalForOnePositiveOneNegativeValues() | One positive one negative number | [-2.0, 8.0], col number = 0 | 6 | Pass | No |
| 3 | calculateColumnTotal(Values2D data, int column) | calculateColumnTotalForOneValue() | One number | [2.0], col number = 1 | 2 | Pass | No |
| 4 | calculateColumnTotal(Values2D data, int column) | calculateColumnTotalForNullValues() | Null - Null Pointer Exception | [null, null], col number = -1 | Error | Pass | No |
| 1 | calculateRowTotal(Values2D data, int row) | calculateRowTotalForTwoPositiveValues() | All positive numbers | [2.0, 8.0], row number = 0 | 10 | Fails | Yes |
| 2 | calculateRowTotal(Values2D data, int row) | calculateRowTotalForOnePositiveOneNegativeValues() | One positive one negative | [-2.0, 8.0], row number = 0 | 6 | Fails | Yes |
| 3 | calculateRowTotal(Values2D data, int row) | calculateRowTotalForOneValue() | One number | [2.0], row number = 1 | 2 | Fails | Yes |
| 4 | calculateRowTotal(Values2D data, int row) | calculateRowTotalForNullValues() | Null - Null Pointer Exception | [null], row number = -1 | Error | Pass | No |
| 1 | createNumberArray(double[] data) | createNumberArrayForArrayOfPositveDoubles() | All positive numbers | {1.0, 2.0, 3.0} | Returns | Fails | Yes |
| 2 | createNumberArray(double[] data) | createNumberArrayForArrayOfPositveDoublesandOneNegative() | All negative numbers | {-1.0, -2.0, -3.0} | Returns | Fails | Yes |
| 3 | createNumberArray(double[] data) | createNumberArrayForArrayOfNegativeDoubles() | Both positive and negative numbers | {-1.0, 2.0, -3.0} | Returns | Fails | Yes |
| 4 | createNumberArray(double[] data) | createNumberArrayForNullInput() | Null - Null Pointer Exception | empty array | Returns | Pass | No |
| 1 | createNumberArray2D(double[][] data) | createNumberArray2DForArrayOfPositveDoubles() | All positive numbers | {{1.0, 2.0, 3.0}, {1.0, 2.0, 3.0}} | Returns | Fails | Yes |
| 2 | createNumberArray2D(double[][] data) | createNumberArray2DForArrayOfPositveDoublesandOneNegative() | All negative numbers | {{-1.0, -2.0, -3.0}, {-1.0, -2.0, -3.0}} | Returns | Fails | Yes |
| 3 | createNumberArray2D(double[][] data) | createNumberArray2DForArrayOfNegativeDoubles() | Both positive and negative numbers | {{1.0,2.0,3.0} ,{-1.0,-2.0,-3.0}} | Returns | Fails | Yes |
| 4 | createNumberArray2D(double[][] data) | createNumberArray2DForNullInput() | Null - Null Pointer Exception | empty array | Returns | Pass | No |
| 1 | getCumulativePercentages(KeyedValues data) | getCumulativePercentagesForThreeValues() | Using three values | {0:5, 1:9, 2:2} | {0: 0.3125, 1:0.875, 2:1.0} | Fails | Yes |
| 2 | getCumulativePercentages(KeyedValues data) | getCumulativePercentagesForOneValueShouldReturnHundred() | Using one value | {0:1} | {0:1.0} | Fails | Yes |
| 3 | getCumulativePercentages(KeyedValues data) | getCumulativePercentagesForOnePositiveAndOneNegativeValues() | One positive, one negitive | {0:-5, 1:5} | {0:0.5, 1:0.5} | Fails | Yes |
| 4 | getCumulativePercentages(KeyedValues data) | getCumulativePercentagesForNullInput() | Null object | {0;5, 1:5} | throws exception | Pass | No |
| 5 | getCumulativePercentages(KeyedValues data) | getCumulativePercentagesForInvalidInput() | Invalid input | null | throws exception | Pass | No |

Figure 2: Test cases for DataUtilities class

The approach to test the *DataUtilities* class is to develop test cases for all possible scenarios for the inputs of each function. Each function’s test cases involved testing all positive numbers, all negative numbers, some positive and some negative numbers, and testing a null input. Mock objects were created for each scenario to test the unit functionality in question. The test cases for this class, including their function names are shown in Figure 2.

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

To begin the lab, the team worked together to set up JUnit in their respective Eclipse environments. Upon completion of this step, the specifications for the required classes to be tested were reviewed and test cases were developed. Once these test cases were developed, the team took the challenge of working on 5 functions each, with constant peer review of each other’s work to ensure that the test runners are implemented correctly.

# Difficulties encountered, challenges overcome, and lessons learned

The first difficulty encountered was developing the test cases. Since this was a black box approach to testing, we were not familiar with developing test cases for software without looking at the source code. Once we were able to become more familiar with the approach towards JUnit and black box testing, we were able to develop our test cases that allowed us to determine the flaws in several units in the data package.

The second challenge that the team encountered was writing efficient code with the JUnit testing framework. Due to the lack of familiarity with the framework, it required some research on both team members ends to learn more about the framework and how mock objects can be implemented. The examples provided in the lab were beneficial to learning this, but what was found most effective was researching the JUnit framework and documentation, and some things learned from doing this was implementation of test runners, fixtures, cases, executions, and assertions, along with creation of mock objects in Java.

# Comments/feedback on the lab itself

The lab had detailed steps that allowed us to set up the lab quite effectively. One thing to consider would be being more detailed in how the test functions should best be laid out, as the JUnit framework and creation of mock objects was still unclear from class. Overall, the team was able to understand how effective a unit testing framework can be to determining the success of a program, but some more examples of different JUnit testing styles could have helped us understand how to begin with the lab more effectively.