SENG 438 - Software Testing, Reliability, and Quality

Lab. Report #2 – Automated Requirements-Based API Unit Testing using JUnit

Group #:	12
Student Names:	Nurullo Boboev
	Trevor Brown
	Will Kerr
	Andrew Lattimer

Table of Contents

1 Introduction

The purpose of this lab is to introduce the fundamentals of automated unit testing using JUnit framework. We are to develop our own automated test code in JUnit frameworks to use and work with mock objects in a testing environment. This lab is to be done in three sections, where students are to first familiarize themselves on how to set up JUnit in Eclipse. Next we will do unit test generation based on javadoc requirements and develop test suites for two distinct classes, comprising of unit tests according to the javadocs. Finally, the tests will be executed on various versions of the SUT and collected and documented.

2 Detailed description of unit test strategy

Data Range class testing strategy:

For the Range class, the testing plan will loosely follow that of a weak-robust ECT, such that there will be a total of 6n+1 tests done, where n = 1 as there is 1 independent range variable, meaning there will be 7 total tests done. The values to be tested will include ranges: (BLB, NOM), (LB, NOM), (NOM, NOM), (NOM, AB), (NOM, AUB), (LB, UB), (BLB, ULB) where the meanings behind these symbols can be found under [3] on page 39. Nominal values will be arbitrarily chosen, and will remain inside the range of {min < NOM < max}.

Each test method will follow different boundary values/conditions such that all tests will be structured around them running as expected, and bugs regarding various boundary values will be uncovered as each test case is used.

DataUtilities class testing strategy:

The plan to test calculateColumnTotal and calculateRowTotal will be to use Weak Robust Testing. Weak Robust Testing as the best feasible solution as calculateColumnTotal and calculateRowTotal are responsible for summing all the values within a row or column. Since the maximum value in a cell is a double, and the total sum is stored in a double, it is easy to overflow the total sum. So it will be critical that we test all variables to their extremes. We decided to select Robustness Testing over WCT Robustness Testing, as with 3 variables there would be a total of 343 test cases which we did not consider feasible. Using the Robustness Testing formula 6n+1 where n is the number of variables being tested, there will be a total of 19 Test Cases per function. However we will be not testing when more than 1 variable is at their extreme. Here is how each test cases will test each variable, where each variable is in the order they were listed above: (NOM, NOM, NOM), (AUB, NOM, NOM), (ALB, NOM, NOM), (UB, NOM, NOM), (LB, NOM, NOM), (BLB, NOM, NOM), (BUB, NOM, NOM), (NOM, AUB, NOM), (NOM, ALB, NOM), (NOM, UB, NOM), (NOM, LB, NOM), (NOM, BLB, NOM), (NOM, BUB, NOM), (NOM, NOM, AUB), (NOM, NOM, UB), (NOM, NOM, BUB), (NOM, NOM, ALB), (NOM, NOM, LB), (NOM, NOM, BLB)

The plan to test getCumulativePercentages will be to use Weak Robust Testing. There is only one parameter which contains a list of Values with a key, index and value. For this function they only use the index and the value. Using the Robustness Testing formula 6n+1 where n is the number of variables being tested, there will be a total of 13 Test Cases per function. We did not use WCT over Robustness testing since WCT would require 49 tests which we did not consider necessary. However selecting Robustness Testing will not be testing more than 1 variable at their extremes. Here is how each test cases will test each variable, where each variable is in the order they were listed above: (NOM, NOM), (AUB, NOM), (ALB, NOM), (UB, NOM), (LB, NOM), (BLB, NOM), (BUB, NOM), (NOM, AUB), (NOM, ALB), (NOM, UB), (NOM, LB), (NOM, BLB), (NOM, BUB)

The plan to test the createNumberArray and createNumberArray2D methods will need to test a wide combination of inputs, as the inputs (double arrays) are complex within themselves. Zero, along with positive and integers will be tested, plus various decimal values will be placed in the array to test the functionality of these methods. In addition, large-input tests will be conducted to stress-test the methods. This will include large input array sizes, as well as large values within each array. Here is how each test case will be formatted: (AUB, NOM), (ALB, NOM), (UB, NOM), (LB, NOM), (BLB, NOM), (BUB, NOM), (NOM, NOM),(NOM, AUB), (NOM, ALB), (NOM, UB), (NOM, LB), (NOM, BLB), (NOM, BUB),

3 Test Cases developed

We will test the following methods: For the DataRange class, we dedicated to test: the contains, getLength, constrain, getLowerBound, and getUpperBound methods. For the DataUtilities class, we will be testing calculateColumnTotal, calculateRowTotal, createNumberArray, createNumberArray2D, and getCumulativePercentages methods.

For test cases involving methods getLowerBound and getUpperBound in the Range class, we will be following the weak-robust ECT testing strategy as described above, in which there are 7 possible test partitions. The actual range of x,y will be the maximum and minimum allowable signed double values, respectively. (-9,223,372,036,854,775,808.0, 9,223,372,036,854,775,807.0) It is expected that the following test cases / ranges will not succeed / output the correct values for either methods: (BLB, NOM), (NOM, ALB)

Test cases for methods constrain, contain, and getLength will use the following data ranges for each respective test: (LB, AB), (AB, AB)/(LB,LB),. For testing the constrain method and contain method, we will test for correctness on values that are NOM, AUB, LUB, AB,LB for each range on both x and y coordinates. Testing the getLength method will include only testing the data ranges. Overall these tests will ensure that the requirements are met for the described methods, however the tests leave small holes where bugs could lie. Had we done WCT, we may have been able to catch more bugs in regards to illegal inputs into the method, or perhaps some kind of valid input causing the method to error out.

For the calculateColumnTotal and calculateRowTotal, there will be 3 variables that we will be testing with these functions. First we will be testing the Row or Column that has been selected. The Upper Bound will be the greatest index of row or column that is available. The Lower Bound will be the lowest index of row or column that is available. The Nominal Value will be arbitrarily set to the Total Number of Rows or Columns divided by 2. Second we will also be testing the total number of rows or columns. While we would like to set the Upper Bound to be the maximum size of an integer, this caused the eclipse JUnit test to crash and would also take too long to complete the tests. So we will have to settle with the Upper Bound being an arbitrary value of 100 rows and columns. The Lower Bound will be 0 rows and columns. The Nominal value will be arbitrarily set to be 50 rows and columns. Lastly we will test the value stored within each cell. The Upper Bound will be the maximum size of a double. The Lower Bound will be the minimum size of a double. The Nominal Value will be arbitrarily set to 1000.

Since the Values2D is an interface, we will have to use JMocking to be able to instantiate the Values2D. Any interface that is taken as a parameter, you would not be able to know how the inherited classes may function, so mocking enables us to be able to return any values or throws that we want. The Drawback to Mocking is that we do not complete the entire Values2D object, and are missing data. So there may be overlapping errors that only occur when some other pieces of the data interact with data we did not mock.

For getCumulative Percentages, there will be 2 variables that we will be testing, The first variable will be the number of values in the KeyedValues. The Upper Bound for this will be arbitrarily set to 5 as if we set this to the maximum size of an integer, eclipse crashes. The Lower Bound will be 0. The Nominal Value will be set arbitrary to 3. The second variable will be the size of each value within each Value. The Upper Bound for this will be the maximum size of a Double. The Lower Bound for this would be the minimum size of a Double. The Nominal Value will be an arbitrary value between 0 and 100.

Since the KeyedValues is an interface, we will have to use JMocking to be able to instantiate the KeyedValues. Any interface that is taken as a parameter, you would not be able to know how the inherited classes may function, so mocking enables us to be able to return any values or throws that we want. The Drawback to Mocking is that we do not complete the entire KeyedValues object, and are missing data. So there may be overlapping errors that only occur when some other pieces of the data interact with data we did not mock.

For the createNumberArray and createNumberArray2d methods, since there are no boundaries on what array sizes are required or on the values that the array contains, the test cases will likely be based on WCT testing for two variables. In this case, the first variable will be the size of the array and the second variable(s) will be the values that are placed in the array.

The boundaries for the array sizing will range from size 1 to the max possible size of an integer, and for the array values will range from bounds of data that can fit within a double.

Extensive results and findings for each test case, as well as the test case methods justifications can be found under the appendix section of this document. [5]

4 How team work/effort was divided and managed

For the exploratory/ familiarization portion of this assignment, all 4 members met on the voice messaging app “Discord” and 1 member shared their screen. We followed steps 2.11-2.14. After this we all followed those steps again such that each member had the required jar files working in their project library. Following this, we chose the 10 methods for our test cases, and then divided it up amongst the 4 of us where we planned test strategies together, and then developed test cases for each method. We divided the functions amongst ourselves as follows:

Nurullo: equals, getLength, getCumulativePercentages

Trevor: getLowerBound, getUpperBound, constrain

Will: calculateColumnTotal, calculateRowTotal, getCumulativePercentages

Andrew: createNumberArray, createNumberArray2D

For the unit testing phase, we remained on the voice call, and each worked on various test cases, asking each other for help as needed. Following this, we equally contributed to writing this lab document, as well as sharing and documenting our findings.

5 Difficulties encountered, challenges overcome, and lessons learned

Some of the difficulties we encountered were we ran into a “java file not found exception” when initially trying to run the SUT demo classes, which we swiftly solved by including the jfreechart folder in the project file path. Another challenge we encountered was time management on this project as the week this assignment was released was a particularly busy week for all members of the team and as such we had to meet at various times both in the evenings and between classes. We also encountered difficulties regarding the workload as the functions we divided varied in complexity

6 Comments/feedback on the lab and lab document itself

We believe this lab could be improved by having even more specific instructions to follow in the set up section of 2.1-2.4 such that other teams do not run into the same issues that our group did as outlined in section 5 of this document. We appreciate being given some choice in this lab with regards to choosing some of the methods for 1 of the test classes, as it followed closer to a more real life scenario. Overall, this lab was very insightful and interesting as it gave us a stronger understanding of how unit testing works, as well as how to create our own unit tests.

7 Appendix

[1] Assignment 2 - via D2L: https://d2l.ucalgary.ca/d2l/le/content/354619/viewContent/4441644/View Author: Professor Behrouz Far

[2] Assignment 2 Repository via Github: https://github.com/seng438-master/seng438-a2 Author: ymbibalan on Github

[3] SENG 438-04: https://d2l.ucalgary.ca/d2l/le/content/354619/viewContent/4441595/View Author: Professor Behrouz Far

[4] Assignment 2 Team Repository https://github.com/trevorbrown18/-seng438-a2-trevorbrown18 Author(s): Nurullo Boboev, Trevor Brown, Will Kerr, Andrew Lattimer

[5] Unit Test Findings / Results:

Class Range:

Passed tests: of the 36 test scenarios we developed for the 5 functions we are testing, 29 of them ran correctly. The test SUIT for the range class and the 5 functions we chose ran with no errors. This means that theLowerBound method as well as getLength, and contain all works perfectly with their given unit tests. All but 1 test case for the constrain method also works correctly.

Bugs found: In total there were 7 bugs found. There were bugs found for ALL upperBounds test cases, in which the upper bound value for its given range did not match that of their expected result, meaning the function likely needs to be rewritten or modified such that it returns the second argument (the upper bounds) of the range object constructed. There was also a bug discovered for our constrainShouldReturnLowerValueForRange1 method, where in the range the upper bound being a positive value, and the lower bound is negative value, a negative input LOWER than the lowerBounds causes the constrain method to return 0, rather than the specified lowerbound in the range. This may be due to some kind of statement in the code which specifies some kind constraint when you have a range spanning from the negative bounds to positive.

Class DataUtilities:

Passed tests: of the 77 test scenarios we developed for the 5 functions we are testing, only 22 of them ran correctly.

Bugs found: There were a total of 10 bugs found. Bugs were found in all of the functions and some of these bugs were major errors that caused entire functions to fail all tests. This made it so that we had 55 failures out of 77 runs. These failures made it more difficult to locate bugs, as the major bugs often masked the minor bugs, and were often only able to be found upon closer inspection.

In both the createNumberArray and creatNumberArray methods, a bug was found where the last element of the newly-created array would contain a null value, instead of the last element of the array that was passed to the function. This caused 24 of a possible 26 tests to fail between the 2 functions.

In the calculateRowTotal there was a major error; they did not count the last element in the row, so the row total was wrong for all tests. This caused it so that all tests on this function failed. As well we had found 2 additional minor bugs in this function. In both the calculateRowTotal and calculateColumnTotal, they are able to sum values that are outside of the bounds of the Values2D if they are simulated there, instead it should not sum any values since it is outside of the range of the Values2D Array. As well as for both calculateRowTotal and calculateColumnTotal they do not throw InvalidParameterException for having the parameter data to be null.

In the getCumulativePercentages there was a major error; they do not include the value at index 0 for the cumulative sum. This makes it so that the percentages are out of the wrong amount, and that it is possible to have values greater than 1 because of it. This caused it so that all tests on this function returned a failed value. As well we had found 2 additional minor bugs in this function. We found that as the getCumulativePercentages does not throw the InvalidParameterException for having the parameter data as null. As well as the getCumulativePercentages when given a KeyedValue of Item Count 0, it gets a null pointer exception from it.