**SENG 438 - Software Testing, Reliability, and Quality**

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

|  |  |
| --- | --- |
| **Group #:** | **19** |
| Student Names: | Eric Renno |
| Ryan Sommerville |
| Quinn Ledingham |
| Kaumil Patel |

# Introduction

The purpose of this lab is to explore how to use Junit tests to thoroughly test the methods of a particular class in order to ensure that the class functions as it should. To do this, the Junit class allows you to create a test function and compares the correct result with the result that the function actually returns. Although the process is straightforward, accurately testing a class requires a thorough analysis of boundary cases and other necessary test cases for each method. Additionally,

# Detailed description of unit test strategy

// including the input partitions, you have designed

The test was performed using the Eclipse IDE. First, java files were created for the test methods using a shortcut on Eclipse. This will create a method in the test classes for each method to be tested in the classes to be tested. Next, test cases were invented for each method so that boundary cases are covered. Once this is done, the test cases are coded. If necessary, a Mocker class is used as a substitute for inputs where we don’t have access to the actual class. After all the test cases are coded, the tests are run and the results recorded.

The benefit of using mocking is that limits the dependencies of a test. It allows for you to test a class independently even when it depends on another class. This makes the scope of the test smaller. A drawback is that the tests that use mocking are made for a specific implementation of the methods. This might cause you to have to also change the tests when you change the code. If you do not, it might keep passing tests that it should not be.

# Test cases developed

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Class | Method | Test Function | Expected Result/ Actual Result | Passes |
| DataUtilities | calculateColumnTotal(Values2D data, int column) | calculateColumnTotalForTwoValues()  Inputs: Mock Values2D object with 1.25 and 2.5. | 3.5 | Pass |
|  |  | calculateColumnTotalForNegativeValues()  Inputs: Mock Values2D object with 1.25, 2.5 and -5.5. | -1.75 | Pass |
|  |  | calculateColumnTotalForOneValues()  Inputs: Mock Values2D object with two columns, the second with 1.25. | 1.25 | Pass |
|  |  | calculateColumnTotalForZeroValues()  Inputs: Mock Values2D object with no values. | 0 | Pass |
|  | clone (double [][] source) | cloneFor1By1ArrayTest()  Inputs: {{10.5}} | Output is a clone of the input and is not a reference to the input. | Fail |
|  |  | cloneFor1By10ArrayTest()  Inputs: Random 1 x 10 array. | Output is a clone of the input and is not a reference to the input. | Fail |
|  |  | cloneFor10By1ArrayTest()  Input: 10 by 1 Array with random numbers. | Output is a clone of the input and is not a reference to the input. | Fail |
|  |  | cloneFor10By10ArrayTest()  Input: 10 by 10 array with random numbers | Output is a clone of the input and is not a reference to the input. | Fail |
|  |  | notCloneFor10By10ArrayTest()  Input: 10 by 10 Array | Output is a clone of input, and doesn’t equal a different array. | Fail |
|  | createNumberArray(double[] data) | createNumberArrayForNull()  Inputs: NULL | Exception | Pass |
|  |  | createNumberArrayOfSize1()  Inputs: double [] = {10.5} | Number [], {10.5} | Fail |
|  |  | createNumberArrayOfSize10()  Inputs: double[10] with random values | Number [10] with random values | Fail |
|  | createNumberArray2D( double[][] data) | createNumberArray2DForNULL() Input: Null | Exception | Pass |
|  |  | createNumberArray2DFor1By1Array()  Input: double [][] with 1 value | Number [][] array with the same value | Fail |
|  |  | createNumberArray2DFor1By10Array()  Input: double[][] that is 1 by 10 with random values | Number [][] array with the same length and values. | Fail |
|  |  | createNumberArray2DFor10By1Array()  Input: double[][] that is 10 by 1 with random values. | Number [][] array with the same length and values. | Fail |
|  |  | createNumberArray2DFor10By10Array()  Input: double[][] that is 10 by 10 with random values. | Number [][] array with the same length and values. | Fail |
|  | equal(double[][] a, double[][] b) | equalFor1By1ArrayTest()  Input: Two 1 by 1 arrays with identical values. | True | Fail |
|  |  | equalFor1By10ArrayTest()  Input: Two 1 by 10 arrays with identical values. | True | Fail |
|  |  | equalFor10By1ArrayTest()  Input: Two 10 by 1 arrays with identical values. | True | Fail |
|  |  | equalFor10By10ArrayTest()  Input: Two 10 by 10 arrays with identical values. | True | Fail |
|  |  | notEqualFor10By10ArrayTest()  Input: Two 10 by 10 arrays with different values. | False | Fail |
|  | getCumulativePercentages( KeyedValues data) | getCumulativePercentagesForThreeValues()  Inputs: Mock KeyedValues object with values 2, 1, and 2. | New values should be 0.4, 0.6, 1.0 | Fail |
|  |  | getCumulativePercentagesForZeroValues()  Inputs: Mock KeyedValues object with no values. | Empty KeyedValues object | Pass |
| Range | combine(Range range1, Range range2) | combineTestIntersect()  Input: range1: 0 to 10  range2: 5 to 15. | range.lower = 0  range.upper = 15 |  |
|  |  | combineTestNoOverlap()  Input: range1: 0 to 10  range2: 15 to 20 | range.lower = 0  range.upper = 20 |  |
|  |  | combineTestNull()  Input: range1 = NULL  range2: 0 to 10 | range.lower = 0  range.upper = 10 |  |
|  | getLowerBound() | getLowerBoundTest()  Input: Range = -10 to 10 | -10 |  |
|  | getUpperBound() | getUpperBoundTest()  Input: Range = -10 to 10 | 10 |  |
|  | constrain(double value) | constrainTestInsideRange()  Input: value = 3, Range = 2 to 7 | 3 |  |
|  |  | constrainTestOutsideRangeAbove()  Input: value = 8, Range = 2 to 7 | 7 |  |
|  |  | constrainTestOutsideRangeBelow()  Input: value = 1, Range = 2 to 7 | 2 |  |
|  |  | constrainTestOnLower()  Input: value = 2, Range = 2 to 7 | 2 |  |
|  |  | constrainTestOnUpper()  Input: value = 7, Range = 2 to 7 | 7 |  |

// write down the name of the test methods and classes. Organize the based on the source code method // they test. identify which tests cover which partitions you have explained in the test strategy section //above

# How the teamwork/effort was divided and managed

Ryan Sommerville: Formatted and put together most of the report.

Eric Renno:

Quinn Ledingham: Created test cases and implemented them in JUnit

Kaumil Patel:

# Difficulties encountered, challenges overcome, and lessons learned

In the beginning, we had some trouble with designing the tests. Initially coming up with the test plan caused a bit of difficulty because we were not sure what exactly that consisted of. But after researching it a little it made more sense, and we knew where to begin. Designing the test cases was also at first a challenge because we did not know exactly what they should look like. After reading the slides again, however, we had a better idea of how to make them. JMock also caused a few problems just getting it set up and learning how to use it. Now we know how to get started designing a test plan and test cases and we know how to use JMock now.

# Comments/feedback on the lab itself

The instructions on how to set up the eclipse project were simple to follow and made it easy to get started working on the lab.