SENG 438 - Software Testing, Reliability, and Quality

Lab. Report #3 – Code Coverage, Adequacy Criteria and Test Case Correlation

Group #: 3

Student Names:

Haniya Ahmed

Apostolos Scondrianis

Beau McCartney

Josh Vanderstoop

(Note that some labs require individual reports while others require one report
for each group. Please see each lab document for details.)

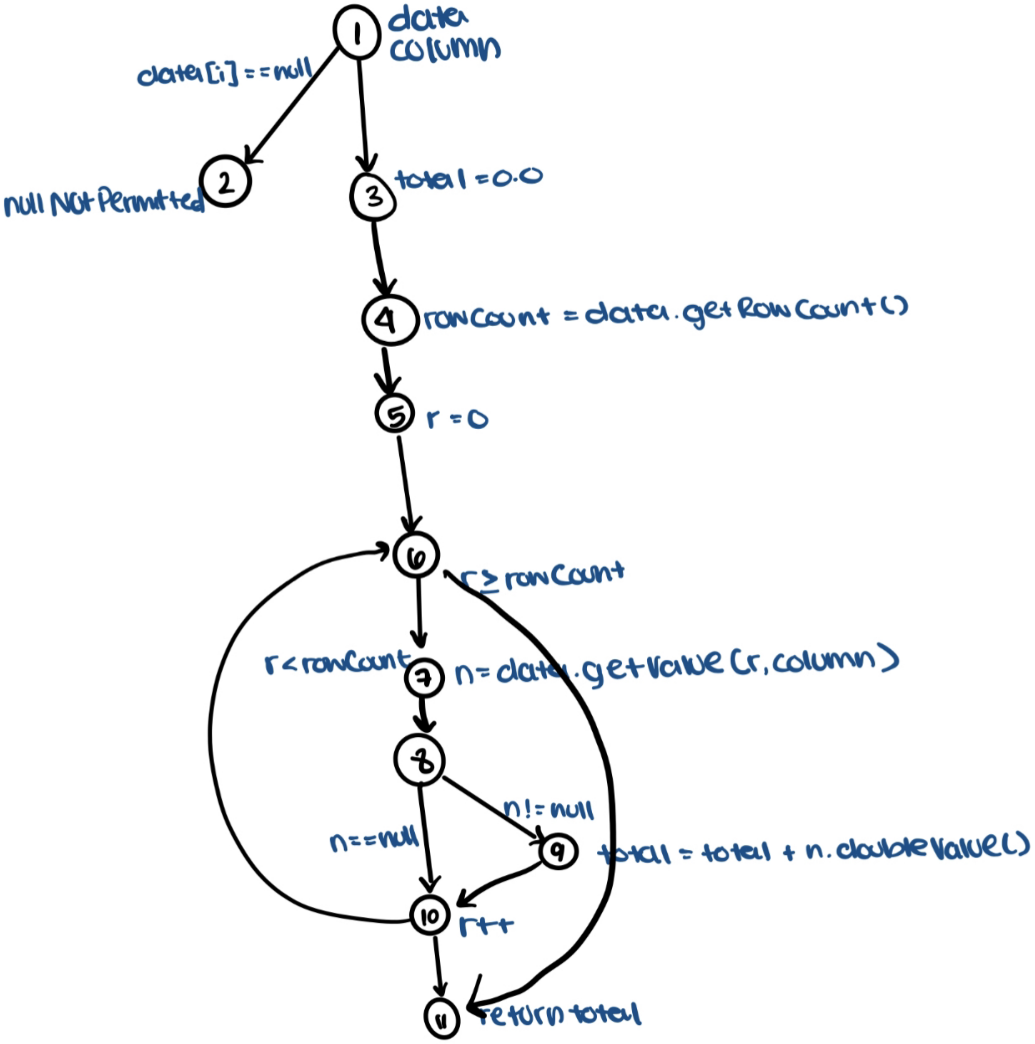
# 1 Introduction

In this assignment we are building on the previous results from Assignment 2 that involved blackbox testing. We are exploring a different methodology for testing called white-box testing. The goal is to see how white-box metrics such as branch, statement and condition coverage can be used to evaluate the quality of a test suite that is used on a System Under Testing and what further test cases can be developed based on these results. This will be done by using JUnit in Eclipse. Part of the lab also includes manual analysis of data-flow coverage. The objective of the lab is to become familiar with code coverage tools, designing test cases to improve code coverage, understanding the benefits and drawbacks of coverage-based test generation, and gain a better understanding of manual data-flow coverage.

# 2 Manual data-flow coverage calculations for calculateColumnTotal and Shift methods

DataUtilities method: calculateColumnTotal

- Data flow graph



- Def-use sets per statement

Statement	Def	Use
1	DEF(1)= {data, column}	USE(1) =NA
2	DEF(2)= NA	USE(2) = {data}
3	DEF(3)= {total}	USE(3) = NA
4	DEF(4)= {rowCount}	USE(4) = {data}
5	DEF(5)= {r}	USE(5) = NA
6	DEF(6)= NA	USE(6) = {r, rowCount}
7	DEF(7)= {n}	USE(7) = {data, r, column}

8	DEF(8)= NA	USE(8) = {n}
9	DEF(9)= {total}	USE(9) = {n, total}
10	DEF(10)={r}	USE(10) = {r}
11	DEF(11)=NA	USE(11) = {total}

- DU-pairs

Variable(v)	Defined at (n)	dcu(v,n)	dpu(v,n)
data	1	{3, 7}	{ (2, 12) }
column	1	{7}	{}
total	3	{9, 11}	{}
total	9	{9, 11}	{}
rowCount	4	{}	{(6,7), (6, 11)}
r	5	{10}	{(6,7), (6, 11)}
r	10	{10}	{(6,7), (6, 11)}
n	7	{9}	{(8,9), (8, 10)}

Pairs:

  - data: (1, 3) , (1, 7),   (2, 12)[in the case of null data]
  - column: (1,7), (1)
  - total: (3, 9, 11)
  - rowCount: (4, 7), (4, 11)
  - r: (5, 10),   (5, 11), (5, 7)
  - n: (7, 9)   (7, 10)

- Show which pairs covered in each test case
  - calculateColumnTotalForTwoValues()
    - r(5,10), r(5,7), r(5,11), n(7, 9), data(1,3), data(1,7), column(1,7), total(3,9,11), rowCount(4, 7), rowCount(4, 11)
  - calculateColumnTotalForThreeValues()
    - r(5,10) , r(5,7), r(5,11), n(7, 9), data(1,3), data(1,7), column(1,6), total(3,9,11), rowCount(3, 6), rowCount(4, 11)
  - calculateColumnTotalForThreeNegativeValues()
    - r(5,10) , r(5,7), r(5,11), n(7, 9), data(1,3), data(1,7), column(1,6), total(3,9,11), rowCount(3, 6), rowCount(4, 11)

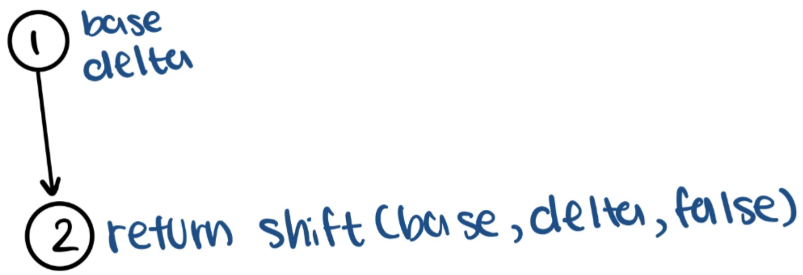
- **Calculate DU-Pair coverage**

  Coverage $= (CU_c + PU_c) / ((CU + PU) - (CU_f + PU_f))$

$= (5 + 4) / ((7 + 7) - (0 + 2))$
$=$ **75%**

Range method: Shift

- <u>Data flow graph</u>



- <u>Def-use sets per statement</u>

Statement	Def	Use
1	DEF(1)= {base, delta}	USE(1) =NA
2	DEF(2)= NA	USE(2) = {base, delta}

- <u>DU-pairs</u>

Variable (v)	Defined at (n)	dcu(v, n)	dpu(v, n)
base	1	{2}	{ }
delta	1	{2}	{ }

Pairs:
- base: (1,2)
- delta: (1,2)

- <u>Show which pairs covered in each test case</u>
  - negativeShiftValueLowerBoundCheck() : base(1,2), delta(1,2)
  - negativeShiftValueUpperBoundCheck() : base(1,2), delta(1,2)
  - positiveShiftValueLowerBoundCheck() : base(1,2), delta(1,2)
  - positiveShiftValueUpperBoundCheck() : base(1,2), delta(1,2)
  - zeroShiftValueLowerBoundCheck() : base(1,2), delta(1,2)
  - zeroShiftValueUpperBoundCheck() : base(1,2), delta(1,2)
  - nullRangeShiftCheck() : NA
  - maximumPositiveShiftUpperBoundaryCheck(): base(1,2), delta(1,2)

- **Calculate DU-Pair coverage**

Coverage $= (CU_c + PU_c) / ((CU + PU) - (CU_f + PU_f))$
$= (2 + 0) / ((2 + 0) - (0 + 0))$
$=$ **100%**

Through manual coverage testing for these two methods, we have reached a conclusion that matches the results of the automated coverage tools.

# 3 A detailed description of the testing strategy for the new unit test

The coverage-based testing metrics we chose were: statement, branch, and condition/method. Prior to creating new tests, we tested the coverage of our unit tests to determine where there would be room for improvement. For the Range class, the methods we tested were:

- toString()
- getUpperBound()
- getLowerBound()
- getLength()
- shift(Range, double)

After analyzing our test suite, we found that our statement coverage for all of these methods was 100%, and therefore did not require more unit tests for statement coverage. None of our tested methods had branches in their code; therefore, branch coverage was inapplicable. Our condition coverage also reached 100% for all the tested methods and therefore required no improvement.

For the DataUtilities class, the methods we tested were:

- createNumberArray2D(double[][])
- calculateRowTotal(Values2D, int)
- calculateColumnTotal(Values2D, int)
- equal(double[][], double[][])
- calculateRowTotal(Values2D, int, int[])

When using a coverage tool to determine the statement coverage of our tests, we found that 100% of all the tested methods were covered, except for equal(double[][], double[][]), which was only at 80%. Some lines were missed due to “if” branches that had implementation within them. Since the if conditions were not met, the implementation inside them was also skipped. When testing branch coverage, the calculateRowTotal(Values2D, int, int[]) and calculateColumnTotal(Values2D, int, int[]) methods were the only ones below 70%, both only having 66.7% of their branches covered. During the condition coverage, all tested methods reached 100%. Therefore the methods that we have to create new unit tests for are equal(double[][], double[][]), calculateRowTotal(Values2D, int, int[]), and calculateColumnTotal(Values2D, int, int[]). Our goal is to increase the statement coverage of equal(double[][], double[][]) to 90% or above by checking what lines in the code were not covered by our test cases, and increasing the branch coverage of calculateRowTotal(Values2D, int, int[]) and calculateColumnTotal(Values2D, int, int[]) to 70% or above by determining what branches were not covered by our test cases. For the equal(double[][], double[][]) method) we need to test the following branches to increase test coverage:

A: if (b == null)

B: if (a.length != b.length)

For the calculateRowTotal(Values2D, int, int[]), we need to test the following branches to increase test coverage:

A: if (col < colCount)
B: if (n != null)

For the calculateColumnTotal(Values2D, int, int[]) , we need to test the following branches to increase test coverage:

A: if (row < rowCount)
B: if (n != null)

4 A high level description of five selected test cases you have designed using coverage information, and how they have increased code coverage

The only method in the class DataUtilities for which we did not manage to achieve over 90% statement coverage in was the following method:

- equal(double[][], double[][]).

The statements that were not covered for method equal(double[][], double[][]) were two return statements after the following branches:

A: if (b == null)
B: if (a.length != b.length)

Each of these “if” statements had a return statement within their scope. Since the conditions within the “if” statements were never fulfilled, the statements within their scope were also never reached. As such, we wrote two additional tests for the method:

- checkNotNullwithNullArrays()
    - This method tests equal(double[][], double[][]) with the first parameter being a not null 2D double array, and the second parameter being a null array. This should execute the return statement within branch A above.
- checkNotEqualLengthArrays()
    - This method tests equal(double[][], double[][]) with two valid 2D double arrays that are of different lengths. This should execute the return statement within branch B above.

This increased the statement coverage from 80% to 100%, and even increased the branch coverage from 83.3% to 100%.

The only methods in the class DataUtilities for which we did not manage to achieve over 70% branch coverage were the following methods:

- calculateRowTotal(Values2D, int, int[])
- calculateColumnTotal(Values2D, int, int[])

The branches that were not covered for method calculateRowTotal(Values2D, int, int[]) were:
A: if (col < colCount)
B: if (n != null)

In order to cover these branches, we developed two additional tests:

- calculateColumnTotalForThreeRowsOneInvalidRow()
    - This method tests calculateRowTotal(Values2D, int, int[]) by trying to calculate for an invalid row to reach branch A described above.
- calculateColumnTotalForThreeRowsOneNullValue()
    - This method tests calculateRowTotal(Values2D, int, int[]) by trying to calculate for a Values2D object with a row containing a null value to reach branch B described above.

The branches that weren't covered for method calculateColumnTotal(Values2D, int, int[]) were:
A: if (row < rowCount)
B: if (n != null)

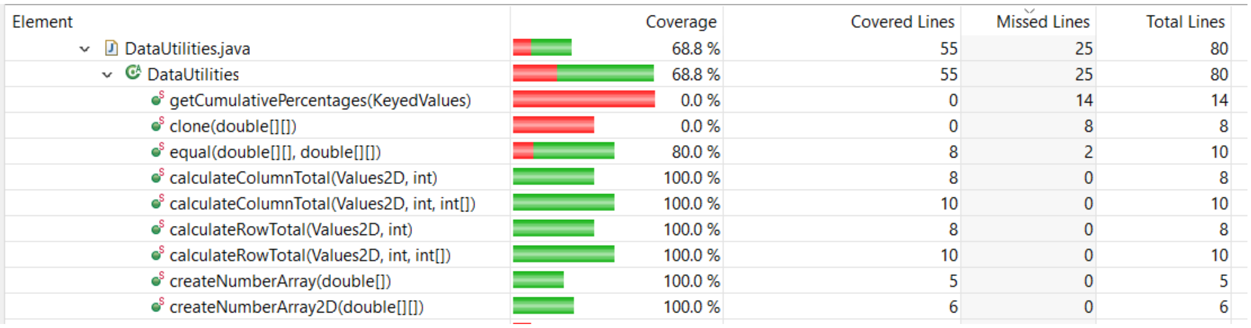
In order to cover these branches, we wrote four additional tests.

- calculateRowTotalForFourColumnsOneInvalidColumn()
    - This method tests calculateColumnTotal(Values2D, int, int[]) by trying to calculate for an invalid column to reach branch A described above.
- calculateRowTotalForFourColumnsOneNullValue()
    - This method tests calculateColumnTotal(Values2D, int, int[]) by trying to calculate for a Values2D object with a column containing a null value to reach branch B described above.

Our condition coverage increased from 66.7% to 100% for both calculateRowTotal(Values2D, int, int[]) and calculateColumnTotal(Values2D, int, int[]).

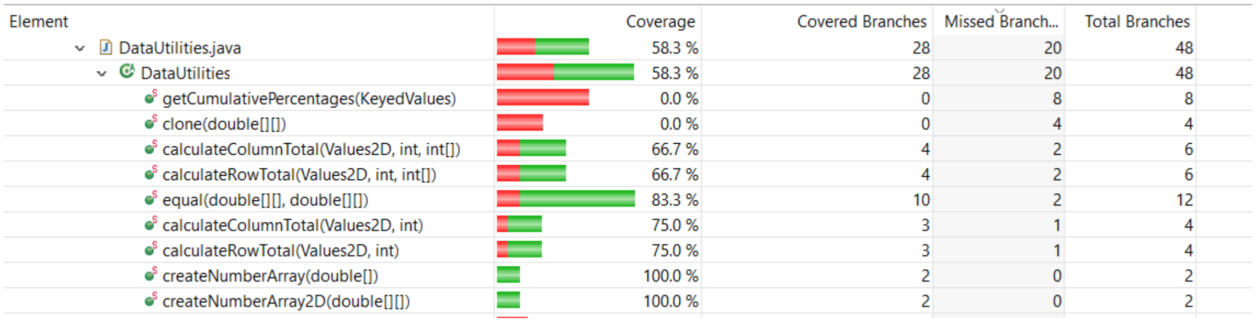
# 5 A detailed report of the coverage achieved of each class and method (a screen shot from the code cover results in green and red color would suffice)

DataUtilities Before:
Statement Coverage



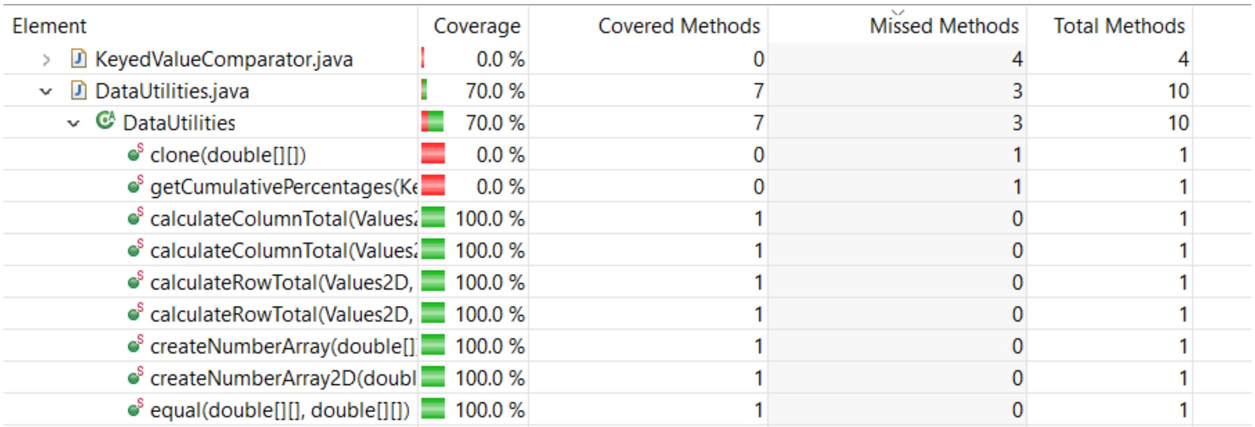
Element	Coverage	Covered Lines	Missed Lines	Total Lines
DataUtilities.java	68.8 %	55	25	80
DataUtilities	68.8 %	55	25	80
getCumulativePercentages(KeyedValues)	0.0 %	0	14	14
clone(double[][])	0.0 %	0	8	8
equal(double[][], double[][])	80.0 %	8	2	10
calculateColumnTotal(Values2D, int)	100.0 %	8	0	8
calculateColumnTotal(Values2D, int, int[])	100.0 %	10	0	10
calculateRowTotal(Values2D, int)	100.0 %	8	0	8
calculateRowTotal(Values2D, int, int[])	100.0 %	10	0	10
createNumberArray(double[])	100.0 %	5	0	5
createNumberArray2D(double[][])	100.0 %	6	0	6

Branch Coverage



Element	Coverage	Covered Branches	Missed Branch...	Total Branches
DataUtilities.java	58.3 %	28	20	48
DataUtilities	58.3 %	28	20	48
getCumulativePercentages(KeyedValues)	0.0 %	0	8	8
clone(double[][])	0.0 %	0	4	4
calculateColumnTotal(Values2D, int, int[])	66.7 %	4	2	6
calculateRowTotal(Values2D, int, int[])	66.7 %	4	2	6
equal(double[][], double[][])	83.3 %	10	2	12
calculateColumnTotal(Values2D, int)	75.0 %	3	1	4
calculateRowTotal(Values2D, int)	75.0 %	3	1	4
createNumberArray(double[])	100.0 %	2	0	2
createNumberArray2D(double[][])	100.0 %	2	0	2

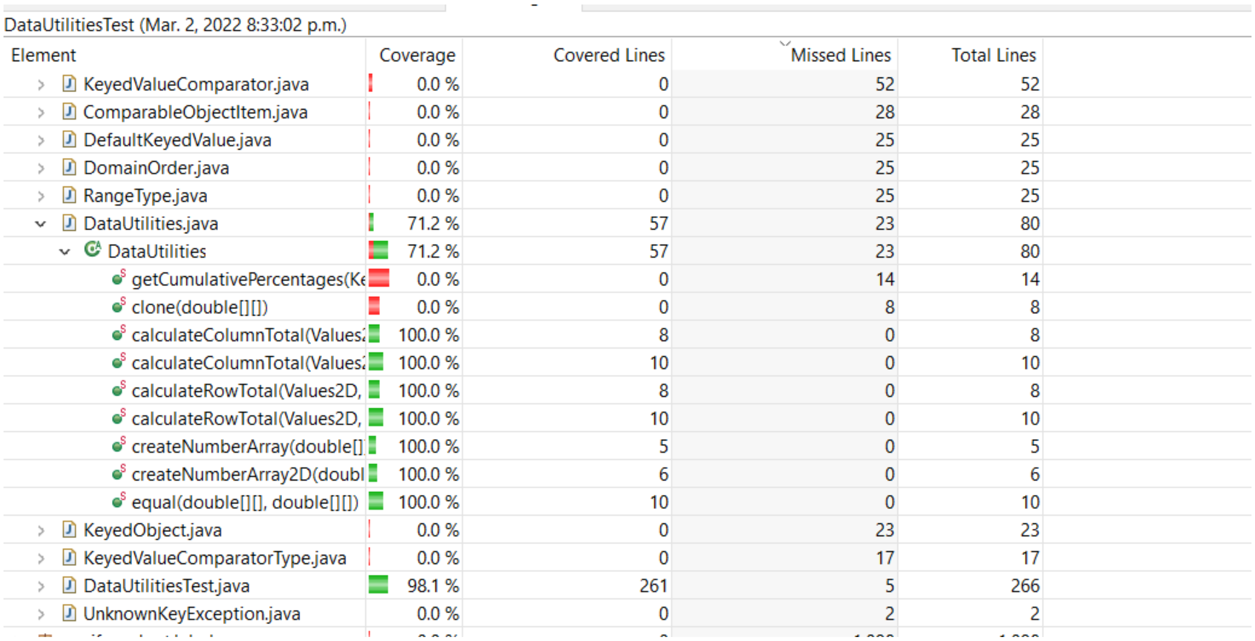
Condition Coverage



Element	Coverage	Covered Methods	Missed Methods	Total Methods
KeyedValueComparator.java	0.0 %	0	4	4
DataUtilities.java	70.0 %	7	3	10
DataUtilities	70.0 %	7	3	10
clone(double[][])	0.0 %	0	1	1
getCumulativePercentages(Ke	0.0 %	0	1	1
calculateColumnTotal(Values2	100.0 %	1	0	1
calculateColumnTotal(Values2	100.0 %	1	0	1
calculateRowTotal(Values2D,	100.0 %	1	0	1
calculateRowTotal(Values2D,	100.0 %	1	0	1
createNumberArray(double[]	100.0 %	1	0	1
createNumberArray2D(doubl	100.0 %	1	0	1
equal(double[][], double[][])	100.0 %	1	0	1

DataUtilities After:

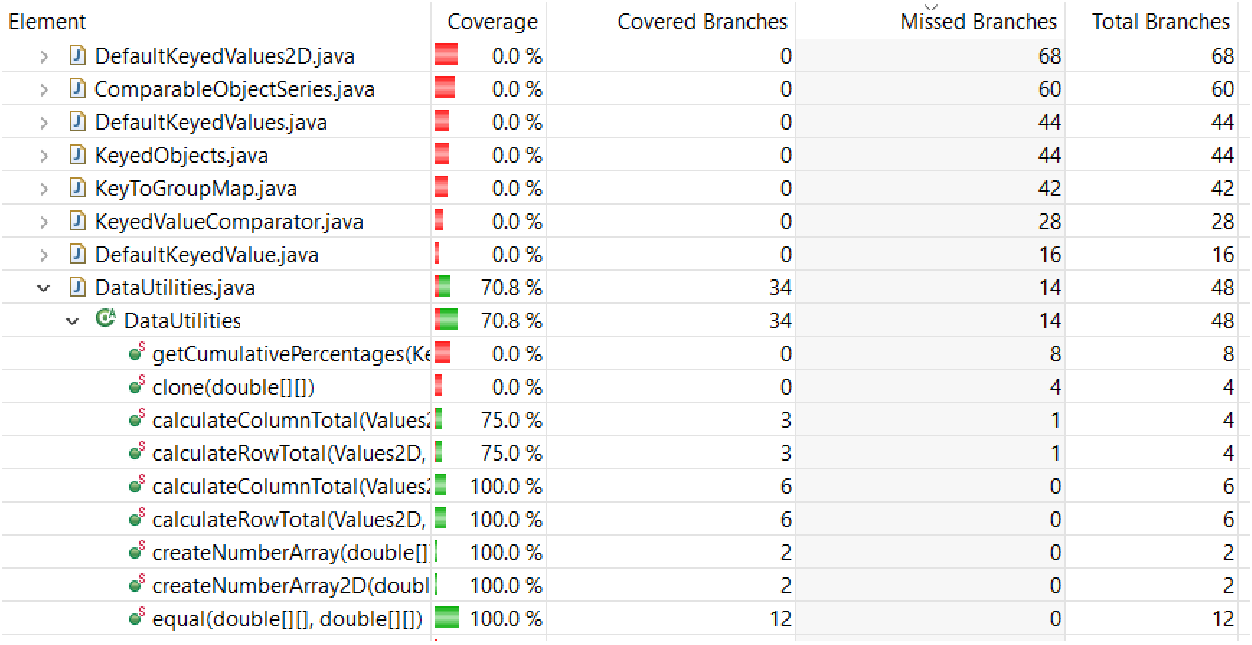
Statement Coverage



DataUtilitiesTest (Mar. 2, 2022 8:33:02 p.m.)

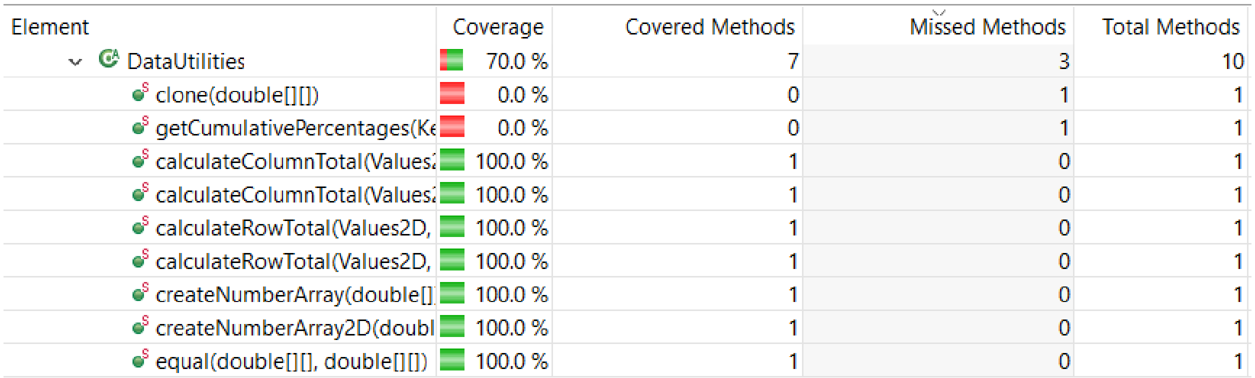
Element	Coverage	Covered Lines	Missed Lines	Total Lines
KeyedValueComparator.java	0.0 %	0	52	52
ComparableObjectItem.java	0.0 %	0	28	28
DefaultKeyedValue.java	0.0 %	0	25	25
DomainOrder.java	0.0 %	0	25	25
RangeType.java	0.0 %	0	25	25
DataUtilities.java	71.2 %	57	23	80
DataUtilities	71.2 %	57	23	80
getCumulativePercentages(Ke	0.0 %	0	14	14
clone(double[][])	0.0 %	0	8	8
calculateColumnTotal(Values:	100.0 %	8	0	8
calculateColumnTotal(Values:	100.0 %	10	0	10
calculateRowTotal(Values2D,	100.0 %	8	0	8
calculateRowTotal(Values2D,	100.0 %	10	0	10
createNumberArray(double[]	100.0 %	5	0	5
createNumberArray2D(doubl	100.0 %	6	0	6
equal(double[][], double[][])	100.0 %	10	0	10
KeyedObject.java	0.0 %	0	23	23
KeyedValueComparatorType.java	0.0 %	0	17	17
DataUtilitiesTest.java	98.1 %	261	5	266
UnknownKeyException.java	0.0 %	0	2	2

Branch Coverage



Element	Coverage	Covered Branches	Missed Branches	Total Branches
DefaultKeyedValues2D.java	0.0 %	0	68	68
ComparableObjectSeries.java	0.0 %	0	60	60
DefaultKeyedValues.java	0.0 %	0	44	44
KeyedObjects.java	0.0 %	0	44	44
KeyToGroupMap.java	0.0 %	0	42	42
KeyedValueComparator.java	0.0 %	0	28	28
DefaultKeyedValue.java	0.0 %	0	16	16
DataUtilities.java	70.8 %	34	14	48
DataUtilities	70.8 %	34	14	48
getCumulativePercentages(Ke	0.0 %	0	8	8
clone(double[][])	0.0 %	0	4	4
calculateColumnTotal(Values:	75.0 %	3	1	4
calculateRowTotal(Values2D,	75.0 %	3	1	4
calculateColumnTotal(Values:	100.0 %	6	0	6
calculateRowTotal(Values2D,	100.0 %	6	0	6
createNumberArray(double[]	100.0 %	2	0	2
createNumberArray2D(doubl	100.0 %	2	0	2
equal(double[][], double[][])	100.0 %	12	0	12

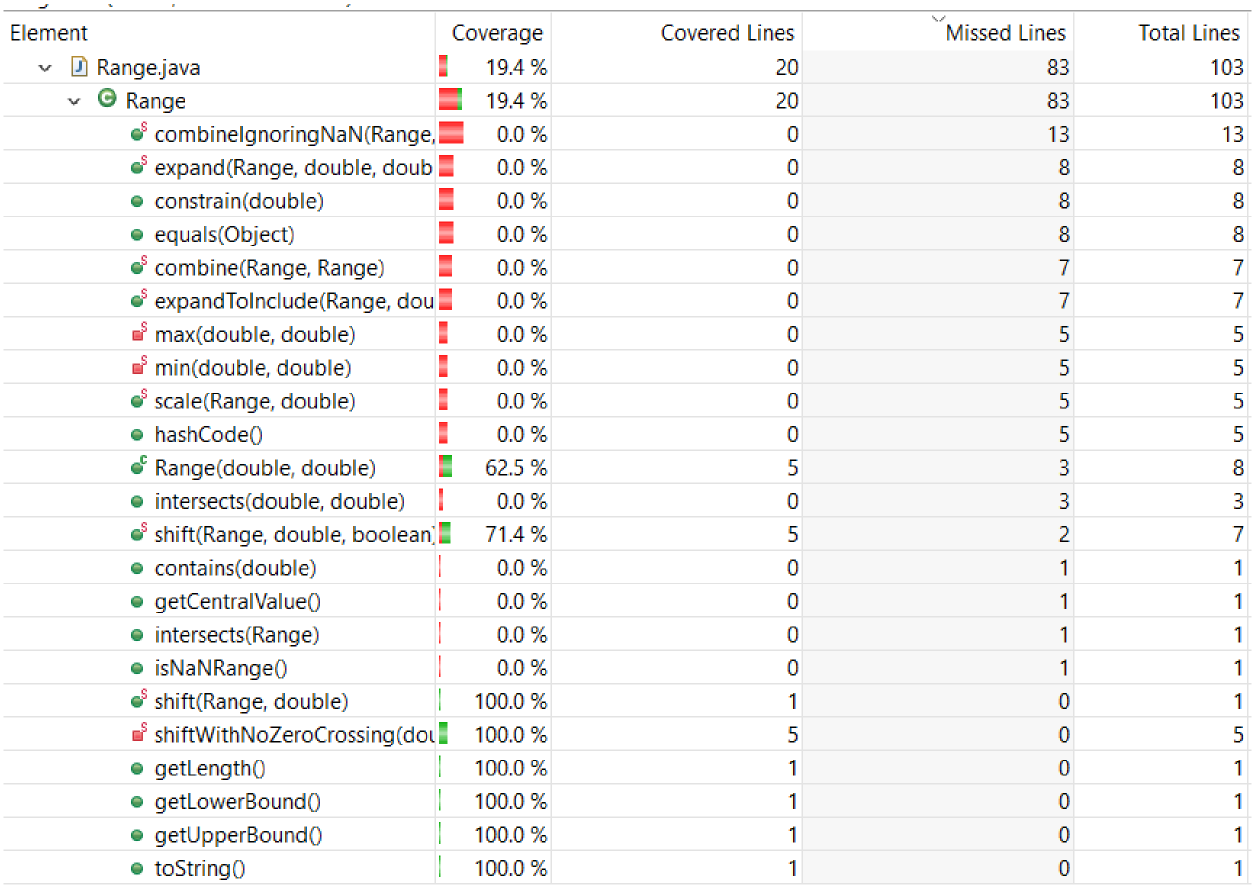
Condition Coverage



Element	Coverage	Covered Methods	Missed Methods	Total Methods
DataUtilities	70.0 %	7	3	10
clone(double[][])	0.0 %	0	1	1
getCumulativePercentages(Ke	0.0 %	0	1	1
calculateColumnTotal(Values2	100.0 %	1	0	1
calculateColumnTotal(Values2	100.0 %	1	0	1
calculateRowTotal(Values2D,	100.0 %	1	0	1
calculateRowTotal(Values2D,	100.0 %	1	0	1
createNumberArray(double[]	100.0 %	1	0	1
createNumberArray2D(doubl	100.0 %	1	0	1
equal(double[][], double[][])	100.0 %	1	0	1

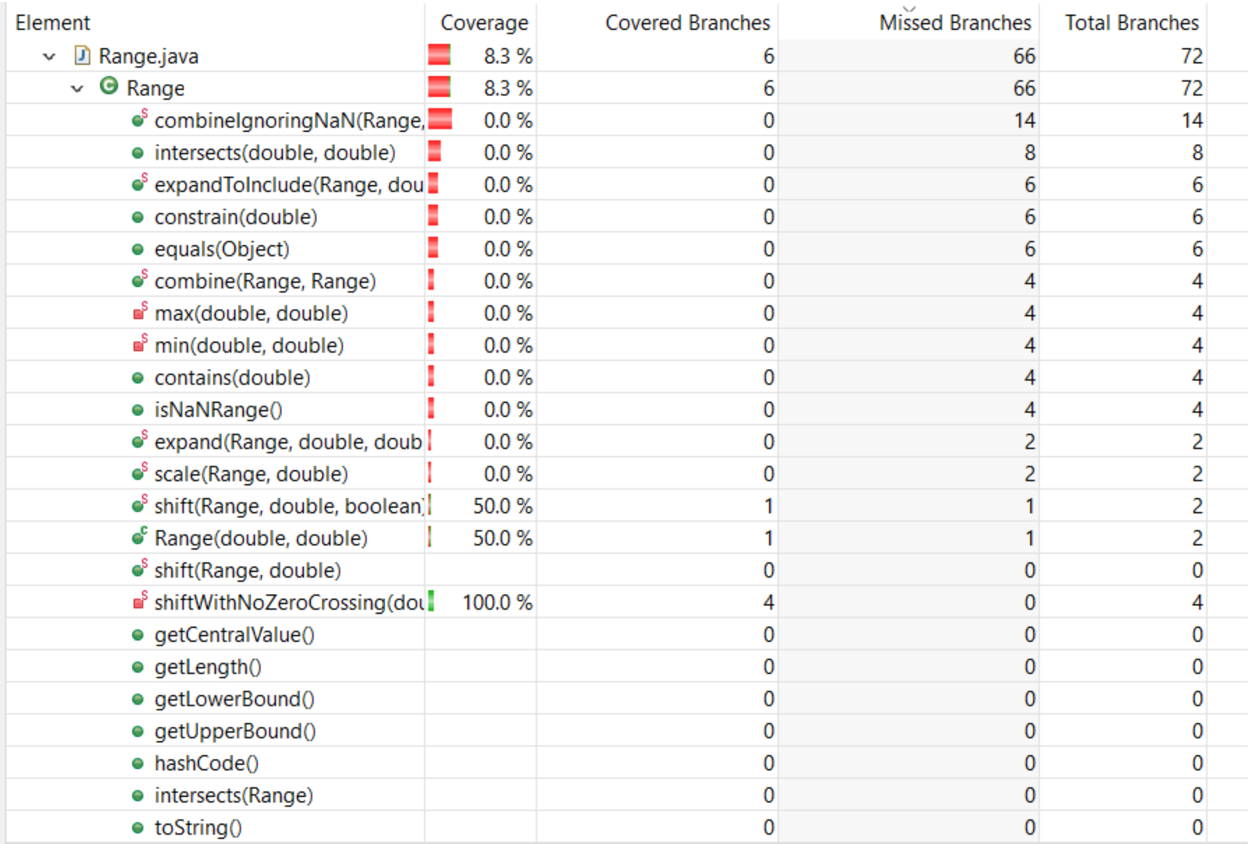
Range Before & After (No Changes Made):

Statement Coverage



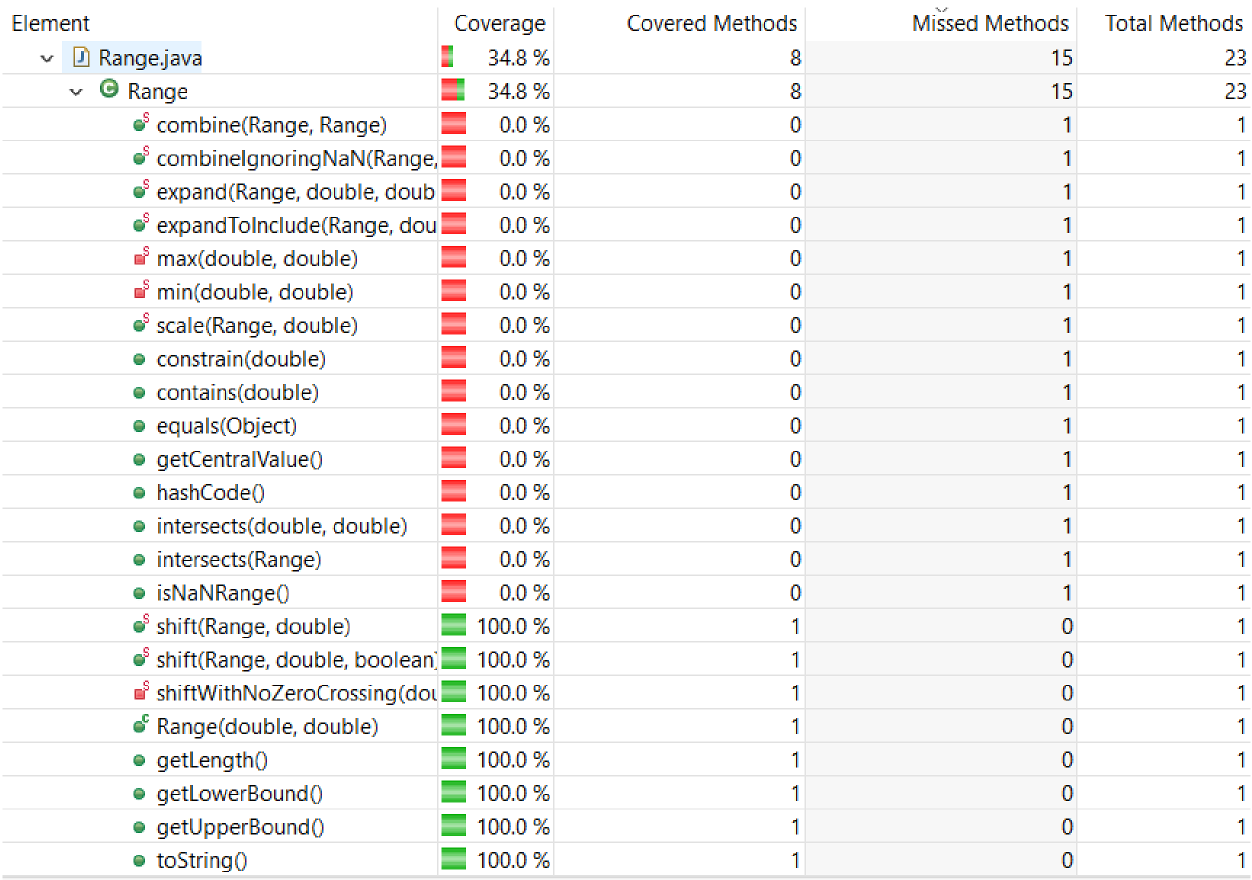
Element	Coverage	Covered Lines	Missed Lines	Total Lines
Range.java	19.4 %	20	83	103
Range	19.4 %	20	83	103
combineIgnoringNaN(Range,	0.0 %	0	13	13
expand(Range, double, doub	0.0 %	0	8	8
constrain(double)	0.0 %	0	8	8
equals(Object)	0.0 %	0	8	8
combine(Range, Range)	0.0 %	0	7	7
expandToInclude(Range, dou	0.0 %	0	7	7
max(double, double)	0.0 %	0	5	5
min(double, double)	0.0 %	0	5	5
scale(Range, double)	0.0 %	0	5	5
hashCode()	0.0 %	0	5	5
Range(double, double)	62.5 %	5	3	8
intersects(double, double)	0.0 %	0	3	3
shift(Range, double, boolean	71.4 %	5	2	7
contains(double)	0.0 %	0	1	1
getCentralValue()	0.0 %	0	1	1
intersects(Range)	0.0 %	0	1	1
isNaNRange()	0.0 %	0	1	1
shift(Range, double)	100.0 %	1	0	1
shiftWithNoZeroCrossing(dou	100.0 %	5	0	5
getLength()	100.0 %	1	0	1
getLowerBound()	100.0 %	1	0	1
getUpperBound()	100.0 %	1	0	1
toString()	100.0 %	1	0	1

Branch Coverage



Element	Coverage	Covered Branches	Missed Branches	Total Branches
Range.java	8.3 %	6	66	72
Range	8.3 %	6	66	72
combineIgnoringNaN(Range,	0.0 %	0	14	14
intersects(double, double)	0.0 %	0	8	8
expandToInclude(Range, dou	0.0 %	0	6	6
constrain(double)	0.0 %	0	6	6
equals(Object)	0.0 %	0	6	6
combine(Range, Range)	0.0 %	0	4	4
max(double, double)	0.0 %	0	4	4
min(double, double)	0.0 %	0	4	4
contains(double)	0.0 %	0	4	4
isNaNRange()	0.0 %	0	4	4
expand(Range, double, doub	0.0 %	0	2	2
scale(Range, double)	0.0 %	0	2	2
shift(Range, double, boolean)	50.0 %	1	1	2
Range(double, double)	50.0 %	1	1	2
shift(Range, double)		0	0	0
shiftWithNoZeroCrossing(dou	100.0 %	4	0	4
getCentralValue()		0	0	0
getLength()		0	0	0
getLowerBound()		0	0	0
getUpperBound()		0	0	0
hashCode()		0	0	0
intersects(Range)		0	0	0
toString()		0	0	0

Condition Coverage



Element	Coverage	Covered Methods	Missed Methods	Total Methods
Range.java	34.8 %	8	15	23
Range	34.8 %	8	15	23
combine(Range, Range)	0.0 %	0	1	1
combineIgnoringNaN(Range,	0.0 %	0	1	1
expand(Range, double, doub	0.0 %	0	1	1
expandToInclude(Range, dou	0.0 %	0	1	1
max(double, double)	0.0 %	0	1	1
min(double, double)	0.0 %	0	1	1
scale(Range, double)	0.0 %	0	1	1
constrain(double)	0.0 %	0	1	1
contains(double)	0.0 %	0	1	1
equals(Object)	0.0 %	0	1	1
getCentralValue()	0.0 %	0	1	1
hashCode()	0.0 %	0	1	1
intersects(double, double)	0.0 %	0	1	1
intersects(Range)	0.0 %	0	1	1
isNaNRange()	0.0 %	0	1	1
shift(Range, double)	100.0 %	1	0	1
shift(Range, double, boolean)	100.0 %	1	0	1
shiftWithNoZeroCrossing(dou	100.0 %	1	0	1
Range(double, double)	100.0 %	1	0	1
getLength()	100.0 %	1	0	1
getLowerBound()	100.0 %	1	0	1
getUpperBound()	100.0 %	1	0	1
toString()	100.0 %	1	0	1

# 6 Pros and Cons of coverage tools used and Metrics you report

The coverage tool that we used was EclEmma. We found the tool to be quite advantageous since it was a built-in Eclipse tool and did not require separate installation, and was easily accessible within Eclipse itself. Furthermore, it allowed us to check different metrics, beyond statement, branch, and condition coverage. The con, however, of this coverage tool was that, although it had several metric options for a user, the metric option used to determine condition coverage, as demonstrated by the TA for this lab, was method coverage. Seeing as method coverage is actually a measure of whether a method was covered or not, it is not actually an accurate measure of condition coverage. Although in this particular case, the methods in the two tested classes did not have more than one conditional clause at a time, in other cases, this form of measure would not prove accurate.

# 7 A comparison on the advantages and disadvantages of requirements-based test generation and coverage-based test generation.

Requirements based test generation generally produces a sufficient level of system testing; however, the developers must be very thorough in their testing strategy, and must comb the documentation several times to find points of failure. This is beneficial since it provides a very in depth analysis of the code as well documentation accuracy; however, the time spent conceptualizing the tests could be better spent elsewhere. Nonetheless, it does ensure the specifications of a code are met and required functionality is present and functioning. However, this testing strategy does not provide coverage testing, and since developers cannot see the source code itself they may miss computational or predicate uses of certain variables.

Coverage based test generation allows confidence about code coverage to the developers through coverage testing tools. With direct access to the source code, developers are able to better design test cases that ensure all statements, branches, conditions, or other important coverage metrics, are met. This ensures that the code has expected functionality, but is limited as it is not able to detect required, missing functionality.

When testing requirements, developers can use a combination of requirement-based test generation, or black-box testing, and cover-based test generation, or white-box testing, to ensure that both missing and unexpected functionality can be detected. This allows the advantage of not only ensuring accurate implementation of code through white-box testing, but also that required functionalities are present. This allows the developer to be confident about minimal failure rate. Although more time consuming, a combined form of testing allows for a more methodological approach and greater confidence in the software implementation and specification.

8 A discussion on how the team work/effort was divided and managed

We made sure that the team met multiple times so we were able to understand the assignment requirements. We were pretty happy to see that our work on assignment 2 resulted in a nearly sufficient amount of coverage. Beau, Haniya and Josh wrote a test case each, and Apostolos did three test cases. We all equally worked on the lab report and the analysis of the results. Josh and Haniya did the diagrams and Josh worked on the manual analysis, while Beau and Apostolos commented on how our tests altered the coverage of our test suite. We made sure that all of us were well informed and understood each other’s work, and completed the final report together.

9 Any difficulties encountered, challenges overcome, and lessons learned from performing the lab

We had a couple issues with synchronizing our work with eclipse and github, but that was expected since we all run different operating systems on our machines and that can be a bit challenging. We learned how different measures of coverage can give a better idea about how well a test suite is designed, and possibly allow us to reevaluate it to account for cases that we might have not thought about. On assignments going forward, we will continue to iron out little things and our team will continue to work more efficiently.

10 Comments/feedback on the lab itself

Haniya - This lab was helpful in acquiring a better understanding of the difference between balck-box testing and white-box testing, and the limitations each of them have. Although white-box testing gives direct access to the source code and therefore allows confidence in test coverage, it is not able to accurately test whether all requirements or specifications for the code were met. It was definitely interesting to also see how our manual calculations matched up with the test coverage tool calculations as well and allowed for new appreciation for tools that make these tasks much easier than if they were to be done manually.

Josh - This lab was helpful in the use of manual as well as automated coverage calculations, and shows the usefulness of the tools that are available. The time it takes to generate the same results manually is considerably higher than using the tools, although they yield the same results in the end.

Apostolos - I found this lab very interesting. It brought a lot of the knowledge that we were taught in class together. I really liked that it built on our previous work and it allowed us to understand the importance of testing on a line per line basis, and how white box testing is affected by branches.

Beau - For me, I found the approach of using coverage tools on our black box tests very instructive. It showed that documentation alone can’t express every possible case that may need to be tested, and strengthened my understanding of where white box testing is more detailed than black box testing.