SENG 438 - Lab 3

Manpreet Singh
Muhammad Shakeel
Tianfan Zhou
Girimer Singh

March 3, 2022

Introduction

In lab2, we did black box testing with some methods of DataUtilities class and Range class within JFree Chart. We used some testing technologies such as equivalence class testing and boundary testing to test the system. In previous testing, we mainly focused on testing the requirements since we do not know the internal structure of the system. However, only performing black box testing is not good enough. In this lab, we have access to see the actual implementation of the system, which enables us to also perform white box testing to the system to analyze the parts of the code uncovered by our black-box testing. In the first part of this lab, we will try to test the coverage rate (statement, branch, method) of our assignment 2 test suite and record the result as a base line. In part 2, we will perform some manual data flow coverage calculation with method Range.Combine and DataUtilities.calculateColumnTotal to have a better understanding of control flow coverage. In part 3, we will write some more test cases based on white box testing technologies to enhance the coverage rate of our testing. We aim to increase statement, branch and method coverage to be over 90%, 70% and 60% respectively. Both black box testing and white box testing is important because they can be complementary to each other. Combining using both testing methods can result in more reliable and comprehensive test cases.

Manual data-flow coverage calculations for X and Y methods

Range Class

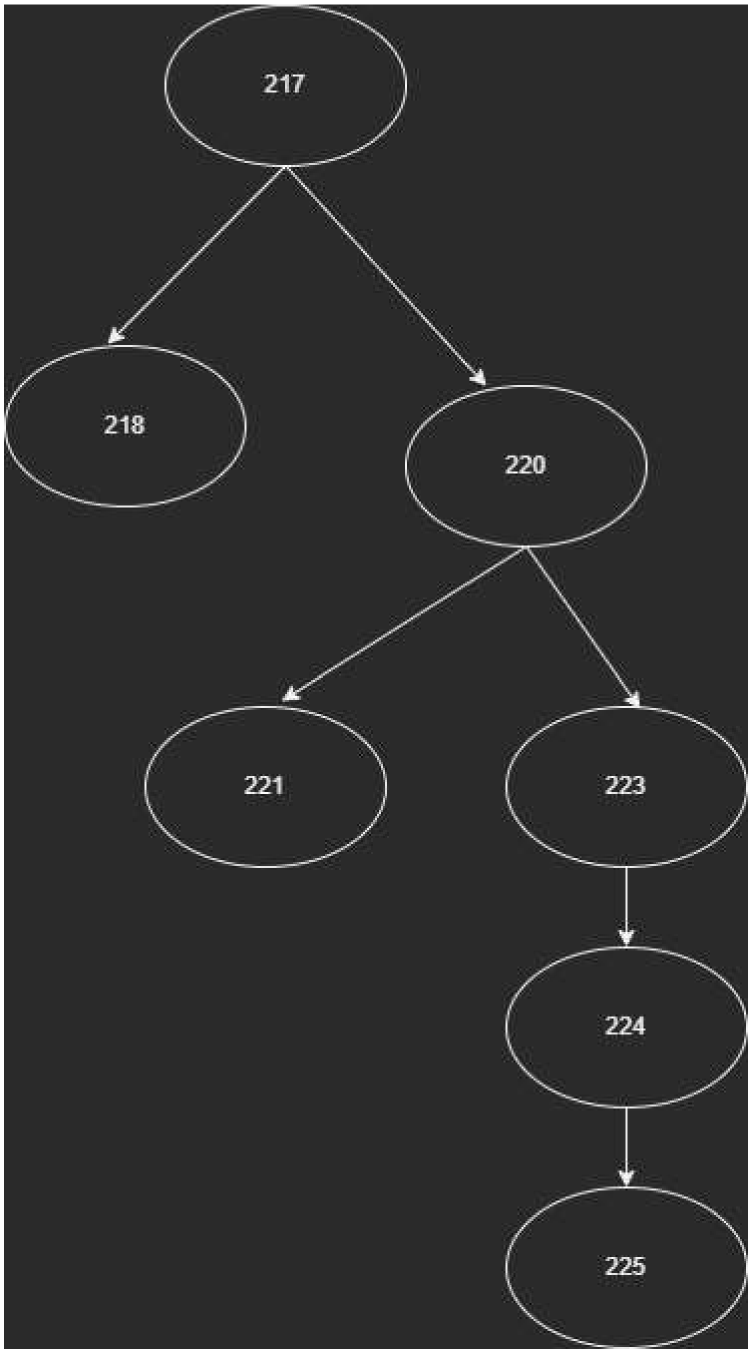


Figure 1- Method: Range.Combine Diagram

Statement	def	use
216	range1, range2	
217		range1
218		range2
220		range2
221		range1
223	l	range1, range2
224	u	range1, range2
225		l, u

Figure 2- def and use for statements

Variable	DU-Pair
l	(223, 225)
u	(224, 225)
range1	(216,217), (216, 221), (216, 223), (216,224)
range2	(216,218), (216, 220), (216, 223), (216,224)

Figure 3- DU-Pair for each variable

Test	(223, 225)	(224, 225)	(216,217)	(216,218)	(216,221)	(216,220)	(216,223)	(216,224)
testFirstRangeNullForMethodCombine	No	No	Yes	Yes	No	No	No	No
testSecondRangeNullForMethodCombine	No	No	No	No	Yes	Yes	No	No
testBothNullRangeForMethodCombine	No	No	Yes	Yes	Yes	Yes	No	No
testRangesForMethodCombine	Yes	Yes	Yes	No	No	Yes	Yes	Yes

$DU\ Pair\ Coverage\ = \frac{Number\ of\ decision\ outcomes\ exercised}{Total\ number\ of\ decisions\ outcomes}\ x\ 100\%\ = \frac{10}{10}\ = 100\%$

Figure 4- DU Pair Coverage

Data Utilities Class

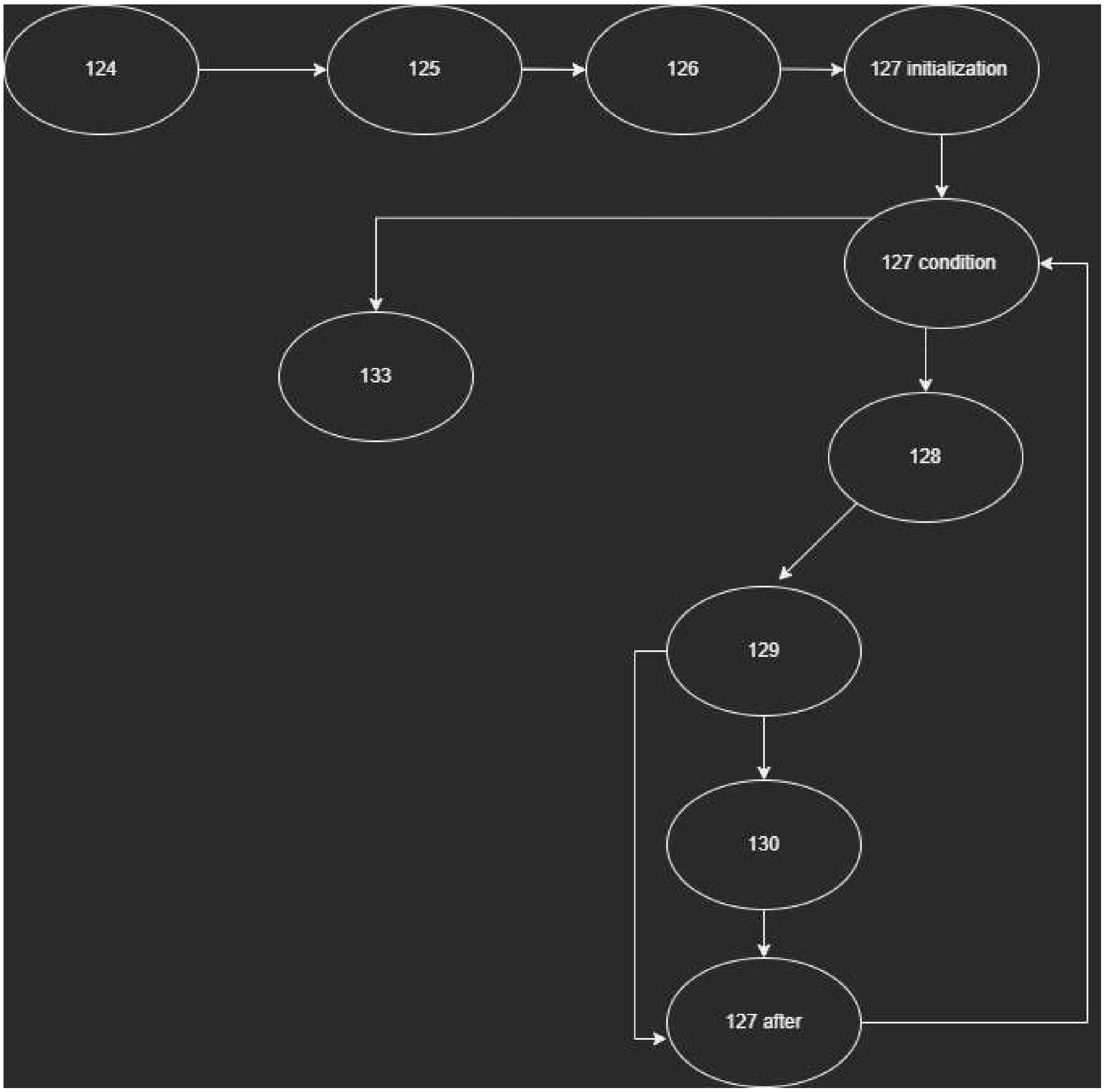


Figure 5- Method: DataUtilities.calculateColumnTotal Diagram

Statement	def	use
123	Data, column	
124		data
125	total	
126	rowCount	data
127 (init)	r	
127 (condition)		r, rowCount
128	n	r,data,column
129		
130	total	n, total
127 (after)	r	r
130		total

Figure 6- def and use for statements

Variable	DU- Pairs
total	(125, 130) (125, 133) (130, 133)
rowCount	(126, 127 condition)
r	(127 initialization, 127 condition) (127 after, 127 condition) (127 initialization, 127 after) (127 initialization, 128) (127 after, 128)
n	(128, 130)
data	(123, 124),(123,126),(123,128)
column	(123, 128)

Figure 7- DU-Pair for each variable

	passingNullAsParameter()	intBoundaryValues()	positiveNegativeColumnValues()	invalidFirstInputTest()	invalidSecondInputTest()	NullColumnValueTest()
(125, 130)	No	Yes	Yes	No	No	No
(125, 133)	No	Yes	Yes	Yes	Yes	Yes
(130, 133)	No	Yes	Yes	No	No	No
(126, 127 condition)	No	Yes	Yes	Yes	Yes	Yes
(127 initialization, 127 condition)	No	Yes	Yes	Yes	Yes	Yes
(127 after, 127 condition)	No	Yes	Yes	Yes	Yes	Yes
(127 initialization, 127 after)	No	Yes	Yes	Yes	Yes	Yes
(127 initialization, 128)	No	Yes	Yes	Yes	Yes	Yes
(127 after, 128)	No	Yes	Yes	Yes	Yes	Yes
(128, 130)	No	Yes	Yes	No	No	No
(123, 124)	Yes	Yes	Yes	Yes	Yes	Yes
(123,126)	No	Yes	Yes	Yes	Yes	Yes
(123,128)	No	Yes	Yes	Yes	Yes	Yes

$$DU\ Pair\ Coverage\ = \frac{Number\ of\ decision\ outcomes\ exercised}{Total\ number\ of\ decisions\ outcomes}\ x\ 100\%\ = \frac{14}{14}\ = 100\%$$

Figure 8- DU Pair Coverage

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

In this lab, our goal was to increase the code coverage for the Range class and for the Data Utilities class by creating the new unit tests. So, to increase the code coverage, we used white box testing technique. Initially, when we started the code coverage test on Range Class and Data Utilities class using the test cases that were created using the black box testing, we found that the statement coverage of the Range class was at about 20% , Branch coverage was at 16% and the Condition Coverage was at 25%. Similarly, for the Data Utilities class, the statement coverage was at 52%, Branch Coverage was at 42% and Condition Coverage was 60%.
To increase the Statement coverage of both the Range Class and the Data Utilities class, we created the test cases that can cover all the statements of the Methods of the Range class and the Data Utilities Class. After implementing the new test cases, the statement coverage for the Range Class and for the Data Utilities class was increased to 93% and 100% respectively. To increase the Branch coverage, we wrote the new test cases as such it covers the most the branches in the Range class and Data Utilities class. We also made DU pairs in some of the methods of the Range and Data Utilities class in order to increase the Branch coverage. After creating and implementing the new test cases, the Branch coverage was increased to 82% for the Range class and was increased to 91.7% for the Data Utilities class. To increase the Condition Coverage for both the Range and Data Utilities Class, We used MC/DC(Modified Condition-Decision Criterion). This criteria helped in increasing the Condition Coverage of Range class to 100% and 100% for the Data Utilities class.

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

The high level descriptions of the five test cases is described as follows:

Test 1: testPositiveMarginsForMethodExpandTest in ExpandTest for the Range Class
This test method checks whether the expected output of the expanded range object is equal to actual output when the positive values of lower and upper margin are provided

as arguments in the expand method. This test method helps in increasing the statement coverage for the Range class by covering all the lines in the expand method of the Range class.

Test 2: stmtCoverageTestForMethodHashCode in hashCodeTest for the Range Class

This test method checks if the output1(hashCode) of range object r1 is equal to output2(hashCode) of the range object r1. This test method increases the statement coverage for the Range class by approximately 5%.

Test 3: testFirstRangeNullForMethodCombine in CombineTest for the Range Class

This test method checks if the range1(null) and range2(not null) is provided as arguments in the combine method and whether the expected output(range2) is equal to actual output obtained from the combine method. This test method increases the branch coverage for the Range Class since it covers one of the branches in the combine method of the range class.

Test 4: BranchCoverageTestForConstructor in constructorTest for the Range Class

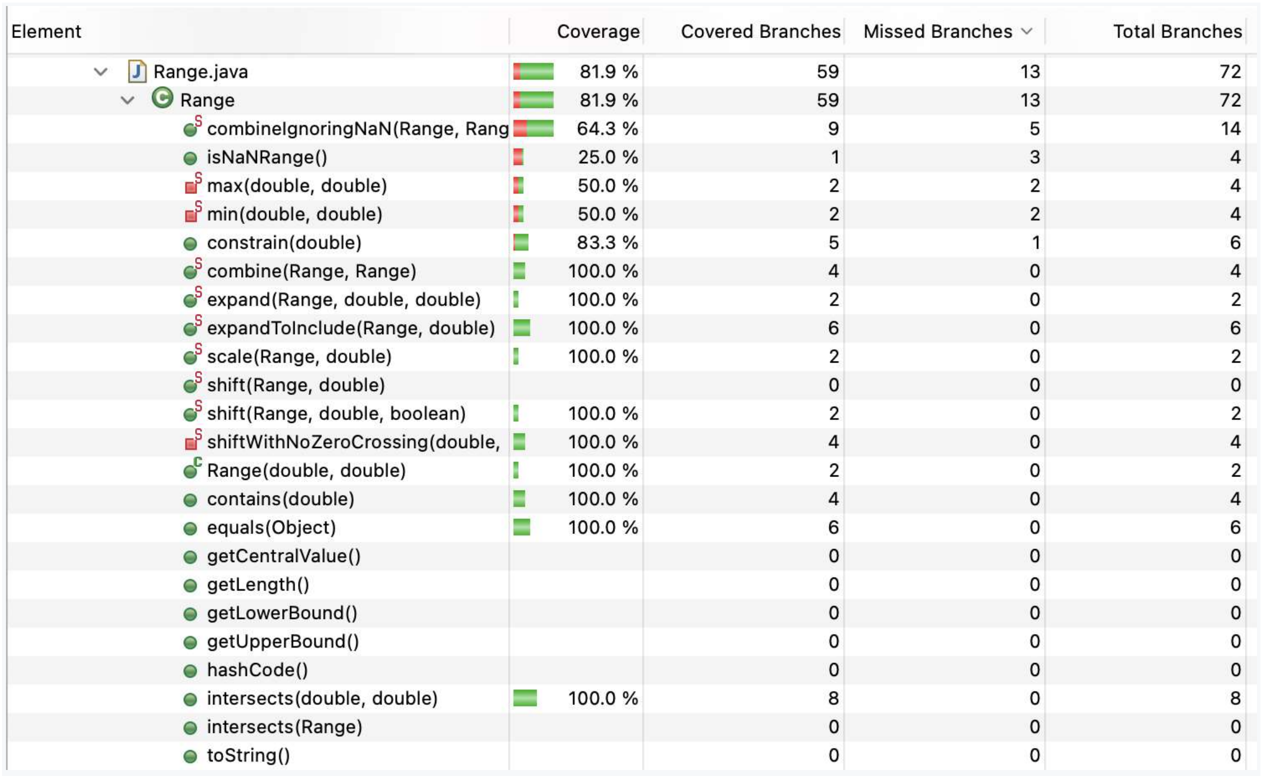
This test method checks whether the constructor throws an illegalArgumentException if the lower limit of the Range Object is greater than the upper limit of the Range Object. This test method increased the branch coverage for the Range Class by approximately 6% as it covers the missing branch in the Range Class Constructor.

Test5: testingStringObject() in EqualsTest for the Range Class

This test Method checks whether a String object is equal to a Range object. This test increases both the branch coverage and the condition coverage. It covers one of the branches in the equals method and also covers decision expression in the equals method and increases the condition coverage by approximately 4% for the Range class.

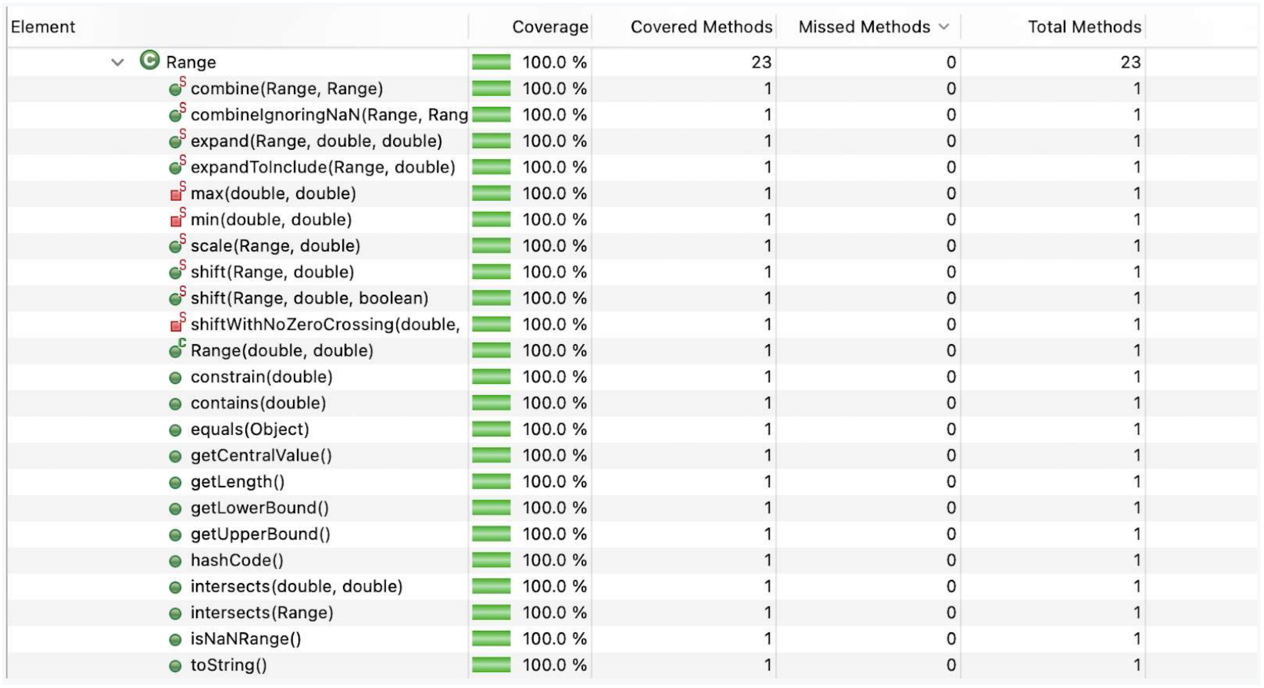
A detailed report of the coverage achieved of each class and method (a screen shot from the code cover results in green and red color would suffice)

Range Class:



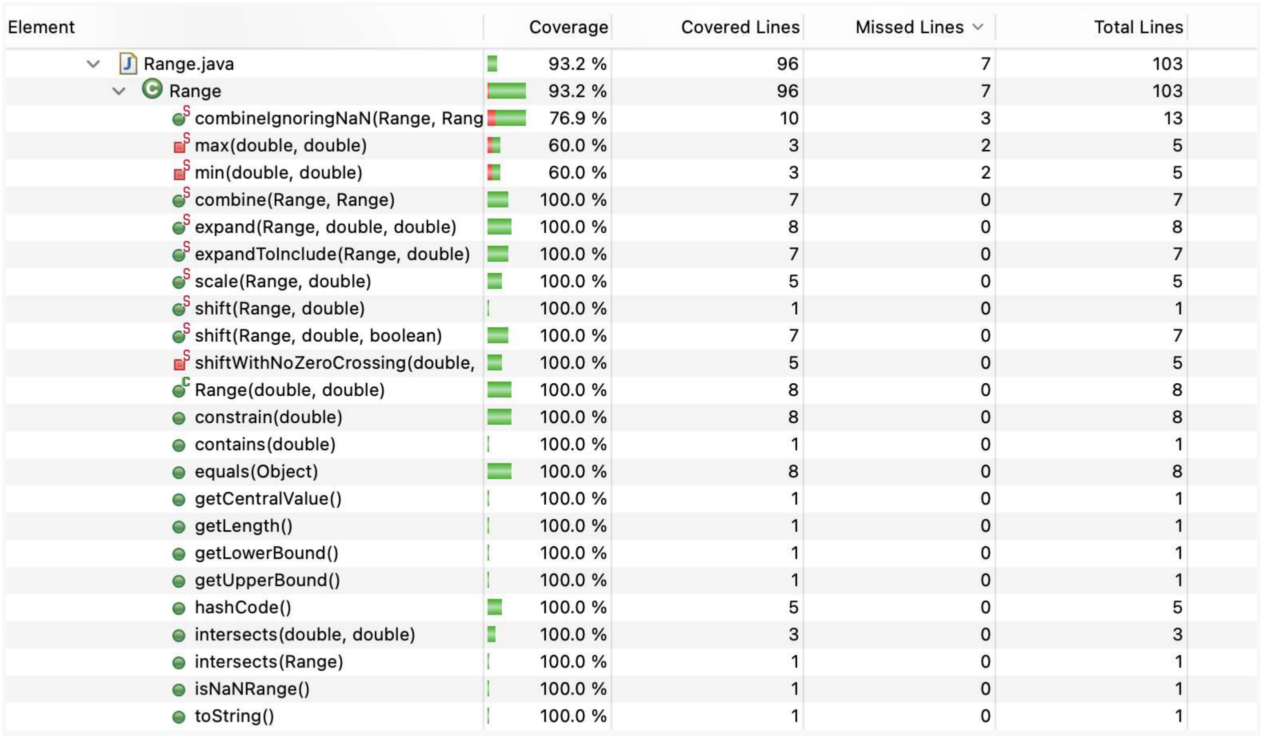
Element	Coverage	Covered Branches	Missed Branches	Total Branches
Range.java	81.9 %	59	13	72
Range	81.9 %	59	13	72
combineIgnoringNaN(Range, Rang	64.3 %	9	5	14
isNaNRange()	25.0 %	1	3	4
max(double, double)	50.0 %	2	2	4
min(double, double)	50.0 %	2	2	4
constrain(double)	83.3 %	5	1	6
combine(Range, Range)	100.0 %	4	0	4
expand(Range, double, double)	100.0 %	2	0	2
expandToInclude(Range, double)	100.0 %	6	0	6
scale(Range, double)	100.0 %	2	0	2
shift(Range, double)		0	0	0
shift(Range, double, boolean)	100.0 %	2	0	2
shiftWithNoZeroCrossing(double,	100.0 %	4	0	4
Range(double, double)	100.0 %	2	0	2
contains(double)	100.0 %	4	0	4
equals(Object)	100.0 %	6	0	6
getCentralValue()		0	0	0
getLength()		0	0	0
getLowerBound()		0	0	0
getUpperBound()		0	0	0
hashCode()		0	0	0
intersects(double, double)	100.0 %	8	0	8
intersects(Range)		0	0	0
toString()		0	0	0

Figure 9- Branch Coverage for Range Class



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

Figure 10- Condition Coverage for Range Class



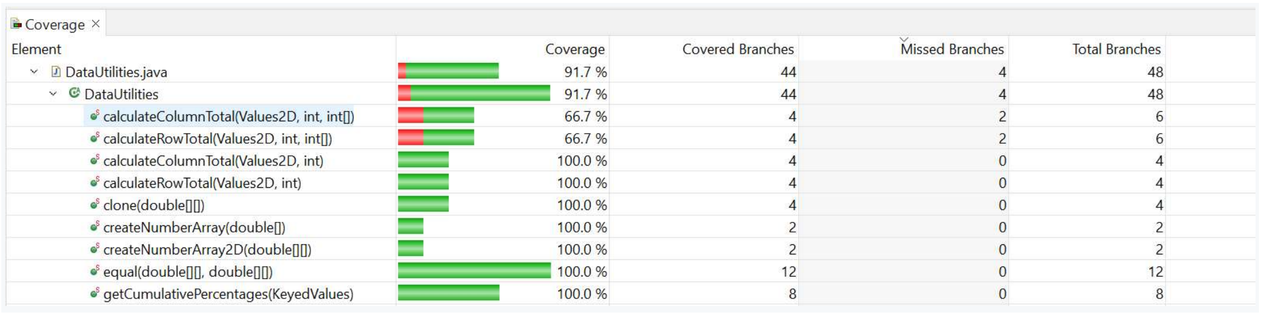
Element	Coverage	Covered Lines	Missed Lines	Total Lines
Range.java	93.2 %	96	7	103
Range	93.2 %	96	7	103
combineIgnoringNaN(Range, Rang	76.9 %	10	3	13
max(double, double)	60.0 %	3	2	5
min(double, double)	60.0 %	3	2	5
combine(Range, Range)	100.0 %	7	0	7
expand(Range, double, double)	100.0 %	8	0	8
expandToInclude(Range, double)	100.0 %	7	0	7
scale(Range, double)	100.0 %	5	0	5
shift(Range, double)	100.0 %	1	0	1
shift(Range, double, boolean)	100.0 %	7	0	7
shiftWithNoZeroCrossing(double,	100.0 %	5	0	5
Range(double, double)	100.0 %	8	0	8
constrain(double)	100.0 %	8	0	8
contains(double)	100.0 %	1	0	1
equals(Object)	100.0 %	8	0	8
getCentralValue()	100.0 %	1	0	1
getLength()	100.0 %	1	0	1
getLowerBound()	100.0 %	1	0	1
getUpperBound()	100.0 %	1	0	1
hashCode()	100.0 %	5	0	5
intersects(double, double)	100.0 %	3	0	3
intersects(Range)	100.0 %	1	0	1
isNaNRange()	100.0 %	1	0	1
toString()	100.0 %	1	0	1

Figure 11- Statement Coverage for Range Class

Coverage Type	Minimum Coverage Required	Old Coverage	New Coverage
Statement	90%	20%	93%
Branch	70%	16%	82%
Condition	60%	25%	100%

Figure 12- Coverage Table for Range Class

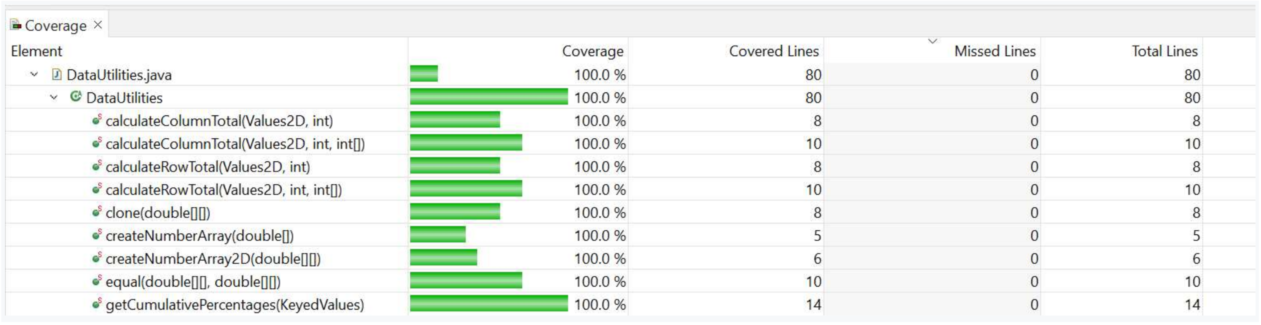
DataUtilities Class:



Coverage

Element	Coverage	Covered Branches	Missed Branches	Total Branches
DataUtilities.java	91.7 %	44	4	48
DataUtilities	91.7 %	44	4	48
calculateColumnTotal(Values2D, int, int[])	66.7 %	4	2	6
calculateRowTotal(Values2D, int, int[])	66.7 %	4	2	6
calculateColumnTotal(Values2D, int)	100.0 %	4	0	4
calculateRowTotal(Values2D, int)	100.0 %	4	0	4
clone(double[][])	100.0 %	4	0	4
createNumberArray(double[])	100.0 %	2	0	2
createNumberArray2D(double[][])	100.0 %	2	0	2
equal(double[][], double[][])	100.0 %	12	0	12
getCumulativePercentages(KeyedValues)	100.0 %	8	0	8

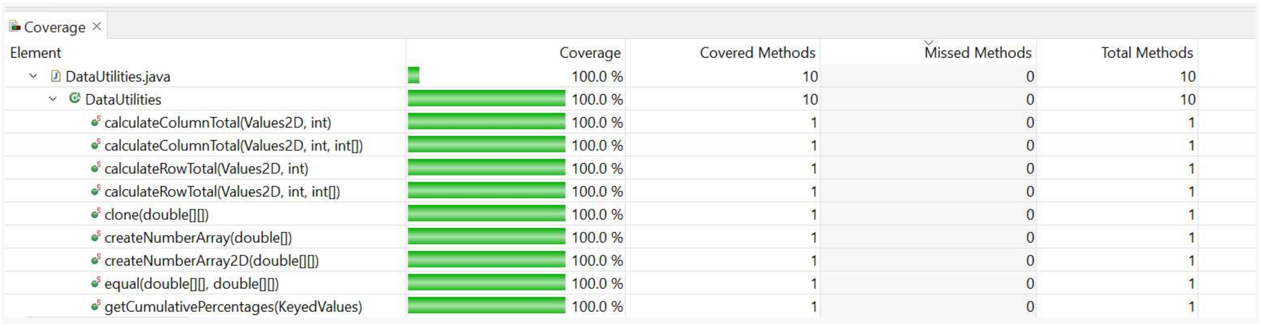
Figure 13- Branch Coverage for DataUtilities



Coverage

Element	Coverage	Covered Lines	Missed Lines	Total Lines
DataUtilities.java	100.0 %	80	0	80
DataUtilities	100.0 %	80	0	80
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
clone(double[][])	100.0 %	8	0	8
createNumberArray(double[])	100.0 %	5	0	5
createNumberArray2D(double[][])	100.0 %	6	0	6
equal(double[][], double[][])	100.0 %	10	0	10
getCumulativePercentages(KeyedValues)	100.0 %	14	0	14

Figure 14- Statement Coverage for DataUtilities



Coverage

Element	Coverage	Covered Methods	Missed Methods	Total Methods
DataUtilities.java	100.0 %	10	0	10
DataUtilities	100.0 %	10	0	10
calculateColumnTotal(Values2D, int)	100.0 %	1	0	1
calculateColumnTotal(Values2D, int, int[])	100.0 %	1	0	1
calculateRowTotal(Values2D, int)	100.0 %	1	0	1
calculateRowTotal(Values2D, int, int[])	100.0 %	1	0	1
clone(double[][])	100.0 %	1	0	1
createNumberArray(double[])	100.0 %	1	0	1
createNumberArray2D(double[][])	100.0 %	1	0	1
equal(double[][], double[][])	100.0 %	1	0	1
getCumulativePercentages(KeyedValues)	100.0 %	1	0	1

Figure 15- Condition Coverage for DataUtilities

Coverage Type	Minimum Coverage Required	Old Coverage	New Coverage
Statement	90%	52%	100%
Branch	70%	42%	91.7%
Condition	60%	60%	100%

Figure 16- Coverage Table for DataUtilities

Pros and cons of the coverage tools tried by your group in this assignment, in terms of reported measures, integration with the IDE and other plug-ins, user friendliness, crashes, etc.

We used the EclEmma coverage tool for finding the coverage for statement, branch, and condition. This tool has many advantages since it can display a green bar which means it has decent coverage or a red bar is displayed and that means more test cases need to be written to get better coverage. Moreover, it also shows the coverage percentage for statement, branch or condition which can be helpful when minimum coverage needs to be reached. The metrics are calculated fast for all related classes (Range and Data Utilities), and the results are directly shown in the Eclipse IDE to view and make changes to the test suite if needed. Another advantage of this tool is that the downloading process is easy as it is a plugin in the Eclipse IDE from the Eclipse Marketplace. It also highlights lines in the code as green (covered lines), red (uncovered lines) and yellow (partially covered). This can also help give a good understanding of which methods require a stronger test coverage. The tool is easy to use and user-friendly since downloading, using, and understanding the results from this tool can be used to increase the coverage of the test suite. This tool never crashes which makes it a reliable tool to use when finding the coverage for statement, branch, and condition. One of the disadvantages of using EclEmma is that it has limited support for conditional coverage. Another disadvantage using a tool such as EclEmma is that it does not guarantee that everything has been tested and moreover the results might not be 100% accurate.

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

Requirement-based test generation is a kind of black-box testing, which performs tests based on functional or non-functional requirements without reference to the internal structure of the component or system. On the other hand, coverage-based test generation refers to white-box testing, its procedure to derive or select test cases based on an analysis of the internal structure of a component or system. Both test generations have their own advantages and limitations. When building our testing suites, it is better to use a combination of both testing methods in order to generate better and more comprehensive results.

The general idea of coverage-based test generation is to use technologies such as statement coverage, branch coverage to try to make sure that each line or branch within source code is being executed. It helps to cover any branch or lines to try to find any possible bugs within them which increases the quality of the application. Since some lines with less chance to be executed in common use can be covered. However, the advantage of coverage-based tests is that it ignores testing the requirements of the system. Since it measures coverage of what has been written, but it cannot identify something that might be missing with the functionality. Also, a 100% line or branch coverage can not ensure bug free code (division by 0 example in lecture). Furthermore, we found that there is sometimes a reachability problem with control flow coverage in our testing. Some lines within the source code are actually not reachable. So that when one does not reach a 100% coverage, sometimes it is difficult to determine the reason.

On the other hand, requirements-based test methods focus on the actual requirement of the system. The advantage is that it directly tests the functionality of the system, any missing or incorrect implementation of functionalities can be easily found. However, one of the advantages is that it does not actually look at the implementation of the code. Errors existing within some lines which will only be executed at rare conditions can hardly be found.

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

Our group ran the coverage tool in eclipse to get an idea of which methods/functions needed better code coverage. Once this was done, we had split the test cases among ourselves. Girimer worked on CalculateColumnTotalTest, CalculateRowTotalTest, Clone, CreateNumberArrayTest, CreateNumberArray2DTest, equal, and

GetCumluativePercentagestTest. Muhammad worked on getUpperBound, getCentralValue, ExpandTest and Combine. Manpreet worked on Constrain, hashCode, Constructor and Intersects. Tianfan worked on combineIgnoringNaN, isNaNRange, Scale, Shift and expandToInclude. Once we were finished with getting better code coverage for our methods, we verified our findings with our group members.

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

By performing this lab, we have learned about white box testing control flow coverage. We had evaluated the functions/methods for org.jfree.data.DataUtilities and org.jfree.data.Range classes so that we can improve our test suite by applying the white box coverage criteria. Our goal was to increase the code coverage for the following: statement coverage, branch coverage and condition coverage. The built-in code coverage tool in eclipse was used for the org.jfree.data.DataUtilities and org.jfree.data.Range and it was helpful to determine which methods or functions need more stronger coverage. More test cases were created so that we can have better code coverage. We also learned how data-flow coverage is calculated by hand. This was done by calculating the DU-pair coverage for DataUtilities.calculateColumnTotal and Range.combine by following the code in these methods.

Comments/feedback on the assignment itself

The lab was extremely good in terms of giving a good understanding of white box testing. We learned to improve our test suite so that we can get better code coverage. The benefit of having access to the code gave us the opportunity to cover all the paths so that we can enhance our test suite. The built-in code coverage tool in eclipse gave us a good measurement of statement coverage, branch coverage and condition coverage and based on these findings we were able to design more test cases for our test suite. Also, the lab manual was very detailed and gave us a good overview of what we had to do in the lab and how to do it. The instructions about getting the set-up of the lab were very detailed and easy to follow.