SENG 438 - Software Testing, Reliability, and Quality

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

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

 # 1: Introduction

The main focus of this lab was to perform white-box testing with statement coverage, branch coverage and condition coverage. The SUT was the Jfreechart which was also used during the previous lab. The testsuite created for the SUT was constructed using the JUnit framework while the code coverage analysis section of the lab was completed using the EclEmma tool integrated with the eclipse IDE.

 # 2: Manual data-flow coverage calculations for DataUtilities.calculateColumnTotal and Range.equals methods

DataUtilities.calculateColumnTotal:

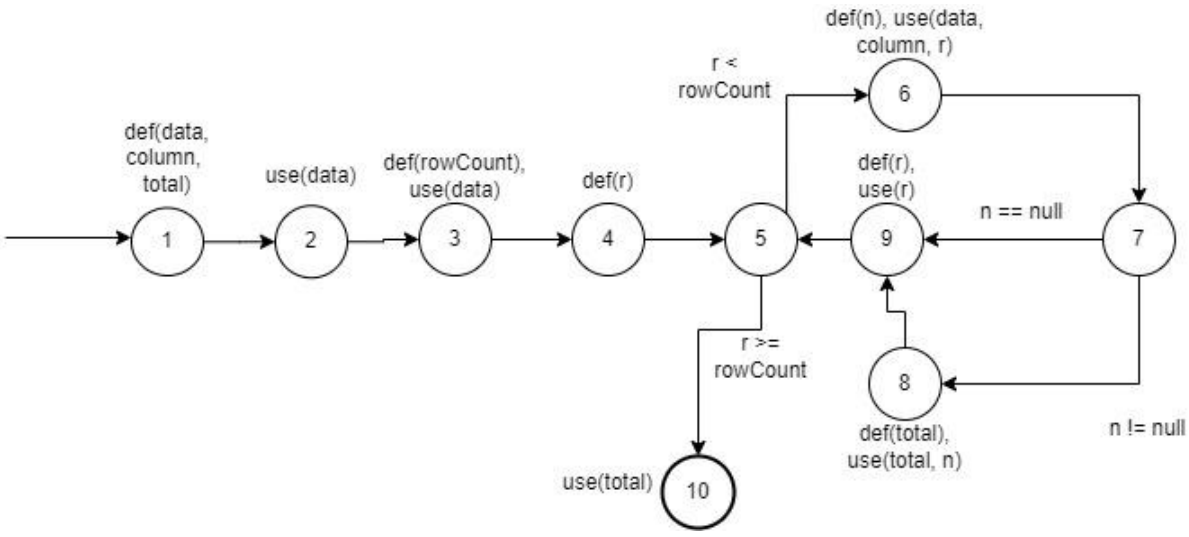

Figure 1 - Data flow graph for method *calculateColumnTotal*

```
1.      public static double calculateColumnTotal(Values2D data, int column) { def(data, column)
2.              ParamChecks.nullNotPermitted(data, "data")----------use(data)
3.              double total = 0.0; ----------def(total)
4.              int rowCount = data.getRowCount(); ----------def(rowCount), use(data)
5.              for (int r = 0; r < rowCount; r++) { ----------def(r), use(r), use(rowCount)
6.                      Number n = data.getValue(r, column); ----------def(n),use(r, data, column)
7.                      if (n != null) { ----------use(n)
8.                       total += n.doubleValue(); ----------def(total), use(total), use(n)
                        }
                }
9.              return total; ----------use(total)
        }
```

DU-pairs: Numbers based on Nodes in Data Flow graph

- data: {(1, 2), (1, 3), (1, 6)}
- column: {(1, 6)}
- total: {(1, 8), (1, 10), (8, 10)}
- rowCount: {(3, 5)}
- r: {(4, 5), (4, 6), (4, 9), (9, 5), (9, 6)}
- n: {(6, 7), (6, 8)}

Test Cases:

- calculateColumnTotalForTwoValues:
    - data: {(1, 2), (1, 3), (1, 6)}
    - column: {(1, 6)}
    - total: {(1, 8), (8, 10)}
    - rowCount: {(3, 5)}
    - r: {(4, 5), (4, 6), (4, 9), (9, 5), (9, 6)}
    - n: {(6, 7), (6, 8)}

- calculateColumnTotalForNegativeValues:
    - data: {(1, 2), (1, 3), (1, 6)}
    - column: {(1, 6)}
    - total: {(1, 8), (8, 10)}
    - rowCount: {(3, 5)}
    - r: {(4, 5), (4, 6), (4, 9), (9, 5), (9, 6)}
    - n: {(6, 7), (6, 8)}

- calculateColumnTotalForOneValues:
    - data: {(1, 2), (1, 3), (1, 6)}
    - column: {(1, 6)}

    - total: {(1, 8), (8, 10)}
    - rowCount: {(3, 5)}
    - r: {(4, 5), (4, 6), (4, 9), (9, 5)}
    - n: {(6, 7), (6, 8)}

- calculateColumnTotalForZeroValues:
    - data: {(1, 2), (1, 3)}
    - column: {}
    - total: {(1, 10)}
    - rowCount: {(3, 5)}
    - r: {(4, 5)}
    - n: {}

Untested DU-pairs:

- data: {}
- column: {}
- total:{}
- rowCount: {}
- r: {}
- n: {}

DU-pair Coverage:

Since there are no DU pairs untested, the DU coverage is 100%.

Rage.equals:

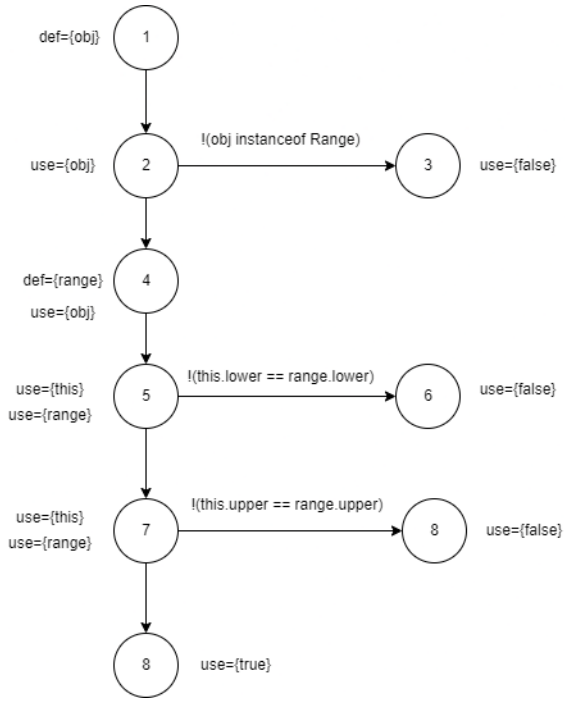

Figure 2 - Data flow graph for method *equals*

```
1.  public boolean equals(Object obj) {----------def(obj)
2.          if (!(obj instanceof Range)) { ------------ use(obj)
3.              return false;
            }
4.          Range range = (Range) obj; ------------ def(range), use(obj)
5.          if (!(this.lower == range.lower)) { ------- use(this), use(range)
6.              return false;
            }
7.          if (!(this.upper == range.upper)) { ------- use(this), use(range)
8.              return false; ------------------------------ use{false}
            }
9.          return true; ---------------------------------- use{true}
    }
```

DU-pairs: - Numbers based on Nodes in Data Flow graph

- obj: {{1,2}, {1,4}}
- range: {{4,5}, {4,7}}

Tests:

- equalsTestForSameRange():
  obj: {{1,2}, {1,4}}
  range: {{4,5}, {4,7}}
- equalsTestForLowerRange():
  obj: {{1,2}, {1,4}}
  range: {{4,5}}
- equalsTestForHigherRange():
  obj: {{1,2}, {1,4}}
  range: {{4,5}, {4,7}}

Untested DU-pairs:

- obj: {}
- range: {}

DU-pair Coverage:

Since there are no DU pairs untested, the DU coverage is 100%.

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

Before designing the test suite for the SUT, a testing plan was first created. The requirements for developing the test suite were:

- 90% statement coverage
- 70% branch coverage
- 60% condition coverage

Using the EclEmma code coverage tool, our group analyzed the unit tests created during the previous lab. Our findings were the following:

Necessary Coverage:

- 90% Statement Coverage
  - DataUtilities: 64.7/90% — 25.3% left which is 84 instructions
  - Range: 24.9% coverage — 60.1% left which is 282 instructions
- 70% Branch Coverage
  - DataUtilities: 47.9/70% — 22.1% left which is 11 branches
  - Range: 25/70% — 45% left which is 33 branches
- 60% Method Coverage
  - DataUtilities: 60%/60%
  - Range: 34.8/60% — 25.2% left which is 6 methods

To meet the minimum requirements of the lab, new tests needed to be constructed. The methods to test were reviewed by the group and then equally split among three group members while the last group member helped with reviewing the code. Before constructing the tests, our group constructed control flow graphs to help visualize the different areas of the method to test (conditions, paths, etc.). In addition to this, the *du-pairs* contained within the methods were also recorded. This information helped our group derive the minimum number of test cases needed to meet the set requirements while also considering the correct flow of data within the methods themselves.

 # 4: A high-level description of five selected test cases we designed using coverage information, and we increased the code coverage

1. scaleNegativeFactor()

This test case was designed to test for an illegal exception to be triggered when passing a negative value to method *Range.scale().* If implemented correctly, the *Range.scale()* method should either throw an illegal exception error if a negative is passed or return a new Range object that is scaled. The observed coverage increase for this method is shown in the table below.

Range.scale()			
	Statement	Branch	Method
Increase (%)	50	50	100

2. expandToIncludeAbove()

This test case was designed to test for a proper increase in range for a Range object when calling the method *Range.expandToInclude().* If implemented correctly, the method should return a range object with a new upper bound, lower bound, or upper and lower bound depending on the argument passed. The observed coverage increase for this method is shown in the table below.

Range.expandToInclude()			
	Statement	Branch	Method
Increase (%)	55.9	50	100

3. shiftBasicValue()

This test case was designed to test for a correct shift in a Range object after calling the method *Range.shift().* If implemented correctly, the method should return a Range object with the correct positive or negative shift, depending on the argument passed. The observed coverage for this method is shown in the table below.

Range.shift()			
	Statement	Branch	Method
Increase (%)	58.6	50	100

4. calculateRowTotalForOneValues()

This test case was designed to test for the correct return value for a Values2D object when *DataUtilities.calculateRowTotal()* is called when it has only one row. If implemented correctly, the method should return a double value that is equal to the sum of all values contained in one row of a *Values2D* object. The observed coverage for this method is shown in the table below.

DataUtilities.calculateRowTotal()			
	Statement	Branch	Method
Increase (%)	100	75	100

5. equalForInequalLengths()

This test case was designed to test for the correct return value of method *DataUtilities.equal()*. If implemented correctly, the method should return true if the input values (double arrays) are the same length, false otherwise. The observed coverage for this method is shown in the table below.

DataUtilities.equal()			
	Statement	Branch	Method
Increase (%)	28.2	25	100

#5: A detailed report of the coverage achieved of each class and method

New Tests Instructions Covered

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
JFreeChart_Lab3	1.7 %	3,785	213,699	217,484
src	1.7 %	3,785	213,699	217,484
org.jfree.chart.util	0.4 %	16	3,682	3,698
org.jfree.data	20.3 %	842	3,313	4,155
DefaultKeyedValues.java	15.7 %	79	423	502
Range.java	94.0 %	437	28	465
Range	94.0 %	437	28	465
combine(Range, Range)	100.0 %	26	0	26
combineIgnoringNaN(Range, Range)	100.0 %	46	0	46
expand(Range, double, double)	100.0 %	40	0	40
expandToInclude(Range, double)	100.0 %	34	0	34
max(double, double)	100.0 %	14	0	14
min(double, double)	100.0 %	14	0	14
scale(Range, double)	100.0 %	24	0	24
shift(Range, double)	100.0 %	5	0	5
shift(Range, double, boolean)	100.0 %	29	0	29
shiftWithNoZeroCrossing(double, double)	100.0 %	24	0	24
Range(double, double)	100.0 %	32	0	32
constrain(double)	100.0 %	25	0	25
contains(double)	100.0 %	14	0	14
equals(Object)	100.0 %	26	0	26
getCentralValue()	100.0 %	10	0	10
getLength()	100.0 %	6	0	6
getLowerBound()	100.0 %	3	0	3
getUpperBound()	100.0 %	3	0	3
intersects(double, double)	100.0 %	27	0	27
intersects(Range)	100.0 %	7	0	7
isNaNRange()	100.0 %	12	0	12
toString()	100.0 %	16	0	16
DataUtilities.java	99.1 %	326	3	329
DataUtilities	99.1 %	326	3	329
calculateColumnTotal(Values2D, int)	100.0 %	29	0	29
calculateColumnTotal(Values2D, int, int[])	100.0 %	37	0	37
calculateRowTotal(Values2D, int)	100.0 %	29	0	29
calculateRowTotal(Values2D, int, int[])	100.0 %	37	0	37
clone(double[][])	100.0 %	42	0	42
createNumberArray(double[])	100.0 %	26	0	26
createNumberArray2D(double[][])	100.0 %	25	0	25
equal(double[][], double[][])	100.0 %	39	0	39
getCumulativePercentages(KeyedValues)	100.0 %	62	0	62
org.jfree.data.test	99.4 %	2,927	18	2,945

New Tests Branches Covered

Element	Coverage	Covered Branches	Missed Branches	Total Branches
JFreeChart_Lab3	0.8 %	167	21,405	21,572
src	0.8 %	167	21,405	21,572
org.jfree.chart.util	0.6 %	2	348	350
org.jfree.data	21.1 %	115	429	544
DefaultKeyedValues.java	4.5 %	2	42	44
DataUtilities.java	91.7 %	44	4	48
DataUtilities	91.7 %	44	4	48
getCumulativePercentages(KeyedValues)	75.0 %	6	2	8
calculateColumnTotal(Values2D, int, int[])	83.3 %	5	1	6
calculateRowTotal(Values2D, int, int[])	83.3 %	5	1	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
Range.java	95.8 %	69	3	72
Range	95.8 %	69	3	72
isNaNRange()	75.0 %	3	1	4
constrain(double)	83.3 %	5	1	6
intersects(double, double)	87.5 %	7	1	8
combine(Range, Range)	100.0 %	4	0	4
combineIgnoringNaN(Range, Range)	100.0 %	14	0	14
expand(Range, double, double)	100.0 %	2	0	2
expandToInclude(Range, double)	100.0 %	6	0	6
max(double, double)	100.0 %	4	0	4
min(double, double)	100.0 %	4	0	4
scale(Range, double)	100.0 %	2	0	2
shift(Range, double, boolean)	100.0 %	2	0	2
shiftWithNoZeroCrossing(double, 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
shift(Range, double)		0	0	0
getCentralValue()		0	0	0
getLength()		0	0	0
getLowerBound()		0	0	0
getUpperBound()		0	0	0
intersects(Range)		0	0	0
toString()		0	0	0
org.jfree.data.test	100.0 %	50	0	50

New Tests Methods Covered

Element	Coverage	Covered Methods	Missed Methods	Total Methods
JFreeChart_Lab3	1.8 %	155	8,435	8,590
src	1.8 %	155	8,435	8,590
org.jfree.chart.util	0.8 %	1	121	122
org.jfree.data	19.3 %	37	155	192
DefaultKeyedValues.java	27.3 %	6	16	22
DataUtilities.java	90.0 %	9	1	10
DataUtilities	90.0 %	9	1	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
Range.java	95.7 %	22	1	23
Range	95.7 %	22	1	23
combine(Range, Range)	100.0 %	1	0	1
combineIgnoringNaN(Range, Range)	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, 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
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
org.jfree.data.test	100.0 %	117	0	117

 # 6: Pros and Cons of coverage tools used and Metrics

EclEmma

Pros:

- A quick way to analyze code stability
- An efficient way to track progress during the testing process
- Can be integrated within the eclipse IDE
- Helpful syntax highlighting and message prompts within the IDE

Cons:

- Requires a comprehensive understanding of coverage-based testing
- Requires tool familiarization
- Standalone data, usually needs to be reinforced with qualitative information
- The limited number of features

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

While both requirements-based test generation and coverage-based test generation can be crucially helpful when testing a system, there are certain cases when one technique should be used before the other. One significant difference between these two techniques is that coverage-based testing will require access to the source code of the SUT in contrast to requirements-based testing where the tester is only concerned with method inputs and outputs. For this reason, requirements-based testing is preferred when you want to limit the source code exposure from the SUT. In regards to requirements-based test generation, it is generally less time-consuming to create test cases because you are utilizing the inputs and outputs, without needing to filter and test any code within a method. While this technique can be useful when testing a system in its later stages for user experience, it is less reliable than coverage-based testing when trying to evaluate blocks of code within a method.

 # 8: A discussion on how the teamwork was divided and managed

During the lab, the coverage of the old test suite was analyzed and new tests were proposed. The new tests were divided among the team members.

Eric Renno:

- Test planning, quality control
- Coverage analysis
- Code review

Ryan Sommerville:

- Initial Instruction Coverage
- Data Flow Coverage for DataUtilities.calculateColumnTotal
- Tests:
    - Completed DataUtilities.equals
    - Completed Range.Range() tests
    - Completed Range.combine() tests
    - Completed Range.equals() tests
    - Wrote Range.shift(Range, double, boolean) and Range.shiftWithNoZeroCrossing() tests

Quinn Ledingham:

- Worked on data flow coverage for Range.equals
- Tests
    - Range.getLength() tests
    - Range.intersects() tests
    - Range.combineIgnoringNaN() tests
    - Range.expandToInclude() tests
    - Range.expand(), .scale(), .shift() tests

Kaumil Patel:

- Reported coverage details
- Tests
    - DataUtilities.equal
    - DataUtilities.calculateRowTotal
    - Range.Range
    - DataUtilities.calculateColumnTotal
    - DataUtilities.clone
    - Range.combineIgnoringNaN

New Test Suite

CLASS	INSTRUCTION MISSED	INSTRUCTION COVERED	INSTRUCTION COVERAGE
DataUtilities	3	326	99.1%
Range	28	437	94.0%

CLASS	BRANCH MISSED	BRANCH COVERED	BRANCH COVERAGE
DataUtilities	4	44	91.7%
Range	3	69	95.8%

CLASS	LINE MISSED	LINE COVERED	LINE COVERAGE
DataUtilities	1	79	98.8%
Range	5	98	95.1%

CLASS	COMPLEXITY MISSED	COMPLEXITY COVERED	COMPLEXITY COVERAGE
DataUtilities	5	29	85.3%
Range	4	55	93.2%

CLASS	METHOD MISSED	METHOD COVERED	METHOD COVERAGE
DataUtilities	1	9	90.0%
Range	1	22	95.7%

Old Test Suite

CLASS	INSTRUCTION MISSED	INSTRUCTION COVERED	INSTRUCTION COVERAGE
DataUtilities	116	213	64.7%
Range	349	116	24.9%

CLASS	BRANCH MISSED	BRANCH COVERED	BRANCH COVERAGE
DataUtilities	25	23	47.9%
Range	54	18	25.0%

CLASS	LINE MISSED	LINE COVERED	LINE COVERAGE
DataUtilities	32	48	60.0%
Range	73	30	29.1%

CLASS	COMPLEXITY MISSED	COMPLEXITY COVERED	COMPLEXITY COVERAGE

DataUtilities	20	14	41.2%
Range	44	15	25.4%

CLASS	METHOD MISSED	METHOD COVERED	METHOD COVERAGE
DataUtilities	4	6	60.0%
Range	15	8	34.8%

 # 9: Difficulties encountered, challenges overcome, and lessons learned from performing the lab

There were a few difficulties that our group members found during the completion of this lab. The first challenge that our group faced was with the initial setup of the code for the assignment. To resolve this issue, our group planned meetings where group members could give and receive feedback from the other members. New difficulties arose during the planning portion of the lab. After realizing that a large number of new test cases would be needed to meet the set requirements, it was unclear how to design the tests so they would meet these requirements. To address this issue, our group created control flow graphs to facilitate identifying unique test cases. The final issue we encountered was during the implementation stage of the lab. When dealing with complex data types it's good practice to use stubs, in order to mitigate dependency issues. This was found to be very difficult to implement when evaluating these structures in following a complex path. For simplicity, our group decided not to use stubs in certain cases but tried to use them whenever it was not too complicated to implement.

 # 10: Comments/feedback on the lab itself

This lab was found to be greatly beneficial in helping our group design and implement test cases using coverage-based test generation techniques. The assignment outline was easy to follow, making it easier to familiarize ourselves with the code-coverage tools needed for the completion of the lab. One possible suggestion for future assignment outlines would be to add a little more context to the grading scheme for the assignment.