

SENG 438 - Software Testing, Reliability, and Quality

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

| | |
|----------------|---------------|
| Group # | 1 |
| Student Names: | Huda Abbas |
| | Nuha Shaikh |
| | Lubaba Sheikh |
| | Rajpreet Gill |

1 Introduction

This lab enabled the team to concentrate on fully grasping the principles of automated unit testing and taking our skills from previous labs to strengthen them. In this lab, we used JUnit, Eclipse, and Eclemma as our top technologies to successfully complete the tasks outlined in the lab document. The lab enabled all of us to properly familiarize ourselves with the usage of the testing tools followed by the implementation and enhancement of the test suite developed. Furthermore, the lab followed the white-box technique which covered techniques such as def-use pairs along with coverage which included branch, method, and statement.

Along with the fundamental automation testing knowledge the lab provided, it also gave us an essence into the strength and influence teamwork offers. Through strong collaboration and cooperation as well as by employing effective teamwork capabilities, we all ensured to maximize our expertise by taking part in active discussions throughout the lab and providing each other guidance and feedback frequently. Overall, this lab was a success, and the following report captures all of the test cases performed, the coverage analysis alongside the added test cases which resulted in increased code coverage. The lab also provides reasoning and conclusions on how the requirements set in the lab document were met.

2 Manual data-flow coverage calculations for X and Y methods

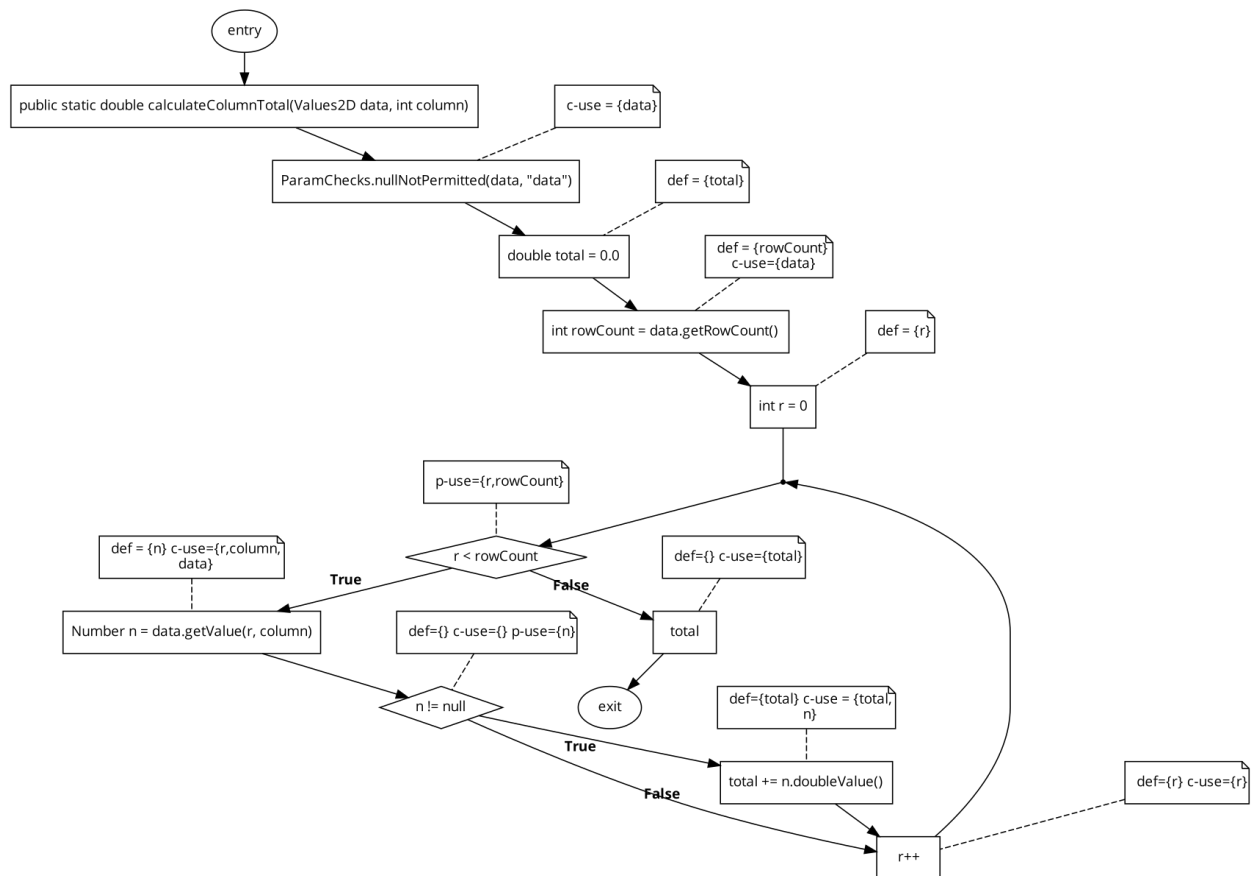
Method: DataUtilities.calculateColumnTotal

```

1 entry;
2 public static double calculateColumnTotal(Values2D data, int column) {
3     ParamChecks.nullNotPermitted(data, "data"); //c-use = {data}
4     double total = 0.0; //def = {total}
5     int rowCount = data.getRowCount(); //def = {rowCount} c-use={data}
6     for (int r = 0; r < rowCount; r++) { //def={r} c-use={r} p-use={r,rowCount}
7         Number n = data.getValue(r, column); //def = {n} c-use={r,column, data}
8         if (n != null) { //def={} c-use={} p-use={n}
9             total += n.doubleValue(); //def={total} c-use = {total, n}
10        }
11    }
12    return total; //def={} c-use={total}
13 }

```

The data flow graph



The def-use sets per statement

| Statement Number | Code line | Def-use Set |
|------------------|---|--|
| 2 | public static double calculateColumnTotal(Values2D data, int column) | def = {data, column} c-use = {} p-use = {} |

| | | |
|----|---|--|
| 3 | ParamChecks.nullNotPermitted(data, "data"); | def = {} c-use = {data} p-use={} |
| 4 | double total = 0.0; | def = {total} c-use = {} p-use={} |
| 5 | int rowCount = data.getRowCount(); | def = {rowCount} c-use={data} p-use={} |
| 6 | for (int r = 0; r < rowCount; r++) | def = {r} c-use = {r} p-use={r, rowCount} |
| 7 | Number n = data.getValue(r, column); | def = {n} c-use={r, column, data} p-use = {} |
| 8 | if (n != null) | def={} c-use={} p-use={n} |
| 9 | total += n.doubleValue(); | def={total} c-use = {total, n} p-use = {} |
| 12 | return total; | def={} c-use={total} p-use = {} |

All DU-pairs per variable

| Variable | Def-line | Use-line | Du-pairs |
|----------|----------|----------|-------------------------------------|
| total | 4 9 | 9 12 | {4,9} {9,12} {4, 12} {9,9} |
| rowCount | 5 | 6 | {5, 6} |
| r | 6 | 6 7 | {6, 6} {6, 7} |
| n | 7 | 8 9 | {7, 8} {7, 9} |
| data | 2 | 3 5 | {2, 3} {2, 5} |

| | | | |
|--------|---|---|--------|
| | | 7 | {2, 7} |
| column | 2 | 7 | {2, 7} |

For each test case show which pairs are covered

- //This test covers the invalid class/partition for first input variable


```
@Test (expected = IllegalArgumentException.class)
    public void test_calculatedColumnTotal_invalidData() {
        DataUtilities.calculateColumnTotal(null, 0);
    }
```

 - **Pairs covered:** {2,3}
- //This test covers invalid class for second input variable, nothing to sum since column values are null


```
@Test
    public void test_calculatedColumnTotal_negativeColumn() {
        double result = DataUtilities.calculateColumnTotal(values, -1);
        assertEquals("Summing all the values in an out of bounds (neg) column is", 0, result,
            .000000001d);
    }
```

 - **Pairs covered:**
 - {4, 12}, {5, 6}, {6, 6}, {6, 7}, {7,8}, {2, 3}, {2, 5}, {2, 7}
- //This test covers invalid class for second input variable where column index //is larger than data size, nothing to sum since column values are null


```
@Test
    public void test_calculatedColumnTotal_outOfBoundsColumn() {
        double result = DataUtilities.calculateColumnTotal(values, 3);
        assertEquals("Summing all the values in an out of bounds column is", 0, result,
            .000000001d);
    }
```

 - **Pairs covered:**
 - {4, 12}, {5, 6}, {6, 6}, {6, 7}, {7,8}, {2, 3}, {2, 5}, {2, 7}
- //This test covers valid input for second variable but at boundary of valid and invalid


```
@Test
    public void test_calculatedColumnTotal_columnZero() {
        double result = DataUtilities.calculateColumnTotal(values, 0);
        assertEquals("Summing all the values in the first column is", 6.5, result, .000000001d);
    }
```

 - **Pairs covered:**
 - {5, 6}, {6, 6}, {6, 7}, {7,8}, {2, 3}, {2, 5}, {2, 7}, {4,9}, {9,12}, {9,9}, {7, 9}

- //This test covers valid input for second variable where it selects one sample within equivalence class not at the boundary

```
@Test
public void test_calculatedColumnTotal_validColumn() {
    double result = DataUtilities.calculateColumnTotal(values, 1);
    assertEquals("Summing all the values in the second column is", 12, result,
        .000000001d);
}
```

- **Pairs covered:**

- {5, 6}, {6, 6}, {6, 7}, {7,8}, {2, 3}, {2, 5}, {2, 7}, {4,9}, {9,12}, {9,9}, {7, 9}

- //This test covers valid input for second variable but at boundary of valid and invalid //index right at the maximum size of the data

```
@Test
public void test_calculatedColumnTotal_columnAtSize() {
    double result = DataUtilities.calculateColumnTotal(values, 2);
    assertEquals("Summing all the values in the last column is", 10, result, .000000001d);
}
```

- **Pairs covered:**

- {5, 6}, {6, 6}, {6, 7}, {7,8}, {2, 3}, {2, 5}, {2, 7}, {4,9}, {9,12}, {9,9}, {7, 9}

Calculate the DU-Pair coverage

A measure of def-clear paths from each variable definition to all uses of that variable that are reachable from the destination.

$$Coverage = \frac{\text{number exercised DU pairs}}{\text{number of DU pairs}} = \frac{CUc + PUc}{(CU+PU) - (CUf + PUf)} = \frac{9+3}{9+3 - (0+0)} = \frac{12}{12} \times 100 = 100\%$$

Method: Range.Constrain

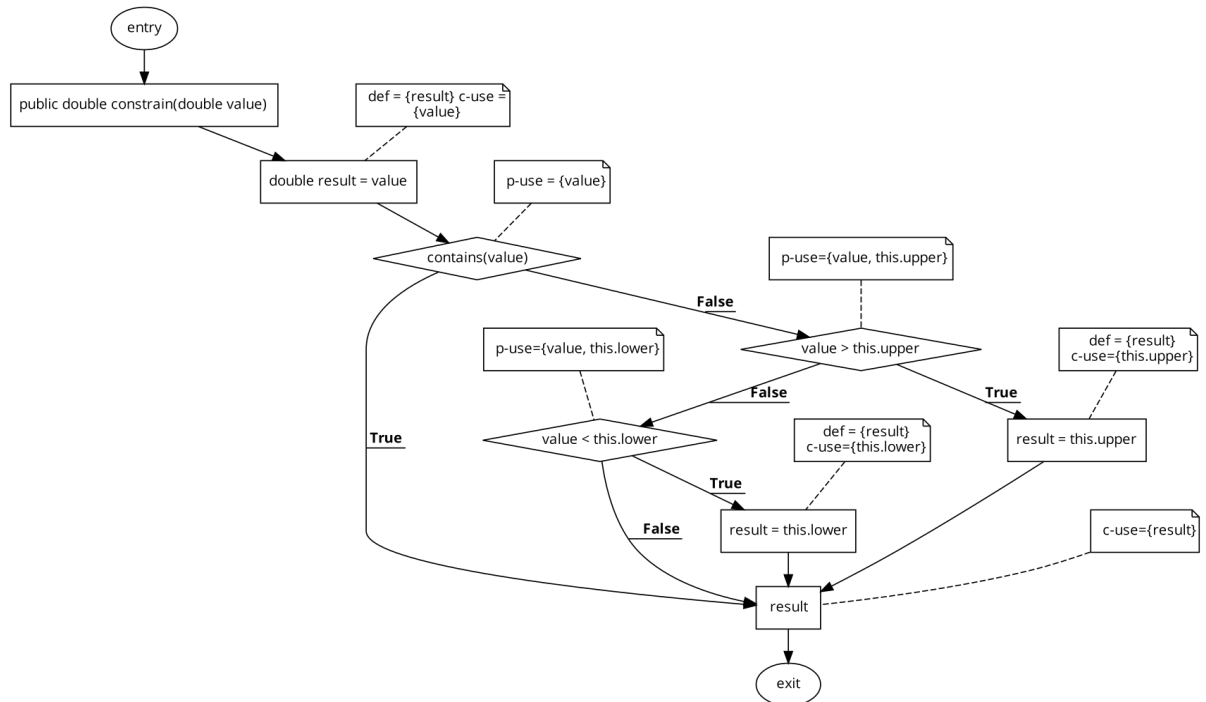
Constrain

```

1  entry;
2  public double constrain(double value) {
3      double result = value; //def = {result} c-use = {value}
4      if (!contains(value)) { //p-use = {value}
5          if (value > this.upper) { //p-use={value, this.upper}
6              result = this.upper; //def = {result} c-use={this
              .upper}
7          }
8          else if (value < this.lower) { //p-use={value, this
              .lower}
9              result = this.lower; //def = {result} c-use={this
              .lower}
10     }
11 }
12 result; //c-use={result}
13 exit;
14 }

```

The data flow graph



The def-use sets per statement

| Statement Number | Code line | Def-use Set |
|------------------|---------------------------------------|---|
| 2 | public double constrain(double value) | def={value} c-use={} p-use={} |
| 3 | double result = value; | def={result} c-use={value} p-use={} |
| 4 | if(!contains(value)) | def={} c-use={} p-use={value} |
| 5 | if(value>this.upper) | def={} c-use={} p-use={value, this.upper} |
| 6 | result=this.upper; | def={result} c-use={this.upper} p-use={} |
| 8 | else if(value<this.lower) | def={} c-use={} p-use={value, this.lower} |
| 9 | result=this.lower; | def={result} c-use={this.lower} p-use={} |
| 12 | return result; | def={} c-use={result} p-use={} |

list all DU-pairs per variable

| Variable | Def-line | Use-line | Du-pairs |
|----------|-------------|------------------|--------------------------------------|
| value | 2 | 3 4 5 8 | {2, 3} {2, 4} {2, 5} {2, 8} |
| result | 3 6 9 | 12 | {3, 12} {6, 12} {9, 12} |

For each test case show which pairs are covered

- // Test for value below lower bound
 // This test uses a valid Range object and a valid value where the value
 // is below the lower bound of the range to test the method constrain(value)
 @Test
 public void test_constrain_valueBelowLower()
 {
 double actual = validRange.constrain(-2);
 double expected = 4;
 assertEquals("Testing constrain with value below lower bound", expected, actual,
 .000000001d);
 }

- **Pairs covered:**

- {2, 3}, {2,4}, {2,5}, {2,8}, {9,12}

- // Test for value between lower bound and upper bound
 // This test uses a valid Range object and a valid value where the value
 // is between the range to test the method constrain(value)
 @Test
 public void test_constrain_valueBetweenRange()
 {
 double actual = validRange.constrain(9);
 double expected = 9;
 assertEquals("Testing constrain with value between the range", expected, actual,
 .000000001d);
 }

- **Pairs Covered**

- {2, 3}, {2,4}, {3, 12}

```
// Test for value above upper bound
// This test uses a valid Range object and a valid value where the value
// is above the upper bound of the range to test the method constrain(value)
@Test
public void test_constrain_valueAboveUpper()
{
    double actual = validRange.constrain(20);
    double expected = 10;
    assertEquals("Testing constrain with value above upper bound", expected, actual, .000000001d);
}
```

- **Pairs Covered**

- {2, 3}, {2, 4}, {2,5}, {6, 12}

- // Test for value at lower bound
 // This test uses a valid Range object and a valid value where the value
 // is the boundary of the lower bound of the range to test the method constrain(value)
 @Test


```

public void test_constrain_valueAtLower()
{
    double actual = validRange.constrain(4);
    double expected = 4;
    assertEquals("Testing constrain with value at boundary of lower bound", expected, actual,
.000000001d);
}

```

- **Pairs Covered**

- {2, 3}, {2, 4}, {3, 12}

• // Test for value at upper bound

// This test uses a valid Range object and a valid value where the value
// is the boundary of the upper bound of the range to test the method constrain(value)

@Test

```

public void test_constrain_valueAtUpper()
{

```

```

    double actual = validRange.constrain(10);

```

```

    double expected = 10;

```

```

    assertEquals("Testing constrain with value at boundary of upper bound",

```

```

expected, actual, .000000001d);

```

```

}

```

- **Pairs Covered**

- {2, 3}, {2, 4}, {3, 12}

Calculate the DU-Pair coverage

$$\text{Coverage} = \frac{\text{number exercised DU pairs}}{\text{number of DU pairs}} = \frac{CUc + PUc}{(CU + PU) - (CUf + PUf)} = \frac{3+4}{3+4 - (0+0)} = \frac{7}{7} \times 100 = 100\%$$

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

How you plan to develop tests to achieve the adequacy criteria?

Our plan is to enhance our test suite by adding additional test cases tackling the methods we have not yet tested. For DataUtilities.java this includes the equal and clone class. Our goal for DataUtilities.java is to get our statement coverage from 76.2% to 90% and our branch coverage from 62.5% to 70%. Our goal for Range.java is to get our statement coverage from 29.0% to 90%, our branch coverage from 29.2% to 70% and our method coverage from 39.1% to 60%.

(Our goal) Minimum coverage:

- 90% statement coverage (76.2 -> 90, 29 -> 90)

- 70% branch coverage (62.5 -> 70, 29.2 -> 70)
- 60% method coverage (39.1 -> 60)

Our general testing plan will follow boundary and branch decisions, while also considering coverage of different partitions. For example for the shift and equals class here is our general preliminary planning work

shift(Range base, double delta, Boolean allowZeroCrossing): Range

- Invalid Data:
 - Null base
- Valid Data:
 - Delta is positive, allow
 - Delta is positive, don't allow
 - Delta is negative, allow
 - Delta is negative, don't allow
- Boundary
 - Delta is 0, allow
 - Delta is 0, don't allow

shift(Range base, double delta): Range

- Invalid Data:
 - Null base
- Valid Data:
 - Delta is positive
 - Delta is negative
- Boundary
 - Delta is 0

equals(Object obj): Boolean

- Pass in a non Range Object to test first if statement
- Pass in an equal range object to test last return statement and remaining two if statements
- Pass in an range object with upper and lower values different from this pointer range to test the inside of the two remaining if statements

Who will create which tests for both Range and DataUtilities classes?

- Huda: public static double[][] clone(double[][] source), isNaNRange, hashCode
- Rajpreet: public static boolean equal(double[][] a, double[][] b), combineIgnoringNaN(Range range1, Range range2): Range
- Lubaba: shift (Range base, double delta, Boolean allowZeroCrossing), equal(Object obj)
- Nuha: combine, ExpandToInclude, expand

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

Below are the additional test cases designed using coverage information

DataUtilities.java

- **clone(double[][] source): double[][]**
 - test_clone_invalidData()
 - Input: source = null
 - Output: IllegalArgumentException.class
 - Covers the invalid class/partition for the input variable
 - test_clone_validData_emptyArray()
 - Input: source = new int[2][2]
 - Output: [[0,0],[0,0]]
 - Covers the valid entry for by supplying non-null empty array
 - test_clone_validData_setValues()
 - Input: source = [[1,2.5],[1.5,2]]
 - Output: [[1,2.5],[1.5,2]]
 - Covers the valid entry for by supplying non-null values
- **equal(double [][] a, double [][] b): boolean**
 - test_equal_invaliddata()
 - Input: null for first argument and null for second argument
 - Output: true
 - Covers the invalid data by inputting null into both the first array argument and the second array argument
 - test_equal_validdata_matchingArray()
 - Input: {{2},{3}} for the first array and {{2},{3}} for the second array
 - Output: true
 - Covers the valid data of array a equaling to {{2},{3}} and b to {{2},{3}} as well
 - test_equal_validdata_unmatchingArray()
 - Input: {{2},{3}} for array a and {{2},{4}} for array b
 - Output: false
 - Covers the valid data of array a equaling to {{2},{3}} and b to {{2},{4}}

Range.java

- **combine(Range range1, Range range2): Range**
 - test_combine_invalidData_null()
 - Input: range1=null, range2= null
 - Output: range2 -> null
 - Covers: invalid value, null for input variable
 - test_combine_invalidData_range1null()
 - Range input @setup: -10, 10
 - Input: range1=null, range2= (-10,10)
 - Output: range2 -> (-10,10)
 - Covers: invalid value, null for input variable
 - test_combine_invalidData_range2null()
 - Range input @setup: -10, 10
 - Input: range1=(-10,10), range2=null
 - Output: range1 -> (-10,10)
 - Covers: invalid value, null for input variable
 - test_combine_validData_equalRanges()
 - Range input @setup: (-10, 10) and (-10,10)
 - Input: range1=(-10,10), range2=(-10,10)
 - Output: (-10,10)
 - Coverage: edges of valid input data with non null arguments
 - test_combine_validData_regular()
 - Range input @setup: (-7,-2), (11,20)
 - Input: range1=(-7,-2), range2=(11,20)
 - Output: (-7,20)
 - Coverage: valid inputs with non null arguments
- **expandToInclude(Range range, double value): Range**
 - test_expandToInclude_Crossing_invalidData_valueNull()
 - Input: range = null, value=10
 - Output: Range=(10,10)
 - Covers: Invalid Data with null values for range object input
 - test_expandToInclude_validData_valueGreaterThan0()
 - Input: Range =(-10,10), value=-11
 - Output: Range=(-11,10)
 - Covers: Valid Data with value less than range lower bound, to cover more branches
 - test_expandToInclude_validData_valueLessThan0()
 - Input: Range =(-10,10), value=11
 - Output: Range=(-10,11)
 - Covers: Valid Data with value greater than range upper bound, to cover more branches
 - test_expandToInclude_validData_valueEqualToZero()
 - Input: Range =(10,10), value=10
 - Output: Range=(10,10)
 - Covers: Valid Data with equivalent values to increase branch coverage

- **shift(Range base, double delta, Boolean allowZeroCrossing): Range**
 - test_shift_override_nullRange
 - Input: null Range, delta = 6
 - Output: IllegalArgumentException
 - Checks behavior of function when a null range is provided, EC invalid input partition
 - test_shift_override_positiveDeltaTrue
 - Input:
 - Delta = 10
 - Valid base range: lower = -5, upper = 5
 - allowZeroCrossing = True
 - Output: lower = 5, upper = 15
 - Checks behavior of function when a positive delta is provided with zero crossing allowed, EC valid input partition. Tests lower bound crossing zero
 - test_shift_override_positiveDeltaFalse
 - Input:
 - Delta = 10
 - Valid base range: lower = -5, upper = 5
 - allowZeroCrossing = False
 - Output: lower = 0, upper = 15
 - Checks behavior of function when a positive delta is provided with zero crossing not allowed, EC valid input partition. Tests lower bound not crossing zero
 - test_shift_override_negativeDeltaTrue
 - Input:
 - Delta = -10
 - Valid base range: lower = -5, upper = 5
 - allowZeroCrossing = True
 - Output: lower = -15, upper = -5
 - Checks behavior of function when a negative delta is provided with zero crossing allowed, EC valid input partition. Tests upper bound crossing zero
 - test_shift_override_negativeDeltaFalse
 - Input:
 - Delta = -10
 - Valid base range: lower = -5, upper = 5
 - allowZeroCrossing = False
 - Output: lower = -15, upper = 0
 - Checks behavior of function when a negative delta is provided with zero crossing not allowed, EC valid input partition. Tests upper bound not crossing zero
 - test_shift_override_zeroDeltaTrue
 - Input:
 - Delta = 0
 - Valid base range: lower = -5, upper = 5
 - allowZeroCrossing = True
 - Output: lower = -5, upper = 5

- Checks behavior of function when a zero delta is provided with zero crossing allowed, EC valid input partition
 - test_shift_override_zeroDeltaFalse
 - Input:
 - Delta = 0
 - Valid base range: lower = -5, upper = 5
 - allowZeroCrossing = False
 - Output: lower = -5, upper = 5
 - Checks behavior of function when a zero delta is provided with zero crossing allowed, EC valid input partition
- **shift(Range base, double delta): Range**
 - test_shift_nullRange
 - Input: null Range, delta = 6
 - Output: IllegalArgumentException
 - Checks behavior of function when a null range is provided, EC invalid input partition
 - test_shift_positiveDelta
 - Input:
 - Delta = 10
 - Valid base range: lower = -5, upper = 5
 - Output: lower = 0, upper = 15
 - Checks behavior of function when a positive number is provided for delta shift, EC valid input partition
 - test_shift_negativeDelta
 - Input:
 - Delta = -5
 - Valid base range: lower = 4, upper = 10
 - Output: lower = 0, upper = 5
 - Checks behavior of function when a negative number is provided for delta shift, EC valid input partition
 - test_shift_zeroDelta
 - Input:
 - Delta = 0
 - Valid base range: lower = 4, upper = 10
 - Output: lower = 4, upper = 10
 - Checks behavior of function when a 0 is provided for delta shift, EC boundary input partition
- **equals(Object obj): Boolean**
 - test_equals_notRangeObj()
 - Input:
 - Obj = a string object str
 - Output: false
 - test_equals_equalRange()
 - Input:

- Obj = new Range(-5, 5)
 - Output: true
 - test_equals_lowerBranch()
 - Input:
 - Obj = new Range(-1, 5)
 - Output: false
 - test_equals_upperBranch()
 - Input:
 - Obj = new Range(-5, 30)
 - Output: false
- **hashCode(): int**
 - test_hashCode_valid()
 - Input: Obj = new Range(10,11)
 - Output: -2076573696
 - Checks testing hashCode function with valid class object Range
 - test_hashCode_invalid()
 - Input: Obj = new Range(10,0)
 - Output: IllegalArgumentException.class
 - Checks invalid Range class where lower is great than upper
- **combineIgnoringNaN(Range range1, Range range2): Range**
 - test_combineIgnoringNaN_both_ranges_null_value()
 - Input:
 - Null for object1 and null for obejct2 being sent to this method
 - Output:
 - null
 - Checks the combineIgnoringNaN with both values equaling to null for the first range object and the second range object
 - test_combineIgnoringNaN_range1_null_only()
 - Input:
 - First value is null and second is isNaN
 - Output:
 - null
 - Checks the combineIgnoringNaN with first Range value being null and second range object is isNaN
 - test_combineIgnoringNaN_range2_null_only()
 - Input:
 - First input is isNaN and second value is null
 - Output:
 - null
 - Checks the comineIgnoringNaN with first object value isNaN and second object value being null
 - test_combineIgnoringNaN_valid_values()
 - Input:
 - new Range(2,3), new Range(4,5)

- Output:
 - new Range(2,5);
- Checks the combineIgnoringNaN method with both values being valid

How these five selected test cases increased code coverage

- **clone(double[][] source): double[][]**
 - This method contains 1 if statements hence covering all the branches in this function helped increase the branch coverage for this class. A test case was created to test for each branch to ensure that the right path is followed given the objects as inputs.
 - Testing this method also covered all nodes corresponding to each program statement. This increased the line coverage of the class due to all the node coverages.
 - Creating test cases for this method contributed to increasing method coverage for the DataUtilities class. The clone(double[][] source) is a public method that can be accessed, therefore ensuring that this method is functional will contribute to accurate outputs of the SUT
- **equal(double[][] a, double[][] b): boolean**
 - This method contains 4 if statements hence covering all the branches in this function helped increase the branch coverage for this class. A test case was created to test for each branch to ensure that the right path is followed given the objects as inputs. This test case belongs to the DataUtilities class.
 - To increase line coverage, testing of all nodes corresponding to each program statement was performed. This increased the line coverage of the class due to all the node coverages.
 - Creating test cases for this method contributed to increasing method coverage for the DataUtilities class. The equal(double[][] a, double [][] b) is a public method that can be accessed, therefore ensuring that this method is functional will contribute to accurate outputs of the SUT
- **equals(Object obj)**
 - This method contains 3 if statements hence covering all the branches in this function helped increase the branch coverage for this class. A test case was created to test for each branch to ensure that the right path is followed given the objects as inputs.
 - Testing this method also covered all nodes corresponding to each program statement. This increased the line coverage of the class due to all the node coverages.
 - Creating test cases for this method contributed to increasing method coverage for the Range class. The equals(Object obj) is a public method that can be accessed, therefore ensuring that this method is functional will contribute to accurate outputs of the SUT
- **shift (Range base, double delta, Boolean allowZeroCrossing)**
 - This method contains 1 if statement and an else statement hence covering all the branches in this function helped increase the branch coverage for the Range class. A test case was created to test each branch to ensure that the right path was being followed given the arguments provided as inputs to the method.
 - Testing this method also covered all nodes corresponding to each program statement. The statements of this method include Null not permitted checks for invalid inputs which

were also covered through line coverage. This increased the overall line coverage of the class to ensure proper functionality..

- Creating test cases for this method contributed to increasing the method coverage for the Range class. The shift (Range base, double delta, Boolean allowZeroCrossing) is a public method that can be accessed, therefore ensuring that this method is functional will contribute to the accuracy of the SUT
- **expandToInclude(Range range, double value): Range**
 - Testing this method increased branch coverage, branch coverage related to decisions in a program, this is because the method encapsulated lots of decision statements and each test goes through a different decision point increasing branch coverage.
 - This method contains 2 if statements, 1 else if and 1 else statement hence covering all the branches in this function helped increase the branch coverage for the Range class.
 - Testing all branches also covered all nodes i.e. all branches of the code.
 - Creating test cases for this method contributed to increasing method coverage for the Range class. The equal(Object obj) is a public method that can be accessed, therefore ensuring that this method is functional will contribute to accurate outputs of the SUT.

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





Original code coverage for our entire test suite using EclEmma as our code coverage tool.

Three coverage metrics:





1. Statement/Line coverage

| | | | | | |
|--|---|--------|----|----|-----|
| >  DataUtilities.java |  | 76.2 % | 61 | 19 | 80 |
| >  Range.java |  | 29.1 % | 30 | 73 | 103 |

2. Branch coverage

| | | | | | |
|--|---|--------|----|----|----|
| >  DataUtilities.java |  | 62.5 % | 30 | 18 | 48 |
| >  Range.java |  | 29.2 % | 21 | 51 | 72 |

3. Method coverage

| | | | | | |
|--|---|--------|---|----|----|
| >  DataUtilities.java |  | 70.0 % | 7 | 3 | 10 |
| >  Range.java |  | 39.1 % | 9 | 14 | 23 |

New code coverage of your entire test suite.

1. Statement/Line coverage

| Element | | Coverage | Covered Lines | Missed Lines | Total Lines |
|---------------------------------|--|----------|---------------|--------------|-------------|
| ▼ DataUtilities.java | | 96.2 % | 77 | 3 | 80 |
| ▼ DataUtilities | | 96.2 % | 77 | 3 | 80 |
| calculateColumnTotal(Values2D | | 100.0 % | 8 | 0 | 8 |
| calculateColumnTotal(Values2D | | 100.0 % | 10 | 0 | 10 |
| calculateRowTotal(Values2D, int | | 100.0 % | 8 | 0 | 8 |
| calculateRowTotal(Values2D, 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[][]) | | 80.0 % | 8 | 2 | 10 |
| getCumulativePercentages(Key | | 100.0 % | 14 | 0 | 14 |

| Element | | Coverage | Covered Lines | Missed Lines | Total Lines |
|----------------------------------|--|----------|---------------|--------------|-------------|
| ▼ Range.java | | 91.3 % | 94 | 9 | 103 |
| ▼ Range | | 91.3 % | 94 | 9 | 103 |
| combine(Range, Range) | | 100.0 % | 7 | 0 | 7 |
| combineIgnoringNaN(Range, Range) | | 84.6 % | 11 | 2 | 13 |
| expand(Range, double, double) | | 100.0 % | 8 | 0 | 8 |
| expandToInclude(Range, double) | | 100.0 % | 7 | 0 | 7 |
| max(double, double) | | 60.0 % | 3 | 2 | 5 |
| min(double, double) | | 60.0 % | 3 | 2 | 5 |
| 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) | | 80.0 % | 4 | 1 | 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() | | 0.0 % | 0 | 1 | 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() | | 0.0 % | 0 | 1 | 1 |

2. Branch coverage

| Element | | Coverage | Covered Branches | Missed Branches | Total Branches |
|---------------------------------|--|----------|------------------|-----------------|----------------|
| ▼ DataUtilities.java | | 87.5 % | 42 | 6 | 48 |
| ▼ DataUtilities | | 87.5 % | 42 | 6 | 48 |
| calculateColumnTotal(Values2D | | 100.0 % | 4 | 0 | 4 |
| calculateColumnTotal(Values2D | | 100.0 % | 6 | 0 | 6 |
| calculateRowTotal(Values2D, int | | 100.0 % | 4 | 0 | 4 |
| calculateRowTotal(Values2D, int | | 100.0 % | 6 | 0 | 6 |
| clone(double[][]) | | 75.0 % | 3 | 1 | 4 |
| createNumberArray(double[]) | | 100.0 % | 2 | 0 | 2 |
| createNumberArray2D(double[] | | 100.0 % | 2 | 0 | 2 |
| equal(double[][], double[][]) | | 75.0 % | 9 | 3 | 12 |
| getCumulativePercentages(Key | | 75.0 % | 6 | 2 | 8 |

| Element | | Coverage | Covered Branches | Missed Branches | Total Branches |
|---|--|----------|------------------|-----------------|----------------|
| Range.java | | 79.2 % | 57 | 15 | 72 |
| Range | | 79.2 % | 57 | 15 | 72 |
| combine(Range, Range) | | 100.0 % | 4 | 0 | 4 |
| combineIgnoringNaN(Range, Range) | | 64.3 % | 9 | 5 | 14 |
| expand(Range, double, double) | | 100.0 % | 2 | 0 | 2 |
| expandToInclude(Range, double, double) | | 100.0 % | 6 | 0 | 6 |
| max(double, double) | | 50.0 % | 2 | 2 | 4 |
| min(double, double) | | 50.0 % | 2 | 2 | 4 |
| 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, double) | | 75.0 % | 3 | 1 | 4 |
| Range(double, double) | | 100.0 % | 2 | 0 | 2 |
| constrain(double) | | 83.3 % | 5 | 1 | 6 |
| 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) | | 75.0 % | 6 | 2 | 8 |
| intersects(Range) | | | 0 | 0 | 0 |
| isNaNRange() | | 50.0 % | 2 | 2 | 4 |
| toString() | | | 0 | 0 | 0 |

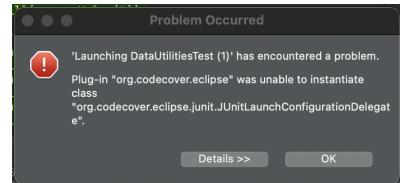
3. Method coverage

| Element | | Coverage | Covered Methods | Missed Methods | Total Methods |
|---|--|----------|-----------------|----------------|---------------|
| DataUtilities.java | | 90.0 % | 9 | 1 | 10 |
| DataUtilities | | 90.0 % | 9 | 1 | 10 |
| calculateColumnTotal(Values2D) | | 100.0 % | 1 | 0 | 1 |
| calculateColumnTotal(Values2D) | | 100.0 % | 1 | 0 | 1 |
| calculateRowTotal(Values2D, int) | | 100.0 % | 1 | 0 | 1 |
| calculateRowTotal(Values2D, int) | | 100.0 % | 1 | 0 | 1 |
| clone(double[][]) | | 100.0 % | 1 | 0 | 1 |
| createNumberArray(double[]) | | 100.0 % | 1 | 0 | 1 |
| createNumberArray2D(double[][], double[]) | | 100.0 % | 1 | 0 | 1 |
| equal(double[][], double[][], double[]) | | 100.0 % | 1 | 0 | 1 |
| getCumulativePercentages(Key) | | 100.0 % | 1 | 0 | 1 |

| Element | | Coverage | Covered Methods | Missed Methods | Total Methods |
|---|--|----------|-----------------|----------------|---------------|
| Range.java | | 91.3 % | 21 | 2 | 23 |
| Range | | 91.3 % | 21 | 2 | 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, 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() | | 0.0 % | 0 | 1 | 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() | | 0.0 % | 0 | 1 | 1 |

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

Table 1.0: Testing tools Eclemma, Codecover and Clover with their pros and cons

| Tool | Pros | Cons |
|-----------|--|---|
| Eclemma | <ul style="list-style-type: none">• Cover statement/line coverage, branch coverage and method coverage• No issues with mocking components | <ul style="list-style-type: none">• Does not have path and condition coverage |
| Codecover | <ul style="list-style-type: none">• In theory, it has condition coverage which is not present in Eclemma | <ul style="list-style-type: none">• Did not work with our configuration• Resulted in this error:  |
| Clover | <ul style="list-style-type: none">• Easy integration with Range.java class | <ul style="list-style-type: none">• Does not work well with mocking |

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

Advantages of requirements-based testing:

- Effective for large code segments
- Uses black-box testing
- Access to code is not required
- Allows us to identify the differences between users' and developers' perspectives. Can remove the bias towards code's functionality since tester and developer are different entities.

Disadvantages of requirements-based testing

- Limits coverage since only some test scenarios are performed
- Inefficient sometimes because of the lack of knowledge testers have about the software internal details

Advantages of coverage-based testing

- Internal software knowledge about SUT is beneficial for thorough testing. This results in efficiently finding errors and problems in code logic quicker and detecting security issues. It allows for code optimization
- Maximum coverage is obtained through the branch, line and method testing
- Uses white box testing

Disadvantages of white box testing

- May not find missing or unimplemented features
- Requires high-level knowledge of SUT internals
- Requires access to code

Referenced from: Farcic, V., Farcic, V., 12, dofree D., 12, ledasl D., & 14, V. F. P. authorD. (2013, December 13). *Black-box vs white-box testing*. Technology Conversations. Retrieved March 4, 2022, from <https://technologyconversations.com/2013/12/11/black-box-vs-white-box-testing/>

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

Measure data coverage flow manually was done as a group (Section 3.2) as it required concept use and tools usage, thus we decided to do it together. Each individual was given a set of methods in both the Range and the DataUtilities class. This is how our work division was done for adding additional test cases:

- Huda: public static double[][] clone(double[][] source), isNaNRange, hashCode
- Rajpreet: public static boolean equal(double[][] a, double[][] b), combineIgnoringNaN(Range range1, Range range2): Range
- Lubaba: shift (Range base, double delta, Boolean allowZeroCrossing), equal(Object obj)
- Nuha: expand(Range range, double lowerMargin, double upperMargin), expandToInclude(Range range, double value): Range, combine(Range range1, Range range2): Range

Additionally, we meet regularly for work sessions and to keep everyone on the same page and help each other with any struggles that we were having. The work was divided evenly and everyone contributed fairly.

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

We learned that white box testing enables testers to deep dive into the method and test various functionalities that make up the method by using converge tools such as EClemma. We gained a better understanding of how coverage testing is done and how the white-box testing principles are integrated. Some challenges we had were connecting the code coverage tools with Eclipse as a lot of them were incompatible and required lots of debugging, but after reading some documentation we were able to solve these issues. We had difficulties with the code2flow interactive code coverage tool which was used to draw the DFG but this allowed us to learn new software for data flow graphs. A lesson we learned is that the local variables accessed through the this-pointer are not defined and there are only uses in the method.

10 Comments/feedback on the lab itself

Overall, the lab allowed us to understand the core principles of the coverage criteria and how it is met during white box testing. The lab instructions and goals required more clarity as there was lots of questions that had to be asked to ensure what we did was what was actually required. Some feedback would be to clarify if the coverage metrics that need to be achieved meant we had to implement test cases for our non-selected methods in Range.java. Additionally, the code coverage tool rules and what was required with how many tools to be tested, how to install and navigate the tools, and just the overall initial setup was also difficult to follow.