Course Title: Software Testing, Reliability, and Quality
Course Code: SENG 438
Assignment #: 3
Student Names:

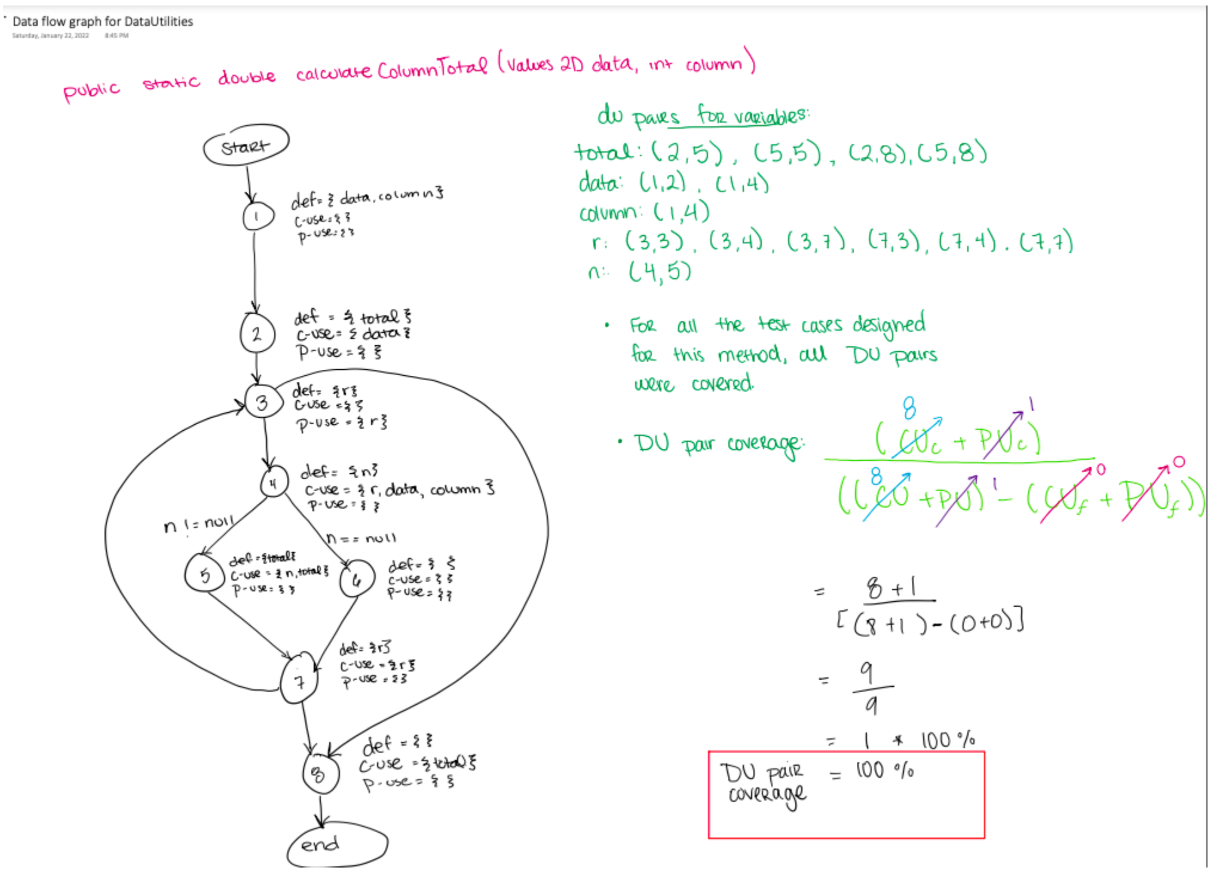
Aashik Ilangovan (30085993)
Emmanuel Omari-Osei (30092729)
Gibran Akmal (30094918)
Priyanka Gautam (30091244)

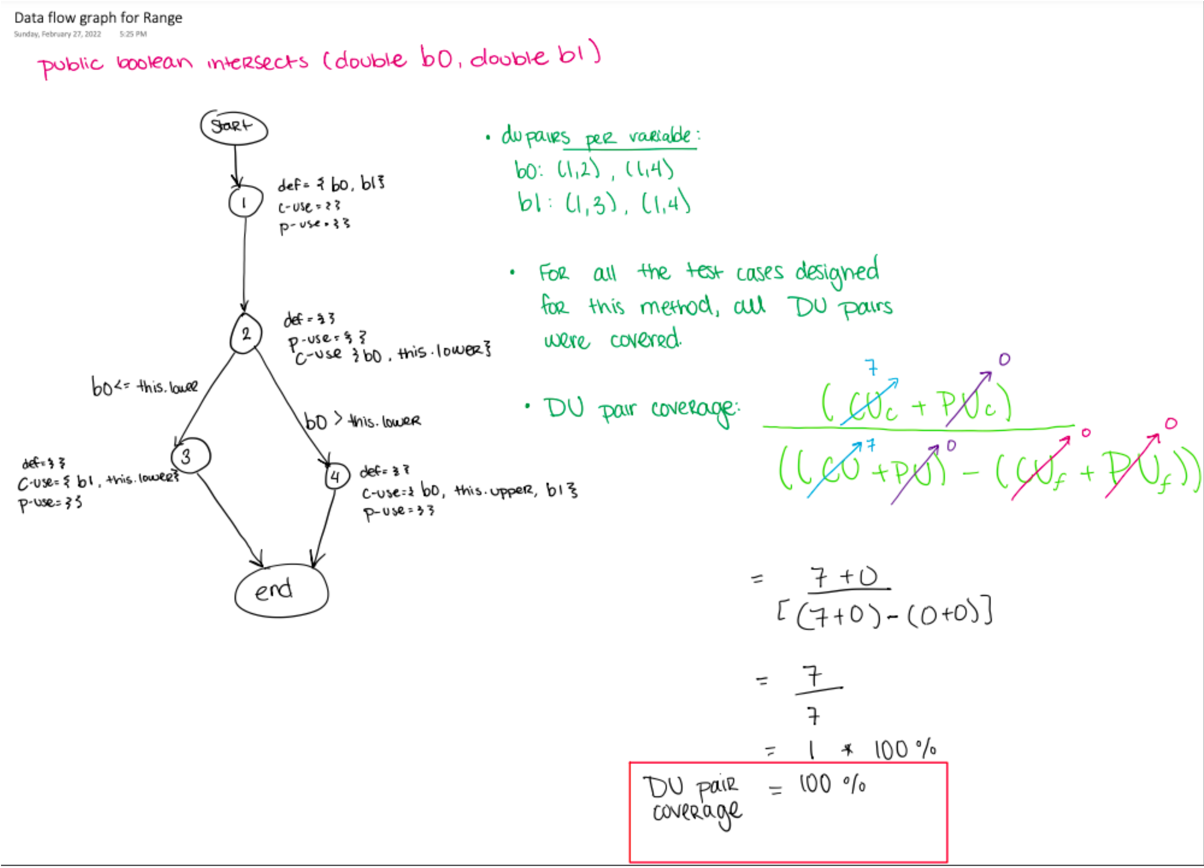
Group Number: 31
Submission Date: 4/03/2022

Introduction:

This report follows an in depth analysis of coverage testing and outlines the requirements found on the assignment sheet. The objectives of this assignment are to introduce the concepts of determining the adequacy of a white-box test suite based on its coverage of the code.

Manual Data-Flow Coverage Calculations the Two Methods





A detailed description of the testing strategy for the new unit tests:

The testing strategy we developed for new unit tests consisted of some key points. Our main goal was to maximise our coverage in the following three categories: branch coverage, method coverage, and line coverage (statement coverage). Firstly, branch coverage is dictated mainly by how many paths our test covers for our conditional outcomes (branches). Our strategy for going about this was to target methods and code with many if statements and nested if statements. These pieces of code consisted of many potential branches, so we achieved large branch coverage by writing tests for these sections of code. Secondly, method coverage is determined by the number of methods and functions covered by our test cases. Our decided plan to tackle as many methods as possible in the least amount

The tests to write were to target methods that are calling methods within them. This allowed us to be more efficient with our testing strategy, as we only had to write code for certain methods' functionalities. They would automatically test the functionality of methods called within them. This approach allowed us to minimise the number of tests we had to write as we avoided redundant writing tests for methods already being called within other methods. Lastly, line coverage is just the raw code or statements that our test cases cover. These include code within all branches and methods. We immediately saw large percentages of the line coverage in implementing the strategies used for the coverage mentioned above types. However, as much code as we tried to encapsulate with each line of our tests, there were certain methods that we unavoidably had to write extra test cases for to achieve the higher statement coverage.

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

Method 1

expandToInclude()

```
public static Range expandToInclude(Range range, double value) {
    if (range == null) {
        return new Range(value, value);
    }
    if (value < range.getLowerBound()) {
        return new Range(value, range.getUpperBound());
    }
    else if (value > range.getUpperBound()) {
        return new Range(range.getLowerBound(), value);
    }
    else {
        return range;
    }
}
```

The expandtoInclude method, as shown above is fully covered through our test cases (as highlighted in green).

As it can be seen, the method expandToInclude(Range range, double value) from Range class is entirely highlighted in green. This means that this method has been thoroughly tested, and has a 100% test coverage. The test cases for this method were designed to ensure that all logical branches, and all lines of the code in this method were tested. Through these test cases that we designed and executed, we increased our coverage for this method, which consequently increased the coverage for the whole Range class.

The first test case, rangeIsNull() is written and designed to test the first if statement in the SUT, which is checking to see whether the argument parameter range = = null. In this test case, the argument parameter *range* that was passed **is null** thus, the first logical branch of the if statement is executed. The expandToInclude(Range range, double value) method will return a new Range where the upper bound = *value* [1.0] (passed in from the argument) and lower bound = *value* [1.0] (passed in from the argument).

The second test case, rangeIsNotNull() is written and designed to test the first if statement in the SUT, which is checking to see whether the argument parameter range = = null. In this test case, the argument parameter *range* that was passed is **not null** thus, the second logical branch of the if statement is executed, which means that the else statement is actually responsible for returning an object of Range. The expandToInclude(Range range, double value) method will return the *range* (passed in from the argument).

The third test case, lessThanLower() is written and designed to test the second if statement in the SUT, which is checking to see whether the argument parameter *value* is less than the argument parameter *range's* lower bound. In this test case, the *value* is 0.5 which is lower than the *range's* lower bound of 1.0. Thus, the first logical branch of the if statement is executed. The expandToInclude(Range range, double value) method will return a new Range where the upper bound = *range's* upper bound [2.0] and lower bound = *value* [0.5](passed in from the argument).

The fourth test case, lessThanLower() is written and designed to test the else if statement in the SUT, which is checking to see whether the argument parameter *value* is greater than the argument parameter *range's* upper bound. In this test case, the *value* is 2.5 which is greater than the *range's* upper bound of 2.0. Thus, the first logical branch of the if statement is executed. The expandToInclude(Range range, double value) method will return a new Range where the upper bound = *value* [2.5](passed in from the argument) and the lower bound = *range's* lower bound [1.0].

Method 2

expand()

```java
    /**
     * Creates a new range by adding margins to an existing range.
     *
     * @param range  the range (<code>null</code> not permitted).
     * @param lowerMargin  the lower margin (expressed as a percentage of the
     *                     range length).
     * @param upperMargin  the upper margin (expressed as a percentage of the
     *                     range length).
     *
     * @return The expanded range.
     */
    public static Range expand(Range range,
                               double lowerMargin, double upperMargin) {
        ParamChecks.nullNotPermitted(range, "range");
        double length = range.getLength();
        double lower = range.getLowerBound() - length * lowerMargin;
        double upper = range.getUpperBound() + length * upperMargin;
        if (lower > upper) {
            lower = lower / 2.0 + upper / 2.0;
            upper = lower;
        }
        return new Range(lower, upper);
    }
```

The expand method, as shown above is fully covered through our test cases (as highlighted in green).

As it can be seen, the method expand(Range range, double lowerMargin, double upperMargin) from Range class is entirely highlighted in green. This means that this method has been thoroughly tested, and has a 100% test coverage. The test cases for this method were designed to ensure that all logical branches, and all lines of the code in this method were tested. Through these test cases that we designed and executed, we increased our coverage for this method, which consequently increased the coverage for the whole Range class.

The first test case, RangeAppropriateValuesTest() is written and designed to test the first if statement in the SUT, which is checking to see whether the argument parameter range **new lower bound is less than the range's new upper bound**. In this test case, the argument parameter *range* that was passed **has a new lower bound of -34.0 and a new upper bound of 74.0** thus, the second logical branch of the if statement is executed. The expand(Range range, double lowerMargin, double upperMargin) method will return a new Range where the upper bound = 20 + 9 (length of range) * 6 (upper margin passed in from the argument) and lower bound = 11 - 9 (length of range) * 5 (lower margin passed in from the argument).

The second test case, RangeAppropriateValuesTest() is written and designed to test the first if statement in the SUT, which is checking to see whether the argument parameter range **new lower bound is greater than the range's new upper bound**. In this test case, the argument parameter *range* that was passed **has a new lower bound of 56.0 and a new upper bound of -16.0** thus, the first logical branch of the if statement is executed because lower is less than upper. The expand(Range range, double lowerMargin, double upperMargin) method will return a new Range where the lower bound = [11 - 9 (length of range) * 5 (lower margin passed in from the argument)] / 2.0 + (-16.0) / 2.0 = 20.0 and the upper bound is 20.0.

Method 3

shift()

```
/**
 * Shifts the range by the specified amount.
 *
 * @param base  the base range (<code>null</code> not permitted).
 * @param delta  the shift amount.
 *
 * @return A new range.
 */
public static Range shift(Range base, double delta) {
    return shift(base, delta, false);
}

/**
 * Shifts the range by the specified amount.
 *
 * @param base  the base range (<code>null</code> not permitted).
 * @param delta  the shift amount.
 * @param allowZeroCrossing  a flag that determines whether or not the
 *                           bounds of the range are allowed to cross
 *                           zero after adjustment.
 *
 * @return A new range.
 */
public static Range shift(Range base, double delta,
                          boolean allowZeroCrossing) {
    ParamChecks.nullNotPermitted(base, "base");
    if (allowZeroCrossing) {
        return new Range(base.getLowerBound() + delta,
                base.getUpperBound() + delta);
    }
    else {
        return new Range(shiftWithNoZeroCrossing(base.getLowerBound(),
                delta), shiftWithNoZeroCrossing(base.getUpperBound(),
                delta));
    }
}
```

The method *shift(Range base, double delta, boolean allowZeroCrossing)* and its overloaded function *shift(Range base, double delta)* from Range class have been completely covered through testing, as visible above. Thus, the methods have 100% test coverage. The test cases for these methods were designed to ensure that all logical branches and all lines of code were tested. These test cases served to increase our test coverage for this method, which consequently increased the coverage for the whole Range class.

The first test case, *shiftBaseRangeIsNull()* is written and designed to test the functionality of the *shift(Range base, double delta)* method when the parameter *base* = = *null*. In this test case, the parameter *base* was passed with a *null* value. The *shift(Range base, double delta)* method will throw an exception in response to this input. When testing, the test case passed.

The second test case, *shiftNotAllowingZeroCrossingWithDeltaNotEqualZero()* is written and designed to test the *else* branch of the *if / else* statement in the SUT, which is checking to see whether the parameter *allowZeroCrossing* = = *false*. In this test case, the argument parameter *allowZeroCrossing* of *shift(Range base, double delta, boolean allowZeroCrossing)* was passed as *false*. This is done automatically in the overloaded method *shift(Range base, double delta)*, so it is used in this test to increase line coverage. As the method was tested, the *else* component of the logical branch of the statement was executed. This correctly outputs a new *Range* object appropriately shifted, meaning the test case passed.

The third test case, *shiftAllowZeroCrossingGivenZero*() is written and designed to test the *if* branch of the *if / else* statement in the SUT, which is checking to see whether the parameter *allowZeroCrossing* = = *true*. In this test case, the argument parameter *allowZeroCrossing* of *shift(Range base, double delta, boolean allowZeroCrossing)* was passed as *false*. Thus, the first logical branch of the if statement is executed. As the method was tested, the *if* component of the logical branch of the statement was executed. This correctly outputs a new *Range* object appropriately shifted, meaning the test case passed.

Method 4

CalculateRowTotal()

```java
/**
 * Returns the total of the values in one row of the supplied data
 * table by taking only the column numbers in the array into account.
 *
 * @param data  the table of values (<code>null</code> not permitted).
 * @param row  the row index (zero-based).
 * @param validCols the array with valid cols (zero-based).
 *
 * @return The total of the valid values in the specified row.
 *
 * @since 1.0.13
 */
public static double calculateRowTotal(Values2D data, int row,
         int[] validCols) {
    ParamChecks.nullNotPermitted(data, "data");
    double total = 0.0;
    int colCount = data.getColumnCount();
    for (int v = 0; v < validCols.length; v++) {
        int col = validCols[v];
        if (col < colCount) {
            Number n = data.getValue(row, col);
            if (n != null) {
                total += n.doubleValue();
            }
        }
    }
```

As it can be seen, the method calculateRowTotal(Values2D data, int row, int[] validCols) from the DataUtilities class is entirely highlighted in green. This means that this method has been thoroughly tested, and has a 100% test coverage. The test cases for this method were designed to ensure that all logical branches, and all lines of the code in this method were tested. Through these test cases that we designed and executed, we increased our coverage for this method, which consequently increased the coverage for the whole DataUtilities class.

The first test case, calculateRowTotal_NothingNull() is written and designed to test the first if statement in the SUT, which is checking to see whether the argument parameter data = = null. In this test case, the argument parameter *date* that was passed **is not null** thus, the first logical branch of the if statement is executed. Also the number of rows set in the test case are within the valid range that was set by the parameter: “validCols”. Therefore this test should return the correct row total of 33.0 according the the assigned values preset for the rows.

The second test case, calculateRowTotal_NullWithinRange() is written and designed to test another logical branch of the method, in checking what happens when a single row entry is null but the rest are not. In this test, the value at the 1st row and 3rd column is set to null while all the other row values are not null. Also the “validCols” argument passed in is not null and the number of rows set in the setup are within that range. Therefore this tests to see if the null value in that single row/column entry affects the calculation. It is expected that the null value is ignored and the rest of the not null row values are added as normal to return the total of 20.0

The third test case, calculateRowTotal_NullExact() is written and designed to test the second if statement in the SUT. In this test case the argument “validCols” passed in is exactly equal to the number of columns being set in the @setup section. As well as just like in the previous test case, the final entry in that row is set to null. This should not change much as the method is expected once again to ignore the null entry in the row total calculations and we should be returned the value of 20.0.

Method 5

clone()

```
public static double[][] clone(double[][] source) {
    ParamChecks.nullNotPermitted(source, "source");
    double[][] clone = new double[source.length][];
    for (int i = 0; i < source.length; i++) {
        if (source[i] != null) {
            double[] row = new double[source[i].length];
            System.arraycopy(source[i], 0, row, 0, source[i].length);
            clone[i] = row;
        }
    }
    return clone;
}
```

The method clone(double[][] source) from DataUtilities class is entirely highlighted in green. This means that this method has been thoroughly tested, and has a 100% test coverage. The test cases for this method were designed to ensure that all logical branches, and all lines of the code in this method were tested. Through these test cases that we designed and executed, we increased our coverage for this method, which consequently increased the coverage for the whole DataUtilities class.

The first test case, sourceIsNull() is written and designed to test the first if statement in the SUT, which is checking to see if the argument parameter source[i] != null, is null. In this test case, the argument parameter *source[i]* that was passed **is null** thus, the if statement is skipped in the for-loop. The clone(double[][] source) method will clone back a double of null.

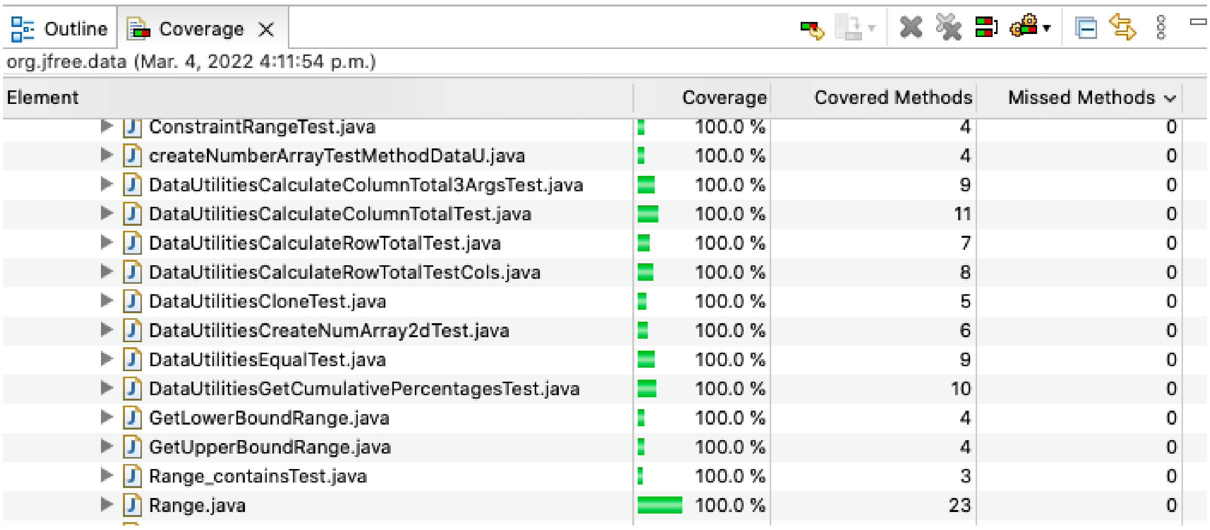
The second test case, sourceIsValid() is written and designed to also test the first if statement in the SUT, which is checking to see whether the argument parameter source[i] != null is not null. In this test case, the argument parameter *source[i]* that was passed is **not null** thus, the logical branch of the if statement is executed, which means the if statement will follow through. The clone(double[][] source) method will clone back the double that was inputted.

The third and final test case,sourceElementIsNULL() is written and designed to also test the first if statement in the SUT, which is checking to see whether the argument parameter source[i] != null is not null. In this test case, the argument parameter *source[i]* that was passed has an element that is null and an element is not not null; we input the following {null, {1,0}} as our input. The logical branch of the if statement is executed, which means the if statement will follow through. The clone(double[][] source) method will clone back the double that was inputted even with part of the input being null.

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

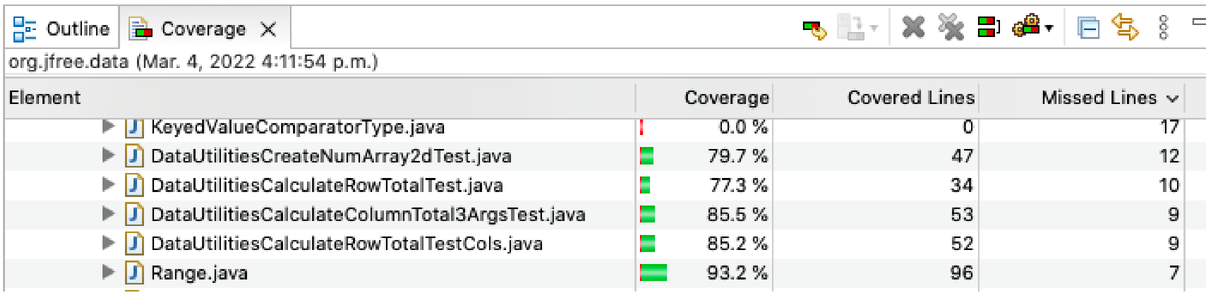
Range

Method Counter ~ 100.0%



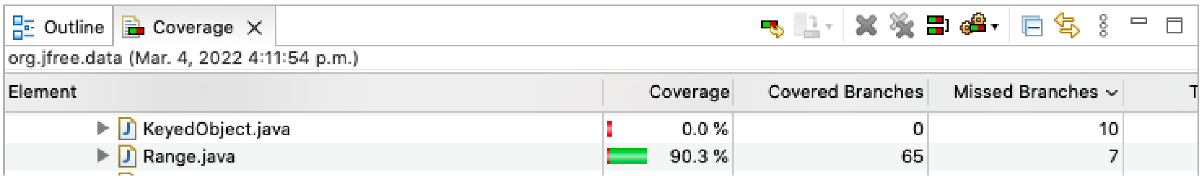
Element	Coverage	Covered Methods	Missed Methods
ConstraintRangeTest.java	100.0 %	4	0
createNumberArrayTestMethodDataU.java	100.0 %	4	0
DataUtilitiesCalculateColumnTotal3ArgsTest.java	100.0 %	9	0
DataUtilitiesCalculateColumnTotalTest.java	100.0 %	11	0
DataUtilitiesCalculateRowTotalTest.java	100.0 %	7	0
DataUtilitiesCalculateRowTotalTestCols.java	100.0 %	8	0
DataUtilitiesCloneTest.java	100.0 %	5	0
DataUtilitiesCreateNumArray2dTest.java	100.0 %	6	0
DataUtilitiesEqualTest.java	100.0 %	9	0
DataUtilitiesGetCumulativePercentagesTest.java	100.0 %	10	0
GetLowerBoundRange.java	100.0 %	4	0
GetUpperBoundRange.java	100.0 %	4	0
Range_containsTest.java	100.0 %	3	0
Range.java	100.0 %	23	0

Line Counter ~ 93.2%



Element	Coverage	Covered Lines	Missed Lines
KeyedValueComparatorType.java	0.0 %	0	17
DataUtilitiesCreateNumArray2dTest.java	79.7 %	47	12
DataUtilitiesCalculateRowTotalTest.java	77.3 %	34	10
DataUtilitiesCalculateColumnTotal3ArgsTest.java	85.5 %	53	9
DataUtilitiesCalculateRowTotalTestCols.java	85.2 %	52	9
Range.java	93.2 %	96	7

Branch Counter ~ 90.3%



Element	Coverage	Covered Branches	Missed Branches
KeyedObject.java	0.0 %	0	10
Range.java	90.3 %	65	7

```java
    /**
     * Returns the lower bound for the range.
     *
     * @return The lower bound.
     */
    public double getLowerBound() {
        return this.lower;
    }

    /**
     * Returns the upper bound for the range.
     *
     * @return The upper bound.
     */
    public double getUpperBound() {
        return this.upper;
    }

    /**
     * Returns the length of the range.
     *
     * @return The length.
     */
    public double getLength() {
        return this.upper - this.lower;
    }

    /**
     * Returns the central value for the range.
     *
     * @return The central value.
     */
    public double getCentralValue() {
        return this.lower / 2.0 + this.upper / 2.0;
    }
```

```java
    /**
     * Returns <code>true</code> if the range contains the specified value and
     * <code>false</code> otherwise.
     *
     * @param value  the value to lookup.
     *
     * @return <code>true</code> if the range contains the specified value.
     */
    public boolean contains(double value) {
        return (value >= this.lower && value <= this.upper);
    }

    /**
     * Returns <code>true</code> if the range intersects with the specified
     * range, and <code>false</code> otherwise.
     *
     * @param b0  the lower bound (should be &lt;= b1).
     * @param b1  the upper bound (should be &gt;= b0).
     *
     * @return A boolean.
     */
    public boolean intersects(double b0, double b1) {
        if (b0 <= this.lower) {
            return (b1 > this.lower);
        }
        else {
            return (b0 < this.upper && b1 >= b0);
        }
    }

    /**
     * Returns <code>true</code> if the range intersects with the specified
     * range, and <code>false</code> otherwise.
     *
     * @param range  another range (<code>null</code> not permitted).
     *
     * @return A boolean.
     *
     * @since 1.0.9
     */
    public boolean intersects(Range range) {
        return intersects(range.getLowerBound(), range.getUpperBound());
    }
```

```java

    /**
     * Returns the value within the range that is closest to the specified
     * value.
     *
     * @param value  the value.
     *
     * @return The constrained value.
     */
    public double constrain(double value) {
        double result = value;
        if (!contains(value)) {
            if (value > this.upper) {
                result = this.upper;
            }
            else if (value < this.lower) {
                result = this.lower;
            }
        }
        return result;
    }

    /**
     * Creates a new range by combining two existing ranges.
     * <P>
     * Note that:
     * <ul>
     *   <li>either range can be <code>null</code>, in which case the other
     *       range is returned;</li>
     *   <li>if both ranges are <code>null</code> the return value is
     *       <code>null</code>.</li>
     * </ul>
     *
     * @param range1  the first range (<code>null</code> permitted).
     * @param range2  the second range (<code>null</code> permitted).
     *
     * @return A new range (possibly <code>null</code>).
     */
    public static Range combine(Range range1, Range range2) {
        if (range1 == null) {
            return range2;
        }
        if (range2 == null) {
            return range1;
        }
        double l = Math.min(range1.getLowerBound(), range2.getLowerBound());
        double u = Math.max(range1.getUpperBound(), range2.getUpperBound());
        return new Range(l, u);
    }
    /**
     * Returns a new range that spans both <code>range1</code> and
     * <code>range2</code>.  This method has a special handling to ignore
     * Double.NaN values.
     *
     * @param range1  the first range (<code>null</code> permitted).
     * @param range2  the second range (<code>null</code> permitted).
     *
     * @return A new range (possibly <code>null</code>).
     *
     * @since 1.0.15
     */
    public static Range combineIgnoringNaN(Range range1, Range range2) {
        if (range1 == null) {
            if (range2 != null && range2.isNaNRange()) {
                return null;
            }
            return range2;
        }
        if (range2 == null) {
            if (range1.isNaNRange()) {
                return null;
            }
            return range1;
        }
        double l = min(range1.getLowerBound(), range2.getLowerBound());
        double u = max(range1.getUpperBound(), range2.getUpperBound());
        if (Double.isNaN(l) && Double.isNaN(u)) {
            return null;
        }
        return new Range(l, u);
    }
```

```java
    private static double max(double d1, double d2) {
        if (Double.isNaN(d1)) {
            return d2;
        }
        if (Double.isNaN(d2)) {
            return d1;
        }
        return Math.max(d1, d2);
    }

    /**
     * Returns a range that includes all the values in the specified
     * <code>range</code> AND the specified <code>value</code>.
     *
     * @param range  the range (<code>null</code> permitted).
     * @param value  the value that must be included.
     *
     * @return A range.
     *
     * @since 1.0.1
     */
    public static Range expandToInclude(Range range, double value) {
        if (range == null) {
            return new Range(value, value);
        }
        if (value < range.getLowerBound()) {
            return new Range(value, range.getUpperBound());
        }
        else if (value > range.getUpperBound()) {
            return new Range(range.getLowerBound(), value);
        }
        else {
            return range;
        }
    }

    /**
     * Creates a new range by adding margins to an existing range.
     *
     * @param range  the range (<code>null</code> not permitted).
     * @param lowerMargin  the lower margin (expressed as a percentage of the
     *                     range length).
     * @param upperMargin  the upper margin (expressed as a percentage of the
     *                     range length).
     *
     * @return The expanded range.
     */
    public static Range expand(Range range,
                               double lowerMargin, double upperMargin) {
        ParamChecks.nullNotPermitted(range, "range");
        double length = range.getLength();
        double lower = range.getLowerBound() - length * lowerMargin;
        double upper = range.getUpperBound() + length * upperMargin;
        if (lower > upper) {
            lower = lower / 2.0 + upper / 2.0;
            upper = lower;
        }
        return new Range(lower, upper);
    }

    /**
     * Shifts the range by the specified amount.
     *
     * @param base  the base range (<code>null</code> not permitted).
     * @param delta  the shift amount.
     *
     * @return A new range.
     */
    public static Range shift(Range base, double delta) {
        return shift(base, delta, false);
    }
```

```java

    /**
     * Shifts the range by the specified amount.
     *
     * @param base  the base range (<code>null</code> not permitted).
     * @param delta  the shift amount.
     * @param allowZeroCrossing  a flag that determines whether or not the
     *                           bounds of the range are allowed to cross
     *                           zero after adjustment.
     *
     * @return A new range.
     */
    public static Range shift(Range base, double delta,
                              boolean allowZeroCrossing) {
        ParamChecks.nullNotPermitted(base, "base");
        if (allowZeroCrossing) {
            return new Range(base.getLowerBound() + delta,
                    base.getUpperBound() + delta);
        }
        else {
            return new Range(shiftWithNoZeroCrossing(base.getLowerBound(),
                    delta), shiftWithNoZeroCrossing(base.getUpperBound(),
                    delta));
        }
    }
```

```java
    /**
     * Scales the range by the specified factor.
     *
     * @param base the base range (<code>null</code> not permitted).
     * @param factor the scaling factor (must be non-negative).
     *
     * @return A new range.
     *
     * @since 1.0.9
     */
    public static Range scale(Range base, double factor) {
        ParamChecks.nullNotPermitted(base, "base");
        if (factor < 0) {
            throw new IllegalArgumentException("Negative 'factor' argument.");
        }
        return new Range(base.getLowerBound() * factor,
                base.getUpperBound() * factor);
    }

    /**
     * Tests this object for equality with an arbitrary object.
     *
     * @param obj  the object to test against (<code>null</code> permitted).
     *
     * @return A boolean.
     */
    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Range)) {
            return false;
        }
        Range range = (Range) obj;
        if (!(this.lower == range.lower)) {
            return false;
        }
        if (!(this.upper == range.upper)) {
            return false;
        }
        return true;
    }
```

```java
	@Override
	public int hashCode() {
		int result;
		long temp;
		temp = Double.doubleToLongBits(this.lower);
		result = (int) (temp ^ (temp >>> 32));
		temp = Double.doubleToLongBits(this.upper);
		result = 29 * result + (int) (temp ^ (temp >>> 32));
		return result;
	}

	/**
	 * Returns a string representation of this Range.
	 *
	 * @return A String "Range[lower,upper]" where lower=lower range and
	 *         upper=upper range.
	 */
	@Override
	public String toString() {
		return ("Range[" + this.lower + "," + this.upper + "]");
	}

}
```

DataUtilities

Method Counter ~ 90.0%

Outline | Coverage

org.jfree.data (Mar. 4, 2022 4:11:54 p.m.)

Element	Coverage	Covered Methods	Missed Methods
KeyedValueComparator.java	0.0 %	0	4
DataUtilitiesCalculatedRowTotal.java	75.0 %	6	2
DataUtilities.java	90.0 %	9	1

Line Counter ~ 98.8%

Outline | Coverage

org.jfree.data (Mar. 4, 2022 4:11:54 p.m.)

Element	Coverage	Covered Lines	Missed Lines
KeyedValueComparatorType.java	0.0 %	0	17
DataUtilitiesCreateNumArray2dTest.java	79.7 %	47	12
DataUtilitiesCalculateRowTotalTest.java	77.3 %	34	10
DataUtilitiesCalculateColumnTotal3ArgsTest.java	85.5 %	53	9
DataUtilitiesCalculateRowTotalTestCols.java	85.2 %	52	9
Range.java	93.2 %	96	7
RangeScaleTest.java	60.0 %	9	6
ConstraintRangeTest.java	55.6 %	5	4
RangeCombineIgnoringNaNTest.java	92.0 %	46	4
DataUtilitiesCloneTest.java	76.9 %	10	3
RangeEqualsTest.java	87.5 %	21	3
RangeShiftTest.java	85.7 %	18	3
createNumberArrayTestMethodDataU.java	86.7 %	13	2
UnknownKeyException.java	0.0 %	0	2
DataUtilities.java	98.8 %	79	1

Branch Counter ~ 95.8%

Outline | Coverage

org.jfree.data (Mar. 4, 2022 4:11:54 p.m.)

Element	Coverage	Covered Branches	Missed Branches	T
KeyedValueComparatorType.java	0.0 %	0	6	
DataUtilitiesCalculateColumnTotalTest.java	16.7 %	1	5	
DataUtilitiesCalculateRowTotalTest.java	16.7 %	1	5	
DataUtilitiesGetCumulativePercentagesTest.java	25.0 %	1	3	
DataUtilities.java	95.8 %	46	2	

```java
    /**
     * Tests two arrays for equality.  To be considered equal, the arrays must
     * have exactly the same dimensions, and the values in each array must also
     * match (two values that are both NaN or both INF are considered equal
     * in this test).
     *
     * @param a  the first array (<code>null</code> permitted).
     * @param b  the second array (<code>null</code> permitted).
     *
     * @return A boolean.
     *
     * @since 1.0.13
     */
    public static boolean equal(double[][] a, double[][] b) {
        if (a == null) {
            return (b == null);
        }
        if (b == null) {
            return false;  // already know 'a' isn't null
        }
        if (a.length != b.length) {
            return false;
        }
        for (int i = 0; i < a.length; i++) {
            if (!Arrays.equals(a[i], b[i])) {
                return false;
            }
        }
        return true;
    }
```

```java
    /**
     * Returns a clone of the specified array.
     *
     * @param source  the source array (<code>null</code> not permitted).
     *
     * @return A clone of the array.
     *
     * @since 1.0.13
     */
    public static double[][] clone(double[][] source) {
        ParamChecks.nullNotPermitted(source, "source");
        double[][] clone = new double[source.length][];
        for (int i = 0; i < source.length; i++) {
            if (source[i] != null) {
                double[] row = new double[source[i].length];
                System.arraycopy(source[i], 0, row, 0, source[i].length);
                clone[i] = row;
            }
        }
        return clone;
    }

    /**
     * Returns the total of the values in one column of the supplied data
     * table.
     *
     * @param data  the table of values (<code>null</code> not permitted).
     * @param column  the column index (zero-based).
     *
     * @return The total of the values in the specified column.
     */
    public static double calculateColumnTotal(Values2D data, int column) {
        ParamChecks.nullNotPermitted(data, "data");
        double total = 0.0;
        int rowCount = data.getRowCount();
        for (int r = 0; r < rowCount; r++) {
            Number n = data.getValue(r, column);
            if (n != null) {
                total += n.doubleValue();
            }
        }
        return total;
    }
```

```java
    /**
     * Returns the total of the values in one column of the supplied data
     * table by taking only the row numbers in the array into account.
     *
     * @param data  the table of values (<code>null</code> not permitted).
     * @param column  the column index (zero-based).
     * @param validRows the array with valid rows (zero-based).
     *
     * @return The total of the valid values in the specified column.
     *
     * @since 1.0.13
     */
    public static double calculateColumnTotal(Values2D data, int column,
             int[] validRows) {
        ParamChecks.nullNotPermitted(data, "data");
        double total = 0.0;
        int rowCount = data.getRowCount();
        for (int v = 0; v < validRows.length; v++) {
            int row = validRows[v];
            if (row < rowCount) {
                Number n = data.getValue(row, column);
                if (n != null) {
                    total += n.doubleValue();
                }
            }
        }
        return total;
    }

    /**
     * Returns the total of the values in one row of the supplied data
     * table.
     *
     * @param data  the table of values (<code>null</code> not permitted).
     * @param row  the row index (zero-based).
     *
     * @return The total of the values in the specified row.
     */
    public static double calculateRowTotal(Values2D data, int row) {
        ParamChecks.nullNotPermitted(data, "data");
        double total = 0.0;
        int columnCount = data.getColumnCount();
        for (int c = 0; c < columnCount; c++) {
            Number n = data.getValue(row, c);
            if (n != null) {
                total += n.doubleValue();
            }
        }
        return total;
    }
```

```java
    /**
     * Returns the total of the values in one row of the supplied data
     * table by taking only the column numbers in the array into account.
     *
     * @param data  the table of values (<code>null</code> not permitted).
     * @param row  the row index (zero-based).
     * @param validCols the array with valid cols (zero-based).
     *
     * @return The total of the valid values in the specified row.
     *
     * @since 1.0.13
     */
    public static double calculateRowTotal(Values2D data, int row,
             int[] validCols) {
        ParamChecks.nullNotPermitted(data, "data");
        double total = 0.0;
        int colCount = data.getColumnCount();
        for (int v = 0; v < validCols.length; v++) {
            int col = validCols[v];
            if (col < colCount) {
                Number n = data.getValue(row, col);
                if (n != null) {
                    total += n.doubleValue();
                }
            }
        }
        return total;
    }

    /**
     * Constructs an array of <code>Number</code> objects from an array of
     * <code>double</code> primitives.
     *
     * @param data  the data (<code>null</code> not permitted).
     *
     * @return An array of <code>Double</code>.
     */
    public static Number[] createNumberArray(double[] data) {
        ParamChecks.nullNotPermitted(data, "data");
        Number[] result = new Number[data.length];
        for (int i = 0; i < data.length; i++) {
            result[i] = new Double(data[i]);
        }
        return result;
    }
```

```java
    /**
     * Constructs an array of arrays of <code>Number</code> objects from a
     * corresponding structure containing <code>double</code> primitives.
     *
     * @param data  the data (<code>null</code> not permitted).
     *
     * @return An array of <code>Double</code>.
     */
    public static Number[][] createNumberArray2D(double[][] data) {
        ParamChecks.nullNotPermitted(data, "data");
        int l1 = data.length;
        Number[][] result = new Number[l1][];
        for (int i = 0; i < l1; i++) {
            result[i] = createNumberArray(data[i]);
        }
        return result;
    }

    /**
     * Returns a {@link KeyedValues} instance that contains the cumulative
     * percentage values for the data in another {@link KeyedValues} instance.
     * <p>
     * The percentages are values between 0.0 and 1.0 (where 1.0 = 100%).
     *
     * @param data  the data (<code>null</code> not permitted).
     *
     * @return The cumulative percentages.
     */
    public static KeyedValues getCumulativePercentages(KeyedValues data) {
        ParamChecks.nullNotPermitted(data, "data");
        DefaultKeyedValues result = new DefaultKeyedValues();
        double total = 0.0;
        for (int i = 0; i < data.getItemCount(); i++) {
            Number v = data.getValue(i);
            if (v != null) {
                total = total + v.doubleValue();
            }
        }
        double runningTotal = 0.0;
        for (int i = 0; i < data.getItemCount(); i++) {
            Number v = data.getValue(i);
            if (v != null) {
                runningTotal = runningTotal + v.doubleValue();
            }
            result.addValue(data.getKey(i), new Double(runningTotal / total));
        }
        return result;
    }

}
```

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

Requirements-based testing is where the test cases are developed by obtaining information from the requirements. Advantages of this testing approach allow the testers to perform both functional and non functional tests. Attributes such as performance, reliability and usability of the system are tested through this approach. An advantage of this type of testing is that it is easy to validate if the requirements of the system are correct. For example, in assignment 2, when using the black-box approach, we used requirements based testing to write test suites. These test suites were able to test the qualitative aspects of the SUT. It ensured the functionality of the methods was behaving appropriately. Disadvantages of this type of testing is that there are a greater number of unidentified bugs. This is because since you cannot actually see how much of the source code your tests have covered, you cannot know for certain if you have tested for every logical path. Since there are bound to be paths that have not been tested, there could be potential bugs in the code that go undetected which weakens the reliability of the whole SUT.

Coverage-based testing is determining how much of the source code is being tested. This type of approach encompasses branch, statement or function coverage. An advantage of this technique is that through this approach, the testers are able to assess if more test cases are needed in their test suite to make sure they reach a certain coverage metric for their SUT. This testing approach also makes the test suite more thorough as the testers can make sure all logical paths of the source code are tested. For example, in this assignment, when using white box approach, we used coverage based testing to write test suites. These test suites were so thorough, that they were able to ensure if the SUT was logically consistent. Since these test cases in the test suite were developed to cover all possible paths that the input could take in the source code, it produced a high coverage which consequently means that the chances of finding unidentified bugs in the code were low. This is one of the biggest benefits of using this technique compared to the other technique. We were able to quantitatively assess our test suites. Disadvantages of this type of testing is that there are several coverage tools that can be utilised to measure coverage. Since different coverage tools can be used, and if the tool is not consistent across the team, then you will not be able to compare the code coverage results. One code coverage's tool's result might be completely different from another code coverage's tool. Lastly, one of the disadvantages of coverage based testing is finding the appropriate coverage tool that suits your specific requirements. For example, in this lab, my group used the coverage tool EclEmma. This coverage tool only had branch, line and method counters whereas the requirements of the assignment required us to find branch, statement and condition coverage. We had to substitute method coverage requirement to condition coverage, as indicated in the lab document.

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.

The coverage tool our team used in this lab assessment was EclEmma. This tool coverage now comes built into the new Eclipse IDE configuration upon downloading Eclipse. This made the coverage tool installation very straightforward for our team since we all had up-to-date versions of Eclipse running on our computers, as they were needed for the previous lab assignment.

With the installation of the software out of the way, our team followed the lab report instructions on setting up coverage tests using EclEmma. This process was also made fairly simple in our group as this application had tools specially made for the Eclipse IDE that made running coverage tests just as easy as running them. Thus, we could conclude that EclEmma was optimally integrated for our IDE (Eclipse).

Our group faced a challenge with this plug-in when performing condition coverage. EclEmma does not have any direct functionality that allows the condition coverage metric to be reported. Thankfully, it is detailed in the lab report that groups are allowed to report on method coverage instead of condition coverage for using EclEmma. While this allowed our group to carry on with the lab normally, for another testing situation this could pose a major hurdle and might require switching to another coverage algorithm.

Aside from the absence of condition coverage, the plug-in was user friendly and made it clear which classes we were covering and by how much. This was displayed using visuals of green and red highlights on code that were covered and not covered respectively. Overall, we had a fairly good experience with the EclEmma coverage testing software.

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

Teamwork is a fundamental core part of software testing. In order to go through the test we read and did the preliminary parts of the test as a group (the setups in part 1) and proceeded to go through one testing process in DataUtilities together as well. This was to ensure that everyone was up to par in the standard, format and overall understanding of coverage testing and being able to read and interpret code coverage amounts. From there, we would proceed to have multiple group sessions where we worked individually on our own tests but configured with each other whenever needed.

A lesson learned throughout the entire process was that having more than one pair of eyes on an issue can provide faster and better solutions. For example, as we were first running through a test as a group, all of us had input our knowledge and ideas on how to set up the tests and do the test from previous courses. It provided an efficient process into creating a test for the example method as well as filled any knowledge gaps that we had going into the lab, allowing us to work on our own split individual tests as well. Each person was responsible for writing test cases for at least two methods and documenting their strategy. Furthermore, collaboration on one or two test cases allowed us to formulate the best estimations on how to increase overall coverage through trial and error and input from other team members. That way, when we went to individually work on each of our own tests, we already knew the main things to target to increase each type of coverage and did not have to spend a lot of time experimenting by ourselves.

Teamwork is incredibly important in software development as “teamwork makes the dream work.” Learning how to work in a team is imperative for engineers across all disciplines because it introduces them to varying new perspectives that only strengthen the final product. One of the key lessons that we learned about teamwork is that open communication, transparency and hard work are the foundations of any successful group.

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

There were a few challenges and lessons that came about while performing through the assignment. The first was downloading and implementing the coverage tools such as EclEmma and getting it to work with our individual ide’s and computers. Next, came the testing part, to which we had discovered multiple errors with our previous tests that we had not yet encountered prior to this assignment. This caused a significant setback in pacing as we had to fix multiple errors before even beginning the coverage part of the assignment. After running our tests, we had discovered our percentages that we had covered were quite low (~10%), and so we had to create multiple additional tests to meet the percentage requirements. It challenged us to test different methods from both classes and we were able to see certain strategic similarities among methods in terms of specific things to target to increase different types of coverage. Overcoming these challenges furthered our understanding of both unit testing and coverage testing as a whole. It also increased our efficiency in writing tests with our goal being to write the least number of tests with the most amount of coverage. The challenges we overcome the best include some of other

Comments/feedback on the assignment itself:

Overall, the lab detailed a simple yet insightful experience in coverage testing. The experience induced a healthy working environment that explored not only our previous knowledge of the course material but more importantly our ability to cooperate in group settings. We discovered in our first run-through of using coverage testing the application produced minor difficulties across various operating systems which was important to understand and resolve, as all team members took on a set of test cases to complete. However, going through the lab we were able to quickly overcome such difficulties, turning it into a smooth and informative session. An improvement that could be implemented in the lab document itself would be the adding of figures in sections 3.2 and 3.3. Coverage testing also provided a more thorough insight into the application of these tests in the real world and helped us understand the magnitude of testing (considering we had to implement many extra tests in order to fulfil the required coverage percentages) All in all, we found this lab easy to follow and an excellent introduction to coverage testing in all aspects.