SENG 438 Lab 3 - Software Testing, Reliability, and Quality

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

Group #: 6 - Student Names: Neha Parmar, Shahed Issa, Jaisumer Sandhu, Jaival Patel

Introduction

This lab report explores white-box testing techniques to improve code coverage for the DataUtilities and Range classes in JFreeChart. Using JUnit for unit testing and EclEmma for code coverage analysis, we aim to increase statement, branch, and condition coverage by identifying and testing unexecuted paths in the source code. By focusing on testing adequacy, we ensure that key methods and decision points are properly exercised. This report details our approach, challenges, and the impact of our enhanced test suite on overall code reliability.

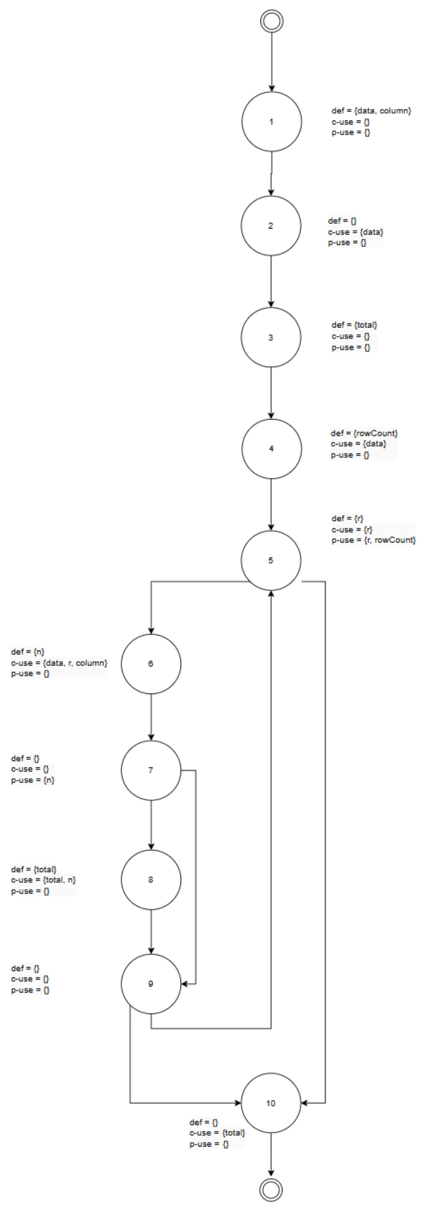
Manual data-flow coverage calculations for methods

DataUtilities.CalculateColumnTotal:

Data flow graph:

code:

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



DU-Pairs by variable:

<u>Variable: data</u>
def(1) → use(2)
def(1) → use(4)
def(1) → use(6)

<u>Variable: total</u>
def(2) → use(8)
def(2) → use(10)
def(8) → use(8)
def(8) → use(10)

<u>Variable: column</u>
def(1) → use(6)

<u>Variable: rowCount</u>
def(4) → use(5)

<u>Variable: r</u>
def(5) → use(5) (loop condition & increment)
def(5) → use(6)

<u>Variable: n</u>
def(6) → use(7)
def(6) → use(8)

Test Case Pair coverage:

Test Case 1: 12/13 pairs

<u>Variable: data</u>
def(1) → use(2)
def(1) → use(4)

<u>Variable: total</u>
def(2) → use(8)
def(2) → use(10)

<u>Variable: column</u>
def(1) → use(6)

def(1) → use(6)

def(8) → use(8)
def(8) → use(10)

Variable: rowCount
def(4) → use(5)

Variable: r
def(5) → use(5) (loop condition & increment)
def(5) → use(6)

Variable: n
def(6) → use(7)
def(6) → use(8)

Test case 2: 4/13 pairs

Variable: data
def(1) → use(2)
def(1) → use(4)
def(1) → use(6)

Variable: total
def(2) → use(8)
def(2) → use(10)
def(8) → use(8)
def(8) → use(10)

Variable: column
def(1) → use(6)

Variable: rowCount
def(4) → use(5)

Variable: r
def(5) → use(5) (loop condition & increment)
def(5) → use(6)

Variable: n
def(6) → use(7)
def(6) → use(8)

Test Case 3: 12/13 pairs

Variable: data
def(1) → use(2)
def(1) → use(4)
def(1) → use(6)

Variable: total
def(2) → use(8)
def(2) → use(10)
def(8) → use(8)
def(8) → use(10)

Variable: column
def(1) → use(6)

Variable: rowCount
def(4) → use(5)

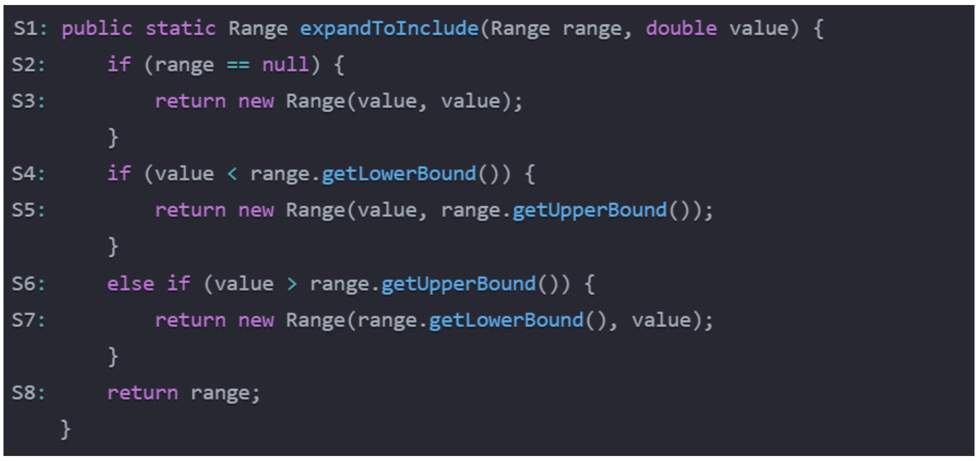
Variable: r
def(5) → use(5) (loop condition & increment)
def(5) → use(6)

Variable: n
def(6) → use(7)
def(6) → use(8)

Between the 3 test cases, we obtain 100% DU-pair coverage overall.

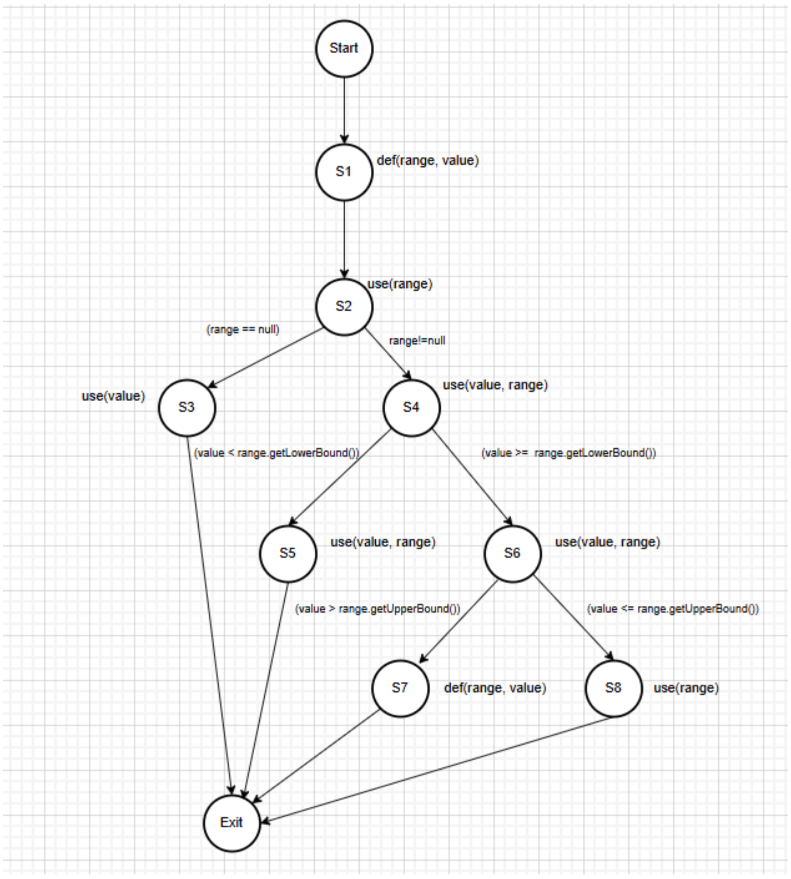
Range.ExpandtoInclude:

Labelled Statements:



```
S1: public static Range expandToInclude(Range range, double value) {
S2:     if (range == null) {
S3:         return new Range(value, value);
        }
S4:     if (value < range.getLowerBound()) {
S5:         return new Range(value, range.getUpperBound());
        }
S6:     else if (value > range.getUpperBound()) {
S7:         return new Range(range.getLowerBound(), value);
        }
S8:     return range;
    }
```

Data Flow Graph:



DU-Pairs per Variable:

Range:

- (S1, S2) - range defined in S1, used in S2
- (S1, S4) - range defined in S1, used in S4 (getLowerBound())
- (S1, S5) - range defined in S1, used in S5 (getUpperBound())
- (S1, S6) - range defined in S1, used in S6 (getUpperBound())
- (S1, S7) - range defined in S1, used in S7 (getLowerBound())

Value:

- (S1, S3) - value defined in S1, used in S3
- (S1, S4) - value defined in S1, used in S4
- (S1, S5) - value defined in S1, used in S5
- (S1, S6) - value defined in S1, used in S6
- (S1, S7) - value defined in S1, used in S7

DU-Pairs by Test Case:

Test Case 1: testExpandToIncludeInsideRange

- Input: range = new Range(5, 15), value = 10
- Execution path: S1 → S2 → S4 → S6 → S8
- DU-pairs covered:
  - For range: (S1, S2), (S1, S4), (S1, S6), (S1, S8)
  - For value: (S1, S4), (S1, S6)

Test Case 2: testExpandToIncludeLower

- Input: range = new Range(5, 15), value = 2
- Execution path: S1 → S2 → S4 → S5
- DU-pairs covered:
  - For range: (S1, S2), (S1, S4), (S1, S5)
  - For value: (S1, S4), (S1, S5)

Test Case 3: testExpandToIncludeUpper

- Input: range = new Range(5, 15), value = 20
- Execution path: S1 → S2 → S4 → S6 → S7
- DU-pairs covered:
  - For range: (S1, S2), (S1, S4), (S1, S6), (S1, S7)
  - For value: (S1, S4), (S1, S6), (S1, S7)

DU-Pairs Coverage Calculation:

Total DU-pairs:

- For range: 6 pairs
- For value: 5 pairs
- Total: 11 pairs

DU-pairs covered by all test cases combined:

- For range: (S1, S2), (S1, S4), (S1, S5), (S1, S6), (S1, S7), (S1, S8) = 6 pairs
- For value: (S1, S3) not covered, (S1, S4), (S1, S5), (S1, S6), (S1, S7) = 4 pairs
- Total covered: 10 pairs

DU-pair coverage = (Number of DU-pairs covered / Total DU-pairs) × 100% DU-pair coverage = (10 / 11) × 100% = 90.91%. The only DU-pair not covered is (S1, S3) for value, which corresponds to the case where range is null.

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

Our approach to improving test coverage for DataUtilities and Range will focus on analyzing conditional branches and ensuring all possible execution paths are tested. First, we will identify all the conditions in each method and determine if they can be reached. If so, we will create a test case to cover that path. For example, in the Range constructor, we will test if a lower bound greater than the upper bound correctly throws an IllegalArgumentException. However, some cases, like testing getLowerBound() when the lower is greater than the upper, are unreachable due to constructor validation and will be noted.

Next, we will work in pairs to write test cases for each reachable branch, covering both true and false conditions to maximize branch and statement coverage. Afterward, we will review method coverage as a group, using EclEmma after each new test to track improvements. The test cases with a significant increase in any type of coverage will be noted down. Any remaining uncovered code will be tested collaboratively to achieve at least 90% statement, 70% branch, and 60% condition coverage. This structured approach ensures thorough and efficient test development. We worked in pairs where first it was Jaisumer and Shahed pair programming, and then by Neha and Jaival.

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

Range Tests:

<u>constrain(double value):</u>

The constrain method was previously untested, meaning its behavior wasn't verified in any existing test cases. By introducing testConstrainUpper, we ensured that when a value exceeds the upper bound, it correctly returns the upper bound instead of the input value. This method includes multiple conditional checks, such as whether the value is greater than the upper bound or less than the lower bound, which weren't being covered before. This test was necessary because untested cases could lead to incorrect handling of NaN inputs, potentially returning an invalid range instead of null.

- Line coverage: 37% → 42%
- Branch coverage: 37.8% → 40.2%
- Method coverage: 47.8% → 52.2%

<u>combineIgnoringNaN(Range range1, Range range2):</u>

The combineIgnoringNaN method had multiple nested conditionals that weren't fully tested, leaving significant portions of its logic unverified. Our test, **testCombineIgnoringNaN_BothNaNRanges**, specifically ensured that when both ranges contain NaN values, the function correctly returns null, as per its intended behavior. This test was necessary because untested cases could lead to incorrect handling of NaN inputs, potentially returning an invalid range instead of null. This was crucial since untested null-handling logic could have led to potential bugs or incorrect calculations in real use cases.

- Line coverage: 49.6% → 58%
- Branch coverage: 48.8% → 56.1%
- Method coverage: 56.5% → 69.6%

The increase in statement and branch coverage indicates that previously uncovered loops and conditional checks were now executed.

DataUtilities Tests:

calculateColumnTotal(Values2D data, int column, int[] validRows):

Before testing calculateColumnTotal, this method had never been explicitly covered, leaving gaps in branch and method coverage. The test **testCalculateColumnTotal3Param_ValidData** verifies that it correctly sums selected rows in a column while handling null values gracefully. By including a mix of positive, negative, and null values, the test ensures that the method properly ignores nulls and sums valid numbers. This was crucial since untested null-handling logic could have led to potential bugs or incorrect calculations in real use cases.

- Line coverage: 60.9% → 77.1%
- Branch coverage: 65.6% → 70.3%
- Method coverage: 80% → 90%

The increase in statement and branch coverage indicates that previously uncovered loops and conditional checks were now executed.

clone(double[][] source):

The clone method was also untested, meaning its ability to properly copy arrays hadn't been verified. The test **testClone_NormalArray** checks that the function creates a deep copy, ensuring that modifying the cloned array does not affect the original. By iterating through both the original and cloned arrays, it confirms that each row is correctly duplicated with matching values. This is crucial since an improper clone could lead to unexpected side effects, such as shared references instead of actual independent copies.

- Line coverage: 57.3% → 65.6%
- Branch coverage: 54.7% → 59.4%
- Method coverage: 70% → 80%

calculateRowTotal(Values2D data, int row, int[] validCols):

Before this test, calculateRowTotal lacked coverage for handling valid column selections and null values, which are key edge cases for data aggregation. The test **testCalculateRowTotal3Param_ValidData** ensures the function correctly sums valid columns in a row, while ignoring null values and handling various numbers (positive, negative, and zero). This was particularly important because null values could potentially cause incorrect summation logic if not properly accounted for.

- Line coverage: 77.1% → 88.5%
- Branch coverage: 71.9% → 81.2%
- Method coverage: 90% → 100%

After adding this test, coverage improved significantly, covering key branches and conditions that were previously skipped.

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

Metrics (After):

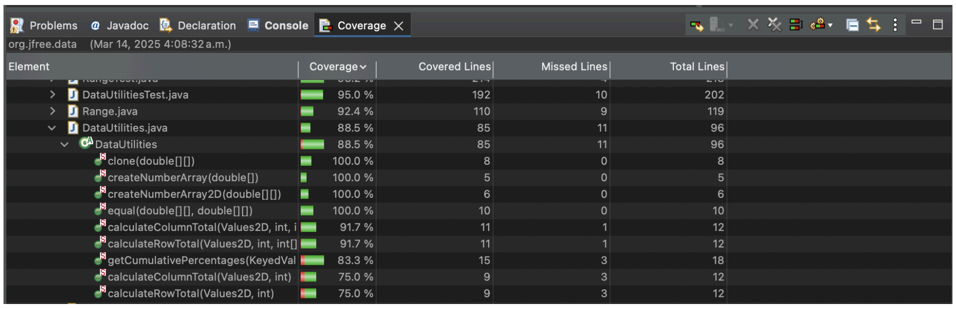
Line Coverage is Statement Coverage and Method Coverage is Condition Coverage

File:	Range.java	DataUtilities.java
Line >= 90%:	92.4%	88.5%
Branch >= 70%:	92.7%	82.8%
Method >= 60%:	100%	100.0%

DataUtilities:

Line:

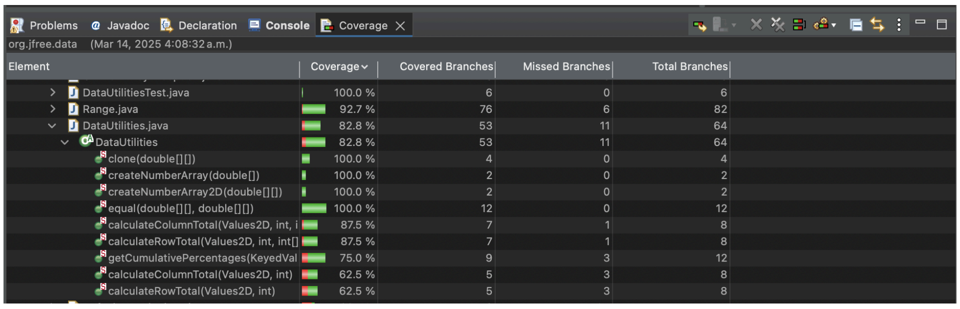
Unfortunately, the line coverage could not go above 90%. However, there is a valid reason. A lot of the code written in DataUtilities is unreachable. For example, there would be a lot of for loops, where the value that is part of a condition, is already disobeying the condition in the for loop statement, because of a predefined value. Let’s say x = 0, the for loop would contain a condition like x > columncount, where column count cannot be physically below 0, so this for loop will never get executed. There were a couple other unnecessary if conditions that could never come out as true, but other than that, this is the most amount of line coverage possible.



org.jfree.data (Mar 14, 2025 4:08:32 a.m.)

Element	Coverage	Covered Lines	Missed Lines	Total Lines
DataUtilitiesTest.java	95.0 %	192	10	202
Range.java	92.4 %	110	9	119
DataUtilities.java	88.5 %	85	11	96
DataUtilities	88.5 %	85	11	96
clone(double[][])	100.0 %	8	0	8
createNumberArray(double[])	100.0 %	5	0	5
createNumberArray2D(double[][])	100.0 %	6	0	6
equal(double[][], double[][])	100.0 %	10	0	10
calculateColumnTotal(Values2D, int, i	91.7 %	11	1	12
calculateRowTotal(Values2D, int, int[]	91.7 %	11	1	12
getCumulativePercentages(KeyedVal	83.3 %	15	3	18
calculateColumnTotal(Values2D, int)	75.0 %	9	3	12
calculateRowTotal(Values2D, int)	75.0 %	9	3	12

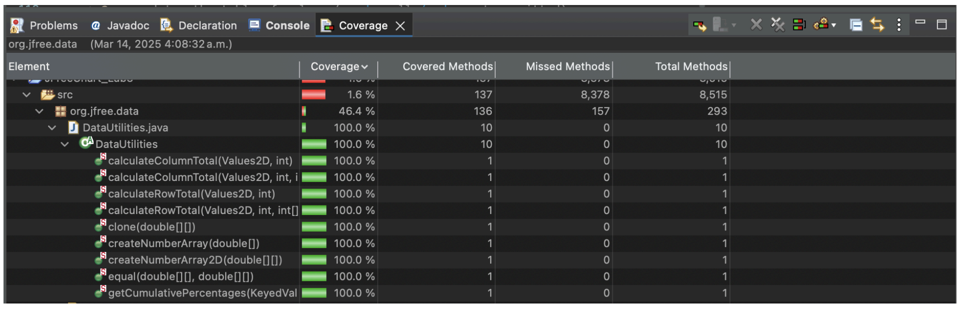
Branch:



org.jfree.data (Mar 14, 2025 4:08:32 a.m.)

Element	Coverage	Covered Branches	Missed Branches	Total Branches
DataUtilitiesTest.java	100.0 %	6	0	6
Range.java	92.7 %	76	6	82
DataUtilities.java	82.8 %	53	11	64
DataUtilities	82.8 %	53	11	64
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
calculateColumnTotal(Values2D, int, i	87.5 %	7	1	8
calculateRowTotal(Values2D, int, int[]	87.5 %	7	1	8
getCumulativePercentages(KeyedVal	75.0 %	9	3	12
calculateColumnTotal(Values2D, int)	62.5 %	5	3	8
calculateRowTotal(Values2D, int)	62.5 %	5	3	8

Method:

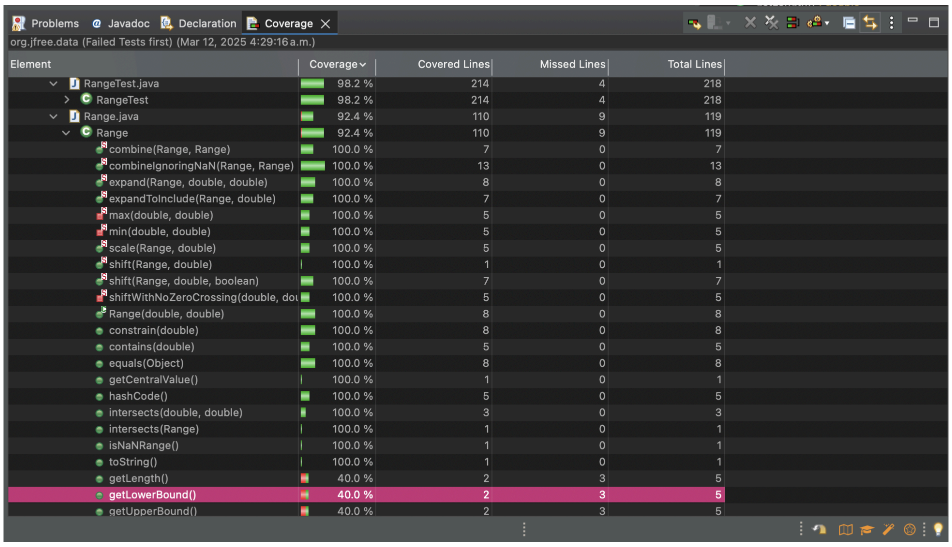


org.jfree.data (Mar 14, 2025 4:08:32 a.m.)

Element	Coverage	Covered Methods	Missed Methods	Total Methods
src	1.6 %	137	8,378	8,515
org.jfree.data	46.4 %	136	157	293
DataUtilities.java	100.0 %	10	0	10
DataUtilities	100.0 %	10	0	10
calculateColumnTotal(Values2D, int)	100.0 %	1	0	1
calculateColumnTotal(Values2D, int, i	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(KeyedVal	100.0 %	1	0	1

Range:

Line:

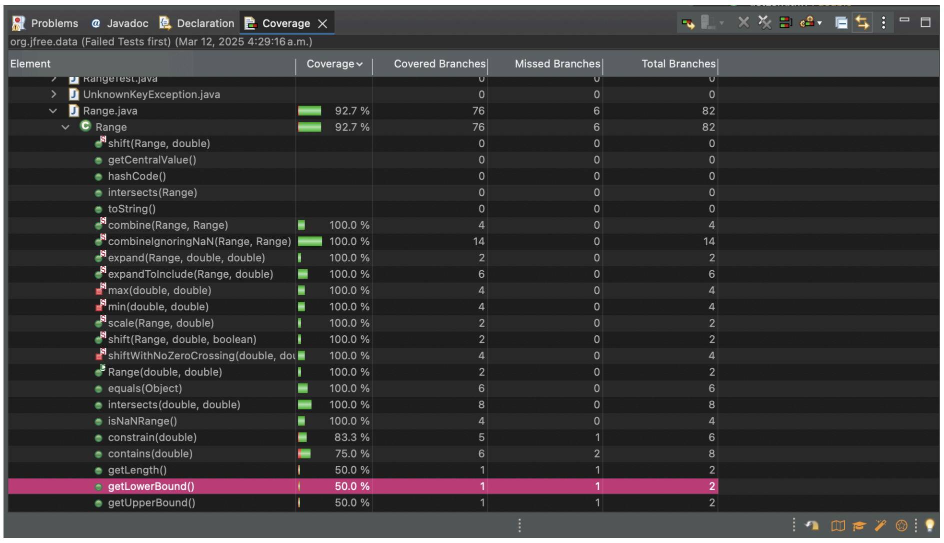


Problems | Javadoc | Declaration | Coverage

org.jfree.data (Failed Tests first) (Mar 12, 2025 4:29:16 a.m.)

Element	Coverage	Covered Lines	Missed Lines	Total Lines
RangeTest.java	98.2 %	214	4	218
RangeTest	98.2 %	214	4	218
Range.java	92.4 %	110	9	119
Range	92.4 %	110	9	119
combine(Range, Range)	100.0 %	7	0	7
combineIgnoringNaN(Range, Range)	100.0 %	13	0	13
expand(Range, double, double)	100.0 %	8	0	8
expandToInclude(Range, double)	100.0 %	7	0	7
max(double, double)	100.0 %	5	0	5
min(double, double)	100.0 %	5	0	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, dou	100.0 %	5	0	5
Range(double, double)	100.0 %	8	0	8
constrain(double)	100.0 %	8	0	8
contains(double)	100.0 %	5	0	5
equals(Object)	100.0 %	8	0	8
getCentralValue()	100.0 %	1	0	1
hashCode()	100.0 %	5	0	5
intersects(double, double)	100.0 %	3	0	3
intersects(Range)	100.0 %	1	0	1
isNaNRange()	100.0 %	1	0	1
toString()	100.0 %	1	0	1
getLength()	40.0 %	2	3	5
getLowerBound()	40.0 %	2	3	5
getUpperBound()	40.0 %	2	3	5

Branch:

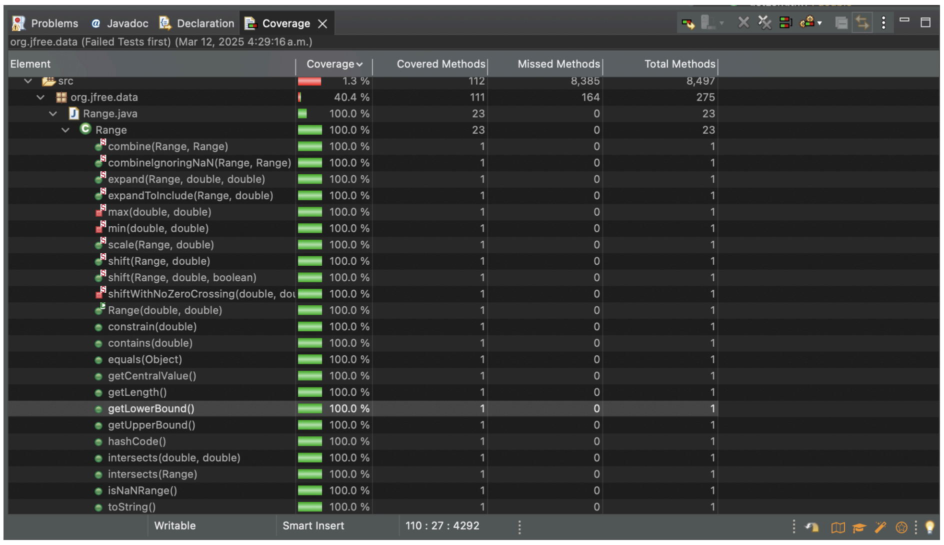


Problems | Javadoc | Declaration | Coverage

org.jfree.data (Failed Tests first) (Mar 12, 2025 4:29:16 a.m.)

Element	Coverage	Covered Branches	Missed Branches	Total Branches
RangeTest.java		0	0	0
UnknownKeyException.java		0	0	0
Range.java	92.7 %	76	6	82
Range	92.7 %	76	6	82
shift(Range, double)		0	0	0
getCentralValue()		0	0	0
hashCode()		0	0	0
intersects(Range)		0	0	0
toString()		0	0	0
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, dou	100.0 %	4	0	4
Range(double, double)	100.0 %	2	0	2
equals(Object)	100.0 %	6	0	6
intersects(double, double)	100.0 %	8	0	8
isNaNRange()	100.0 %	4	0	4
constrain(double)	83.3 %	5	1	6
contains(double)	75.0 %	6	2	8
getLength()	50.0 %	1	1	2
getLowerBound()	50.0 %	1	1	2
getUpperBound()	50.0 %	1	1	2

Method:



Problems | Javadoc | Declaration | Coverage

org.jfree.data (Failed Tests first) (Mar 12, 2025 4:29:16 a.m.)

Element	Coverage	Covered Methods	Missed Methods	Total Methods
src	1.3 %	112	8,385	8,497
org.jfree.data	40.4 %	111	164	275
Range.java	100.0 %	23	0	23
Range	100.0 %	23	0	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, dou	100.0 %	1	0	1
Range(double, double)	100.0 %	1	0	1
constrain(double)	100.0 %	1	0	1
contains(double)	100.0 %	1	0	1
equals(Object)	100.0 %	1	0	1
getCentralValue()	100.0 %	1	0	1
getLength()	100.0 %	1	0	1
getLowerBound()	100.0 %	1	0	1
getUpperBound()	100.0 %	1	0	1
hashCode()	100.0 %	1	0	1
intersects(double, double)	100.0 %	1	0	1
intersects(Range)	100.0 %	1	0	1
isNaNRange()	100.0 %	1	0	1
toString()	100.0 %	1	0	1

Writable | Smart Insert | 110 : 27 : 4292

Pros and Cons of coverage tools used and Metrics you report

Using EclEmma for code coverage in our JUnit tests for Range and DataUtilities was very straightforward and informative. It provided coverage for instruction, line, branch, and method, allowing us to see exactly which parts of the code were tested. While the assignment asked for statement, branch, and condition coverage, EclEmma displayed line instead of statement coverage and method instead of condition coverage, which essentially measured the same thing but under different names. The color-coded visualization (green for fully covered, yellow for partially covered, and red for uncovered) made it easy to track which parts of the code needed more tests.

One challenge was understanding how it covered the test code itself, especially when dealing with expected exceptions and mocks. If a test was designed to trigger an IllegalArgumentException, the exception would be tested but the line that threw it wouldn't be marked as covered. Similarly, context-checking statements for mocks in DataUtilitiesTest weren't counted as covered, even though they were executed. Another con, it counts lines very literally. Let's say the string in a line is too long so I continue it in the next line, EclEmma will count that as two lines, and not as a single instruction. Despite this, EclEmma was still a powerful tool that gave us a clear breakdown of coverage and helped us refine our test cases effectively.

Metrics (Before):

Line Coverage is Statement Coverage and Method Coverage is Condition Coverage

File:	Range.java	DataUtilities.java
Line:	25.2%	46.9%
Branch:	18.3%	32.8%
Method:	30.4%	60.0%

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

Requirements-based testing makes sure the software does what it's supposed to by following the given requirements. It checks if all expected features work correctly and helps catch missing or unclear requirements early on. Since the tests come from real user needs, they are easy to understand and explain. This method is great for making sure the software does what people actually need it to do. However, just because something meets requirements doesn't mean all the code is tested. Some parts of the program might never run, which means hidden bugs could still exist. If the requirements are unclear or incomplete, the tests will be too. Plus, this type of testing doesn't measure how much of the actual code is covered, so some bugs could slip through.

Coverage-based testing makes sure as much of the code as possible runs during testing. It helps find untested parts of the program and catches hidden issues. Since it gives clear coverage numbers (like how many lines or branches are tested), it's easy to see what's missing. It's also useful for making sure new changes don't break existing code. But high coverage doesn't mean the software actually works as expected. A test can run a piece of code without checking if the result is correct. This method can also lead to unnecessary tests just to boost coverage numbers. Plus, it won't catch missing features—it only tests what's already written. It is also a lot more time consuming incase the code is much longer.

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

Our team divided the work strategically to maximize efficiency. Jaisumer and Shahed pair-programmed to develop DataUtilitiesTest.java, while Neha and Jaival worked together on RangeTest.java. Since focusing on branch coverage often leads to achieving statement (line) coverage and some method coverage, our pairs focused on ensuring all branches were tested first.
Once the branch tests were in place, we revisited the tests to improve statement (line) coverage and condition (method) coverage. The entire team worked together to find any uncovered lines using EclEmma, which made it easy to spot missing or partially covered code. While doing the tests, we also looked for which tests had the most significant impact on code coverage and we noted those down to report later. For manual data flow coverage, Neha and Shahed worked on the DataUtilities method and a method of their choice from Range. Lastly, Shahed, Jaisumer, and Jaival collaborated on writing the lab report.

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

One of the biggest challenges was handling dependencies, especially org.hamcrest.collection, which was referenced from Assignment 2. The instructions only mentioned looking at the previous lab for dependencies, which caused confusion as we assumed everything needed was included in the new assignment's setup. This led to wasted time troubleshooting errors that could have been avoided with clearer instructions. Once we figured out the missing dependencies and properly configured the build path, things went smoothly, and the tests executed as expected.

The lab was tedious at times, but ultimately, it was a valuable learning experience. We saw firsthand how test coverage tools like EclEmma help visualize what gets executed and what doesn't, making it easier to identify gaps in our test suite. This process highlighted the importance of writing detailed and thorough tests, as even minor untested lines could lead to missed bugs or unexpected behavior. By the end of the lab, we all gained a better appreciation for structured testing and how every small detail in test coverage matters for building reliable software.

Comments/feedback on the lab itself

A major improvement for future assignments would be to bundle all required dependencies in a single ZIP file within the new lab's repository. Having to go back to a different lab section to retrieve dependencies was confusing, especially since we assumed everything in the new assignment was self-contained and up to date. This small change would make the setup process much smoother and save students unnecessary troubleshooting time. Apart from that, it was a very helpful lab.