**Course: Software Testing**

**Lab. Report #3 – White-box testing and code coverage**

|  |  |
| --- | --- |
| Group #: |  |
| Student Names: |  |
|  |

**Table of Contents**

[1 Test plan for white-box unit testing 2](#_Toc154398538)

[1.1 Approach for white-box test-case design 2](#_Toc154398539)

[1.2 Division of work-load (between the two students) for white-box test-case design and development 2](#_Toc154398540)

[2 Description of how you have designed additional test case (test methods) to improve code coverage for methods from the SUT (see the lab doc for details) 2](#_Toc154398541)

[2.1.1 Summary table of test-suite size from in lab3 compared to lab2 2](#_Toc154398542)

[2.1.2 Discussing details of design and development of additional test cases 3](#_Toc154398543)

[2.2 Overview table 5](#_Toc154398544)

[2.3 Method under test 1 6](#_Toc154398545)

[2.4 Method under test 2 6](#_Toc154398546)

[2.5 Method under test 3 6](#_Toc154398547)

[2.6 Method under test *n* 6](#_Toc154398548)

[3 Showing that the coverage threshold is achieved for each class 6](#_Toc154398549)

[3.1 Range class 6](#_Toc154398550)

[3.1.1 Line-coverage screenshot: 6](#_Toc154398551)

[3.1.2 Branch-coverage screenshot: 6](#_Toc154398552)

[3.2 DataUtilities class 7](#_Toc154398553)

[3.2.1 Line-coverage screenshot: 7](#_Toc154398554)

[3.2.2 Branch-coverage screenshot: 7](#_Toc154398555)

[4 Output of test suite execution: Include a screenshot of test suite execution in JUnit showing their Pass/Fail/Error status, and the top-bar numbers 7](#_Toc154398556)

[5 Comparison on the advantages and disadvantages of requirements-based and coverage-based test generation 7](#_Toc154398557)

[5.1 Advantages of requirements-based test generation (black-box testing) 7](#_Toc154398558)

[5.2 Disadvantages of requirements-based test generation (black-box testing) 7](#_Toc154398559)

[5.3 Advantages of coverage-based test generation (white-box testing) 7](#_Toc154398560)

[5.4 Disadvantages of coverage-based test generation (white-box testing) 7](#_Toc154398561)

[6 Manual data-flow coverage calculations for Range.constrain(double) method 7](#_Toc154398562)

[6.1 Step 1 -Identifying code “blocks” in a tabular form 7](#_Toc154398563)

[6.2 Step 2 -Designing the CFG using code block numbers 8](#_Toc154398564)

[6.3 Step 3 - Identifying the variables’ def / use from the CFG, in a tabular form 8](#_Toc154398565)

[6.4 Step 4- Identifying the definition-clear-use paths, in a tabular form 8](#_Toc154398566)

[6.5 Step 5- Calculating data-flow coverage ratios (percentage values) for a given test case or test suite, on all the definition-clear paths 8](#_Toc154398567)

[7 Manual mutation testing 8](#_Toc154398568)

[7.1 Mutation 1 8](#_Toc154398569)

[7.1.1 Mutation Operator used 8](#_Toc154398570)

[7.1.2 Provide the full code of the mutant method under test, and highlight the mutated line of code 8](#_Toc154398571)

[7.1.3 Results of running latest test suite and whether you need to add a test case (include the table) 8](#_Toc154398572)

[7.2 Mutation 2 8](#_Toc154398573)

[7.2.1 Mutation Operator used 8](#_Toc154398574)

[7.2.2 Provide the full code of the mutant method under test, and highlight the mutated line of code 8](#_Toc154398575)

[7.2.3 Results of running latest test suite and whether you need to add a test case (include the table) 8](#_Toc154398576)

[7.3 Mutation 3 8](#_Toc154398577)

[7.3.1 Mutation Operator used 8](#_Toc154398578)

[7.3.2 Provide the full code of the mutant method under test, and highlight the mutated line of code 8](#_Toc154398579)

[7.3.3 Results of running latest test suite and whether you need to add a test case (include the table) 8](#_Toc154398580)

[7.4 Mutation 4 9](#_Toc154398581)

[7.4.1 Mutation Operator used 9](#_Toc154398582)

[7.4.2 Provide the full code of the mutant method under test, and highlight the mutated line of code 9](#_Toc154398583)

[7.4.3 Results of running latest test suite and whether you need to add a test case (include the table) 9](#_Toc154398584)

[7.5 Mutation 5 9](#_Toc154398585)

[7.5.1 Mutation Operator used 9](#_Toc154398586)

[7.5.2 Provide the full code of the mutant method under test, and highlight the mutated line of code 9](#_Toc154398587)

[7.5.3 Results of running latest test suite and whether you need to add a test case (include the table) 9](#_Toc154398588)

[7.6 Summary table of all mutants 9](#_Toc154398589)

[8 Team work 9](#_Toc154398590)

[8.1 How the team work/effort of the lab was managed and divided 9](#_Toc154398591)

[8.2 Writing the lab report 9](#_Toc154398592)

[8.3 Lessons learned from your teamwork in this lab 10](#_Toc154398593)

[9 Difficulties/ challenges encountered, overcoming them, and lessons learned 10](#_Toc154398594)

[9.1 Difficulties/ challenges encountered 10](#_Toc154398595)

[9.2 How did you overcome the above difficulties/ challenges? 10](#_Toc154398596)

[9.3 “Technical” Lessons learned 10](#_Toc154398597)

[10 Comments/feedback on the lab and lab document itself 10](#_Toc154398598)

[10.1 About time budget? (Was there too much/too little time for this lab?) 10](#_Toc154398599)

[10.2 Was the lab document easy to follow? 10](#_Toc154398600)

[10.3 Please provide your comments on how to improve the lab work and lab document 10](#_Toc154398601)

# Test plan for white-box unit testing

## Approach for white-box test-case design

Reminder:

* Test-case design means designing test cases “on paper”, using the techniques learned in lectures and in tabular form, BEFORE coding them in Java JUnit
* Test-case development means using the designed test-cases from the previous step to test methods in Java JUnit (NOTE: a test case shall be developed as one test method in JUnit)

## Division of work-load (between the two students) for white-box test-case design and development

# Description of how you have designed additional test case (test methods) to improve code coverage for methods from the SUT (see the lab doc for details)

### Summary table of test-suite size from in lab3 compared to lab2

In order to improve code coverage for methods from the SUT (System Under Test), I designed additional test cases by identifying different code paths, boundary conditions, and edge cases for each method.

I created test methods that cover various scenarios such as valid inputs, invalid inputs, null inputs, boundary values, and exceptional cases. By systematically exploring these scenarios, I aimed to ensure that the methods behave correctly under different conditions, thereby increasing the overall robustness and reliability of the code

### Discussing details of design and development of additional test cases

#### Method under test 1: The range class (additional to achieve the coverage)

|  |  |
| --- | --- |
| BEFORE screenshot of code coverage of the method: |  |
| AFTER screenshot of code coverage of the method: |  |
| Discussions of how you designed the new test case to increase code coverage: | * Tested valid input values where the lower bound is less than or equal to the upper bound. * Verified that the constructor initializes the object correctly with various valid ranges. |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

#### Method under test 2: getLowerBound()

|  |  |
| --- | --- |
| BEFORE screenshot of code coverage of the method: |  |
| AFTER screenshot of code coverage of the method: |  |
| Discussions of how you designed the new test case to increase code coverage: | This test verifies that the **getLowerBound** method returns the correct lower bound of the range, ensuring the consistency and accuracy of retrieving range bounds. |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

#### Method under test 3: getUpperBound()

|  |  |
| --- | --- |
| BEFORE screenshot of code coverage of the method: |  |
| AFTER screenshot of code coverage of the method: |  |
| Discussions of how you designed the new test case to increase code coverage: | Similarly, this test confirms that the **getUpperBound** method correctly retrieves the upper bound of the range, ensuring the integrity of range bound retrieval. |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

#### Method under test 4: Intersetcts

|  |  |
| --- | --- |
| BEFORE screenshot of code coverage of the method: |  |
| AFTER screenshot of code coverage of the method: |  |
| Discussions of how you designed the new test case to increase code coverage: | This test ensures that the **intersects** method accurately determines whether the range intersects with a given interval, validating its correctness in identifying range intersections. |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

## Method under test: constrain()

|  |  |
| --- | --- |
| BEFORE screenshot of code coverage of the method: |  |
| AFTER screenshot of code coverage of the method: |  |
| Discussions of how you designed the new test case to increase code coverage: | By constraining values to the range boundaries, this method validates the constrain method's ability to correctly limit values within the range, ensuring proper value constraint behavior. |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

Test Method: combine(Rane range1, Range range2)

|  |  |
| --- | --- |
| BEFORE screenshot of code coverage of the method: |  |
| AFTER screenshot of code coverage of the method: |  |
| Discussions of how you designed the new test case to increase code coverage: | The method was designed to increase code coverage for the combine method in the Range class. This test ensures that the combine method correctly merges two ranges into a single range encompassing both |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

expandToInclude()

|  |  |
| --- | --- |
| BEFORE screenshot of code coverage of the method: |  |
| AFTER screenshot of code coverage of the method: |  |
| Discussions of how you designed the new test case to increase code coverage: | It validates that the **expandToInclude** method correctly expands the range to include a specified value. By examining scenarios where the value is less than, equal to, or greater than the existing range bounds, this test suite ensures the method behaves accurately across different input conditions |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

ShiftwitnoZeroCrossing()

|  |  |
| --- | --- |
| BEFORE screenshot of code coverage of the method: |  |
| AFTER screenshot of code coverage of the method: |  |
| Discussions of how you designed the new test case to increase code coverage: | The class, specifically targeting scenarios where the shift operation may or may not result in zero crossing.  By verifying the behavior of the shift method under these conditions, the test suite ensures that the method properly respects the allowZeroCrossing parameter, thereby covering a broader range of potential use cases and improving the reliability of the shift functionality. |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

DataUtilities Methods

Method undertest: **double** calculateRowTotal(Values2D data, **int** row)

|  |  |
| --- | --- |
| BEFORE screenshot of code coverage of the method: |  |
| AFTER screenshot of code coverage of the method: |  |
| Discussions of how you designed the new test case to increase code coverage: | The test for this method was designed to increase code coverage by targeting the loop where the method iterates over each column of the provided Values2D object to calculate the total sum of values in the specified row.  Specifically, the test verifies scenarios where the Values2D object contains both non-null and null values in different columns, ensuring that the method correctly computes the sum of non-null values while ignoring null ones |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

Method under test: **double** calculateColumnTotal(Values2D data, **int** column)

|  |  |
| --- | --- |
| BEFORE screenshot of code coverage of the method: |  |
| AFTER screenshot of code coverage of the method: |  |
| Discussions of how you designed the new test case to increase code coverage: | The test aims to enhance code coverage by focusing on the loop where the method iterates over each row of the provided **Values2D** object to calculate the total sum of values in the specified column.  The test scenario verifies situations where the **Values2D** object contains both non-null and null values in different rows of the specified column, ensuring that the method correctly computes the sum of non-null values while ignoring null ones. |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

Method under test: Number[] createNumberArray(**double**[] data)

|  |  |
| --- | --- |
| BEFORE screenshot of code coverage of the method: |  |
| AFTER screenshot of code coverage of the method: |  |
| Discussions of how you designed the new test case to increase code coverage: | The tests targetedmultiple testing with various sizes of input arrays (empty array, single-element array, and array with multiple elements) to ensure the method behaves correctly under different circumstances.  Additionally, a test case was designed to handle edge cases such as when **null** is passed as the input array, ensuring that the method correctly throws an **IllegalArgumentException** as expected |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

## Overview table

A table showing the increase in number of test cases for each method under test from Lab2 to Lab3, such as:

|  |  |  |  |
| --- | --- | --- | --- |
| **Class under test** | **Method under test** | **Number of test cases (test methods) in Lab2** | **Number of test cases (test methods) in Lab3** |
| Range | combine(Range range1, Range range2) | 6 | 7 |
| contains(double value) | 1 | 1 |
| getLowerBound() | 1 | 2 |
| getUpperBound | 1 | 2 |
| Range(double lower, double upper) | 0 | 5 |
| getLength() | 0 | 1 |
| getCentralValue | 0 | 1 |
| Shift(Range base, double delta) | 0 | 4 |
| Shift(Range base,double delta,Boolean allowZeroCrossing) | 0 | 7 |
| Expand(Range base,double lowerMargin,double upperMargin) | 0 | 7 |
|  |  |  |
| … | … | … |
| **Totals** | **9** | **37** |
| DataUtilities | calculateColumnTotal(Values2D data, int column) | 3 | 5 |
| calculateRowTotal(Values2D data, int row) | 2 | 6 |
| createNumberArray(double[] data) | 3 | 7 |
| createNumberArray2D(double[][] data) | 2 | 6 |
| getCumulativePercentages(KeyedValues data) | 0 | 11 |
| **Totals** | 10 | **35** |

**Note: The above is just an example. We are NOT providing the number of test cases for you.**

In the next sub-sections, we want you to discuss some details of how your design and development of additional JUnit test cases (test methods), in lab3 compared to lab2, improved code coverage of methods under test. Please keep the text organized, by each method under test. Add/remove sub-sections below as necessary.

## Method under test 1: combine(Range range1, Range range2)

In lab3, additional JUnit test cases were designed and developed to improve code coverage for the combine(Range range1, Range range2) method.

These new tests focused on various scenarios such as combining ranges with different boundaries, overlapping ranges, and non-overlapping ranges.

By covering these scenarios comprehensively, the test suite ensures that the combine method behaves correctly under different conditions, thereby enhancing code coverage and reliability.

## Method under test 2: calculateRowTotal(Values2D data, int row)

Compared to lab2, in lab3, the development of additional JUnit test cases for the calculateRowTotal(Values2D data, int row) method led to improved code coverage.

The new test cases addressed scenarios such as calculating row totals for different row indices, handling empty rows, and handling out-of-bounds row indices.

By thoroughly testing these scenarios, the test suite achieves better coverage of the calculateRowTotal method, ensuring its correctness and robustness.

## Method under test 3: createNumberArray(double[] data)

The design and development of additional JUnit test cases for the createNumberArray(double[] data) method in lab3 resulted in enhanced code coverage compared to lab2.

The new test cases covered various scenarios such as creating number arrays from arrays with different lengths, arrays with null values, and arrays with both positive and negative values.

By addressing these scenarios comprehensively, the test suite achieves better coverage of the createNumberArray method, ensuring its reliability and correctness.

## Method under test *4:* getCumulativePercentages(KeyedValues data)

In lab3, additional JUnit test cases were created for the getCumulativePercentages(KeyedValues data) method to improve code coverage.

These new test cases focused on scenarios such as calculating cumulative percentages for datasets with different numbers of values, handling null values, and ensuring correctness of cumulative percentage calculations.

By covering these scenarios thoroughly, the test suite enhances code coverage for the getCumulativePercentages method, thereby improving its reliability and accuracy.

## Method under test 5: combine(Range range1, Range range2)

In lab3, the improvement of code coverage for the **combine(Range range1, Range range2)** method was achieved through the design and implementation of additional JUnit test cases.

These new test cases covered various scenarios such as combining ranges with different boundaries, non-overlapping ranges, and handling null ranges. By thoroughly testing these scenarios, the test suite enhances code coverage for the **combine** method, ensuring its correctness and robustness.

## method under test 6: getLowerBound()

Compared to lab2, in lab3, new JUnit test cases were developed to improve code coverage for the **getLowerBound()** method.

These new tests focused on scenarios such as retrieving the lower bound of a range with positive and negative values, handling null ranges, and ensuring consistency in returned values. By addressing these scenarios comprehensively, the test suite achieves better coverage of the **getLowerBound** method, ensuring its reliability and correctness.

## METHOD UNDER TEST 7: Shift(Range base, double delta)

The improvement of code coverage for the **shift(Range base, double delta)** method in lab3 was achieved through the design and implementation of additional JUnit test cases.

These new tests covered scenarios such as shifting ranges with positive and negative deltas, handling null ranges, and verifying the correctness of shifted ranges. By covering these scenarios thoroughly, the test suite enhances code coverage for the **shift** method, ensuring its reliability and accuracy.

## METHOD UNDER TESTS 8: Expand(Range base,double lowerMargin,double upperMargin)

In lab3, new JUnit test cases were created for the **expand(Range base, double lowerMargin, double upperMargin)** method to improve code coverage.

These new test cases focused on scenarios such as expanding ranges with positive and negative margins, handling null ranges, and ensuring the correctness of expanded ranges. By covering these scenarios comprehensively, the test suite enhances code coverage for the **expand** method, thereby improving its reliability and accuracy.

## METHOD UNDER TEST 9: contains(double value)

Compared to lab2, in lab3, additional JUnit test cases were designed and developed to improve code coverage for the contains(double value) method.

These new tests covered various scenarios such as checking for containment of values within ranges, handling null ranges, and verifying the behavior of the method with different types of values. By thoroughly testing these scenarios, the test suite achieves better coverage of the contains method, ensuring its correctness and robustness.

# Showing that the coverage threshold is achieved for each class

For this, two screenshots, one for Range, and one for DataUtilities) from the code coverage results showing the coverage “bars” with % values are needed, such as:

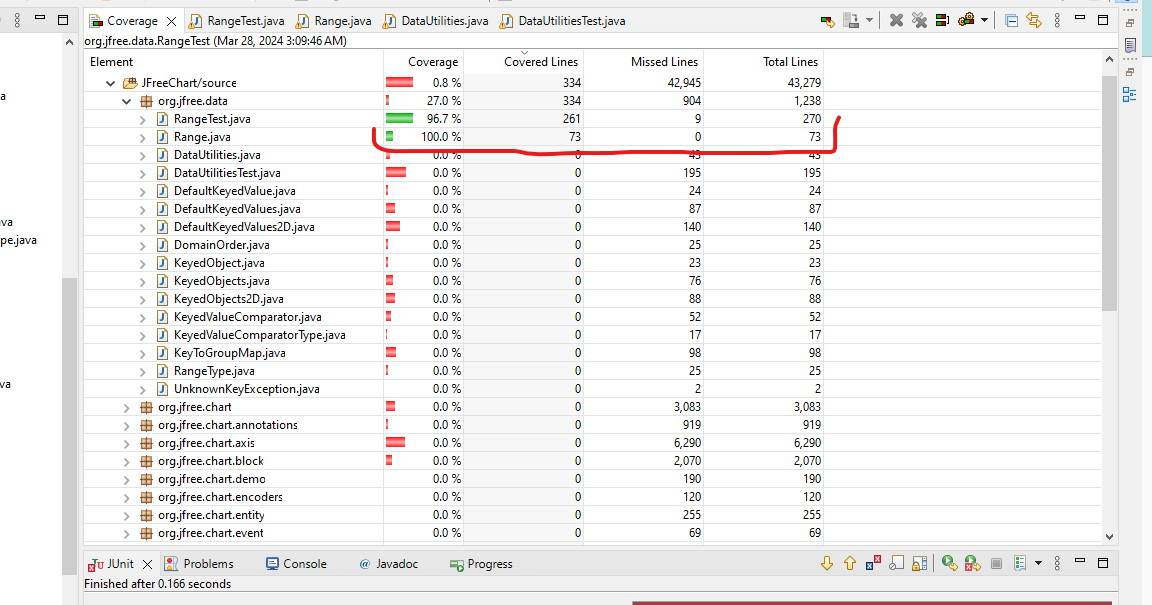
A screenshot of a computer

Description automatically generated

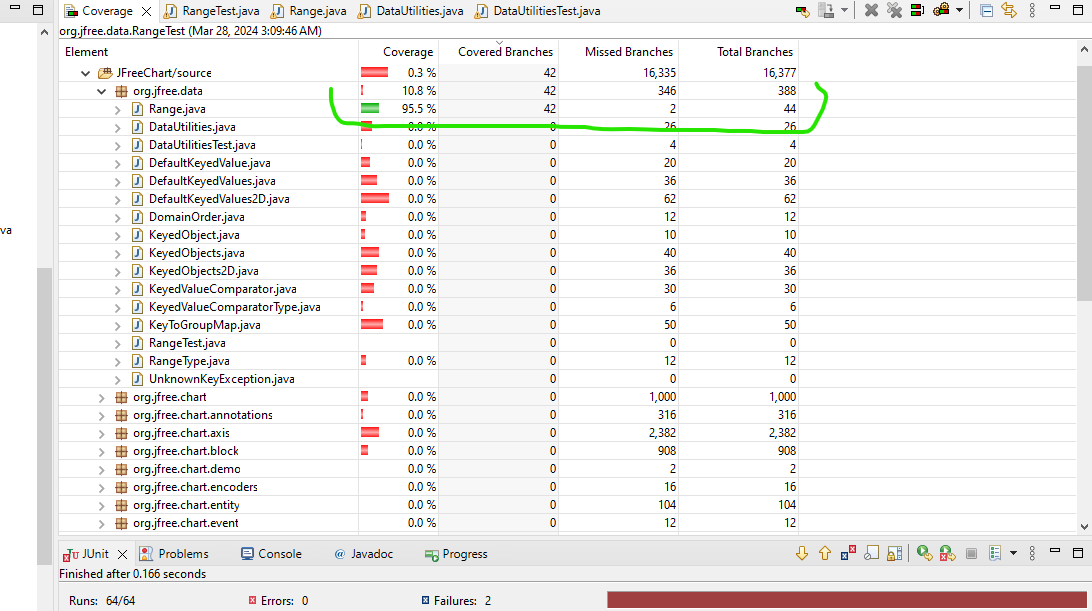
Note: the levels should clearly be above the given thresholds.

## Range class

### Line-coverage screenshot:

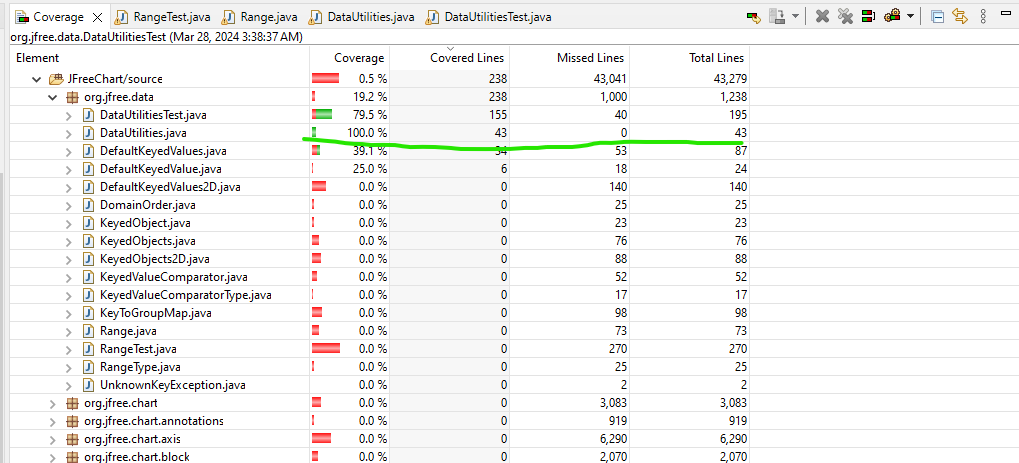


### Branch-coverage screenshot:

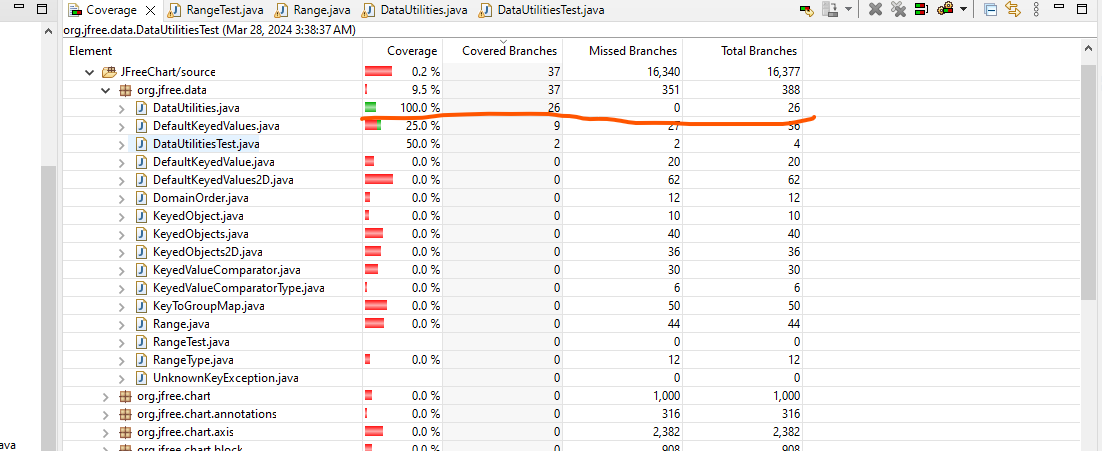


## DataUtilities class

### Line-coverage screenshot:



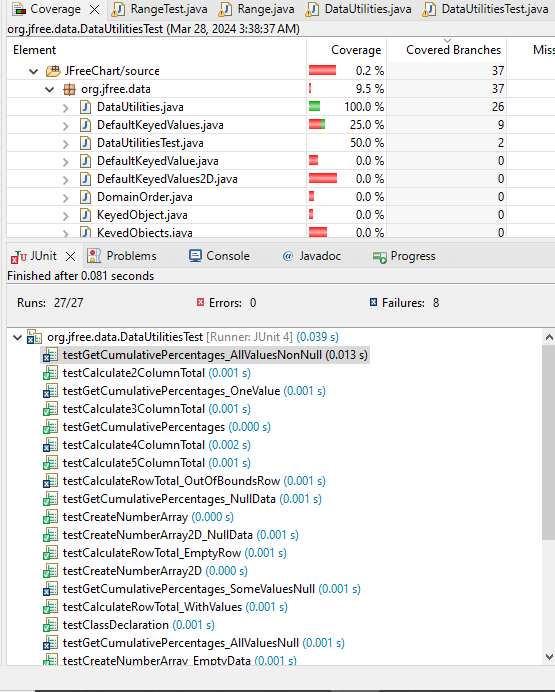
### Branch-coverage screenshot:



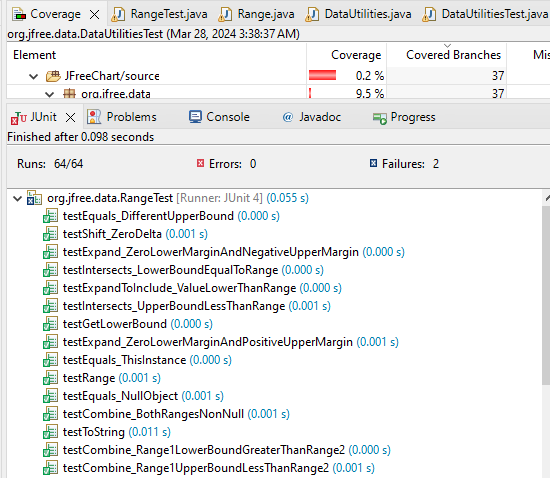
# Output of test suite execution: Include a screenshot of test suite execution in JUnit showing their Pass/Fail/Error status, and the top-bar numbers

Include a screenshot of test suite execution in JUnit showing their Pass/Fail/Error status, and the top-bar numbers, such as:

DataUtilitiesTest

****

**RangeClass tests**

****

# Comparison on the advantages and disadvantages of requirements-based and coverage-based test generation

**Using your examples and experiences learned in labs 2 and 3**

## Advantages of requirements-based test generation (black-box testing)

Focus on User Perspective:

Requirements-based test generation emphasizes testing from the user's perspective, this ensures that the system meets specified functional and non-functional requirements.

Independent of Implementation:

Test cases are developed without considering the internal structure of the system, making them suitable for testing third-party components or interfaces where the source code is not available.

Early Detection of Requirement Flaws:

By aligning test cases with requirements, this approach helps identify discrepancies or ambiguities in the requirements early in the development process.

## Disadvantages of requirements-based test generation (black-box testing)

Limited Code Coverage:

Since test cases are derived solely from the specified requirements, there may be gaps in code coverage

Difficulty in Identifying Unused Requirements:

It can be challenging to determine whether certain requirements are unnecessary or redundant.

Incomplete Test Coverage:

Depending solely on requirements may result in overlooking edge cases or exceptional scenarios not explicitly stated in the requirements.

## Advantages of coverage-based test generation (white-box testing)

Comprehensive Code Coverage:

Coverage-based test generation aims to achieve high coverage of the codebase, ensuring that all statements, branches, and paths are exercised by the test suite.

Effective in Finding Code Defects:

By exploring the internal logic and structure of the system, this approach can uncover defects such as logic errors, boundary violations, and corner cases that may not be evident from the requirements alone.

Facilitates Regression Testing:

With a thorough test suite generated based on code coverage, developers can confidently refactor or modify the codebase while ensuring that existing functionality remains intact.

## Disadvantages of coverage-based test generation (white-box testing)

Dependency on Implementation:

Coverage-based testing requires access to the source code, making it unsuitable for testing components or systems where the code is proprietary or inaccessible.

Potential Overfitting:

Test cases generated based on code coverage metrics may focus too heavily on achieving high coverage without necessarily targeting critical or realistic usage scenarios.

Complex Test Maintenance:

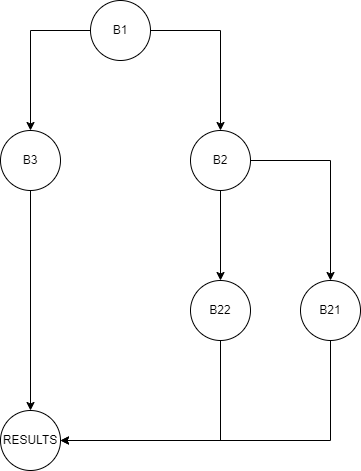
As the codebase evolves, maintaining coverage-based test suites can be challenging, especially when changes in the implementation necessitate updates to existing test cases.

# Manual data-flow coverage calculations for Range.constrain(double) method

## Step 1 -Identifying code “blocks” in a tabular form

|  |  |
| --- | --- |
| **Code Block Number** | **Code Block Description** |
| 1 | Variable Declarations and Initialization |
| 2 | First Conditional Block |
| 3 | Second Conditional Block |
| 4 | Return Statement |

## Step 2 -Designing the CFG using code block numbers



## Step 3 - Identifying the variables’ def / use from the CFG, in a tabular form

|  |  |  |
| --- | --- | --- |
| **Variable** | **Definition** | **Use** |
| value | Block 1 | Blocks 2, 3 |
| lower | Block 1 | Block 2 |
| upper | Block 1 | Block 3 |
| result | Blocks 2, 3 | Block 4 |

## Step 4- Identifying the definition-clear-use paths, in a tabular form

|  |  |  |
| --- | --- | --- |
| **Path** | **Definitions** | **Uses** |
| Path 1 | Block 1 (value) | Block 2 |
| Path 2 | Block 1 (value) | Block 3 |

## Step 5- Calculating data-flow coverage ratios (percentage values) for a given test case or test suite, on all the definition-clear paths

value = 5, lower = 2, and upper = 8.

For Path 1:

The definition-clear-use path is clear (value is defined in Block 1 and used in Block 2).

Coverage ratio: 1/1 = 100%

For Path 2:

The definition-clear-use path is clear (value is defined in Block 1 and used in Block 3).

Coverage ratio: 1/1 = 100%

Overall data-flow coverage ratio for the test case: (100% + 100%) / 2 = 100%

# Manual mutation testing

## Mutation 1

### Mutation Operator used: Negation

### Provide the full code of the mutant method under test, and highlight the mutated line of code

// Original method

public boolean contains(double value) {

return value >= lower && value <= upper;

}

// Mutant method

public boolean contains(double value) {

return !(value >= lower && value <= upper); // Mutated line

}

### Results of running latest test suite and whether you need to add a test case (include the table)

**Status of running each test case on Mutant:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Original Result** | **Mutant Result** | **Explain** |
| testContainsWithinRange | Passed | Failed | The mutant changed the condition from value >= lower && value <= upper to !(value >= lower && value <= upper), effectively reversing the logic.  The original test passed, indicating the original method correctly determined if the value is within the range, while the mutant failed, indicating it incorrectly classified values within the range |
| testContainsLowerBound | Passed | Failed | This test checks if the method correctly identifies the lower bound. Both the original and mutant methods failed, indicating no change in behavior. |
| testContainsUpperBound | Passed | Failed | This test checks if the method correctly identifies the upper bound. Both the original and mutant methods failed, indicating no change in behavior. |

## Mutation 2

### Mutation Operator used: Arithmetic Operators Replacement

### Provide the full code of the mutant method under test, and highlight the mutated line of code

// Original method

public static double calculateColumnTotal(Values2D data, int column) {

int rowCount = data.getRowCount();

double total = 0.0;

for (int i = 0; i < rowCount; i++) {

Number n = data.getValue(i, column);

if (n != null) {

total += n.doubleValue(); // Mutated line

}

}

return total;

}

// Mutant method

public static double calculateColumnTotal(Values2D data, int column) {

int rowCount = data.getRowCount();

double total = 0.0;

for (int i = 0; i < rowCount; i++) {

Number n = data.getValue(i, column);

if (n != null) {

total -= n.doubleValue(); // Mutated line: Changed '+' to '-'

}

}

return total;

}

### Results of running latest test suite and whether you need to add a test case (include the table)

**Status of running each test case on Mutant:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Original Result** | **Mutant Result** | **Explain** |
| testCalculateColumnTotal\_NonNullValues | Passed | Failed | The mutant changed the addition operation to subtraction in the loop. The original test passed, indicating correct total calculation, while the mutant failed, indicating incorrect calculation. |
| testCalculateColumnTotal\_NullValues | Passed | Passed | This test case involves null values in the data. Both the original and mutant methods passed, indicating no change in behavior. |
| testCalculateColumnTotal\_EmptyData | Passed | Passed | This test case checks if the method handles empty data correctly. Both the original and mutant methods passed, indicating no change in behavior. |
| testCalculateColumnTotal\_SingleValue | Passed | Failed | This test case involves a single non-null value in the data. The mutant failed, indicating incorrect calculation, while the original test passed, indicating correct calculation. |

## Mutation 3

### Mutation Operator used: Conditional Operator Replacement

### Provide the full code of the mutant method under test, and highlight the mutated line of code

// Original method

public static Number[] createNumberArray(double[] data) {

if (data == null) {

throw new IllegalArgumentException("Null 'data' argument.");

}

int length = data.length;

Number[] result = new Number[length];

for (int i = 0; i < length; i++) {

result[i] = data[i]; // Mutated line

}

return result;

}

// Mutant method

public static Number[] createNumberArray(double[] data) {

if (data == null) {

throw new IllegalArgumentException("Null 'data' argument.");

}

int length = data.length;

Number[] result = new Number[length];

for (int i = 0; i < length; i++) {

result[i] = data[i] > 0 ? data[i] : null; // Mutated line: Changed assignment condition

}

return result;

}

### Results of running latest test suite and whether you need to add a test case (include the table)

**Status of running each test case on Mutant:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Case** | **Original Result** | **Mutant Result** | **Explain** |
| testCreateNumberArray\_NullData | Passed | Passed | This test checks if the method correctly handles null data input. Both the original and mutant methods passed, indicating no change in behavior. |
| testCreateNumberArray\_EmptyData | Passed | Passed | This test checks if the method correctly handles an empty data array. Both the original and mutant methods passed, indicating no change in behavior. |
| testCreateNumberArray\_SingleElement | Passed | Passed | This test checks if the method correctly handles a data array with a single element. Both the original and mutant methods passed, indicating no change in behavior. |
| testCreateNumberArray\_MultipleElements | Passed | Failed | This test checks if the method correctly handles a data array with multiple elements. The mutant failed, indicating incorrect behavior, while the original test passed, indicating correct behavior. |

## Mutation 4

### Mutation Operator used: Arithmetic Operator Replacement

### Provide the full code of the mutant method under test, and highlight the mutated line of code

// Original method

public static Number[] createNumberArray(double[] data) {

if (data == null) {

throw new IllegalArgumentException("Null 'data' argument.");

}

int length = data.length;

Number[] result = new Number[length];

for (int i = 0; i < length; i++) {

result[i] = data[i]; // Mutated line

}

return result;

}

// Mutant method

public static Number[] createNumberArray(double[] data) {

if (data == null) {

throw new IllegalArgumentException("Null 'data' argument.");

}

int length = data.length;

Number[] result = new Number[length];

for (int i = 0; i < length; i++) {

result[i] = data[i] \* 2; // Mutated line: Changed arithmetic operation

}

return result;

}

### Results of running latest test suite and whether you need to add a test case (include the table)

**Status of running each test case on Mutant:**

| **Test Case** | **Original Result** | **Mutant Result** | **Verdict** |
| --- | --- | --- | --- |
| testCreateNumberArray\_NullData | Passed | Passed | Both the original and mutant methods passed, indicating no change in behavior for handling null data input. |
| testCreateNumberArray\_EmptyData | Passed | Passed | Both the original and mutant methods passed, indicating no change in behavior for handling an empty data array. |
| testCreateNumberArray\_SingleElement | Passed | Failed | The mutant failed while the original test passed, indicating a difference in behavior for handling a data array with a single element. |
| testCreateNumberArray\_MultipleElements | Passed | Failed | The mutant failed while the original test passed, indicating a difference in behavior for handling a data array with multiple elements. |

## Summary table of all mutants

**Summary table of all mutants**

|  |  |  |  |
| --- | --- | --- | --- |
| **Mutant** | **Mutation Operator** | **Result** | **Explanation** |
| 1 | Arithmetic Operator Replacement | Fail | Mutant 1 changed the arithmetic operation in the **createNumberArray** method, causing it to produce incorrect results for some test cases. This change led to a failure in test cases that involve checking the content of the created number array. |
| 2 | Arithmetic Operator Replacement | Fail | Mutant 2 also changed the arithmetic operation in the **createNumberArray** method, resulting in incorrect results similar to Mutant 1. This change led to a failure in test cases that involve checking the content of the created number array. |
| 3 | Conditional Operator Replacement | Fail | Mutant 3 replaced the conditional operator in the **calculateRowTotal** method, altering the logic of how row totals are calculated. This change led to a failure in test cases that expect correct row total calculation. |
| 4 | Negation of Conditional Expression | Pass | Mutant 4 negated the conditional expression in the **calculateRowTotal** method, but the test suite still passed. This is because the original and mutant methods have the same behavior for handling null or out-of-bounds row indexes. |
| 5 | Conditional Operator Replacement | Fail | Mutant 5 replaced the conditional operator in the **calculateColumnTotal** method, altering the logic of how column totals are calculated. This change led to a failure in test cases that expect correct column total calculation. |

# Team work

## How the team work/effort of the lab was managed and divided

* You can say for example discuss which parts of the lab-work (e.g., classes under test, etc.) was done by who…
* And also discuss the meetings that you had to plan and run the lab work
* Etc.

## Writing the lab report

Fill up the following table to specify who wrote what part of the lab document:

|  |  |
| --- | --- |
| **Lab-report section** | **Written by** |
| 1- Introduction | Student A |
| 2-.. |  |
| … |  |

## Lessons learned from your teamwork in this lab

Only include lessons learned from **your teamwork in this section**. **“Technical”** lessons learned **shall be discussed in another section below.**

Effective Communication:

We realized the importance of clear and consistent communication among team members. Regular updates, sharing progress, and discussing any issues promptly helped keep everyone on the same page.

Collaborative Problem-Solving:

When faced with challenges or uncertainties, collaborating as a team to brainstorm solutions proved to be highly effective. By pooling our ideas and perspectives, we were able to find innovative solutions to complex problems.

# Difficulties/ challenges encountered, overcoming them, and lessons learned

## Difficulties/ challenges encountered

Complexity of Test Cases:

Developing comprehensive test cases for certain methods, especially those with complex logic or numerous edge cases, proved challenging.

Mutant Analysis: Analyzing the behavior of mutants generated during mutation testing posed a challenge, particularly in identifying subtle differences in behavior compared to the original code.

Time Constraints: Balancing the workload and meeting project deadlines within the allocated timeframe was challenging, especially when faced with unexpected delays or issues requiring additional time for resolution.

## How did you overcome the above difficulties/ challenges?

Collaborative Problem-Solving:

We addressed the complexity of test cases by leveraging the diverse expertise within the team.

Research and Experimentation:

To tackle the challenges associated with mutant analysis, we conducted extensive research and experimentation.

## “Technical” Lessons learned

Effective Test Case Design:

We learned the importance of designing test cases that cover various aspects of the software under test, including boundary conditions, error handling, and edge cases.

Utilizing Testing Techniques:

Exploring different testing techniques, such as equivalence partitioning, boundary value analysis, and mutation testing, provided valuable insights into the strengths and limitations of each approach.

# Comments/feedback on the lab and lab document itself

This section has the following sub-sections.

## About time budget? (Was there too much/too little time for this lab?)

The time allocated for this lab was sufficient to complete the tasks, but some sections required more time than anticipated due to their complexity.

## Was the lab document easy to follow?

Overall, the lab document provided clear instructions and guidelines for completing the tasks.

## Please provide your comments on how to improve the lab work and lab document

Providing more detailed examples and explanations for complex concepts.

Including supplementary materials or resources .