**SENG 438 - Software Testing, Reliability, and Quality**

**Lab. Report \#4 – Mutation Testing and Web app testing**

| Group \#: | 20 |
| --- | --- |
| Student Names: | Siddhartha Paudel |
| | Monil Patel |
| | Arsalan Baig |
| | Saim Khalid |

# Introduction

This lab focuses on two essential software testing techniques: Mutation Testing and GUI Testing. In the first part, we use Pitest to introduce faults (mutants) into our code and evaluate the effectiveness of our test suite from previous assignments. We aim to improve mutation scores for the Range and DataUtilities classes.

The second part involves GUI Testing using Selenium IDE to automate user interactions with web applications. We will design and execute test cases for selected websites(Home Deport) and compare Selenium with Sikulix. This lab helps develop practical skills in fault detection and automated UI testing.

# Analysis of 10 Mutants of the Range class

1.) toString()
Mutation: "" was used in place of the return value → SURVIVED
Evaluation: Since there were no test cases in the original test suite that used the toString() method, this modification was able to persist. The modification was not detectable without calling toString(). The creation of a range between 2 and 6 and the assertion that its string representation equals "Range[2.0,6.0]" are the contents of a newly added test case. In this test case, the mutation was killed since the assertion failed when the return value was changed to an empty string.

2.) contains(double value)
Mutation: Conditional border was altered, resulting in survival.
Evaluation: Because there were no instances in the test suite where the value being tested equaled the lower or higher bound of the range, this mutation was able to survive. To determine whether a value equivalent to a

Range(2,6)'s lower bound (2) and upper bound (6) returns true, more test cases were added. By claiming boundary inclusivity, these cases killed the mutation and confirmed the proper boundary logic.

3.) intersects(double b0, double b1)
Mutation: Conditional border was altered, resulting in survival.
Evaluation: Because the exact situation where b0 equals the top bound of the range or b1 equals the lower bound was not covered by the test cases, this mutation initially survived. To address these edge cases, two new tests were introduced: one against Range(2,6) where b0 = 6 and another where b1 = 2. These instances effectively eliminated the mutation and validated that boundary logic operated as intended.

4.)intersects (Range range)
Mutation: True → SURVIVED was used in place of the boolean return.
Evaluation: A legal false return was changed to true by the mutation. This survived because there was no test in the original suite that required two non-overlapping ranges to intersect (Range). A test with the base range being Range (2,6) and the tested range being Range (7,9) was added. The procedure should return false because these don't intersect. The test failed because the mutated method returned true, killing the mutation.

5.)shiftWithNoZeroCrossing(double value, double delta)
Mutation: Replaced double addition with subtraction → SURVIVED
Evaluation: Cases in which the value parameter is negative were not covered by the initial test suite. To make sure the method enters the conditional block intended for negative values, a test case with values = -1 and delta = 1 was added. The mutation was eliminated because it changed value + delta to value - delta, producing an inaccurate result and failing the test.

6.)intersects(double b0, double b1)
Mutation: Negative conditional → KILLED The border condition was changed from <= to < by this alteration. The altered logic yielded inaccurate results since the tests already contained edge situations where b0 = lowerBound and b1 = upperBound. As anticipated, these tests were unsuccessful, killing the mutation.

7.)expandToInclude(Range range, double value)

Mutation: The reason this mutation was killed was because the test cases expandToIncludeNullLower and expandToIncludeNullUpper expected a valid non-null Range to be provided when extending a null range with a given value. The mutation changed the return value to null. The mutation was effectively killed by returning null, which resulted in a NullPointerException or failed assertions.

8.)expand(Range range, double lowerMargin, double upperMargin)
Mutation: Division was used in place of double multiplication, resulting in KILLED Analysis: The flaw caused by this modification was made visible by the test case expandLowerGreaterThanUpper_Lower(). Because the mutated technique divided rather than multiplied, it calculated the bottom bound improperly, producing a different range. This terminated the mutation and made the test fail.

9.)shift(Range base, double delta)
Analysis: Mutation: Incremented (++delta) → KILLED Incorrect range shifting resulted from the mutation's alteration of the delta value. For example, the mutation created an unanticipated range by moving by -1.2 instead of -2.2. The shiftByNegativeDoubleLowerBound test case detected this disparity and eliminated the mutation.

10.)scale(Range base, double factor)
Mutation: Double local variable number 1 was negated, resulting in KILLED Evaluation: The scaling factor was changed from its initial value (for example, 2) to 1 by this alteration. Expecting an upper bound of 12, the test case scalePositiveRangePositiveFactor applied a factor of 2 on a range of (0,6). The mutation was eliminated when the test failed because the mutated procedure gave an inaccurate result of 6.

**# Report all the statistics and the mutation score for each test class**

RangeTest Class:

testCentralValueOfPositiveRange()
The purpose of this test was to validate the Range class's getCentralValue() method. We constructed a Range object with values between 4 and 8, and we used getCentralValue() to find the middle value. Using assertEquals, we verified that the expected result was 6. Since this approach had never been tried before, the new test was able to eliminate a

number of mutants linked to inaccurate central value computations, which greatly increased the mutation score.

testContainsValueAtLowerBound()
This test examines the behavior of the contains(double value) method when the input value is precisely equal to the lower constraint. We checked to see if it included two using a range of two to six. assertTrue should be used to confirm that the result is true. Since the lower boundary requirement had not been discussed, this test was especially crucial because it assisted in the death of several mutants that had made improper border comparisons.

testContainsValueAtUpperBound()
Like the last test, this one determines whether the higher border value is appropriately identified as being inside the range by the contains method. We tested with the value 6 and a range of 2, 6. Mutants that handled upper bound inclusivity erroneously were eliminated, and the test validates that the method returns true.

testIntersectsRangeShouldBeFalse_AboveBaseline()
When the given range is completely above the base range, this test determines whether the intersects(double b0, double b1) method appropriately returns false. We tried with (7, 10) and utilized a base range of (2, 6). Several mutants that had previously returned true, even for non-intersecting ranges, were destroyed by the test.

testIntersectsRangeBoundaryAtLowerBoundShouldBeFalse()
This test case was created to identify edge-case problems in intersects (double b0, double b1) where the lower bound of the base range meets the upper bound of the given range. We tested using (1, 2) and claimed that the result should be false for a Range (2, 6). Mutants where the lower bound comparison reasoning was faulty were eliminated by this test.

testIntersectsRangeBoundaryAtUpperBoundShouldBeFalse()
Here, we tested the intersections approach once more, but this time, the tested range's lower bound was equal to the base range's higher bound. Range(2, 6) and testing with (6, 7) were used to confirm that false is returned correctly. It assisted in eliminating mutations resulting from incorrect upper boundary logic processing.

testIntersectsRangeInsideRange()
This test was created to make sure that when the given range is entirely contained within the base range, the intersects(double b0, double b1) method returns true. In order to verify that intersection is accurately recognized, we tried using a base of (2, 3). When internal range overlap logic was handled incorrectly, the test assisted in the death of mutants.

testIntersectsRangeShouldBeFalse_InvalidBoundOrder()
This test determines whether the method returns false correctly and gives an invalid range where b0 > b1 (for example, 5 to 4). For robustness, these kinds of edge cases are essential, and this test eliminated mutants that were unable to correctly validate the order of limits.

testExpandRangeWhereLowerExceedsUpper()
We tested the expand() function in which the new lower bound is bigger than the new upper bound due to negative margins. The test assisted in eliminating mutations linked to incorrect computations or a lack of validation in such uncommon circumstances, and this scenario is heavy on the edge.

testScaleRangeWithZeroFactor()
This test was included to examine how the scale (double factor, range base) behaves when the scaling factor is zero. A lower and upper bound of 0 should result from scaling a range of (2, 6) by 0. Mutants that failed because of divide-by-zero problems or produced erroneous boundaries were eliminated from the test.

testScaleRangeWithPositiveFactorCheckLowerBound()
When a positive factor is applied, this test case guarantees that the scale() method will behave correctly. A lower bound of 4 should be obtained by scaling a (2, 6) range by 2. Mutants related to arithmetic that handled multiplication or scaling logic incorrectly were eliminated by this test.

testShiftWithNoZeroCrossingAtZeroValue()
This test was designed to verify the outcome of using shift() with allowZeroCrossing = false to shift a range that begins at zero. (158, 158) should be the result of shifting a range of (0, 0) by 158. This assisted in eliminating mutants that disregarded the zero-crossing criterion or improperly applied the shift.

testShiftNegativeRangeByLargeNegative()
By shifting a negative range (-2, -1) by a significant negative delta -158 with allowZeroCrossing = false, this test investigated the shift() technique. It verified that the relocation was carried out correctly and in accordance with zero-crossing regulations. When handling negative ranges and substantial shifts, the test assisted in eliminating logical mistakes.

testRangeEqualsNullShouldBeFalse()
This test determines whether comparing a Range object with null returns false, as it should, in order to guarantee that the equals() method operates correctly. Mutants that handled nulls were killed by the test.

testRangeEqualsDifferentLowerBoundShouldBeFalse()
This test verifies that two Range objects are not regarded as equal if their lower limits differ. When (2, 6) and (3, 6) are compared, the result should be false. Mutants that disregarded lower bound differences in the equality check were eliminated by this test.

testRangeEqualsDifferentUpperBoundShouldBeFalse()
This test confirms that ranges with differing upper limits are not equal, just like the one before it. When comparing (2, 6) to (2, 5), the results should be false. Mutants associated with improper upper bound comparison logic were eliminated by this test.

testCombineIgnoringNaN_BothHaveNaN()
When NaN is present in both input ranges, this test case verifies the combineIgnoringNaN() method. This test assisted in identifying mutants that were unable to correctly filter or validate NaN values, which is the expected outcome.

testCombineIgnoringNaN_BothValidNoNaN()
This test verified that a new Range(2, 7) is successfully produced when two valid ranges (2, 6) and (4, 7) are combined. Mutants that mismerged bounds or improperly compared the two ranges were killed.

testToStringTest()
This test verifies that the expected string representation of a range is returned by the toString() method. It should return "Range[2.0,6.0]" for a

Range(2, 6). This test eliminated a number of string-related mutations because this technique had not been discussed before.

DataUtilities Test:

testCreateNumberArray()
The purpose of this test was to verify that the DataUtilities class's createNumberArray(double[]) method worked as intended. It creates an array of Number objects from a raw array of doubles. Due to the lack of coverage in our first test suite, a number of modifications pertaining to array formation and element conversion were able to persist. This test improved total mutation coverage by successfully killing mutants involving wrong conversion and inappropriate indexing by passing an array {1.1, 2.2, 3.3} and claiming that the length and values match the intended outcome.

testCreateNumberArrayWithNull()
The purpose of this test was to precisely target the mutation that modifies the createNumberArray() method's conditional if (data == null). The mutant was able to survive since passing a null input was not verified in our first test suite. This test makes that the defensive check is operating as planned by anticipating an IllegalArgumentException when null is provided. The mutant that eliminated or altered the null-check condition was essentially killed.

testGetCumulativePercentages()
The purpose of this test is to confirm that the getCumulativePercentages(KeyedValues data) method operates as intended. Since cumulative percentage computations were not covered in our first suite, a number of mutations linked to arithmetic were able to persist. We examined if the technique yielded percentages of 0.5, 0.8, and 1.0, respectively, using a DefaultKeyedValues object whose values for "A," "B," and "C" added up to 10. Mutants relating to improper cumulative value computation, wrong index iteration, and improper summation logic were effectively eliminated by these claims.

testGetCumulativePercentagesWithNull()
We included this test to make sure that getCumulativePercentages(null) returns an IllegalArgumentException in order to evaluate robustness and eliminate mutations associated with incorrect null handling. Because this branch was not tested in our first test suite, the mutation that got

around the null check was able to persist. We eliminated the mutant and ensured proper exception handling by calling the procedure directly with null.

**# Analysis drawn on the effectiveness of each of the test classes**

Range Test (NEW MUTATION COVERAGE):

# Pit Test Coverage Report

## Project Summary

| Number of Classes | Line Coverage | Mutation Coverage | Test Strength |
|---|---|---|---|
| 1 | 91% 94/103 | 72% 901/1259 | 80% 901/1133 |

## Breakdown by Package

| Name | Number of Classes | Line Coverage | Mutation Coverage | Test Strength |
|---|---|---|---|---|
| org.jfree.data | 1 | 91% 94/103 | 72% 901/1259 | 80% 901/1133 |

Report generated by PIT 1.6.8

We saw an improvement of more than 10% when comparing the mutation scores from our first test suite (created in Assignment 3) to the updated scores with the addition of new test cases, exceeding the lab manual's minimal increase requirement. Our improved test suite shows a respectable degree of reliability in software validation with a final mutation score of 72%. There is still opportunity for improvement, though, as 232 mutations are still unidentified (i.e., passed the tests), suggesting that there may be gaps in the test coverage or the existence of similar mutants.

DataUtilities (NEW MUTATION COVERAGE):

# Pit Test Coverage Report

## Project Summary

| Number of Classes | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|
| 1 | 98% | 78/80 | 88% | 602/687 | 89% | 602/673 |

## Breakdown by Package

| Name | Number of Classes | Line Coverage | | Mutation Coverage | | Test Strength | |
|---|---|---|---|---|---|---|---|
| org.jfree.data | 1 | 98% | 78/80 | 88% | 602/687 | 89% | 602/673 |

Report generated by PIT 1.6.8

We saw that the remaining mutants were getting harder to find and get rid of as we kept improving our test suite. We saw an improvement of more than 10% when we compared the mutation scores from our original test set (given in Assignment 3 via GitHub) to the updated scores following the addition of our new test cases. This rise shows that our test coverage and fault detection capabilities have significantly improved. Our test suite's overall strength and dependability are confirmed by an appropriate mutation score, which indicates that it is effective in detecting software flaws. Nevertheless, the fact that 71 mutants survived shows where more advancements could be achieved.

**# A discussion on the effect of equivalent mutants on mutation score accuracy**

Equivalent mutants are mutated versions of the code that, despite changes, behave exactly like the original program. These mutants cannot be killed by any test case because their functionality and output remain the same as the original code. As a result, they significantly impact the accuracy of mutation scores.

For mutation score the presence of equivalent mutants lowers the perceived effectiveness of the test suite. Mutation testing tools, such as Pitest, treat these mutants as surviving, therefore falsely indicating that the test cases are inadequate. This leads to an artificially reduced mutation score, suggesting that the test suite is less effective than it truly is.

The challenge is identifying these equivalent mutants manually, as automated detection is often not feasible. This process requires a thorough code review and domain knowledge to determine whether a mutant is genuinely equivalent or simply unkillable by existing test cases.

The inability to automatically detect equivalent mutants introduces a significant disadvantage to mutation testing. It increases manual effort and can lead to incorrect conclusions about the quality of the test suite. However, if correctly identified, excluding equivalent mutants from the calculation would yield a more accurate representation of the test suite's effectiveness.

In conclusion, equivalent mutants negatively affect mutation score accuracy by falsely lowering it. Identifying and eliminating them is crucial for obtaining a more realistic assessment of the test suite's quality.

**# A discussion of what could have been done to improve the mutation score of the test suites**

Improving the mutation score of the test suites requires spotting weaknesses in the existing tests and enhancing coverage to detect more mutants effectively. One way to do that is by analyzing the surviving mutants to understand why they were not killed. Often, these mutants point to gaps in the test suite, such as untested edge cases or insufficient assertion coverage. By examining the mutation logs and understanding the changes introduced by specific mutation operators (like negation changes or arithmetic operator replacements), we can pinpoint areas where the test suite needs improvement.

To increase the mutation score, it is essential to diversify test cases by including boundary conditions, edge cases, and combinations of inputs that could have been overlooked. Strengthening assertions is also essential, as some surviving mutants might result from weak or incomplete checks. Adding tests that specifically target conditional logic, arithmetic operations, and intermediate variables can make the test suite more robust. Additionally, increasing code coverage by writing new tests for untested methods or branches ensures that mutants do not survive due to untested code segments.

Finally, addressing equivalent mutants is crucial for accurate mutation scoring. Equivalent mutants, which are functionally identical to the original code, unfairly reduce the mutation score because they cannot be killed. Identifying these mutants manually and documenting them helps isolate the genuinely surviving mutants that indicate real flaws. By strategically enhancing test diversity, assertion strength, and code coverage, while also accounting for equivalent mutants, we can significantly improve the mutation score and the overall effectiveness of the test suite.

**# Why do we need mutation testing? Advantages and disadvantages of mutation testing**

Advantages of Mutation Testing:

1. Effectiveness Measurement: Mutation testing provides a quantitative measure of how well a test suite can detect faults, giving a more accurate assessment than simple code coverage metrics.

2. Improved Test Quality: By identifying undetected mutants, developers can detect weaknesses in their tests and improve them, leading to higher software reliability.

3. Enhanced Test Case Design: Mutation testing encourages developers to think about edge cases and potential faults, resulting in more thorough and well-designed test cases.

4. Early Bug Detection: Since mutation testing involves systematically injecting faults, it can reveal latent bugs that may not surface during regular testing.

Disadvantages of Mutation Testing:

1. High Computational Cost: Generating and testing numerous mutants can be time-consuming, especially for large projects, making it impractical in some scenarios.

2. Equivalent Mutants Issue: Some mutants are functionally equivalent to the original code, making it impossible for any test case to

detect them. This can artificially lower the mutation score and requires manual inspection, thus increasing manpower required to test.

3. Complexity of Analysis: Understanding why a particular mutant survived can be challenging, as it may not always clearly indicate a problem with the test suite.

4. Tool Limitations: Mutation testing tools might not support certain programming languages or frameworks adequately, leading to incomplete mutation analysis.

**# Explain your SELENUIM test case design process**

We first browsed the website to determine which important features we wanted to test before starting to create our Selenium test cases. We developed a number of features, including the ability to log in, add products to the cart, conduct successful and unsuccessful searches, and more. We gained a solid grasp of the site's behavior after testing each function with sample inputs.

Armed with this information, we wrote our Selenium test cases and implemented each one individually. We created six test cases in total, some of which had several test scenarios to cover various conditions. Instead of concentrating on merely rerouting the site, we verified the essential features, particularly those that included user input. This enabled us to verify that the website was appropriately accepting input data and redirecting.

**# Explain the use of assertions and checkpoints**

Assertions and checkpoints are fundamental components in software testing used to ensure that a program behaves as expected during execution. Assertions are primarily used in unit tests to verify whether a specific condition holds true. If an assertion fails, it indicates that the output did not match the expected result, and the test case is marked as failed. For instance, in JUnit, an assertion like assertEquals(expectedValue, actualValue); checks whether the actual output matches the expected one.

If they differ, the test fails, highlighting a potential issue in the code. Assertions help catch faults early by verifying the correctness of functions and calculations, therefore making them essential in mutation testing to detect errors effectively.

On the other hand, checkpoints are used in automated GUI testing to verify that the application's visual or functional state matches the expected state at specific moments during test execution. They are particularly useful for confirming the presence of UI elements or the correctness of displayed content. For example, in Selenium, a checkpoint like `assertTrue(driver.findElement(By.id("welcomeMessage")).isDisplayed());` ensures that a login message appears after a successful login operation. Checkpoints are essential for automated validation, as they help detect any discrepancies in the GUI without manual inspection. Both assertions and checkpoints contribute to maintaining software quality by providing automated ways to validate functionality and quickly identifying issues when failures occur.

**# how did you test each functionaity with different test data**

We found that we could change the "Value" field in Selenium once recording had finished while executing our test cases. We were able to test several input circumstances and confirm that the application behaved appropriately thanks to this flexibility. However, we ran across restrictions while changing test inputs that had an impact on the application's functionality. For example, the test would fail if we changed the input to Airdrie after first searching for retailers in Calgary and choosing one in particular. This is due to the test's continued attempt to reroute to a store located in Calgary, which is not accessible via the Airdrie store locator.

To accommodate for various outcomes, we chose in certain circumstances to develop distinct test cases for the same functionality. The login mechanism is a nice example; we made two test cases: one for the right username and another for the wrong one. Having separate test cases enabled us to precisely capture and validate both behaviors because each scenario had different outcomes.

**# Discuss advantages and disadvantages of Selenium vs. Sikulix**

In order to track user inputs such as clicks, scrolls, and cursor movements, Selenium uses locators to communicate with the HTML structure of an online application. It is quite useful for web pages with intricate navigation or frequent redirections because these interactions are captured based on the DOM elements. Sikulix, on the other hand, uses image recognition to recognize and interact with user interface elements. It is especially helpful for interfaces with a lot of distinctive graphical features because it can take screenshots of buttons, icons, and other visual components.

The automation feature of both programs enables users to record test cases and replay them without the need for human interaction. But every tool has limitations. For example, Sikulix might have problems if the image that was captured looks like more than one element on the screen, which could result in incorrect interactions when the image is played back. Although Selenium is accurate in identifying HTML components, it can also be brittle; tests may fail due to unanticipated user interface conditions. One real-world instance we ran across was during login testing; if we didn't specifically log out at the end of each test, the session would remain alive and subsequent tests would fail.

Furthermore, Sikulix is more adaptable and can test a wider variety of GUI apps because of its image-based methodology, while Selenium is restricted to web-based applications.

**# How the team work/effort was divided and managed**

Our team divided the work based on each member's strengths and areas of expertise. For Part 1 (Mutation Testing), one group member focused on installing Pitest and setting up the project environment, while others worked on integrating our previous test suites (RangeTest.java and DataUtilitiesTest.java) and running mutation tests. We collaborated to analyze the mutation reports and identify surviving mutants. In Part 2 (GUI Testing), each member selected a specific website and designed test cases independently using Selenium IDE. We then came together to compare Selenium with Sikulix and discuss our findings. Regular meetings and effective communication ensured that everyone tried contributed evenly and stayed updated on progress.

**# Difficulties encountered, challenges overcome, and lessons learned**

One of the primary challenges was dealing with equivalent mutants during mutation testing. It was time-consuming to manually analyze mutation logs and verify whether the surviving mutants were genuinely undetectable or functionally equivalent. We addressed this by dividing the mutants among the team and discussing findings collectively, which sped up the process. This taught us the importance of teamwork and that even the most mundane of tasks can be accomplished if everyone works as a team.

# Comments/feedback on the lab itself

The lab was great, offering practical experience with both mutation testing and GUI testing. The instructions regarding equivalent mutants were somewhat vague, making it difficult to approach this part systematically. Overall the lab greatly enhanced our collective knowledge on software testing and improved our teamwork skills.