

Lab. Report #4

Mutation Testing (Fault Injection) and Web App Test Automation

SENG 438 - Software Testing, Reliability, and Quality

Group #:	18
Student Names:	Ashley Brown
	Nathan Godard
	Rebecca Reid
	Ines Rosito

TABLE OF CONTENTS

1. Introduction	2
2. Analysis of Range Mutants	3
Mutant 1	3
Mutant 2	3
Mutant 3	3
Mutant 4	3
Mutant 5	4
Mutant 6	4
Mutant 7	4
Mutant 8	4
Mutant 9	5
Mutant 10	5
3. Statistics	5
Repairs	5
Mutation Testing Results	5
4. Analysis	6
5. Discussion I	7
5.1 Equivalent Mutations	7
5.1.1 Effect of Equivalent Mutations on Score Accuracy	7
5.1.2 Detection of Equivalent Mutations	7
5.2 Improving Mutation Scores	7
5.3 Mutation Testing: Advantages and Disadvantages	8
5.3.1 Why Do We Need it?	8
5.3.2 Advantages and Disadvantages	8
6. GUI Test Case Design Process	9
6.1 Automated Verification	9
6.2 Test Data	9
Ebay:	9
IKEA:	10
Amazon:	10
7. Selenium versus. SikuliX	10
7.1 Selenium	11
Advantages:	11
Disadvantages:	11
7.2 SikuliX	11
Advantages:	11
Disadvantages:	12
8. Division of Teamwork	12
9. Difficulties Encountered, Challenges Overcome, and Lessons Learned	13
10. Comments and Feedback	13
11. Appendices	15
Appendix A: Initial Pit Test Coverage Report and Surviving Mutants	15

1. Introduction

This report provides discussion and insight to the approach used on two distinct testing topics. The first section addresses the topic of mutation testing. The purpose was to apply the technique to JFreeChart's Range and DataUtilities classes. The SUT included code provided from d2L and tests designed in previous assignments. New tests were developed to satisfy conditions stated in the assignment handout. Under the topic of mutation testing, statistics were gathered and analyzed, which provided necessary insight to design new test cases. The report also includes sections with discussions regarding how to improve mutation scores, the reason to use mutation testing, and the advantages and disadvantages associated with it.

The second half of the report addresses the topic of GUI testing. Testing was completed through the use of a portable software-testing framework for web applications called Selenium. This section provides insight towards the GUI test case design process, and a discussion towards the advantages and disadvantages of different testing tools. It also looks at an alternative GUI testing tool, Sikulix.

The remainder of the report consists of an overview of the teamwork division, difficulties encountered, and general comments and feedback of this assignment. It should be noted that prior to this lab, the group had minimal experience with either mutation testing or GUI testing tools. For each member, it was the first time using Pitest and Selenium.

2. Analysis of Range Mutants

It is assumed

Mutant 1

Location: `shiftWithNoZeroCrossing()`, Line 322

Original: `else if (value < 0.0) {`

Mutant: `else if (value < 1.0) {`

Status: Killed. This test would have been killed with a test case of $(0.0 \leq x \leq 1.0)$. A test case using 1.0 was used. It is tested by `shift_2Inputs_notAllowCrossing_Test()`.

Mutant 2

Location: `expand()`, Line 262

Original: `double upper = length * upperMargin;`

Mutant: `double upper = length / upperMargin;`

Status: Killed. This test would have been killed as the returned value would be different for every `upperMargin` except 1. A number that does not equal 1 is used in tests. For example, `expand_Good_Test()` catches the mutation.

Mutant 3

Location: `constrain()`, Line 172

Original: `if (!contains(value)) {`

Mutant: `if (true) {`

Status: Survived. Because there are further checks inside this if statement, the if statement is unnecessary. The mutated version always returns the same result as the original. If the range contains the value, the value will not be larger than the range's upper bound or smaller than the range's lower bound, so it will not execute the contents of the if statements.

Mutant 4

Location: `constrain()`, Line 173

Original: `if(value > this.upper) {`

Mutant: `if(value >= this.upper) {`

Status: Survived. If `value == this.upper`, then `result = this.upper` (the content of the if statement) is equivalent to `result = result`, so executing the if statement has no effect. The mutated version returns the same result as the original in all cases.

Mutant 5

Location: `getCentralValue()`, Line 127

Original: `return this.lower / 2.0 + this.upper / 2.0;`

Mutant: `return this.lower / 2.0 - this.upper / 2.0;`

Status: Killed. The mutation would be caught with any test where `this.upper` does not equal zero. There is a test in the test suite that fits that requirement, `centralValueTest()`.

Mutant 6

Location: `constrain()`, Line 176

Original: `else if (value < this.lower) {`

Mutant: `else if (true) {`

Status: Survived. This `if/elseif` is within another `if` statement, which is true if the value is not in the range. The only options for a value not in a given range are that its a) lower, or b) higher. Therefore, if the line 173 `if (if (value > this.upper))` is false, then for sure the value is lower than the range. The `else if` could be an `else` and still be correct.

Mutant 7

Location: `expandToInclude()`, Line 233

Original: `else if (value > range.getUpperBound()) {`

Mutant: `else if (value >= range.getUpperBound()) {`

Status: Survived. Although this creates a new range, the created range would be the same as the old range. This is because if `value == range.getUpperBound()`, making a new range with `value` gives the same range as the original range, as they are the same value.

Mutant 8

Location: `intersects()`, Line 152

Original: `if (lower <= this.lower) {`

Mutant: `if (false) {`

Status: Survived. This makes the code assume that the lower of the inputted range always is above the called-range's lower. This leads to the inputted range always intersecting, unless inputted values have `lower > higher`. This was not covered. A test case to cover this would have to have both values larger than called-ranges upper.

Mutant 9

Location: `combine()`, Line 200

Original: `if (range1 == null) {`

Mutant: `if (range1 != null) {`

Status: Killed. Any tests where `range1` is not null, (and is not fully contained within range 2), would catch this, as the expected value expected range 1's numbers to affect the resulting new Range. This is killed by `combine_bothNotNull_Test()`.

Mutant 10

Location: `getLength()`, Line 118

Original: `return this.upper - this.lower;`

Mutant: `return this.upper + this.lower;`

Status: Killed. Most any "normal"/ad-hoc test of this method would catch this mutation. Any test where `this.lower` is NOT zero would catch this. One test that catches it is `getLengthTest()`.

3. Statistics

Repairs

Before mutation testing could be run, several errors in the source code had to be fixed. This included changing which errors were thrown by the functions in `DataUtilities`, as well as fixing the formula in `intersects()`. `Intersects` was changed from "`upper < this.upper`" to "`lower < this.upper`".

Mutation Testing Results

Upon completing the necessary changes to ensure that Pitest could properly run, the following results were calculated as the initial mutation coverage:

Class	Coverage Ratio	Mutation Coverage Percentage
Range	171/204	84%
DataUtilities	95/98	97%

An image of the initial Coverage Report generated by Pitest by running the test suites developed in the previous assignment can be found in [Appendix A](#), as well as, an image of surviving mutants.

Upon completing the necessary changes to ensure that Pitest could properly run, the following results were calculated as the final mutation coverage:

Class	Coverage Ratio	Mutation Coverage Percentage
Range	191/204	94%
DataUtilities	97/98	99%

An image of the Coverage Report generated by Pitest by running the test suites with the new tests may be found in [Appendix B](#), as well as, an image of surviving mutants. It should also be noted that in the assignment document, it is stated that the coverage percentage for each class should be raised by 10%. However, in the case of the DataUtilities due to its initial percentage of 97%, it would be impossible to satisfy the condition. Therefore, it was attempted to raise the coverage as high as possible.

4. Analysis

After resolving defects from the source code and modifying the necessary cases from the previous assignment, the tests on Range and DataUtilities produced a mutation coverage of 84% and 94%, respectively. Line coverage for both these classes were also calculated to be 100%. With these values, it is reasonable to infer that the initial test suites are of high quality, not to state that improvements cannot be made.

Reviewing the surviving mutants, it was possible to identify areas of improvement. All methods in the two classes had been included in the test suite, therefore, issues arose from some conditions being missed. For example, in DataUtilities `getCumulativePercentages()`, the existing test for that method did not include the possibility of a null value for its input. Another example is with the `hashCode()`, which had been tested to ensure the output was a number but not that it was the correct output.

After adding the additional tests in an attempt to increase the mutation score, the final calculated coverage for Range and DataUtilities were 94% and 99%, respectively. These final results imply that the current test suites are highly effective in killing mutants. At the current percentages of these test suites, equivalent mutants are the biggest barrier preventing a the full 100% coverage.

5. Discussion I

5.1 Equivalent Mutations

5.1.1 Effect of Equivalent Mutations on Score Accuracy

Equivalent mutants are equivalent to the original program. This implies that they are functionally identical to the original program, providing the same result under all circumstances, although syntactically they may appear different. Therefore, it is not possible to develop a test to detect or kill them.

Equivalent mutations prevent the reported mutation coverage from ever reaching 100%. This is due to the fact that equivalent mutants increase the number of mutants, but does not change the number of mutants killed in the test suite. These equivalent mutations reduce the accuracy of the score, since a lower score typically implies that a test suite requires improvements.

5.1.2 Detection of Equivalent Mutations

In order to identify equivalent mutants, review of the source code is required. Tests inputs should be reviewed to ensure correct expectations, this involves contrasting the source code to the mutant code and highlighting any anomalies. Testers should essentially review behaviour of their suites from a black-box to white-box perspective, and locate the logic changes and logic outputs. Unfortunately, this approach can prove tedious and time-consuming and therefore, not a favoured approach by some.

Another approach involves the use of mathematical constraints to automatically detect the equivalent mutations. In short, changes in the conditional boundaries should result in some calculation of truth tables, there is an algorithm associated with feasibility paths, to see if the logic output differs. This should be emphasized in cases containing nested if-else statements or a sequence of if statements.

An alternative method, would be to input all possible values into both the original and mutated version of the code, and then comparing results to identify any differences in results. A downside of approach is that it would require massive amounts of computer power, and would be infeasible for most inputs.

5.2 Improving Mutation Scores

The principle approach for which the new tests were designed was based on the analysis of individual mutations and writing specific tests to target them. From the initial coverage report, surviving mutants were reviewed and were classified as either equivalent mutants or not. If they were deemed non-equivalent, a test was written to ensure the test suite would catch them. This approach was feasible due to the low number of mutations that survived the original suite.

In addition, some of the groups of uncaught mutations revealed a structural/systemic problem to the tests. For example, due to the numerous uncaught mutations in the hash method, it was revealed that the tests for that method never verified that the function returned the correct result, only that it returned a result. It was these types of fixes that resulted in the significant increase of the mutation coverage percentage, which was relative to the number of tests added.

5.3 Mutation Testing: Advantages and Disadvantages

5.3.1 Why Do We Need it?

Through the use of other testing techniques, such as white or black-box testing, testers are able to develop quality test suites. However, even with a high percentage of coverage metrics, the tests may not necessarily be considered effective. Mutation testing provides a method in which to measure the robustness of test suites by injecting faults into the source code. These mutations in the original code are tested in order to identify whether or not the existing suite is able to detect these faults.

5.3.2 Advantages and Disadvantages

To be able to identify the advantages and disadvantages, it is important to understand the concept of mutation testing and how it works. Mutation testing differs from the other testing methodologies as it requires the use of the actual code structure to guide its testing process. Mutation testing will rewrite certain components of the source code with minor differences and observe the impact of the changes on the remaining system.

Advantages:

- Can catch problems with the quality of tests.
 - For example, it provides insight into missing test cases, such as untested input values
- Results in high coverage of the SUT.
- Can detect hidden defects
 - Highlights defects by changing code and measuring whether these changes result in a difference of execution for the program.
- Able to be fully automated for “quick” testing -- Actual computation may take a while
- Using tools, such as Pitest, provide testers with metrics to determine whether the system is adequately tested.

Disadvantages:

- Requires familiarity with the code base, as testers must determine how to catch each mutation. Thus, can be tedious and time consuming
 - Especially when dealing with equivalent mutants, which require the tester to determine if the defects are valid or not

- Can be unclear how the program was mutated if only a general description is given. More dependent on the tools used for testing (i.e. Pitest)
- Depending on the system, computation time required to run automation may be lengthy
- Incompatible with black box testing due to its reliance on source code.
- Injecting changes to source code can result in equivalent mutants

6. GUI Test Case Design Process

Test cases were developed for a variety of different environment, which include:

- Amazon.ca
- Ikea.ca
- Ebay.ca

The GUI test cases were designed around processes that require more than a trivial number of inputs, that for the most part are essential to operation of the systems being tested.

Other factors were also taken into account during the design phase of the test cases. For example, test cases required a measurable success metric, to ensure the tests success, or failure, could be validated completion. This pass or fail criteria was based of how the team interpreted the expected behaviour to be. There was an avoidance to using tests cases that involved personal information, such as passwords, for obvious reasons.

Ten functionalities of the web services were identified, as a group, for testing purposes. While each team member took at least two test cases to ensure GUI testing was properly understood.

6.1 Automated Verification

Each test case, with one exception, uses a visual cue from the screen to verify success. In every case, the verification is built into the end of the test script, so the test will return an error upon failure. In one case, the test involves a redirect to a different webpage as it's final step, so the successful redirection, as measured by the title bar of the webpage, is measured by the automated verification step at the end of the test. No intermediate checkpoints were used, since Selenium checks off each line as it gets completed. This removes the need for intermediate checkpoints since it's quite easy to identify where a test case broke down on failure.

6.2 Test Data

Ebay:

- Advanced Search
 - **About:** Ebay allows searches for items according to many parameters.
 - **Testing:** This functionality was tested by filling out the fields necessary to run an advanced search, then testing that the correct page was opened to display results by checking that the correct text and elements are present.
- Create Order

- **About:**
- **Testing:** This functionality was tested by selecting an item, then proceeding to checkout. The test finishes before buying the item. The presence of appropriate text and elements were used to verify output.
- Incorrect Login
 - **About:** Ebay allows users to create and login to accounts that would provide them with some additional features on the site. For example, saving a payment method to be associated with an account helps speed up the process of checkout.
 - **Testing:** The functionality was testing using a single input, a random invalid account. The presence of appropriate text and elements were used to verify output.

IKEA:

- Job Application
 - **About:** Ikea's website includes a career search, which provides a bulletin board of various positions currently available at the company. The steps in which to apply, include using their search engine to find jobs relating to the data inputted, and submit user information.
 - **Testing:** The test selects a job position and fills out most of the form. The test fails to fill in a field. The output is verified by checking that the correct error message is displayed.
- Location Finding
 - **About:** Ikea's website allows users to locate stores in various geographic locations, which are then displayed on Google Maps.
 - **Testing:** The test navigates to the location finder and views nearest Ikea location. The test verifies that Google Maps is open and displaying an Ikea store.
- Add to Cart
 - **About:** Ikea allows the online purchase of items using their website.
 - **Testing:** Selects an item and progresses part of the way through checkout. On the address page does not enter all fields. Verifies that execution is correct by checking the error that is displayed on the screen.

Amazon:

- Kindle
 - **About:** Amazon sells kindles for reading books electronically.
 - **Testing:** This test selects an ad about amazon kindles and proceeds part way through the purchase. It verifies that the correct output is reached by checking elements on the screen.
- Wishlist
 - **About:** Amazon provides a wishlist functionality to save item that users are interested in to buy later.

- **Testing:** This test searches for an item and adds it to the wishlist. It can only run when amazon is logged in to an account. It verifies that the correct output is reached by checking elements on the screen.

7. Selenium versus. SikuliX

As outline in this assignment, teams were to familiarize themselves with a pair of GUI testing tools, which included Selenium and SikuliX. Selenium, as defined by their site, is a open source record and playback test automation for the web. SikuliX, as defined by their site, allows users to automate anything seen on the screens of desktop computers running any OS. After creating a few tests on each tool, the following list was compiled regarding their advantages and disadvantages.

7.1 Selenium

Advantages: ¹

- Simple IDE
- Open source tool
- Wide range of supported languages
 - Save tests as HTML, Ruby, Python, C# scripts, or any other format
- Supports various operating environments (i.e. Windows, Linux, Mac, etc.)
- Multiple online resources in which to familiarize yourself with
 - Documentation found on their site is very helpful
- Supports “popular” web browsers
- Has resilient tests
 - Records multiple locators for each element interacted with. If one were to fail during playback, others are attempted until successful
- Test Case Reuse
 - Run command allows the ability to reuse a test inside another.
- Control Flow
 - Implements controlled flow structure with if, while and times command
- Extended through use of plugins

Disadvantages:

- Applicable for only web-based applications (i.e. no Windows application).
- Recording all possible cases for automation is not possible
 - (i.e)
 - Cannot automate file uploads, or screen shot capabilities
 - Cannot test connections with databases
 - Cannot create accounts when authentication requires use of captcha
- Not ideal for image testing

¹ <https://www.seleniumhq.org/selenium-ide/>

- Cannot be used to test “Big Data”
- Cannot handle dynamic portion of web applications, or multiple windows

7.2 SikuliX

Advantages: ²

- Supports various scripting languages:
 - Includes: Python, RobotFramework text-scripts, Ruby, JavaScript
 - Allows for use of java programming with any java aware programming/scripting language
- Uses image recognition to identify and control GUI components
 - Useful when there is no easy access or available insight to GUI's internals/source code
- Text recognition (OCR)
- Useful in automating repetitive tasks in various environments when you lack adequate tools, such as:
 - Daily use of web pages
 - Playing games
 - Administration of IT systems and networks
- Open source
- Good for testing overall behaviour of applications

Disadvantages:

- Not available on mobile devices
- Image recognition is imperfect
 - However, you can specify regions of images on the screen if you wish to improve accuracy
- Harder to modify existing tests, compared to Selenium
 - As a results, it is likely extremely sensitive to any changes of the GUI
- Resolution dependent
 - Script written for one resolution may not work in others

8. Division of Teamwork

Since this lab was divided into two sections, the approach for each component was slightly different.

During the mutation testing component, the first step involved calculating the initial coverage of the pre-existing test suite. After collecting the list of surviving mutants, each individual took on the task of identifying the logic resulting in the survival of various mutants. During this section,

² <http://www.sikulix.com/>

mutants that could be killed with additional test cases were identified (i.e. not equivalent mutants). Then those most familiar with the classes, went about developing the necessary test cases to increase mutation coverage. Pair programming was used to ensure each individual understood the process and that the developed code was proper. After satisfying the conditions of the lab assignment, members proceeded to GUI testing.

Outlined in the lab document, is the minimum requirement of 8 automated test cases. With 4 team members it made logical sense for each individual to take on a minimum of 2 tests. Everyone programmed in Selenium, however, who ever was able to get SikuliX to run properly on their system did some additional tests to familiarize themselves with the tool. Certain members chose to take on testing on different sites (i.e. amazon, ebay, and ikea) to test different functionalities. Those working on the same site, discussed which functionality they would undertake prior to beginning any tests in order to minimize overlap.

Constant use of a messenger group was used in an attempt to minimize miscommunication previously encountered.

9. Difficulties Encountered, Challenges Overcome, and Lessons Learned

As with many of the previous assignments, difficulties were encountered when using the recommended software. Certain plugins were not capable of functioning correctly on all devices, particularly MacOS. In this situation, when the team member attempted to run mutation tests, the program would not progress past the point where it connected to the localhost. To resolve the issue, individuals encountering such issues would assist those developing test cases through the technique of peer programming. This ensured each team member would be capable of satisfying the objective of the lab of understanding the process and concept of mutation testing.

Version control was a bit of an issue. Although there was ongoing communication through the use of messaging applications, some individuals may have not been up to date on the latest progression of team members. When multiple messages are being sent back and forth, sometimes people will lack the motivation to review all 200+ notifications. Therefore, there were merge conflicts resulting from people working on the test suite developing the same methods. In those situations, group meetings had to be called to order to clarify progress, state necessary tasks to be completed, and resolve existing issues. Although communicating online can be accommodating, it definitely lacks the same clarification provided by discussing in person the intentions of what code is suppose to do.

10. Comments and Feedback

Ensuring tools work across upon multiple platforms

A recurring issue with these labs is that the recommended software is not guaranteed to work properly across multiple platforms, despite what is advertised. As discussed in the previous

section, issues arose with Pitest plugin on Macs. There was also an issue Selenium on some occasions, which ultimately reduced the capacity of the impacted group members to complete the lab.

- **Solution:** Ensure that tools required for the completion of this lab are thoroughly tested across all popular platforms

Tools with guidelines are extremely useful

In the case of selenium, what appears to be a popular and well used testing tools, many sources of documentation and guides can be found. These are extremely useful when attempting to familiarize yourself with its layout and functionalities. This easily accessible information goes a long way and assists in reducing the time of trying to learn tools through trial and error, and therefore, do not go unappreciated by the students.

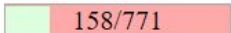
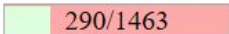
11. Appendices

Appendix A: Initial Pit Test Coverage Report and Surviving Mutants

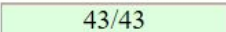
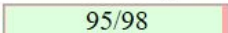
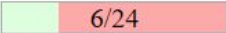
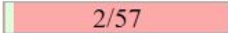
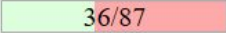
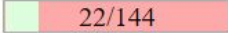
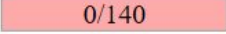
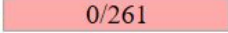
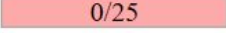
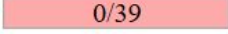
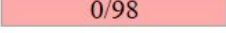
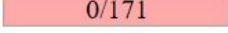
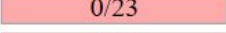
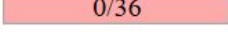
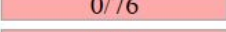
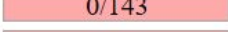
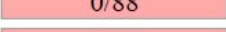
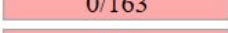
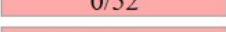
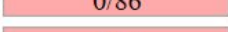
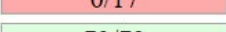
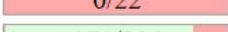
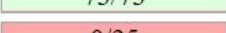
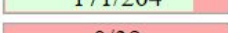
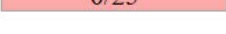
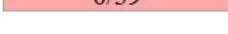
Pit Test Coverage Report

Package Summary


org.jfree.data





































Number of Classes	Line Coverage	Mutation Coverage
13	20%  158/771	20%  290/1463







Breakdown by Class

Name	Line Coverage	Mutation Coverage
DataUtilities.java	100%  43/43	97%  95/98
DefaultKeyedValue.java	25%  6/24	4%  2/57
DefaultKeyedValues.java	41%  36/87	15%  22/144
DefaultKeyedValues2D.java	0%  0/140	0%  0/261
DomainOrder.java	0%  0/25	0%  0/39
KeyToGroupMap.java	0%  0/98	0%  0/171
KeyedObject.java	0%  0/23	0%  0/36
KeyedObjects.java	0%  0/76	0%  0/143
KeyedObjects2D.java	0%  0/88	0%  0/163
KeyedValueComparator.java	0%  0/52	0%  0/86
KeyedValueComparatorType.java	0%  0/17	0%  0/22
Range.java	100%  73/73	84%  171/204
RangeType.java	0%  0/25	0%  0/39

Report generated by [PIT](#) 1.1.9

 SURVIVED (70)

-  JFreeChart_Lab4 (70)
 -  org.jfree.data (70)
 -  org.jfree.data.DataUtilities (3)
 -  181: removed conditional - replaced equality check with true
 -  188: removed conditional - replaced equality check with true
 -  193: mutated return of Object value for org/jfree/data/DataUtilities::getCumulativePercentages to (if (x != null) null
 -  org.jfree.data.DefaultKeyedValue (2)
 -  org.jfree.data.DefaultKeyedValues (32)
 -  org.jfree.data.Range (33)
 -  86: removed call to java/lang/StringBuilder::toString
 -  152: changed conditional boundary
 -  152: removed conditional - replaced comparison check with false
 -  157: changed conditional boundary
 -  157: changed conditional boundary
 -  157: removed conditional - replaced comparison check with true
 -  172: removed call to org/jfree/data/Range::contains
 -  172: removed conditional - replaced equality check with true
 -  173: changed conditional boundary
 -  176: changed conditional boundary
 -  176: removed conditional - replaced comparison check with true
 -  209: removed call to org/jfree/data/Range::getLowerBound
 -  210: removed call to org/jfree/data/Range::getUpperBound
 -  210: replaced call to java/lang/Math::max with argument
 -  230: changed conditional boundary
 -  230: removed call to org/jfree/data/Range::getLowerBound
 -  233: changed conditional boundary
 -  233: removed call to org/jfree/data/Range::getUpperBound
 -  280: Substituted 0 with 1
 -  322: Substituted 0.0 with 1.0
 -  322: changed conditional boundary
 -  363: removed call to java/lang/Double::doubleToLongBits
 -  365: Replaced Unsigned Shift Right with Shift Left
 -  365: Replaced XOR with AND
 -  365: Substituted 32 with 33
 -  366: removed call to java/lang/Double::doubleToLongBits
 -  367: Replaced Unsigned Shift Right with Shift Left

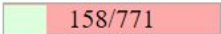
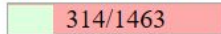
-  367: Replaced XOR with AND
-  367: Replaced integer addition with subtraction
-  367: Replaced integer multiplication with division
-  367: Substituted 29 with 30
-  367: Substituted 32 with 33
-  369: replaced return of integer sized value with (x == 0 ? 1 : 0)

Appendix B: Final Pit Test Coverage Report and Surviving Mutants

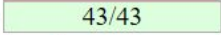
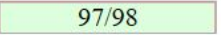
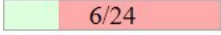
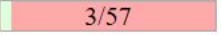
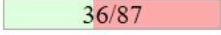
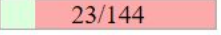
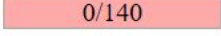
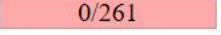
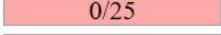
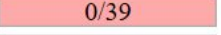
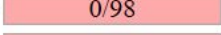
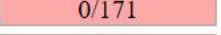
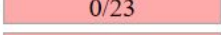
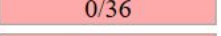
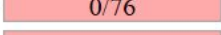
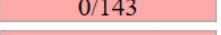
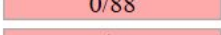
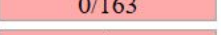
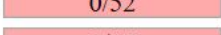
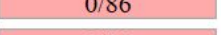

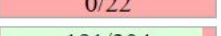
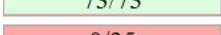
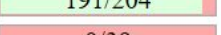
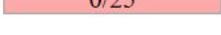
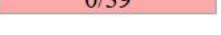
Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage	Mutation Coverage
13	20%  158/771	21%  314/1463

Breakdown by Class

Name	Line Coverage	Mutation Coverage
DataUtilities.java	100%  43/43	99%  97/98
DefaultKeyedValue.java	25%  6/24	5%  3/57
DefaultKeyedValues.java	41%  36/87	16%  23/144
DefaultKeyedValues2D.java	0%  0/140	0%  0/261
DomainOrder.java	0%  0/25	0%  0/39
KeyToGroupMap.java	0%  0/98	0%  0/171
KeyedObject.java	0%  0/23	0%  0/36
KeyedObjects.java	0%  0/76	0%  0/143
KeyedObjects2D.java	0%  0/88	0%  0/163
KeyedValueComparator.java	0%  0/52	0%  0/86
KeyedValueComparatorType.java	0%  0/17	0%  0/22
Range.java	100%  73/73	94%  191/204
RangeType.java	0%  0/25	0%  0/39

Report generated by [PIT](#) 1.1.9

