

Lab. Report #4 – Mutation Testing (Fault Injection) & GUI and Web Testing

Group #: 17

Student Names: Matteo Morrone, Adeshpal Virk, Sam Farzamfar, Taimoor Abrar

Table of Contents

1. Introduction
2. Analysis of 10 Mutants of the Range class
3. Report all the statistics and the mutation score for each test class
4. Analysis drawn on the effectiveness of each of the test classes
5. A discussion on the effect of equivalent mutants on mutation score accuracy
6. A discussion of what could have been done to improve the mutation score of the test suites
7. Why do we need mutation testing? Advantages and disadvantages of mutation testing
8. Explain your SELENIUM test case design process
9. Explain the use of assertions and checkpoints
10. How did you test each functionality with different test data
11. Discuss advantages and disadvantages of Selenium vs. Sikulix
12. How the team work/effort was divided and managed
13. Difficulties encountered, challenges overcome, and lessons learned
14. Comments/feedback on the lab itself

Introduction

In this lab we were tasked with performing both mutation and GUI testing. In the first half of the lab we learned about injecting mutated faults into our given code and how to use a mutation testing tool on said code. New test cases were developed to kill the surviving mutants and increase the mutation coverage by at least 10%. The second half utilized Selenium, a rather popular web-based tool for GUI testing, in order to practice the record and replay method of interface testing. An alternative tool was also used to compare and advantages/disadvantages were discussed.

Analysis of 10 Mutants of the Range class

1. `intersects(double b0, double b1): removed conditional - replaced comparison check with true → SURVIVED`

This mutation was able to survive our original test suite because unfortunately none of our tests accounted for what happens when the line “`return (b1 > this.lower)`” is changed to “`return (true)`”. This bug must have been missed as every time our tests entered the conditional statement this return was nested inside, a return value of true happened to be expected anyways.

2. `constrain(double value): removed conditional - replaced comparison check with false → SURVIVED`

This mutation was able to survive our original test suite because unfortunately none of our tests accounted for what happens when the line “`if(value > this.upper)`” is changed to “`if(false)`”. This bug must have been missed as every time our tests entered the conditional statement this conditional was nested within, an evaluation of false for this conditional happened to be expected anyways.

3. `shiftWithNoZeroCrossing(double value, double delta): removed conditional - replaced comparison check with false → SURVIVED`

This mutation was able to survive our original test suite because we neglected to test the method `shiftWithNoZeroCrossing(double value, double delta)` at all, thus the only time mutations were caught for this method was when it was called from other tested methods.

4. `scale(Range base, double factor): Substituted 0.0 with 1.0 → SURVIVED`

This mutation was able to survive our original test suite because our only test case for `scale(Range base, double factor)` had a value of 5 for factor, thus as 5 is greater than both 0.0 and 1.0, the substitution of 0.0 with 1.0 was not detected by this test.

5. `equals(Object obj): removed conditional - replaced equality check with false → SURVIVED`

This mutation was able to survive our original test suite because we neglected to test the overridden method `equals(Object obj)` at all, thus the only time mutations were caught for this method was when it was called from other tested methods.

6. `intersects(double b0, double b1): greater than to equal → KILLED`

This mutation was killed by our original test suite. The code “`return (b1 > this.lower)`” would have been changed to “`return (b1 == this.lower)`” and thus would have been caught by our `testIntersects_lower()` test. Here we test a case wherein `b1` is greater than `this.lower`, and `b0` is less than `this.lower`, therefore we would expect a return value of true, but instead received false by this mutant and was therefore killed.

7. `constrain(double value): negated conditional → KILLED`

This mutation was killed by our original test suite. The code “if(value > this.upper)” would have been changed to “if(!(value > this.upper))” and thus would have been caught by our testConstrain_below() test. Here we test a case wherein value is less than this.upper but contains(value) would still return false, therefore we would expect the function to return this.lower, but instead we received this.upper by this mutant and was therefore killed.

8. intersects(double b0, double b1): negated conditional → KILLED

This mutation was killed by our original test suite. The code “if(b0 <= this.lower)” would have been changed to “if(!(b0 <= this.lower))” and thus would have been caught by our testIntersects_nonRealRange() test. Here we test a case wherein b0 is greater than this.lower and b1 is greater than this.lower, therefore we would expect false, but instead we receive true by this mutant and was therefore killed.

9. intersects(double b0, double b1): greater or equal to less than → KILLED

This mutation was killed by our original test suite. The code “return (b0 < this.upper && b1 >= b0)” would have been changed to “return (b0 < this.upper && b1 < b0)” and thus would have been caught by our testIntersects_upperInRange() test. Here we test a case wherein b0 is less than this.upper but greater than this.lower and b1 is greater than b0, therefore we would expect the function to return true, but instead we get false from this mutant and was therefore killed.

10. getLength(): Replaced double subtraction with addition → KILLED

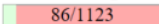
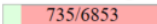
This mutation was killed by our original test suite. The code “return this.upper - this.lower” would have been changed to “return this.upper + this.lower” and thus would have been caught by our testGetLength_SameUpperLower() test. Here we test a case wherein this.upper equals this.lower, therefore we would expect the function to return 0, but instead we receive 10 by this mutant and was therefore killed.

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

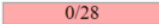
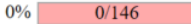
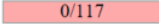
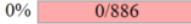
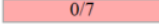
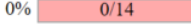
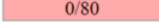
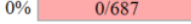
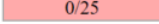
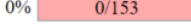
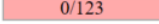
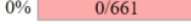
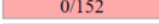
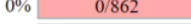
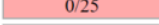
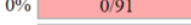
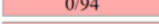
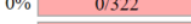
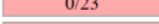
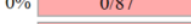
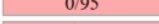
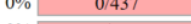
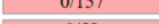
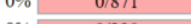
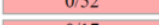
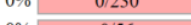
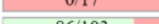
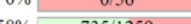
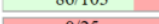
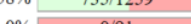
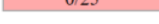
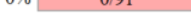
Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage	Mutation Coverage
16	8%  86/1123	11%  735/6853

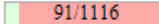
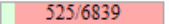
Breakdown by Class

Name	Line Coverage	Mutation Coverage
ComparableObjectItem.java	0%  0/28	0%  0/146
ComparableObjectSeries.java	0%  0/117	0%  0/886
ComparableObjectSeriesTest.java	0%  0/7	0%  0/14
DataUtilities.java	0%  0/80	0%  0/687
DefaultKevedValue.java	0%  0/25	0%  0/153
DefaultKevedValues.java	0%  0/123	0%  0/661
DefaultKevedValues2D.java	0%  0/152	0%  0/862
DomainOrder.java	0%  0/25	0%  0/91
KeyToGroupMap.java	0%  0/94	0%  0/322
KevedObject.java	0%  0/23	0%  0/87
KevedObjects.java	0%  0/95	0%  0/437
KevedObjects2D.java	0%  0/157	0%  0/871
KevedValueComparator.java	0%  0/52	0%  0/230
KevedValueComparatorType.java	0%  0/17	0%  0/56
Range.java	83%  86/103	58%  735/1259
RangeType.java	0%  0/25	0%  0/91



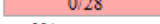
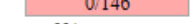
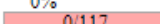
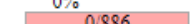

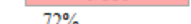
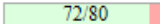
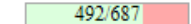


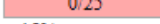
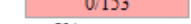
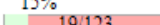
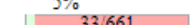
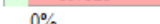
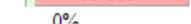
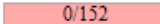
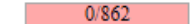


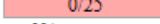
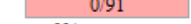
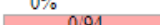
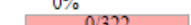
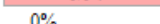

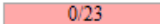
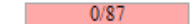
Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage	Mutation Coverage
15	8%  91/1116	8%  525/6839

Breakdown by Class

Name	Line Coverage	Mutation Coverage
ComparableObjectItem.java	0%  0/28	0%  0/146
ComparableObjectSeries.java	0%  0/117	0%  0/886
DataUtilities.java	90%  72/80	72%  492/687
DefaultKevedValue.java	0%  0/25	0%  0/153
DefaultKevedValues.java	15%  19/123	5%  33/661
DefaultKevedValues2D.java	0%  0/152	0%  0/862
DomainOrder.java	0%  0/25	0%  0/91
KeyToGroupMap.java	0%  0/94	0%  0/322
KevedObject.java	0%  0/23	0%  0/87
KevedObjects.java	0%  0/95	0%  0/437
KevedObjects2D.java	0%  0/157	0%  0/871
KevedValueComparator.java	0%  0/52	0%  0/230
KevedValueComparatorType.java	0%  0/17	0%  0/56
Range.java	0%  0/103	0%  0/1259
RangeType.java	0%  0/25	0%  0/91

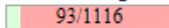
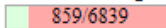
Report generated by [PIT](#) 1.4.11

The above images show the Pit Test Coverage Reports for our original test suite classes for both the Range and DataUtilities mutated classes. For our original RangeTest file we were able to achieve 58% mutation coverage, and for our original DataUtilitiesTest file we were able to achieve 72% mutation coverage.

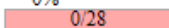
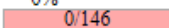
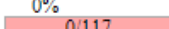
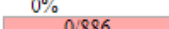


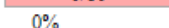
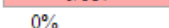
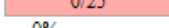
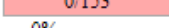
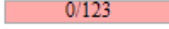
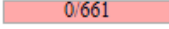
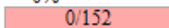
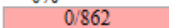
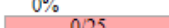
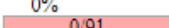
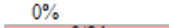



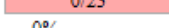
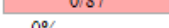
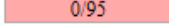
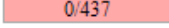
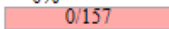
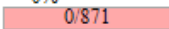
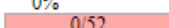
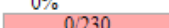
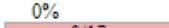
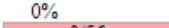
Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage	Mutation Coverage
15	8%  93/1116	13%  859/6839

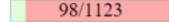
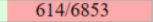
Breakdown by Class

Name	Line Coverage	Mutation Coverage
ComparableObjectItem.java	0%  0/28	0%  0/146
ComparableObjectSeries.java	0%  0/117	0%  0/886
DataUtilities.java	0%  0/80	0%  0/687
DefaultKeyedValue.java	0%  0/25	0%  0/153
DefaultKeyedValues.java	0%  0/123	0%  0/661
DefaultKeyedValues2D.java	0%  0/152	0%  0/862
DomainOrder.java	0%  0/25	0%  0/91
KeyToGroupMap.java	0%  0/94	0%  0/322
KeyedObject.java	0%  0/23	0%  0/87
KeyedObjects.java	0%  0/95	0%  0/437
KeyedObjects2D.java	0%  0/157	0%  0/871
KeyedValueComparator.java	0%  0/52	0%  0/230
KeyedValueComparatorType.java	0%  0/17	0%  0/56
Range.java	90%  93/103	68%  859/1259
RangeType.java	0%  0/25	0%  0/91

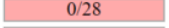
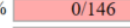
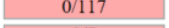
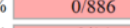
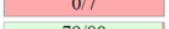
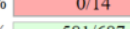
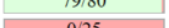
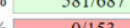
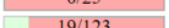
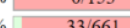
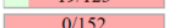
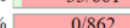
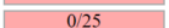
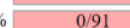
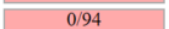
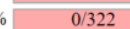
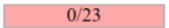
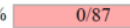
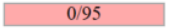
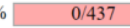
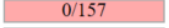
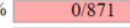
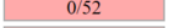
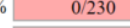
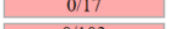
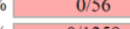
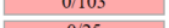
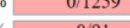
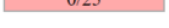
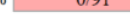


Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage	Mutation Coverage
16	9%  98/1123	9%  614/6853

Breakdown by Class

Name	Line Coverage	Mutation Coverage
ComparableObjectItem.java	0%  0/28	0%  0/146
ComparableObjectSeries.java	0%  0/117	0%  0/886
ComparableObjectSeriesTest.java	0%  0/7	0%  0/14
DataUtilities.java	99%  79/80	85%  581/687
DefaultKeyedValue.java	0%  0/25	0%  0/153
DefaultKeyedValues.java	15%  19/123	5%  33/661
DefaultKeyedValues2D.java	0%  0/152	0%  0/862
DomainOrder.java	0%  0/25	0%  0/91
KeyToGroupMap.java	0%  0/94	0%  0/322
KeyedObject.java	0%  0/23	0%  0/87
KeyedObjects.java	0%  0/95	0%  0/437
KeyedObjects2D.java	0%  0/157	0%  0/871
KeyedValueComparator.java	0%  0/52	0%  0/230
KeyedValueComparatorType.java	0%  0/17	0%  0/56
Range.java	0%  0/103	0%  0/1259
RangeType.java	0%  0/25	0%  0/91

The above images show the Pit Test Coverage Reports for our updated test suite classes for both the Range and DataUtilities mutated classes. For our updated RangeTest file we were able to achieve 68% mutation coverage (a 10% increase from the original), and for our original DataUtilitiesTest file we were able to achieve 85% mutation coverage (a 13% increase from the original).

Additional screenshots of statistics are provided at the end of the lab manual so as not to interfere with the organization of the writeup.

Analysis drawn on the effectiveness of each of the test classes

Both classes saw significant improvement in our updated test suites, with 10% and 13% improvement for RangeTest and DataUtilitiesTest respectively. With our final mutation scores coming out to 68% and 85%, it is safe to say that our test classes are fairly accurate. As is discussed in the next section, the accuracy of said mutation score relies heavily on the number of equivalent mutants, so it is also possible that a significant portion of the remaining 32% and 15% are made up of equivalent mutations as well. Both test suites ended up with more than double the original amount of tests for each of the two classes, with the creation of 113 new test cases total. The DataUtilities test class is clearly more effective than that of the Range class, however it is a significantly smaller file to test, resulting in less overall mutations (including equivalent mutations) and thus depicting RangeTest as having a deflated effectiveness rating.

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

A mutation score is calculated through dividing the number of mutants killed by a test suite by the number of all non-equivalent mutants. It is then clear that the number of equivalent mutants can very easily affect the accuracy of a mutation score if the exact number is not known. An equivalent mutant is functionally identical (though syntactically different) to the original code, and thus can be difficult to pick out of a large number of generated mutations. In class we were told that there can be between 4% to 45% equivalent mutants, which shows that it is not easy to predict the number of equivalent mutants that will be generated by your tool. If a program contains 40% equivalent mutants and you do a mutation score calculation assuming only 10% equivalent mutants, the accuracy of the score would be greatly skewed.

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

In order to improve our mutation score by 10% between our original and updated test suites for both the Range and DataUtilities files, we decided to start out by developing test cases for the boundary conditions we missed in the previous lab. This resulted in a plethora of new tests for our suite which drastically increased both mutation scores, however in order to actually break through the required 10% threshold we decided to test some equivalence classes that we had forgotten to cover last lab, as well as some additional tests to account for a few

functions that had been missed previously. All in all, 18 tests were added to the DataUtilities test suite, and a whopping 100 new tests were created for the Range test suite.

Why do we need mutation testing?

Advantages and disadvantages of mutation testing

The rationale behind mutation testing hinges on the idea that a better test suite should be able to detect more faults in a program. Thus, we utilize the injection of artificial bugs (mutations) and test the test suite's ability to find said bugs. This style helps users create effective test data in an interactive manner, helping with both test-case design and test evaluation. It is rather useful at a unit testing level, and forces the programmer to inspect the code with a clear goal in mind of killing mutants. However, it is very computationally intensive, and creates its own general and undecidable problem of equivalent mutants. Though it is automated and systematic with a quantitative score produced, the resources required, equivalent mutants, and the suggestion that mutants themselves may not actually be representative of real faults show that mutation testing is not an infallible style of testing.

Explain your SELENIUM test case design process

We designed our test cases by keeping in mind the elements on the sportchek webpage where data could be entered and verified. We chose the sportchek website as it was simple and had the most possible executable elements. Mainly we decided to split the elements into catalog items (user entered keywords and the results that would pop up), user registration (verifying the user information used for registration of a triangle account) , user promotional codes (verifying the user's promotional code in cart) and the cart (checking to see what effects user data would have on the cart, ie removing adding items).

Explain the use of assertions and checkpoints

In our test cases we decided that assertions would be enough to suffice with verification of the tests. In our test that tested the addition of an item to the cart we made an assertion to verify the correct input was put in the search bar and we asserted that the cart was not empty by checking if a button existed that only shows up once an item is in the cart. Similarly for emails and passwords we decided to put an assert right after the data had been entered to artificially

error check the entries at hand. For search bars as well a multitude of asserts and verifies were deployed in order to test proper functionality.

How did you test each functionality with different test data

Anytime there was a box to input data we decided to test it with any inputs. This would include any login boxes or search bars. Furthermore we also analyzed how the search functionality would perform given different sets of data. For the login boxes we created both a working account with a valid login and an invalid login and password. We essentially created a variety of data sets and deployed them in order to see the functionality of the asserts and the data handling capabilities of the website at hand. For drop down menus we tried erratic clicking and many more random methods to break the website. Similarly we tried null values in any data input boxes, in order to really solidify our tests.

Discuss advantages and disadvantages of Selenium vs. Sikulix

Selenium:

Selenium is a web browser automation tool. It is open-source , and has a large online community for web browser UI testing. It can be used for testing on multiple browsers (IE, Chrome, Firefox, Safari, Opera). It also has support in multiple languages. It's main drawback is its difficulty to use as you must be experienced in programming in order to use it.

Sikulix:

image-based recognition testing tool. It uses images for comparison and performs various operations like click(), doubleClick(), rightClick(), text(). It compares images by percentage value comparison (how much the given image matches the region on which it is searching). Sikuli can be combined with Selenium to perform both browser based and mouse clicks operations. Sikuli has limitation that it can work only what it sees and it cannot read the text, therefore its main drawback is it cannot perform asserts and verifications automatically

How the team work/effort was divided and managed

For the mutation portion of the lab, test cases were split evenly amongst group members for the development phase, followed by a review phase wherein we went over the developed tests as a group to pick out any inconsistencies or functionality errors before combining all test cases together into the test suite. For the GUI testing portion of the Lab each group member was assigned 2 tests to write and create assertions for in order to complete the necessary tests needed. Group members then joined together again to review the individually completed tests and verify their success. Each student then described their own test cases they had written in the report. This assured that everyone would get an equal workload. If ever a test was not working as intended the person responsible for the test was made aware and resolved the issue.

Difficulties encountered, challenges overcome, and lessons learned

Similar to the last lab, our first difficulty some of our team members encountered was our eclipse software missing the necessary jar files and configurations to properly run JUnit on the new Java project. After we remedied this issue, we found it rather difficult to work with the runtime of the mutation testing, as we would routinely need to wait for 20 minutes or more for the testing to calculate our mutation coverage. Additionally, it was nearly impossible for the Selenium program to find the username bar during our login tests and would only continue the rest of the test if any key was pressed to type something into the login box. This is a possible bug in the sportchek website as the script never seems to find the email box element until we manually type something in. Furthermore it took quite a long time to set up the Selenium IDE and to learn the commands with no prior knowledge.

Comments/feedback on the lab itself

The lab itself was very challenging yet interesting as it gave us an exciting glimpse into mutation and GUI testing, which presumably will be used in the field. The introduction of tools such as Pitest and Selenium that we could potentially be using in our careers in the future was quite beneficial. The instructions were significantly less descriptive in this lab compared to the previous three, though there was a helpful introduction preceding the document explaining useful terms and concepts.

ComparableObjectItemTest.java

Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage	Mutation Coverage
16	2% 17/1123	1% 59/6853

Breakdown by Class

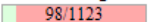
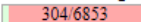
Name	Line Coverage	Mutation Coverage
ComparableObjectItem.java	61% 17/28	40% 59/146
ComparableObjectSeries.java	0% 0/117	0% 0/886
ComparableObjectSeriesTest.java	0% 0/7	0% 0/14
DataUtilities.java	0% 0/80	0% 0/687
DefaultKeyedValue.java	0% 0/25	0% 0/153
DefaultKeyedValues.java	0% 0/123	0% 0/661
DefaultKeyedValues2D.java	0% 0/152	0% 0/862
DomainOrder.java	0% 0/25	0% 0/91
KeyToGroupMap.java	0% 0/94	0% 0/322
KeyedObject.java	0% 0/23	0% 0/87
KeyedObjects.java	0% 0/95	0% 0/437
KeyedObjects2D.java	0% 0/157	0% 0/871
KeyedValueComparator.java	0% 0/52	0% 0/230
KeyedValueComparatorType.java	0% 0/17	0% 0/56
Range.java	0% 0/103	0% 0/1259
RangeType.java	0% 0/25	0% 0/91

ComparableObjectSeriesTest.java

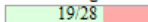

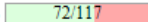
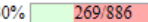
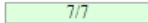
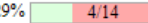
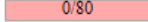
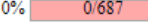
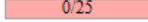

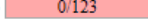
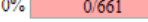
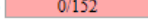
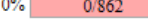
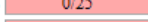

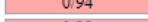

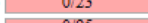

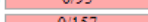

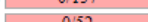

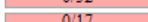

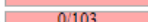
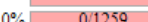
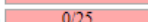

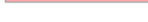
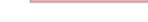
Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage	Mutation Coverage
16	9% 	4% 

Breakdown by Class

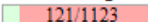
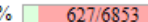
Name	Line Coverage	Mutation Coverage
ComparableObjectItem.java	68% 	21% 
ComparableObjectSeries.java	62% 	30% 
ComparableObjectSeriesTest.java	100% 	29% 
DataUtilities.java	0% 	0% 
DefaultKeyedValue.java	0% 	0% 
DefaultKeyedValues.java	0% 	0% 
DefaultKeyedValues2D.java	0% 	0% 
DomainOrder.java	0% 	0% 
KeyToGroupMap.java	0% 	0% 
KeyedObject.java	0% 	0% 
KeyedObjects.java	0% 	0% 
KeyedObjects2D.java	0% 	0% 
KeyedValueComparator.java	0% 	0% 
KeyedValueComparatorType.java	0% 	0% 
Range.java	0% 	0% 
RangeType.java	0% 	0% 

DataUtilitiesTest.java

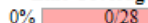
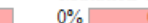

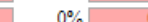

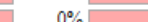
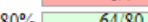
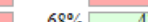
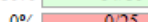
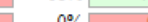

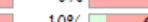
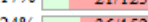
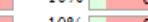
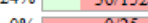
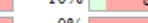

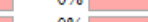

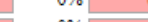

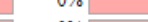

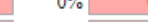

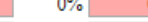

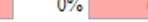

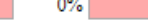

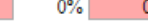
Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage	Mutation Coverage
16	11% 	9% 

Breakdown by Class

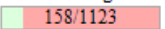
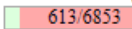
Name	Line Coverage	Mutation Coverage
ComparableObjectItem.java	0% 	0% 
ComparableObjectSeries.java	0% 	0% 
ComparableObjectSeriesTest.java	0% 	0% 
DataUtilities.java	80% 	68% 
DefaultKeyedValue.java	0% 	0% 
DefaultKeyedValues.java	17% 	10% 
DefaultKeyedValues2D.java	24% 	10% 
DomainOrder.java	0% 	0% 
KeyToGroupMap.java	0% 	0% 
KeyedObject.java	0% 	0% 
KeyedObjects.java	0% 	0% 
KeyedObjects2D.java	0% 	0% 
KeyedValueComparator.java	0% 	0% 
KeyedValueComparatorType.java	0% 	0% 
Range.java	0% 	0% 
RangeType.java	0% 	0% 

DefaultKeyedValues2DTest.java

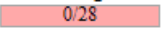
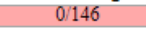
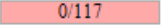
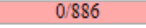
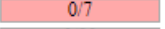
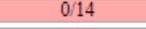
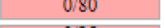
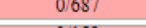
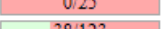
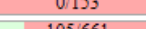
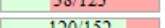
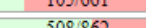
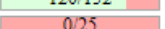
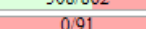
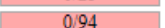
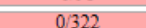
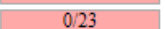
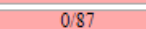
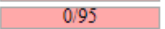
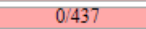
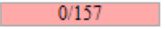
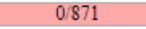
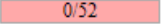
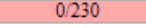
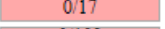
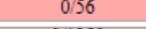
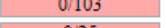
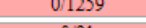
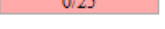
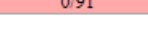


Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage	Mutation Coverage
16	14%  158/1123	9%  613/6853

Breakdown by Class

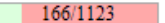
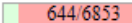
Name	Line Coverage	Mutation Coverage
ComparableObjectItem.java	0%  0/28	0%  0/146
ComparableObjectSeries.java	0%  0/117	0%  0/886
ComparableObjectSeriesTest.java	0%  0/7	0%  0/14
DataUtilities.java	0%  0/80	0%  0/687
DefaultKeyedValue.java	0%  0/25	0%  0/153
DefaultKeyedValues.java	31%  38/123	16%  105/661
DefaultKeyedValues2D.java	79%  120/152	59%  508/862
DomainOrder.java	0%  0/25	0%  0/91
KeyToGroupMap.java	0%  0/94	0%  0/322
KeyedObject.java	0%  0/23	0%  0/87
KeyedObjects.java	0%  0/95	0%  0/437
KeyedObjects2D.java	0%  0/157	0%  0/871
KeyedValueComparator.java	0%  0/52	0%  0/230
KeyedValueComparatorType.java	0%  0/17	0%  0/56
Range.java	0%  0/103	0%  0/1259
RangeType.java	0%  0/25	0%  0/91

DefaultKeyedValuesTest.java

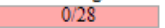
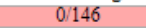
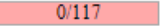
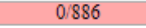
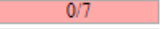
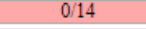
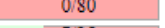
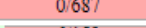
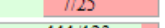
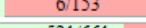
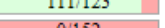
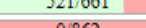
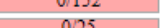
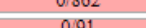
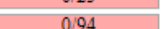
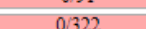
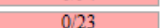
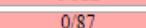
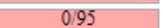
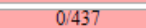
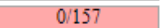
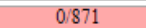
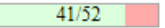
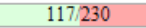
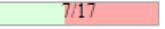
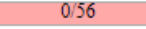
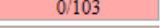
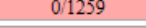
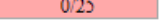
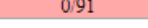


Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage	Mutation Coverage
16	15%  166/1123	9%  644/6853

Breakdown by Class

Name	Line Coverage	Mutation Coverage
ComparableObjectItem.java	0%  0/28	0%  0/146
ComparableObjectSeries.java	0%  0/117	0%  0/886
ComparableObjectSeriesTest.java	0%  0/7	0%  0/14
DataUtilities.java	0%  0/80	0%  0/687
DefaultKeyedValue.java	28%  7/25	4%  6/153
DefaultKeyedValues.java	90%  111/123	79%  521/661
DefaultKeyedValues2D.java	0%  0/152	0%  0/862
DomainOrder.java	0%  0/25	0%  0/91
KeyToGroupMap.java	0%  0/94	0%  0/322
KeyedObject.java	0%  0/23	0%  0/87
KeyedObjects.java	0%  0/95	0%  0/437
KeyedObjects2D.java	0%  0/157	0%  0/871
KeyedValueComparator.java	79%  41/52	51%  117/230
KeyedValueComparatorType.java	41%  7/17	0%  0/56
Range.java	0%  0/103	0%  0/1259
RangeType.java	0%  0/25	0%  0/91

DefaultKeyedValueTest.java

Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage	Mutation Coverage
16	2% 19/1123	1% 59/6853

Breakdown by Class

Name	Line Coverage	Mutation Coverage
ComparableObjectItem.java	0% 0/28	0% 0/146
ComparableObjectSeries.java	0% 0/117	0% 0/886
ComparableObjectSeriesTest.java	0% 0/7	0% 0/14
DataUtilities.java	0% 0/80	0% 0/687
DefaultKeyedValue.java	76% 19/25	39% 59/153
DefaultKeyedValues.java	0% 0/123	0% 0/661
DefaultKeyedValues2D.java	0% 0/152	0% 0/862
DomainOrder.java	0% 0/25	0% 0/91
KeyToGroupMap.java	0% 0/94	0% 0/322
KeyedObject.java	0% 0/23	0% 0/87
KeyedObjects.java	0% 0/95	0% 0/437
KeyedObjects2D.java	0% 0/157	0% 0/871
KeyedValueComparator.java	0% 0/52	0% 0/230
KeyedValueComparatorType.java	0% 0/17	0% 0/56
Range.java	0% 0/103	0% 0/1259
RangeType.java	0% 0/25	0% 0/91