SENG 438 - Software Testing, Reliability, and Quality

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

Group #: 3

Student Names:

Haniya Ahmed

Apostolos Scondrianis

Beau McCartney

Josh Vanderstoop

(Note that some labs require individual reports while others require one report for each group. Please see each lab document for details.)

1 Introduction

The objective of this lab is to gain a better understanding of mutation testing, mutation testing tools, and their effectiveness, analyze the effectiveness of our test suite by measuring our mutation score, and become familiar with automated GUI testing. In this lab, we will select one mutation testing tool to analyze our test suite, and then improve our mutation testing by 10% for each class under test: Range.java and DataUtilities.java. We will also implement the record and replay testing method to conduct GUI test automation on Walmart.ca. This will be done using Selenium, a common web-interface testing tool, and will be compared with results from using Sikulix. This will not only help familiarize us with GUI testing, but allow us to learn the difference between automated GUI testing and manual GUI testing.

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

Name	Line Coverage		Mutation Coverage	
ComparableObjectItem.java	0%	0/28	0%	0/146
ComparableObjectSeries.java	0%	0/117	0%	0/886
DataUtilities.java	71%	57/80	64%	438/687
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

We had a 64% mutation coverage score for our DataUtilities.java class with 438 of 687 mutations successfully killed.

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	18%	19/103	6%	77/1259
RangeType.java	0%	0/25	0%	0/91

We had a 6% mutation coverage score for our Range.java class with 77 of 1,259 mutants killed.

For both classes, we had all the test cases in one test suite for simplicity sake. As such, the reports above are cumulative for the entire class.

In the range class, the following 10 mutants were noticed:

1. Line 371: replaced return value with null for org/jfree/data/Range::shift
    - SURVIVED: This mutation returns a null object instead of the appropriate Range object. We did not have a test that tested both the upper bounds and lower bounds of a Range object to ensure neither of them are null when returned, so this mutant survived.
2. Line 123: replaced double return with 0.0d for org/jfree/data/Range::getLength
    - KILLED: Since we had several test cases to test the *getLength* method in Range.java, which requires the correct lower bound value to achieve the correct result, and also ensured that the correct number of decimal points were returned, this mutant was killed.
3. Line 349: removed call to org/jfree/data/Range::shift
    - KILLED: There are two *shift* functions in Range.java, and the first *shift* function only makes a call to another *shift* function. As such, the function is incomplete and empty without the call to org/jfree/data/Range::shift and will have no return. Since we have test cases that test this method, and require a return, this mutant was killed
4. Line 105: negated double field lower
    - KILLED: This mutant was killed since we have test cases for methods such as *getLength* for ranges with lower bounds other than 0 which will have incorrect results if the lower bound is negated.
5. Line 391: replaced double operation by second member
    - SURVIVED: This is an equivalent mutant that merely switches the order of two members passed to a Math.*min* method, which does not alter the results returned from the method. As such, this mutant survived.
6. Line 390: substituted 0.0 with 1.0
    - SURVIVED: This mutant survived since, although we made calls to the *shift* method, we did not test for shifts that no zero crossing, as such, although the *shiftWithNoZeroCrossing* method is automatically called in the *shift* method, the part of the method where the mutant appears (value == 0) was never needed or used, and mutants in this method survived.
7. Line 114: replaced double return with 0.0d for org/jfree/data/Range::getUpperBound

- KILLED: Since we had several test cases to test the *getLength* method in Range.java, which requires the correct upper bound value to achieve the correct result, and also ensured that the correct number of decimal points were returned, this mutant was killed.

8. Line 388: replaced return value with -(x+1) for org/jfree/data/Range::shiftWithNoZeroCrossing
    - KILLED: Since we have test cases which have an upper bound that is greater than 0, the return value of this line of code would be incorrect, and x is not a variable in the scope either. As such, our several *shift* method test cases are able to kill this mutant.
9. Line 475: removed call to java/lang/StringBuilder:: append
    - KILLED: Appending is needed for the *toString* method in order to concatenate the objects from the Range.java class, along with other string fragments. Without appending, the incorrect result is produced, and our test cases will fail. Thus, this mutant is killed.
10. Line 475: replaced return value with “” for org/jfree/data/Range::toString
    - KILLED: The *toString* method should never return an empty string, as such, all our *toString* test cases fail in the presence of this mutant and kill the mutant.

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

Based on the results above, the DataUtilitiesTest.jave class was moderately successful in killing mutants that were created in the class by the PIT test, with a 64% mutant coverage score. However, the RangeTest.java class was much less successful in killing mutants with only a 6% mutant coverage score. A great reason for this is that 5 of 7 methods were covered in DataUtilities.java, whereas 5 of 15 methods were tested in Range.java - DataUtilities.java simply had more methods in its class covered in our test suite. Accordingly, the line coverage in DataUtilities.java is 71% in comparison to the 18% line coverage in Range.java. Despite this, when looking into the PIT summary report, we found that many of the mutants present in the DataUtilities class were equivalent mutants, whereas the Range.java class had much less equivalent mutants present. As such, when taking a deeper look, both classes had effective test cases; however, DataUtilities.java was more effective as it had more equivalent mutants than Range.java. Unfortunately, the report does not account for equivalent mutants as those must be identified manually.

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

Equivalent mutants can decrease the mutation coverage of our code, without actually validating the quality of our test suites, since an equivalent mutant maintains the same behavior as the original program. We could claim that it really is not a real mutant from a test perspective, i.e., it does not inject a bug that will test the ability of our test cases to catch a fault in the System under Test. It is important for the Quality Assurance Team when inspecting the Mutation Score to take into consideration how many equivalent mutants exist and perhaps adjust the Mutation Score to get a better idea about how robust their test suite actually is, since automating a mutation report will not be able to isolate equivalent mutants. Furthermore, the larger a program becomes, the larger the number of equivalent mutants, so this must be taken into account when checking mutation score accuracy.

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

To improve the mutation score of the DataUtilitiesTest.java test suite, we could have been more robust with our testing, such as ensuring that certain exceptions were actually thrown, like the *IllegalArgumentException* for null arguments. However, many of the mutants that survived from the DataUtilities.java were equivalent mutants that did not change the behavior of the program. As such, although our test cases covered several boundaries and equivalencies, they would all still pass in the presence of the mutant. However, the mutation score and analysis definitely made us realize where we could add more exception testing. With this, we were able to kill more than 1 mutant where *ParamChecks.nullNotPermitted* was removed from the code. Beyond this, another way to improve the mutation score is by adding more line coverage, which will likely address the mutants that survived due to lack of coverage.

To improve RangeTest.java, our group could have further tested methods that were called within the methods that we tested, such as *shiftWithNoZeroCrossing*. In doing so, we would be able to kill 8 more of the surviving mutants. Beyond this, if our group was to go above the expectations of previous labs and test all 15 methods in Range.java, every metric for assessing the effectiveness of a test class would have experienced improvements.

Accordingly, the new test cases we designed for DataUtilities.java were:

- *null2DArrayTest()*: mutant created which removes the call to *nullNotPermitted* in ParamChecks.java. Ensures that the Illegal Argument Exception from the removed call is thrown when a null object is passed in.
- *testCumulativePercentage()*: several mutants created in the *getCumulativePercentages* method which alter the return value and calculation

conditions and variables. Ensures that the returned percentage is correct with a mock KeyedValues object.

- *cloneTenByOneArray()*: several mutants in the *clone* method which alter the contents of the clones array by mutating conditions and variables. Ensures that the contents of the returned array are all correct.

The new test cases designed for Range.java were:

- *shiftExpectNotNullMutantKiller1()*: mutant created returned null instead of a shifted Range object. Asserts Not Null.
- *shiftExpectNotNullMutantKiller3()*: mutant created returned null in *shiftWithNoZeroCrossing*, asserts range returned is correct.
- *shiftWithNoZeroCrossingMutantKiller1()*: mutant created allowed zero crossing due to a condition negation. Asserts that zero crossing is properly enabled. Checks lower bounds.
- *shiftWithNoZeroCrossingMutantKiller2()*: mutant created allowed zero crossing due to a condition negation. Asserts that zero crossing is properly enabled. Checks upper bounds.
- *shiftWithNoZeroCrossingMutantKiller3()*: mutant created a subtraction where addition was present. Asserts range returned used correct arithmetic.
- *shiftWithNoZeroCrossingMutantKiller4()*: mutant created a subtraction where addition was present. Asserts range returned used correct arithmetic.
- *testTwoNumberBounds()*: several mutants created on line 448 for the *isNaNTange* method where the values and conditions of the return value are altered. Ensures the returned value is the correct boolean.
- *testPositiveScalingDouble()*: several mutants created to alter the return value. Ensures the Range object is scaled correctly.
- *testNegativeScalingDouble()*: mutant created which removes call to *IllegalArgumentException*. Passes in a negative scaling factor and expects an Illegal Argument Exception.

Our final results are included in the screenshots below:

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	36%	37/103	22%	271/1259
RangeType.java	0%	0/25	0%	0/91

Name	Line Coverage		Mutation Coverage	
ComparableObjectItem.java	0%	0/28	0%	0/20
ComparableObjectSeries.java	0%	0/117	0%	0/97
DataUtilities.java	99%	79/80	88%	63/72
DefaultKeyedValue.java	0%	0/25	0%	0/21
DefaultKeyedValues.java	15%	19/123	7%	5/73
DefaultKeyedValues2D.java	0%	0/152	0%	0/99
DomainOrder.java	0%	0/25	0%	0/15
KeyToGroupMap.java	0%	0/94	0%	0/43
KeyedObject.java	0%	0/23	0%	0/13
KeyedObjects.java	0%	0/95	0%	0/54
KeyedObjects2D.java	0%	0/157	0%	0/99
KeyedValueComparator.java	0%	0/52	0%	0/27
KeyedValueComparatorType.java	0%	0/17	0%	0/9
Range.java	0%	0/103	0%	0/123
RangeType.java	0%	0/25	0%	0/15

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

Mutation testing can facilitate the design or the evaluation of the quality of a test suite. Compared to simple coverage metrics such as line coverage which only guarantees the execution of the code segment (reachability), mutation coverage actually adds a stressor that challenges a test suite (i.e. a bug) and ensures a statement is reached and checks the state of the execution. It creates unit copies of the SUT with a single change which is called a mutation, and then the test suite is run against the new code to assess if the bug injection is captured by the test suite. Some advantages and disadvantages of mutation testing follow :

Advantages :

- Mutation Testing can be automated using frameworks such as PIT
- Mutation Testing is Systematic
- There is a tangible goal to be reached when testing using mutation coverage, which gives an indication to the Quality Assurance Team when to stop testing.
- Allows to judge different suites for the a SUT using the same scale (mutation score)
- Forces the programmer to inspect code and create a data set that will expose certain kinds of faults
- Ensure that bugs that alter the program behavior are caught and fixed

Disadvantages :

- Computationally intensive, large number of mutants possible which means we need to find a way of selecting a subset of mutation operators to stress the test suite
- Equivalent mutants alter the true mutation score of a test suite and should always be considered when judging the quality of a test suite
- Equivalent mutations must be manually detected and may consume a lot of time before being identified as equivalent

 # 7 Explain your SELENIUM test case design process

For GUI testing, we used Selenium and chose to test the Walmart webpage with 8 core functionalities selected for testing which were:

- Login
    - Entering valid credentials
    - Entering an invalid username
    - Entering an invalid password
- Update Cart
    - Adding a new clothing item to cart
    - Increasing the quantity of an item in the cart

        - Decreasing the quantity of an item in the cart
        - Removing an item from cart
        - Adding a food item to cart (kitkat)
    - Find Store Near Me
        - Find store using postal code
        - Find store using current location
    - Search Item By Number
        - Search for an item using keystrokes
        - Search for an item using clicks
        - Search an item found on WalmartGrocery from main page
    - Filter Items By Price
        - Search for item with a screen size requirement (17 inch)
        - Search for items in a price range
            - Two different departments
    - Alter address information for registered user
        - Change the phone number
    - Change password
        - Change password to current password
        - Change password to a different password
        - Change password to a password that is too short
    - Search item by name
        - Search for Mario Kart 8
        - Search for KitKat

We used assertions in each of our test cases to create verification points and ensure that each test case was not only executed, but produced accurate results.

 # 8 Explain the use of assertions and checkpoints

Using assertions and checkpoints when testing GUI's for functionalities ensures that the tests run successfully throughout the entire process, and also produce the correct results. For tests that involve longer runtimes, such as signing into an account and checking items in a user's cart, checkpoints are available to ensure that there are not intermittent mistakes that arise during the tests and to pause the test case execution to view the state of the web page. To provide a concrete conclusion of whether the functionality was successful, developers can assert that unique elements of the final page are present. For example, when searching for an item, if the final page does not contain the description or image of that item, or produces an incorrect result the test may pass if assertion points are not added, despite the test being unsuccessful.

9 how did you test each functionality with different test data

All selected functionalities were tested as described in section 7 of our report. For tests such as searching items by number, the differing test data is the item that is being searched. To further test the functionality, a combination of mouse clicks and keystrokes such as ‘ENTER’ allow for greater breadth in functionality testing.

When testing store locations, using postal codes and current locations was the best method to differentiate the test data that is available for the functionality. For the test of updating an address, we first successfully logged in with a valid account and then proceeded to edit the current address that the user has in its profile, specifically we altered one value of the phone number on the address field, and then asserted that an updated occurred by checking that the walmart page returned success update message.

For the filter functionalities we tested 2 different departments, electronics (specifically windows laptops with a size of 17”) and small appliances (specifically single brew coffee makers). In the case of windows laptops we limited the tests in the range of 499.98 and 505 in order to have a single result and we asserted to see if the laptop that is supposed to appear in the results appeared (2 Filters) and in the case of the coffee maker we also used a limited range to see if the expected coffee maker would appear and subsequently also asserted to see if the right coffee maker appeared.

For the tests that tested the login function, there was a valid username and password passed in, and a verification point added to ensure the system successfully logs in. The functionality was also tested individually with an invalid username and an invalid password; for these two tests, verification points were added to ensure that an unsuccessful login pop-up appeared.

When testing cart updates, the functionality was first tested by adding a new item from the clothing department, then by incrementing and decrementing the same item. For each of these tests, a verification point was added to ensure the correct quantity of the item, and the order of the tests was also taken into account. Lastly, the functionality was tested by removing the item from the cart, and a verification point was added to ensure the cart was empty. This functionality was tested with both a clothing item (swimsuit) and a food item (chocolate).

For testing password change, we tried changing the password to the same password as the current one, changing the password to a different one, and changing the password to an invalid password that was too short.

To test the searching functionality (searching by name), we searched for chocolate and for Mario Kart 8, and added assert statements to ensure that the results were accurate and the items did indeed show up (if they were available).

10 Discuss advantages and disadvantages of Selenium vs. Sikulix

Selenium allows for recording of the activity of your mouse and keyboard as you interact with the webpage as well as writing testing using languages such as Ruby, Python, C#, Java, for the creation of tests. Additionally, it is available as a Chrome and Firefox extension, which are the two most popular browsers that are available for multiple platforms such as Mac OS, Linux, and Windows platforms. Furthermore, it allows you to interact with the web page while creating the test cases to search for specific elements and/or create events that interact with the elements you discovered. The problem is that it limits the testing on web pages. Comparatively, Sikulix is an IDE/Software framework that you can download to your machine independently written in Java that works on Windows, and most Linux systems. It allows for automation of any repetitive task (not limited to testing web pages), which means you can test any sort of software such as a Java GUI application. Furthermore, where Selenium is a browser extension (limited to Chrome and Firefox), SikuliX is an application downloaded straight to one's system. Although Selenium is then limited in availability across various browsers, it is much more portable and can work on any device that has the mentioned browsers; however, it can only test public websites. Sikulix can test any application on the system and even multiple applications at once that might interact with one another. Furthermore, it also allows insertion of screenshots that can help with locating elements that must be pressed, and even allows the user to set offsets to better locate where the area that must be clicked, but this limits the portability of the tests as they become machine-specific. Both Sikulix and Selenium have functionalities that automate testing, such as recording mouse and keyboard actions. Finally, Sikulix does not allow the usage of the machine even if two separate monitors are connected, while Selenium allows usages of the local machine while test cases are being simulated on the web pages.

11 How the team work/effort was divided and managed

Each group member created a number of automated selenium tests for two functionalities on the Walmart.ca website. With the struggles that a member encountered, the other three were able to complete the mutation testing section together. Haniya, Josh, and Apostolos reviewed the mutations that survived the test suite and determined which were able to be killed through further testing. Each member curated several tests to reduce the number of surviving mutants in both RangeTest and DataUtilitiesTest.

12 Difficulties encountered, challenges overcome, and lessons learned

Overall, the assignment was not too challenging. The biggest challenge once again was to run the projects of eclipse across different configurations. All the previous labs set us up for success during mutation testing and the information provided in lectures regarding mutation testing was very helpful in successfully completing the lab. Apart from this, there were some small challenges such as manually finding equivalent mutants, which consumed a lot of time, and generating the PIT reports. Once again, the importance of starting your assignments early to deal with complications and good communication between the team members allowed us to enjoy our meetings. It was fun to explore GUI Testing with Selenium IDE on web pages.

 # 13 Comments/feedback on the lab itself

Beau: Due to a family member being severely injured, I was not able to take on as much responsibility for this lab as the previous labs. I learned about discussing personal situations with teammates, and finding work where I can when circumstances don't allow for me to consistently meet with my group.

Josh: The design of this lab was confusing at times, as with previous labs, however our group was able to collaborate with each other to determine the best expected outcomes from this process. I enjoyed the opportunity to use automated gui testing, as this pertains to a number of my personal projects.

Apostolos : I found this lab more interesting compared to the previous ones, because it allowed me to envision how development of software can go wrong. It showed me how during the design and implementation of a big project with multiple developers one line of code being altered can create faults and subsequent failures and how we can emulate that behavior using mutation testing to see if our test suite is able to identify those errors across different versions of test suites and Systems Under Testing.

Haniya: This lab definitely had more of a learning curve than previous labs. Finding where bugs occurred, why they occurred, and how to resolve them, sometimes to find that they were not resolvable, definitely was the most time-consuming aspect of this lab. Some instructions were unclear; however, we were still able to complete the lab on time with our combined effort. GUI testing was definitely the easier portion of the lab and I realized the significance of having verification points in the test scripts.