Introduction

The objective of this lab is to gain familiarity with both Mutation and GUI Testing. This lab is composed of two parts; the first's focus being on Mutation testing and the second's focus being on GUI testing. In part 1, we will understand why mutation testing is important and useful, and become familiar with the PITEST software tool. In the second part, the Selenium software tool is used to test the sportchek website.

Mutation Analysis of 10 Mutants from Range Class (Original Test Suite)

Method: *Range shift(Range base, double delta, boolean allowZeroCrossing)*
Mutant Generated: Removed call to org/jfree/chart/util/ParamChecks::nullNotPermitted
Status of Survival: Survived
Killed By: None
Test Suite: RangeShiftTest.java
Discussion: The mutant removed the line that calls the notNullPermitted function in the ParamChecks. This function throws an *IllegalArgumentException* if the Range object *base = null*, preventing errors propagating in the following lines of code. This mutation, thus, allows null values to pass into the code past the line the function is called. This would result in potential *NullPointerExceptions* being called with attempted use of the Range object *base* when *base = null* and propagate errors.

Method: *boolean equals(Object obj)*
Mutant Generated: Negated conditional
Status of Survival: Killed
Killed By: *equalsRange2IsNotARange()*
Test Suite: RangeEqualsTest.java
Discussion: The mutant created in this line takes the condition *!(obj instanceof Range)* and negates it to *(obj instanceof Range)*. This in turn, would cause reverse functionality in the if statement surrounding it. In this case, it would cause the function to return false if the *obj* input is of the class Range. The unit test *EqualsRange2IsNotARange()* enforces the intended behaviour by checking to see if the output boolean matches to false if when *obj* is not of the class type Range, catching this mutant's reversed effects.

Method: *boolean intersects(double b0, double b1)*
Mutant Generated: Changed conditional boundary
Status of Survival: Survived
Killed By: None
Test Suite: RangeIntersectsTest.java
Discussion: The mutation's change to the boundary condition causes the if statement of intersects() method to run only the code within it if b0 < this.lower ignoring the case where b0 = this.lower, different from the source code behaviour. This change results in an incorrect output from the function where b0 = this.lower and b0 = this.upper as it will cause the function to return false where it should be returning true. There was no specific test case that checked for this possibility, and thus, the mutant survived.

Method: *Range expandToInclude(Range range, double value)*
Mutant Generated: removed conditional - replaced equality check with false
Status of Survival: Killed
Killed By: *rangeIsNull()*
Test Suite: RangeExpandToIncludeTest.java
Discussion: This mutation removes the condition *range == null* in the method's first if statement and replaces it with *false*. This change effectively makes it so that the code within the if statement will never run as the statement will always have a false condition. Even so, this mutation is caught by the unit test *rangeIsNull()* which makes sure that when *range == null*, the function will always return a range with its upper and lower bound equal to *value*.

Method: *Range combineIgnoringNaN(Range range1, Range range2)*
Mutant Generated: removed conditional - replaced equality check with true
Status of Survival: Killed
Killed By: *combineNoNullParams()*
Test Suite: RangeCombineIgnoringNaNTest.java
Discussion: This mutation removes the condition *range1 == null* in the method's first if statement and replaces it with *true*. This change effectively makes it so that the code within the if statement will always run as the statement will always have a true condition. Even so, this mutation is caught by the unit test *combineNoNullParams()* which makes sure that when *range1 != null* and *range2 != null*, the function will always return a range with its upper and lower bound appropriate to its input test values, killing this mutant.

Method: Range expand(Range range, double lowerMargin, double upperMargin)
Mutant Generated: removed call to org/jfree/chart/util/ParamChecks::nullNotPermitted
Status of Survival: Survived
Killed By: None
Test Suite: RangeExpandTest.java
Discussion: This mutation removes the ParamChecks.nullNotPermitted() call on the range argument that is passed. This call is present to ensure that the range that is passed in the argument is not null, however, once you remove this check, there is nothing present in the function which checks to see if the range is null or not. In the RangeExpandTest test suite, there is no unit test that tests the functionality of the function when a null range is passed. According to the source code (without the mutation), an invalid argument exception should be thrown, if any exception other than that is thrown, the test case should fail. Since our test suite did not have a unit test that tested for a) null range b) exceptions thrown from function, the mutant survived.

Method: Range expandToInclude(Range range, double value)
Mutant Generated: replaced return value with null for org/jfree/data/Range::expandToInclude
Status of Survival: Killed
Killed By: rangeIsNotNull
Test Suite: RangeExpandToIncludeTest.java
Discussion: The mutant returns null in function expandToInclude(Range range, double value) instead of returning an object of range. However, this mutant is caught and killed by the test case rangeIsNotNull because of the assertEquals statement. This mutant was detected by this statement because the test case rangeIsNotNull was checking if an appropriate value was returned by the method expandToInclude, which means it was able to identify that a 'bad' value was being returned and thus failed, killing the mutant.

Method: boolean contains(double value)
Mutant Generated: changed conditional boundary
Status of Survival: Survived
Killed By: None
Test Suite: Range_containsTest.java
Discussion: This mutation changes the second conditional statement in the return statement by changing value <= this.upper to value < this.upper. This mutation survived because the test suite for this method does not check what happens when the argument value is equal to the upper bound of the range. Since there is no unit test that asserts that even when the value is == to this.upper, the value, assuming the first conditional also holds true, the function should return true. However, the mutant passed since the unit tests do not consider the case where value == this.upper and only consider value > than this.upper, thus this means mutant survived.

Method: Range expand(Range range, double lowerMargin, double upperMargin)
Mutant Generated:removed call to org/jfree/data/Range::getLength
Status of Survival: Killed
Killed By: RangeAppropriateValuesTest
Test Suite: RangeExpandTest.java
Discussion: This mutant removed the getLength() function call which would cause invalid lower and upper bound to be calculated in the function expand(). In the test suite RangeAppropriateValuesTest() tests to check if the current bounds have been evaluated from the return range result value. There are individual assert statements that check if the correct value for both the upper and lower bound was evaluated. Since the mutant caused the absence of the getLength function it would have given incorrect values for the bounds, which was caught by assert statements in RangeAppropriateValuesTest unit test case, thus mutant was killed.

Method: boolean contains(double value)
Mutant Generated: changed conditional boundary
Status of Survival: Survived
Killed By: None
Test Suite: Range_containsTest.java
Discussion: This mutation changes the first conditional statement in the return statement by changing value >= this.lower to value > this.lower. This mutation survived because the test suite for this method does not check what happens when the argument value is equal to the lower bound of the range. Since there is no unit test that asserts that even when the value is == to this.lower, the value, assuming the second conditional also holds true, the function should return true. However, the mutant passed since the unit tests do not consider the case where value == this.lower and target only value > than this.lower, thus this means mutant survived.

Mutation Score and Statistics

Original Test Suite

Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage		Mutation Coverage	
15	15%	169/1116	18%	147/798

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
ComparableObjectItem.java	0%	0/28	0%	0/20
ComparableObjectSeries.java	0%	0/117	0%	0/96
DataUtilities.java	90%	72/80	73%	56/77
DefaultKeyedValue.java	0%	0/25	0%	0/21
DefaultKeyedValues.java	4%	5/123	0%	0/79
DefaultKeyedValues2D.java	0%	0/152	0%	0/100
DomainOrder.java	0%	0/25	0%	0/16
KeyToGroupMap.java	0%	0/94	0%	0/44
KeyedObject.java	0%	0/23	0%	0/13
KeyedObjects.java	0%	0/95	0%	0/55
KeyedObjects2D.java	0%	0/157	0%	0/100
KeyedValueComparator.java	0%	0/52	0%	0/27
KeyedValueComparatorType.java	0%	0/17	0%	0/9
Range.java	89%	92/103	73%	91/125
RangeType.java	0%	0/25	0%	0/16

Enhanced Test Suite

Pit Test Coverage Report

Package Summary

org.jfree.data

Number of Classes	Line Coverage		Mutation Coverage	
15	16%	174/1116	21%	171/798

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
ComparableObjectItem.java	0%	0/28	0%	0/20
ComparableObjectSeries.java	0%	0/117	0%	0/96
DataUtilities.java	90%	72/80	86%	66/77
DefaultKeyedValue.java	0%	0/25	0%	0/21
DefaultKeyedValues.java	4%	5/123	0%	0/79
DefaultKeyedValues2D.java	0%	0/152	0%	0/100
DomainOrder.java	0%	0/25	0%	0/16
KeyToGroupMap.java	0%	0/94	0%	0/44
KeyedObject.java	0%	0/23	0%	0/13
KeyedObjects.java	0%	0/95	0%	0/55
KeyedObjects2D.java	0%	0/157	0%	0/100
KeyedValueComparator.java	0%	0/52	0%	0/27
KeyedValueComparatorType.java	0%	0/17	0%	0/9
Range.java	94%	97/103	84%	105/125
RangeType.java	0%	0/25	0%	0/16

How Mutation Score was Improved (Design Strategy)

Our design strategy to increase the mutation score of our original test suite was centred around further enforcing the behaviour that the source code functions were intended to implement. In this way, testing suites were enhanced to check for correct service function regardless of the implementation of the current code (the mutation). We decided upon this strategy since it is much more appropriate for unit tests to test whether code is functioning correctly rather than creating tests based on their implementation when that implementation is constantly being modified by mutants.

This testing strategy opposes what we had implemented for coverage testing where we only checked for all the paths of the given code that had been covered by the testing suites. By testing in this way, it soon became clear that the coverage testing strategy gives little to no focus on the intended system behaviour being implemented.

Clear examples of our new testing strategy's implementation are tests for source code functions that made a call to the nullNotPermitted() function from the ParamChecks class. Functions calling this function intend to avoid having null objects in their implementation to avoid unintended errors in the computation of lines following after this call. This is enforced by the nullNotPermitted() by throwing an IllegalArgumentException when the checked object is null at that point in the code. Mutations of this implementation often remove this line and its functionality, allowing null values to propagate throughout the code. Our original test cases simply caught any exceptions thrown so that this line in the code would be confirmed to be covered. However, this ignores cases where other exceptions are thrown because of serious computational errors other than the IllegalArgumentException like NullPointerExceptions, which occur when actually accessing an object that is null. Our mutation testing strategy further specifies whether an exception thrown passes or fails its intended function. If an exception other than the IllegalArgumentException is thrown, this means that there is a serious computational error and thus the test would be deemed a failure. This kills the mutant and strengthens our test cases. Our other test suite enhancements follow this same strategy similarly, specifying the proper function of the SUT.

With the use of this testing strategy, we were able to increase of mutation coverage by more than 10% for each class. The DataUtilities class increased its mutation coverage from 73% to 86% (13% increase). The Range class increased its mutation coverage from 73% to 84% (11% increase).

Equivalent Mutants and Their Effect on Test Accuracy

Mutants help with testing test suites to determine whether or not they are able to detect changes in the SUT when there is improper behaviour. This is done by modifying the code in the SUT and checking to see if test cases have caught the discrepancies. If the change is caught by the test cases, it means that the mutant was killed. This is an effective strategy for testing to see whether the unit testing was performed correctly. However, equivalent mutants can have a negative effect on the validity of a mutation score when determining the accuracy of test suites.

Equivalent mutants are mutants that perform the same behaviour on a specific method as other existing mutants. Since the mutation score for a mutation test is calculated based on how many mutants were caught, having unit tests catch equivalent mutants decreases this calculation's accuracy. Mutants that are essentially the same should not both be considered as we are only concerned with different discrepancies in the SUT that are detected.

To find equivalent mutants, we decided to use a method that involves taking a test case that kills a certain group of mutants and breaking down its components as much as possible with more test cases to distinguish the mutants in the group apart from each other. Using this method, we would be able to see if the mutants have any different behaviour apart from each other for that specific test case. If two mutants cannot be distinguished from one another, in theory, this would mean that these two mutants are equivalent, as they both exhibit the same behaviour.

To test this method of detecting equivalent mutants, we added test methods to the DataUtilities test suite, *DataUtilitiesEqualTest*, testing for the function *equal(double[][] a, double[][] b)* and the Range test suite, *RangeEqualsTest*, testing for the function *equals(Object obj)*. Extra tests were created from the test suite to attempt to distinguish between groups of mutants that were either all killed by the same unit test or all survived that unit test. These tests were also created based on the mutants of the selected groups. If the mutations did not exhibit any different behaviour from each other, the mutants were assumed to be the same.

The advantage of this method was its effectiveness when the distinguishing factors of groups of mutations were clear. This made their distinction straightforward and made it easy to spot equivalent mutations. Issues with this method arose when distinguishing factors were not clear as there was no proper way to tell whether or not the equivalent mutations had been caught. Overall, this method was fairly effective.

Advantages and Disadvantages of Mutation Testing

We need mutation testing because it helps a user create effective test data in an interactive manner. The goal is to have a strong test suite which will be able to catch typical faults so we seed bugs to find more bugs. Bugs can be seeded systematically (which are automated mutations), such as the mutations generated by PITEST. PITEST, the automation tool we used in this lab made small changes to the source code and produced a mutation log showing which of the generated mutations survived or which were killed. An advantage of this testing approach is that it is an indicator of the remaining bugs in the system and also that it helps the tester locate weakness in the source code which was never detected in coverage testing. For example, in the expand(Range range, double lowerMargin, double upperMargin) method in Range class, the test suite for that class never actually tested what happens when a null range object is passed. This was brought to our attention because the mutation removed the ParamChecks() call on the range object. The fact that we did not actually have a unit test case which tested for the scenario where the range argument object was null meant that our test suite was not as strong as the coverage report had led us to believe. Thus, as a result of mutation testing, we were able to strengthen our test suite. A disadvantage of mutation testing, specifically automated testing, it is hard to interpret the mutants. For instance, interpreting the ‘changed conditional boundary’ mutant was difficult. However we were able to use the PITEST documentation site to understand the mutants further. Not only that, some of the disadvantages of mutation testing is that some of the surviving mutants that were produced were a little too complex to interpret and kill. For example, the mutant ‘negated conditional’ which survived for the if(allowZeroCrossing) conditional in Range Shift method. It was almost impossible to kill this mutant because it was too complex to understand how to develop a unit test case that tested for that scenario. Thus, mutation testing does have its disadvantages, but its biggest benefit is that it strengthens test suites.

SELENIUM Test Case Design Process

We went about our test cases in a procedural way. We considered the main functionalities the customers will need from the website: www.sportchek.ca in order to perform the major tasks the average customer would use. These functionalities include: adding and checking out, filtering by price, and searching by categories. We considered how many paths the average customer can take to achieve the functionality of the different GUI elements that are involved in these processes. This led us to having multiple test cases for certain scripts where many different paths or input data could be given in order to test that specific component functionality. The assertion checkpoints and unique descriptions of each test case/ test script can be seen below. (Total 8 test cases for 4 team members: 2 per member)

Test Script 1 - FilterPrice

Assertion/Checkpoint: Filter Option Located at the top right corner, checkpoint at this step

This script FilterPrice.side tests the 'Filter' options on the www.sportchek.ca. This script consists of only two test cases. This test case HighToLowInStockMensJacket uses the link Mens -> Winter Jackets, landing page to use the filter feature located on the top right corner. The test sets the filter to re-order the page from prices, high to low. This test case passes as the page was properly viewed and there are no other possible paths to this GUI page. The second test case RatingHighToLow uses the Mens -> Winter Jackets, landing page to use the filter feature located on the top right corner. The test sets the filter on the page to Rating High to Low, which reorders the page of winter jackets from the highest-rated jackets to the lowest rated jackets. This test case passes as the page was properly viewed and there are no other possible paths to this GUI page.

Test Script 2 - SelectionFilter

Assertion/Checkpoint: Filter Selection Option Located at the left hand side of the page, checkpoint at this step [At each box]

This script SelectionFilter.side tests the 'Customization Filter' options on the www.sportchek.ca. This script consists of only two test cases. This test case FilterMenPumaRed uses the link Jersey's and Fan Wear -> European Club Soccer, landing page to use the selection filter features located left side of the page. The test checks the Mens box, the Puma box and the color red box to filter the clothing with these parameters. This test case passes as the page was properly viewed and there are no other possible paths to this GUI page. The second test case FilterMenPumaRed uses the link Jersey's and Fan Wear -> European Club Soccer, landing page to use the selection filter features located left side of the page. The test checks the Mens box, the Adidas box and the XL size box to filter the clothing with these parameters. This test case passes as the page was properly viewed and there are no other possible paths to this GUI page.

Test Script 3 - Search
Assertion/Checkpoint: Filter Selection Option Located at the left hand side of the page, checkpoint at this step [At each box]

This script Search.side tests the 'Search' options on the www.sportchek.ca website. This script consists of only a single test case. This test case is searchTennis. This test uses the search bar feature by searching for a "Tennis Racket" (typed in). A user may now search for products that are related to the typed keyboard and this test case passes as it relates products (tennis rackets) as searched for by the user.This is the only possible path to reach this GUI element and it has been determined to have past and functions perfectly.

Test Script 4 - SwitchPages
Assertion/Checkpoint: Page change options located at the bottom

This script SwitchPages.side tests the 'Page number' options on the www.sportchek.ca website. This script consists of only a single test case. This test case is SwitchProductPages. After clicking the electronics -> Beats by Dre, user scrolls to bottom to find an arrow button to go the next page, which lists even more Beats by Dre products. This test passes, as it properly views the second page of items listed and allows users to view and add them into the cart if interested. This is the only possible path to reach this GUI element and it has been determined to have past and functions perfectly.

Test Script 5 - LocateShippingMethodsFAQ
Assertion/Checkpoint: Checkpoint located at the Shipping Methods link that's at the gym

This script LocateShippingMethods.side tests the 'Shipping methods' options on the www.sportchek.ca. This script consists of only a single test case. This test case is ClickAndFindShippingMethodsDescription. This test case uses the link at the bottom of the landing page called: "Help & FAQs". Through this link the 'Shipping methods' option can be selected and all the available options can be viewed. This test case passes as the page was properly viewed and there are no other possible paths to this GUI page.

Test Script 6 - FeedbackTest
Assertion/Checkpoint: Checkpoint located at the feedback icon on the right

This script FeedbackTest.side tests the 'Feedback' options on the www.sportchek.ca website. This script consists of only a single test case. This test case is FeedbackSubmit. This test case uses the link at the right edge of the landing page called: "Feedback". Through this link the 'Please submit feedback'' option can be selected and all the available options can be viewed. Here the customer can submit a rating out of 10 as well as written feedback for the service. This test case passes as the page was properly viewed and opened and any valid rating can be selected and any feedback can be written, there are no word limitations. This is the only possible path to reach this GUI element and it has been determined to have past and functions perfectly.

Test Script 7 - StoreLocator
Assertioin/Checkpoint: Checkpoint located top right at the store located

This script StoreLocator.side tests the 'Prefered store' and 'store locator' GUI elements options on the www.sportchek.ca website. This script consists of only a single test case. This test case is SetPreferredStore. This test case uses the store link at the tope right of the landing page called: "Your Store". Through this link the 'Prefered store' option can be selected and our default location in our example was set to Calgary Alberta near the university of calgary. The default store is Westbrook SportChek however in our test case we can see that we have changed it to Chinook SportChek and this functionality performed correctly as the default store on our next load of the page was Chinook SportChek. Since this 'change store' functionality was the only functionality that could be tested on this GUI element, there was no need to write multiple test cases. Therefore this test has passed successfully.

Test Script 8 - VerifyCartAndCheckout
Assertion/Checkpoint: Checkpoint located at the cart icon, locate top right and the checkout button

This script VerifyCartAndCheckout.side tests the view and checkout functionality of the items in the cart options on the www.sportchek.ca. This script consists of only two test cases. The first test case is ViewCartItems. This test case uses the link at the top of the landing page in the shape of the cart. Through this link we are shown all the items that we have placed in the cart to make sure the quantity, size, color etc is just as we have selected. The second test case involves pressing the 'checkout' button in order to test the purchase functionality of the website of all the items in the cart GUI element. Both test cases passed which confirm that these GUI functionalities work properly.

Advantages and Disadvantages of Selenium Vs. Sikulux

Selenium's advantages provide a simple, easy to add (through extensions) IDE interface that allows new users (even new to programming) with a testing tool. This is achieved through its testing of web pages functionality that include not more than clicks or inputs with the keyboard, including search features, filter switches, and more specifically to our websites (such as Sport Chek that we tested in our lab) features like check out and add to cart. In addition, it allows a re-run of your inputs through scripts and is able to convert these inputs into java code that can be later referenced, modified and used.

Selenium's disadvantages also come with its simplicity, as the capability of testing in depthly from a website's interface is limited to what is seen through the html. That being it is only limited to clicks and keyboard inputs and nothing more. Other disadvantages come with the extension and ide, itself. Perhaps due to unstable internet connection or other issues, the IDE may sometimes not register inputs, commands etc which can lead to a tedious and repetitive testing process. For example, when we tried to run the scripts on a different day, the process to run was tedious and slow due to unstable internet connection.

Sikulux's advantages also provide a testing interface for users, that is more testing functionalities. Sikilux also provides image recognition in its testing, allowing for more in depth testing when compared to Selenium. It is also not a chrome extension meaning, it's not limited to internet websites

but may also be used to test different applications located on a user's desktop. Its open source nature allows it to be easily modified for better and more efficient use.

Sikulux's disadvantages comes in the use of it in itself, that is it is susceptible and known to be slow, not work, crash etc. Further using the testing, its image recognition depends greatly on the clarity of the image and may not be able to recognize the same image if the image is somewhat blurry.

Both these tools have great features that uniquely provide benefits of their own to testing, a combination of both tools may lead to the most efficient testing process.

How the Teamwork was Divided/Managed

Teamwork is a fundamental core part of software testing. In order to go through the test we read and did the preliminary parts of the mutation tests as a group (the setups in the initial stages of the lab). Specifically this involved the setups for Pitest and Silenium on each member's computer. After this we proceeded to go through the sample test cases to run mutation tests in part 2.5.4 together as well. This was to ensure that everyone was up to par in the standard, format and overall understanding of mutation testing and being able to read and interpret the mutation report. From there, we would proceed to have multiple group sessions where we worked individually on our own test cases to try to increase the number of mutants being killed. We implemented a similar methodology when we moved onto GUI testing with selenium. Each member was responsible for producing 2 test scripts testing two different functionalities after we all worked on one example test script together to understand the workings of selenium. Surprisingly no one faced any issues with selenium as the IDE was relatively lightweight and was as simple as installing a chrome extension.

A lesson learned throughout the entire process was that having more than one pair of eyes on an issue can provide faster and better solutions. For example, as we were first running through the mutation tests as a group, all of us had input our knowledge and ideas on how to set up the tests and improve upon tests from previous labs in order to catch more mutants. It provided an efficient process into creating a test for some of the sample methods as well as filled any knowledge gaps that we had going into the lab, allowing us to work on our own split individual tests as well. Each person was responsible for writing test cases for at least two methods and documenting their strategy. Furthermore, collaboration on one or two test cases allowed us to formulate the best estimations on how to catch the most amount of mutants through trial and error and input from other team members. That way, when we went to individually work on trying to improve mutation coverages separately, we already knew the main things to target to increase the mutation coverage and did not have to spend a lot of time experimenting by ourselves.

Teamwork is incredibly important in software development as "teamwork makes the dream work." Learning how to work in a team is imperative for engineers across all disciplines because it introduces them to varying new perspectives that only strengthen the final product. One of the key lessons that we learned about teamwork is that open communication, transparency and hard work are the foundations of any successful group.

Difficulties Encountered, Challenges Overcome, and Lessons Learned

There were a few challenges and lessons that came about while performing through the assignment. The first was downloading and implementing the mutation coverage tool Pitest and getting it to work with our individual ide's and computers. In this lab was the first time we observed a software/plugin that caused issues across all group member's computers and caused us to work together and find different debugging solutions for each of our unique errors with Pitest. Next, came the testing part, to which we had discovered that in order for the mutation coverage to work efficiently, we had to modify or possibly delete certain test files from the combined zip folder we submitted for our previous lab. These factors combined caused a significant setback in pacing as we had to fix multiple errors before even beginning the mutation testing. After running our initial tests, we had discovered via the mutation report our percentages that we had covered were not as high as it could be for both classes of DataUtilities and Range(~73% each), and so we had to add new test cases to enhance our previous tests in order to catch even more variations of the input passed in by many mutants. It challenged us to add test cases for test files from both classes and we were able to see certain strategic similarities among methods in terms of specific things to target to kill mutants. As a result we were able to increase our mutation coverage for DataUtilities by 13% and increase mutation coverage for Range class by 10%. In addition we also faced some problems with Selenium IDE, not installation or usage problems but however we found that sometimes selenium was a bit buggy in the playback. There were occasions where certain events were not being played back properly even though they were scripted correctly. However these were rare occurrences and for the most part Selenium worked ok.

In conclusion, overcoming these challenges furthered our understanding of both mutation testing and GUI testing as a whole. It also increased our efficiency in writing tests with our goal being to catch the most amount of mutants as well as capture the major functionalities of any GUI in relation to it's main user base.

Comments/feedback on the lab itself

Overall, the lab detailed a simple yet insightful experience in mutation testing and GUI testing. The experience induced a healthy working environment that explored not only our previous knowledge of the course material but more importantly our ability to cooperate in group settings. We discovered in our first run-through of using mutation testing the application produced minor difficulties across various operating systems which was important to understand and resolve, as all team members took on a set of test cases to complete. However, going through the lab we were able to quickly overcome such difficulties, turning it into a smooth and informative session. An improvement that could be implemented in the lab document itself would be the adding of figures in sections 3.3 (GUI testing). Mutation testing also provided a more thorough insight into the application of these tests in the real world and helped us understand the magnitude of testing (considering we had to implement many extra tests in order to fulfil the required coverage percentages). The GUI testing provided an additional area of testing we had no prior knowledge of, and allowed for a lot of exploratory of the tools and its features. All in all, we found this lab easy to follow and an excellent introduction to mutation and GUI testing in all aspects.