**Course: Software Testing**

**Lab. Report #3 – White-box testing and code coverage**

|  |  |
| --- | --- |
| Group #: | Group 59 |
| Student Names: | Patrick Hamill |
| James Cassidy |

**Table of Contents**

[1 Detailed description of the test plan (strategy) for unit testing 2](#_Toc100856054)

[2 Description of five selected test cases you have designed using coverage information, and how they have increased code coverage 2](#_Toc100856055)

[2.1 testCombineValidRangeOneWithValidRangeTwo () 2](#_Toc100856056)

[2.2 testExpandAValidRangeWithPositiveMargins() 3](#_Toc100856057)

[2.3 testGetCentralValueofPositiveRange() 3](#_Toc100856058)

[2.4 testInvalidRangeInstantiation() 3](#_Toc100856059)

[2.5 testTwoEqualRanges() 3](#_Toc100856060)

[3 Detailed report of the coverage achieved of each class and method 3](#_Toc100856061)

[3.1 Branch Coverage Range Class 3](#_Toc100856062)

[3.2 Line Coverage Range Class 4](#_Toc100856063)

[3.3 Data Utilities Branch Coverage 5](#_Toc100856064)

[3.4 Data Utilities Line Coverage 5](#_Toc100856065)

[4 Manual data-flow coverage calculations for Range.constrain(double) method 6](#_Toc100856066)

[4.1 Calculating the percentage coverage of uses for the test case value > upper bound. Nodes 1, 2, 3, 6 are executed 7](#_Toc100856067)

[4.2 Calculating the percentage coverage of uses for the test case value < lower bound. Nodes 1, 2, 4, 5, 6 are executed 7](#_Toc100856068)

[4.3 Calculating the percentage coverage of uses for the test case value is in the range. Nodes 1, 6 are executed 7](#_Toc100856069)

[5 Comparison on the advantages and disadvantages of requirements-based and coverage-based test generation 7](#_Toc100856070)

[6 A discussion on how the TEAMWORK/effort was divided and managed 8](#_Toc100856071)

[7 Difficulties/ challenges encountered, overcoming them, and lessons learned 8](#_Toc100856072)

[7.1 Difficulties/ challenges encountered 8](#_Toc100856073)

[7.2 How did you overcome the above difficulties/ challenges? 9](#_Toc100856074)

[7.3 Lessons learned 9](#_Toc100856075)

[8 Comments/feedback on the lab and lab document itself 9](#_Toc100856076)

[8.1 Did you find the lab a useful learning experience? How it helped you learn the new testing topics 9](#_Toc100856077)

[8.2 Was the lab document easy to follow? 9](#_Toc100856078)

[8.3 About time budget? (Was there too much/too little time for this lab?) 9](#_Toc100856079)

[8.4 Please provide your comments on how to improve the lab work and lab document 9](#_Toc100856080)

# Detailed description of the test plan (strategy) for unit testing

Previously in Lab 2, we both agreed to take 5 methods each. James devised a test plan for Data Utilities class and Patrick devised a test plan for the Range class. Because of our familiarization with our respective test plans, we agreed to undertake the same classes again for Lab 3. As this was the case, 2 separate test plans were created: James’ Data Utilities Test Plan and Patrick’s Range Test Plan.

**James’ Test Plan**

With completion of black box testing of the system under test (SUT), I can now import my Data Utilities test suite. I can now devise a test plan for my White box testing of Data Utilities. My initial black box test suite can now be enhanced using a variety of white box testing techniques, being able to analyze sections of code that have not be covered by black box testing.

These techniques will include:

* Control-Flow Coverage
* Data-Flow Coverage

Once coverage testing has ran, I will be able to devise test cases to increase line and branch code coverage past the 90% and 70% thresholds respectively.

Branch coverage is a testing method that indicates whether all branches within the codebase are executed by test. The branch is one of the possible executions paths the code can take after a decision statement such as an if or while statement. Utilizing a control-flow graph I can assess any of the branches that have not be executed by my original black box test suite. Increasing branch coverage in turn will also increase line coverage.

**Patrick’s Test Plan**

As I oversaw the black box unit testing for the Range class in Lab 2, it made sense that I would continue my work with the class for Lab 3, aiming to achieve 70% branch coverage and 90% line coverage, an increase of 10.9% from 59.1% and 46.2% from 43.8% respectfully.

Due to the nature of the range class, and only being required to unit test 5 methods in Lab 2, this left the other 10 methods, and majority of the code untested. The simplest way to increase the branch and line coverage to the desired goals would be to test more methods, however, to achieve the desired code coverage, I plan on using the code coverage tool built into Eclipse to first improve my existing suite of tests completed in Lab 2, ensuring that all lines and branches are covered.

Next, using the statement and decision coverage techniques as detailed in the white box testing lectures, I intend to design control flow graphs for 5 additional methods in the range class to boost the coverage area of the code being tested. This would mean that a total of 10 methods from the Range class will be tested, 5 for lab 2 and 5 for lab 3, not including the improvements planned for the 5 methods completed for lab 2.

On Page 15 of the Lab 3 document, it states that “*Note that although the focus in adequacy criteria has changed (it is now on source code), to develop the test cases, the test oracle should always be derived from the requirements (as provided in the Javadocs of the SUT in Lab2).*” I have taken this to mean that the control flow graphs should be designed not from going line by line in the source code as shown in the lectures, but instead going off what is written in the Java doc, similar to the previous lab and using the code coverage tool to ensure that all branches and lines have been covered.

# Description of five selected test cases you have designed using coverage information, and how they have increased code coverage

## testCombineValidRangeOneWithValidRangeTwo ()

This test case tests the combine method on lines 198 to 214 of the range class, combining 2 valid ranges to produce an equally valid range with the lower and upper bounds expanded accordingly. The test case covers 2 of the 4 branches at line 199 and 203 as well as 7 of the 9 lines from 199 to 211. As a result, this test case increases the branch coverage by 4.54% and the line coverage by 9.59%.

## testExpandAValidRangeWithPositiveMargins()

The method being test by this test case is located from lines 253 and 263. It takes a positive range and expands the margins (LB) by 0.25 and (UB) 0.5 respectfully. The method being test was part of the 10 methods previously untested in Lab 2. The method consisted of 2 branches and 7 lines. This particular test case covered 1 of the 2 branches at line 255 and 5 of the 7 lines from line 258 to 262, therefore increasing the branch coverage of the range class by 2.27% and the line coverage by 6.84%

## testGetCentralValueofPositiveRange()

The method being test by this test case is located from lines 126 and 128. It returns the central value of the current range by dividing both bounds by 2 and adding the results. The method contains no branches and only 1 line, therefore it couldn’t increase the branch coverage, but increased the line coverage for the class by 1.37% by reading line 127.

## testInvalidRangeInstantiation()

This test case tests the constructor method on lines 84 to 92. In particular it tests the outcome when trying to instantiate an invalid range, this being a range where the lower bound is greater than the upper bound. The method covers 1 of the 2 branches at line 85 and 5 out of the 8 lines from lines 84 to 88. Therefore, the method increased the total branch coverage by 2.27% and line coverage by 6.84%

## testTwoEqualRanges()

This test case tests the equals method on lines 334 to 346. It takes the current object and tests for equality with an arbitrary object. For the purposes of the test case, I was testing the equality of 2 given ranges. For this particular test case, 3 out of the 6 branches are covered at line 335, 339 and 342 in order and 5 out of the 8 lines are covered from 335 to 345. Therefore, this test case increased the branch coverage by 6.82% and the line coverage by 6.84%

# Detailed report of the coverage achieved of each class and method

## Branch Coverage Range Class

Before: 59.1%Coverage

Table

Description automatically generated with low confidence

After: 95.5% CoverageTable

Description automatically generated

## Line Coverage Range Class

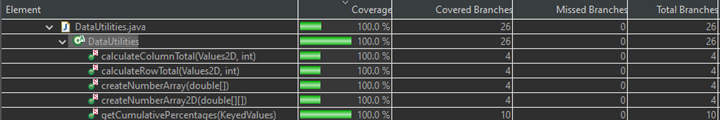
Before: 43.8% CoverageTable

Description automatically generated

After: 93.2%Graphical user interface, table

Description automatically generated

## Data Utilities Branch Coverage



## Data Utilities Line Coverage

Graphical user interface

Description automatically generated

# Manual data-flow coverage calculations for Range.constrain(double) method

Diagram

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| **Nodes** | **Defines** | **c-uses** | **p-uses** |
| 1 | value, result | value | value |
| 2 |  |  | value, this.upper |
| 3 | result | this.upper |  |
| 4 |  |  | value, this.lower |
| 5 | result | this.lower |  |
| 6 |  | result |  |

|  |  |  |
| --- | --- | --- |
| **Node** | **dcu(v, i)** | **dpu(v, i)** |
| 1 | dcu(result, 1) = {6} | dpu(result, 1) = {{2 , 3} , {4, 5}} |
| 2 |  |  |
| 3 | dcu(result, 3) = {6} |  |
| 4 |  |  |
| 5 | dcu(result, 5) = {6} |  |
| 6 |  |  |
|  | Total number of def-clear c-use paths to cover = 3 | Total number of def-clear p-use paths to cover = 2 |

## Calculating the percentage coverage of uses for the test case value > upper bound. Nodes 1, 2, 3, 6 are executed

* Def of result in node 1 and its c-use at node 6 : 1 instance
* Def of value in node 1 and p-use in edge (2,3): 1 instance
* Def of result in node 3 and its c-use at node 6: 1 instance

Therefore, 3 of the 5 definition-clear paths would be covered, yielding an all uses coverage ratio of 60%

## Calculating the percentage coverage of uses for the test case value < lower bound. Nodes 1, 2, 4, 5, 6 are executed

* Def of result in node 1 and its c-use at node 6 : 1 instance
* Def of value in node 1 and p-use in edge (4,5): 1 instance
* Def of result in node 5 and its c-use at node 6: 1 instance

Therefore, 3 of the 5 definition-clear paths would be covered, yielding an all uses coverage ratio of 60%

## Calculating the percentage coverage of uses for the test case value is in the range. Nodes 1, 6 are executed

* Def of result in node 1 and its c-use at node 6 : 1 instance
* Def of result in node 5 and its c-use at node 6: 1 instance

Therefore, 2 of the 5 definition-clear paths would be covered, yielding an all uses coverage ratio of 40%

# Comparison on the advantages and disadvantages of requirements-based and coverage-based test generation

Having worked though both labs thoroughly, we believe both black box and white box testing have advantages and disadvantages associated with each.

**Black Box Testing**

In designing black box tests, a tester does not have access to the source code. While with white box testing, the source code is available and tests may be more comprehensive, it can lead to biased tests by the developer. With black box tests, the tester, and the developer are independent of each other, therefore making tests more balanced, unbiased, and unprejudiced as tests are derived from the end user’s point of view. This can also be known as Acceptance Testing

With a tester unable to access the source code, black box tests are derived from the specifications of the program. In our case with Lab 2, our black box tests were derived from the Javadocs. This can be seen as an advantage as there is no need for a tester to have detailed functional knowledge of the system, and therefore the tester can be non-technical. This allows a tester to easily derive black box tests for various languages such as Java, C++, and C# etc. and be more efficient with their test cases. In black box testing, test cases can also be designed almost immediately after the functional specifications are complete.

However, if functional specifications are not completed to a high standard, it can lead to subpar test cases. It can also be difficult to identify all possible inputs based on the functional specifications alone. As a result, writing test cases can be slow and sometimes difficult.

There is also a high chance that several paths are left unidentified in the testing process It is also impossible to test every possible input in black box testing as it is too time consuming. In our case, once importing our black box test suite into Lab 3 and testing the code coverage, various methods had line and branch coverage below the threshold that was expected.

**White Box Testing**

With white box testing, the source code is available to the tester. This allows for testing to be very extensive as more comprehensive test cases can be derived from the source code that cover all paths. In our case, we were able to see what branches were not covered initially by our black box test suite. Analyzing the code coverage and source code we were then able to derive more test cases that covered these branches, therefore increasing our branch and line coverage threshold. However, certain functionalities of the program could be missed out through white box testing as only the available code is being tested. As we utilized black box testing before this, it is unlikely we missed much functionality but a business that only utilities coverage-based testing may gloss over certain functionally in their test suite.

As emphasis on white box testing is on the source code, tests may become obsolete when frequent changes are made to the codebase. Black box testing does not suffer this drawback as a test suite is derived from the functional specification, in our case it was the Javadocs in Lab 2.

A prerequisite before commencing white box testing is understanding what the source code does. Because of this, it can become easier to find out what data types can be passed through etc. This allows for more efficient testing. This can also be seen as a disadvantage. Testers need in-depth programming knowledge of the system under test will be needed, unlike with black box testing. As a result of this, white box testing is more expensive. This may not be feasible if the system under test is too large to derive white box tests from.

# A discussion on how the TEAMWORK/effort was divided and managed

Following the same approach as we took from Lab 2, James worked again on the Data Utilities class while Patrick worked on the Range class. This was done as we both have familiarized ourselves with our respective classes and therefore found we may be more efficient in deriving a white box test suite from the system under test.

Finishing the familiarization section of Lab 3, we realized that James’ black box test suite already passed the threshold for Branch and Line coverage at 100%. The lab document states that the if the thresholds have already been achieved several other tests could be created to increase the coverage further. This could not be achieved with Data Utilities as it was already at 100%.

As this was the case, James agreed to tackle most of the report and help Patrick in analyzing the Range code coverage and help derive test cases from missed lines and branches from the source code.

# Difficulties/ challenges encountered, overcoming them, and lessons learned

## Difficulties/ challenges encountered

Several challenges were faced when tackling this lab.

With already achieving 100% branch and line coverage for the Data Utilities class, there was little to no advice on how to cover this in the lab document.

In part 4 of the lab document, we found creating a manual data-flow graph challenging for the Range.Constrain method. Even when analyzing Appendix, A, we still found this section difficult.

## How did you overcome the above difficulties/ challenges?

As James was tasked with deriving a white box test suite of Data Utilities, it was agreed that he would tackle most of the report, to lessen the workload of Patrick as he derived more test cases for the Range class. This also left more time for James to assist Patrick if he had any trouble in deriving any test cases for the Range class.

In overcoming the Manual Data Flow Graph for Range.constrain, we began looking through more of Vahid’s lecture videos and notes and various YouTube tutorials. Here we found numerous examples on how to tackle this graph. A cumulation of all these tutorials helped us in completing this task.

## Lessons learned

Using our own initiative, we learned to step out of our comfort zone to tackle by using various sources available to us in tackling white box testing to develop a manual data flow graph.

# Comments/feedback on the lab and lab document itself

## Did you find the lab a useful learning experience? How it helped you learn the new testing topics

We both found Lab 3 to be an extremely useful learning and revision experience. We were able to practice and develop our white box testing and code coverage skills using the built-in code coverage tool in eclipse, using it to ensure our unit test cases derived from our white box testing techniques were accurate and covered all bases.

## Was the lab document easy to follow?

Overall, we found the lab document much easier to follow than the previous lab, due possibly the inclusion of more images and story boards to take us through key topics. In general, the lab document was much shorter as well, ensuring a higher level of focus to complete the work.

## About time budget? (Was there too much/too little time for this lab?)

In terms of the time given to complete this lab, we found the extra time to be an asset as we had other assignments due around the same time, not to mention Lab 2 which was required to be completed first before beginning Lab 3. The budget offered us the ability to prioritize more difficult tasks and enabled us to arrange better suited meetings as to not place each other under any unnecessary stress.

## Please provide your comments on how to improve the lab work and lab document

Overall, the Lab doc was well structured and provided all the necessary information to complete the lab work. We were both happy with the current approach and can’t think of any suggestions that would in anyway improve the quality of the Lab and the Lab document itself. However, a line or two telling students what to do if one or both classes already achieved 100% line and branch coverage would be helpful.