**Course: Software Testing**

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

|  |  |
| --- | --- |
| Group #: |  |
| Student Names: |  |
|  |

**Table of Contents**

[URL of your project in GitHub.com 2](#_Toc159855312)

[1 Introduction 2](#_Toc159855313)

[2 Test plan for white-box unit testing 2](#_Toc159855314)

[2.1 Approach for white-box test-case design 2](#_Toc159855315)

[2.2 Division of work-load (between the two students) for white-box test-case design and development 3](#_Toc159855316)

[3 Description of how you have designed additional test case (test methods) to improve code coverage for methods from the SUT (see the lab doc for details) 3](#_Toc159855317)

[3.1 Summary table of test-suite size from in lab3 compared to lab2 3](#_Toc159855318)

[3.2 Discussing details of design and development of additional test cases 3](#_Toc159855319)

[3.2.1 Range class under test 3](#_Toc159855320)

[3.2.2 DataUtilities class under test 4](#_Toc159855321)

[4 Showing that the coverage threshold is achieved for each class 6](#_Toc159855322)

[4.1 Range class 6](#_Toc159855323)

[4.1.1 Line-coverage screenshot: 6](#_Toc159855324)

[4.1.2 Branch-coverage screenshot: 6](#_Toc159855325)

[4.2 DataUtilities class 6](#_Toc159855326)

[4.2.1 Line-coverage screenshot: 6](#_Toc159855327)

[4.2.2 Branch-coverage screenshot: 6](#_Toc159855328)

[5 Output of test suite execution: Include a screenshot of test suite execution in JUnit showing their Pass/Fail/Error status, and the top-bar numbers 6](#_Toc159855329)

[6 Comparison on the advantages and disadvantages of requirements-based and coverage-based test generation 7](#_Toc159855330)

[6.1 Advantages of requirements-based test generation (black-box testing) 7](#_Toc159855331)

[6.2 Disadvantages of requirements-based test generation (black-box testing) 7](#_Toc159855332)

[6.3 Advantages of coverage-based test generation (white-box testing) 7](#_Toc159855333)

[6.4 Disadvantages of coverage-based test generation (white-box testing) 7](#_Toc159855334)

[7 Manual data-flow coverage calculations for Range.constrain(double) method 7](#_Toc159855335)

[7.1 Step 1 -Identifying code “blocks” in a tabular form 7](#_Toc159855336)

[7.2 Step 2 -Designing the CFG using code block numbers 7](#_Toc159855337)

[7.3 Step 3 - Identifying the variables’ def / use from the CFG, in a tabular form 7](#_Toc159855338)

[7.4 Step 4- Identifying the definition-clear-use paths, in a tabular form 7](#_Toc159855339)

[7.5 Step 5- Calculating data-flow coverage ratios (percentage values) for a given test case or test suite, on all the definition-clear paths 7](#_Toc159855340)

[8 Manual mutation testing 8](#_Toc159855341)

[8.1 Mutation 1 8](#_Toc159855342)

[8.1.1 Mutation Operator used 8](#_Toc159855343)

[8.1.2 Provide the full code of the mutant method under test, and highlight the mutated line of code 8](#_Toc159855344)

[8.1.3 Results of running latest test suite and whether you need to add a test case (include the table) 8](#_Toc159855345)

[8.2 Mutation 2 8](#_Toc159855346)

[8.2.1 Mutation Operator used 8](#_Toc159855347)

[8.2.2 Provide the full code of the mutant method under test, and highlight the mutated line of code 8](#_Toc159855348)

[8.2.3 Results of running latest test suite and whether you need to add a test case (include the table) 8](#_Toc159855349)

[8.3 Mutation 3 8](#_Toc159855350)

[8.3.1 Mutation Operator used 8](#_Toc159855351)

[8.3.2 Provide the full code of the mutant method under test, and highlight the mutated line of code 8](#_Toc159855352)

[8.3.3 Results of running latest test suite and whether you need to add a test case (include the table) 8](#_Toc159855353)

[8.4 Mutation 4 8](#_Toc159855354)

[8.4.1 Mutation Operator used 8](#_Toc159855355)

[8.4.2 Provide the full code of the mutant method under test, and highlight the mutated line of code 8](#_Toc159855356)

[8.4.3 Results of running latest test suite and whether you need to add a test case (include the table) 8](#_Toc159855357)

[8.5 Mutation 5 9](#_Toc159855358)

[8.5.1 Mutation Operator used 9](#_Toc159855359)

[8.5.2 Provide the full code of the mutant method under test, and highlight the mutated line of code 9](#_Toc159855360)

[8.5.3 Results of running latest test suite and whether you need to add a test case (include the table) 9](#_Toc159855361)

[8.6 Summary table of all mutants 9](#_Toc159855362)

[9 Team work 9](#_Toc159855363)

[9.1 How the team work/effort of the lab was managed and divided 9](#_Toc159855364)

[9.2 Writing the lab report 9](#_Toc159855365)

[9.3 Lessons learned from your teamwork in this lab 9](#_Toc159855366)

[10 Difficulties/ challenges encountered, overcoming them, and lessons learned 9](#_Toc159855367)

[10.1 Difficulties/ challenges encountered 9](#_Toc159855368)

[10.2 How did you overcome the above difficulties/ challenges? 9](#_Toc159855369)

[10.3 “Technical” Lessons learned 10](#_Toc159855370)

[11 Comments/feedback on the lab and lab document itself 10](#_Toc159855371)

[11.1 About time budget? (Was there too much/too little time for this lab?) 10](#_Toc159855372)

[11.2 Was the lab document easy to follow? 10](#_Toc159855373)

[11.3 Please provide your comments on how to improve the lab work and lab document 10](#_Toc159855374)

**URL of your project in GitHub.com**

[www.github.com/…](http://www.github.com/…)

**General advice for writing high-quality lab reports:**

* Ensure following technical writing advice and resources on the internet. Do a Google search for:
  + <https://www.google.com/search?q=technical+writing+for+software+engineers>
  + And <https://www.google.com/search?q=technical+writing+for+students>
* Avoid common mistakes of technical writing ([google.com/search?q=technical+writing+mistakes](https://www.google.com/search?q=technical+writing+mistakes)), such as very long paragraphs, etc.

# URL of your project in GitHub.com

www.github.com/…

# Introduction

# Test plan for white-box unit testing

## Approach for white-box test-case design

Reminder:

* Test-case design means designing test cases “on paper”, using the techniques learned in lectures and in tabular form, BEFORE coding them in Java JUnit
* Test-case development means using the designed test-cases from the previous step to test methods in Java JUnit (NOTE: a test case shall be developed as one test method in JUnit)

## Division of work-load (between the two students) for white-box test-case design and development

# Description of how you have designed additional test case (test methods) to improve code coverage for methods from the SUT (see the lab doc for details)

## Summary table of test-suite size from in lab3 compared to lab2

Find more details in the lab doc

## Discussing details of design and development of additional test cases, for two randomly-chosen methods under test from each of the two classes under test

### Range class under test

For the Range class, even if you have developed tests for many of its methods, just choose randomly, as examples, two of its methods under test and fill out the following info (tables) for each of them. Do the same for the DataUtilities class.

#### Method under test 1: method\_name

|  |  |  |
| --- | --- | --- |
| BEFORE making the coverage improvement | Screenshot of code coverage of the method (BEFORE making the coverage improvement) - **Line coverage** |  |
| Screenshot of code coverage of the method (BEFORE making the coverage improvement) - **Branch coverage** |  |
| AFTER making the coverage improvement | Screenshot of code coverage of the method (AFTER making the coverage improvement) - **Line coverage** |  |
| Screenshot of code coverage of the method (AFTER making the coverage improvement) - **Branch coverage** |  |
|  | Discussions of how you designed the new test case(s) to increase code coverage for this method. Hint: Recall from lectures that, to increase code coverage, we analyze the code coverage of a method, and find out which parts (lines or branches) are not yet covered (tested), and determine the input(s) to the method which would cover those uncovered parts.  Then, we design and develop new additional test cases (methods) to actually cover those uncovered parts. |  |
|  | Screenshot of test-code of the new test cases, developed to increase coverage: |  |

#### Method under test 2: method\_name

|  |  |
| --- | --- |
| Screenshot of code coverage of the method (BEFORE making the coverage improvement) -- IMPORTANT: MAKE FORMATTING CHANGES (ADD NEW COLUMNS / ROWS TO THIS TABLE) LIKE THE ABOVE DFIRST METHOD TO HAVE BOTH LINE AND DECISION COVERAGE SCREENSHOTS |  |
| Screenshot of code coverage of the method (AFTER making the coverage improvement) |  |
| Discussions of how you designed the new test case(s) to increase code coverage for this method. Hint: Recall from lectures that, to increase code coverage, we analyze the code coverage of a method, and find out which parts (lines or branches) are not yet covered (tested), and determine the input(s) to the method which would cover those uncovered parts.  Then, we design and develop new additional test cases (methods) to actually cover those uncovered parts. |  |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

### DataUtilities class under test

#### Method under test 1: method\_name

|  |  |
| --- | --- |
| Screenshot of code coverage of the method (BEFORE making the coverage improvement) |  |
| Screenshot of code coverage of the method (AFTER making the coverage improvement) |  |
| Discussions of how you designed the new test case(s) to increase code coverage for this method. Hint: Recall from lectures that, to increase code coverage, we analyze the code coverage of a method, and find out which parts (lines or branches) are not yet covered (tested), and determine the input(s) to the method which would cover those uncovered parts.  Then, we design and develop new additional test cases (methods) to actually cover those uncovered parts. |  |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

#### Method under test 2: method\_name

|  |  |
| --- | --- |
| Screenshot of code coverage of the method (BEFORE making the coverage improvement) |  |
| Screenshot of code coverage of the method (AFTER making the coverage improvement) |  |
| Discussions of how you designed the new test case(s) to increase code coverage for this method. Hint: Recall from lectures that, to increase code coverage, we analyze the code coverage of a method, and find out which parts (lines or branches) are not yet covered (tested), and determine the input(s) to the method which would cover those uncovered parts.  Then, we design and develop new additional test cases (methods) to actually cover those uncovered parts. |  |
| Screenshot of test-code of the new test cases, developed to increase coverage: |  |

# Showing that the coverage threshold is achieved for each class

For this, two screenshots, one for Range, and one for DataUtilities) from the code coverage results showing the coverage “bars” with % values are needed, such as:

A screenshot of a computer

Description automatically generated

Note: the levels should clearly be above the given thresholds.

## Range class

### Line-coverage screenshot:

### Branch-coverage screenshot:

## DataUtilities class

### Line-coverage screenshot:

### Branch-coverage screenshot:

# Output of test suite execution: Include a screenshot of test suite execution in JUnit showing their Pass/Fail/Error status, and the top-bar numbers

Include a screenshot of test suite execution in JUnit showing their Pass/Fail/Error status, and the top-bar numbers, such as:

A screenshot of a computer

Description automatically generated

**(Note: The above is just an example. We are NOT providing the number of test cases for you.)**

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

**Using your examples and experiences learned in labs 2 and 3**

## Advantages of requirements-based test generation (black-box testing)

## Disadvantages of requirements-based test generation (black-box testing)

## Advantages of coverage-based test generation (white-box testing)

## Disadvantages of coverage-based test generation (white-box testing)

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

## Step 1 -Identifying code “blocks” in a tabular form

## Step 2 -Designing the CFG using code block numbers

## Step 3 - Identifying the variables’ def / use from the CFG, in a tabular form

## Step 4- Identifying the definition-clear-use paths, in a tabular form

## Step 5- Calculating data-flow coverage ratios (percentage values) for a given test case or test suite, on all the definition-clear paths

# Manual mutation testing

## Mutation 1

### Mutation Operator used

### Provide the full code of the mutant method under test, and highlight the mutated line of code

### Results of running latest test suite and whether you need to add a test case (include the table)

**Status of running each test case on Mutant:**

|  |
| --- |
| Include the table shown in the lab report |

## Mutation 2

### Mutation Operator used

### Provide the full code of the mutant method under test, and highlight the mutated line of code

### Results of running latest test suite and whether you need to add a test case (include the table)

**Status of running each test case on Mutant:**

|  |
| --- |
| Include the table shown in the lab report |

## Mutation 3

### Mutation Operator used

### Provide the full code of the mutant method under test, and highlight the mutated line of code

### Results of running latest test suite and whether you need to add a test case (include the table)

**Status of running each test case on Mutant:**

|  |
| --- |
| Include the table shown in the lab report |

## Mutation 4

### Mutation Operator used

### Provide the full code of the mutant method under test, and highlight the mutated line of code

### Results of running latest test suite and whether you need to add a test case (include the table)

**Status of running each test case on Mutant:**

|  |
| --- |
| Include the table shown in the lab report |

## Mutation 5

### Mutation Operator used

### Provide the full code of the mutant method under test, and highlight the mutated line of code

### Results of running latest test suite and whether you need to add a test case (include the table)

**Status of running each test case on Mutant:**

|  |
| --- |
| Include the table shown in the lab report |

## Summary table of all mutants

**Summary table of all mutants**

|  |
| --- |
| Include the table shown in the lab report |

# Team work

## How the team work/effort of the lab was managed and divided

* You can say for example discuss which parts of the lab-work (e.g., classes under test, etc.) was done by who…
* And also discuss the meetings that you had to plan and run the lab work
* Etc.

## Writing the lab report

Fill up the following table to specify who wrote what part of the lab document:

|  |  |
| --- | --- |
| **Lab-report section** | **Written by** |
| 1- Introduction | Student A |
| 2-.. |  |
| … |  |

## Lessons learned from your teamwork in this lab

Only include lessons learned from **your teamwork in this section**. **“Technical”** lessons learned **shall be discussed in another section below.**

# Technical difficulties/ challenges encountered, overcoming them, and lessons learned

## Technical difficulties/ challenges encountered

Text…

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

Text…

## “Technical” Lessons learned

Only include **“technical”** lessons learned from **in this section**. Lessons learned **your teamwork shall be discussed in another section above.**

# Comments/feedback on the lab and lab document itself

This section has the following sub-sections.

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

Text…

## Was the lab document easy to follow?

Text…

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

Text…