SWE2721 Lab 7: Code Coverage Testing with the Word Changer

1. Introduction

The tests you have developed thus far have predominantly been Blackbox in nature. While you had the source code available to you, the test cases were developed mainly based on the requirements and the input domain. This exercise will change that in that your tests will be developed based upon the source code, good or bad.

To be successful in this lab, you will need to make a slight change to your IntelliJ setup. By default, the IntelliJ code coverage tool does not trance execution but samples which lines have been reached. In order to track branch coverage, we need to enable tracing. Tracing will internally keep track of which execution paths have been followed during testing.

2. Lab Objectives

- Compare the results of Input Domain Analysis for generating test cases with Control Flow Graphs
- Use graph theory to develop control flow graphs for segments of source code
- Develop TestNG test cases to verify program operation.
- Verify code coverage using a code coverage tool.

3. Prelab

Prior to coming to lab, you will want to watch a brief video which explains how to use code coverage with IntelliJ. This video is linked in the Canvas course.

4. Deliverables / Submission

Now that you have completed your lab assignment, tag your final version (which should be on the trunk) with the label LabSubmission1.0. Your code test cases should be thoroughly commented and build cleanly without warnings. The CI system should also show a clean execution of your tests. Your repository also should contain a directory with your code coverage html report.

In addition, submit one report in **pdf format**. The report should contain the following contents:

1. Title Page
    a. Name
    b. Date
    c. Assignment Title
    d. Repository Link
2. Introduction
    a. What are you trying to accomplish with this lab?
    b. This section shall be written IN YOUR OWN WORDS. DO NOT copy directly from the assignment.
3. Input Domain Analysis
    a. Submit your completed input domain analysis table for the class
4. Control flow graphs
    a. Include a control flow graph for each method. If hand drawn, take a picture with a camera and paste them into your document or scan them with a scanner. However, be sure they are legible in

the final pdf file. Alternatively, if you would like to draw them electronically, you may use a tool such as draw.io (https://www.draw.io/), Visio, or Google Drawings.

   b. Include a screen shot of the graph generated by the Graph Coverage Web application for each method. Try to rotate the graph to maximize legibility. Try to keep the two images together.
   c. Note in text any massive differences that you might see – there should not be many.

5. Table
   a. Include your table showing for each method being tested the method name, the paths you identified which meet cyclomatic complexity criteria for testing the method, and the paths identified with the Graph Coverage Web application for prime path coverage using either algorithm 1 or algorithm 2, which ever you feel is better to compare with the tests you developed yourself
6. Code coverage snapshots
   a. Include a screen capture of the module showing branch coverage for the given methods as you execute your TestNG tests. To do this, execute each testNG test independently and verify that only the method you are looking at is reached and that all branches are covered. If this is true, the code segments for the method you are testing will be all green, but all other methods will be all pink. Then repeat this with the next method. You should have n code coverage snapshots, where n is the number of methods.
7. Discussion
   a. Can you identify any segments of code you might have missed had you used input domain analysis to construct your tests?
   b. Were there any paths you listed based upon your cyclomatic complexity analysis which you could not write a test for?
   c. Were there any prime paths that were generated you could not write a test for or revise the path to make a test for?
   d. What is the main difference between the approach using cyclomatic complexity and the approach using prime paths in terms of how loops are handled?
   e. Were there any statements or branches that were infeasible to test based upon the code structure? Why was this the case?
   f. Without making the changes to the code, could you change the code structure so that the code would be reachable and testable?
   g. Given you your experience with Input Domain Analysis and this method, which approach feels easier to do and why?
8. Things gone right / Things gone wrong
   a. This section shall discuss the things which went correctly with this experiment as well as the things which posed problems during this lab.
9. Conclusions
   a. What have you learned with this experience?
   b. What improvements can be made in this experience in the future?
10. Appendix:
   a. Include the final html file generated showing branch coverage and your source code. You should be able to simply import this into your word document before you save it as a pdf.

This material should be submitted as a single **pdf file** in Canvas. If you have any questions, consult your instructor.

5. Program Background

The project that you are testing is a program which will automatically replace commonly used words with "better" words which are synonyms. The program prompts the user to enter a message without punctuation and then replaces various words with "better words" prior to printing both the old and new messages out to the console. For example:

```
Enter the phase that you would like to have words changed in.  The phase
should not contain any punctuation.
it is apparent that I had a terrible day yesterday I wanted some comic
relief but instead received cheesy breadsticks which I went to the pizza
parlor  I ordered french bread but instead was given cheesy breadsticks
oh well at least the encounter allowed me to grab some mild napkin humor

it is apparent that I had a terrible day yesterday I wanted some comic
relief but instead received cheesy breadsticks which I went to the pizza
parlor I ordered french bread but instead was given cheesy breadsticks
oh well at least the encounter allowed me to grab some mild napkin humor

it is obvious that I had a terrible day yesterday I wanted some comedian
relief but instead received corny breadsticks which I went to the pizza
parlor I ordered vinaigrette bread but instead was given corny
breadsticks oh well at least the come across allowed me to seize some
gentle serviette humor
```

The class which you are to test is the WordChanger class within the project. To work with the project, first clone the project to your local machine. Then import the code into IntelliJ as you have done previously. You have been given a very, very basic test file that you will need to expand upon.

Once you have done this, run the sample tests with code coverage by selecting the run item as is shown below on Figure 1. This will cause the tests to be run with basic coverage enabled. Next, select the edit configurations menu item and change the coverage to enable branch coverage and test tracing. as is shown in Figure 2.

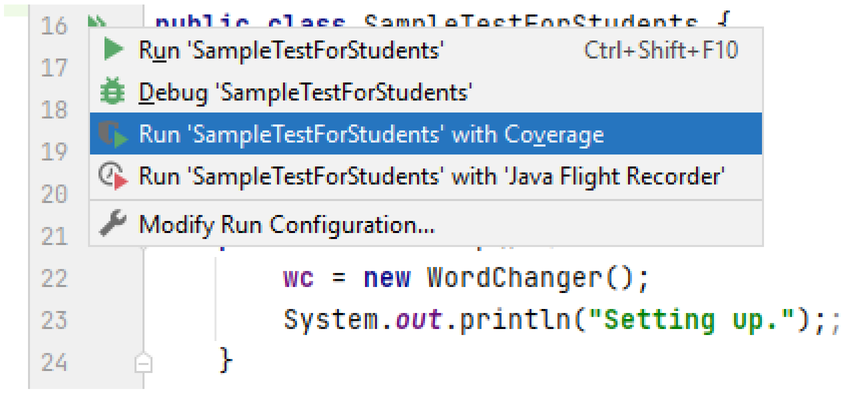
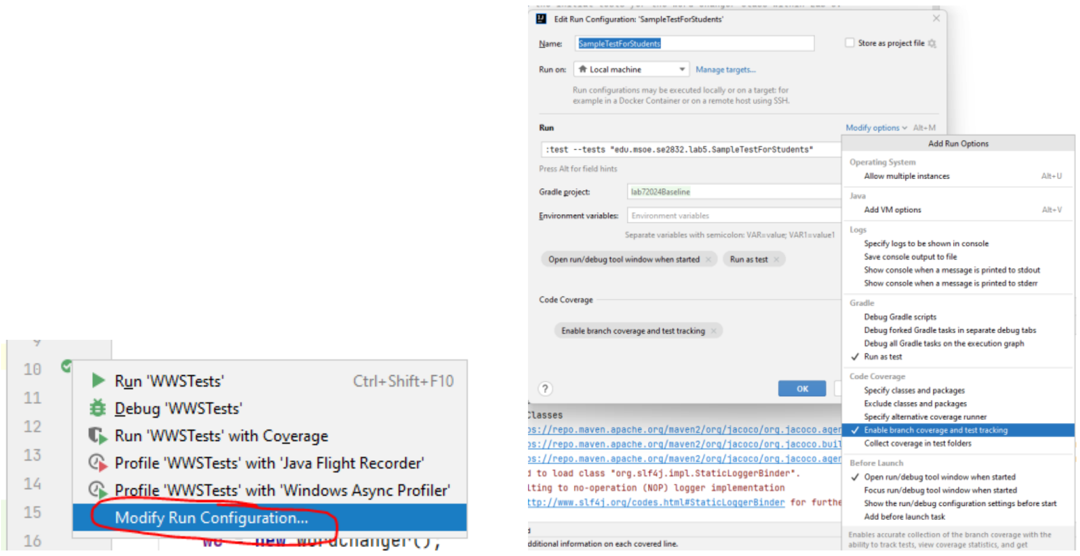

```
public class SampleTestForStudents {
        wc = new WordChanger();
        System.out.println("Setting up.");;
    }
```
Run 'SampleTestForStudents'
Ctrl+Shift+F10
Debug 'SampleTestForStudents'
Run 'SampleTestForStudents' with Coverage
Run 'SampleTestForStudents' with 'Java Flight Recorder'
Modify Run Configuration...

Figure 1: Running the tests with coverage.


Run 'WWSTests' Ctrl+Shift+F10
Debug 'WWSTests'
Run 'WWSTests' with Coverage
Profile 'WWSTests' with 'Java Flight Recorder'
Profile 'WWSTests' with 'Windows Async Profiler'
Modify Run Configuration...
Add Run Options
Operating System
Allow multiple instances Alt+U
Java
Add VM options Alt+V
Logs
Specify logs to be shown in console
Save console output to file
Show console when a message is printed to stdout
Show console when a message is printed to stderr
Gradle
Debug Gradle scripts
Debug forked Gradle tasks in separate debug tabs
Debug all Gradle tasks on the execution graph
✓ Run as test
Code Coverage
Specify classes and packages
Exclude classes and packages
Specify alternative coverage runner
✓ Enable branch coverage and test tracking
Collect coverage in test folders
Before Launch
✓ Open run/debug tool window when started
Focus run/debug tool window when started
Show the run/debug configuration settings before start
Add before launch task
OK

Edit Run Configuration: 'SampleTestForStudents'

Name: SampleTestForStudents
Store as project file
Run on: Local machine
Manage targets...
Run configurations may be executed locally or on a target: for example in a Docker Container or on a remote host using SSH.

Run
Modify options Alt+M

```
:test --tests "edu.msoe.se2832.lab5.SampleTestForStudents"
```

Press Alt for field hints
Gradle project: lab72024Baseline
Environment variables:
Separate variables with semicolon: VAR=value; VAR1=value1
Open run/debug tool window when started
Run as test

Code Coverage
Enable branch coverage and test tracking

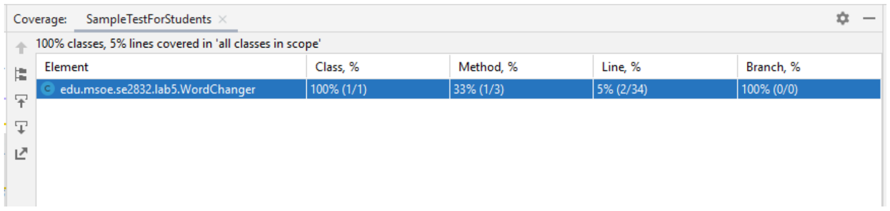
Enables accurate collection of the branch coverage with the ability to track tests, view coverage statistics, and get

```
Classes
ps://repo.maven.apache.org/maven2/org/jacoco/org.jacoco.agen
ps://repo.maven.apache.org/maven2/org/jacoco/org.jacoco.buil
ps://repo.maven.apache.org/maven2/org/jacoco/org.jacoco.agen
d to load class "org.slf4j.impl.StaticLoggerBinder".
lting to no-operation (NOP) logger implementation
ttp://www.slf4j.org/codes.html#StaticLoggerBinder for furthe
```

dditional information on each covered line.

Figure 2: Editing configurations

Now, when you run the tests, you will not only get basic code coverage, but also branch coverage, as is shown in Figure 3.



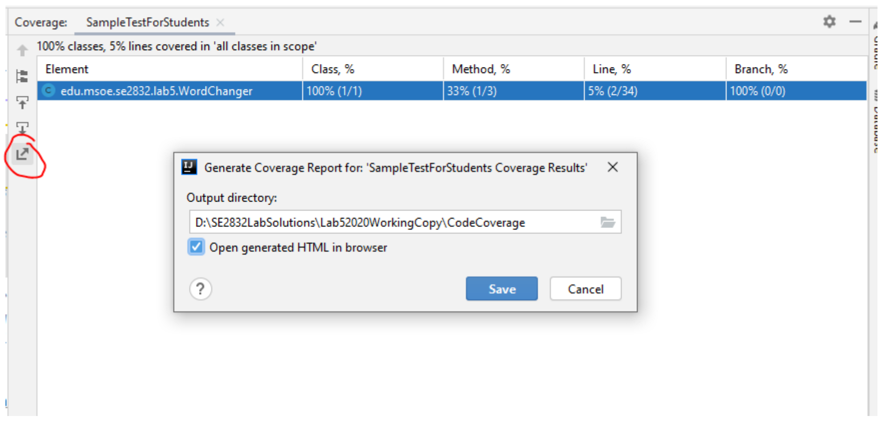
Coverage: SampleTestForStudents

100% classes, 5% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %	Branch, %
edu.msoe.se2832.lab5.WordChanger	100% (1/1)	33% (1/3)	5% (2/34)	100% (0/0)

Figure 3: Branch coverage display.

After setting this up, click on the upward slanting arrow icon, as is shown in Figure 5, and create an output directory where you would like to generate an html report and click on the open generated HTML in browser checkbox. This will automatically create a code coverage report when the tool is executed. At the very end of the project, as one of the last steps, you will want to add this folder to version control.


Coverage:
SampleTestForStudents
Generate Coverage Report for: 'SampleTestForStudents Coverage Results'
Save
Cancel

100% classes, 5% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %	Branch, %
edu.msoe.se2832.lab5.WordChanger	100% (1/1)	33% (1/3)	5% (2/34)	100% (0/0)

Output directory:
D:\SE2832LabSolutions\Lab52020WorkingCopy\CodeCoverage

☑ Open generated HTML in browser

Figure 4: Generating a code coverage report.

6. Step 1: Input Domain Analysis

As you did in previous labs, you want to start by performing an input domain analysis on the class which is to be tested (i.e. the WordChanger class). This should be simple, as this is not a tremendously complex class. You will not go so far as to develop test cases from the Input Domain Analysis, but you want to do this to get a feeling for the class.

Overall, there are four testable methods within the WordChanger class. It is recommended that each partner handle the Input domain analysis for two methods.

7. Step 2: Creating Control Flow Graphs

For each method within the WordChanger class, draw a control flow graph. Label the nodes with the line numbers from the source code, clearly delineating the initial and final nodes. The graphs can be hand sketched, just be as neat as is possible. Try to avoid looping back over, when possible, though some overlap may be necessary. Since you will be working with a partner, each of you should select two methods to test. Try to be equitable – two of the methods are much more complex than the other two.

Once you have sketched these graphs, calculate the cyclomatic complexity for the given method. You'll need to adjust your formula to consider methods that either are or are not SESE in structure.

From this, identify MCC[1] independent paths through the methods and jot them down, noting the nodes which are executed in each path.

Once you have entered these, enter the graph into the tool at https://cs.gmu.edu:8443/offutt/coverage/GraphCoverage.

Use this tool to determine the test requirements for prime path testing and record the prime paths generated with either algorithm 1 or algorithm 2, whichever you feel is closer to the tests you developed using cyclomatic complexity analysis. In addition, spot check the graph to verify it matches what you have sketched.

8. Step 3 Developing your test cases

When you have completed the definition of the prime path tests, create a data provider to prove test data to the given method. As a comment, copy the test paths into the data provider. Then, if possible, define an input and expected value that would cause the given test path to execute. Remember that you may have some side trips needed to cover certain paths, and there may be infeasible test paths that have been generated. If there is a prime path that absolutely cannot be covered by a test case, indicate that the given path is infeasible and clearly explain why testing that path is infeasible.

In addition, identify the paths that are part of your analysis using cyclomatic complexity. Many of the paths should be present in both methods.

[1] MCC is often an abbreviation for Cyclomatic complexity, as is CC.

Each of your individual tests should be in two groups. All tests will be in the ‘all’ group. The second group depends on the method name. For example, constructor tests should be in the group ‘constructor’ while addSynonymPair method tests should be in the ‘addSynonymPair’ group. Each method should be in its own group, using the method for the group name.

Once you have entered your tests into testing, execute them and measure the branch coverage. If the branch coverage is insufficient, determine which branches have not been covered and then go back to the control flow graphs and prime paths to determine why that branch was not covered or if it can be covered. In doing this, indicate what you have changed in the comments for the values passed into the method by the data provider.

When finished, commit your final versions in github.

In the comments section, be certain to indicate which lab partner is developing the given test cases. If it is easier for you to manage, you may each work in a separate file to avoid merge conflicts, though there should not be huge problems with merging given the scope of what you will be doing.