SENG3320/SENG6320 Assignment 1 - Instructions (Semester 1 2019)

Weighting: 25%
Due: 23:59 05/05/2019

9th April 2019

1 Introduction

This assignment will assess your ability to perform black box and white box testing, focusing on content from the first 3 weeks of the course.

The testing will be performed on code which implements simple interpretations of two types of ballot processes for the election of a single representative.

The following source materials have been included in a zip file for this assignment (the paths given below are relative to the root folder of the zip):

- Java code, which can be found in the folder *src/seng3320/election*.
- Javadoc documentation describing the code and intended behaviour. This can be found in the folder *documentation/javadoc* and browsed by opening *documentation/javadoc/index.html* in a web browser.
- This document, providing a description of, and instructions for completing the assignment. This can be found at the path *documentation/instructions.pdf*.
- A marking guide, which describes how marks will be allocated for various tasks in the assignment. This can be found at the path *documentation/marking-guide.pdf*.
- Sample inputs, which may be sent to the main program's standard input, in order to gain familiarity with the program and code. The input samples can be found in the *resources* folder. For instructions on how to use these samples, refer to the Javadoc documentation for the main method, *Main.main(String[])*.

Please refer to Section 3.3 for instructions on what to submit and how.

2 Tasks

2.1 Control Flow Graphs

Draw a Control Flow Graph for each of the following methods:

- *FirstPastThePostElection.Result.toString()*
- *FirstPastThePostElection.count(T...)*
- *PreferentialElection.isFormal(T)*
- *Main.main(String[] args)*

2.2 Black Box Testing

Carry out Black Box Testing on the following methods:

- *FirstPastThePostElection.Result.equals(Object)*
- *FirstPastThePostElection.isFormal(T)*
- *FirstPastThePostElection.count(T...)*
- *PreferentialElection.Result.equals(Object)*
- *PreferentialElection.isFormal(T)*
- *PreferentialElection.count(T...)*

For each method:

1. Specify and describe the boundary values for the method. If there are no boundaries, mention this instead.
2. Specify and describe the equivalence classes that arise from these boundaries (if any).
3. Specify and describe the equivalence partitions for the method.
4. Write JUnit tests to test your partitions and classes. You should use the mimimum number of tests required to test each partition or class.

For each of your tests:

1. Clearly specify which of partitions/classes is being tested.
2. Clearly specify the inputs.
3. Clearly specify the expected/correct outputs.
4. Report on whether or not the code passes the test. If the test fails, provide an explanation of the bug(s) which caused the failure.

Note: You are not required to fix any bugs you find.

2.3 White Box Testing

Carry out white box testing on the following methods, under the specificed branch coverage scheme:

- *PreferentialElection.count(T...)*. (Use decision coverage).
- *FirstPastThePostElection.count(T...)*. (Use condition coverage).
- *PreferentialElection.Result.equals(Object)*. (Use decision/condition coverage).
- *PreferentialElection.isFormal(T)*. (Use modified decision/condition coverage).

For each method:

1. Specify the number of statements.
2. Specify the number of branches (under the respective scheme).
3. Specify the number of paths (under the respective scheme).
4. Write a collection of JUnit tests which achieve maximal statement, branch, and path coverage. You may reuse tests for multiple types of coverage. For each type, you should use the minimum number of tests for the desired coverage.

For each type of coverage (statement, branch, path):

1. Specify as the proportion of coverage you achieved (as a percentage or exact fraction).
2. Specify explicitly which of your tests were used to achieve that coverage.
3. Specify whether or not it is possible to achieve full coverage of that type.
4. Specify the maximum possible test coverage, and explain why this is the case.

For each of your tests:

1. Clearly specify the inputs.
2. Clearly specify the expected/correct outputs.
3. Report on whether or the code passes the test. If the test fails, provide an explanation of the bug(s) which caused the failure.

Note: You are not required to fix any bugs you find.

3 Detailed Instructions

3.1 JUnit Testing and Reporting

For this assignment, you must use JUnit 5 (https://junit.org/junit5/) to create and execute your tests.

Use a coverage testing tool to identify the coverage of your tests, and specify which tool you used. Available tools include:

- IntelliJ Idea's built-in tool (https://www.jetbrains.com/help/idea/code-coverage.html)
- EMMA (http://emma.sourceforge.net/)
- JaCoCo (https://www.eclemma.org/jacoco/)

When documenting a test, you should include the Java identifier for each test method in the relevent section of your report.

When documenting a method and coverage type, you should include the Java identifier for the tests in the relevant section of your report. You may either:

- List the identifiers of every individual test method used, or
- If the tests for that coverage of that method are all neatly contained in one class and file, you may specify the identifier for that containing class.

When specifying the inputs and expected outputs for a given test, you may either describe them in your report, or identify them in-code with a javadoc-style comment. For an example of how to do this in-code, refer to Figure 1.

3.2 Control Flow Graphs

You may draw the graphs either by hand (and, if neccessary, across multiple sheets), or using digital tools such as Microsoft Visio or Draw.io (https://www.draw.io/). However, if you draw by hand it must be photographed, and the photograph must be suitably legible (consult with your tutor).

To facilitate marking, follow these instructions when drawing the graphs:

1. Every graph *must* include exactly one start node (two concentric unfilled circles), a one end node (a single filled circle). Any node that ends a path should therefore connect towards the end node. If these are absent the graph will be considered incomplete.
2. The edges between nodes *must* have an outgoing arrow close to the destination node. Edge labels should uniquely identify the branch. E.g. if labels could be true/false/T/F, or yes/no/Y/N, and switch labels may take text from the case-label in the code.
3. In the label of each node, please use the format *<start#>-<end#>: <id>* where: *start#* is the line where the block starts, *end#* is the line where the block ends, and *id* is a **short** identifier for the node (1-2 words).
4. Non-decision nodes should have rounded edges or corners.
5. Decision nodes should ideally be diamond shaped, but sharp corners and edges will suffice.

For an example of how to draw these graphs, refer to Figures 2 and 3.

Note:

- You do not need to include the contents of called methods in your graphs.
- Only the start node, end node, and edge arrows directly affect your mark. However, following all instructions may become a relevent factor in the event of a marking dispute.

3.3 Submission

Only one person should submit their completed assignment for their group, via Blackboard with the following items:

- A zip folder including:
  - JUnit test code (in one or more Java code files).
  - A ReadMe text file describing:
    - How to compile and run your tests
    - Which information is specified in-code and which is specified in the report (see Section 3.1)
    - The root folder for your code files. The contents of your root folder will be copied into the *src* folder from the source materials (see Section 1)
  - A report, in PDF format, including the Control Flow Graphs as well as the details of your tests.
- A completed and signed group assessment item cover sheet (see https://www.newcastle.edu.au/current-students/study-essentials/forms-and-guides)

Figure 1: Simple example test code, which tests the *Greeting.greeting(boolean)* method in Figure 2.

```
package seng3320.assignment1.instructions;
/**
 * An example class for a group of tests
 * The identifier for this class would be:
 * seng3320.assignment1.instructions.Tests
 */
class Tests {
    /**
     * An example Test.
     * The identifier for this method would be:
     *  seng3320.assignment1.instructions.Tests.testForBye()
     */
    @Test
    public static void testForBye() {
        /**
         * Inputs
         */
        boolean theInput = true;

        /**
         * Expected Outputs
         */
        String theOutput = "bye";


        /**
         * Get result
         */
        boolean result = Greeting.greeting(theInput);

        assertEquals(result, theOutput);
    }
}
```

Figure 2: Simple example code corresponding to Figure 3.

```
1  package seng3320.assignment1.instructions;
2  class Greeting {
3      public static String greeting(boolean notGreet) {
4          boolean greet = !notGreet;
5          if(greet) {
6              String message = "hi";
7              return message;
8          } else {
9              String message = "bye";
10             return message;
11         }
12     }
13 }
```

Figure 3: A simple sample Control Flow Diagram corresponding to Figure 2.