DIT636/DAT560 - Assignment 2:
Unit and Structural Testing

Due Date: Sunday, February 26th, 23:59 (Via Canvas)

There are two questions worth a total of 100 points. You may discuss these problems in your teams and turn in a single submission for the team (zipped archive) on Canvas. Answers must be original and not copied from online sources.

Cover Page: On the cover page of your assignment, include the name of the course, the date, your group name, and a list of your group members.

Peer Evaluation: All students must also submit a peer evaluation form. This is a separate, individual submission on Canvas. Not submitting a peer evaluation will result in a penalty of five points on this assignment.

Problem 1 - Unit Testing (60 Points)

Software engineers love caffeine, so we plan to install a new coffee maker in the classroom. Fortunately, the CSC department at North Carolina State University (NCSU) has developed control software for a shiny new CoffeeMaker and has provided us with that code. We just have to test it.

You will be working with the JUnit testing framework to create unit test cases, find bugs, and fix the CoffeeMaker code from NCSU's OpenSeminar project repository (thanks to the authors!). The example code comes with some seeded faults.

For this exercise, you are required to create a plan, implement the plan as unit test cases for all the application classes with JUnit (excluding Main and the exceptions), execute those against the code, detect faults, and fix as many issues as possible.

Your submission should include:

1) Test Descriptions. The system's core functionality is defined by the user interface (offered by the Main class). Based on your exploration of the system and its functionality, you will formulate a plan for how you will write unit tests for CoffeeMaker, Inventory, Recipe, and RecipeBook (excluding Main and the exceptions) to ensure that they deliver system functionality, free of faults. This document will describe the unit tests you have created, including a description of what each test is intended to do and how it serves a purpose in verifying system functionality. Your tests must cover the major system functionality, including both normal usage and erroneous input. You must explain why your tests will cover all major functionality.

2) Unit tests implemented in the JUnit framework.
3) Instructions on how to set-up and execute your tests (if you used any external libraries other than JUnit itself, or did anything non-obvious when creating your unit tests. You may include a build script if you created one).
4) List of faults found, along with a recommended fix for each and a list of which of your test cases expose the fault.

If you find faults by other means, such as exploratory testing, that are not detected by your unit tests, you should try to create unit tests that expose those faults.

Relevant links:

- CoffeeMaker code and sample tests **(do not turn in the sample tests as your own)** -
- jUnit - http://junit.org/
  (We recommend using a Java IDE - such as Eclipse or IntelliJ - that makes it easier to integrate JUnit into the development environment.)
- Instructions for executing JUnit tests in your IDE of choice:
  https://junit.org/junit5/docs/current/user-guide/#running-tests

Points will be divided up as follows: 15 points for test plan, 25 points for unit tests, 10 points for detecting faults, and 10 points for the suggested fixes to the codes.

Problem 2 - Structural Coverage (40 Points)

After testing the CoffeeMaker using your knowledge of the functionality of a coffee machine and your own intuition, you have decided to also use the source code as the basis of additional unit tests intended to strengthen your existing testing efforts.

You have identified the following methods in particular as worthy of attention:

- CoffeeMaker::makeCoffee
- Inventory::addSugar
- Recipe::setPrice
- RecipeBook::addRecipe

1. Design new unit tests to achieve full Branch Coverage over these four methods. Describe what code elements each test should cover, and why. Do not reuse test cases from unit testing. Instead, approach test design with the goal of covering these code elements and design new test cases.
2. Using any available coverage measurement tool, measure the Line Coverage (this is the same as Statement Coverage) of your unit tests (including those from both Problems 1

and 2)[1]. Include a coverage report in your submission, detailing the coverage achieved by your code[2].

3. If your tests have not achieved 100% Line Coverage of CoffeeMaker, Inventory, Recipe, and RecipeBook (excluding Main and the exceptions), design additional test cases to complete Line Coverage. Explain which additional elements those tests are designed to cover. If Line Coverage cannot be fully achieved, explain why.

Points will be divided up as follows: 25 points for unit tests designed for parts 1 and 3 of the problem, 10 points for the coverage report, and 5 points for the explanation of how coverage was either completed (or why it cannot be achieved).

---

[1] To measure coverage in IntelliJ, see https://www.jetbrains.com/help/idea/code-coverage.html. You may use any of the three coverage runners. In Eclipse, use EclEmma: https://www.eclemma.org/. If using the command line, use EMMA: http://emma.sourceforge.net/.

[2] Both the IntelliJ coverage runner and EclEmma can output a report (e.g., https://www.jetbrains.com/help/idea/viewing-code-coverage-results.html#coverage-in-editor). Be sure that you include all files from the generated report (i.e., not just index.html).