DIT636 / DAT560 - Assignment 3:
Unit and Structural Testing

Due Date: Sunday, March 2, 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 submission for the team (consisting of a PDF report and a zipped archive containing test cases) on Canvas. Answers must be original and not copied from online sources or generated by AI tools.

Report Template: A template for your submission report is available here. You may modify this template for your own purposes, if needed.

Cover Page: On the cover page of your report, 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.

Problem 1 - Unit Testing (70 Points)

Software engineers love caffeine, so we are planning 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.

The core functionality of the system is defined by the user interface (offered by the Main class).

Based on your exploration of the system and its functionality, you will write unit tests using JUnit for CoffeeMaker, Inventory, Recipe, and RecipeBook (excluding Main and the exceptions), execute those tests against the code, detect faults, and fix as many of the faults as possible.

The source code is available from
https://github.com/Greg4cr/dit636_examples/tree/main/src/as2-coffeemaker

Your submission should include:

1) **A document containing:**
    a) **Test descriptions**
        - Describe the unit tests that 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.
    b) **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).
    c) List of faults found, along with a recommended fix for each, and a list of which of your test cases expose the fault.
2) **Unit tests implemented using the JUnit framework**
3) **(Optional) You may include a build script if you created one.**

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:

- JUnit guide: https://junit.org/junit5/docs/current/user-guide/#writing-tests

- Instructions for executing JUnit tests in your IDE of choice: https://junit.org/junit5/docs/current/user-guide/#running-tests
- More instructions for running JUnit tests in IntelliJ IDEA: https://www.jetbrains.com/help/idea/junit.html

We recommend using a Java IDE - such as Eclipse or IntelliJ - that makes it easier to integrate JUnit into the development environment.

Points will be divided up as follows: 20 points for test descriptions, 25 points for unit tests, 10 points for detecting faults, and 15 points for the suggested fixes to the codes.

Problem 2 - Structural Coverage (30 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. **Either design new unit tests to achieve full Branch Coverage over these four methods, or if you already have achieved full Branch Coverage over these methods in Problem 1, identify the subset of tests that achieve this coverage.**
    - **In either case, describe what specific code elements each test covers, and explain exactly why the chosen inputs cover those elements in the manner needed for Branch Coverage.**
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).**
    - **Include a coverage report in your submission, detailing the coverage achieved by your code.**
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.**
    - **If Line Coverage cannot be fully achieved, explain why.**

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/.

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)

Points will be divided up as follows: 20 points for unit tests and explanations of those tests (and the coverage they achieve) for parts 1 and 3 of the problem, 5 points for the coverage report, and 5 points for the explanation of how line coverage was either completed (or why it cannot be achieved).