DIT636 / DAT560 - Assignment 3:
Unit and Structural Testing

Due Date: Sunday, March 1, 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 zipped archive containing a PDF report and 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)

You are a software engineer, which means two things:

1) You like to develop software.
2) You like cute animals.

Naturally, then, you have been hired for your dream job - **designing the control software for a new smart pet feeder system**. This is a control system designed to automate the feeding of your pets. It manages an inventory of food ingredients and allows users to configure meal plans (recipes) to dispense food based on dietary needs and a simple energy model.

The system provides the following core features via its API and Console Interface (both offered by the Main class):

- **Add, Edit, or Delete Meal Plans:** Configure feeding profiles by specifying the required amount of each ingredient; the energy cost is then calculated automatically.
- **Replenish Food:** Add raw ingredients (Kibble, Water, Wet Food, Treats) to the internal food container.
- **Check Stock:** Query the current quantity of ingredients in the container.
- **Dispense Meal:** Select a meal plan. The system checks if there is sufficient stock and remaining energy budget before dispensing.
- **Configure Scheduled Feeding:** Set up a recurring schedule to automatically dispense a selected meal while the program is running.

You will be working with the JUnit testing framework to create unit test cases for, find bugs in, and fix the Smart Pet Feeder system. The example code comes with some seeded faults.

Based on your exploration of the system and its functionality, you will write unit tests using JUnit for the individual classes in the system (excluding the class Main and the two exception classes), execute those tests against the code, detect failures, and fix as many faults as possible.

The source code is available from
https://github.com/EsmeYi/dit636_examples/tree/main/as2-petfeeder

Your submission should include:

- **A document containing:**
  - **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.
  - **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).
  - **List of faults found, along with a recommended fix for each, and a list of which of your test cases expose the fault.**
- **Unit tests implemented using the JUnit framework**
  - The test case code should be clearly written and the code should be documented.
  - They should map to the tests listed in the description document.
  - The tests should be concise and focus on one clear scenario or segment of the code.
  - Proper JUnit syntax should be followed.
  - Remove the sample test cases, or modify them to create new test cases.
- **(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 IntelliJ - that makes it easier to integrate JUnit into the development environment.

Points will be divided up as follows: 30 points for test descriptions, 15 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 Smart Pet Feeder System using your knowledge of the functionality 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:

- PetFeeder::dispenseMeal
- FoodContainer::addKibble
- Recipe::setAmtWetFood
- MealPlanBook::addMealPlan

If you identified any faults in these methods in Problem 1, use the fixed version of the method in this problem.

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 in detail 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 all classes (excluding Main and the two exceptions), design additional test cases to complete Line Coverage.**
    - **If Line Coverage cannot be fully achieved, explain why.**

To measure coverage:

- 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 VSCode, see https://code.visualstudio.com/docs/debugtest/testing#_test-coverage.
- In Eclipse, use EclEmma: https://www.eclemma.org/.
- If using the command line, you can use Cobertura (https://cobertura.github.io/cobertura/) or EMMA (http://emma.sourceforge.net/).

Most coverage tools (e.g., the IntelliJ coverage runners and EclEmma) can output an HTML 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). If you are using a tool that cannot generate an HTML report, create your own report (with screenshots/logs).

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).