CS321 Final Project

Program (80%)

For the final project, you'll be making use of all the new tools and techniques you've learned to build a simple simulation of your choice. There are a number of options that you have here:

- Robot Simulation
- Electrical Power Grid (or other public utility) Simulation
- Dyson Sphere Simulation
- Traffic Simulation
- Planetary Cargo Transport Simulation
- Social Media Simulation
- …

This list is not exhaustive. You are free to do any kind of simulation that you want, but it absolutely must use the skeleton provided. As a bonus, you can also implement a Java Swing visualization for your simulation.

Before starting, carefully read everything below, and look at the grading criteria to know what you will be marked on. **You need to implement at least 5 design patterns of your choice (worth 4% each). You will be graded on the choice, and the implementation.** You'll have a chance to justify your choices in a design decisions report.

Example: Robot Simulation

- You could make three types of robots: Retriever, Producer, Depositor
  - The Retriever would visit the Producer on one action, then visit the Depositor on the next action.
  - The Producer would create some form of produce (e.g. ore, wood, iPhones, etc.) and give some to a Retriever (all in a single action).
  - The Depositor would receive something from a Retriever, and deposit into some form of storage.
- You can make multiple types of Retriever/Producer/Depositor robots that perform their actions slightly differently.
- Make sure that you can make a simulation/program that uses 5 design patterns.
  - There are many patterns that are simple enough to implement but you will certainly need to use at least 1 behavioural pattern to simplify communications between the various classifications of the robots.

Skeleton Program

Along with this document, you'll find a skeleton program that you can build from.

It has the following components:

- **Matrix**: This is where your simulation is run from, and where you will be creating your objects and threads.

- **Unit:** Create concrete classes which inherit from this abstract class. **See the Robot example in Main.java.**

- **SimulationInput:** This is a helpful container for the input to the simulation. It will hold things like the amount of time to run for, and the number of actions that should be performed per second.

- **Statistics:** This is a singleton that will hold the statistics for the simulation. You will need to modify it to add new statistics and provide a way to output them.

- **Main:** This is where the program starts from. It should be used for testing. You will find helper methods for running tests with different forms of input.

Be sure to read through all the code in those files, there are hints to help you scattered throughout. You are free to modify them but the final product needs to do the same thing as the original (I suggest not changing the basic functionality).

The skeleton program is straightforward:

- It creates an input object called SimulationInput and adds the input to it.
- Then it starts a test with the Robot example class (in Main.java):
    - Here, we initialize the Statistics singleton with the given input.
    - Then, we start the Matrix simulation.
        - The test initializes the Robot/Unit example and runs it with the given input.
    - Finally, we return the statistics object to the main method.
- After the test, we output the final statistics from the test run.

Your units are all Runnable so they absolutely must be run as a Thread. They need to output information as they perform their actions as well, they can't stay silent in the background (unless that is purposeful). **The simulation must be entirely multi-threaded. You must use 1 semaphore for some form of interactivity.**

Testing (10%)

Our standard techniques for testing also apply here. You will want to check expected vs. actual values produced. Another thing to check is logging, but ensure that you make a comment about this if you are looking at the logs. That said, in a multi-threaded environment we are leading ourselves into non-deterministic runtime environments so it will be impossible to check expected vs actual values for all of your metrics (some will still work though).

There is only one way around this: **build unit tests for all of your code**. These tests only run a single portion of your code (e.g. one test for a single method in a class). Given the time constraint, building unit tests is unfeasible as it can take as much, or more time to write than the source code.

In this case, **you can test your program by running a handful of known successful tests, along with a collection of tests that probe edge cases in your program** (e.g. can it handle 0 actions/second?).

Design Decisions Report (10%)

As a future software engineer, you need to get used to writing out your reasons for doing one thing over another. This will help you with keeping track of decision history, explaining your solution to others, and finding possible simplifications in your program. If it helps, assume that you are attempting to convince one of your managers that the way you built this program, is the only way to build it.

You will need to produce a report of at least 2 pages (or 1000 words) describing your design decisions, as well as listing all the design patterns you used and where they are located. This will include reasoning for things like “why I used the Factory method in location XYZ”.

Include a visual design of your program. The visualization does not need to follow UML conventions, but it needs to be clear, and simple. This design should/could be partially completed prior to programming as well.

I expect to see all of the following in your document:

- Overview of your program.
  - Idea you selected.
  - Overview of how it’s implemented.
- Design/implementation decisions
  - Patterns used
  - Choice of logic and/or data structures
  - Usage of DRY, SOLID
  - Synchronization choices
  - Etc.
- Visual design of the program
  - Try using an extension in your IDE to produce a UML diagram
  - Alternatively, you can build a simplified one that doesn’t need to follow UML

Grading Criteria:

Style/submission guidelines: https://gmierzwinski.github.io/bishops/cs321/style_guidelines.html

Comments, Formatting, & Readability	**5%**
Submission Guidelines	**5%**
Design Patterns (4% per Pattern)	**20%**
General Logic of Program **(e.g., algorithms)**	**15%**
Object-Oriented Programming **(e.g., encapsulation, polymorphism, DRY, …)**	**20%**
Multi-Threading Usage and Implementation (Semaphores, Synchronization, …)	**10%**
Testing	**10%**
Design Decisions Report	**10%**
Creativity	**5%**
BONUS: Visualization	**5%**
Total	**105%**