Assignment 3 – Design Patterns

Be careful that you follow the instructions in this document exactly.

Objective:

I am your boss. I just hired you and I'm ready to test you out on this new contract our company acquired writing a warehouse robotics simulation in Java. I read on your resume that you learned about design patterns when you were at Dalhousie. That means that you and I now speak a common language, the language of design patterns. I can tell you things like "I want you to implement that feature by using Mementos" and you should know what I'm talking about right? Right.

The simulation is simplistic, but we are going to write the code in the best way we can so that we can later grow the complexity of the simulation without having to perform significant refactoring. In its current state the simulation simulates a single shelf in a warehouse, and an unknown number of trucks that need to be loaded with items from the shelf. The truck is loaded by robots. These robots are battery powered and can do all of the steps required to take an item from the shelf and load it onto the truck. Their batteries are rechargeable, and we assume that there are unlimited recharging stations all over the warehouse. A robot has a backup reserve of power such that no matter where it is in the warehouse it can always move to a recharging station when needed.

The robots do not have complicated AI, they simply follow the state diagram in **Appendix A**.

I have written the shell of the simulation, mostly the support code that gets you up and running and straight to writing the simulation logic, as well as some interfaces to get you on the right path.

I used the following patterns to implement the simulation so far:

Pattern	Classes Involved	Purpose
Singleton		
	Simulation	Central place where the simulation logic is constructed and executed. It uses HAS-A to contain an instance of the abstract factory used to create all objects in the simulation, and it has the list of robots you can use to detect when the simulation is complete, and finally it contains the reference to the Shelf object, that tracks how many items need to be loaded onto trucks.
Abstract Factory		

	IWarehouseFactory	Interface of the Abstract Factory that instantiates all of the simulation objects. **It needs to be completed by you.**
	WarehouseFactory	Concrete factory class that does the work. **You must complete this class.**

You must do the following:

1. Use the **State** design pattern to model all of the possible states a Robot can be in (see the state diagram in Appendix A). I have started this process for you by providing you with the abstract base class for the pattern (State.java). You must implement the following concrete State classes:
    a. IdleState
    b. RechargeState
    c. FinishedState
    d. ClaimItemState
    e. MoveToShelfState
    f. TakeItemState
    g. MoveToTruckState
    h. LoadTruckState
    i. **Note:** In the state diagram, the **number in the yellow circle next to each state** is how many minutes of battery charge are required to execute the logic in the state. Each state must remain in the state for the number of minutes in that circle, this is where the system simulates time spent doing the task. You must set this value correctly in the state classes you write in order for your program to produce the correct results when the marker runs it.
2. Use the **Observer** design pattern to handle the time logic for the simulation. A class named **TimerSubject** should allow ITimerObserver objects to attach and detach, and should have a **notifySubjects()** method to notify all of the observers that **1 minute has passed.** For convenience, the TimerSubject class should also be a singleton. This makes it simple for the Simulation object to retrieve it, add robots to it and to call the notifySubjects() method in the simulation loop.
    a. The observers will be the Robot objects. Each robot when notified via timerElapsed() will update their internal State object and potentially transition to a new state. Note the logic for transitioning to a new state, first the Robot must check whether its Battery object has enough charge to complete the new state before entering into the state. If it does not have enough charge it must enter the RechargeState and transition to the desired state AFTER recharging.
    b. Once implemented, replace the //TODO comment in the Simulation.run() method with your TimerSubject's notifySubjects() method to indicate 1 minute has passed to all observers.
3. Implement the recharge logic in the Robot class where you see the //TODO comment. Your RechargeState class must be written in a way such that it knows what state to transition to when the recharge is complete.

4. Complete the IWarehouseFactory and WarehouseFactory Abstract Factory pattern. All instantiation of State objects, Battery objects, Robot objects and decorator objects should be done in the factory class. All classes can use the Simulation singleton to get an instance of the factory to use to instantiate their objects. This will let us substitute more complicated objects later on when we expand the simulation.
    a. When you have a makeIdleState() method, use that in the methods that create Robot objects to ensure the Robot objects start in the IdleState.
5. Use the **Decorator** pattern to implement a BatteryPack decorator class that decorates a Battery object. The BatteryPack acts as an additional battery for the robot. Electricity is drawn from the battery pack first, and then the object the BatteryPack is decorating. Recharging recharges the decorated battery first, then the battery pack. Battery pack batteries recharge at the same rate as normal batteries (see Battery.recharge())
6. When you have written all of your classes and completed your WarehouseFactory class, use the factory to build all of the objects needed for the simulation in the Simulation.build() method. Use the Arguments object passed into the method to know how many shelf items there are, how many robots to create, the battery capacity and also the number of battery pack robots to create.

Command Line Arguments:

The program is written already to take 5 integer command line arguments. The command to run the simulation is:

java Main <shelf item count> <num robots> <default battery capacity> <num battery packs> <battery pack capacity>

E.g.: java Main 9734 100 10 4 20

The logic to parse the arguments and get them to you in the Simulation class is already done, see the Arguments class.

When the simulation runs, the program outputs how many minutes it took for all the robots to load all of the items from the shelf to the truck. The marker knows how long it should take based on the inputs, and this will be used to assess your program's correctness.

Requirements:

1. You do not require unit tests for this assignment
2. Pull the CSCI5308 sample repo's latest, you will need the code from the Assignments\A3 folder for this assignment.
3. Everything for this assignment must be in a folder labeled "A3" at the root folder in your private repo, the same repo you used for assignments 1 and 2. No other folders should be in the A3 folder. A3 should contain nothing but .java files. **Do not commit .class files, readme files, environment or IDE files or anything but .java files.**

4. Begin by copying the .java files from the Assignments\A3 folder to your A3 folder.
5. Program your portion of the assignment.
6. Your code must compile (without error or warnings) and run via the following commands executed in **your A3 folder**:
    a. javac *.java
    b. java Main <arguments> *(see above)*
7. Note the above command-line arguments, these will be passed to your program by the marker, and your program will run the simulation with the provided inputs. The arguments will not be wrapped in <>'s obviously. They are parsed automatically by the Arguments class, so you simply need to use the values loaded in that object for your simulation.
8. **Note: You are not allowed to modify in ANY WAY the following files that were supplied to you to get you started on the assignment:**
    a. Main.java
    b. Arguments.java
    c. Battery.java and IBattery.java
    d. Shelf.java
    e. Robot.java and IRobot.java
    f. State.java
    g. ITimerObserver.java
9. Submit your code by pushing all or your changes to the main branch of your individual repo **before** the assignment deadline.

Marking Rubric:

- Program correctly functions and correctly processes command line arguments as defined in this document: 20%
- Proper use of design patterns as instructed in this document: 60%
- Code readability / understandability: 20%
- **Code does not compile = Automatic 0 on entire assignment.**
- **Modifying any of the files you are not allowed to modify = Automatic 0 on entire assignment.**
    - This is important because you are learning to work on a team and to follow through on a specification given to you, you cannot do this if you just change interfaces and other people's classes to suit your own designs.
    - We will test this by replacing these files with the originals from the course repo, so modifying them will not work, and will result in your code not compiling.

Appendix A – Robot State Diagram