This assignment is intended to be done by your team of two to three students. You may collaborate on answers to all questions or divide the work for the team. In any case, the team should review the submission as a team before it is turned in.

Project 3 is intended as a continuation of Project 2's simulation of the Friendly Neighborhood Music Store (FNMS). You may reuse code and documentation elements from your Project 2 submissions. You may also use example code from class examples related to Project 2. In any case, you need to cite (in code comments at least) any code that was not originally developed by your team.

Part 1: UML exercises – 25 points

Provide answers to each of the following in a PDF document:

1) (15 points) For the existing FNMS Project 2, create either a detailed UML Activity diagram to describe the flow of actions and decision points in the simulation, or a detailed UML State diagram that shows program states and transitions that cause state changes.
2) (10 points) Draw a class diagram for extending the FNMS simulation described in Project 3 part 2. The class diagram should contain any classes, abstract classes, or interfaces you plan to implement. Classes should include any key methods or attributes (not including constructors). Delegation or inheritance links should be clear. Multiplicity and accessibility tags are optional. You should note what parts of your class diagrams are implementing the three required patterns below: Strategy, Decorator, and Observer.

Part 2: FNMS simulation extended – 50 points (with possible 10 point bonus)

Using the Project 2 Java code developed previously as a starting point, your team will create an updated Java program to simulate extended daily operations of the FNMS. The simulation should perform all functions previously enabled in Project 2. Clerks will continue to perform all functions they performed in Project 2. The simulation will be refactored with the following extensions.

Changes to Store:

- The decision has been made to stop selling Clothing. After initially populating the store inventory of Items, when individual subclasses of Clothing items run out (inventory goes to 0) they will not be reordered by the PlaceAnOrder action. Once all Clothing items are sold (i.e. hats, shirts, and bandanas are all at 0 inventory), the Store will no longer purchase Clothing items from Customers.
- The number of customers arriving each day to buy Items is changing from a uniform 4 to 10 arrivals to the following: 2 plus a random variate from a Poisson distribution with mean 3 (this will result in random numbers from 1 to about 6 or 7 with a rare spike to 10 or so). The number of selling customers remains a uniform random 1 to 4.

Changes to Clerks:

- An additional Clerk will be hired – Daphne. The pool of three Clerks will be used to send one random Clerk to the store each day. As before a Clerk cannot work more than three days in a row. Decide on the new Clerk's chance of damage during CleanTheStore, the chance should be unique to the Clerk.
- There is a 10% chance in each workday that one randomly chosen Clerk may be sick for that day. If a Clerk is sick, they will not be available to work at the store. If a Clerk is sick, that should be announced.

Changes to Items:

- There are new Items to add to the simulation:
  - new Wind Instrument subclass, the Saxophone (like a Flute, a Saxaphone has a type)
  - new Music subclass, the Cassette
  - new Player subclass, the Cassette Player
  - new Accessory subclass, the Gig Bag
- Players will have a new property = equalized. When players are first added to inventory, they are not equalized (equalized = false). If a player has been equalized, there is a 10% increase in the likelihood it will be sold at any customer sale point.
- Stringed Instruments will have a new property = tuned. When stringed instruments are first added to inventory, they are not tuned. If a Stringed Instrument has been tuned, there is a 15% increase in the likelihood it will be sold at any customer sale point.
- Wind Instruments will have a new property = adjusted. When wind instruments are first added to inventory, they are not adjusted. If a Wind Instrument has been adjusted, there is a 20% increase in the likelihood it will be sold at any customer sale point.

New Behaviors:

- Use a Decorator pattern to add these optional sales to the normal item sale methods for the following cases. When a Stringed Instrument is sold, there is a chance of selling accessories as well. If the Stringed Instrument is electric, there is a 20% chance of selling a single Gig Bag, a 25% chance of selling a single Practice Amp, a 30% chance of selling 1 or 2 Cables, and a 40% chance of selling 1 to 3 Strings. If the Stringed Instrument is not electric, each of these chances of an additional sale is reduced by 10%. These additional items should be tracked in Inventory as usual. Announce each additional item sold.

- Use a Strategy pattern to assign a Tune algorithm to each Clerk when they are instantiated. There are three different Tune algorithms – Haphazard, Manual, and Electronic – each of the three clerks should have a unique Tune method assigned to them at their instantiation. The three Tune algorithms are:
  - Haphazard Tuning: The Clerk will have a 50% chance of changing the property (equalized, tuned, or adjusted) by flipping the state (if it was true, it becomes false; if it was false, it becomes true).
  - Manual Tuning: The Clerk will change the property from false to true 80% of the time, and from true to false 20% of the time.
  - Electronic Tuning: The Clerk will change the property from false to true automatically, and never from true to false.

- During the DoInventory step, Clerks will now Tune each Player, Stringed Instrument, and Wind Instrument in inventory by attempting to change the equalized, tuned, or adjusted property (ideally from false to true) using their assigned Tune algorithm. If the property being adjusted changes from true to false during the Tune behavior, there is a 10% chance the item will be damaged (as in CleanTheStore, with the same resulting condition change and possible removal from inventory). All results from tuning should be announced.

- Use an Observer Pattern to publish a summary of Clerk actions (this is in addition to any announcements that are made in the normal running of a Store). Include the name of the Clerk causing the action that requires the published event
    - Publish the following events:
        - ArriveAtStore: Publish which clerk has arrived at the store.
        - ArriveAtStore: Publish number of items added to inventory (if any).
        - CheckRegister: Publish the amount of money in the register.
        - GoToBank: Publish the amount of money in the register.
        - DoInventory: Publish the total number of items.
        - DoInventory: Publish the total purchase price value of inventory items.
        - DoInventory: Publish the total number of items damaged in tuning.
        - PlaceAnOrder: Publish the total number of items ordered.
        - OpenTheStore: Publish the total number of items sold.
        - OpenTheStore: Publish the total number of items purchased.
        - CleanTheStore: Publish the total number of items damaged in cleaning.
        - LeaveTheStore: Publish which clerk has left the store.
    - Create an event consumer class called a Logger. The Logger object should be instantiated at the beginning of each day and should close at the end of each day. The Logger object should subscribe for the published events of a day's run and write each of them in a human readable form as they are received to a text file named "Logger-n.txt" where n is the day of the simulation.
    - Create an event consumer class called a Tracker. The Tracker object will be instantiated at the beginning of the simulation run and stay active until the end. The Tracker will subscribe for the published events and maintain a data structure in memory by Clerk with the total items sold, purchased, or damaged by that Clerk up to that point in the simulation. At the end of each day the Tracker should print a summary of the cumulative data like:

      Tracker: Day 4

Clerk	Items Sold	Items Purchased	Items Damaged
Daphne	0	0	0
Shaggy	8	2	3
Velma	4	1	1

Simulate the running of the FNMS store for 30 days. At the end of the 30 days, print out a summary of the state of the simulated store as before, including:

- the items left in inventory and their total value (purchasePrice)
- the items sold, including the daySold and the salePrice, with a total of the salePrice
- the final count of money in the Cash Register
- how much money was added to the register from the GoToBank Action

There may be possible error conditions due to insufficient funds or lack of inventory that you may need to define policies for and then check for their occurrence. You may find requirements are not complete in all cases.

The code should be in Java 8 or higher and should be object-oriented in design and implementation.

In commenting the code, clearly indicate where instances of the Decorator, Observer, and Strategy patterns are used.

Capture all announcement output from a single simulation run in your repository in a text file called Output.txt (by writing directly to it or by cutting/pasting from a console run). This should include output from the Tracker object as well. You should also include each of the "Logger-n.txt" files created by the Logger object in your repo.

Also include in your repository an updated version of the FNMS UML class diagram from part 1 that shows your actual class implementations in part 2. Note what changed between part 1 and part 2 (if anything) in a brief comment paragraph. Again – note where patterns are in use in the diagram.

Bonus Work – 10 points for JUnit test example

There is a 10-point extra credit element available for this assignment. For extra credit, import a version of JUnit of your choice, and use at least ten JUnit test (assert) statements to verify some of your starting expected objects are instantiated or to perform other similar functionality tests. For full bonus points you must document how you run your JUnit tests (e.g. with a command line or in the IDE), and you must capture output that shows the results of your tests. You can decide how the test methods are integrated with your production code.

In practice, writing your tests before development is recommended, but for this academic example, I recommend you do not pursue this bonus work until you are sure the simulation itself is working well. If you need support on using JUnit, I mention several references in the TDD lecture, but here are key helpful ones:

- The JUnit sites for JUnit 5 (https://junit.org/junit5/) and JUnit 4 (https://junit.org/junit4/)
- The Jenkov JUnit tutorials (they are for JUnit 4, but are extremely clear and helpful regardless): http://tutorials.jenkov.com/java-unit-testing/index.html
- Organizing your JUnit elements in your code: https://livebook.manning.com/book/junit-recipes/chapter-3/1

Grading Rubric:

Homework/Project 3 is worth 75 points total (with a potential 10 bonus points for part 2)

Part 1 is worth 25 points and is due on Wednesday 2/16 at 8 PM. The submission will be a single PDF per team. The PDF must contain the names of all team members.

Question 1 will be scored based on your effort to provide a thorough UML activity or state diagram that shows the flow of your Project 2 simulation. Poorly defined or clearly missing elements will cost -1 to -3 points, missing the diagram is -15 points.

Question 2 should provide a UML class diagram that could be followed to produce the FNMS simulation program in Java with the new changes and patterns. This includes identifying major contributing or communicating classes (ex. Items, Staff, etc.) and any methods or attributes found in their design. As stated, multiplicity and accessibility tags are optional. Use any method reviewed in class to create the diagram **that provides a readable result**, including diagrams from graphics tools or hand drawn images. **The elements of the diagram that implement the Observer, Strategy, and Decorator patterns should be clearly annotated.** A considered, complete UML diagram will earn full points, poorly defined or clearly missing elements will cost -1 to -2 points, missing the diagram is -10 points.

Part 2 is worth 50 points (plus possible 10 point bonus) and is due Wednesday 2/23 at 8 PM. The submission will be a URL to a GitHub repository. The repository should contain well-structured OO Java code for the simulation, a captured Output.txt text file with program results, the Logger-n.txt files, the updated UML class diagram for Part 2, and a README file that has the names of the team members, the Java version, and any other comments on the work – including any assumptions or interpretations of the problem. Only one URL submission is required per team.

20 points for comments and readable OO style code: Code should be commented appropriately, including citations (URLs) of any code taken from external sources. We will also be looking for clearly indicated comments for the three patterns to be illustrated in the code. A penalty of -2 to -4 will be applied for instances of poor or missing comments, poor coding practices (e.g. duplicated code), or excessive procedural style code (for instance, executing significant program logic in main).

15 points for correctly structured output as evidence of correct execution: The output from a run captured in the text file mentioned per exercise should be present, as should be the set of Logger-n.txt files. A penalty of -1 to -3 will be applied per exercise for incomplete or missing output.

5 points for the README file: A README file with names of the team members, the Java version, and any other comments, assumptions, or issues about your implementation should be present in the GitHub repo. Incomplete/missing READMEs will be penalized -2 to -5 points.

10 points for the updated UML file showing changes from part 1 to part 2 as described. Incomplete or missing elements in the UML diagram will be penalized -2 to -4 points.

Please ensure all class staff are added as project collaborators to allow access to your private GitHub repository. Do not use public repositories.

Overall Project Guidelines

Assignments will be accepted late for four days. There is no late penalty within 4 hours of the due date/time. In the next 48 hours, the penalty for a late submission is 5%. In the next 48 hours, the late penalty increases to 15% of the grade. After this point, assignments will not be accepted.

Use e-mail or Piazza to reach the class staff regarding homework/project questions, or if you have issues in completing the assignment for any reason.