This assignment is intended to be done by your team of two 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 Pet Store (FNPS). 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 – 20 points

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

1) (10 points) For the existing FNPS 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 FNPS 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: FNPS simulation extended – 55 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 FNPS. The simulation should perform all functions previously enabled in Project 2. Employees will continue to perform all functions they performed in Project 2. The simulation will be refactored with the following extensions.

Changes to the store:

- The decision has been made to stop selling Toys. After initially populating the store inventory of items, if the individual subclass of toy items runs out (inventory goes to 0) they will not be reordered by the PlaceAnOrder action.
- The number of customers arriving each day to buy Items is changing from a uniform 3 to 10 arrivals to the following: 2 plus a random variate from a Poisson distribution with mean 3 (this will result in random Poisson numbers from 1 to about 6 or 7 with a rare spike to 10 or so). The Poisson variates can be created with a library function or a small piece of custom Java code.

Changes to employees:

- An additional clerk and trainer will be hired. The pool of three of each employee type will be used to send one random employee of each type to the store each day. As before an employee cannot work more than three days in a row.
- There is a 10% chance in each workday that a chosen employee may be sick for that day. If an employee is sick, they will not be available to work at the store, and an alternate should be selected. If an employee is sick, that should be announced.

Changes to items:

- There are new Items to add to the simulation:
  - New Pets
    - Ferret (color, housebroken)
    - Snake (size)
  - New Supplies
    - Treats (animal)

New behaviors:

- Use a Decorator pattern to add these optional sale add-ons to the normal sale methods for pets. Add-ons are not tracked as inventory. When buying a pet, there is a random chance each of these things may be added to the sale:
  - Can add a microchip (50% chance) for $50
  - Can add pet insurance (25% chance) for $50
  - Can add pre-paid vet checkups (1 to 4 of these, 25% chance) for $25 each
  - Announce each additional add-on sold when selling a pet

- Add a new action for trainers called TrainAnimals, which happens before the OpenTheStore action. The trainers will work with all pets in the inventory. Depending on the training method used by the trainer, the pet may change the value of its housebroken attribute.

- Use a Strategy pattern to assign a training algorithm to each trainer when they are instantiated. There are three different training algorithms – Haphazard, Negative Reinforcement, and Positive Reinforcement – each of the three trainers should have a unique train method assigned to them at their instantiation. The three training algorithms are:
  - Haphazard – 10% chance of toggling the housebroken attribute (if True, becomes False; if False becomes True)
  - Negative Reinforcement – 20% chance of housebroken changing from True to False; 40% chance of changing from False to True
  - Positive Reinforcement – 50% chance of changing from False to True

- Use an Observer Pattern to publish a summary of employee actions (this is in addition to any announcements that are made in the normal running of a Store). You can decide how to structure published events and how to model them (as push or pull observers). You can use any Observer support in libraries or write your own. Include the name and type of the employee causing the action that requires the published event.
  - Publish the following events:
    - ArriveAtStore: Publish which employee has arrived at the store
    - ProcessDeliveries: 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 after adding new funds
    - DoInventory: Publish the total number of pets and supplies
    - DoInventory: Publish the total purchase price value of all inventory items

        - PlaceAnOrder: Publish the total number of items ordered
        - OpenTheStore: Publish the total number of items sold by each employee
        - OpenTheStore: Publish the total purchase price of items sold by each employee
        - OpenTheStore: Publish the total number of customer visits
        - CleanTheStore: Publish an animal escape event
        - LeaveTheStore: Publish which employees have 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 will stay active until the end. The Tracker will subscribe for the published events it needs and will maintain a data structure in memory by employee with the total items sold by that employee 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

Clerks	Items Sold	Purchase Price Total
Dante	0	$0
Randal	8	$210
Caitlin	6	$125
Trainers	Items Sold	Purchase Price Total
Tugg	0	$0
Kirk	3	$95
Alpa	2	$64

Simulate the running of the FNPS 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 pets remaining in the sick pets collection
- 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 for this simulation are not complete in all cases. In such cases, either document your interpretation or find class staff for clarifications.

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 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 now 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.

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 running your tests.

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 in part 2). The project is due on Canvas as a URL to your repository on Wednesday 6/22 at 8 PM.

Part 1 is worth 20 points. The diagrams for Part 1 should be captured in a PDF that should also 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 -2 points, missing the diagram is -10 points.

Question 2 should provide a UML class diagram that could be followed to produce the FNPS simulation program in Java with the new changes and patterns. This includes identifying major contributing or communicating classes (ex. Items, Employees, 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 55 points (plus possible 10 point bonus). 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 PDF of Part 1, 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.

30 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).

20 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.

Please ensure all class staff are added as project collaborators (brmcu, gayathripadmanabhan1, wmaxdonovan-cu) 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.