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. You may reuse code and documentation from your Project 2 submissions. You may also use example code from class solutions to Project 2. However, 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 FLGS Project 2, create either a detailed UML Activity 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 FLGS 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: FLGS simulation extended – 50 points

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 a friendly local game store (FLGS). The simulation should perform all functions previously enabled in Project 2. Cashiers will continue to perform all functions they performed in Project 2. The simulation will be refactored with the following extensions.

1) Use a three-level inheritance model for Games: Game <- Family Game <- Monopoly (for instance). Use a Decorator pattern implementation to support the following optional purchases and additional cost for these games:
    a. Monopoly – someone buying a Monopoly game will optionally add 1 Special Tokens pack to their purchase 50% of the time
    b. For all Card Games – someone buying a Card game will optionally add 1 to 6 Special Cards to their purchase 20% of the time
    c. Mousetrap – someone buying a Mousetrap game will optionally add 1 to 2 Spare Parts to their purchase 30% of the time
    d. Gloomhaven – someone buying Gloomhaven will optionally add 1 to 4 Custom Miniatures to their purchase 20% of the time

    You can determine the cost of each added feature that the customer will pay the store. The use of Decorator should be clearly shown in the code.

2) Add a new Cashier (named "Bart"). Bart acts as previous Cashiers with two changes. First, Bart stacks games based on their width, with the widest games in the first shelf position. However, Bart will wait to stack games with an inventory count of one until all games with a greater inventory count have been stacked. Second, Bart will only damage games during the vacuum step with a 2% chance. In refactoring

to add Bart, Cashiers should be assigned a stacking behavior/algorithm using the Strategy pattern when they are initialized. The use of Strategy for this should be clearly commented in code. Note that Cashiers should be a subclass of Employees if they were not before. There is an even chance of any of the three Cashiers working on a given day.

3) Add a new Employee type called an Announcer (named "Guy"). Guy arrives each day just before the Cashier in the morning and is the last Employee to leave the store. Guy must subscribe for events from all other Employees, each of which becomes an observable publisher (using the Observer pattern). When an Employee (other than Guy) has an event to announce, they must publish the event to Guy. Guy will then announce their event, such as: "Guy says: Ernie stacks 2 Magic games in shelf position 4 (pile height = 14")". Guy is the only Employee capable of sending strings to System.out for output to files or to the console. Guy should also announce his own arrival at and departure from the store each day. The uses of Observer should be clearly commented in code. You can use any Observer support in Java you like, including writing your own. State in your README which Observer approach you used.

4) Add a new Employee type called a Baker (named "Gonger"). Gonger arrives each day at the store after the Cashier Count the Money event. Gonger will drop off packages of a dozen chocolate chip cookies that will be added to the store's new cookie inventory. Initially, Gonger will drop off 1 package of 12 cookies. The money in the store Cash Register should be reduced by ½ the sale price of the cookies being delivered (you decide the price), and that money should go into Gonger's pocket. Cookie events at the store may increase or decrease the number of cookie packages Gonger drops off (as described below). Gonger is an observable publisher, and will issue events for his arrival, the count of packages dropped off, the money he receives, and his departure (which Guy will announce).

5) Modify the Open The Store Cashier event as follows. The number of customers arriving each day is 1 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). Before each normal customer decides to buy games (as in Project 2), they may decide to buy cookies. If they buy cookies, they will decide to buy 1 to 3 cookies (randomly), adding the price of cookies to the Cash Register money and reducing the store cookie inventory. If they buy cookies, the chance they will buy a game increases by 20%. If they want to buy, for instance, 3 cookies, and only 2 remain, they will buy those remaining 2. If there are no cookies for a customer to buy, the chance a customer will buy a game decreases by 10%. The working Cashier should announce any cookie sales (via Guy).

6) When a customer visits, there is a 1% chance that the customer is the Cookie Monster in disguise. If there are any cookies in the store, the Cookie Monster will eat all the cookies in the store without paying for them and damage 1 to 6 games (randomly). If there are no cookies in the store, the Cookie Monster will sadly leave the store, taking no action. The working Cashier should announce any Cookie Monster events (via Guy). The logic for damaging games should be delegated and referred to by both this logic and the logic for Vacuum the Store.

7) Modify the Order New Games Cashier event as follows. If there are no cookies in the store at the end of a day, increase the number of cookie packages Gonger delivers the next day by 1. If there are cookies in

the store at the end of the day, have Gonger deliver 1 less package of cookies the next day (with a minimum of 1 package delivered).

Simulate the running of the store for 30 days. As in Project 2, at the end of the 30 days, for each game type, list the number in inventory, the number sold, and the total sales for that game type. Also list the contents of the Damaged Game container, the final count of money in the Cash Register, as well as how many times money was added to the register due to low funds in the Count the Money step. Additionally, list the number of cookies sold each day and in total, as well as total cookies lost to the Cookie Monster; and the total amount of money paid to Gonger (which he puts in his pocket) for his cookies.

Capture all 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).

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

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 statements to verify some of your starting expected objects are instantiated or other similar functionality tests. For full bonus points you must document how to run your code with or without running the JUnit tests (e.g. with a command line keyword), and you must capture a version of your output that shows how the output and run results differ when the tests are run. You can decide how the tests 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 fairly 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 9/29 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 diagram that shows the flow of Project 2. 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 FLGS simulation program in Java. This includes identifying major contributing or communicating classes (ex. Games, Employees) 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 tools or hand drawn images. 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 10/6 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, 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 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. 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 from part 1 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.