Benjamin Tsai

CSCI3353

Boba Guys Store

1. Background

The inspiration for this project originated from a very popular boba drink shop from my home area of the Bay Area called Boba Guys. While they only sell boba drinks, I decided to expand their offerings by adding purchase options for egg puff snacks and official Boba Guys merchandise. If you have never heard of boba drinks, they are a dessert-like beverage made with a variety of teas, other toppings, and boba, which are chewy tapioca pearls. If you have never heard of egg puff snacks, they are a waffle/pancake-like snack made with different flavored batter in the shape of an egg typically, but they can be made into other fun shapes. In my explanations of my design patterns, I will include pictures of both boba drinks and egg puffs for visual reference.

Through the use of Scanners to take in user input through the terminal, my project mimics a Boba Guys store offering boba drinks, egg puff snacks, and merchandise. **To run the program, please run the main method within the BobaGuysDriver.java file.**

2. State

The first design pattern I utilized was the state design pattern. The state design pattern is a behavioral design pattern designed to enable the global context's state to allow the program to which the design is applied to adapt its behavior situationally. This pattern is most often used when the global context of a project can be characterized by a small number of states and when the global context of a project can change at run-time, thus also changing the software's behavior. Therefore, I decided to use the state design pattern to mimic the hours of an actual shop. First, I use a Calendar object in my main driver program to retrieve the current time when the program is run. Then, based on that time, I assign the appropriate state to my overall project. I decided to establish that my shop's hours were from 11AM- 11PM, with the lunch menu being available from 11AM-5PM, the dinner menu being available from 5PM-11PM, and no menu being available outside of these hours. To represent these constraints, I created 3 classes that implemented the Store_State interface: Lunch, Dinner, and Closed. Each state has unique options displayed to the user. For instance, during lunch hours, you cannot order Egg Puffs, while during dinner, you can. For grading purposes, even when the store is "closed," there is an option to override the store state and select either the lunch menu or the dinner menu.

Files Involved: Store.java, Store_State.java, Lunch.java, Dinner.java, Closed.java

3. Builder

The next design pattern I utilized was the builder design pattern. The builder design pattern is a creational design pattern designed to organize the creational details of an object in an efficient hierarchy, handling the chronological and logical order of creating an object. This pattern is most often used when the creation of an object involves algorithmic steps and when you can think of all objects related to a class hierarchy as different versions of a generic object. Therefore, I thought the Builder pattern was perfectly suited to the making of a boba drink, as there are steps when creating the drinks (tea first, toppings after, etc.) and each type of drink is a variation of a generic boba drink. First, I created a BobaDrink class with basic instance variables for name, price and ingredients. Then, I created an abstract BobaDrinkMaker class equipped with protected instance variables reflecting all the possible ingredients, "add-ingredient" methods for all possible ingredients, and two abstract methods for making the BobaDrink and retrieving it. Finally, I created 3 subclasses of BobaDrinkMaker to represent 3 types of drinks: Classic Milk Tea, Strawberry Matcha Latte, and Thai Tea (all pictured above). Each of these subclasses implements a makeBobaDrink method that will throw a DrinkMakingStepViolation if the methods to make the drink are not called in the correct chronological order. Various Thread.sleep calls are utilized to mimic the waiting process an actual customer would experience.

Files involved: BobaDrink.java, BobaDrinkMaker.java, BobaGuysEmployee.java, ClassicBobaDrinkMaker.java, StrawberryMatchaLatteBobaDrinkMaker.java, ThaiTeaBobaDrinkMaker.java, Ingredient.java, DrinkMakingStepViolation.java

4. Prototype

In addition, I also utilized the prototype design pattern. The prototype design pattern is a creational design pattern designed to create objects by cloning a stored prototype of that object. This pattern is most often used when the various kinds of instances of the class needed are small and when these kinds of instances are not composed of deeply nested data structures. Therefore, I thought the Prototype pattern was perfectly suited to the making of egg puffs, as there are a set amount of flavors (plain, chocolate, strawberry, etc.) and each EggPuff object only needed to store its flavor and its price. First, I created an EggPuff class with a private constructor that can only be accessed by a makeNewEggPuffInstance method. The class, as aforementioned, also has flavor and price instance variables. Then, I created an EggPuffPrototypeManager class to manage the prototypes for the various flavors I planned on serving. The class is equipped with order methods corresponding to each flavor meant to be called by a master order method that takes in a String representing the desired flavor. Thread.sleep is utilized here as well, but in the Lunch and Dinner classes instead.

Files involved: EggPuff.java, EggPuffPrototypeManager.java, UnknownEggPuffFlavorException.java

5. Abstract Factory + Singleton

Finally, I utilized the Abstract Factory design pattern, which also inherently uses the Singleton design pattern. The Abstract Factory design pattern is a creational design pattern designed to allow for the creation of a family of objects by programming to a limited number of interfaces. The Singleton design pattern, also a creational pattern, is designed to allow for the creation of only one instance of an object. Abstract factory is most often used when the different variations of all objects can be characterized under a common theme and when all related objects belonging to the same theme can be created by the same factory class. Therefore, I thought the Abstract Factory pattern was perfectly suited to the ordering of shirts, hats, and cups, as they all fall under the umbrella of being Merchandise objects and each type of Merchandise object can be created by their own respective Factory classes (i.e. ShirtFactory). First, I created an abstract Merchandise class with basic instance variables storing item type and price. Then, I created 3 subclasses of Merchandise: Shirt, Hat, and Cup. Furthermore, I created a FactoryStore class that serves as the go-between of the driver program and the factory classes I was anticipating to create. The FactoryStore class keeps track of the stocking and delivering of Merchandise objects. I then created an abstract Factory class with a FactoryStore instance variable to keep track of the

FactoryStore object we were using to order instances of Shirt, Hat, and Cup and an abstract method of deliverItem that all subclasses of Factory would have to implement. Finally, I created 3 subclasses of Factory corresponding to the 3 different types of Merchandise objects I wanted to sell.

Files involved: Factory.java, FactoryStore.java, Shirt.java, Hat.java, Cup.java, ShirtFactory.java, HatFactory.java, CupFactory.java, Merchandise.java

6. Footnotes

- Further comments can be found in the java files themselves explaining the code
- If the lines in the UML diagram I provided are unclear, here's the link to the lucidchart project so you can more freely drag around the classes to see their connections: https://lucid.app/lucidchart/invitations/accept/da0c6a96-491b-404c-a5d9-f6e0a3ba8fdf