Project 6

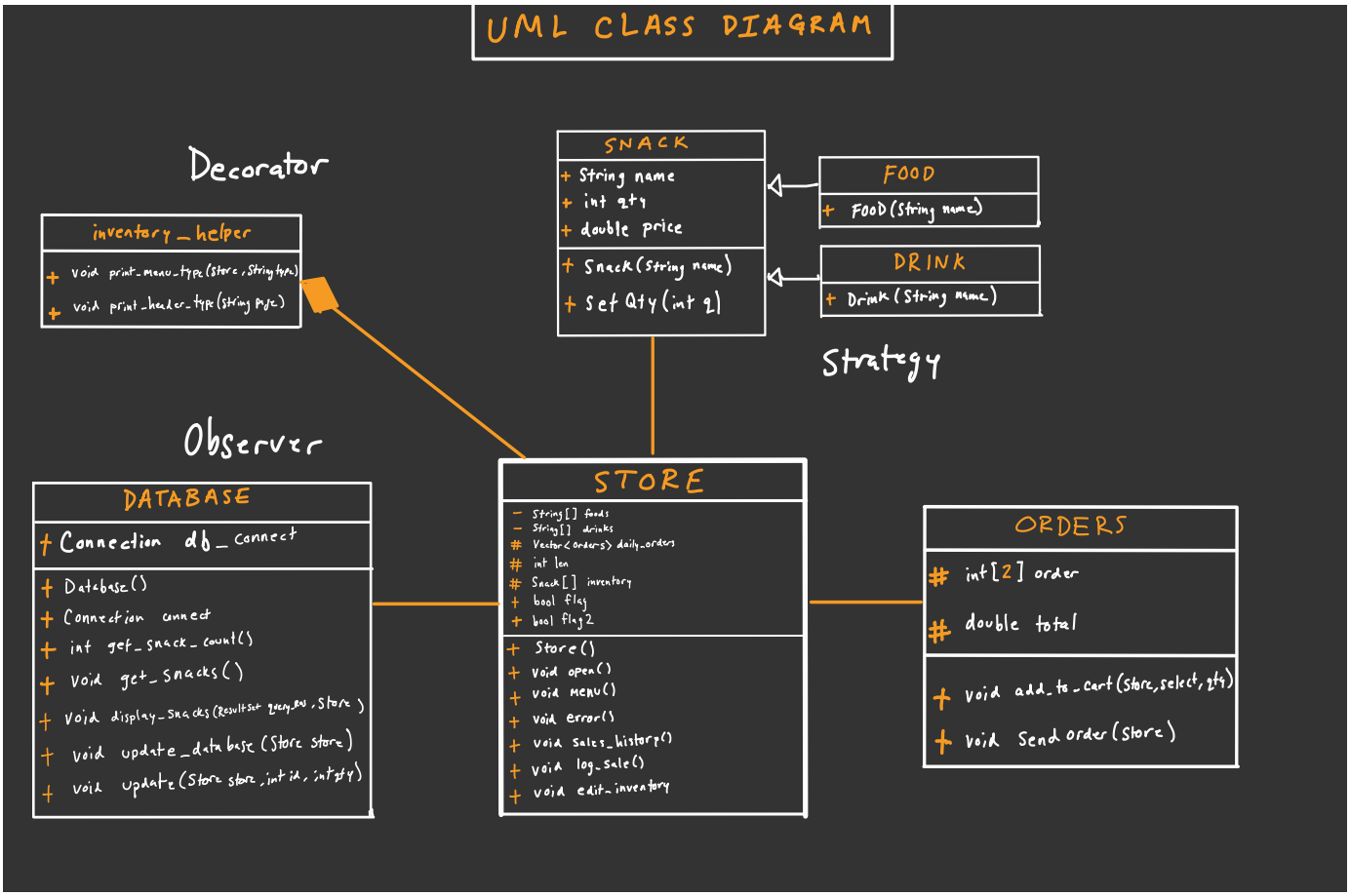
Project Name: StoreKeeper

Team Members: Max Marfuggi (Solo)

Final State of System:

The final delivery of my project was able to successfully implement all of the core features that I planned to implement in earlier design phases. One aspect I was unable to implement was a UI that I attempted to create using HTML and JavaScript, but I was unable to model my problem domain in a way that allowed me to utilize my class structure using the approach with JavaScript. In order to implement the design patterns that I needed to have my system function, I transitioned to using Java to create a console app that has all of the same features. I was able to successfully implement the sale logging, inventory editing, and sales summary features while also enabling the app to save prior session states by importing and exporting session data to a Postgres database. In summary, I was able to implement the patterns I wanted successfully, as well as the features I needed, at the cost of losing the UI due to a framework switch.

UML Class Diagram:



Pattern Use:

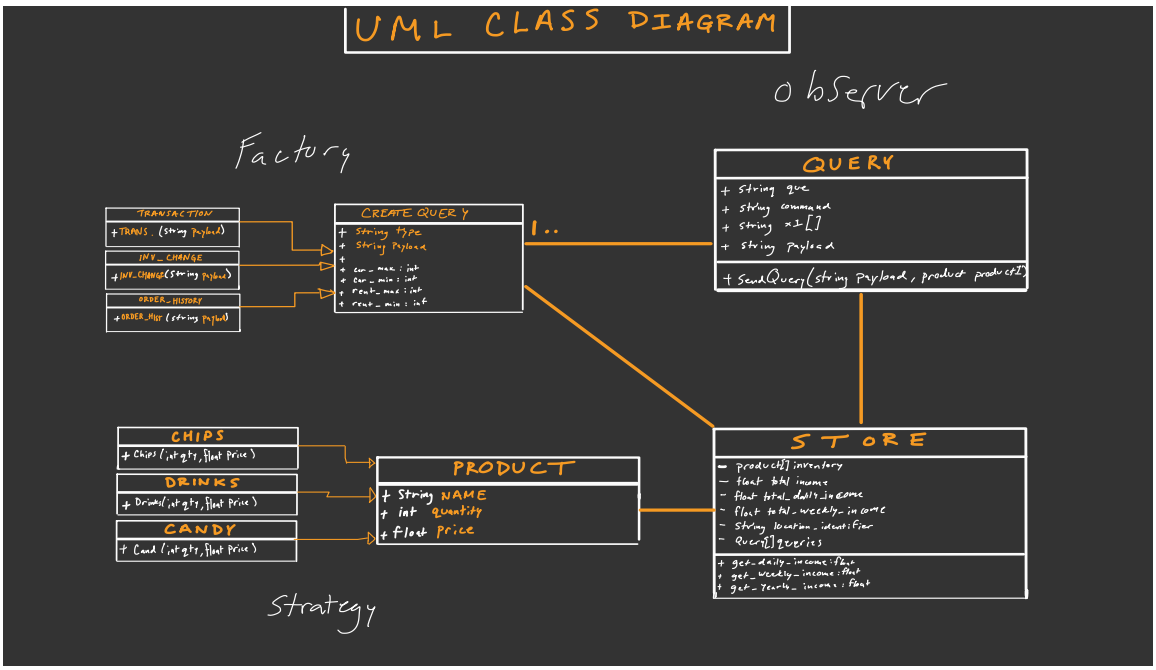
I used 3 patterns in the final delivery of StoreKeeper, which were the strategy, decorator and observer patterns.

For my observer pattern, I had helper classes that would use a method whenever a sale was confirmed to alert the store that it needs to deduct the quantities from the relevant snacks. Once this method was run, it would change the states in the store class and run the function that adds the most recent sale to the sale archive. I also implemented the observer pattern in the implementation of my database connection, where I had my database class observe my Main class for whenever the user chose to log out. Once the user logs out, a flag is raised and the database class is notified to update the database with the days sales, and to close the connection.

For the strategy pattern, I had an abstract class called Snack that had two derived classes, Food and Drink, each representing a type of item sold at the store. When the store is instantiated at run-time, each snack is created with a parameter that creates the appropriate type of Snack object, and adds it to the list of snacks. I would have liked to have a more diverse implementation of the strategy method, but I was having trouble controlling the complexity of the derived classes and their interactions with one another as I did so, so I settled with using strategy to help create each snack object.

For the decorator pattern, I needed a way to print several large menus that all changed slightly. To solve this, I created a singular print_menu function, and used a helper interface called print_helper which I used to delegate specific menus to each circumstance that was necessary while maintaining a singular print function that was simple as opposed to 9 different functions that all accomplish very similar tasks.

Project 5 UML Diagram:



UML Diagram Comparisons:

The largest changes in my system from the project 5 version were caused when I changed from an HTML/JavaScript/Postgres framework to a Java/Postgres framework. This change drastically changed the way that my system interacts with the database, so I could no longer create the 'query' class I originally planned on making. This also caused me to switch from using the factory pattern to instead using the decorator pattern to help with menu navigation, which also removed the need for the create query class. To replace the purpose that these classes served, I created the database class which is now responsible for all of the interactions with the Postgres database, and uses the observer pattern to dynamically update the database as sales are made. I was successfully able to implement the product and store classes as I intended with project 5, with additional methods to compensate for the other new classes they were interacting with.

Third-Party Code vs. Original Code:

The only third-party code I required for this project was for the process of connecting to the Postgres server from Java, which I had never done before. I used the two links below to help get a better understanding of how to connect, pull data from, and update a table in the database. All of the rest of the code in this system is original.

https://www.postgresqltutorial.com/postgresql-jdbc/connecting-to-postgresql-database/
https://www.postgresqltutorial.com/postgresql-jdbc/update/

OOAD Process Project Impact:

One positive OOAD process that I thought was really beneficial to the development of the program was the extensive planning that went into the prototype UML class diagrams. Thinking about all of the pieces that model the system and how they needed to interact gave me the ability to make the necessary changes in the framework to get the app working. It just gave me a more confident sense of direction, and a better understanding of how the system works.

One difficulty I experienced in the OO analysis phase of the project was trying to plan how to use the patterns and where to implement them. I felt at times I was just searching for a place to put an arbitrary pattern rather that recognizing which pattern was a best fit in certain places. I think this may be due to a lack of experience in knowing the situations where a pattern helps, and having control of the design made creating a schema that was conducive to using patterns made me overthink there placement as well.

Another difficulty I experienced in terms of analysis and development was understanding how to use design patterns and OO principles that were outside of the context of just console applications. I had trouble planning out what frameworks would be essential to the system, and seeing the limitations in what I had chosen before I tried to make it work. So the process of making my structural diagrams gave me trouble in the sense that I was unsure how to represent a UI feature when integrated with the program I made, while also communicating with a database. I felt like organizing the structure of the program was probably my weakest area.