Name: Cafe POS Software
Team: Marissa Bueno

Final State of System Statement:
The System is divided into two parts: the simulation and the POS Software. The simulation part of the system has been abstracted out into its own class. This part of the system facilitates the entire simulation and demonstrates how the POS system would run in a real situation. It makes use of randomness to demonstrate the different capabilities that the POS portion of the system can handle (ex: random drink orders, random kitchen orders, random number of customers). The POS side of the system is responsible for handling Transactions, Records, and more.

- Transactions - The POS software can facilitate the running of transactions. A transaction is defined as the placing & taking of an order, processing inventory accordingly, recording sales information, applying loyalty points when appropriate, and ends with the starting of the order preparation. The simulation selects a random number of customers for each day (loyal and new customers) and has them place a random order (beverages, food, and extras).
    - Random numbers of customers will visit the store daily and order random items from the menus (Beverages, Pastries, and Extras)
    - Random employees are selected to work daily
    - Cashiers take customer orders and then Chefs/Baristas fill those orders.
    - Receipts are generated as part of the Cashiers responsibilities. These receipts are added to the sales record and inventory is updated automatically.
    - Cashiers inquiry about Customer Loyalty Memberships and handle them accordingly.
- Sales Records & Inventory - As mentioned above, the cashier will log the order from the customer & create a receipt which displays the price of the order and the savings in appropriate. This gets added to the Cafe's sales record and updates the Cafe's inventory.
    - When an inventory supply gets low, the manager is notified and they perform a restock (observer design pattern).
    - As the order is logged and a receipt is created, the sales record is updated as well as the inventory after that order has been placed.
    - The updating of both of these records happens automatically as the Cashier processes the orders.
- Preparing Products - After a transaction has finished, the working Chef and the working Barista begin preparing the order.
    - Baristas brew while Chefs cook, however, they follow a similar process to filling the orders.
    - Cashiers use the OrderUp() method to begin the preparations. Each class overwrites the method according to their own specific duties.
    - They both serve the order and announce when the order has been filled.
- A Rewards Program

    - The cashier will check to see if the current customer placing the order is a Loyal Member or not -
        - If they are, then the current order is added to their loyalty points. If they have over _x_ loyalty points, then $5.00 is taken off of their purchase.
        - If they are not, the cashier will ask the customer if they would like to sign up. Based on randomness from the simulation, the customer may sign up and begin building up loyalty points or they do not and nothing changes.
        - Only Customers who are Loyalty Members are placed on the Customer Record.
- The program will work for other cafes. As a new cafe is created, it takes input in the form of a .txt file for the (1) menus, (2) a "full" inventory, and (3) employees for a specific cafe.
    - These text files are read in and the cafe is then set up accordingly.
    - For example, one Cafe may not serve Tea while others do.
- Payroll and ORM persistence were not implemented. This is linked back to the time constraints of the project and the value that each would bring. I didn't think an ORM would be needed for this specific application. The Payroll system would be the next feature I would work towards if given the time.

Final Class Diagram and Comparison Statement
See UML Diagrams.

Third-Party code vs. Original code Statement
All code was created on my own with help from the Head's First Textbook used in class. No Third-Party code was implemented in my project.

Statement on the OOAD process for your overall Semester Project
Principles:

- **Encapsulation**: I have made use of this OOP principle by using private instance variable and public accessor methods.This ensures that important data like Customer names, Employee names, and Sales Records are kept private.
- **Abstraction**: Abstraction is used in this system through the use of Abstract Classes & Abstract Methods. This allows different parts of the system to easily interaction with other parts of the system without needing to know the specifics behind the implementations. For example, the `Cashier` class does not need to know how the `Barista` and `Chef` create `Products`, it just needs a common way to let them know that there is a new order they need to prepare so they just call the `OrderUp` abstract method that `Barista` and `Chef` implement it as being a `KitchenEmployee`.
- **Inheritance**: Inheritance is used throughout the system. You see it in `Employees`, `Products`, and `Records`. All of these classes have some type of an "is-a" relationship with another class.
    - Barista "is-an" employees

    - Muffin “is-a” pastry
- **Polymorphism**: This principle is used in the system by the use of code that works on the superclass which allows the code to work with any subclass type as well. For example, every `Beverages` and `Pastries` are subclasses of `Product`, so when we the cashier is dealing with the specifics on an order - they can just write code for a `Product` that works with all of its subclasses.

Design Patterns:

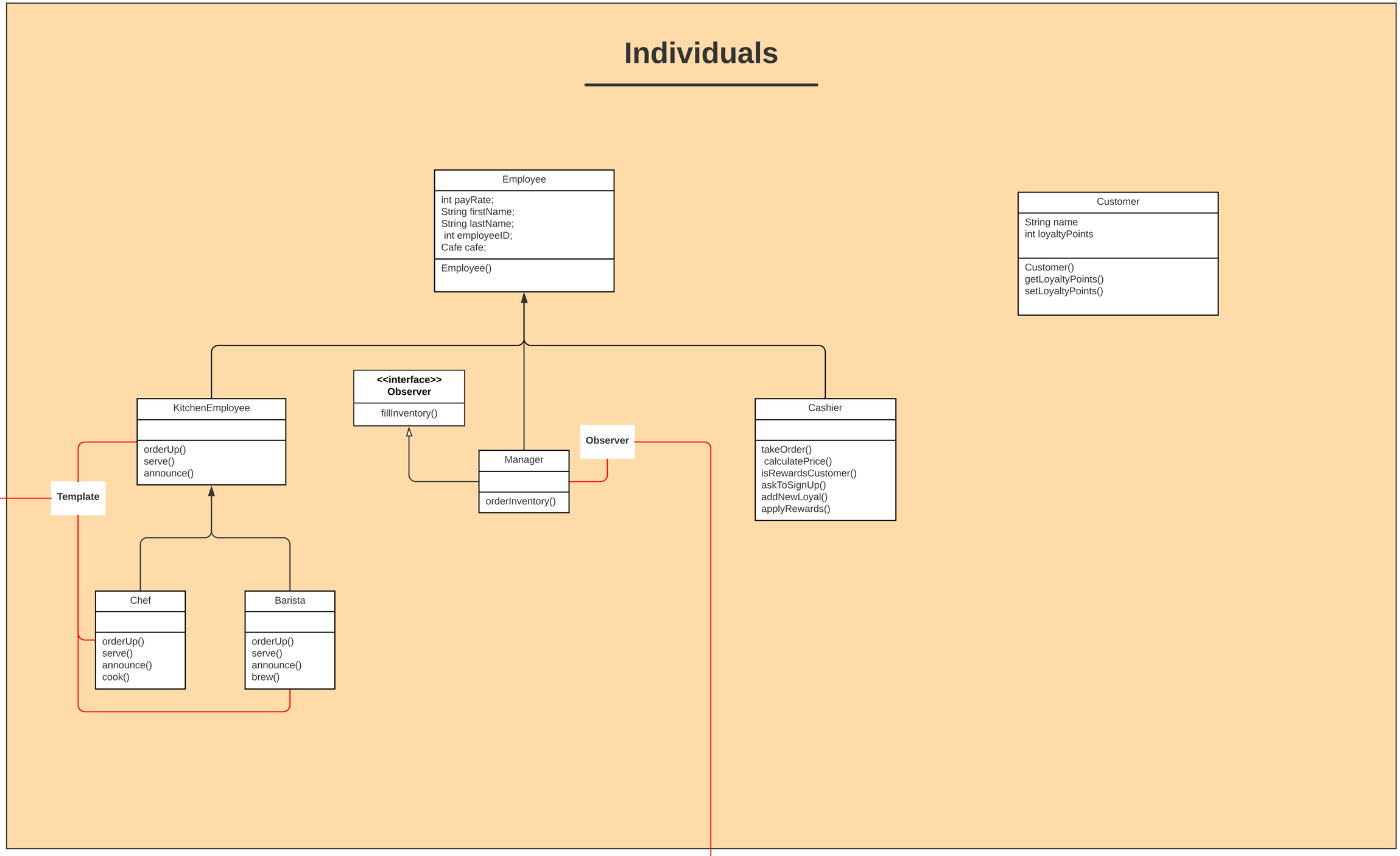
- **Factory**: The factory pattern is used to instantiate `Products` without exposing the instantiation logic to the client. The `Barista` or `Chef` class will ask the appropriate `Stores` to create a `Product`. The `Stores` then use a specific `Factory` which has the `CreateMethod()`. The `Factory` is related to the Abstract Class `Product` which has Concrete `Product` subclasses.
- **Command**: The Command Pattern is used in the transactions portion of the system. The `Order` implements the `Command` interface, the `Cashier` is the Invoker, and the `Chef/Barista` is the receiver. The pattern decouples the `Cashier` which invokes the operations from the `Chef/Baristas` (the ones that know how to perform the action). The `Chef/Barista` know what to do once the `Cashier` invokes the command as all of the necessary details are kept in the `Order` object.
    - As a walk-through: the `Cashier` takes an order and when it is time, will call the `orderUp()` method to begin the order’s prep; the `Order` has everything that the `Chef/Barista` needs to produce the `Customer’s` order. Once the command object `Order` has been executed by the `Cashier`, the `Barista/Chef` perform their actions as a receiver.
- **Decorator**: The Decorator pattern is used in the `Topping’s` feature. `Customers` have option to add extras into their `Beverages`. The `BeverageDecorator` extends Beverage (to have same type) and provides a flexible way of attaching additional responsibilities at runtime. The original `Beverage` is encapsulated and decorated by objects like `Whip Cream`. When calculating the price of a `Beverage`, delegation is used. Each decorator “has-a” variable that holds a reference to the original `Beverage`; when `cost()` is called, we use this reference to find the total price of the decorated `Beverage`.
- **Template**: I chose to use a template method pattern since the process for `Chefs` and `Barista` were very similar, but behaved differently in certain parts. There is a common set of steps that both `Employees` follow to prepare orders, however, the subclasses are allowed to override specific steps of the algorithms so that they perform the respective duties.

Challenges:

One of the bigger challenges that I struggled with was the Command design pattern. One decision that I was back and forth on was which class I wanted to have create the products. In reality the Chefs and Baristas are making the literal products, however, I wasn't sure if it would make sense to have them also create the Objects for the program. I ultimately decided to have the Chef and Barista instances create the Products. My thought process for this was to have a virtual Cafe experience in a physical place; The idea would be to have a live board in the cafe that would constantly update so that Customers could watch the status of their orders while they are being made. In practice this may be slightly more work for the KitchenEmployees, but all they would need to do is note when they begin to work on a new order and when they are finished with the order. I think an idea like this would be interesting for many other contexts than just a Cafe, like a Restaurant or Bar. You could input your order through a mobile device and watch as your order is being processed live.

UML Class Diagram - Individuals

Bueno | December 7, 2019

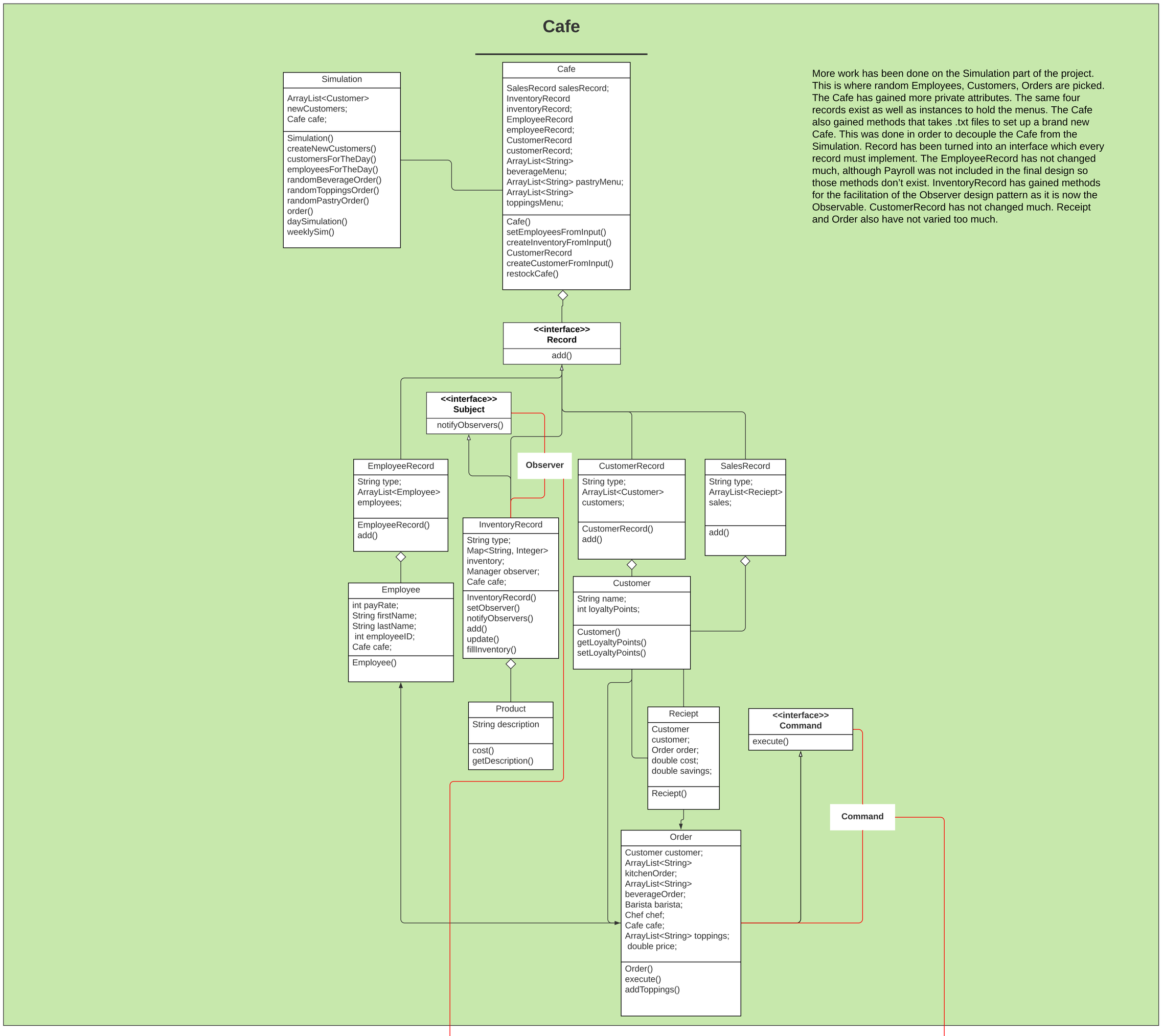


Template: The Chef and Barista classes perform similar duties, however, the way they prepare their specific Products varies slightly.

Observer: The Manager class is implements the Observer interface. This class gets notificed whenver the Cafe

UML Class Diagram - Cafe

Bueno | December 7, 2019



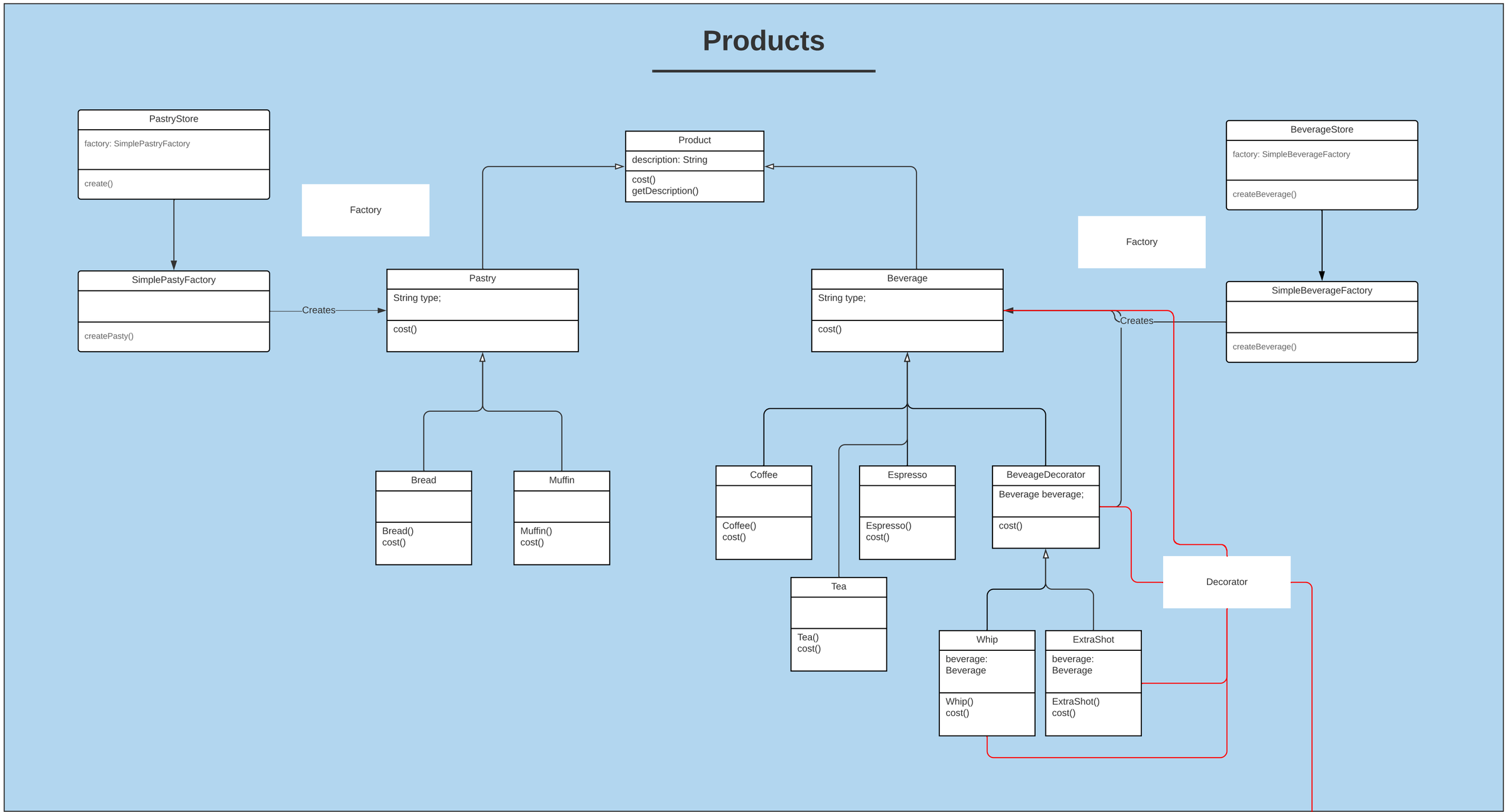
More work has been done on the Simulation part of the project. This is where random Employees, Customers, Orders are picked. The Cafe has gained more private attributes. The same four records exist as well as instances to hold the menus. The Cafe also gained methods that takes .txt files to set up a brand new Cafe. This was done in order to decouple the Cafe from the Simulation. Record has been turned into an interface which every record must implement. The EmployeeRecord has not changed much, although Payroll was not included in the final design so those methods don't exist. InventoryRecord has gained methods for the facilitation of the Observer design pattern as it is now the Observable. CustomerRecord has not changed much. Receipt and Order also have not varied too much.

Observer: The InventoryRecord class is implements the Subject interface. This class notifies the Manager whenver the Cafe runs out of stock for a Product.

Command: The Order class acts as a Command Object which is invoked by the Cashier onto either the Barista/Chef which acts as the Reciever object. The Order class implements the the Command interface therefore it must define an execute method.

UML Class Diagram - Products

Bueno | December 7, 2019



Factory: Factories are used to decouple instantiation of Products away from the rest of the system. This allows for creating objects without exposing the creation logic to the client and provides a common interface for Chefs/Baristas to create objects.

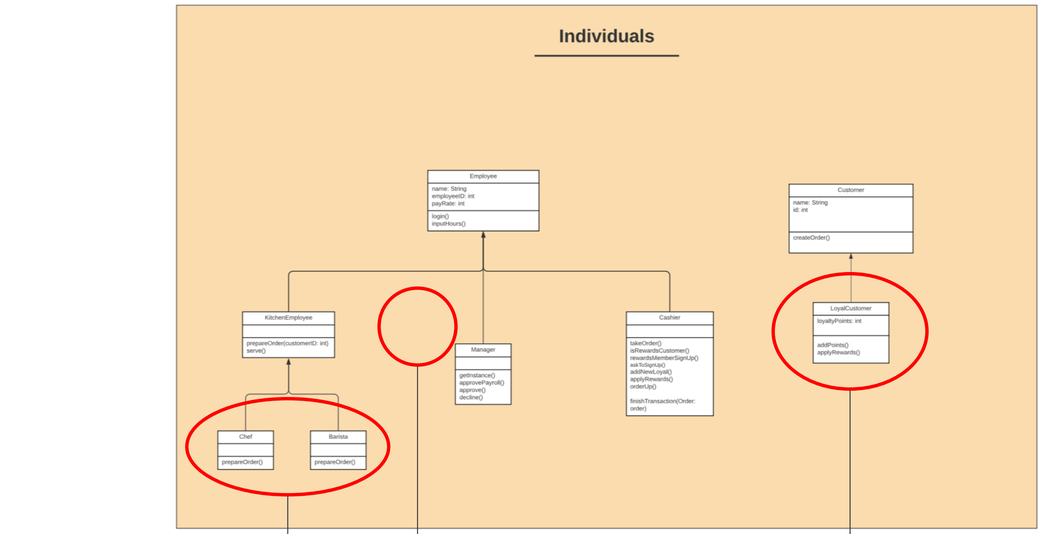
Decorator: The BeverageDecorator class acts as a Decorator for Beverage so that additional toppings could be added at runtime. This allows for Beverages to be wrapped with toppings such as Whip or an ExtraShot.

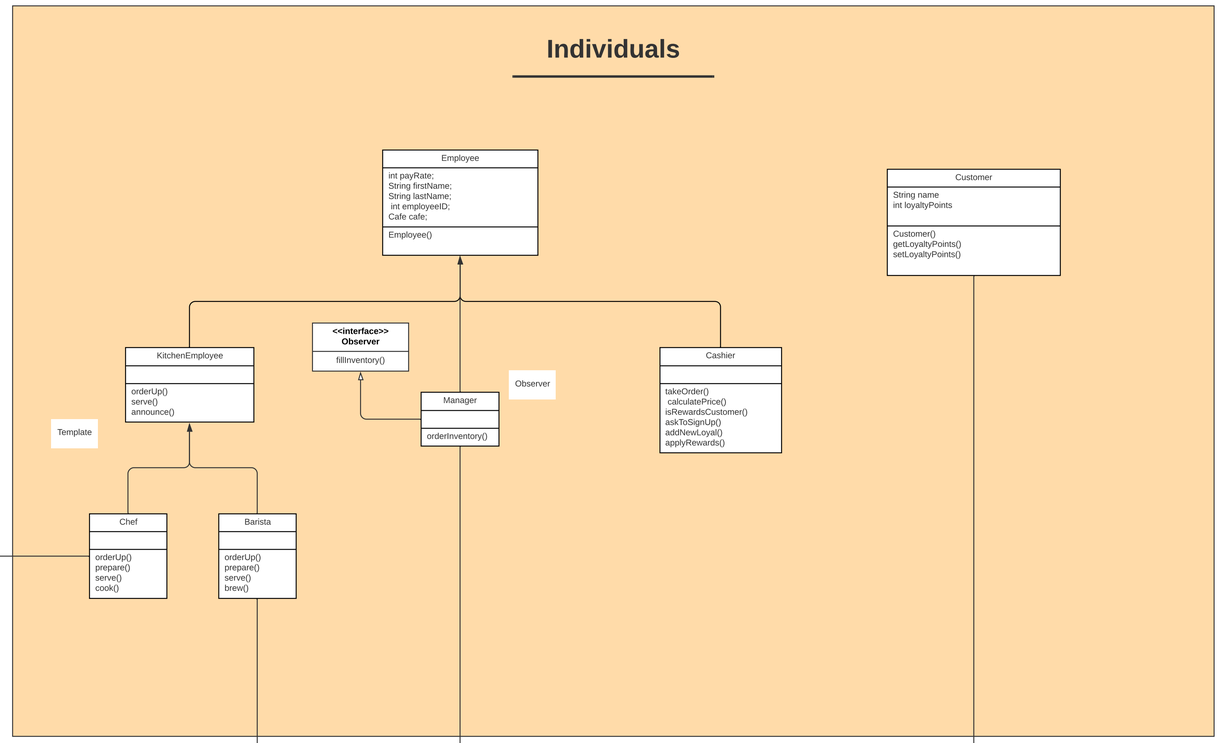
UML Class Comparison - Individuals

Bueno | December 7, 2019

UML Class Diagram - Individuals

Bueno | November 3, 2019





More methods were added to each KitchenEmployee. This is where the template design pattern is implemented.

Manager now inherits the Observer interface and is notificed whenever the Cafe needs to restock.

The LoyalCustomer has been removed as the System will only deal with Customers who are Loyal Members.

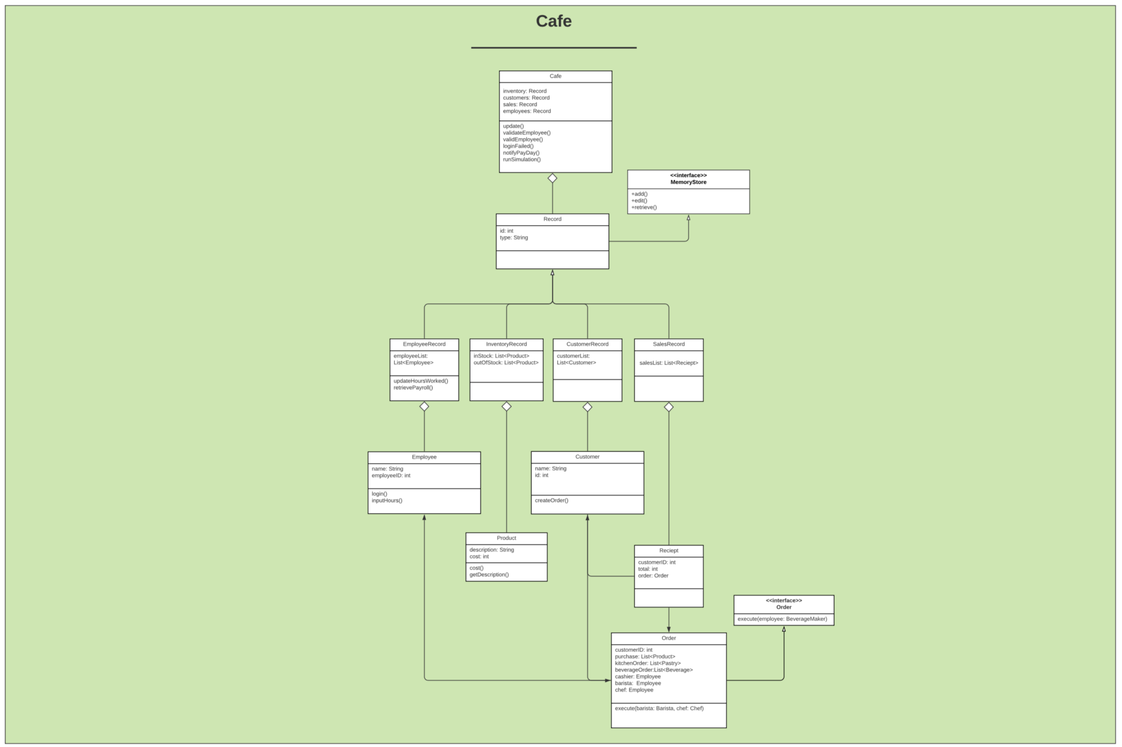
Key Changes: Minor changes include the addition of methods and attributes that were needed as implementation progressed, however, the major design decision changes are noted above. The payroll feature was not included in the final system as planned originially so those methods as well as those related attributes were not implemented. However, the manager class now has newly added functionality to handle restocking the cafe which was not originally planned. I also chose not the make the Manager class a Singleton. This decision came about after seeking advice from TA's and the potential it caused in breaking the system. The KitchenEmployee class and its two subclasses demonstrate the Template Design Pattern due to the similarity in their methods but having a few differing parts (cooking vs brewing). The cashier class did not change too much from what was originally spec'd out. The Customer also didn't vary too much from Project 3. Methods were added to handle the Rewards Membership feature.

UML Class Comparison - Cafe

Bueno | December 7, 2019

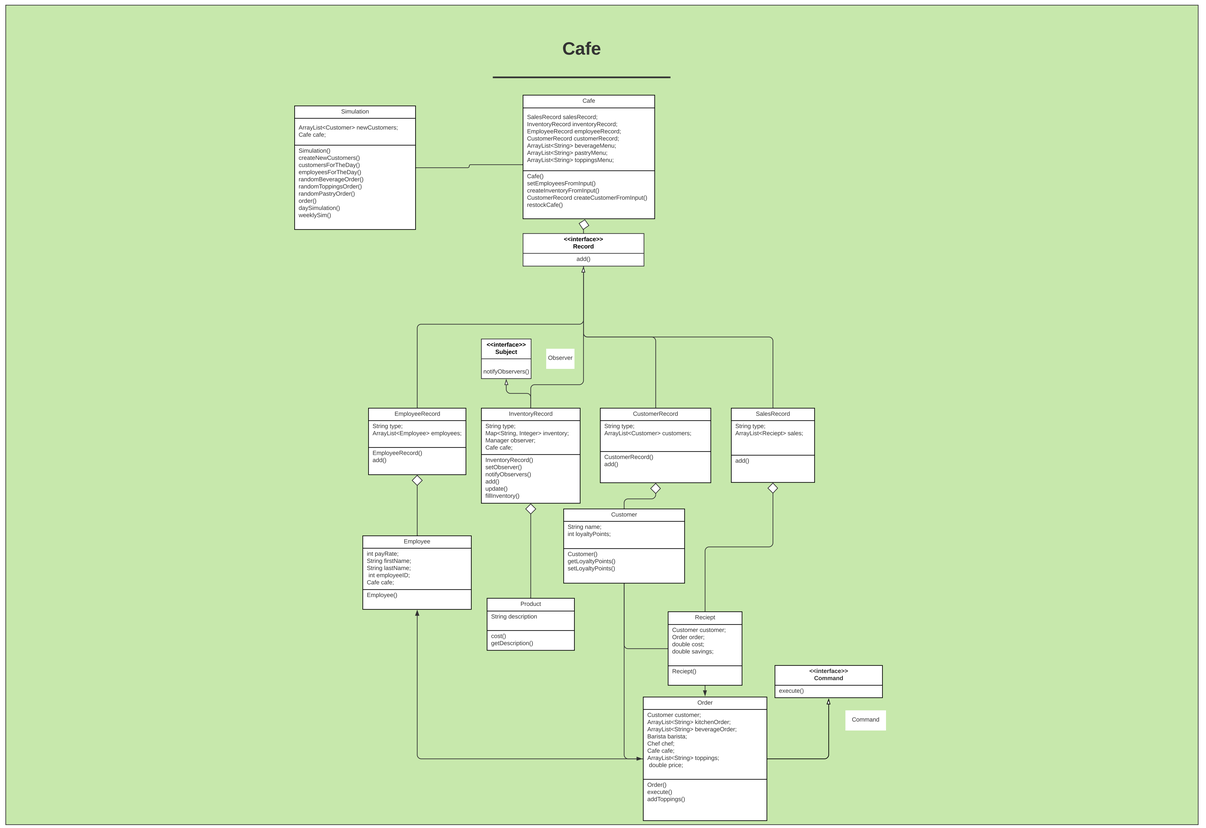
UML Class Diagram - Cafe

Bueno | November 3, 2019



UML Class Diagram - Cafe

Bueno | December 7, 2019



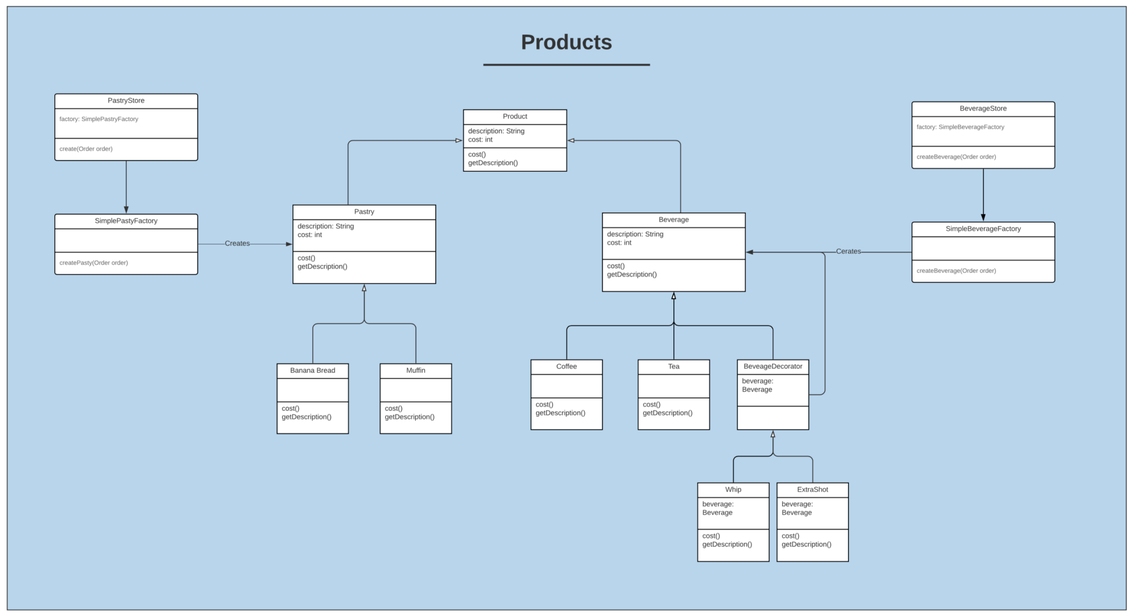
More work has been done on the Simulation part of the project. This is where random Employees, Customers, Orders are picked. The Cafe has gained more private attributes. The same four records exist as well as instances to hold the menus. The Cafe also gained methods that takes .txt files to set up a brand new Cafe. This was done in order to decouple the Cafe from the Simulation. Record has been turned into an interface which every record must implement. The EmployeeRecord has not changed much, although Payroll was not included in the final design so those methods don't exist. InventoryRecord has gained methods for the facilitation of the Observer design pattern as it is now the Observable. CustomerRecord has not changed much. Receipt and Order also have not varied too much.

UML Class Comparison - Products

Bueno | December 7, 2019

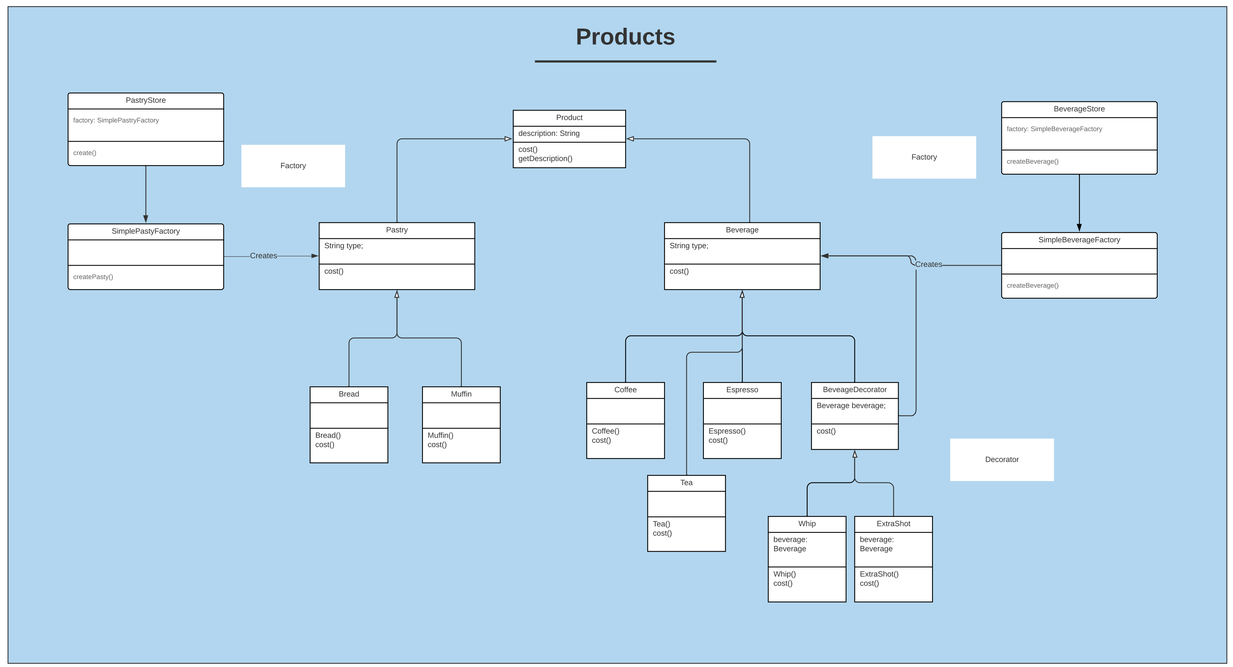
UML Class Diagram - Products

Bueno | November 14, 2019



UML Class Diagram - Products

Bueno | December 7, 2019



Overall, the Products aspect of the project have not changed significantly, and they really weren't expected to change much. The two factories still exist for the