Name - Mridul Bhatia
ID - 2019B3A70410P
Name - Hari Sankar
ID - 2019B3A70564P'

GitHub - https://github.com/Hari-01/OOP---Final_Restaurant_Management-_System
(has all the java files)

Video -
https://drive.google.com/file/d/15IEjZgw0Hu9ALCHHdXGUgKuLbAoxIaNR/view?usp=sharing

Classes and their descriptions

1. Menu- This class contains three maps which map the item names, prices and the preparation times of each item to a code respectively. It also contains the getter and setter functions for each. The simpleTable() method is used to display the menu. The addItem method is used to add items by the admin when called.The PriceList class had to be used for knowing the prices as per the initial guidelines of the project and it had to be extended by the Menu class. **However, the prices have been added to the Menu class itself so there was no need for the PriceList class**. **Pls note this.**
2. CookState- This class is used to allot the dishes for preparation to the cooks. This class also handles whether a given order is used in taken state or preparing state or served state. There are 5 cooks and the dishes are allocated in such a manner that the dish which takes most time is given to the cook who has the minimum remaining time to complete the dish previously allocated to him. For each order min_prep_time and final_prep_time are calculated. min_prep_time is the time from ordering for the first dish in the order to start being cooked. Final_prep_time is the time from ordering for the last dish in the order to get ready. These 2 variables are calculated to find out in which state the order is present. Like if the time difference between the user checking the state of order and the user ordering is less than than mmin_prep_time it will be in taken state if it is between min_prep_time and final_prep_time it is in in preparing state and if it is greater than final_prep_time then it is in served state.
3. SeatAllottment- This class is used to allocate the seats to the incoming customers. It has static variables to handle the number of tables of six,four and two. There are some extra chairs too. The algorithm which is used to decide how the tables will be allocated to the customers goes as follows. The object gets initialised with the number of customers placing that particular order. Suppose the number of customers were 9. The efficient allocation of seats will be 1 table of six, 1 table of two and 1 extra chair. (% is the mod operation).So the number of tables of six allocated =9%6, number of tables of four allocated =((9-9%6)=3)%4 which is equal to 0 ,number of tables of two allocated= 3%2

which is equal to 1, number of extra chairs allocated=1 since the customer size is odd.The tables_left() function does this and returns true if the allocation can be done and false if it can't be done. The updateTables method updates the number of tables left in the restaurant.

4. Bill- This is a class just to encapsulate the details of the item being ordered.
5. Order- This class takes care of placing the orders. **It is composed of Bill class,** since it uses an arraylist of Bill objects to add items to a particular order with a particular order ID. The interact() method is used to interact with the customer placing the order. The user is asked to enter a code (displayed in the menu) corresponding to the item he/she wants to order. The order_print method prints the order along with the final bill amount and minimum preparation time after which you can expect the order to be served. The revenue_calculation() method calculates the revenue from that order and adds it to the revenue variable of Revenue class. The CGST and SGST rates taken are 2.5%each.
6. Revenue- This class is used to know about the revenue if the admin wishes to. The initial revenue (before any order) is hard coded to 7000 and the expenses are hard coded to 3000. The static variable revenue holds the revenue value. The getProfit() method returns the profit.
7. RestaurantState- This class **implements multithreading.** This class is used to calculate the state in which the restaurant is in. This thread is implemented first before the main thread (join method is used in main), so if this is not implemented fully the part of main thread inside the synchronised block cannot be executed. It is considered that the restaurant opens after 12pm and closes at 12 am. So if the current time is not in the given range, the thread displays that the restaurant is closed, the thread goes to sleep mode (up till 12pm) not allowing the main thread to take orders. If the restaurant is not closed, it asks for the number of members to be inputted which is used to initialize the SeatAllottment object. If the tables_left() of the SeatAllotment object returns false, it goes into full state, bringing the thread into sleep mode till the time one of the customer group is finished eating (for this the minimum time among all the customers is used). If the restaurant state is neither full nor closed, it is in an open state allowing the main thread to take orders.
8. Admin- This class allows the admin to add items to the current menu and also change the price of an item in the menu.
9. DisplayServed- This class also **implements multithreading.** This class is used to notify the customer that his/her order is served. It's object is initialised with the time after which it should notify that the order is served i.e the time after which the order is ready , i.e the minimum preparation time as mentioned in the bill printed. For that particular time, the thread goes to sleep mode. As soon as the thread wakes, it displays.
10. SeatUnoccupied- This class also **implements multithreading.** This class is used to judge when the seats which were occupied are made unoccupied. An assumption is made that the time taken by the customers to eat their ordered items is 15 min. As soon as the order

is served (notified by the thread of DisplayServed class), this thread starts. For fifteen minutes it goes to sleep mode after which it adds those occupied seats to the pool of unoccupied seats using the sixTablesUsed, fourTablesUsed,twoTablesUsed and extraChairsUsed variables of the SeatAllottment object.

11. Main- This is the main class of the project. It maintains the list of orders as per their IDs and the list of the cook states. It displays two types of menu. One for the customer and one for the admin. In order to access the menu for admin, a password has to be provided. The password for accessing the menu is Hello@123. The admin menu allows you to choose between the options of changing the price of an item, knowing the revenue and adding the item (one at a time)to the menu. Accordingly, the instructions are provided. For the menu of the customer, there are options for either placing the new order or taking an update on the previous placed order. Accordingly the instructions are displayed. It should be noted that an assumption has been made that whenever the program starts, the order placed first is the first order and there are no previous orders. For placing a new order, follow the displayed instructions. When the number of members is entered, the thread in the RestaurantState first checks if the seats are available. If the seats are available, it allows for the placement of order, otherwise the main thread can take order when the seats get vacant.As soon as the order is placed, the cooks are allotted the time accordingly, the bill is displayed and the customer is notified as soon as the order is served due to the threads working in the background.

Analysis of the code on design principles

1. **Favour composition over inheritance**
    - The Order class has an ArrayList of Bill Objects. It doesn't extend the Bill Class even though it was possible. So here **we favour composition over inheritance** because alteration in the ArrayList would not bring any changes in Bill Class.
    - The DisplayServed Class and SeatUnoccupied Class have SeatAllottment objects respectively.They did not inherit the SeatAllottment class. So here also this principle holds good.
    - CookState class has an object of Order which is further used to access the variables of that order. We can manipulate those variables without any change in the order class.
    - The RestaurantState class uses a SeatAllottment object in order to access it's variables.
    - The Main Class uses the Admin object in order to modify something in Menu class which further uses getter and setter methods. It also has a RestaurantState object in order to access the state of the restaurant. It also makes use of the DisplayServed object in order to start the thread.

- However, the Order class and Admin class extends the Menu Class. Hence, this principle is violated here. It would have been a better idea to use composition here.

  So it can be observed that this principle is followed in every case whenever (except in two cases) there was a requirement of accessing the methods and variables of another class in a particular class.

2. **Program to interface and not to an implementation** :-
   - No interface/abstract class has been used in our project. So this principle is completely violated.
   - An interface is used when a behaviour is repeated over a number of classes.
   - In many classes, we have dealt with time to know about the difference between the time at which the order is placed and the current time and notify when the allocated time is over. We could have developed an interface with such a method so that this property could have been used at multiple places. Further, if there would have been a change, we had to do it at one place and not change implementations in all the classes. We would have prevented programming to an implementation.

3. **Strive for loose coupling between objects that interact**:-
   - This principle is violated extensively in the project since there have been many instances when the objects are tightly coupled.
   - For instance,the Main class uses RestaurantState Object which further uses SeatAllotment object to initialize the DisplayServed object. The DisplayServed object uses this SeatAllottment object to access the variables sixTablesUsed, fourTablesUsed, twoTablesUsed, extraChairsUsed. So there is a tight coupling between the objects.
   - One good method would have been to use an interface or an abstract class which would have contained methods to get the state of the objects and variables.

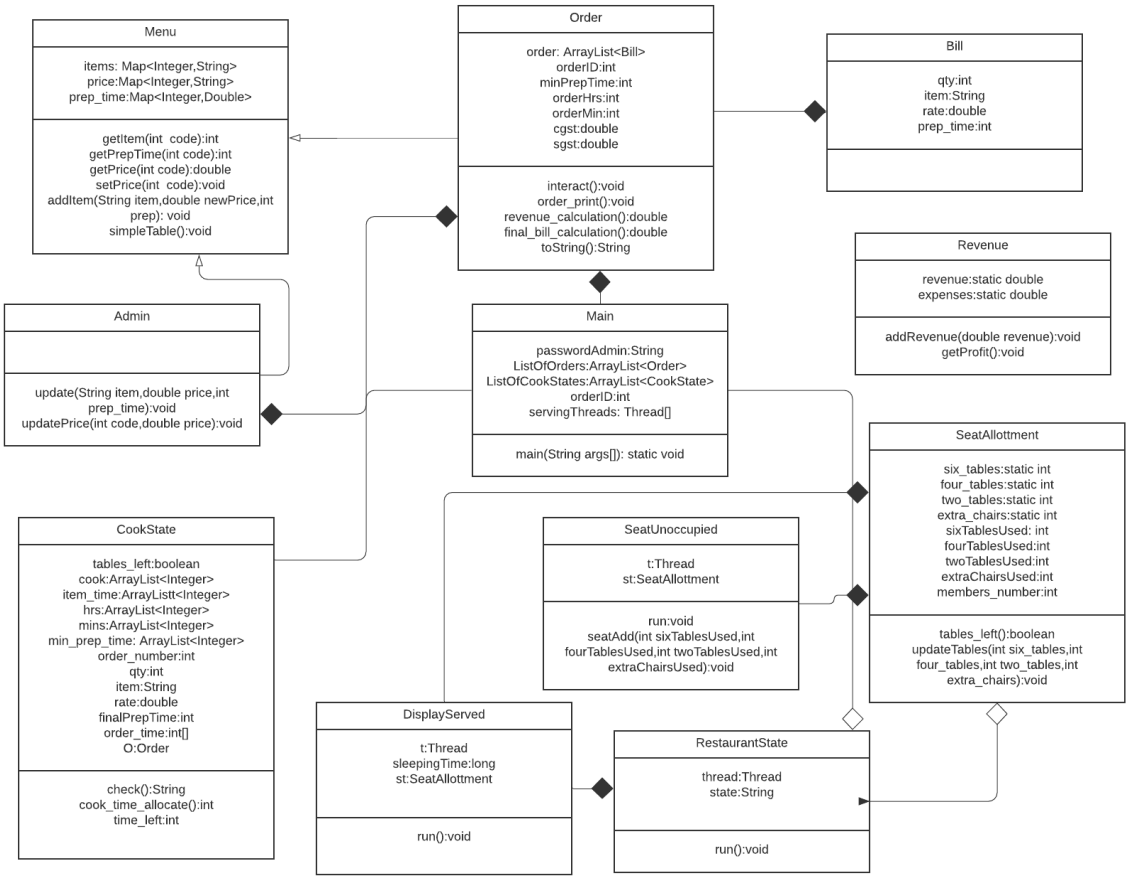
4. **Classes should be open for extension and closed for modification:-**
   - This principle is also violated in our code. The Main class itself is an example of it. Whenever we would like to add some more functionality to the code we will have to change the methods in Main class. We will have to manage the states of other objects as well.
   - Another example which is violating this rule is the CookState class. The major question arises that the number of cooks has been taken to be 5. What will happen if the number of cooks changes? We will have to modify it in the class itself, so it is open for modification.

Design Pattern Analysis -

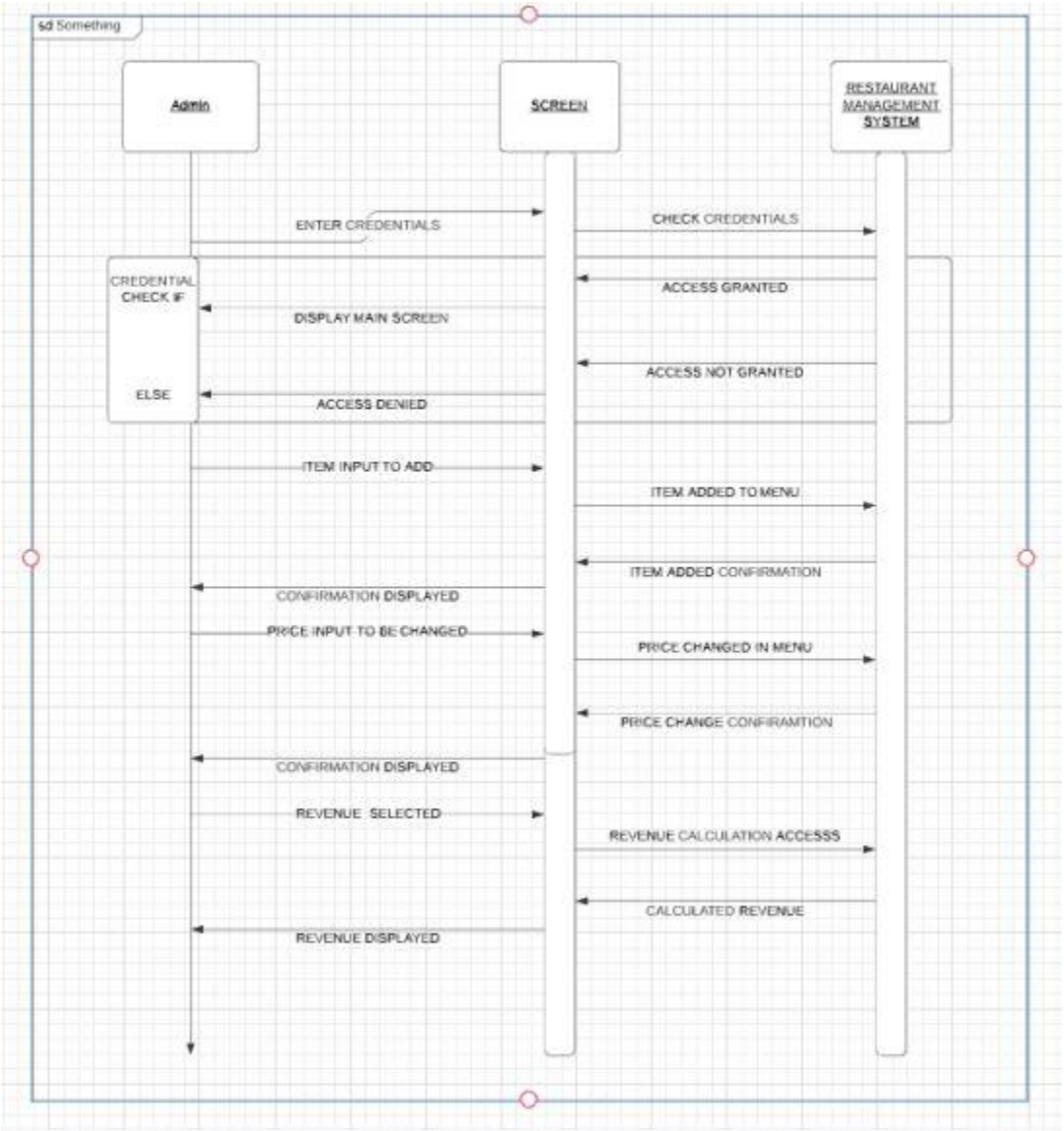
1. Decorator Design Pattern - Our code does not implement this design pattern but we could have used a decorator pattern for decorating a food class with some sidings/toppings without altering the food class. We can decrease the lines of code and add new functionality to an existing object without altering its structure by creating classes which implement an interface and each class adding a different variety of siding to the food.
   We can do this by creating a food interface. Then create a veg class which extends the food interface and overrides its methods. We can have a foodadd class which implements the food interface and gives the possibility of adding other stuff with the food already being served. For example - Creating a non-veg class gives us a chance to add roasted chicken with the food or creating a chinese class which adds manchurian apart from the food ordered or we wish to add toppings which can be of many types on the pizza. In this way we are able to wrap the original class and provide additional functionality keeping class methods signature intact

2. Singleton Design Pattern - We could create an application based on the serialization technique. Let's say we create a RestaurantSerialization class. This class provides two static methods for saving and loading a restaurant instance from/to a file In our code we have stored the data in the form of hashmap.. Interaction with the file system needs to be provided. We could create a class FileWriter which provides a method for writing to a file. This class needs to be implemented using the singleton design pattern, because each time a new instance of this class is created, some setup needs to be performed regarding the folder in which bills are to be saved. This way we are able to access the class only object directly without need to instantiate the object of the class.

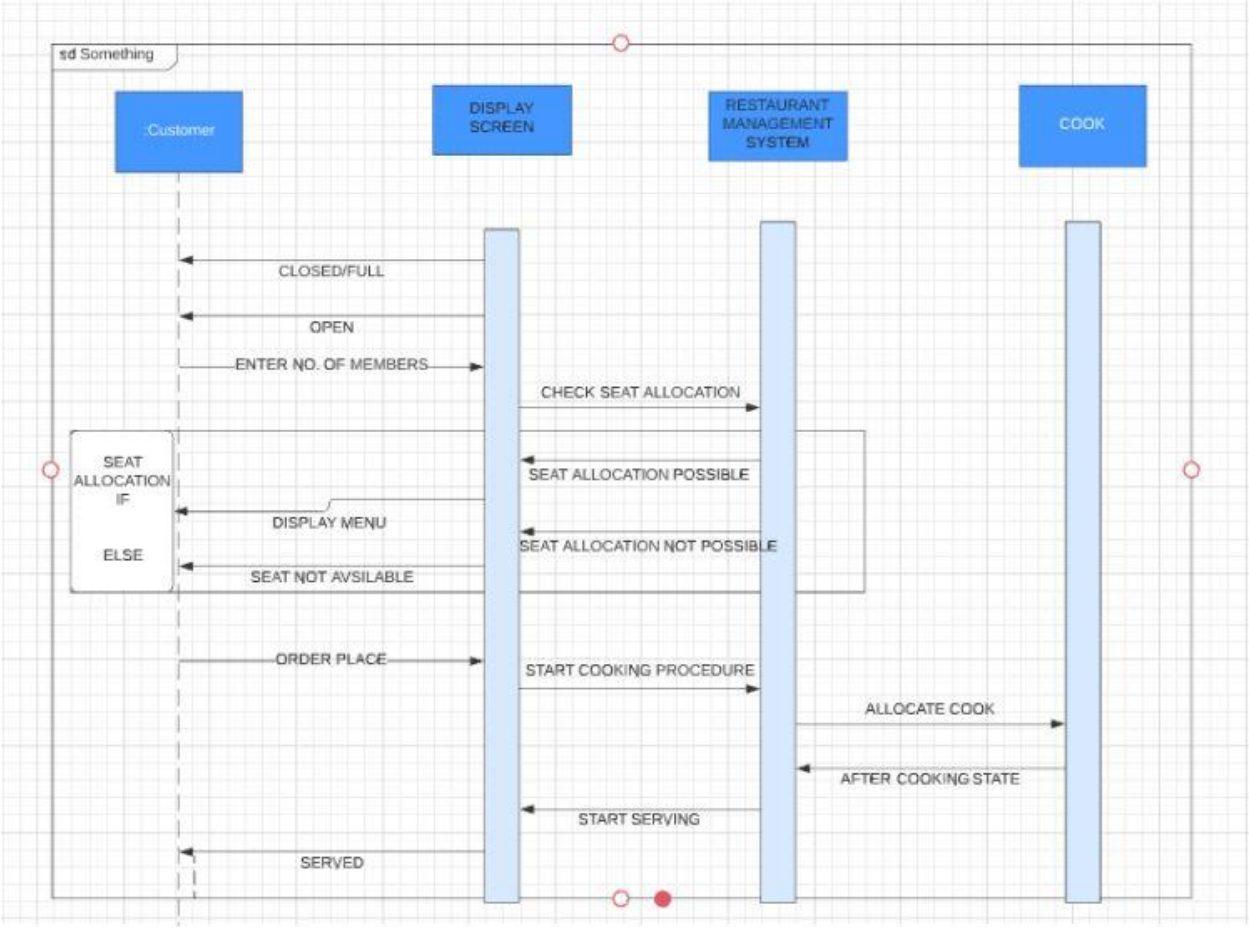
UML CLASS DIAGRAM



UML SEQUENCE DIAGRAM - ADMIN



UML SEQUENCE DIAGRAM - CUSTOMER



UML USE CASE DIAGRAM