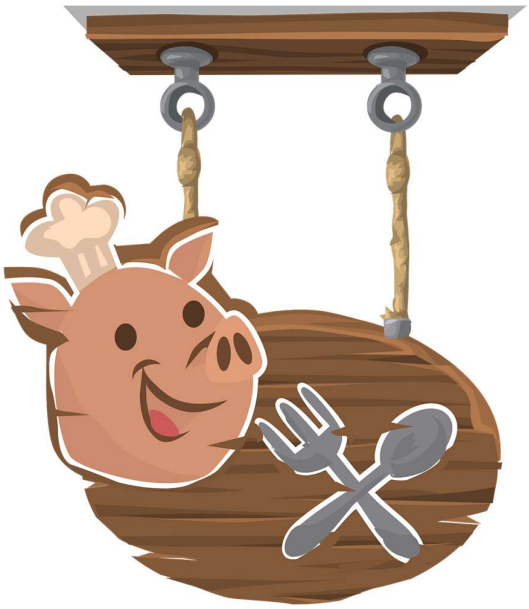


# CS3343 Software Design

## Analysis and Design Report

## Group 12

| Name | Student ID | Title |
|---|---|---|
| CHENG Yin | | Project Manager |
| SONG Tao | | Assistant Project Manager |
| LUO Peiyuan | | Programmer |
| ZHOU Junchen | | Programmer |
| DU Wenxi | | Tester |
| FENG Yong | | Tester |

**Table of contents**

## 1. Introduction

GoGoEat is a Food Court Management System aiming at various target users, including customers, merchants, and admin of the food court.

For customers, GoGoEat consists of a dine-in system and an online queuing service to allow customers to queue in the application. The online queuing service would prevent customers from getting worried about missing turns and prevent potential crowds from gathering outside the restaurant while queuing physically.

On the other hand, merchants can edit restaurants' information, including editing dish information. Merchants can also help to check customers' orders and further process cash payments through GoGoEat.

Food court admin can set the opening hours of the food court, which would restrict the available time slots for reservations, check customers' orders and reservation status, and edit restaurants' and tables' information.

The major components and functionalities of GoGoEat:

1. Customer Module
    - Login & Register
    - Dine-in (including Online Queuing)
    - Reservation
    - Food Ordering
2. Payment
    - Pay by Cash
    - Pay by Online Payment Methods
3. Merchant Module
    - Login & Register
    - Edit Dish Information
    - Check Customers' Orders
    - Cash Payment for Customer
4. Admin Module
    - Set Food Court's opening hours
    - Check Customers' orders
    - Check Customers' reservation status
    - Add / Remove Restaurants
    - Add / Remove Tables
5. Tables Management
6. Time Management

The functionalities will be further explained and illustrated in the use case diagrams and program flows.

## 2. Design Constraints

Due to the nature of this project, multiple constraints are needed to be considered when designing. Some constraints of the GoGoEat project are listed below:

- **Scope & Time Constraint**
  The project scope involves a wide range of operations for admin, merchants, and customers. It is necessary to refer to reality in many aspects and implement and improve it in formulating business logic. Hence, the project cycle is limited to 13 weeks.

- **Cost Constraint**
  To better fulfill the scope requirements of the project, appropriate designs are needed. However, the resource for the project is limited due to cost constraint. Thus, the project's design needs to help save the resource for implementing and testing.

As stated above, the general design of the GoGoEat Project is as follows:

1. Fulfil the functional requirements.
2. Use object-oriented programming techniques, pay attention to design details, apply corresponding design patterns and principles, and avoid design defects.

Due to constraints, the current design has the following weaknesses:

1. No Graphical User Interfaces, and system operation is based on the command line.
2. Not designed to connect to a real database.
3. Concurrency requirements unresolved.
4. Multi-threaded design is not implemented; hence, asynchronous operations are not supported.

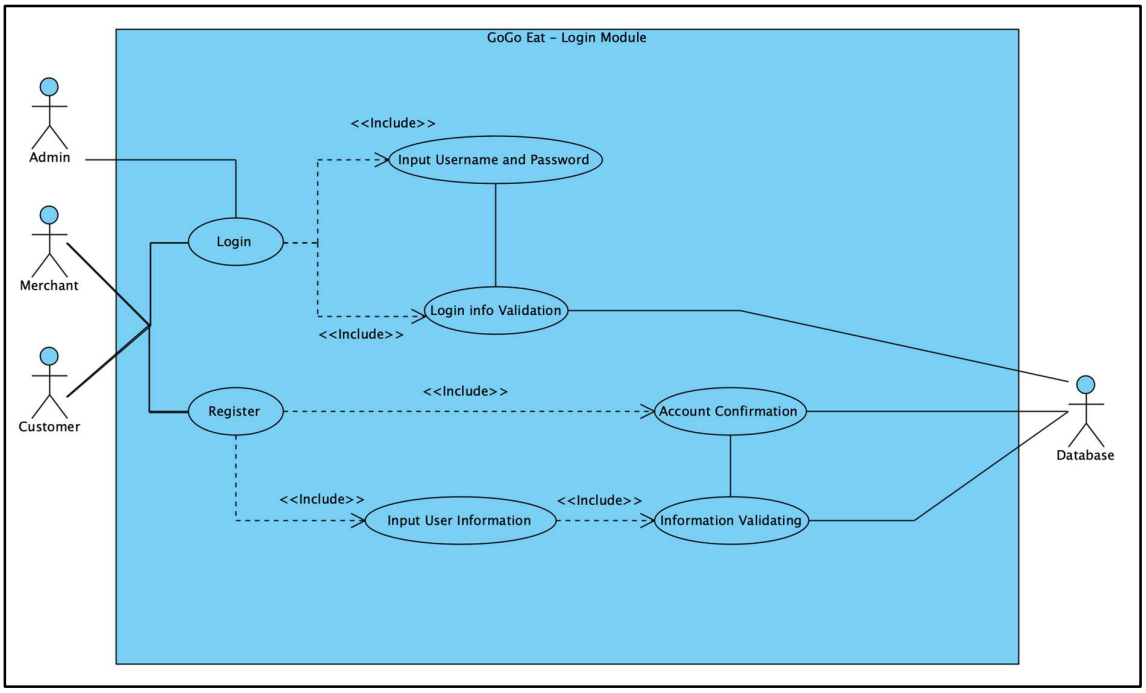
## 3. Requirement Specifications & Use Case Diagrams

### 3.1 Requirements of the stakeholder

| Stakeholder | Requirement |
|---|---|
| Admin, Merchant, Customers | Log in to users' module |
| Customers | Reserve a table to dine |
| Customers | Dine-in (Walk-in) |
| Customers | Ordering dishes |
| Merchant, Customers | Payment |
| Admin, Merchants | Check Customers' Bill |
| Admin | Check Customers' Reservations |
| Admin, Merchants | Modify Restaurants |
| Merchants | Modify Dishes |
| Admin | Modify Tables |

### 3.2 Use Case Diagrams

#### 3.2.1 Login

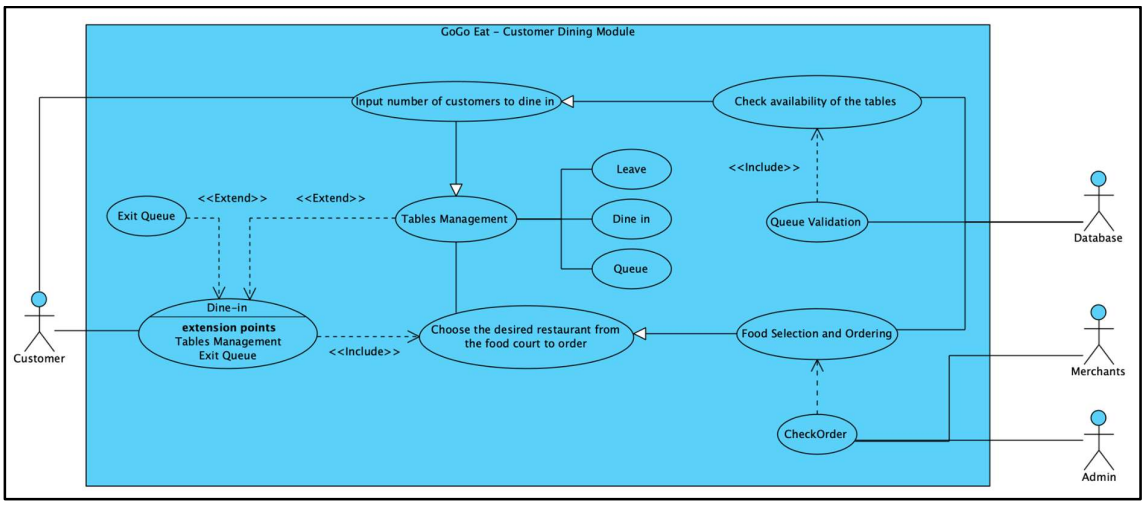



Description:

| **Use Case Name:** | **Login** | |
|---|---|---|
| **Actor(s):** | **Customer / Merchant / Admin / Database** | |
| **Description:** | This use case describes the process of admins, merchants, and customers using the login module. On completion, they will be brought to their modules. | |
| **Event:** | **Actor Action** | **System Response** |

| | | |
|---|---|---|
| | **Step 1:**<br>The user selects the login option.<br><br>**Step 2a:**<br>The user chooses to register an account.<br>**Step 2b:**<br>The user chooses to log in.<br>**Step 2c:**<br>The user chooses to delete his/her account.<br><br>**Step 3:**<br>The user input the username and password. | **Step 4a:**<br>The System validates the input information to check if it already exists. And returns a success or fail message.<br>**Step 4b:**<br>The System validates the input account information. And returns a success or fail message.<br>**Step 4c:**<br>The System finds a matching account to delete and returns a success or fail message. |
| **Postcondition:** | The user will be logged into the system. | |

### 3.2.2 Dine in operation for customer

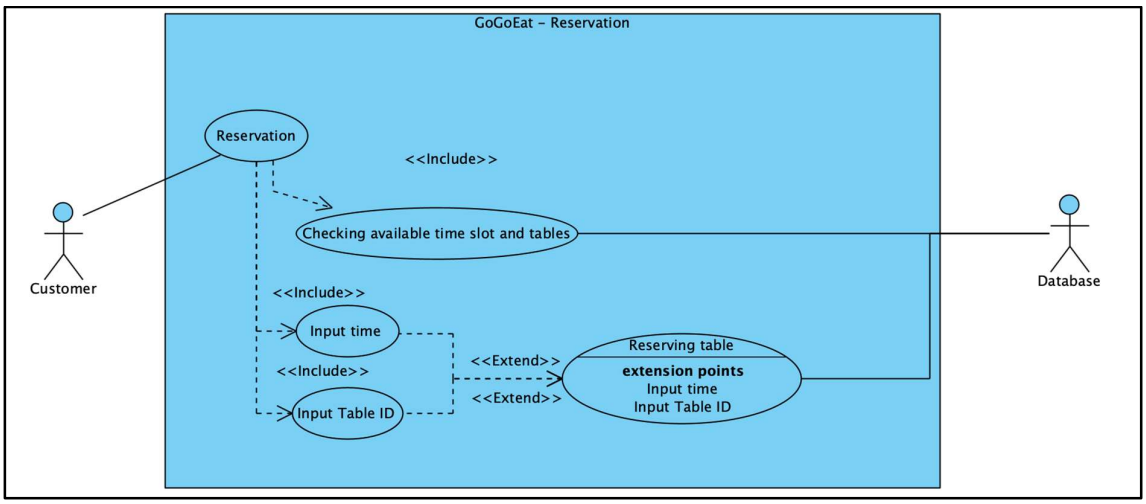



Description:

| **Use Case Name** | **Dining Options** | |
|---|---|---|
| **Actor(s):** | **Customer / Merchant / Database** | |
| **Description:** | This use case describes the process of merchants and customers using the Dining Options module with assistance from the database. On completion, customers can queue or dine in. | |
| **Events:** | **Actor Action** | **System Response** |

| | | |
|---|---|---|
| | **Step 1:**<br>The customer chooses to dine in. | |
| | | **Step 2:**<br>The system checks if the customer has a reservation and it is time for their walk-in.<br>● Reserved walk-in: Goto Step 5<br>● Not reserved: Goto Step 3 |
| | **Step 3:**<br>The customer input the number of people to dine. | |
| | | **Step 4:**<br>The system will give suggested table arrangements for the customer. If there are available tables, customers can walk in, else need to queue. |
| | **Step 5:**<br>The customers choose to walk in and select their desired restaurant. | |
| | | **Step 6:**<br>The list of the available restaurants will be shown. |
| | **Step 7:**<br>The customer can order from the menu of the restaurant chosen. Then proceed to payment. | |
| | | **Step 8:**<br>The system will output the corresponding menu of the chosen restaurant. |
| | **Step 9:**<br>The customer chooses the desired payment method and pays the bill. | |
| | | **Step 10:**<br>The system will calculate the price of the orders by matching their customer state, i.e., VIP or SuperVIP. |
| **Postcondition:** | The customer can dine in and get their meals. | |

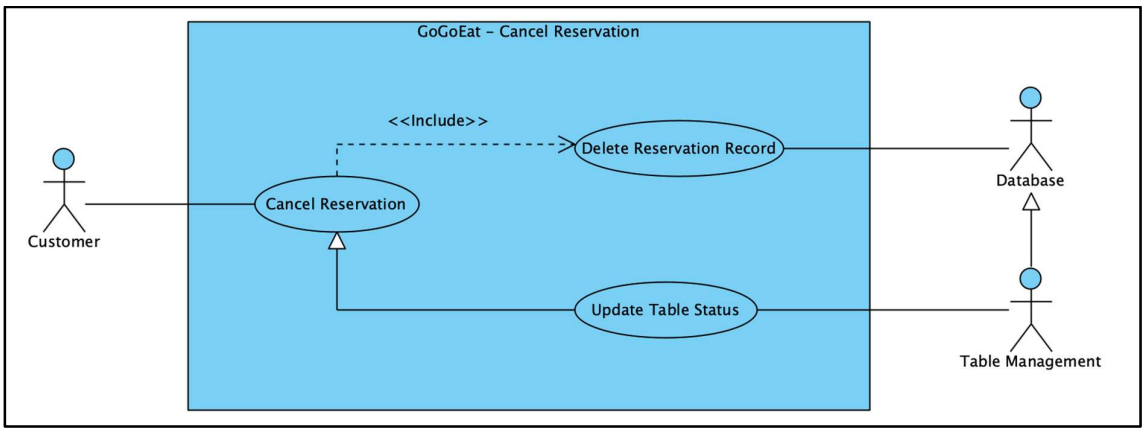
### 3.2.3 Reservation




Description:

| **Use Case Name** | Reservation | |
|---|---|---|
| **Actor(s):** | Customer / Database | |
| **Description:** | This use case describes the operation where the customer can reserve the tables in the food court for the next day. | |
| **Events:** | **Actor Action** | **System Response** |
| | **Step 2:**<br>The user input the time slot he/she wants to reserve.<br><br>**Step 3:**<br>The customer input the table id he/she wants to reserve. | **Step 1:**<br>The system lists the available time slots for tomorrow and the table IDs.<br><br>**Step 4:**<br>The system checks the validity of the inputted time slot and table ids.<br><br>**Step 5:**<br>The System will change the status of the table inputted to reserved. |
| **Postcondition:** | The reservation information, including time slot and table id, is printed out for the customers for reference. | |

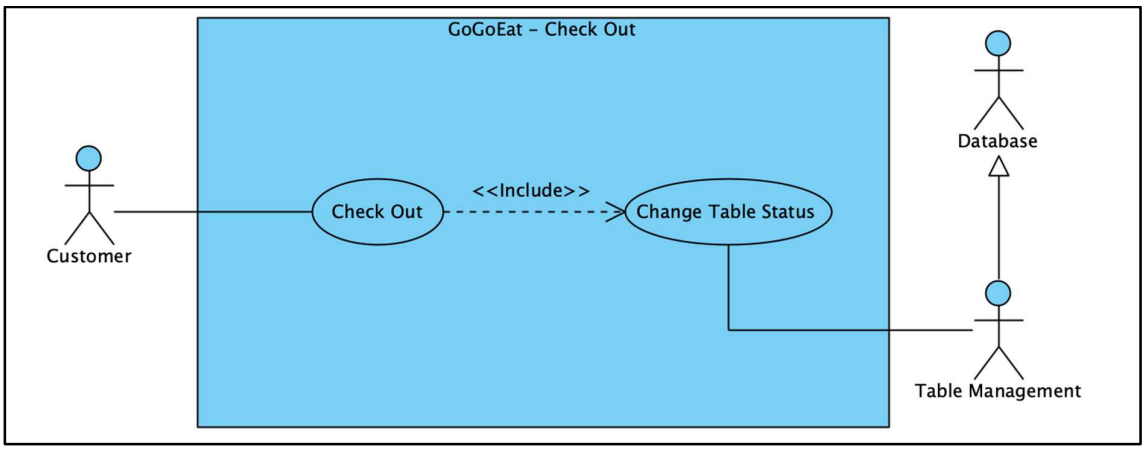
### 3.2.4 Cancel Reservation




Description:

| **Use Case Name** | **Cancel Reservation** | |
|---|---|---|
| **Actor(s):** | Customer / Admin / Database | |
| **Description:** | This use case describes the process of the customer choosing to cancel the reservation after the system shows the reservation reminder. | |
| **Events:** | **Actor Action** | **System Response** |
| | **Step 2:**<br>The customer cancels the reservation. | **Step 1:**<br>The system automatically returns the reminder of customers' reservations recorded in the database after customers log in and show a cancel reservation choice. |
| **Postcondition:** | The system returns "Cancel success!" and deletes the reservation record in the database. | |

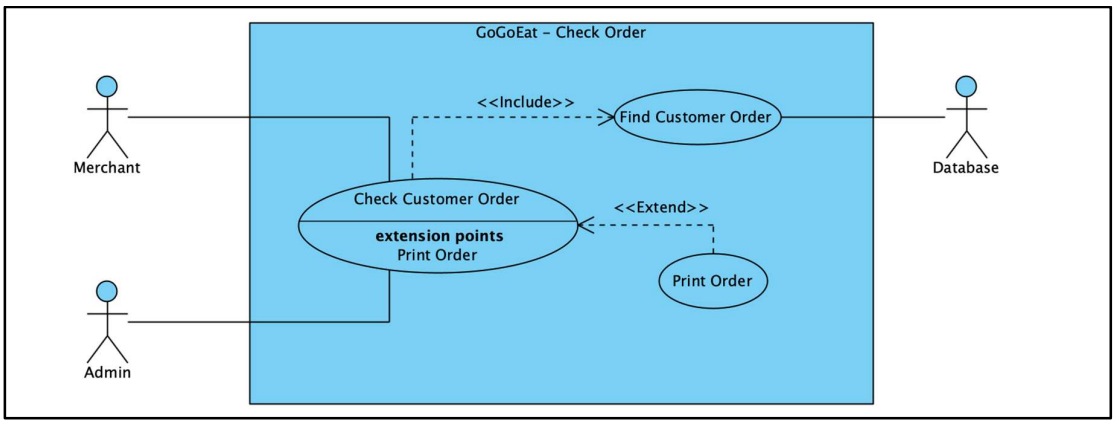
### 3.2.5 Check Out




Description:

| **Use Case Name** | **Checking Out by the customer** | |
|---|---|---|
| **Actor(s):** | **Customer, Database** | |
| **Description:** | This use case describes the operation where table status can change from occupied to available by the check-out function of the customer. | |
| **Events:** | **Actor Action** | **System Response** |
| | **Step 1:**<br>The customer chooses to check out after finishing dining. | **Step 2:**<br>The system will obtain the occupied tables from the customer.<br><br>**Step 3:**<br>The system will change the status of the occupied table of the customer to available. |
| **Postcondition:** | The customer will get the success message from the system and can leave the food court. | |

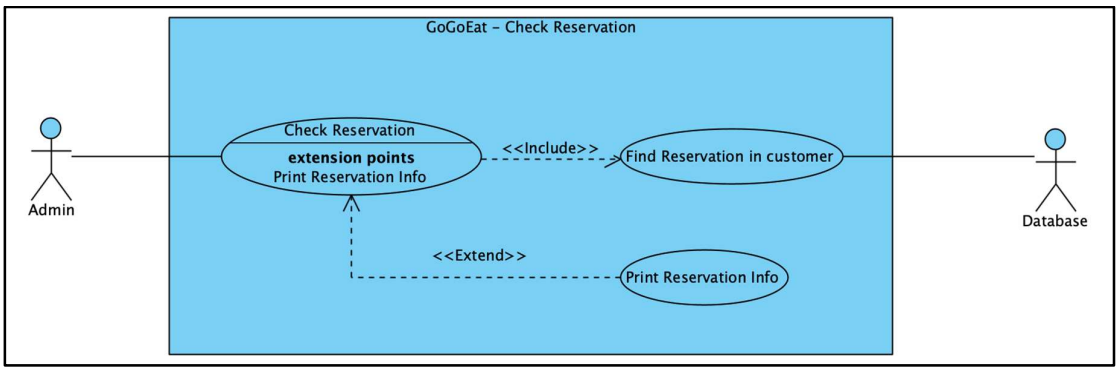
### 3.2.6 Check Order for admin and merchant




Description:

| | | |
|---|---|---|
| **Use Case Name** | **Checking customers' order by the Admin** | |
| **Actor(s):** | **Admin, Database** | |
| **Description:** | This use case describes the operation of the admin where they can check the orders made by the customer regardless of the restaurants of the orders through inputting customer id. | |
| **Events:** | **Actor Action** | **System Response** |
| | **Step 1:**<br>Admin enters the module and inputs the customer id. | **Step 2:**<br>The database will find a matched customer instance from the id.<br><br>**Step 3:**<br>The system will check if there are any orders made by the customer and print the information for the admin. |
| **Postcondition:** | The admin can obtain the orders made by the customer by the customer id. | |

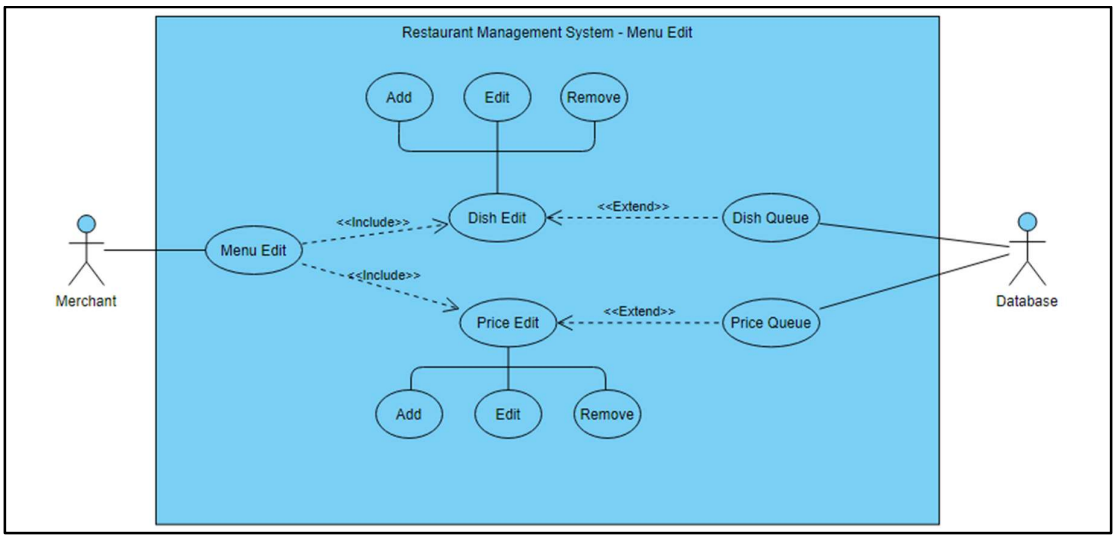
### 3.2.7 Check Reservation




Description:

| **Use Case Name** | **Check Customers' Reservation by the Admin** | |
|---|---|---|
| **Actor(s):** | **Admin, Database** | |
| **Description:** | This use case describes the operation where the admin can check the reservation information of the customer to help them to check in for the reservation. | |
| **Events:** | **Actor Action** | **System Response** |
| | **Step 1:**<br>Admin enters the module and inputs the customer id to check the reservation information of the customers. | **Step 2:**<br>The database will find a matched customer instance from the id.<br><br>**Step 3:**<br>The system will check if there is any reservation information stored in the customer and print the information for the admin. |
| **Postcondition:** | Admin can obtain the relevant reservation information of the customer by the customer id. | |

### 3.2.8 Dish Manipulation

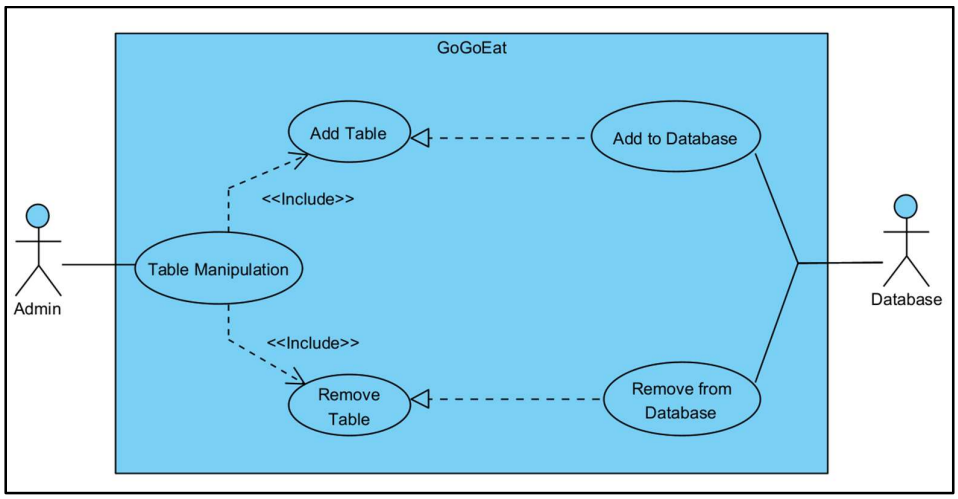



Description:

| **Use Case Name** | **Dish Manipulation by the merchant** | |
|---|---|---|
| **Actor(s):** | **Merchant, Database** | |
| **Description:** | This use case describes the operation where merchants can edit dish name or price, add dish, or remove dish from the menu of the restaurant owned by the merchant. | |
| **Events:** | **Actor Action** | **System Response** |
| | **Step 1a:**<br>Merchants choose to edit the dish name or price, and input the numbering of the dish.<br>**Step 1b:**<br>Merchants choose the add dish operation,<br>**Step 1c:**<br>Merchants choose the remove dish operation<br><br>**Step 3ai:** | **Step 2:**<br>The system finds a dish instance match from the inputted numbering. |

| | | |
|---|---|---|
| | Merchants choose to edit the dish name.<br>**Step 3aii:**<br>Merchants choose to edit dish price.<br>**Step 3b:**<br>Merchants input the name of the new dish.<br>**Step 3c:**<br>Merchants input the numbering of the dish. | **Step 4ai:**<br>The system changes the name of the dish.<br>**Step 4aii:**<br>The system changes the price of the dish<br>**Step 4b:**<br>The system adds the dish instance by the name.<br>**Step 4c:**<br>The system removes the dish instance by the corresponding numbering. |
| **Postcondition:** | The merchant will receive a success message if the chosen operation is completed successfully. | |

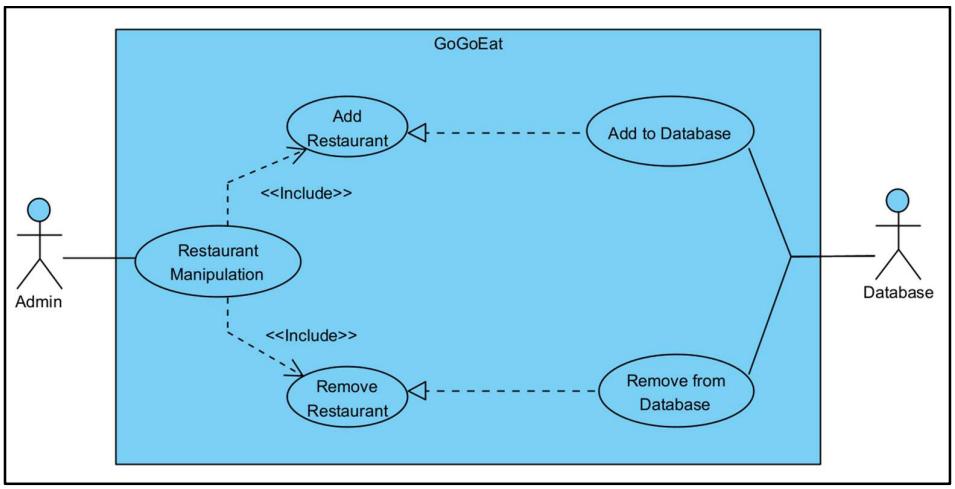
### 3.2.9 Table Manipulation




Description:

| **Use Case Name** | Table Manipulation by the Admin | |
|---|---|---|
| **Actor(s):** | Admin, Database | |
| **Description:** | Use case describes the operations of admin adding or removing tables. | |
| **Events:** | **Actor Action** | **System Response** |
| | **Step 1a:**<br>Admin chooses to add table.<br>**Step 1b:**<br>Admin chooses to remove table.<br><br>**Step 3a:**<br>The admin input the table ID and the capacity.<br>**Step 3b:**<br>The admin input the table ID of the table. | **Step 2:**<br>The system prompts the admin to input the information.<br><br>**Step 4a:**<br>The system will add the new table instance by the inputted table ID and capacity.<br>**Step 4b:**<br>The system will remove the table instance by the table ID. |
| **Postcondition:** | The admin will receive a success message if the chosen operation is completed successfully. | |

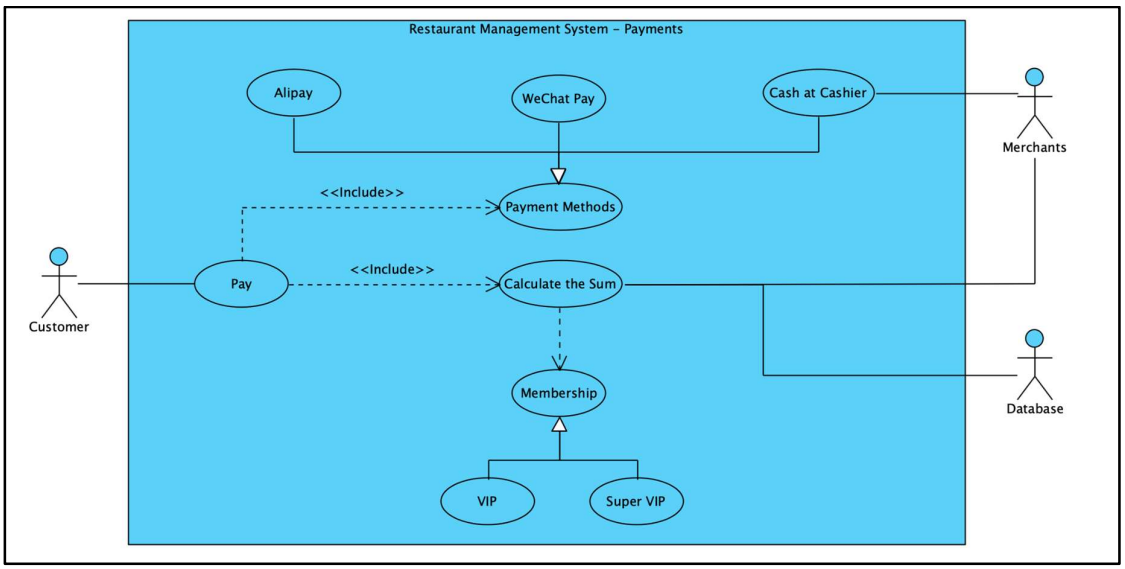
### 3.2.10 Restaurant Manipulation




Description:

| **Use Case Name** | **Restaurant Manipulation by the Admin** | |
|---|---|---|
| **Actor(s):** | **Admin, Database** | |
| **Description:** | This use case describes the operations where the admin can add restaurants or remove restaurants from the database. | |
| **Events:** | **Actor Action** | **System Response** |
| | **Step 1a:**<br>Admin chooses to add restaurant<br>**Step 1b:**<br>Admin chooses to remove restaurant<br><br>**Step 3:**<br>The admin input the name of the restaurant that he wants to add or remove. | **Step 2:**<br>The system prompts the admin for input.<br><br>**Step 4a:**<br>The database will add the restaurant instance by name.<br>**Step 4b:**<br>The database will first match the instance to the inputted restaurant name, then remove the restaurant instance. |
| **Postcondition:** | The admin will receive a success message if the chosen operation is completed successfully. | |

3.2.11 Payment



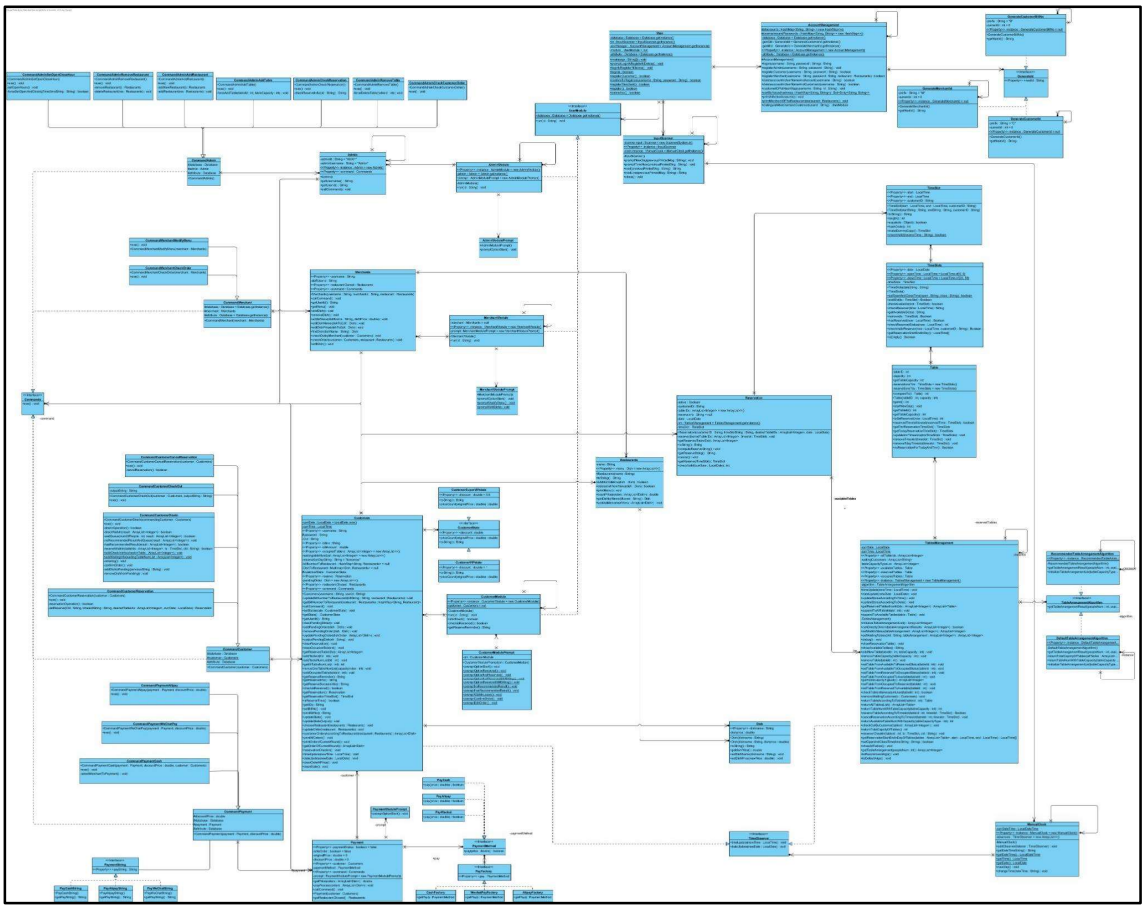

Description:

| **Use Case Name** | **Payment** | |
|---|---|---|
| **Actor(s):** | **Customer / Merchant / Admin / Database** | |
| **Description:** | This use case describes the process of admins, merchants, and customers using the payment module with an assist from the database. On completion, customers or merchants can proceed and complete the payment. | |
| **Events:** | **Actor Action** | **System Response** |

| | | |
|---|---|---|
| | **Step 1:**<br>The use case is initiated when the customer selects the payment option.<br><br>**Step 2a:**<br>The customer chooses to pay by cash. His/her dining bill will pass to the merchant by inputting the merchant id.<br>**Step 2b:**<br>The user chooses to pay with online payment methods. He/she chooses the payment method. | **Step 3a:**<br>The merchants can extract the information of the customers' bills and help to proceed with the payment by cash. By completion, return confirmation of completion.<br>**Step 3b:**<br>The customers pay through the online payment gateway. By completion, return notice of completion. |
| **Postcondition:** | The customer can pay their bills. | |

## 4. Class Diagram

The overview of the class diagram:

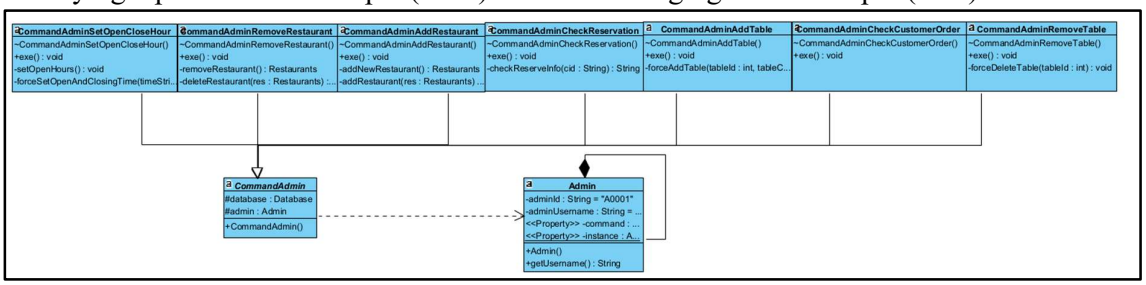


The clear PDF version of the class diagram is attached in the Doc folder, named ***ClassDiagram.pdf***.

## 5. Design Principles and Patterns

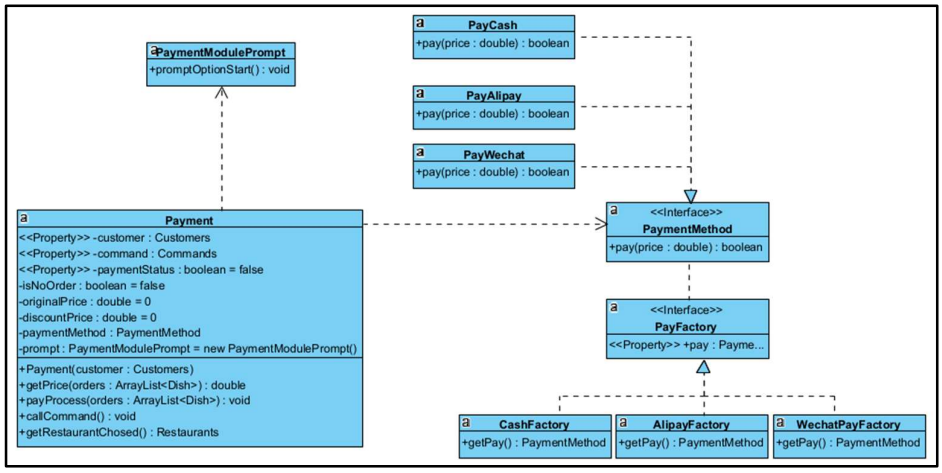
### 5.1 Design Principles

Example 1:
Satisfying Open-closed Principle (OCP) and Liskov Segregation Principle (LSP)




Description:
The admin class plays the role of the invoker, using the command interface to run commands for the admin. The ***commandAdmin*** class is an abstract class, which obtains resources that an admin's command may need, and an abstract execution function is also in it and allows its subclasses to implement. Detailed commands were then implemented by its subclasses, respectively.

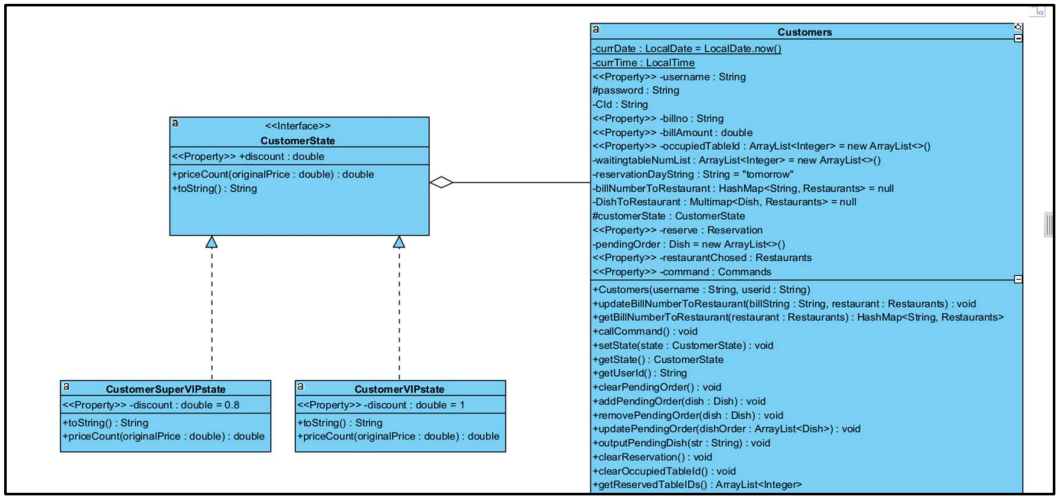
Example 2:
Satisfying Single Responsibility Principle (SRP) and Dependency Inversion Principle (DIP)




Description:
***Payment*** uses the interface of payment methods which was implemented by their single payment methods in a single responsibility. A payment factory interface was used by the payment methods interface. Three payment factory classes were used to implement the factory interface to avoid incorrect dependencies and make future extensions easier.

## 5.2 Design Patterns

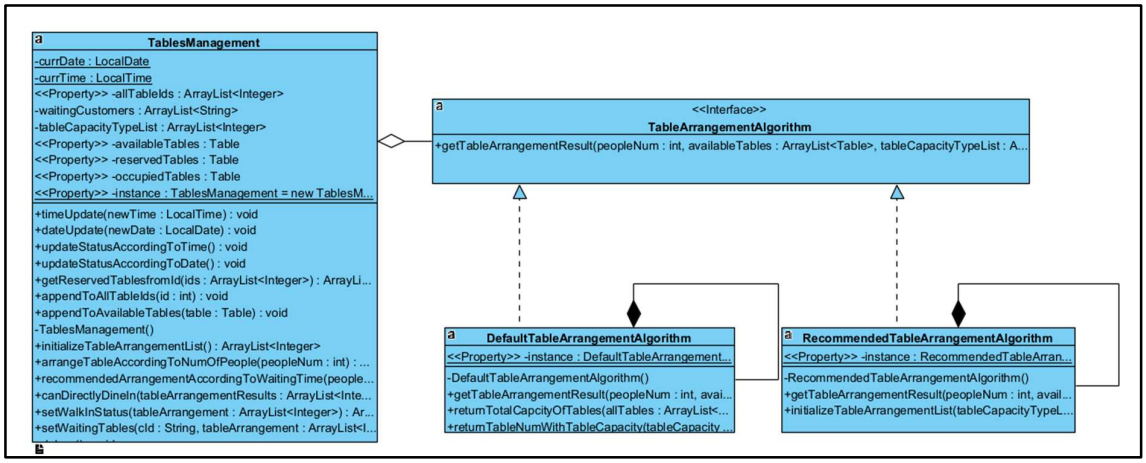
1. State Pattern




Description:

In this structure, the ***Customers*** class has ***CustomerState*** class which is used to decide the customer's state according to his bill amount. If the bill amount reaches a certain level, the customer's state will become SuperVIP and get a corresponding discount. Otherwise, he will be just the VIP state without any discount. This pattern allows the customer to transition between different states.

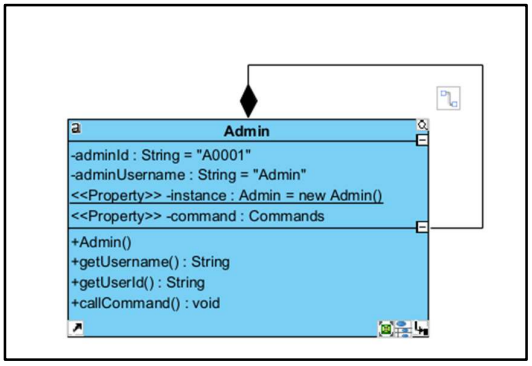
2. Strategy Pattern




Description:

In this structure, strategy pattern is used, which is similar to a state pattern. The strategy pattern is to define a family of algorithms, encapsulate each one, and make them interchangeable.

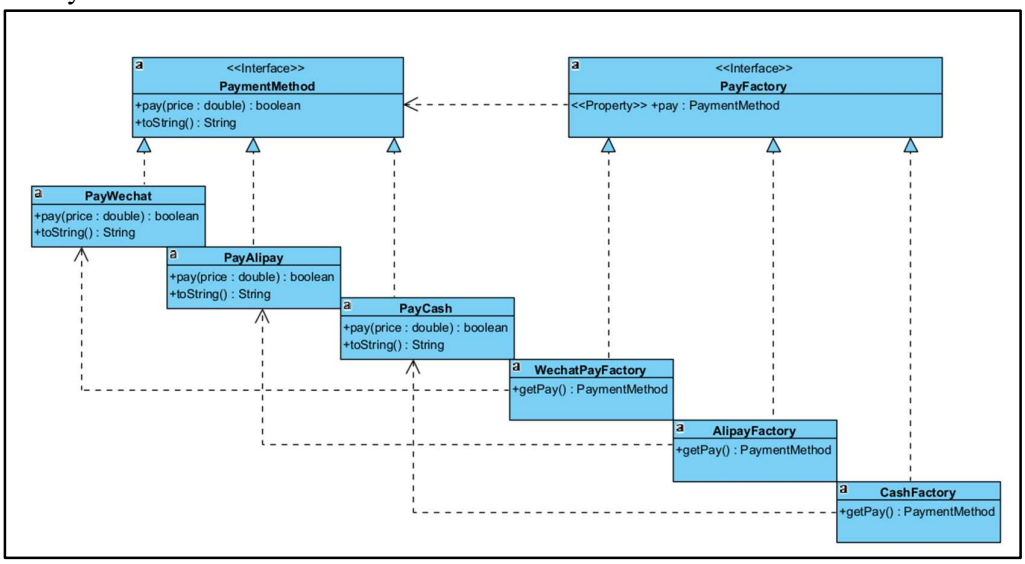
3. Singleton Pattern




Description:

We adopt a singleton pattern in the ***Admin*** class. There will be only one instance created for ***Admin*** class, and only the admin has the highest authority to check relevant information.

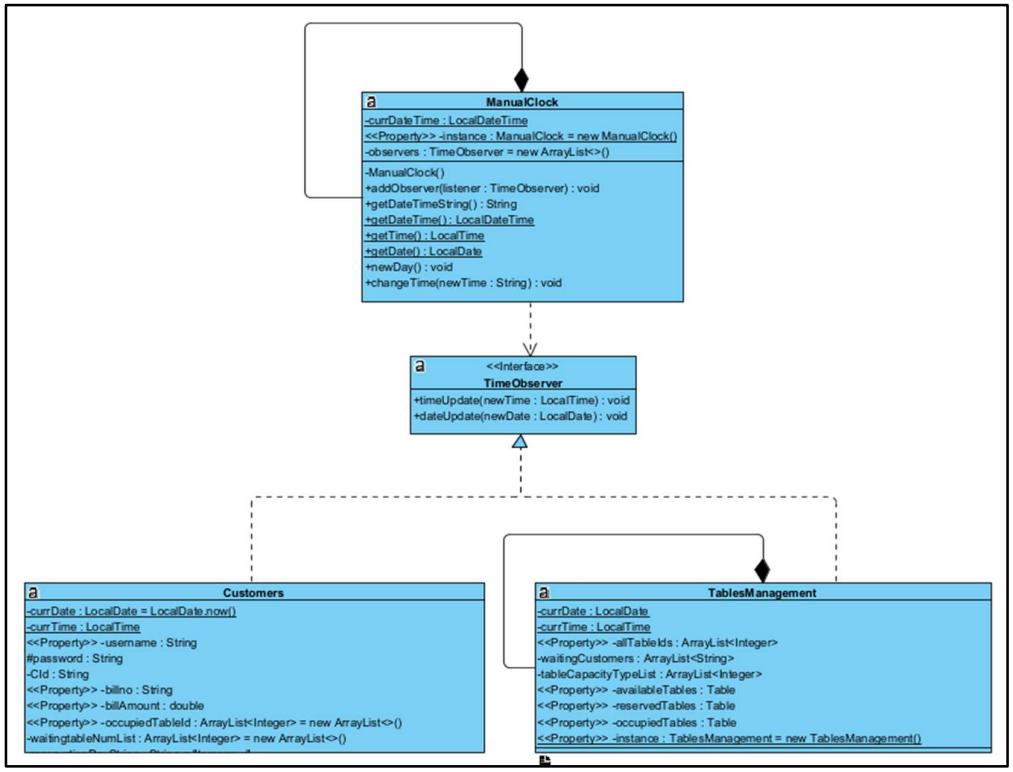
4. Factory-method Pattern




Description:

This structure uses a factory-method pattern. ***PayFactory*** class is the creator, and concrete payment factories such as ***WechatFactory***, ***AlipayFactory*** will implement this interface. Different factories will connect to concrete payment methods. And in the running time, objects can be created dynamically.

5. Observer Pattern




Description:

This structure uses the observer pattern. As ***ManualClock*** class changes the time, it will notify its observers—***TableManagement*** class and ***Customers*** class to update, and each observer will call back to the object. And the information will be updated correspondingly.

## 6. Program Flow & Algorithms

### 6.1 Program Flow

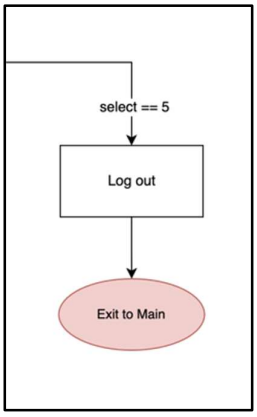
There are three main user modules for the GoGoEat - Food Court Management System, as shown below:

- Customer Module
- Admin Module
- Merchant Module

There are some common operations that apply to all customers, admin, and merchant modules, that is the login and logout function.

**Login Operation:**
Description:
For the login module, users can choose to register a new account, delete their account, or log in. For the login, users will be required to input their username and password, if there is a match of account found in the database, they can log in successfully.

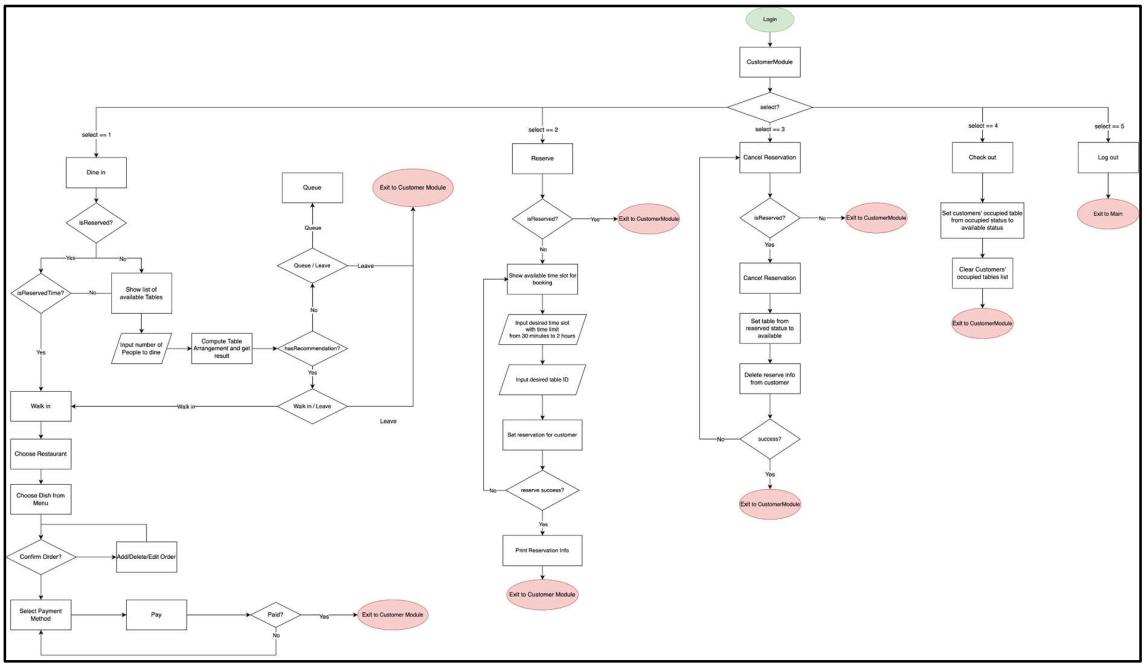
**Logout Operation:**




Description:
All the users can access the logout function. By choosing the logout function, they will exit their corresponding user modules and return to the main. After that, they can log in again.

### 6.1.1 Customers' Flow

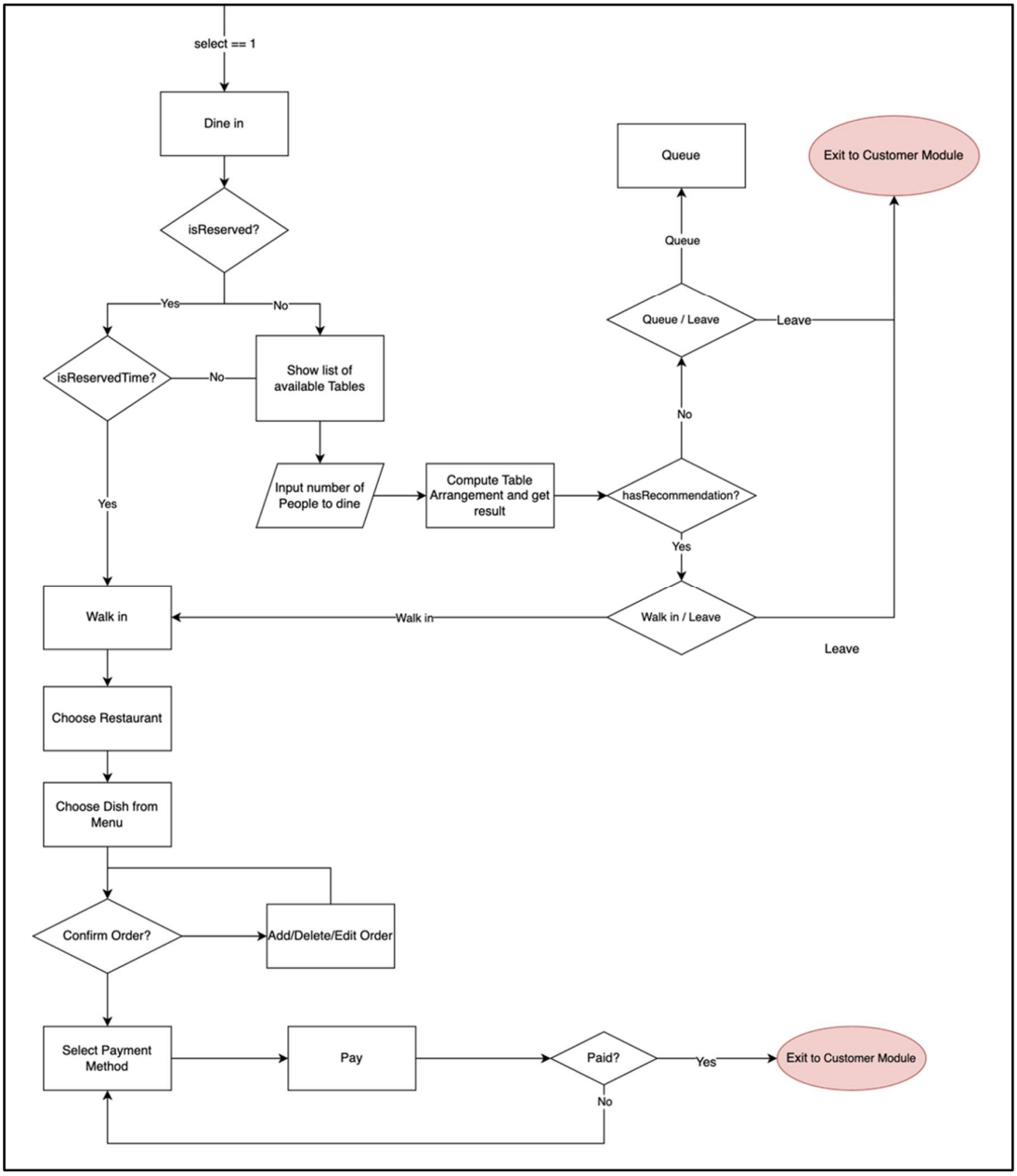
An overview of customers' flow:




Description:

The customer module includes five operations that can be done by the customers in the GoGoEat System. The five operations are dine-in, reserve, cancel reservation, check out, and logout.

**Dine-in operation:**




Description:

In the dine-in operation, only the customers who have not yet sat down can enter the flow. Then the system will check if the customer has a reservation. If there exists a valid reservation for the customer, and it is now time for them to walk in for the reserved time slot, they can directly walk in to dine and start ordering.

For those who are not reserved, they need to input the number of people to dine into the system, the system will then assign tables according to the inputted number of people. There are enough available tables for the number of people to dine so that the system will provide

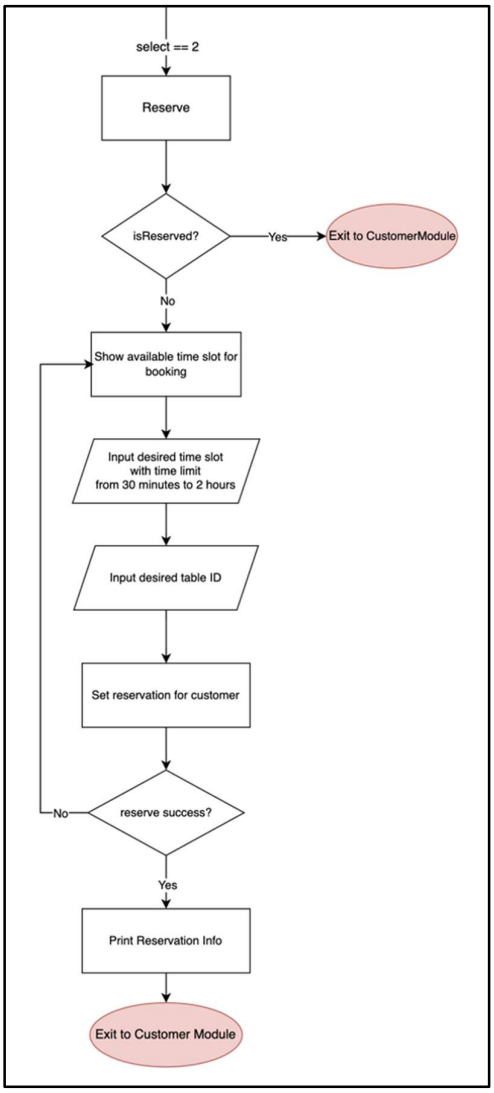
table arrangements for them. The customer can choose to accept the arrangement and walk in or choose to leave.

If there are not enough available tables, there will be no suggested table arrangements, and the customer needs to queue up to wait for the dining customers to check out.

After walking in,  the customer will need to choose from the list of restaurants and the dish from the menu of the restaurant chosen. The ordered dish will first be added to the list of pending orders and is not yet to be confirmed to order. After ordering, the system will prompt the customers if they want to confirm the pending orders made previously. If the customers do not confirm the order, they can always go back to add or remove the dish ordered, and the pending orders will be updated after each modification. After confirmation, the pending orders will be officially ordered and added to the customers' list of official orders.

Finally, after ordering, the customers can proceed to the payment.

**Reservation Operation:**




Description:
For the reservation process, the system will check on the customers on whether they already have a valid reservation. The customers who are already reserved cannot reserve again.

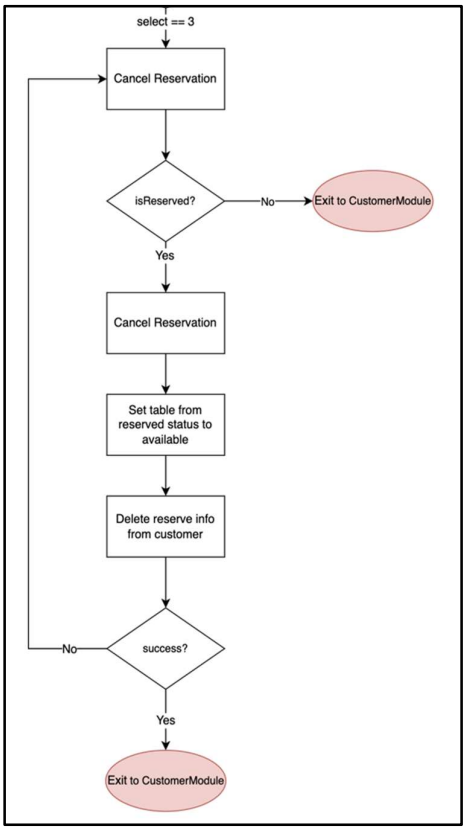
If there are no existing reservations for the customer, they will be prompted with a list of tables, including table id, together with the available time slots for tomorrow. The customer will input their desired table id and time slots for reservation.

However, there are also some constraints for the reservation regarding the duration and format as listed below:

- Duration of the reservation must be longer than 30 minutes and less than 2 hours
- The input format of the reservation time slot is xx:xx-xx:xx (without any spaces)

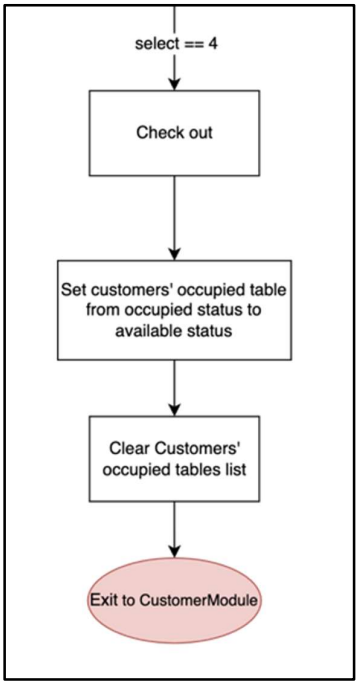
After successful reservation, the program will prompt a success message to notify the customers.

**Cancel Reservation Operation:**




Description:
For canceling the reservation, this function will only be accessed when the customer has an existing valid reservation. To cancel the reservation made, the customer will enter this operation, and the table status will change from reserved to available, and the reservation information stored in the customer will be cleared.

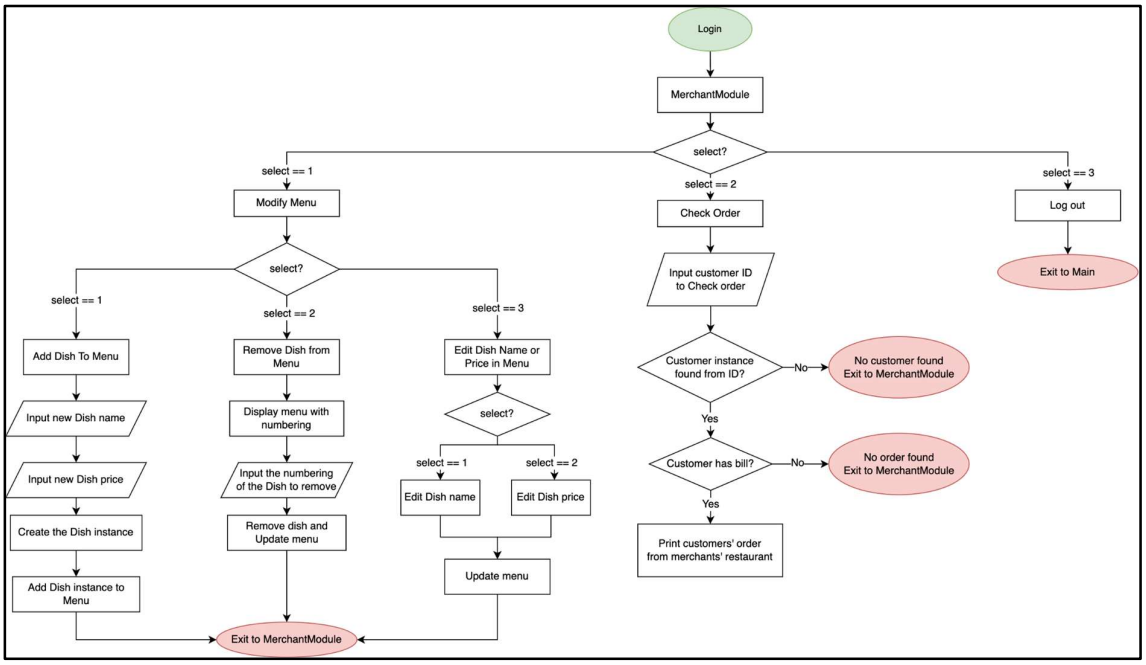
**Check-Out Operation**




Description:
Whenever the customer finishes dining and plans to leave, they need to log in to the system to check out. By choosing to check out, the table status will change from occupied to available, and the table can be assigned to other customers in the queue.

### 6.1.2 Merchants' Flow

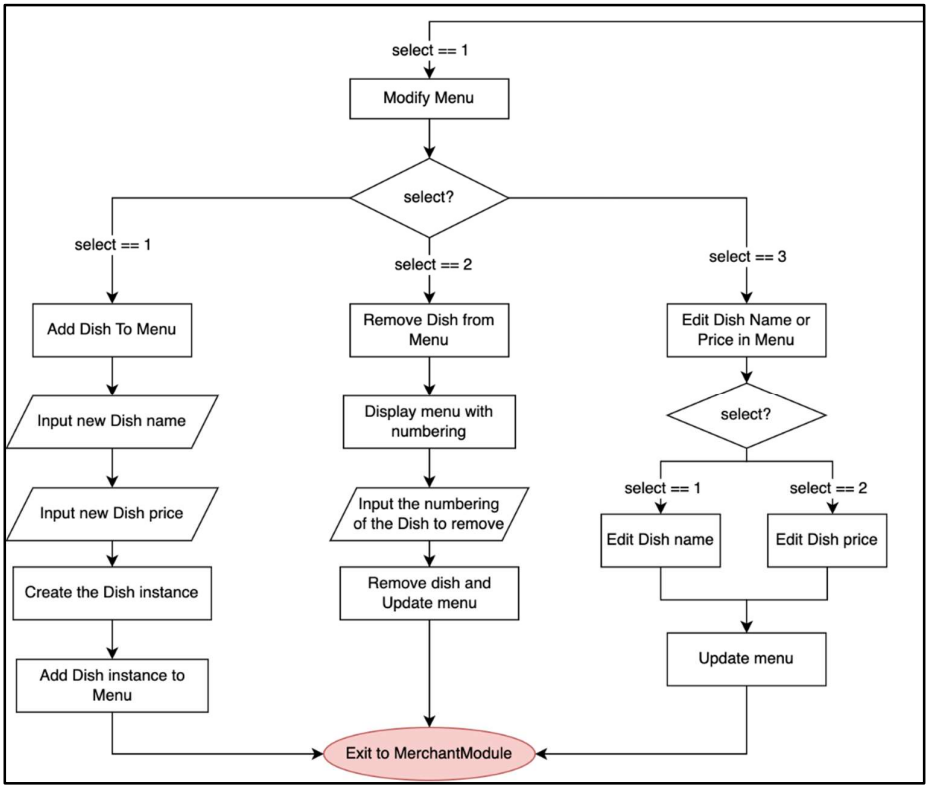
An overview of the merchants' flow:




Description:

There are three main operations for the merchants, which are modifying the menu of the restaurants that they belong to, checking customers' orders, and logout.

**Modify Menu Operation:**



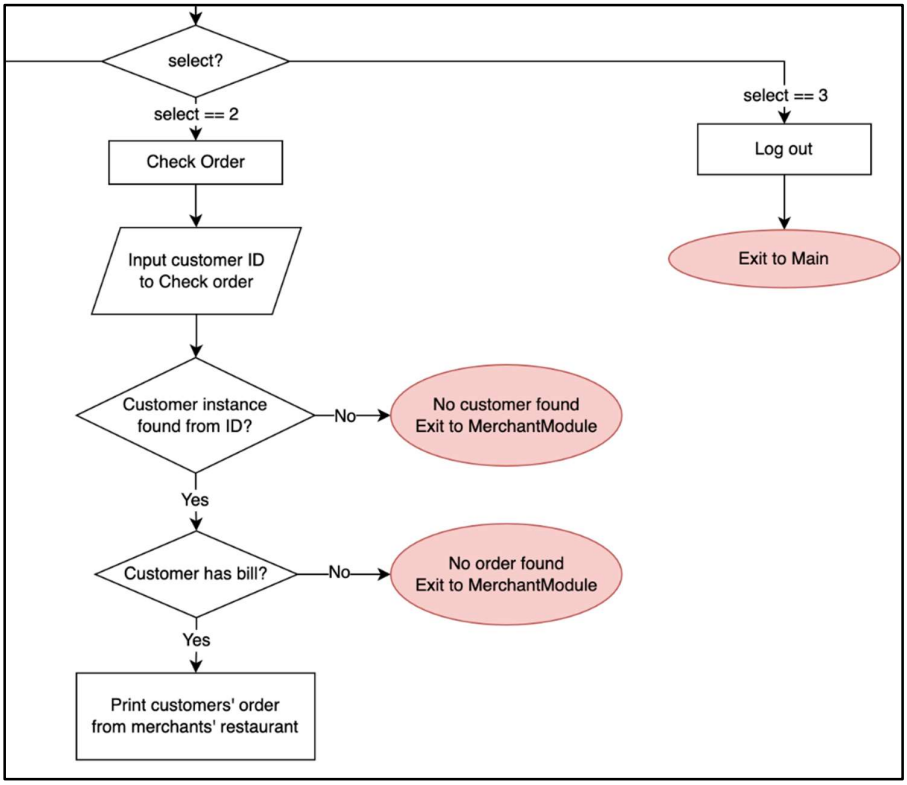

Description:
For modifying the menu, the merchant can choose to manipulate the menu, that is, to add dishes or remove dishes from the menu, or to edit the dish information, that is, the name and price of the dish separately.

For adding dishes, the merchants simply need to input the new name and price of the dish. Then the system will create a dish instance to be added to the menu and update the database.

To remove the dish from the menu, the system will first output the list of current dishes in the menu with the numbering. The merchants need to input the corresponding numbering of the dishes to delete. Then it will be deleted from the menu.

For editing name and price, the dish instance will be matched by the dish's name. After an instance is matched, the merchant can choose to edit the name or price and input the new name or price, then the corresponding information about the dish will be updated in the database.

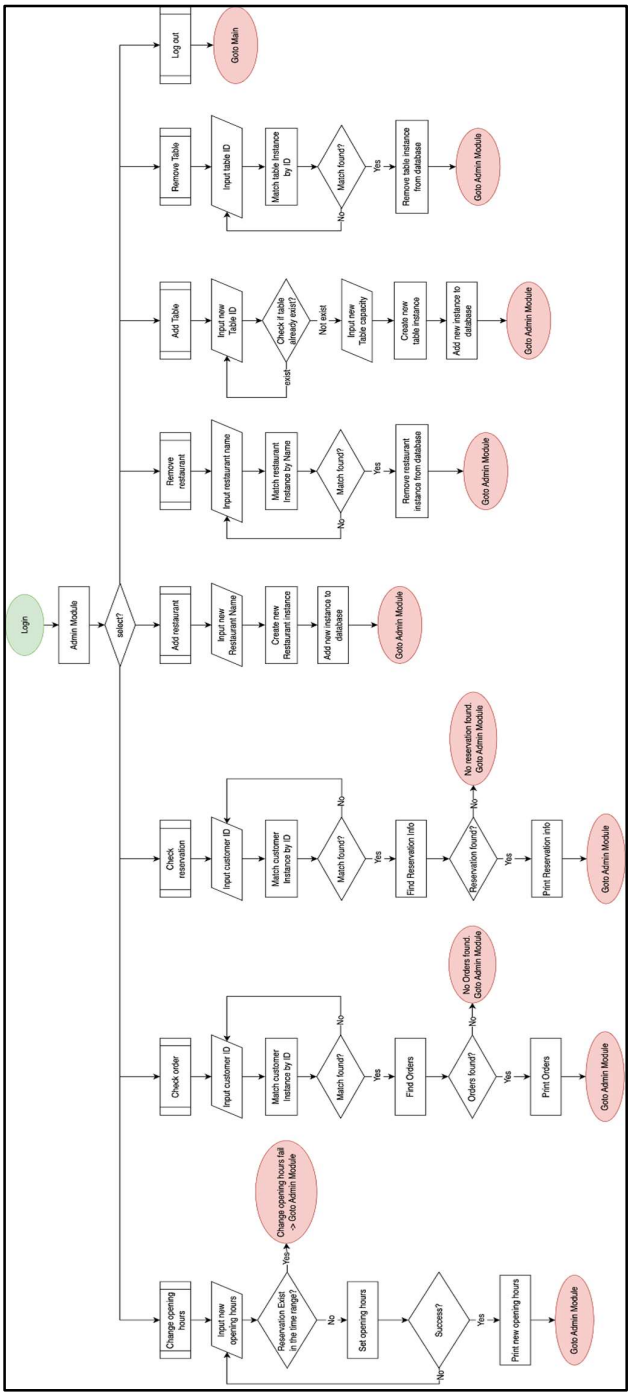
**Check Customers' Orders Operation**




Description:
For checking customers' orders, merchants will need to input the customer id of the customer that needs to check the order. The system will first match the customer instance, and if there is a successful match, all the orders made by that customer that belongs to the restaurant owned by the merchant will be passed to the merchant and get printed out.

### 6.1.3 Admin's Flow

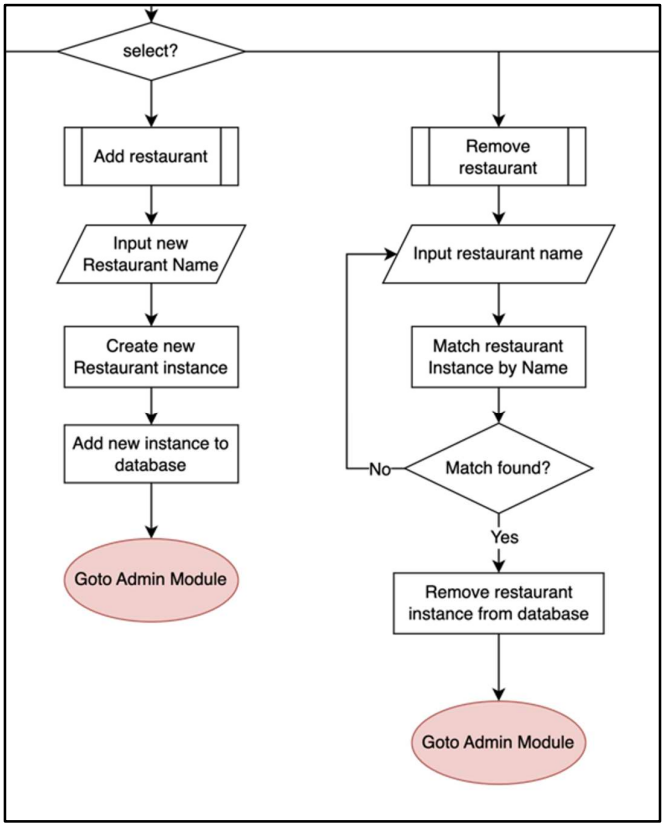
An overview of the admin's flow:




Description:

For the admin module, there are eight operations, that is changing opening hours, checking customers' orders, checking customers' reservations, manipulating restaurants, manipulating tables, and logout.

**Restaurants Modification:**



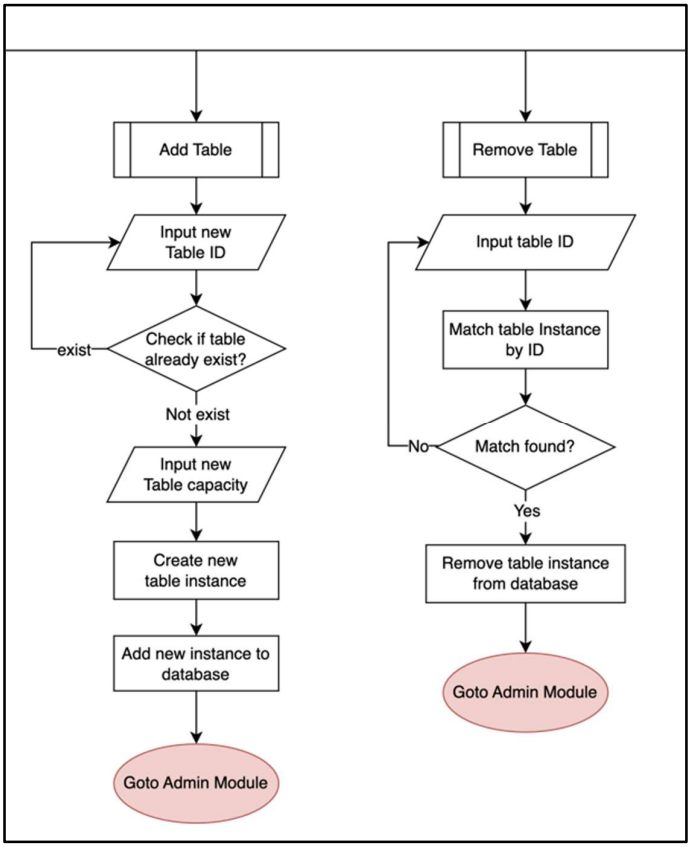

**Description:**

To register a new restaurant into the food court management system, the name of the restaurant is the key to matching the restaurant instance.

For adding restaurants, the name of the new restaurant is inputted, then a new restaurant instance will be created and added to the database.

To remove a restaurant from the database, the restaurant's name is inputted to match the instance. If there is a match, the restaurant can be deleted.

**Tables Modification:**

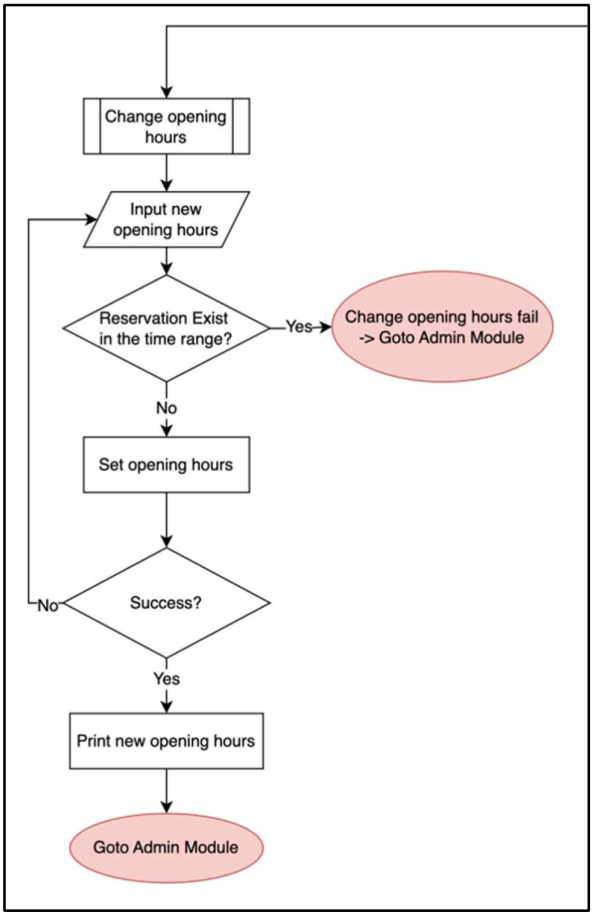



Description:
To register a new table into the food court management system, the table's id is the key to match the table instance.

For adding tables, the id of the new table is inputted. The system will check if the id already exists in the system and reject the request if there is any collision. Then a new table instance will be created and added to the database.

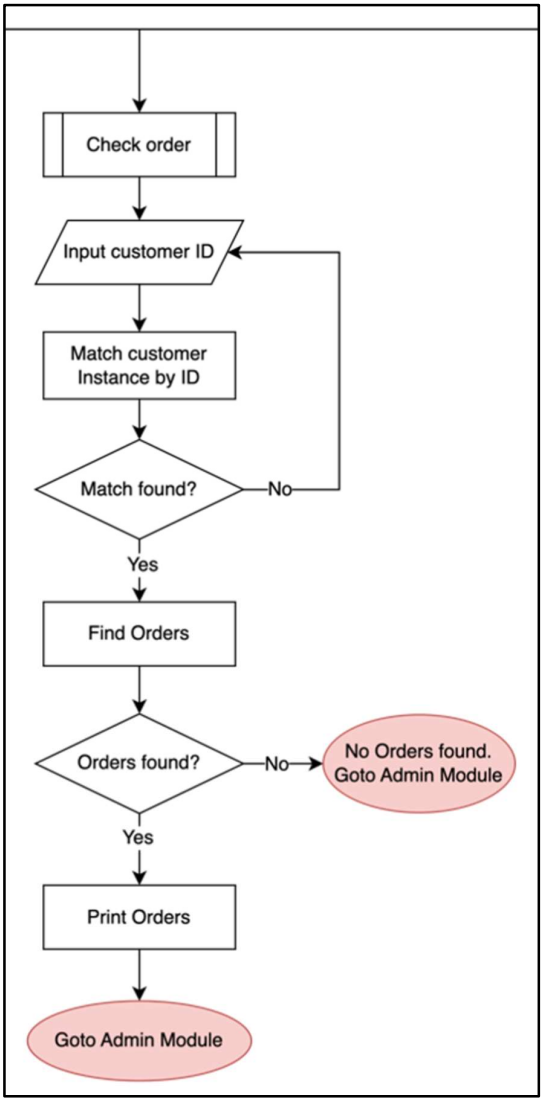
To remove a table from the database, the table's id is inputted to match the instance. If there is a match, the table can be deleted.

**Change Opening Hours of the food court:**




Description:
To change the opening hour of the food court, the admin will need to input the new time range in the format of xx:xx-xx:xx. The system will check if there is a time collision by checking if there are any reservation time slots crossing over the new opening hours. The change cannot be made if there is a collision. If the change is successful, the system will print out the new opening hour to notify the admin.

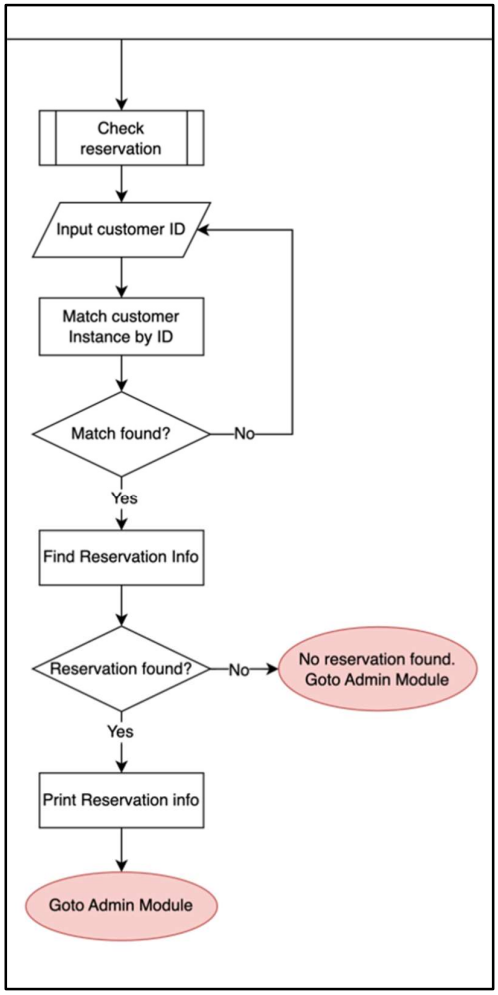
**Check Customers' Orders Operation:**




Description:
For checking customers' orders, the admin will need to input the customer id of the customer that needs to check the order. This process is similar to the order checking in the merchant module.

The system will first match the customer instance, and if there is a successful match, all the orders made by that customer will be passed to the admin and get printed out, regardless of the restaurant chosen.

**Check Customers' Reservation Operation:**




Description:
For the checking of the reservation, similarly, the customer id is inputted by the admin to the system, and a matching instance of the customer can be found. The system will output the reservation information made by that customer if there is any. When finished, the process will return to the admin module.

### 6.2 Algorithms

#### 6.2.1 TablesManagement.java

**Objective of this section:** This includes the management functions that act on tables. It includes functions to add/remove tables, table status setting functions, reservation functions for different tables, table arrangement implementation functions, walk-in setting functions, and waiting-table setting functions.

**Main fields:**

- ***availableTables***: store the list of available tables
    - type: ArrayList<Table>
- ***reservedTables***: store the list of reserved tables
    - type: ArrayList<Table>
- ***occupiedTables***: store the list of occupied tables
    - type: ArrayList<Table>
- ***waitingCustomers***: store the list of customers waiting for tables
    - type: ArrayList<Customers>
- ***tableCapacityTypeList***: store the list of table capacity types
    - type: ArrayList<Integer>

Implementation of ***waitingCustomers***:
When the customer chooses to wait for the default arrangement, then the system will add the customer to the ***waitingCustomers*** list. Afterward, if some table(s) is released, which means some other customers check out, the system will check this list and find the customer(s) waiting for the specified tables, then add the table(s) to their occupied list. Then after the customer's waiting tables list is empty, the system will remove the customer from the ***waitingCustomers*** list.

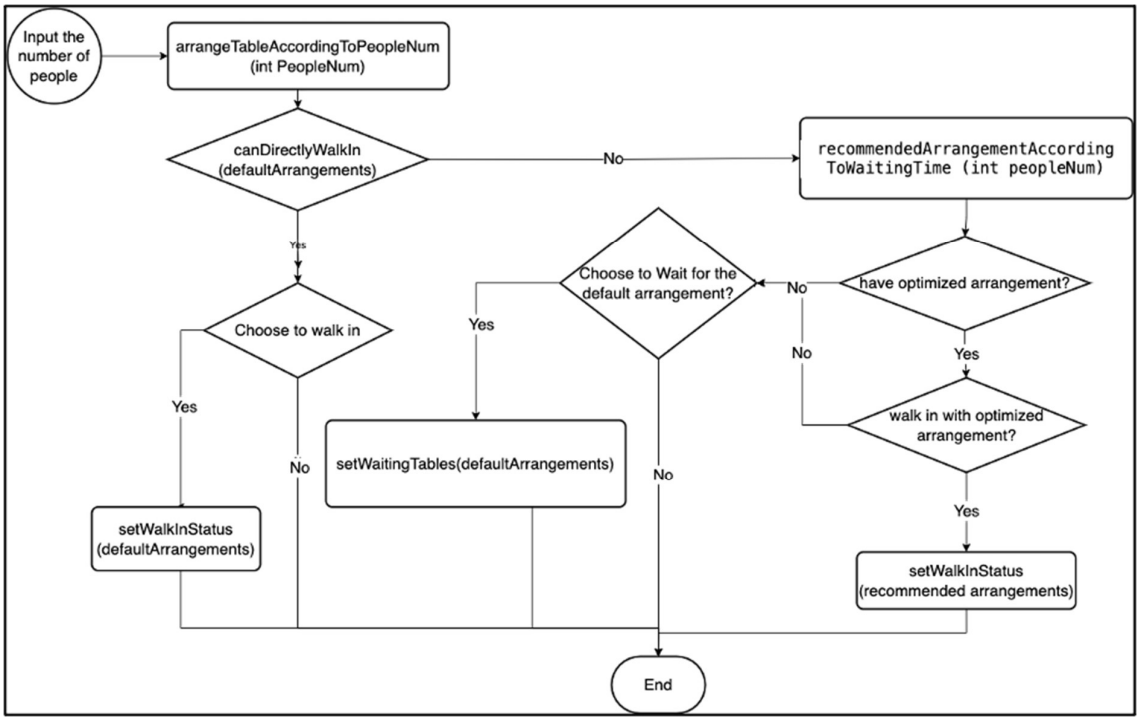
Implementation of ***tableCapacityTypeList***:
This field stores all the table types in this program in descending capacity order. For example, if there are three table types(2,4,8), then the list will store the three types: 8, 4, 2. The ***tableCapacity*** will be modified in two situations. The first one is when the admin adds a new table with a capacity that is not stored in the ***tableCapacityTypeList***, then the system will automatically add the capacity to the list and sort the list to keep it in the right order. The second situation is when the admin removes a table and if, after that, the number of tables with that particular type is 0, then remove the table capacity type from this list.

**Flow graph of the TablesManagement**

**Explanation:**

The following flow graph is for the table arrangement. After customers enter the number of people, the system will call the ***DefaultArrangementAlgorithm*** function to make arrangements according to the number of people.

Then, the system will call the ***canDirectlyWalkIn*** function to judge whether the default arrangement is currently available. If the default arrangement is available and the customer chooses to walk in now, then it will trigger the ***setWalkInStatus*** function to set the status of the related tables and add them to the customer's occupied table list. However, if the ***canDirectlyWalkIn*** function returns false, it will first trigger the ***RecommendedTableArrangementAlgorithm*** function. If there is an optimized arrangement, then the system will let the customer choose whether to walk in with recommended arrangements or continue to wait for the default arrangements.



**Main Functions 1:** ***DefaultArrangementAlgorithm***

Description: To make default arrangements according to the number of people and to minimize the number of tables according to the current table capacity type and the number of corresponding table types.

Output: ArrayList<Integer> ***tableArrangementResult***, this field is stored in the descending order of table capacity type.

Implementation: Generate the table arrangements by choosing the most suitable table according to the number of people.

Logics(Pseudo Code)

```
//Pseudo Code for Default Arrangment: only for demonstration
List<Integer> capacitylist; //store the table capacity types in descending order
List<Integer>resultList; //store the arrangement results in the order of capacitylist
CapacityIndex=0; //initial capacity index
DefaultResult(int peopleNum)={
    // initialize the tableNumForCurrentCapacity
    Int tabeNumForCurrentCapacity=0;
    // if it is the following situation then
    If(peopleNum==0 and capacityIndex=max){
    End this function;
    }
    //get current capacity
    Capacity=capacitylist[capacityIndex];
    //situation1
    if(peopleNum<=capacity and next capacity<peopleNum){
    tableNumForCurrentCapacity=1;
    }
    //situation2
    Else if (peopleNum>capacity{
        Int tableNumForCurrentCapcity+=peopleNum/capacity;
        PeopleNum=peopleNum%capacity;
        //situation3
        If(peopleNum<capacity and peopleNum> next capacity){
            PeopleNum=0;
            tableNumForCurrentCapacity+=1;
        }
    }
    //store the result and go to next sub-problem
    ResultList[capacityIndex]=tableNumForCurrentCapacity;
    CapacityIndex++;
}
```

Detailed Implementation:

```java
@Override
public ArrayList<Integer> getTableArrangementResult(int peopleNum, ArrayList<Table> availableTables,
        ArrayList<Integer> tableCapacityTypeList, ArrayList<Table> allTables)
        throws ExPeopleNumExceedTotalCapacity {

    if (peopleNum ⩽ returnTotalCapcityOfTables(allTables)) {
        int tmpPeopleNum = peopleNum;
        StringBuilder arrangementResultMessage = new StringBuilder(str: "\nYour arranged tables are: \n");

        // Store the number of tables of the corresponding table type of that index
        ArrayList<Integer> tableArrangementResults = new ArrayList<Integer>();
        tableArrangementResults.addAll(initializeTableArrangementList(tableCapacityTypeList));
        for (int i = 0; i < tableCapacityTypeList.size(); i++) {
            int tmpResults = 0;
            int tableCapacity = tableCapacityTypeList.get(i);

            // This happens when the last table type comes, it should store all the remaining people
            if (i == tableCapacityTypeList.size() - 1) {
                tmpResults = (tmpPeopleNum % tableCapacity == 0) ? (tmpPeopleNum / tableCapacity)
                        : ((tmpPeopleNum / tableCapacity) + 1);
            } else {
                // minimize the table num , e.g. if exists table type of 2,4,8; 7 people → [1] 8-seats tables
                int addingTableNum = (tmpPeopleNum / tableCapacity ⩽ returnTableNumWithTableCapacity(tableCapacity, allTables))
                                ? (tmpPeopleNum / tableCapacity)
                                : returnTableNumWithTableCapacity(tableCapacity, allTables);
                tmpResults += addingTableNum;
                tmpPeopleNum = tmpPeopleNum - tableCapacity * addingTableNum;
                if (tableCapacity ⩾ tmpPeopleNum
                        && tmpPeopleNum > tableCapacityTypeList.get(i + 1)
                        && tmpResults < returnTableNumWithTableCapacity(tableCapacity, allTables)) {
                    tmpResults += 1;
                    tmpPeopleNum = 0;
                }
            }
            if (tmpResults > 0) {
                arrangementResultMessage
                        .append(String.format(format: "[%d] [%d-Seats] Tables \n", tmpResults, tableCapacity));
            }
            tableArrangementResults.set(i, tmpResults);
            if (tmpPeopleNum == 0) {
                break;
            }
        }
        System.out.println(arrangementResultMessage);
        return tableArrangementResults;
    }
    throw new ExPeopleNumExceedTotalCapacity(returnTotalCapcityOfTables(allTables));
}
```

Real Usage in the program:

```
Below is the available tables:
Num of Available 2-Seats Table: 5
Num of Available 4-Seats Table: 3
Num of Available 8-Seats Table: 2

Please input the number of people: 24

Your arranged tables are:
[2] [8-Seats] Tables
[2] [4-Seats] Tables
```

```
Please select your operation: 1

Below is the available tables:
Num of Available 2-Seats Table: 5
Num of Available 4-Seats Table: 3
Num of Available 8-Seats Table: 2

Please input the number of people: 98
Exceeds the capacity of 38, please input an valid num
```

**Main Functions 2: *RecommendedTableArrangementAlgorithm***

Description: To make arrangements according to the currently available tables based on the principle of minimizing the usage of tables. And to provide the optimized table arrangement results when the default arrangement is not available and there exists another arrangement that allows the customer to walk in directly.

Output: ArrayList<Integer> ***tableArrangementResult***, this field is stored in the descending order of table capacity type.

Implementation: Generate the table arrangements by putting the table into a table list in the descending order of capacity

Detailed Implementation:

```
16      @Override
17      public ArrayList<Integer> getTableArrangementResult(int peopleNum, ArrayList<Table> availableTables,
18              ArrayList<Integer> tableCapacityTypeList, ArrayList<Table> allTables) {
19          Collections.sort(availableTables);
20          int tmpPeopleNum = peopleNum;
21          ArrayList<Integer> tableArrangementResults = new ArrayList<Integer>();
22          tableArrangementResults.addAll(initializeTableArrangementList(tableCapacityTypeList));
23          for (Table t : availableTables) {
24              if (t.getTableCapacity() < peopleNum) {
25                  int tCapacity = t.getTableCapacity();
26                  int capacityIndex = tableCapacityTypeList.indexOf(tCapacity);
27                  int num = tableArrangementResults.get(capacityIndex);
28                  tableArrangementResults.set(capacityIndex, num + 1);
29                  tmpPeopleNum = tmpPeopleNum - tCapacity;
30                  if (tmpPeopleNum <= 0) {
31                      break;
32                  }
33              }
34          }
35          if (tmpPeopleNum > 0) {
36              System.out.println(x: "No Optimized Recommended Arrangements!");
37              return null;
38          } else {
39              StringBuilder recommendedArrangementMsg = new StringBuilder(str: "The Optimized Recommended Arrangements are: ");
40              for (int i = 0; i < tableArrangementResults.size(); i++) {
41                  if (tableArrangementResults.get(i) > 0) {
42                      int tCapacity = tableCapacityTypeList.get(i);
43                      recommendedArrangementMsg
44                              .append(String.format(format: "\n[%d] [%d-seats] ", tableArrangementResults.get(i), tCapacity));
45                  }
46              }
47              System.out.println(recommendedArrangementMsg);
48              return tableArrangementResults;
49          }
50
51      }
```

Real Usage:

```
Below is the available tables:
Num of Available 2-Seats Table: 5
Num of Available 4-Seats Table: 1
Num of Available 8-Seats Table: 0

Please input the number of people: 30

Your arranged tables are:
[2] [8-Seats] Tables
[3] [4-Seats] Tables
[1] [2-Seats] Tables

For default arrangements, you still need to wait for
[2] [8-Seats] Table(s)
[2] [4-Seats] Table(s)
No Optimized Recommended Arrangements!
```

```
Below is the available tables:
Num of Available 2-Seats Table: 5
Num of Available 4-Seats Table: 1
Num of Available 8-Seats Table: 0

Please input the number of people: 8

Your arranged tables are:
[1] [8-Seats] Tables

For default arrangements, you still need to wait for:
[1] [8-Seats] Table(s)
The Optimized Recommended Arrangements are:
[1] [4-seats]
[2] [2-seats]
```

**Main Functions 3: *CanDirectlyWalkIn*** Function

Description: Judge whether the default result can directly walk in.

Implementation:
According to the input array list:

1. iterate for each type of table and compare the number of available tables and the number of that type of table which is needed;
2. if ***canDirectlyWalkIn***: return true;
3. if not ***canDirectlyWalkIn***:  print the waiting table message and return false;

Detailed Implementation:

```
196     * Determine: whether the table arrangement results is available now
197     * 1.Yes: return true;
198     * 2.No: return false
199     *
200     * Testing:
201     * - Need to know the current available tables
202     * - Then you may create a arraylist as table arrangemnet results,
203     * and put it into the argument
204     */
205
206    public boolean canDirectlyDineIn(ArrayList<Integer> tableArrangementResults) {
207        boolean canDirectlyWalkIn = true;
208        StringBuilder waitingTablesListMessage = new StringBuilder(
209                str: "For default arrangements, you still need to wait for: ");
210        for (int i = 0; i < tableArrangementResults.size(); i++) {
211            int tableCapacityType = tableCapacityTypeList.get(i);
212            int availableNumOfThisTableType = returnAvailableTableNumWithCapacity(tableCapacityType);
213            int num = tableArrangementResults.get(i);
214            if (num > availableNumOfThisTableType) {
215                int numOfWaitingTables = num - availableNumOfThisTableType;
216                canDirectlyWalkIn = false;
217                waitingTablesListMessage
218                        .append(String.format(format: "\n[%d] [%d-Seats] Table(s) ", numOfWaitingTables, tableCapacityType));
219            }
220        }
221        if (canDirectlyWalkIn) {
222            return true;
223        }
224        System.out.println(waitingTablesListMessage);
225        return false;
226
227    }
```

**Main Functions 4: *setWalkInStatus*** Function

Purpose: When the customer chooses to walk in, then call this function to set walk-in status.

Implementation:
According to the ***tableArrangementsResult***:

1) Iterate the table capacity, and see whether this capacity type is needed.
2) Find the tables with corresponding table capacity.
3) Add these table IDs to ***checkedInTableIds***.
4) Set the tables from available to occupied.
5) Return the ***checkedInTableIds***.

Detailed Implementation:

```
236     public ArrayList<Integer> setWalkInStatus(ArrayList<Integer> tableArrangement) {
237         ArrayList<Integer> checkedInTableIds = new ArrayList<Integer>();
238         for (Integer tableCapacity : tableCapacityTypeList) {
239             int needOfThisTableCapcity = tableArrangement.get(tableCapacityTypeList.indexOf(tableCapacity));
240             if (needOfThisTableCapcity > 0) {
241                 int tmpCount = 0;
242                 ArrayList<Table> copyOfAvailableTables = new ArrayList<Table>();
243                 copyOfAvailableTables.addAll(availableTables);
244                 for (Table t : copyOfAvailableTables) {
245                     if (t.getTableCapacity() == tableCapacity) {
246                         tmpCount++;
247                         checkedInTableIds.add(t.getTableId());
248                         setTableFromAvailableToOccupiedStatus(t.getTableId());
249                         if (tmpCount == needOfThisTableCapcity) {
250                             break;
251                         }
252                     }
253                 }
254             }
255         }
256         System.out.println(x: "Successfully Dine In!");
257         return checkedInTableIds;
258     }
```

**Main Functions 5:** ***setWaitingTables*** Functions

Purpose: When the customer chooses to wait for the default arrangements, set occupied status for the currently available tables and add the remaining tables to the customer's ***waitingTableNum*** List.

Implementation:
According to the ***tableArrangementsResult***:

1) Iterate the table capacity and see whether this capacity type is needed.
2) Find the tables with corresponding table capacity and add these available table ID of that type to ***checkedInTableIds***, and set the tables from available to occupied.
3) Update the ***waitingTableNumList*** for the remaining tables that are needed but currently not available.
4) Pass the ***checkedInTableIds*** to the customer's ***occupiedTableList***, and add the ***waitingTableNumList*** to the customer.

```
260     /*
261      * According to table arrangements, put the table that are available into
262      * customer's occupied, and store the remaining into wairing list
263      */
264     public ArrayList<Integer> setWaitingTables(String cId, ArrayList<Integer> tableArrangement) {
265         ArrayList<Integer> waitingTablesNumList = new ArrayList<Integer>();
266         ArrayList<Integer> checkedInTableIds = new ArrayList<Integer>();
267         waitingCustomers.add(cId);
268         waitingTablesNumList.addAll(initializeTableArrangementList());
269         StringBuilder waitingTablesListMessage = new StringBuilder(
270                 str: "For selected arrangements, you still need to wait for: \n");
271         for (Integer tableCapacityType : tableCapacityTypeList) {
272             int index = tableCapacityTypeList.indexOf(tableCapacityType);
273             int needOfThisTableCapcity = tableArrangement.get(index);
274             if (needOfThisTableCapcity > 0) {
275                 int tmpCount = 0;
276                 ArrayList<Table> copyOfAvailableTables = new ArrayList<Table>();
277                 copyOfAvailableTables.addAll(availableTables);
278                 for (Table t : copyOfAvailableTables) {
279                     if (t.getTableCapacity() == tableCapacityType) {
280                         tmpCount++;
281                         checkedInTableIds.add(t.getTableId());
282                         setTableFromAvailableToOccupiedStatus(t.getTableId());
283                         if (tmpCount == needOfThisTableCapcity) {
284                             break;
285                         }
286                     }
287                 }
288                 // if available num is not enough, then pass the remaining num to customer
289                 if (tmpCount < needOfThisTableCapcity) {
290                     int waitingCount = needOfThisTableCapcity - tmpCount;
291                     waitingTablesNumList.set(index, waitingCount);
292                     waitingTablesListMessage.append(String.format(format: "[%d] [%d-Seats] Table\n", waitingCount,
293                             tableCapacityType));
294                 } else {
295                     waitingTablesNumList.set(index, element: 0);
296                 }
297             }
298         }
299         CommandCustomerDineIn.addCheckInInfo(checkedInTableIds);
300         CommandCustomerDineIn.addWaitingInfo(waitingTablesNumList);
301         System.out.println(waitingTablesListMessage);
302         return waitingTablesNumList;
303     }
```

### 6.2.2 AccountsManagement.java

```java
public static Set<Entry<String, String>> sortByValue(HashMap<String, String> hashmap) {

    /*
     * Account List Sorting by UserId
     * 1. Sort by prefix: A, C, M
     * 2. Sort by ID: 4-index number
     */

    Set<Entry<String, String>> entries = hashmap.entrySet();

    // Sort Method comparator
    You, 3 weeks ago | 1 author (You)
    Comparator<Entry<String, String>> valueComparator = new Comparator<Entry<String, String>>() {
        @Override
        public int compare(Entry<String, String> e1, Entry<String, String> e2) {
            String v1 = e1.getValue();
            String v2 = e2.getValue();
            return v1.compareTo(v2);
        }
    };

    List<Entry<String, String>> listOfEntries = new ArrayList<Entry<String, String>>(entries);

    // sorting HashMap by values using comparator
    Collections.sort(listOfEntries, valueComparator);
    LinkedHashMap<String, String> sortedByValue = new LinkedHashMap<String, String>(listOfEntries.size());

    // List to Map
    for (Entry<String, String> entry : listOfEntries) {
        sortedByValue.put(entry.getKey(), entry.getValue());
    }

    Set<Entry<String, String>> entrySetSortedByValue = sortedByValue.entrySet();

    return entrySetSortedByValue;
}
```

Description:
At the program start, the account information will be first printed out before logging in. For the output of the accounts, it is sorted with this ***sortByValue*** function. All the accounts will be sorted by user id. The user id will first be sorted by its prefix: A, C, M. Then, it will sort by its last four digit numbers.

Effect:

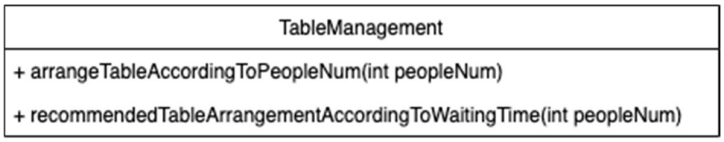
```
List of All Active Accounts:
[01] A0001 | admin
[02] C0001 | yinch33
[03] C0002 | ta123
[04] C0003 | wedu2
[05] C0004 | lpy
[06] M0001 | KFCWorker
[07] M0002 | McDonaldWorker
[08] M0003 | TamJaiWorker
[09] M0004 | TamJaiWorker2
[10] M0005 | TamJaiWorker1
[11] M0006 | PepperLunchWorker
```

## 7. Code Refactoring

### 7.1 *TableArrangementAlgorithm.java*

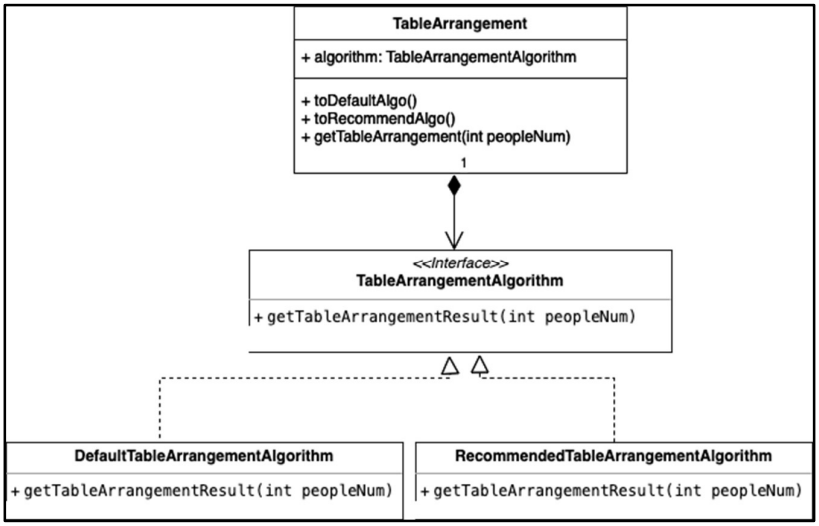
**Method:** Extract Interface

Before Code Refactoring:




Description:
We have two separate methods that contain two different algorithm implementations in TableManagement class. In this situation, if we want to add a new algorithm, it will be very inconvenient and will make TableManagement class even longer.

After Code Refactoring:

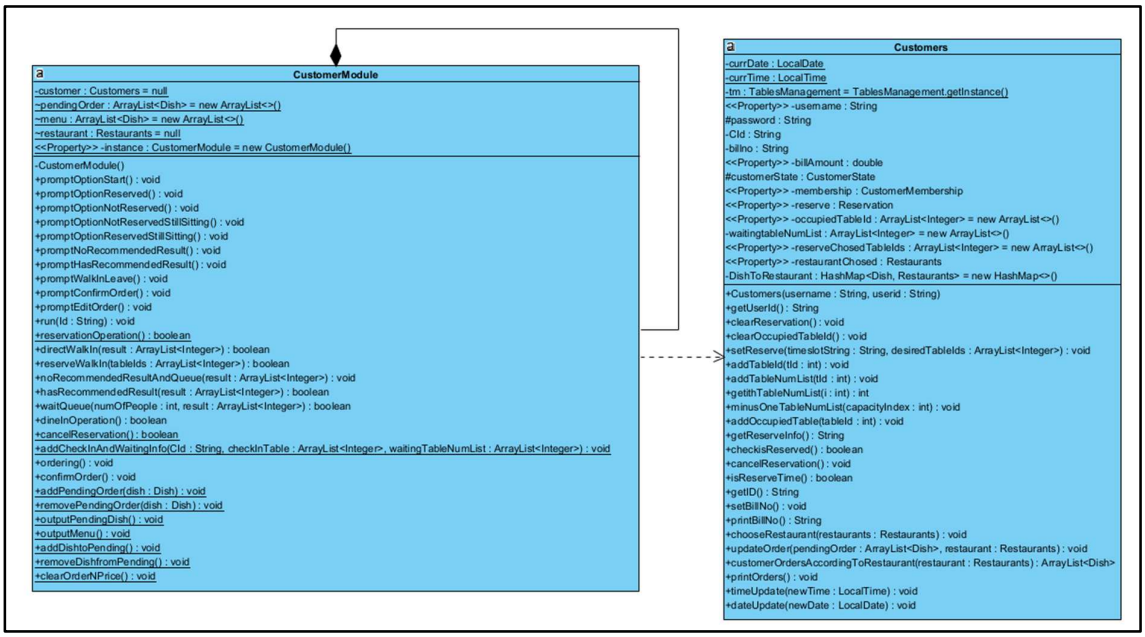



Description:
We extract the two methods into separate classes and share one common interface ***TableArrangementAlgorithm***. After this, we can feel free to add more algorithms according to our needs and won't influence the main code structure of the ***TableManagement*** class.

### 7.2 CustomerModule.java and CommandCustomerModulePrompt.java

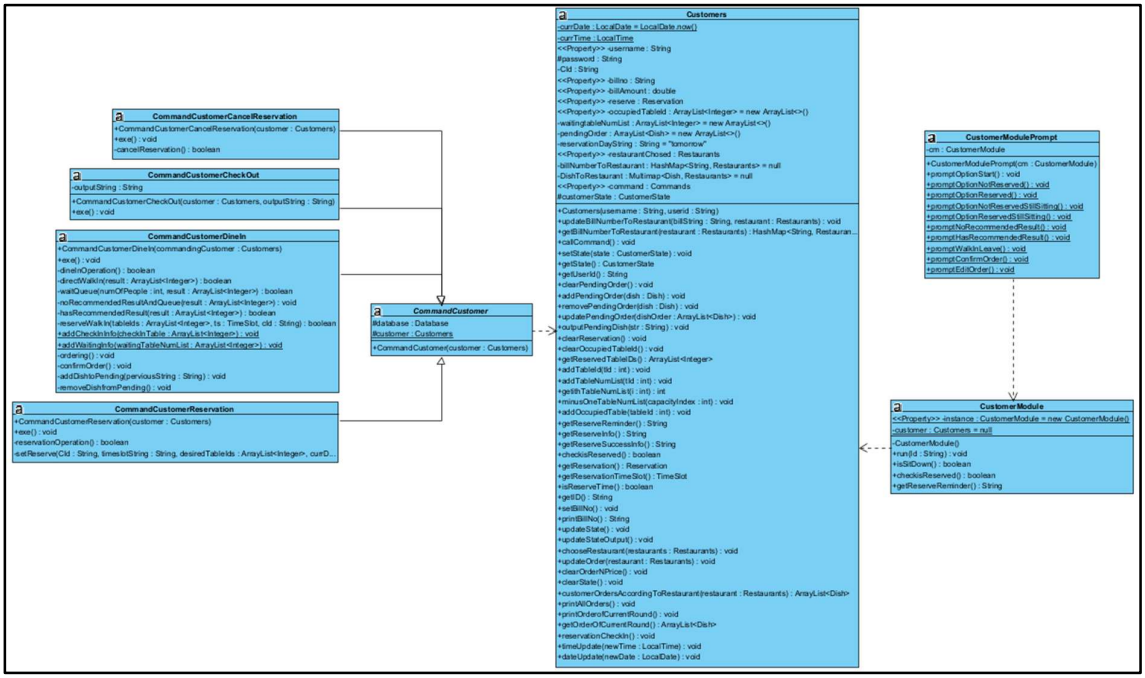
**Method:** Extract Class

Before refactoring:




Description:
Before code refactoring, only just two classes –***Customers*** class and ***CustomerModule*** class are set. The purpose of setting ***CustomerModule*** class is to put all of the processes related to customers in it, such as ***CustomerDineIn***, and ***CustomerCheckOut***, which means that ***CustomerModule*** class will call the functions in the ***Customers*** class. This causes the code in the ***CustomerModule*** class to be too long, making it inconvenient to maintain related functions.

After refactoring:




Description:

We extracted functions about ***Customers*** from the ***CustomerModule*** class into ***CommandCustomerCancelReservation*** class, ***CommandCustomerCheckOut*** class, ***CommandCustomerReservation*** class, and ***CommandCustomerDineIn*** class. And we extracted the CustomerModulePrompt function from ***CustomerModule*** class, which is used to initialize the options available for customer selection. In this way, the code size in each class will be reduced, which is beneficial for maintaining the code.

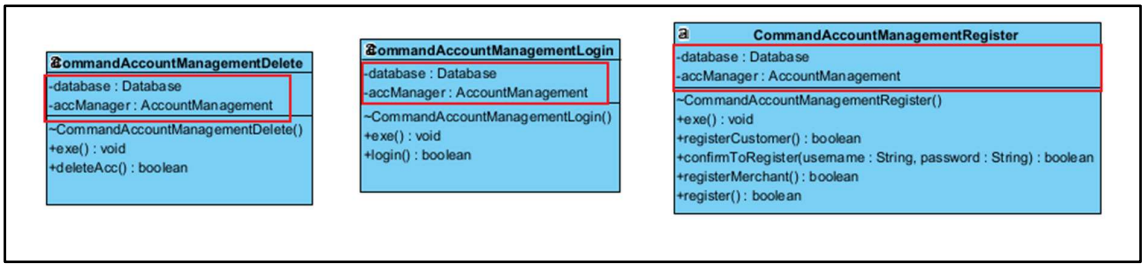
### 7.3 Account Management related

**Classes:**

- ***CommandAccountManagementDelete***
- ***CommandAccountManagementLogin***
- ***CommandAccountManagementRegister***

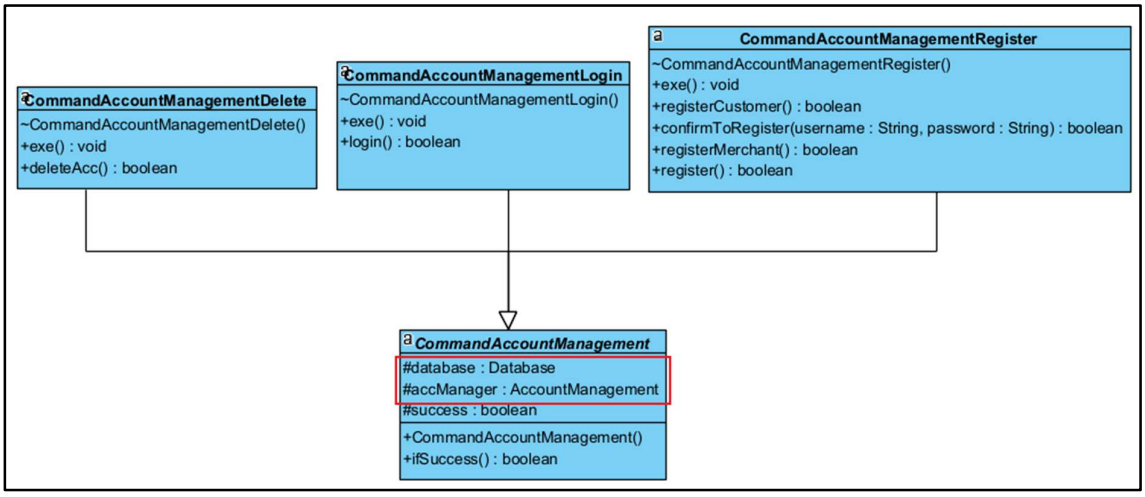
**Method:** Pull Up Field

Before refactoring:




Description:
Before code refactoring, the three classes have the same fields – Database and AccountManagement.

After refactoring:




Description:
After refactoring, we put the database field and AccountManagement field into the ***CommandAccountManagement*** class. And let the three classes mentioned in "before refactoring" extend ***CommandAccountManagement*** class.

### 7.4 Customer and Dine-in related

**Method:** Extract Method

Classes Related:

- ***CommandCustomerDineIn.java***

Before Refactoring:

```
140     public static void addCheckInAndWaitingInfo(String CId, ArrayList<Integer> checkInTable,
141             ArrayList<Integer> waitingTableNumList) {
142
143         // checkInTable append to customer occupiedtable
144         // add waitingtableNumList to customer's waitingtableNumList arrayList
145
146         for (int i : checkInTable) {
147             customer.addTableId(i);
148         }
149         if (waitingTableNumList ≠ null) {
150             for (int i : waitingTableNumList) {
151                 customer.addTableNumList(i);
152             }
153         }
154     }
```

Description:

Originally, the ***addCheckInAndWaitingInfo*** function includes two operations that are not related. If the two operations are written in one function, then we cannot perform separate operations. Therefore, this function can be separated into two sub-functions.

After Refactoring:

```
238     public static void addCheckInInfo(ArrayList<Integer> checkInTable) {
239         /*
240          * Add Check-in table into customer's occupiedtable
241          */
242         for (int i : checkInTable) {
243             customer.addTableId(i);
244         }
245     }
246
247     public static void addWaitingInfo(ArrayList<Integer> waitingTableNumList) {
248         /*
249          * Add waiting table into customer's waitTableNumList
250          */
251         if (waitingTableNumList ≠ null) {
252             for (int i : waitingTableNumList) {
253                 customer.addTableNumList(i);
254             }
255         }
256     }
```

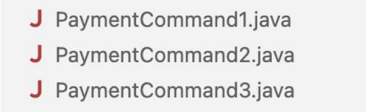
Description

After refactoring, the ***addCheckInAndWaitingInfo*** is separated into addCheckInInfo and addWaitingInfo. Hence, we can do each operation separately and reduce cohesion in between.

### 7.5 Payment-Related

**Method:** Better Naming
We did this throughout the refactoring processes and abandoned some bad naming variables that are named.

Before refactoring:

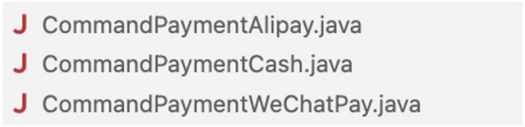



Description:
Before refactoring, the different payment methods, for instance, pay by cash, pay by Alipay, and pay by WeChat Pay, are named ***PaymentCommand1***, ***PaymentCommand2***, and ***PaymentCommand3***, which are very confusing, and may cause misunderstanding in development afterward.

```
59                  if (choice ≠ -1) {
60                      if (choice == 1) {
61                          Commands cmd1 = new PaymentCommand1(this,discountPrice);
62                          this.setCommand(cmd1);
63                          this.callCommand();
64                          break;
65                      } else if (choice == 2) {
66                          Commands cmd2 = new PaymentCommand2(this,discountPrice);
67                          this.setCommand(cmd2);
68                          this.callCommand();
69                          break;
70                      } else if (choice == 3) {
71                          PayFactory payFactory=new CashFactory();
72                          PaymentMethod paymentMethod=payFactory.getPay();
73                          boolean result=paymentMethod.pay(discountPrice);
74                          if(result){
75                              setPaymentStatus(result);
76                          }
77                          selectMerchantToPayment();
78                          break;
79                      } else {
80                          System.out.println(x: "\nInvalid Payment method, please try again.");
81                      }
82                  }
```

Description:
In ***Payment.java***, the three payment commands are executed, but there is no way that we know what it is actually doing by looking at the names only.

After Refactoring:




Description:

The different payment methods had been renamed to ***CommandPaymentAlipay***, ***CommandPaymentCash***, and ***CommandPaymentWeChatPay***.

```
78                  // 1. Alipay 2. Wechat Pay 3. Cash
79                  if (choice ≠ -1) {
80                      if (choice == 1) {
81                          Commands cmd1 = new CommandPaymentAlipay(this, discountPrice);
82                          this.setCommand(cmd1);
83                          this.callCommand();
84                          break;
85                      } else if (choice == 2) {
86                          Commands cmd2 = new CommandPaymentWeChatPay(this, discountPrice);
87                          this.setCommand(cmd2);
88                          this.callCommand();
89                          break;
90                      } else if (choice == 3) {
91                          Commands cmd3 = new CommandPaymentCash(this, discountPrice, customer);
92                          this.setCommand(cmd3);
93                          this.callCommand();
94                          break;
95                      } else {
96                          System.out.println(x: "\nInvalid Payment method, please try again.");
97                      }
98                  }
```

Description:

After refactoring, it is easier to understand what methods are used for payment in the ***Payment*** Class.

End of Report