Project Report

IS1220 - CentraleSupélec

Team n°5

Computer Science Project

MyFoodora - Food Delivery Service

Alexandre Carlier

Zexi Deng

December 2016

Contents

1 Introduction

Ordering a meal on-line has never been as easy as it is now. Only with a few clicks on your screen, a delicious meal will be delivered to your house just in time. We never thought about how this kind of system works until we must build our own ordering system, MyFoodora.

In MyFoodora, there are four kinds of user: *Customer, Restaurant, Courier, Manager*. In general, the customer can order meals from the menu given by the restaurants. Restaurant can edit its menu and can be notified by the system if there is a new order arriving. Within the system, registered couriers can be delegated to deliver the meal. And no matter what happens, the Manager can take complete control of system and set parameters to make it run as it should be.

Briefly, there are three main features in our project:

User-friendly In order to ensure every fresh user can use our system without any pressure, we carefully design each menu and command to make sure there are enough suggestions and feedbacks. We also carefully design our graphic user interface to simplify the operations but without losing any functions.

Reliability A good system should be robust. We carefully thought about all possible situations in our program and handle all of them cautiously. Thanks to the implementations of different kinds of design patterns and numerous practical features of Java, we can develop them without many difficulties.

Extensibility The key feature of object-oriented program is its extensibility, which allows us to reuse the code or extend the function easily. During our development, we always remembered the OPEN-CLOSE principle and applied it to our code design. For further development, we can also easily add new functions without much code refactoring.

2 Background

Git In order to cooperate better in our team work, we use *Git* to merge our codes. We never use this system before, but after finishing some tutorials on-line we gradually get start to use git in our work flow. And finally, it dramatically reduced a lot time to solve the conflicting code between us and helped us develop our program with high efficiency. Our project was uploaded to Github: `https://github.com/dennis101251/MyFoodora`, and you can see our whole process of developing.

Intellij IDEA We didn't use Eclipse as our default IDE tool, instead we use Intellij IDEA from Jetbrain. It has a lot of practical and intelligent features which allows us to develop very seamlessly. It's also able to export project into an Eclipse project. And it has a built-in UML designer which helps us a lot at the beginning of development.

3 Analysis and Design

3.1 User

According to the requirement, there are four types of users.

- Customer
- Restaurant
- Courier
- Manager

At the beginning, we constructed an abstract class named *User* to store the common attributes like *Username, Password, Name, ID, etc* and make it possible to invoke the polymorphism, which allows us handle them in a flexible way.

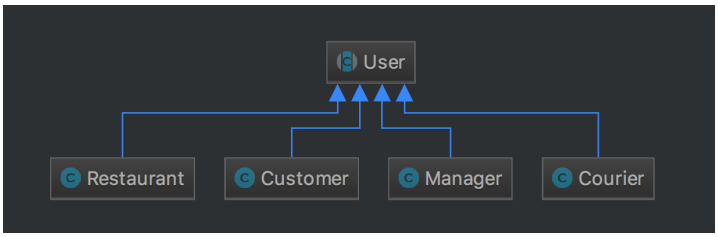


Figure 1: The inheritance structure of user

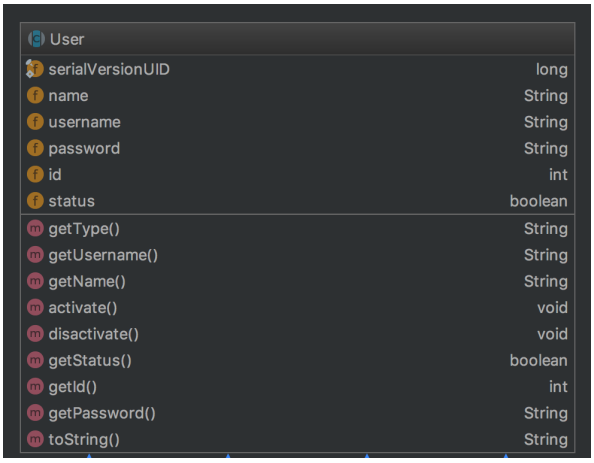


Figure 2: The UML of User

3.1.1 Customer

The most important function of a customer is to place an order, which was developed in the main system and here it only stores several variables. Two important variables needed to be point out are **the infoboard**, which stores the messages received from restaurants and **the fidelity card**, which would be invoked during the process of ending an order. The address is stored in the class Coordinate and we can use the API of Coordinate to calculate the distance.

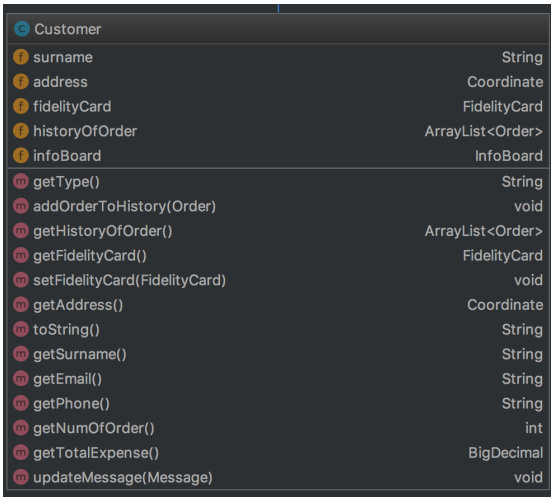


Figure 3: The UML of Customer

3.1.2 Restaurant

For a restaurant user, he can store all the **items** and **meals** as his private attributes. And we also provide a set of methods to let restaurant user edit and remove any kinds of items. And the class Item and Meal will be explained later. When an order is been placed, it can be added the list of Order History. It also can set some financial parameters.

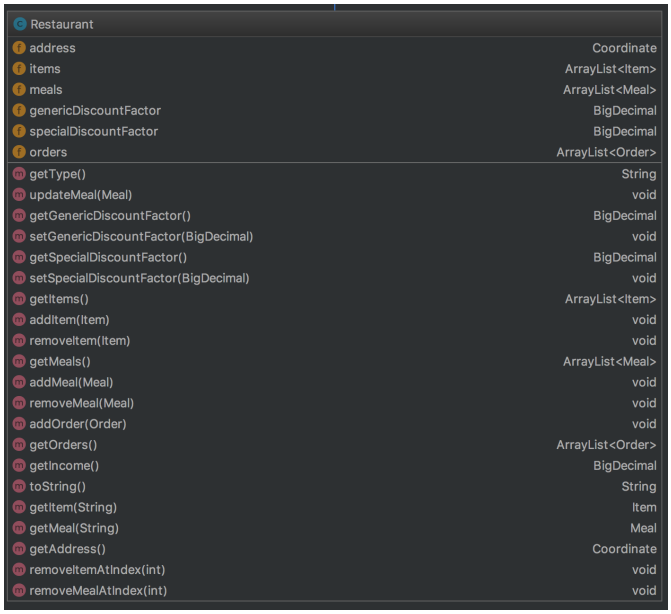


Figure 4: The UML of Restaurant

3.1.3 Courier

Courier is simpler than the first two users. Courier can accept or refuse the order delegated by the system and save the finished order. It can also change its working state and position. The price of delivery is set by system.

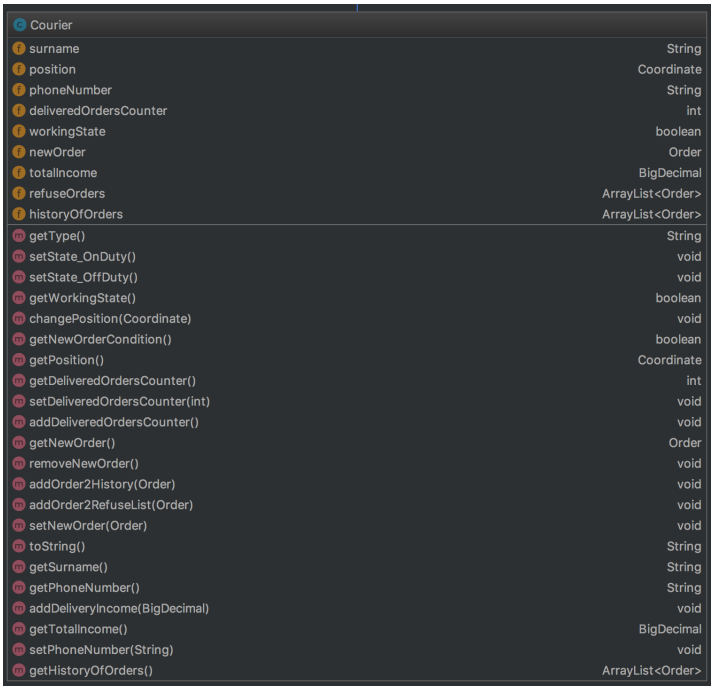


Figure 5: The UML of Courier

3.1.4 Manager

There are not many attributes in the class Manager, because most of attributes can be modified by manager are stored in the system. Our idea is deploying the operating methods at the main system and only giving manager authorization to invoke them.

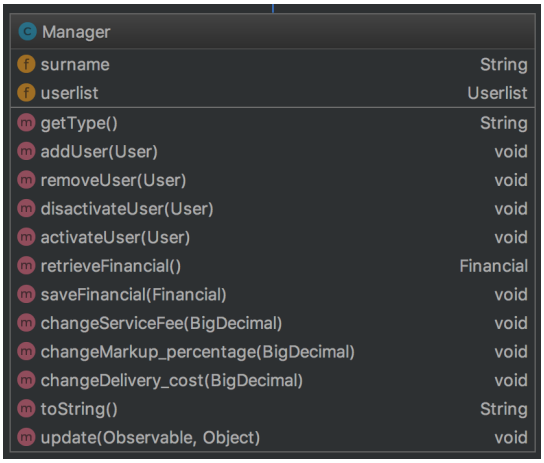


Figure 6: The UML of Manager

3.2 The Design of Storyboard

The storyboard connects all kinds of class in our system and define the definite ways of interacting between each other. So basically, the design of storyboard is also the design of our **MyFoodora** system. Our system has only one current user and he can only invoke the commands corresponding to his own type. Figure 7 shows the complete structure of our system.

The entrance of our program is the **login** module. If the user is the first time to login, it can choose the registering module to register as a new user. If the user has been registered before, he can login with his user name and password. When he logs in successfully, the system will show the login information which is corresponded with his user category.

3.2.1 Customer

The customer has three optional modules:

- Order meal
- Fidelity card and order history
- Message box

The most important thing of customer is placing an order which has three steps.

1. Choose a restaurant from the list
2. Add items to an order or remove items
3. End the order

And when a customer end the order, which means he has confirmed and payed the bill, the system will automatically send this order to the restaurant he has chosen and delegate a courier to deliver the meal. The system will automatically calculate the order price and imply the fidelity cad. And in **Fidelity card** module, the customer can see the state of his fidelity card and the history of ordering. In the **Message Box** module, he can check the promotion messages sent by the restaurants.

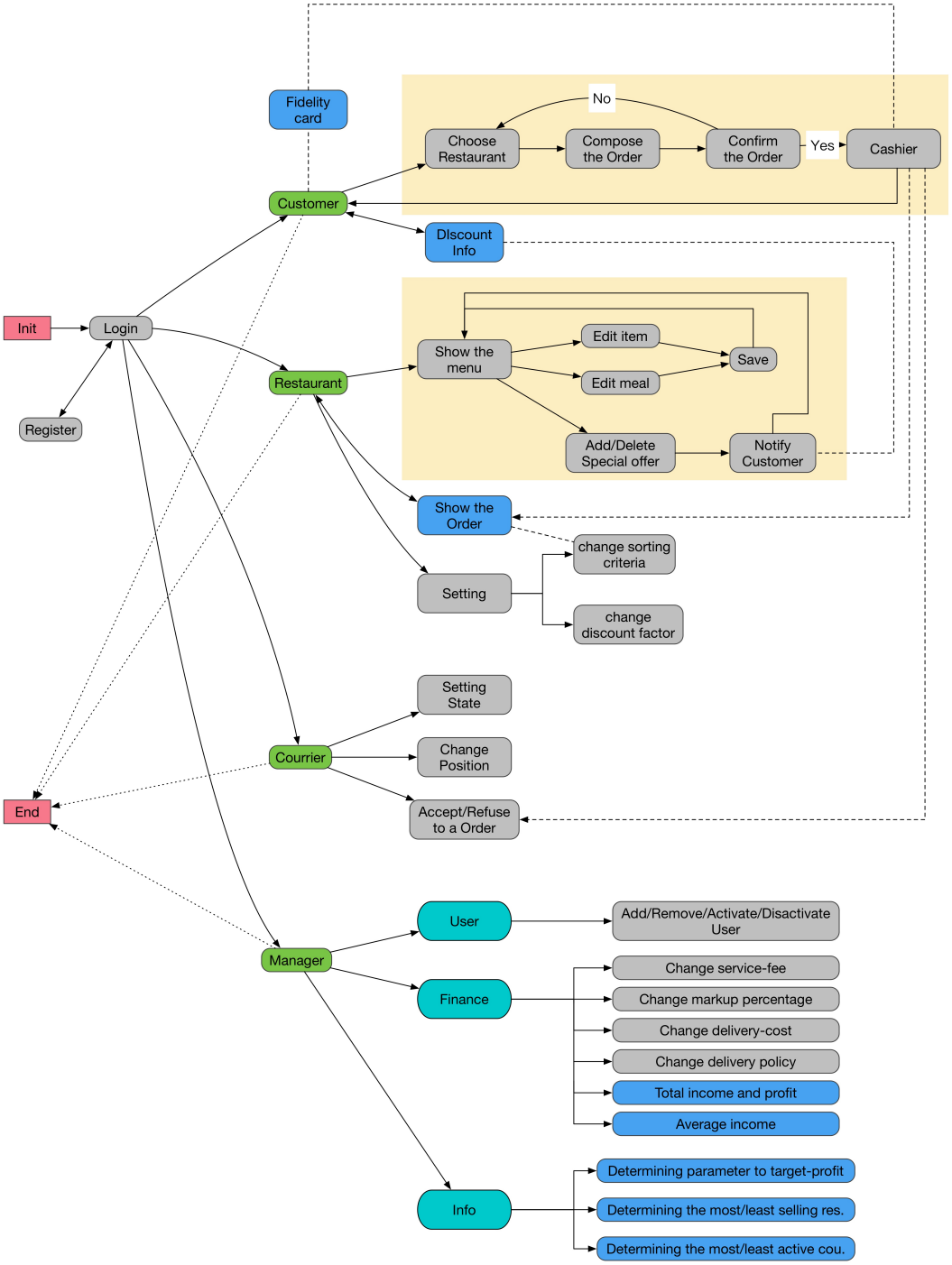


Figure 7: The general design of storyboard

3.2.2 Restaurant

The restaurant user can control every part of the restaurant. Normally he has to create different types of Item first and then add them to the Meal. Finally he can set a special offer for the meal and this action would send a message to customers automatically. He can also improve the quality or change the price with the help of items' statics.

3.2.3 Courier

When a new order is created by the customer, an appropriate courier will be delegated to deliver this order. For the courier user, he can accept or refuse this task. If he accepts, this order will be added to the courier's history and the courier will change his position(From the current position to restaurant's address, finally to customer's position). If he refuses, this order will be added to the refusing list and the system will find another courier to deliver this order. Of course, the courier can always set his working state as off duty if he has satisfied the money he made.

3.2.4 Manager

There are three parts for a manager user.

- User
- Finance
- Delivery

User A manager can activate or deactivate a user (means that this user can't login in system but the profile still exits in system). He can also remove a user totally from the system. More than this, he can see all the information about user, like the order history, contact information, etc.

Finance The profit of each order is defined in this formula:

$$\Pi_{Profit} = S_{Order} \cdot k_{Markup} + P_{Service\_fee} - D_{distance} \cdot P_{delivery\_price}$$

As shown in the formula, the manager can choose different values of k_{Markup}, $P_{Service\_fee}$ and $P_{delivery\_price}$ to meet the target profit. Each parameter can be calculated according to previous data. At the mean time, the manager can obtain all the necessary data about the system like average income, total income, etc.

Delivery The manager can choose different kinds of delivery policies to delegate courier. And in the GUI, we also provide a map to show all users' locations.

3.3 Design Pattern

With the help of design pattern of object-oriented program, we can not only solve our problems in a standard way but also increase the flexibility in our program.

3.3.1 Singleton Pattern

Our system can only be instantiated once, so we have to use the singleton pattern.

```
private static MyFoodoraSystem onlySystem;

public static MyFoodoraSystem getIstanceOfSysteme(){
    if (onlySystem == null){
        onlySystem = new MyFoodoraSystem();
        return onlySystem;
    }
    else {
        return onlySystem;
    }
}
```

And later in the development of GUI, there are some frames like message board which should only be created once. And the best way to achieve this object is also the singleton pattern.

3.3.2 Observer Pattern

The observer pattern maybe is the most essential pattern among them all and we used a lot in our project. Mainly observer pattern provides a general solution for sharing data between different classes. In this way, we can develop a more flexible program. There are many examples which use the observer pattern. For instance, in the part of GUI, we have several panels which needs to interact between each other. At this time, we have to use observer pattern to let them share data. In general, we define an operable panel as observable, and add other containers to the operable panel's observer list. Then we define an update method in the observer class to .

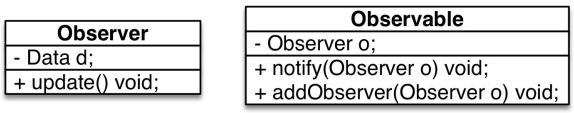


Figure 8: The UML of Observer and Observable

3.3.3 Strategy Pattern

In our system, there are many policies we can choose and the best way to switch among the different policies is using the strategy pattern. For example, the manager has to be able to choose different kinds of delivery policies to delegate courier. The best way to achieve this object is using the strategy pattern. We provide two kinds of strategy to select an appropriate courier and it will choose a strategy depending what the manager has chosen.

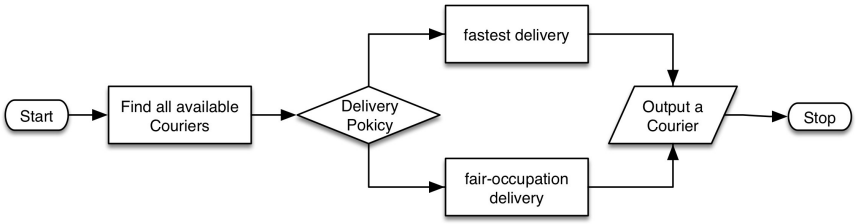


Figure 9: The process of delegating a courier

3.4 Serialization

In order to separate data and the program execution, we solidify our users' data in an external folder. Every time it initializes, it will retrieve all the related files (*Users, Orders, Financial*) from this place. Once the user modifies the objects, for example register a new user or place a new order, the system will update these files automatically. Therefore, there are three methods needed to be implemented:

- Retrieve method
- Save method
- Update method

3.5 Safety

When we serialize the objects, it also makes other people know the critical information like password and contact information. It is not safe to serialize plainly. So the best way to enhance security is to hash the information. The detail of implementation would be explained in the following parts.

3.6 Fidelity card

After analyzing the requirements, we construct an abstract class named FidelityCard and every kind of fidelity card is the inheritance of this class. With this design, we can invoke the fidelity card in a more general way and it simplifies operations when we calculate the order price through invoking the fidelity card.

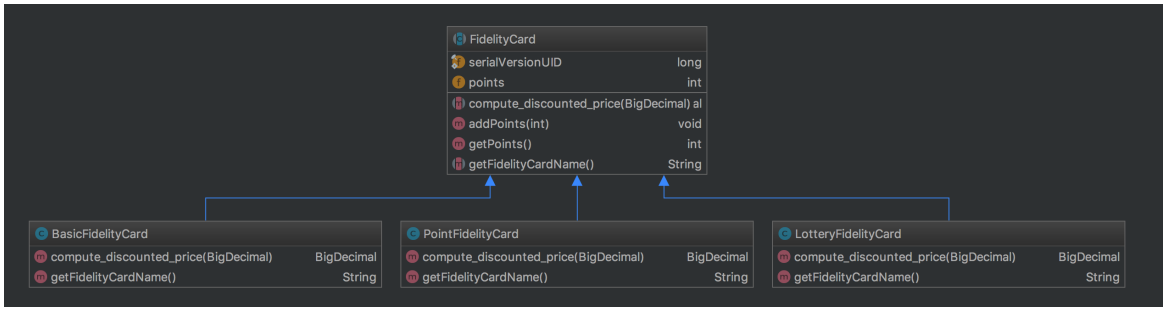


Figure 10: The design of fidelity card

As you can see, all the fidelity have the same field but have different kind of method to calculate the price.

3.7 Food

The classes about food are essential to the system and we thought very carefully about the design. First of all, in order to sort the list of item and meals in a more general way, we create the interface Food to deal with this problem. Then we create the class Item, which represents the basic element. The class Meal is made by several Items. The restaurant has the authority to modify the fields of Item and Meal and add or remove an Item from a meal. And finally we create the enumeration class to represent the category of meal and item.

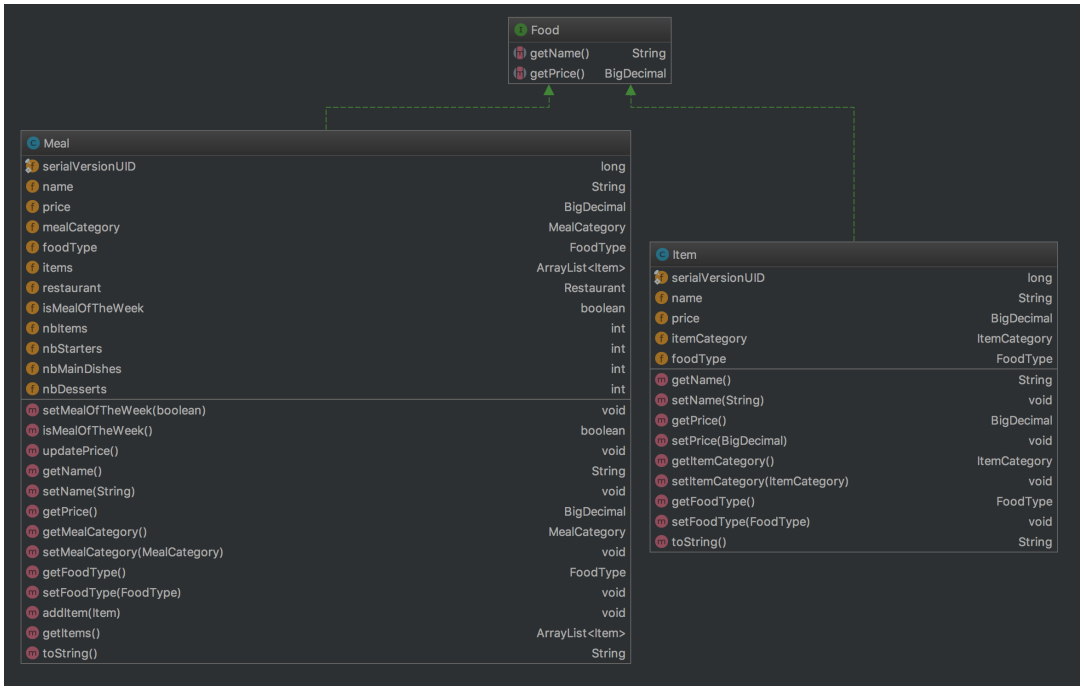


Figure 11: The UML of food

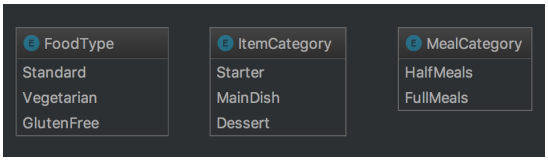


Figure 12: The enumeration of category

3.8 Order

In our system, the Order class is the key component to connect all the other parts. An order is created when a customer selected a restaurant. When the customer confirm and pay the order, the order would be saved into the system and notify the related restaurant. After accepted by the courier, the order would be noted as terminated. And for the financial issue, the manager can use the API of the order to calculate the income and profit easily. We also added the Calender class to record the date when the order was created.

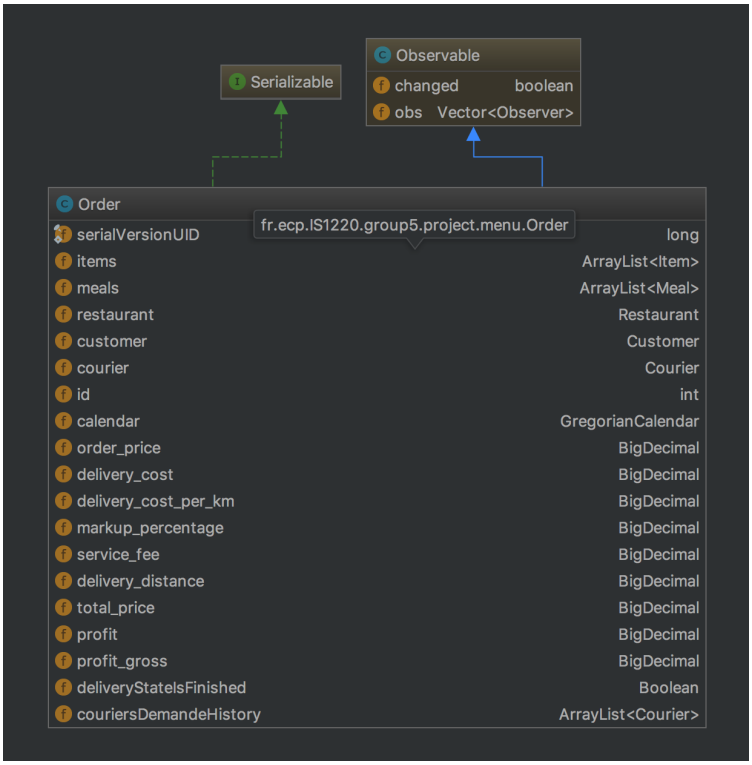


Figure 13: The UML of Order

3.9 Message System

According to the requirement, the restaurant can send messages to the customers who are willing to receive. At the first approach, we use the observer pattern and define a customer list within every restaurant object. But later we realize there is no need to save the customer list separately and it's more convenient to invoke the customer list directly in the main system. For sorting the message, we create the class Message that can store the title, the author and the main message. We also create a class named Infoboard to receive the message and we make the infoboard as a public attribute of Customer class.

3.10 The Defense of Input Error

Since our program has to interact with users, it needs to be able to handle all kinds of input error without any crush.

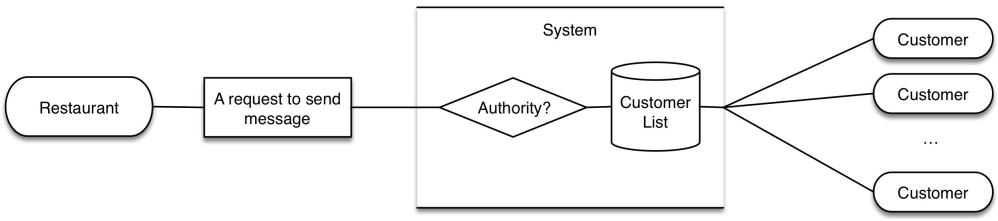


Figure 14: The process of sending message

3.10.1 Not enough inputs error

In the part of command line interface, user has to type commands into the program. If there are several arguments, it's possible to have this kind of error. And solution is simple: count the number of arguments and compare it with the requirement. For example, when a user registers as the *Restaurant* user, he has to give enough arguments.

```
case "registerRestaurant":
    if (commands.length == 5) {
        myFoodoraSystem.registerRestaurant(commands[1], commands[2],
            String2Coordinate(commands[3]), commands[4]);
    } else {
        System.out.println("not enough input");
    }
    break;
```

3.10.2 Invalid input error

This kind of error is also very common. When a textfield should be given a number, the user typed in an alphabet and then the program crushed. The solution is not also very difficult. In the command line interface, we should verify the type of argument. And in the GUI, we should use the KeyListener to defend the error directly.

```
targetProfitText = new JTextField(myFoodoraSystem.getTarget_profit().toString());
targetProfitText.addKeyListener(new KeyAdapter() {
    @Override
    public void keyTyped(KeyEvent e) {
        int keyChar = e.getKeyChar();

        if((keyChar >= KeyEvent.VK_0 && keyChar <= KeyEvent.VK_9)||(keyChar == '.')){
            if (keyChar == '.'){
                if (!point){
                    point = true;
                }
                else {
                    e.consume();
                }
            }
        }else{
            e.consume();
        }
        String input = targetProfitText.getText();
        if (input.contains(".")){
            point = true;
        }
        else {
            point = false;
        }
    }
});
```

This is a good example to show how it works. When a manager wants to set a new target profit, the input has to meet two requirements. 1) The input should be numbers. 2) If there is a decimal mark, there should be only one in the input.

3.10.3 Logical error

Some inputs are valid apparently, but the program can't achieve the objects with these inputs because there are logical errors. Some errors are obvious, like registering a username which has already existed or typing in the wrong password. But some errors are not so obvious like that. For example, when the manager wants to determine the delivery price by the previous results, if the target profit he set is higher than the gross profit ($S_{Order} \cdot k_{Markup} + P_{Service\_fee}$), it's not possible reach the target profit with the change of delivery price since the delivery price can't be negative.

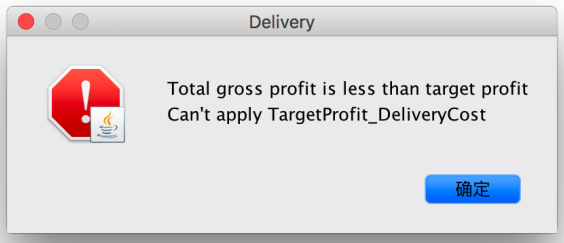


Figure 15: The warning message

4 Implementation

In this part, we will discuss the most interesting code details.

All our classes are arranged in a dedicated package, which are:

- exception : contains customs subclasses of Exception, including DuplicateNameException (which is thrown when someone tries to register with an already existing username), or TooManyItemsExceptions (which is thrown for instance when a HalfMeal contains more than two dishes).
- fidelity : contains the classes that represent the different fidelity programs a customer can join.
- GUI : contains all the JFrames of the Graphical User Interface
- menu : contains all the classes related to the content of an order: items, meals (collection of items), and the Order class itself.
- test : contains all our JUnit tests.
- user : contains all our User classes (Customer, Restaurant, Courier and Manager).
- util : several classes that are helpful for our program.

In order to run our program, a user has to run either the CommandLine class (for the CLUI) or the Login class (for the GUI).

4.1 Storing prices with BigDecimal

In order to manipulate price values, stored values have to be very precise. This isn't the case for primitive types such as float or double.

```
double value = 10*0.09;
System.out.println(value); //output: 0.8999999999999999
```

As a consequence, we store prices as BigDecimal instances. This enables us to perform simple operations (e.g. to compute the total price of an order).

```
BigDecimal price1 = new BigDecimal("2.50");
BigDecimal price2 = new BigDecimal("1.40");
BigDecimal result = price1.add(price2);
System.out.println(result); //output: 3.90
```

Finally, to display a price in the French format, we only have to write:

```
String formattedPrice = NumberFormat.getCurrencyInstance(Locale.FRANCE).format(result);
System.out.println(formattedPrice); //output: 3,90 €
```

4.2 PBKDF2 Password hashing

Passwords must never be stored as they are in a database. A solution to this problem is to hash the password, then to store the hashed value and finally to compare this value with the hashed input of a user who wants to log in to the system.

The algorithm we have used for hashing password is called PBKDF2, which stands for Password-Based Key Derivation Function 2. It applies a pseudorandom funtion to the input password along with a salt value and repeats the process a huge number of times in order to generate a derived key (1000 times in our program).

Thus, the following code appears in the constructor of the User class.

```
this.password = PasswordHash.createHash(password);
```

Finally, in order to log in a user, we call the static method validatePassword, which compares the good hash to the user's hashed input.

```
if (PasswordHash.validatePassword(inputPassword, user.getPassword()){
   System.out.println("Welcome to MyFoodora");
} else {
   System.out.println("Wrong password");
}
```

4.3 Sort

In order to perform an effective sort, we used the already implemented Collections.sort(List<T> list, Comparator<? super T> c) method, with a $\mathcal{O}(n \cdot \log(n))$ complexity.

To use this method, we had to implement several sort comparators; an example is given here:

```
public class SortByIncomeDown implements Comparator<Restaurant> {
    @Override
    public int compare(Restaurant o1, Restaurant o2) {
        return (int) (o2.getIncome().doubleValue() - o1.getIncome().doubleValue());
    }
}
```

4.4 Calender

According to the requirements, we have to calculate the finance in a given period. So we have to define a field in the Order class to record the date when the order is created. Fortunately, in the java.util, we found the class **GregorianCalendar** which can perfectly achieve this object.

Recording In the default constructor of GregorianCalendar, when a new Calender is instantiated, it can record the creating time automatically.

```
this.calendar = new GregorianCalendar();
```

Comparing Java provides a set of practical API for the GregorianCalendar class. According to the requirement, we have to compare our date with a given date. So there are two step: 1) Create a Calender by a given date. 2) Compare these two dates. The first one is simple, just use the constructor to create date.

```
public class Date {
    static public GregorianCalendar date(String string){
        String[] commands = string.split(",");
        return new
            GregorianCalendar(Integer.parseInt(commands[0]),Integer.parseInt(commands[1])
            - 1, Integer.parseInt(commands[2]));
    }
}
```

The second has to use the Date class, which is also provided by Java. Date can compare two dates and return a boolean value. And the GregorianCalendar is a subclass of Date, so it can also use this API.

```
order.getDate().getTime().before(calendar2.getTime());//getTime()return the Date
order.getDate().getTime().after(calendar1.getTime());
```

4.5 Delegation process of orders to couriers

Because the delegation process of orders to couriers is a bit tricky, we will explain it in detail:

1. Send the customer's order as a *waiting order* to the order list in the system after the customer has paid the order
2. The system tries to find a courier according to different policies:
   - Find the available couriers that are:
     - On duty
     - Doesn't have another order to do
     - Has not been notified for this order before
   - Sort the couriers according to the currently selected policy:
     - The least distance policy:
       - Calculate the distance between the courier and the restaurant
       - Return the sorted list of all the couriers with the least distance on the top
     - The fair delegating policy:
       - Calculate the number of orders which has been delivered by every courier
       - Return the sorted list with the least number of finished orders on the top
3. Delegation
   - If a courier has been found:
     - Notify this courier that he has a new order to deliver.

- Memorize that this courier has been notified by this order, so that the system won't notify the courier with this order again

- If no courier has been found:
  - Send a message to the manager that no courier has been found for this order
  - Leave this order's state as *waiting*

4. Decision made by the courier

   - If the courier accepts the order:
     - Connect the courier to the order
       - Find the saved order in the system and update its state
       - Add this order to the courier's history of orders
     - Update the counter of delivered order
     - Change the position
   - If the courier refuses the order:
     - Add this order to the refusing list
     - Call the system to delegate a new courier for this order

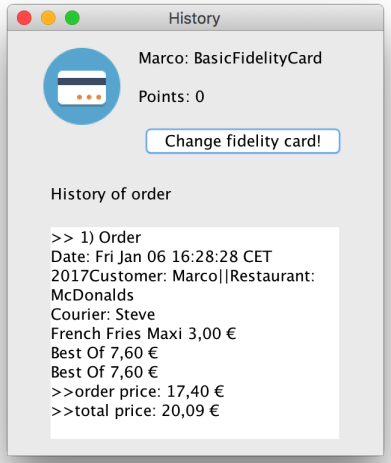
4.6 GUI classes

We spent the last couple of weeks to develop a nice looking, user-friendly Graphical User Interface, which includes a Login window, a Register window and a different Dashboard for every type of user (enabling to show the available operations only to the right category of user). Those classes can be found in the GUI package (apart from the Login class).

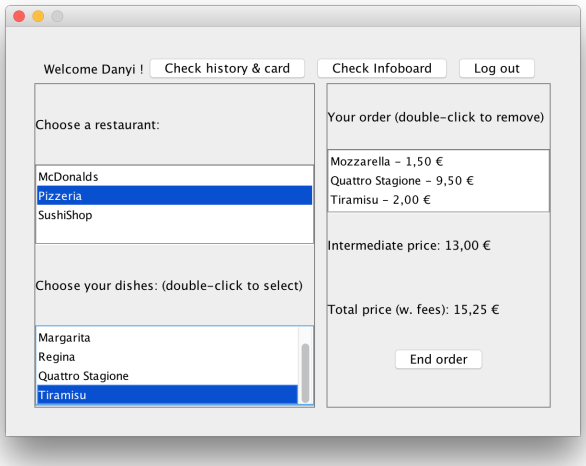
This part requested a lot of work as we weren't initially familiar to GUI development. So we looked to online documentation for Swing frames, panels and various components.

Last but not least, it is important to notice that we applied the Singleton pattern to MyFoodoraSystemGUI, in order to have the same reference of MyFoodora's system when we switch from one window to another.

(a) Menu panel

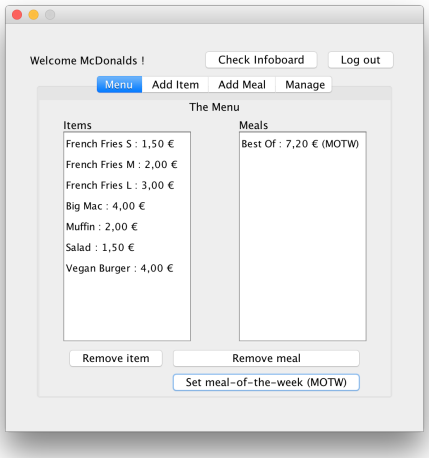


(b) Add Item panel

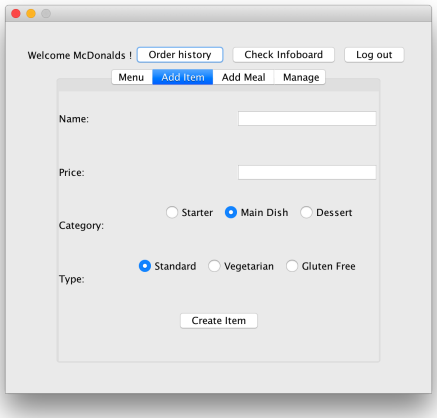


(c) Add Item panel

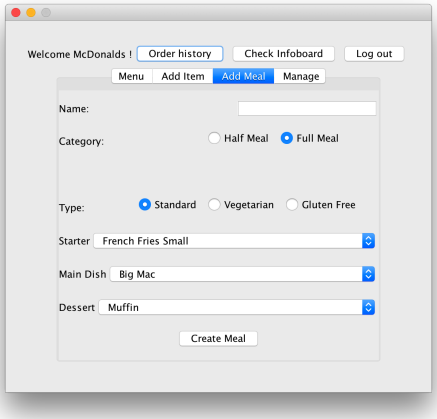
Figure 16: The Customer Dashboard



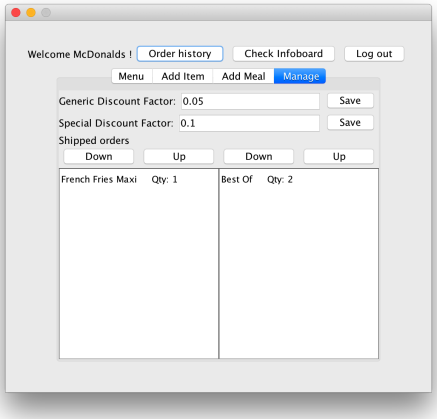
(a) Menu panel



(b) Add Item panel



(c) Add Meal panel



(d) Manage panel

Figure 17: The Restaurant Dashboard

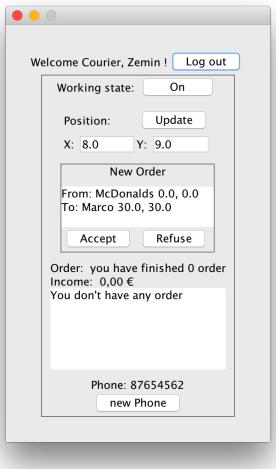
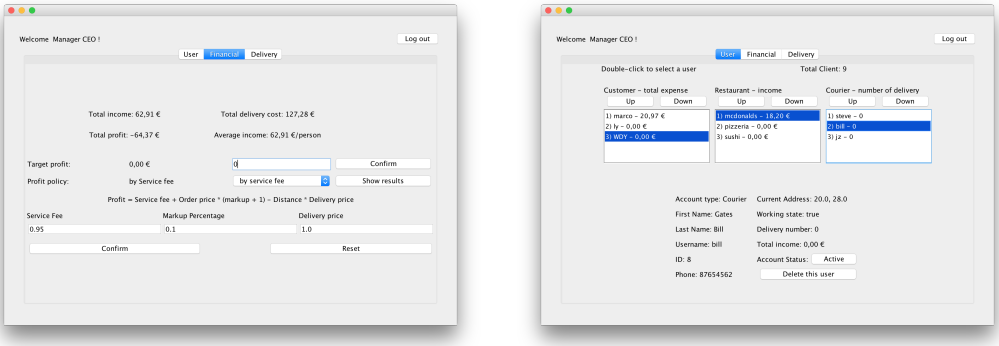


Figure 18: The Courier Dashboard



(a) User panel

(b) Financial panel

(c) Delivery panel

Figure 19: The Manager Dashboard

4.7 Making the Order observable

In the Customer's dashboard, when a customer adds an item or a meal to his order, the right panel (which summarizes the content and price of the order) must be updated automatically. To make this happen, we made the CustomerDashboard class implement the Observer interface and the Order class inherit from the Observable class.

```
public class CustomerDashboard extends JFrame implements Observer
public class Order extends Observable implements Serializable
```

In this way, we simply had to write the update method to update the content and the price of the order on the dashboard.

4.8 FreeChart

In order to give the manager a complete control and overview of where the users were, we have created a "world map" with the users locations by using the java chart library JFreeChart, as visible on figure 19c. You can find our class in GUI → managerDashboard → MapPanel.

Basically, we had to add the locations into a two-dimensional array, then add it to a DefaultXYDataset, and finally call the static createScatterPlot from ChartFactory:

```
ChartFactory.createScatterPlot("","","",xydataset, PlotOrientation.VERTICAL, false,
    false, false);
```

5 Testing

5.1 JUnit Testing

Even if Test Driven Development wasn't fundamentally our approach, we wrote all the JUnit tests for the most important methods. They are available in the "test" package.

5.2 Test Scenarios

We wrote two test scenarios:

- TestScenario1.txt : contains a list of commands to test the CLUI, according to a meaningful scenario (registering users of each type, creating restaurants' menus, simulating order placements by a few customers,
- TestScenario2.txt : contains a list of incorrect commands (syntax errors in the command name, wrong number of arguments after a command, etc.)

5.3 GUI testing

For GUI classes, it was much harder to test them properly as it requires a Human-Computer interaction, which can't be done with JUnit tests. As a consequence, we tried our program by hand on multiple examples that simulate the usage of the program by different kind of users and corrected the code when something didn't work as we expected. The next step of testing could be to send our program to several beta testers.

6 Results

According to the requirement, we create a txt file to test our system. And apparently it runs as we expect.

In general, there are 5 steps in the scenario:

1. Register all the users
2. Restaurant creates items and meals
    - Create items
    - Add items to a meal
    - Set the special offer
3. Customer places an order
4. Courier delivers the order
5. Manager reviews the performance
    - Set target profit
    - Determine the financial parameter
    - Show the statistic

Based on the output, our program meets all the requirements. However, there are still some limitations in our program: 1) We didn't consider very clearly about how the data of user is saved at the beginning of development, and we use a Userlist class to respond all the operation related with user like find a user or remove a user. But eventually it turned out this class is not necessary since we could edit directly in the main system. But since we found it too late, it was too difficult to do the code refracting. 2) The layout of GUI is really hard to control, and since this is first time we develop the graphic interface, we don't have much experience to create a elegant interface. If we have more time, we can develop much better.

7 Conclusion

The development of MyFoodora has been a very valuable experience for both of us. Putting into practice what we have learned during the course is definitely the best way to learn how indispensable design patterns are. Moreover, the knowledge of Git will be without doubt a very helpful skill for our future.

We think we have achieved the three objectives we had set, which are: a user-friendly interface, reliability and extensibility.

Appendices

Workload division

Task	Written by
Basic program structure	Alexandre
Infoboard	Zexi
Message system	Zexi
Fidelity cards	Alexandre
Sorts	Zexi
Order delegation & delivery	Zexi
CommandLine + MyFoodoraSystem	Alexandre + Zexi
GUI: Register + Customer & Restaurant Dashboard	Alexandre
GUI: Login + Courier & Manager Dashboard	Zexi
Exception handling	Alexandre
UML Diagrams	Zexi
Javadoc	Alexandre + Zexi
Testing	Alexandre + Zexi
Report: Background + Analysis & Design + Results	Zexi
Report: Implementation + Testing	Alexandre

Table 1: Workload division

System manual

Test results