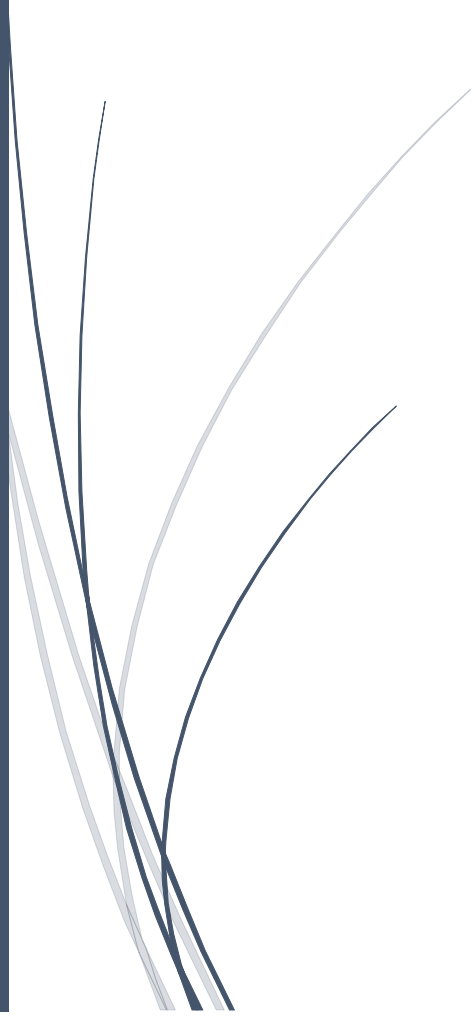
CentraleSupélec

07/05/2017

Project MyFoodora

Module IS1220 – Group 4



Raymond Ji & Xiaoan He

CENTRALESUPELEC 2016-2017

Index

Introduction

As the world moves rapidly towards a complete digital age, modern food delivery systems such as Foodora or Deliveroo, capable of providing food delivering services to everyone through internet within a very short delay, have become increasingly mainstream.

The goal of this project is to develop a software solution, called MyFoodora, whose functionality is similar to that of those nowadays Food Delivery Systems.

The project is written in *Java*, a pure object-oriented programming language, under the *Eclipse Papyrus* development environment.

The first part of this project is focused on the design and development of basic functionalities of the MyFoodora system, alias the MyFoodora core, while the second part of this project is focused on the user interface, consisting in command lines and GUI.

To launch the command line user interface, please execute the Main method contained in the src/clui/runCLUI.java file.

To launch the graphical user interface, please execute the Main method contained in the src/gui/runGUI.java file.

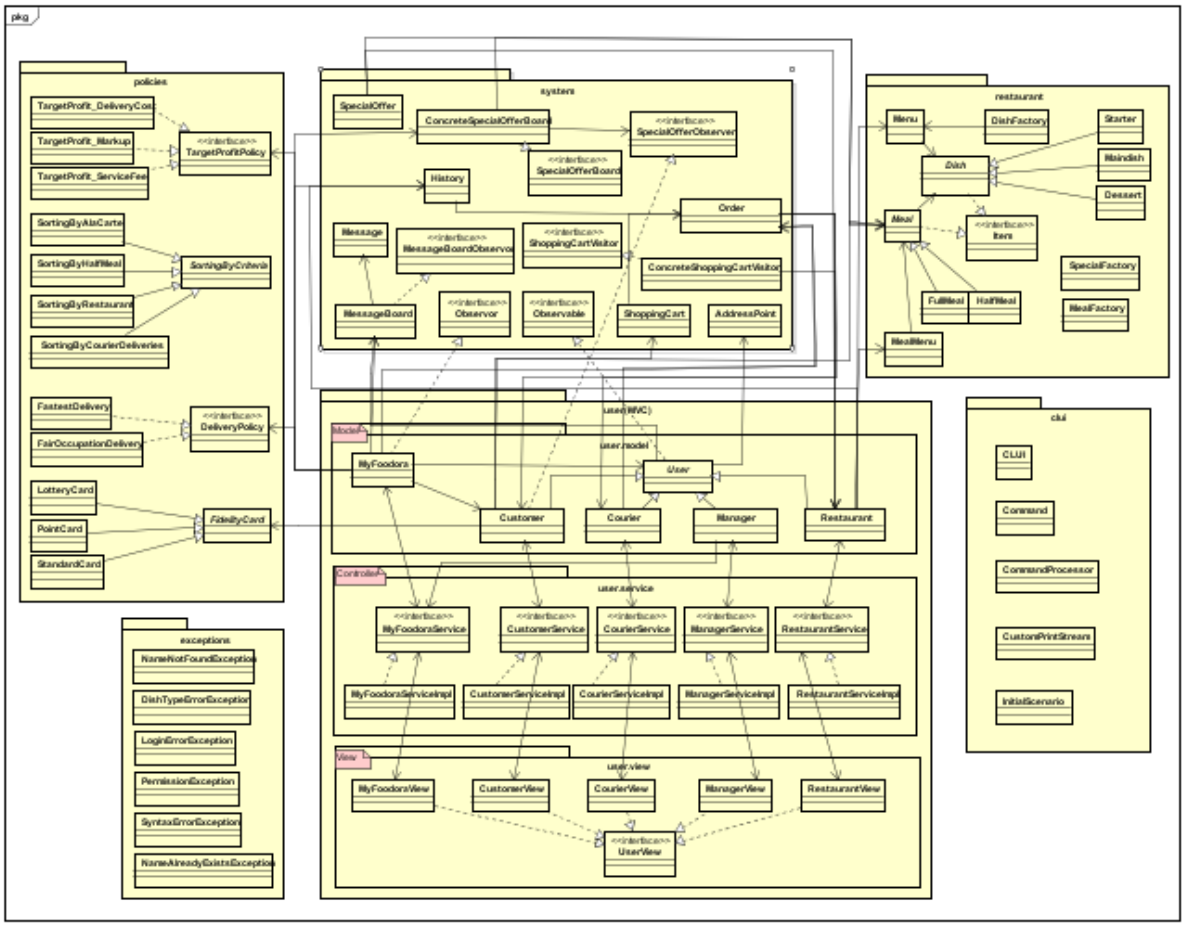
Design & UML

I. UML

For the UML, we used the Astah software, a popular UML tool. The installation file can be found within our project zip file. For more information about Astah please visit http://astah.net/

A complete UML diagram (.asta format) is submitted with the project, however it can only optimally viewed using the Astah program.

We have added the corresponding .pdf and .jpg files of the UML as well, even though the visibility may not be optimal. We only included simplified UMLs (without methods) in the report for better readability.



II. Detailed commentary

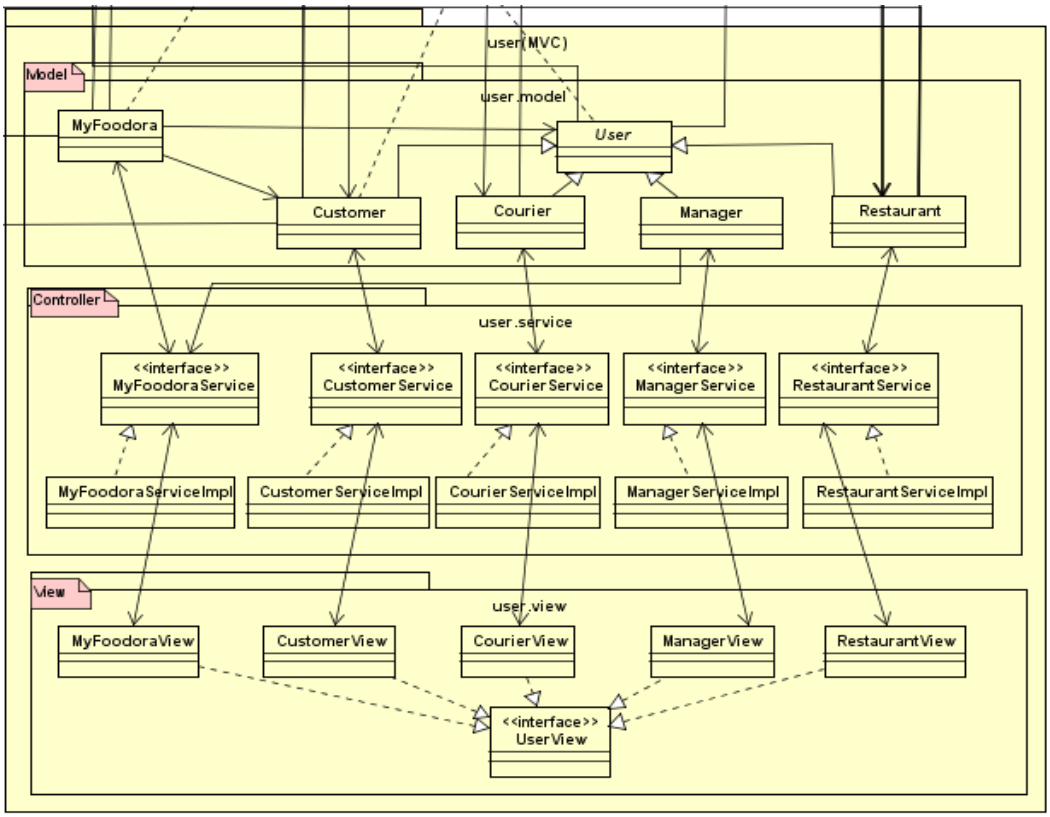
1. Package user

The core of the system, regrouping the 4 kinds of users (Customer, Courier, Manager,Restaurant) as well as the Myfoodora core class. The user classes all inherit from a super abstract class User.

The MyFoodora class is the core class of the system. It stores a list of users and is capable of basic functionalities as well as applying policies on different matters. The key components of the system (Order, history, message board…) are placed in a separate system package (package system).

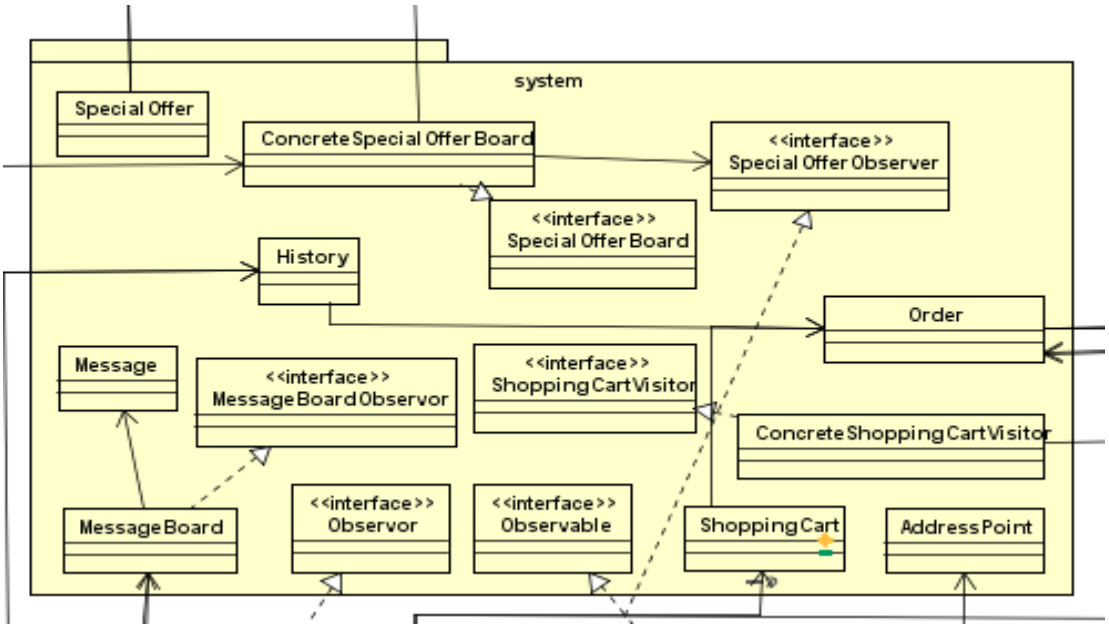
The Manager class is unique as the only class who has directly access to data from MyFoodora. The Customer class also inherits from the SpecialOfferObserver class which allows him to observe the special offers. The Courier class has a waiting order arraylist which stores his waiting orders. He can accept/refuse any waiting order. The Restaurant class also has a History of shipped orders. The key components of the restaurant (dishes, meals, menus…) are placed in a separate package (package restaurant).

The users are implemented following a MVC pattern.



2. Package system

Includes key tools for the functioning of the Myfoodora system, for example the ShoppingCart class, the Order class or the History class.



3. Package restaurant

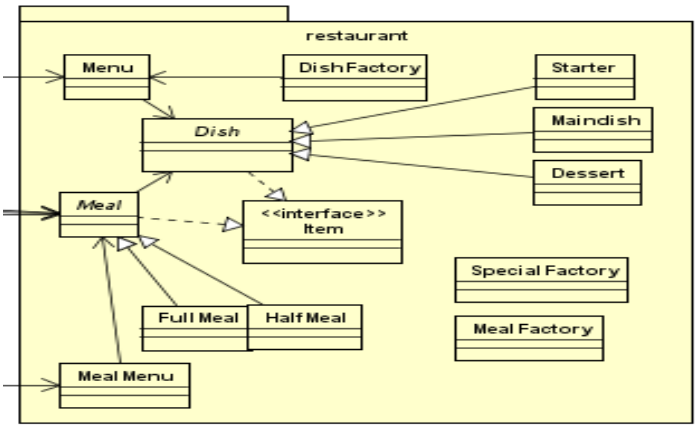
Classes that allows the set-up of a restaurant.

A restaurant can have Starters, MainDishes and Desserts which inherit from a Dish (=Item) superclass. The dishes are stored in a Menu class.

A restaurant can also have HalfMeals and FullMeals, which inherit from an abstract Meal class. They are composed by 2 or 3 Dishes and are stored in MealMenus.

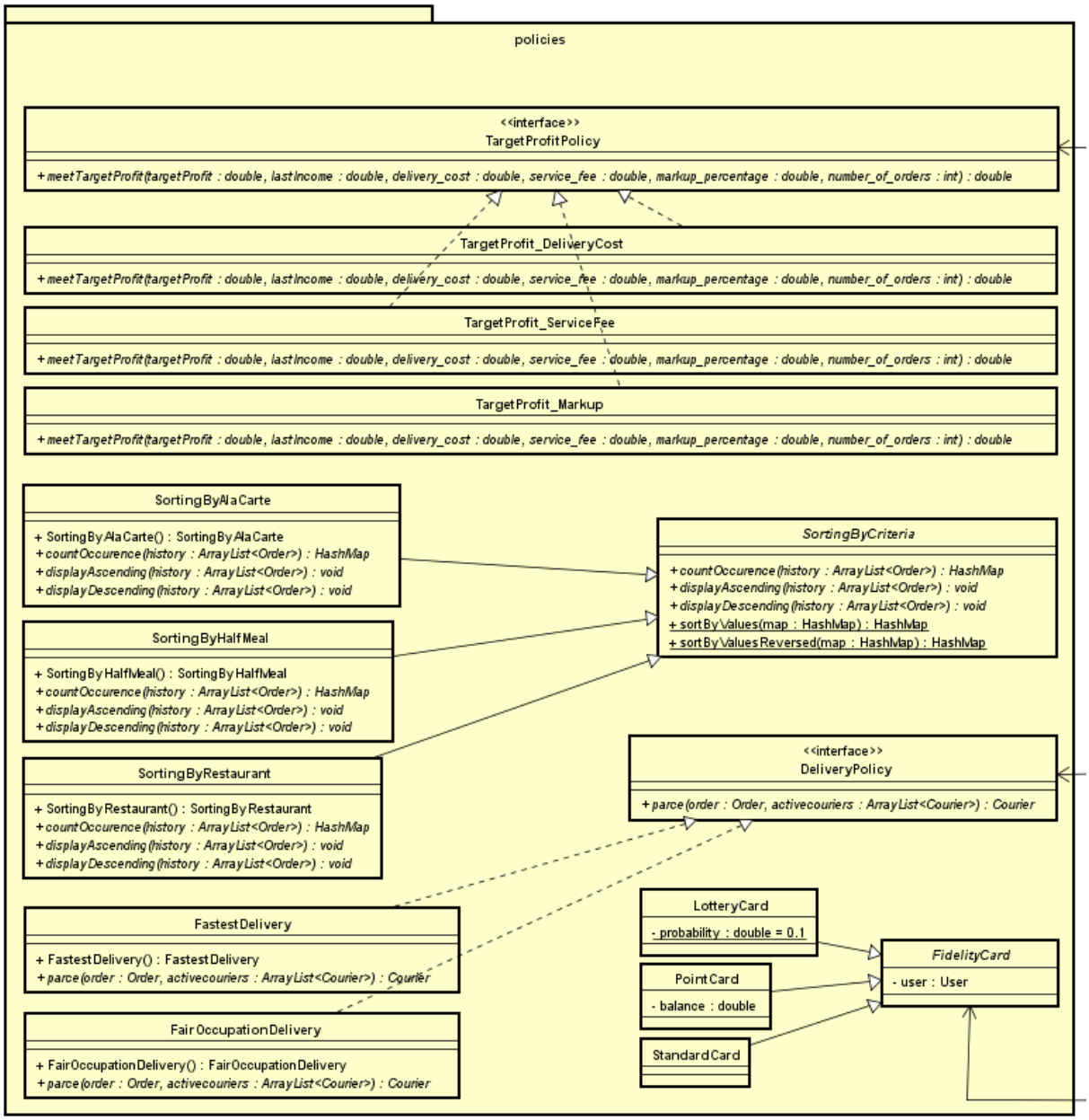
Therefore, a restaurant has 3 menus: a dish-menu (menu) for dishes, a meal-menu (mealmenu) for half-meals and full-meals and a meal-menu (specialmealmenu) for meal-of-the-weeks.

The production of the meals/dishes are done following a Factory Pattern that will be explained more in depth later.



4. Package policies

All policy/strategy related classes, including target profit policies, fidelity cards, delivery parsing policies and shipped order sorting policies.



5. Package exceptions

Includes all custom Exceptions. There are 6 custom exceptions:

- *NameFoundException*: thrown whenever a name is not recognized, for example when trying to order a dish which is not in the menu, or trying to finalize an order that is not in the shopping cart.
- *DishTypeErrorException*: thrown when a restaurant tries to save a meal that does not respect the following rule: 1 starter/dessert + 1 main-dish for a HalfMeal and 1 starter + 1 dessert + 1 main-dish for a FullMeal.

- *NameAlreadyExistsException*: a username must be unique in the MyFoodora database, mealnames and dishnames must be unique in the menu of a restaurant, and ordernames must be unique in the shopping cart of a customer. This exception is thrown if the contrary occurs.
- *LoginErrorException*: thrown when a username/password don't match.
- *SyntaxErrorException:* thrown when trying to enter a command that is not recognized in the CLUI.
- *PermissionException:* thrown when a user tries to use a command that he/she is not permitted to (for example when a customer tries to remove a user).

6. Package test

Software testing is mandatory because all programmers make mistakes. Some of those mistakes are unimportant, but some of them are expensive or dangerous. It is required to point out the defects, make sure of the customer's reliability and the software quality.

The JUnit tests are organized in packages corresponding to the code packages. For example, the test for user package's CourierService.java is named CourierServiceTest.java and is in the test.user packge.

As the MVC pattern is used for the key classes (Customer, Courier, Manager, Restaurant, MyFoodora), the most important tests are the tests of the Controller layers of these classes which contains the key functionalities of the MyFoodora project.

The JUnit test part of the project has already been completed in the 1st part, and the good functioning of the CLUI and the GUI also proves the correctness of the code.

7. Package clui

Contains the classes for the Command Line User Interface (CLUI).

The main method for CLUI is inside RunClui.class.

8. Package gui

Contains the classes for the graphical user interface (GUI). The code is divided into two packages, the gui.model package containing the basic models needed for graphical display and gui package containing the specific code for MyFoodora graphical user interface.

The main method for GUI is inside RunGUI.class.

III. Design patterns

In consideration of functional demands of MyFoodora system, design patterns below are used to follow the Open-close principle.

1. MVC pattern: Users & MyFoodora

The main functionalities of MyFoodora system are based on operations of Customer, Manager, Restaurant and Courier which are subclasses of the User class. The entities (layer Model) and functions provided by them (layer Controller) are separated using the MVC pattern:

- Model
  Responsible for representation of Java entities and data management
- Controller
  Responsible for forwarding requests and processing requests
- View
  Responsible for user interface.

Although using MVC pattern increase the code quantity, it has several advantages. It makes the java entities dependent only on the layer Model, which is generally unchanging. All entities provide their functions in the layer Controller. Therefore, the layer Model/View/Controller are out of binding and the degree of coupling decreases. This is better for extension and maintenance.

Consequently, for our project, all Users classes and the MyFoodora class are implemented using a MVC pattern combined with interface programming:

- *Manager.class ---- ManagerService.class ---- ManagerView.class*
- *Courier.class ---- CourierService.class ---- CourierViewl.class*
- *Customer.class ---- CustomerService.class ---- CustomerView.class*
- *Restaurant.class ---- RestaurantService.class ---- RestaurantView.class*
- *MyFoodora.class ---- MyFoodoraService.class ---- MyFoodoraView.class*

***Service.class are interfaces of functions provided by users and MyFoodora system and ***ServiceImpl.class are the corresponding concrete implementations.

2. Observer pattern: SpecialOfferBoard & MessageBoard

Whenever a new special offer is added to the special offer board of MyFoodora system, all customers who gave their consensus to receive news are notified on their personal Messageboard.

The Observer pattern is chosen for the implementation of these functionalities. The *Customer.class* implements a *SpecialOfferObserver (Observer) interface* which provides an updateNewOffer() method react to a new offer. The updateNewOffer() method posts a new message on the MessageBoard of the user through its update() method. It

interacts with an *SpecialOfferBoard (Observable)* interface who provides the notifyAll(), registerObserver() and addSpecialOffer() methods.

3. Abstract Factory pattern: DishFactory/MealFactory

The factory pattern fits here for the code system to produce dishes and meals of a restaurant. It is usual for a restaurant to produce an already existing dish/meal. If it wants to produce a new dish/meal, it can first add it to the menu/meal-menu.

Therefore, following the Open-Close principle, an abstract factory pattern is used. Three concrete factories inherit from the abstract factory: DishFactory, MealFactory and SpecialMealFactory. They take a String dishname/mealname as argument and look for the name in the menu/meal-menu to create the concrete dish/meal.

4. Singleton pattern: MyFoodora

The MyFoodora system, despite being unique, can be managed by multiple managers. The interruption of multi-threads and the safety of threads then must be taken into consideration. Not only methods such as internal/external lock should be used to avoid it, but also the MyFoodora should be designed as a Singleton. Consequently, the MyFoodora.class can only have one instance, obtained by its *getInstance*() static method.

To avoid potential thread interruption because of the gap between the object creation and object assignment, these operations are synchronized.

```
public class MyFoodora implements Observable{

    //Singleton Pattern
    private static MyFoodora instance = null;

    private MyFoodora(){

    };

    private static synchronized void syncInit(){
        if(instance==null){
            instance = new MyFoodora();
        }
    }

    public static MyFoodora getInstance(){
        //if no instance of myfoodora exists, re
        if(instance == null){
            syncInit();
        }
        return instance;
    }

    public Object readResolve(){
        return instance;
    }
```

5. Strategy pattern: Target profit policies

MyFoodora dynamically computes the parameter to change (service fee/delivery cost/mark-up percentage) to meet a given target profit. The system should allow for adding new implementations of the policies and selecting a new at the run time. That's why, a Strategy pattern is used for target profit policies.

6. Strategy pattern: Shipped order sorting policies

Managers can dynamically use different sorting strategies without modifying the code too much to compute a most/least active restaurant/courier. Restaurants can as well display their shipping history their menu sorted in an ascending/descending order w.r.t the number of shipped half-meals/a-la-carte orders.

That's why, a strategy pattern is used for sorting policies. To store the number of shipped orders of a certain meal/dish, we used the HashMap interface of Java.

A History class, implemented by MyFoodora and Restaurant, is also written to store the shipped orders and decreases the degree of coupling.

7. Strategy pattern: Delivery policies

MyFoodora dynamically uses the delivery policy to allocate orders to couriers, allowing for adding new implementations of the policies and selecting a new at the run time. Therefore, a Strategy pattern is also used for delivery policies.

8. Strategy pattern: Fidelity cards

MyFoodora dynamically computes the total price to pay for a customer and consider whether he/she has a Point Fidelity card or a Lottery Card. The Fidelity card system should allow for adding new implementations of the cards and selecting a new at the run time. Therefore, a Strategy pattern is used for fidelity cards.

9. Visitor pattern: Shopping cart

Each dish/meal/order has its own price, if there is a common method for them calculate their price, the code will be more concise and more extensible. That's why the visitor pattern is used. The price of the dish/meal/order(Visitable) is called by the visit() method of ShoppingCartVisitor(Visitor).

Teamwork

The project is the result of teamwork between the two members of the project team.

GitHub has been used to put together the works:
https://github.com/raym96/raymxiaoan

Team work chart

Coding & Testing

TASK	CODING	TESTING
Restaurant MVC	Xiaoan & Raymond	Xiaoan
Customer MVC	Raymond	Xiaoan
Manager MVC	Raymond	Xiaoan
Courier MVC	Xiaoan	Raymond
MyFoodora MVC	Xiaoan	Raymond
Package system	Xiaoan & Raymond	Xiaoan
Package restaurant	Raymond	Xiaoan
Policies	Xiaoan & Raymond	Xiaoan
CLUI	Raymond	Xiaoan
GUI	Xiaoan	Raymond
testScenario	Raymond	Xiaoan
Javadoc	Xiaoan	----------
UML	Xiaoan	----------

Report writing

TASK	Author
Introduction	Raymond
Structure	Xiaoan
Design Patterns	Xiaoan & Raymond
CLUI	Raymond
GUI	Xiaoan
Scenarios	Raymond
Conclusion	Raymond

CLUI: Command Line User Interface

The main method to be executed to launch the command line user interface is contained in **src/clui/runCLUI.java**

Once started, you may enter "help" into the CLUI to access detailed information about available commands.

Initial configuration

The initial configuration contains 5 restaurants, 2 managers, 4 couriers, 7 customers, 6 dishes per restaurant, 9 meals (4 half-meal, 4 full-meal, 1 special-meal) per restaurant and 40 completed orders in the history.

The one-line commands needed to configure the initial state of the system are contained in the **my_foodora.ini** file and are automatically loaded.

A detailed description (list of users with info, list of menus, list of past orders) can be found in **my_foodora_description.txt**.

Every user, except for Manager <ceo> (password = 123456789), uses the default password "password".

Test scenarios

The test scenarios can be loaded by entering the command "runTest testScenarioN.txt" with N = 1,2,3,4 in the command line user interface contained in runCLUI.java.

The output, while being displayed on screen, is also stored in the file testScenarioNoutput.txt contained in the folder testScenario.

Please find below a detailed description of what each test scenario does.

Scenario 1: Creation of users, menus and orders & error handling

To run this test please enter in the Eclipse console: ***runTest testScenario1.txt***

A manager (user <ceo> password 123456789) logs into MyFoodora. Registers 1 customer (Herve Biausser), 1 restaurant (Bonheur d'Antony), 1 courier (John Cagnol). Tries to register a customer with an already taken username, an error message is displayed. Displays the list of users. Logs out.

The new restaurant Bonheur d'Antony logs into MyFoodora. Adds 3 dishes to the restaurant, create a new meal "BA01", adds 3 dishes to the meal, sets the meal "BA01" as

a Meal-of-the-week. Creates a new meal “BA02”, tries to add 2 starters to the meal, an error message is displayed. Displays the menu. Logs out.

The new customer Herve Biausser logs into MyFoodora. Displays all restaurant menus. Places creates a new order “myorder” containing several items at Italian Restaurant <italian>. Pays for “myorder”. Logs out.

Scenario 2: System management by a Manager

To run this test please enter in the Eclipse console: ***runTest testScenario2.txt***

A manager (user <ceo> password 123456789) logs into MyFoodora. Deletes the customer account <mlepen>. Displays the list of users. Displays the list of couriers. Displays the menu of Chinese Restaurant <chinese>.

The manager sets the delivery policy as Fair Occupation Delivery Policy. Shows the system policies.

Computes the total income, profit & average income per customer of the system over the creation and over a period.

Changes the delivery cost to 1.5. Sets the target profit policy as Service fee policy. Applies the target profit policy. Shows the newly computed system values. Computes the system profit over the last month.

Displays the best-selling restaurants in a descending/ascending order. Displays the best couriers w.r.t the number of delivered orders in a descending/ascending order. Logs out.

Scenario 3: Completion of an order, delivery task assignment

To run this test please enter in the Eclipse console: ***runTest testScenario3.txt***

Customer <emacron> logs into MyFoodora. Checks his message board. Registers for a Point fidelity card. Displays all restaurants with menus.

Places an order “myorder” at French Restaurant (user <french>) containing several items. Places an order “myorder2” at Chinese Restaurant (user <chinese>) containing several items. Displays the shopping cart. Pays for both orders. The orders are automatically assigned to the courier w.r.t the delivery policy (by default the delivery policy is set as Fastest Delivery Policy). The points earned on the balance of the fidelity card are displayed. The customer logs out.

The courier assigned to the order “myorder” at French Restaurant (Donald Trump <dtrump>) logs into MyFoodora. Displays the list of waiting orders containing the new order. Accepts the order. Shows history of delivered orders. Shows delivery count. Sets state as off-duty. Changes position. Displays the updated personal information. Logs out.

The courier assigned to the order “myorder2” at Chinese Restaurant (Hillary Clinton <hclinton>) logs into MyFoodora. Displays the list of waiting orders containing the new order. Refuses the order “myorder2”. A new courier (Barack Obama <bobama>) has been automatically assigned to the order w.r.t the delivery policy. Logs out.

Courier <boboma> logs in. Accepts the order at Chinese Restaurant. Logs out.

Scenario 4: Creating/removing a special-offer, customer notification

To run this test please enter in the Eclipse console: ***runTest testScenario4.txt***

Customer Francois Fillon <ffillon> logs into MyFoodora. Registers for a Lottery Card. Turns on notifications about special offers. Checks his message board. Logs out.

Korean Restaurant <korean> logs into MyFoodora. Creates a meal “Hotpot”. Sets the new created meal as a special-offer. Displays the new menu. Logout.

Logon again as <ffillon>. He has received a notification (new message) of new special-offer. Displays the message board.

Displays the list of available Special-offers. Places an order “myorder” on the new special-offer of Korean Restaurant, “Hotpot”. Pays for the order “myorder”.

French Restaurant <french> logs into MyFoodora. Displays menu. Removes its special offer <I_love_paris>. Displays menu. Notices that the price has been automatically updated (generic discount factor applies instead of special discount factor). User <french> logs out.

User registration

The test of user registration (not by a manager: a user connects into MyFoodora and registers for an account) is different from the other tests because it requires active input from the user into the Eclipse console.

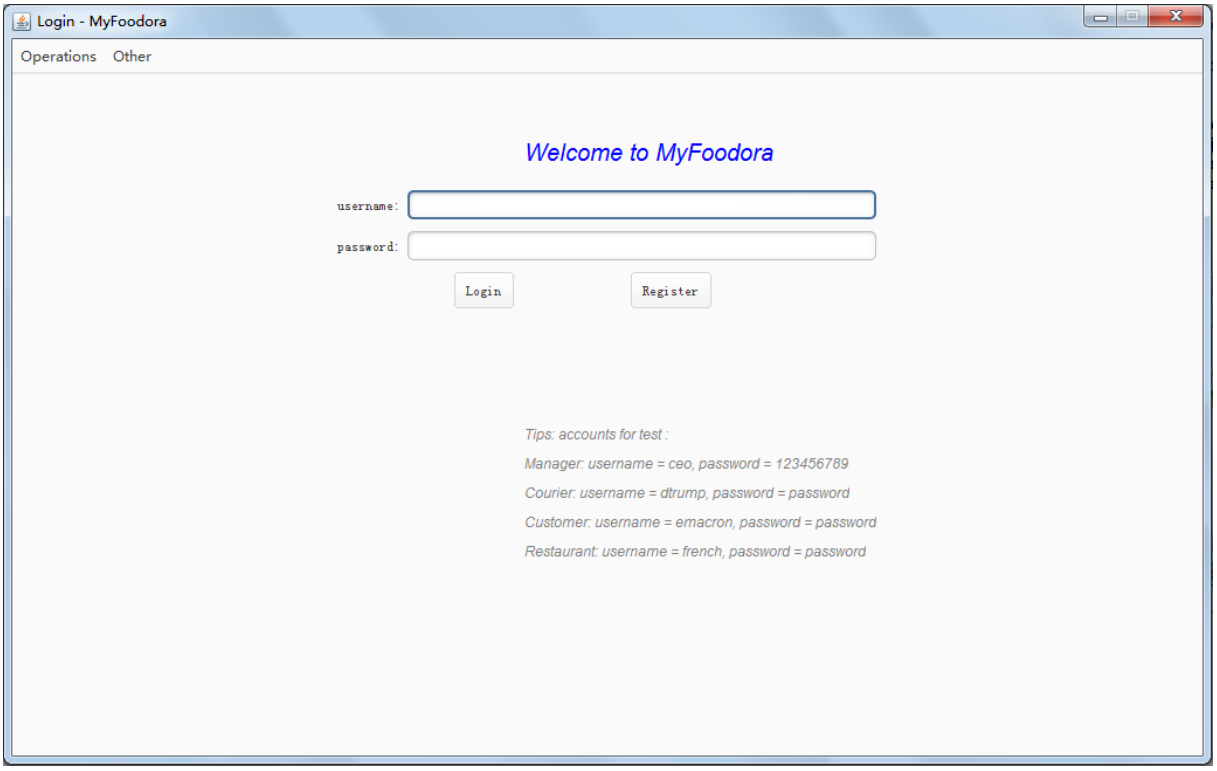
To test the user registration, please simply enter “register” after launching the CLUI and follow the steps given by the CLUI.

GUI: Graphical User Interface

The main method to be executed to access the graphical user interface is contained in **src/gui/runGUI.java**

The GUI can also be accessed by executing the **GUI.jar** file submitted within the project file.

The graphical user interface is written based on the Command Line User Interface and supports most commands available to the CLUI.



Conclusion

This challenging project allowed us to sharply improve ourselves in programming and was a very instructive experience.

It took us a lot of time and energy, but we are proud of ourselves as the performance of our final product exceeded our original expectations.

The Command Line User Interface (CLUI) is fully functional, it meets the requirements of the project and can handle input errors. Several test scenarios have been written to test the functionalities and ensure the correctness of the code.

The Graphical User Interface (GUI), despite being basic, is capable to fulfill most of the functionalities of the CLUI.

We may still improve our product by adding extra functionalities (sending messages between users, allowing multiple connections at the same time…) or adding some complex exception handling, for example preventing a restaurant from deleting a meal contained in an unpaid order.

In the end, we would like to thank Professor Paolo Ballarini, M. Arnault Lapitre and Ms. Celine Hudelot for their help during the completion of this project.