CentraleSupélec

Module IS1220

Software Engineering

Project MyFoodora Part I

Group4_Project_IS1220_Part1_He_Ji

Xiaoan He

Raymond Ji

10/04/2017

Index

Index ____ *2*

Introduction ____ *3*

Design & UML ____ *4*

I. Design principles ____ 4

II. UML ____ 4

III. Code organization ____ 5

IV. Design patterns ____ 10

V. Key functionalities explained ____ 16

Teamwork ____ *18*

Team work chart ____ 18

Initialization ____ *19*

Tests ____ *20*

JUnit simple tests ____ 20

Use case scenario ____ 21

Conclusion ____ *23*

Introduction

As the world moves rapidly toward a complete digital age, modern food delivery systems such as Foodora or Deliverro, 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 are similar to that of those nowadays Food Delivery Systems.

The project is written in Java, which has several advantages:

- It is a pure Object-Oriented language
- It is easy to write and more readable and eye catching.
- It has a concise, cohesive set of features that makes it easy to learn and use.

The first part of this project (subject of this report) will focus on the design and development of basic functionalities of the MyFoodora system, alias the MyFoodora core.

The second part of this project will focus on the user interface, consisting in command lines and GUI.

For this project, we adopted the following convention while naming our variables:

- mealType, dishType stand for the properties standard, vegetarian and gluten-free
- mealCategory, dishCategory stand for the distinction between classes inheriting from a superclass: HalfMeal/FullMeal for meals and Starter/MainDish/Dessert for dishes.

Design & UML

I. Design principles

The programming of this project will follow 6 principles to support an easily extensible and maintainable system.

- ✓ *Open Close Principle*
  Open to the extension, close to the changes.

- ✓ *Liskov Substitution Principle (LSP)*
  Where a base class can appear, his subclasses can appear

- ✓ *Dependence Inversion Principle.*
  Interface/abstract programming. Dependence are on abstraction rather than on concrete.

- ✓ *Interface Segregation Principle*
  Using multiple isolated interfaces is better than using a single interface. Meaning a reduction of the degree of coupling between the classes.

- ✓ *Demeter Principle*
  Also called least knowledge principle, meaning an entity should interact as little as possible with other entities, making the system function modules relatively independent.

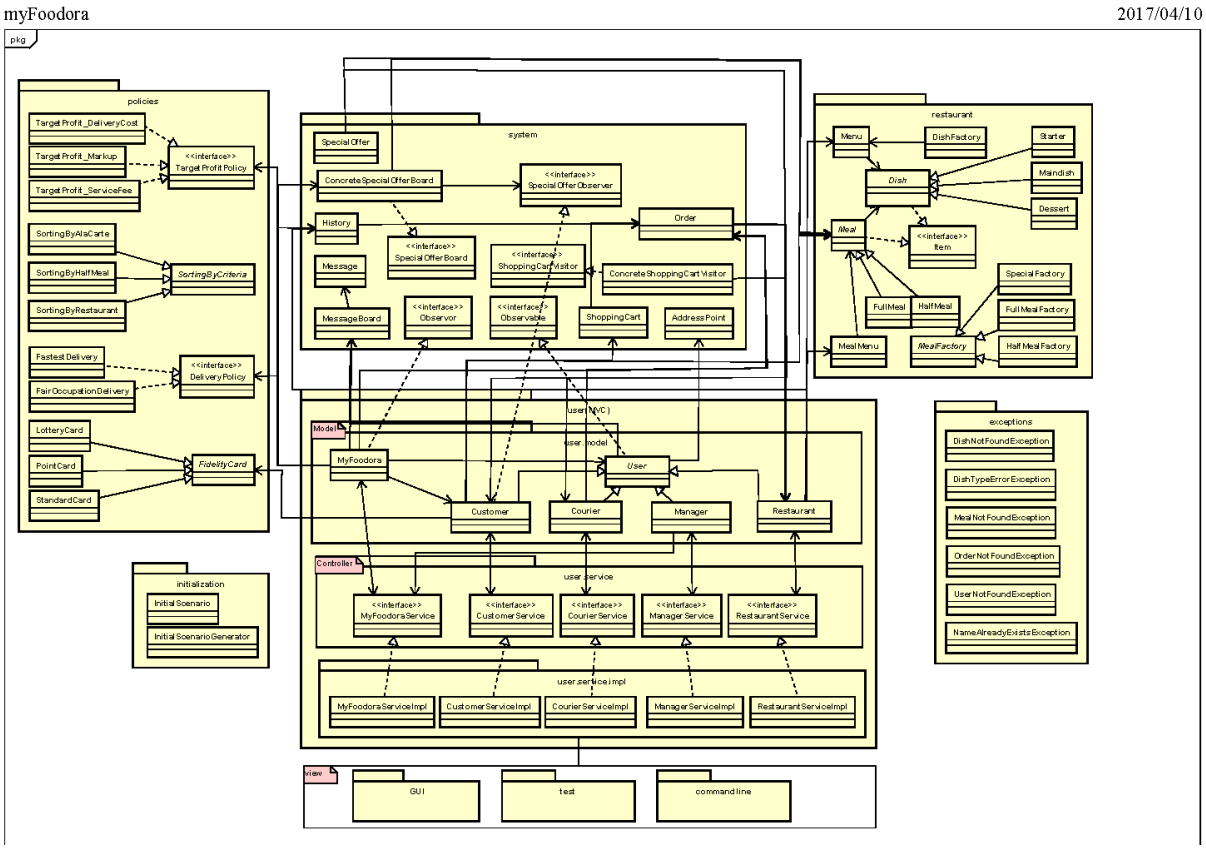
- ✓ *Composite Reuse Principle*
  Avoid abusing the inheritance, and make good use of the composition.

II. UML

For the UML, we used the Astah software, a popular UML tool. The installation file has been included into the project 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, however the visibility is not optimal.



III. Code organization

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, in which they store username, password and userID.

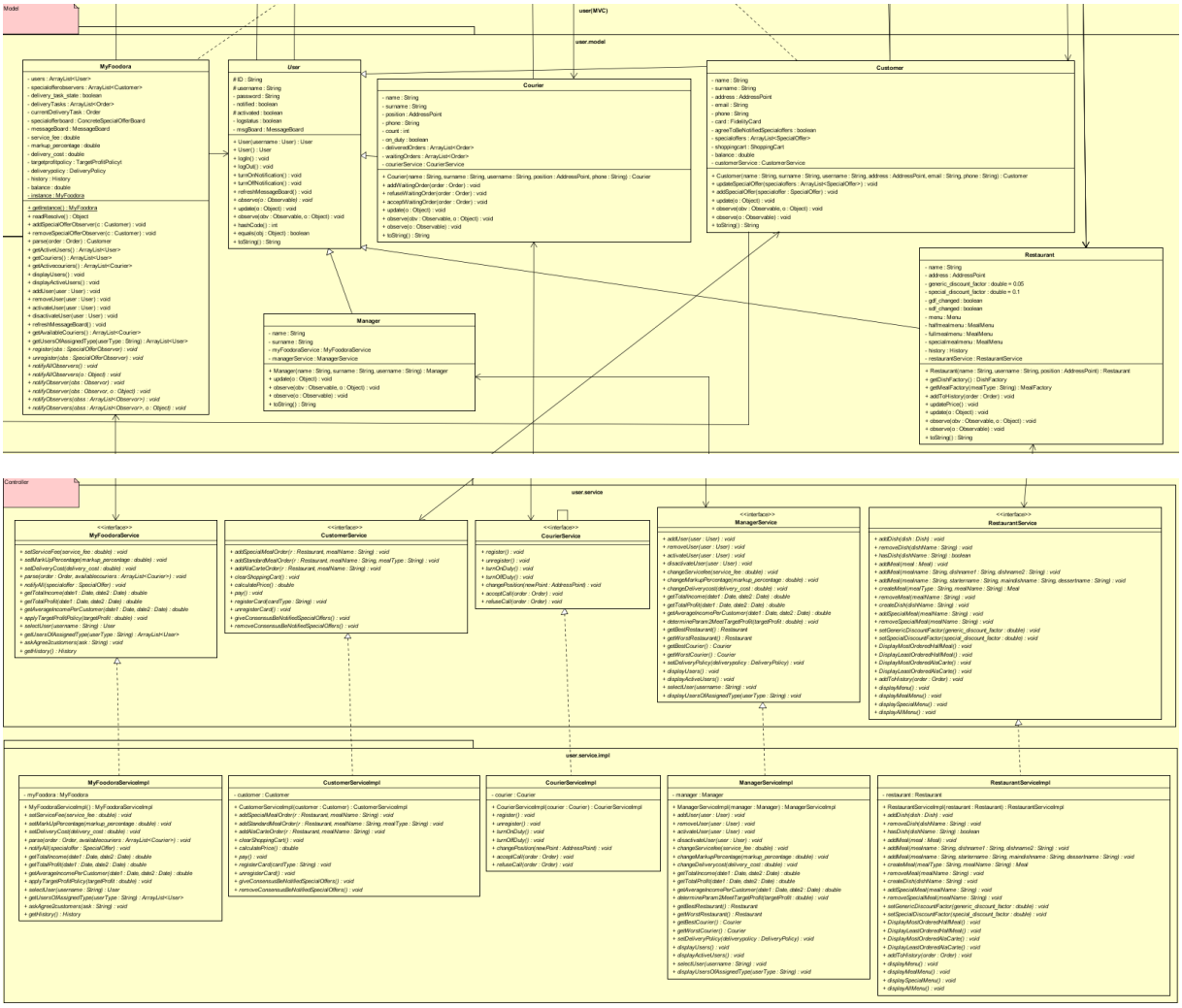
The MyFoodora class is the core class of the system. It stores a list of users and is capable of basic functionalities as well as applying policies on different matters. The key components of the system (Order, history, message board…) are placed in a separate system package (package system).

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 it to observe the special offers available.

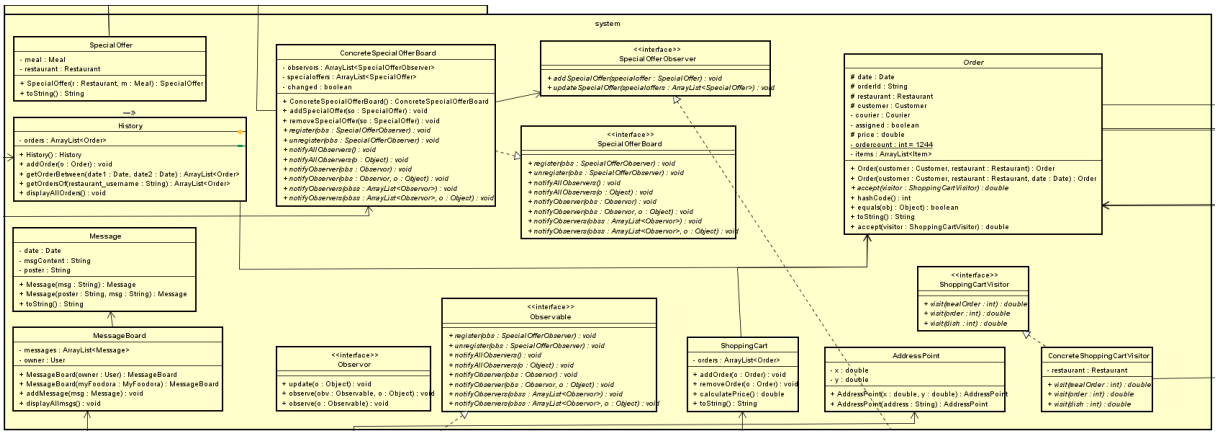
The Courier class has an History which stocks his past deliveries. He can register and unregister from MyFoodora.

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).



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

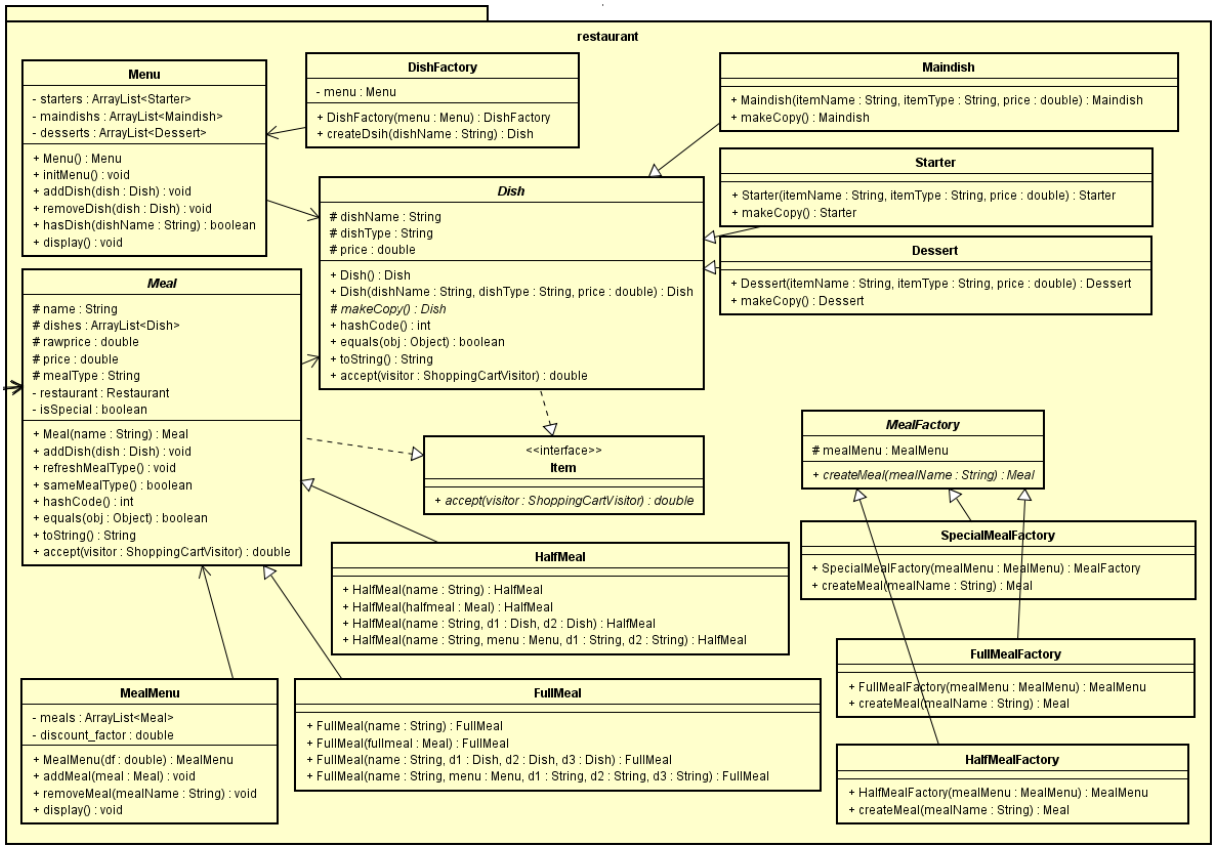
Classes that allows the set-up of a restaurant.

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.

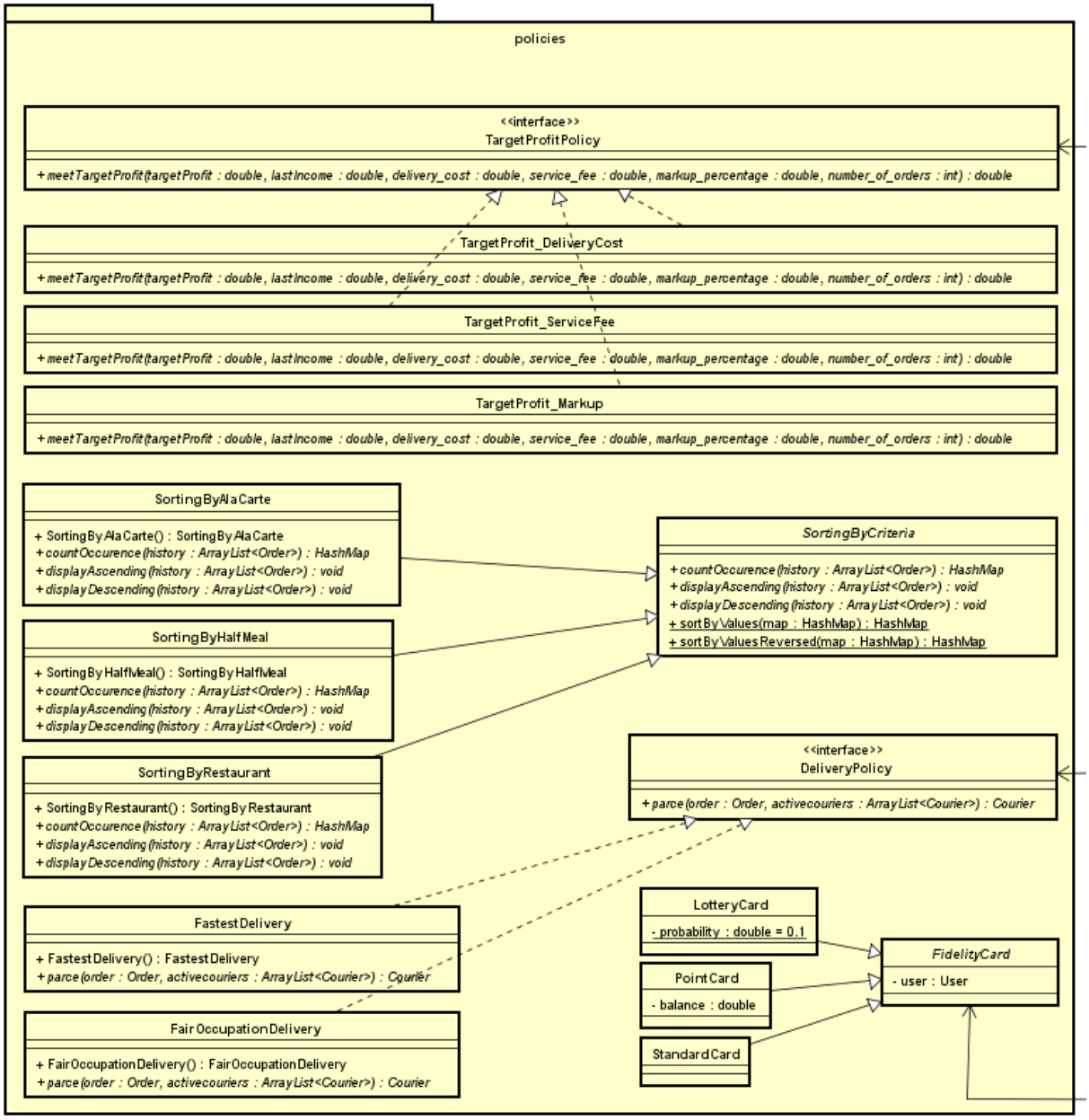
Therefore, a restaurant has 3 MealMenus: a meal-menu for half-meals, a meal-menu for full-meals and a meal-menu for meal-of-the-weeks.

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.

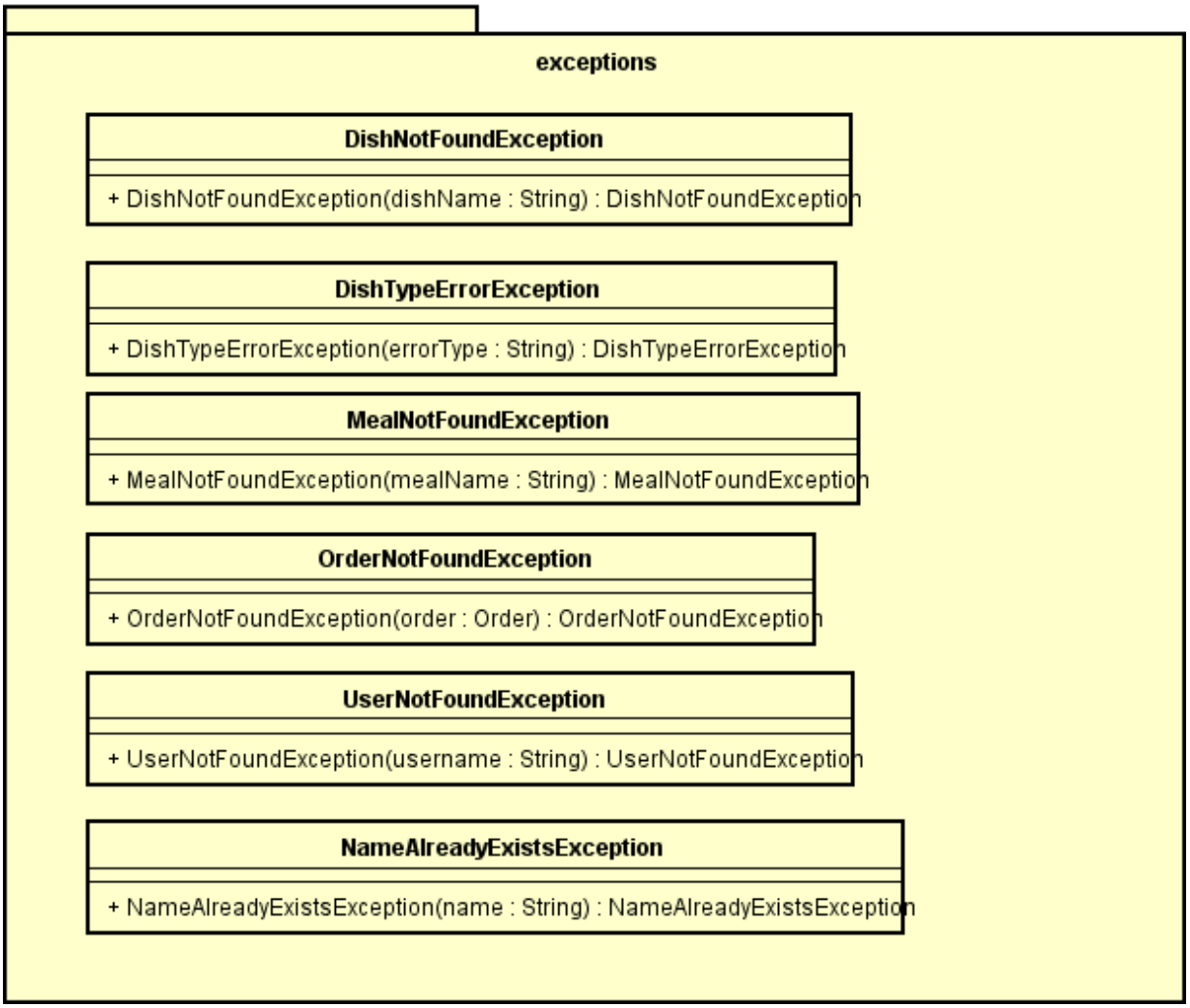


5. Package exceptions

Includes all custom Exceptions. There are 6 custom exceptions:

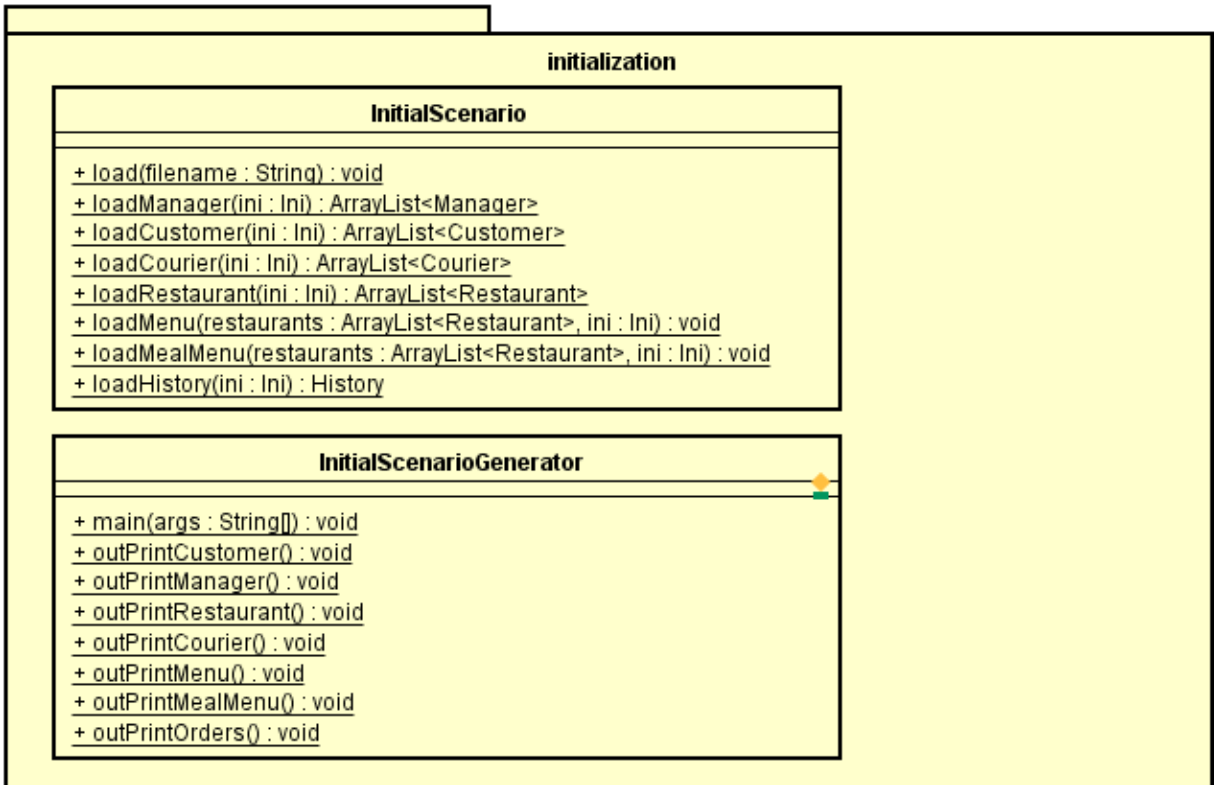
- *UserNotFoundException*: thrown whenever a user is not recognized, for example when a manager tries to activate a user who didn't register to MyFoodora.
- *DishNotFoundException*: thrown whenever a dish name is not recognized, for example when a customer tries to order a dish that is not in the menu of athe restaurant.
- *MealNotFoundException*: thrown whenever a meal name is not recognized, for example when a customer tries to order a meal that is not in the meal-menu of a restaurant, or when a restaurant tries to upgrade a meal that is not in the mealmenu to a meal-of-a-week.
- *OrderNotFoundException*: thrown whenever an order is not recognized.

- *DishTypeErrorException*: thrown when a restaurant tries a add a meal that does not respect the 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, and mealnames and dishnames must be unique in the menu of a restaurant. This exception is thrown if the contrary occurs.



6. Package initialization

An independent package that provides a Class which can load a custom initial scenario from a .ini file, as well as a generator of custom scenarios (all you need is to enter the parameters like the number of restaurants etc., however you will still need to fill some blanks spaces manually). It will be discussed more in depth later.



7. Package test

Contains all the Junit4 tests. The tests will be discussed in the dedicated chapter.

IV. Design patterns

Following design principles above and in consideration of functional demands of MyFoodora system, design patterns below are used.

1. MVC pattern: Users & MyFoodora

The major functions of MyFoodora system are based on functions of Customer, Manager, Restaurant and Courier which are subclasses of the User class. So the MVC pattern is a nice choice in the case. The entities (layer Model) and functions provided by them (layer Controller) are divided, and these functions will be used by the yet to be implemented UI (layer View).

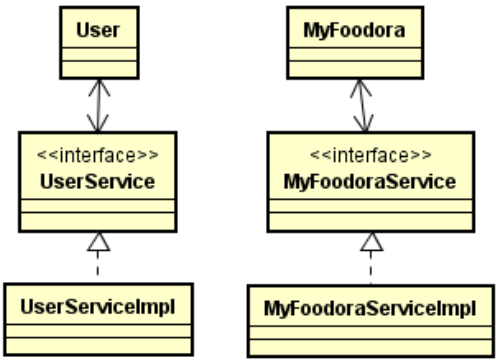
- Model
  Responsible for representation of Java entities and data management
- Controller
  Responsible for forwarding requests and processing requests
- View
  Responsible for user interface.

Although using MVC pattern increase the code quantity, it has several advantages. It makes the java entities dependent in the layer Model, who are generally changeless. All

entities provide their functions in the layer Controller, which are called in the layer View. Therefore, the layer Model and View are out of binding so the degree of coupling decreases. It's 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: ***Service.class are interfaces of functions provided by users and MyFoodora system and ***ServiceImpl.class are the corresponding concrete implementations.

- *Manager.class ---- ManagerService.class ---- ManagerServiceImpl.class*
- *Courier.class ---- CourierService.class ---- CourierServiceImpl.class*
- *Customer.class ---- CustomerService.class ---- CustomerServiceImpl.class*
- *Restaurant.class ---- RestaurantService.class ---- RestaurantServiceImpl.class*
- *MyFoodora.class ---- MyFoodoraService.class ---- MyFoodoraServiceImpl.class*

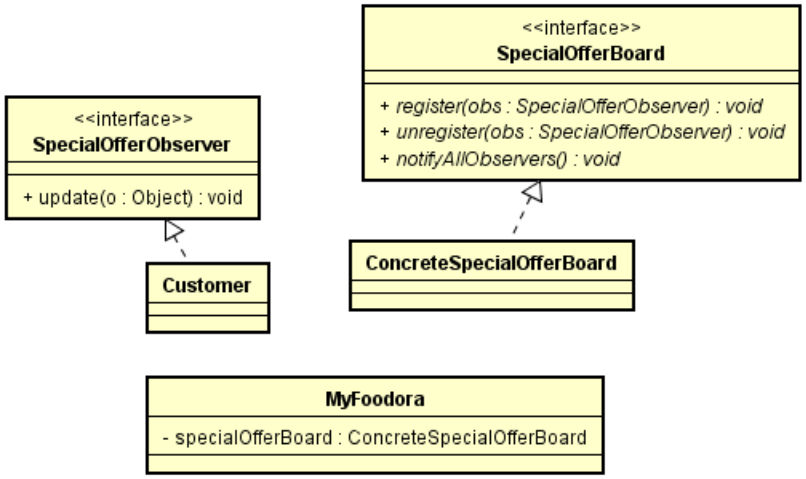


2. Observer pattern: SpecialOfferBoard/SpecialOfferObserver

Whenever a new special offer is added to the special offer board of MyFoodora system, all customers that gave their consensus to receive news shall be notified.

That's why we chose the Observer pattern for the implementation of these functionalities. The *Customer.class* implements a *SpecialOfferObserver (Observer) interface* which provides methods like update(), observe() to react. It interacts with an *SpecialOfferBoard (Observable)* interface who provides methods like register(), unregister(), notify() to inform its observers.

Each MyFoodora class has a unique ConcreteSpecialOfferBoard.



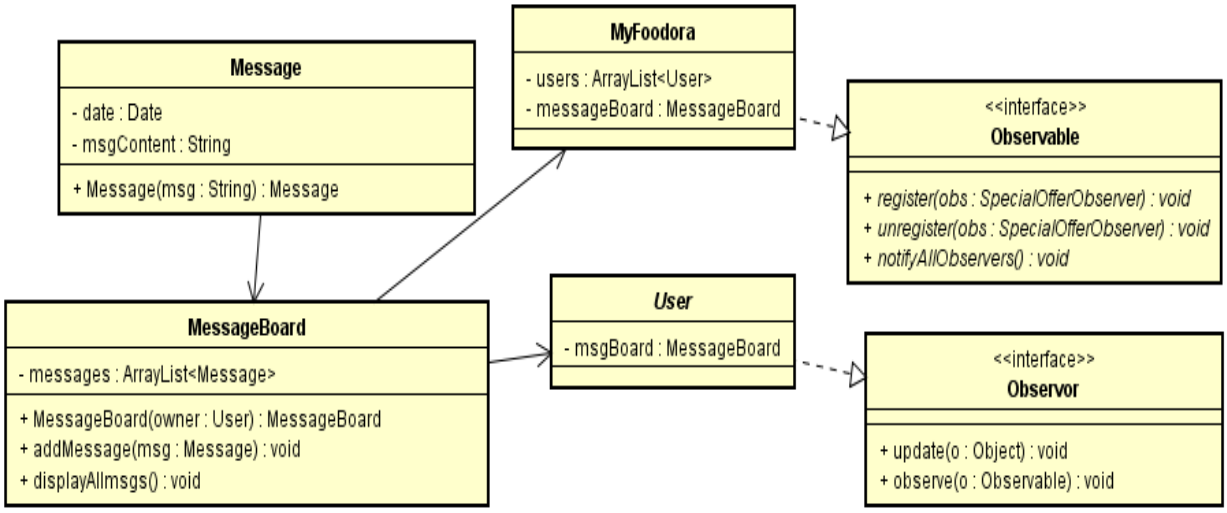
3. Observer pattern: Message and MessageBoard

We need a platform which allows exchange of information/message between the MyFoodora system and all users. As soon as a new message is posted, the corresponding user should be notified. That's why we used an Observer Pattern for the message board functionalities.

Therefore, the MyFoodora.java class has an MessageBoard(Observable). The abstract *User.class* (superclass of *Manager, Courier, Customer* and *Restaurant*), owners of an individual MessageBoard, implement an Observer interface.

Both MyFoodora and User have a MessageBoard as a field, where new Messages can be added.

Meanwhile, the addition of other types of potential communication is supported if necessary in the future.

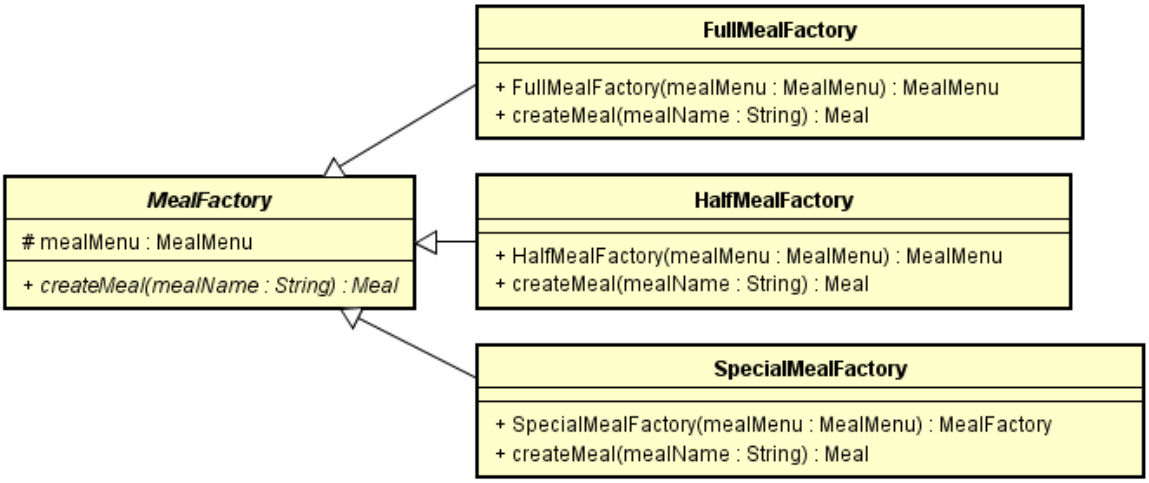


4. Abstract Factory pattern: DishFactory/MealFactory

The factory pattern fits here for the code system to produce dishes and meals of a restaurant. However, it's 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, the factories of our abstract factory used to create a dish/meal take a String dishname/mealname as argument and checks the name in the menu/meal-menu to obtain the concrete dish/meal.

To follow the Open-Close principle, an abstract factory pattern is used. For meals, an abstract factory class *MealFactory.class* including *createMeal()* is inherited by two factories *FullMealFactory.*class, *HalfMealFactory.class* and *SpecialMealFactory.class*, respectively for producing *FullMeal.class, HalfMeal.class* and meals contained in the Special-offer meal menu .



5. Singleton pattern: MyFoodora

The MyFoodora, 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 construct one reference, which is obtained by its *getInstance*() static method.

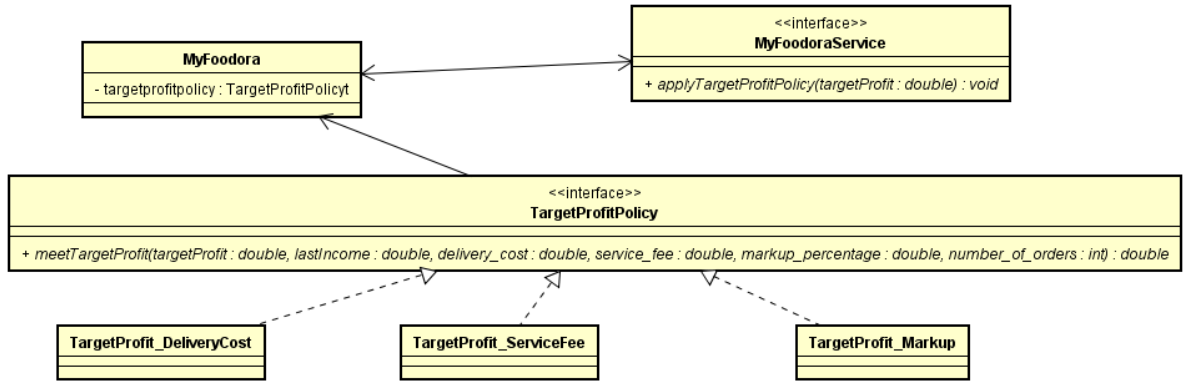
Moreover, to avoid potential thread interruption in the gap between the operation of object creation and of object assignment, the operation of object creation is picked up and locked.

To serialize object, it also supports a readResolve() method.

```java
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;
    }
```

6. Strategy pattern: Target profit policies

MyFoodora should be able to dynamically compute 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.

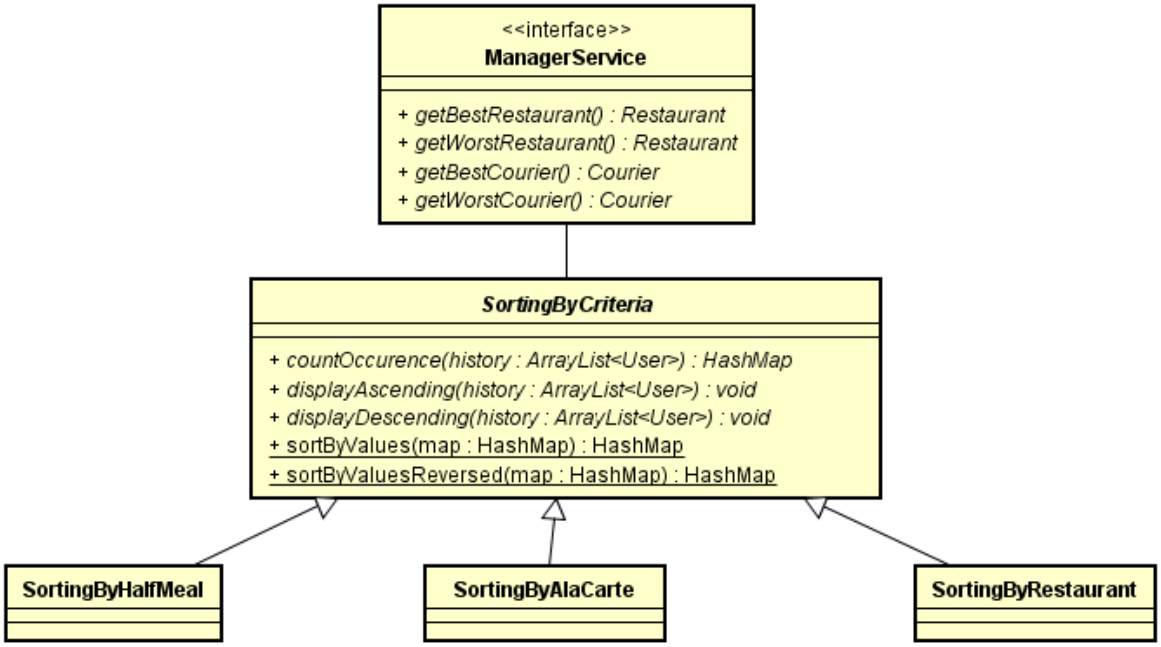


7. Strategy pattern: Shipped order sorting policies

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 w.r.t the number of shipped half-meals/a-la-carte orders.

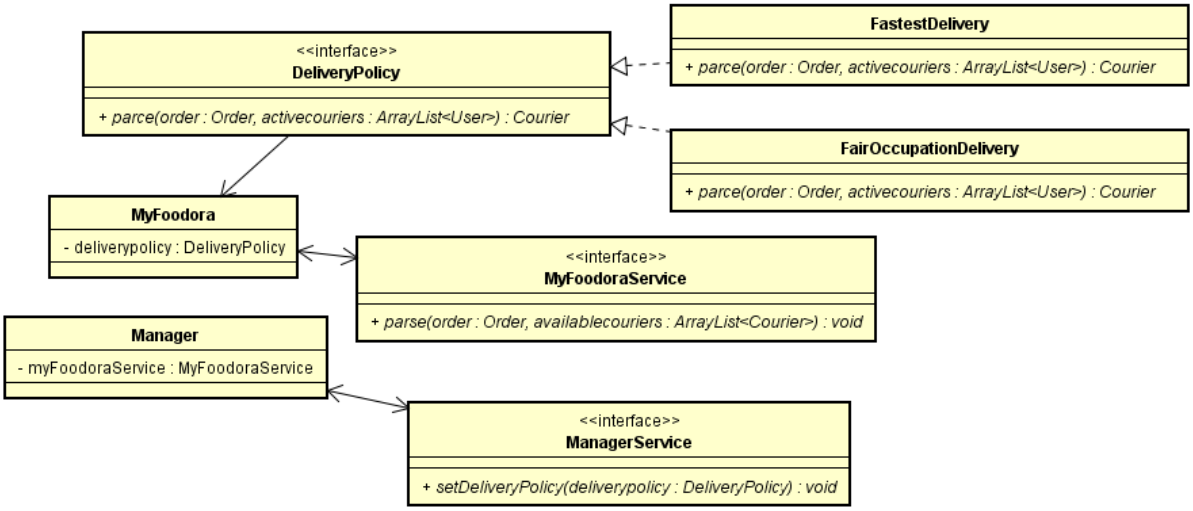
That's why, a strategy pattern is used for sorting policies. To store the number of shipped orders of a certain meal/dish, we used the HashMap interface of Java.

We also wrote a History class, implemented by MyFoodora and Restaurant, which stores the shipped orders and decreases the degree of coupling.



8. Strategy pattern: Delivery policies

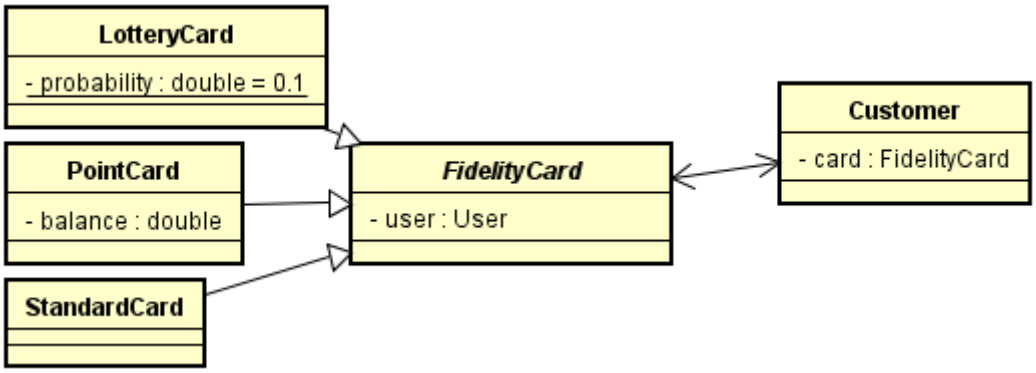
MyFoodora should be able to dynamically use the delivery policy to allocate orders to couriers, allowing for adding new implementations of the policies and selecting a new at the run time. That's why, a Strategy pattern is used for delivery policies.



9. Strategy pattern: Fidelity cards

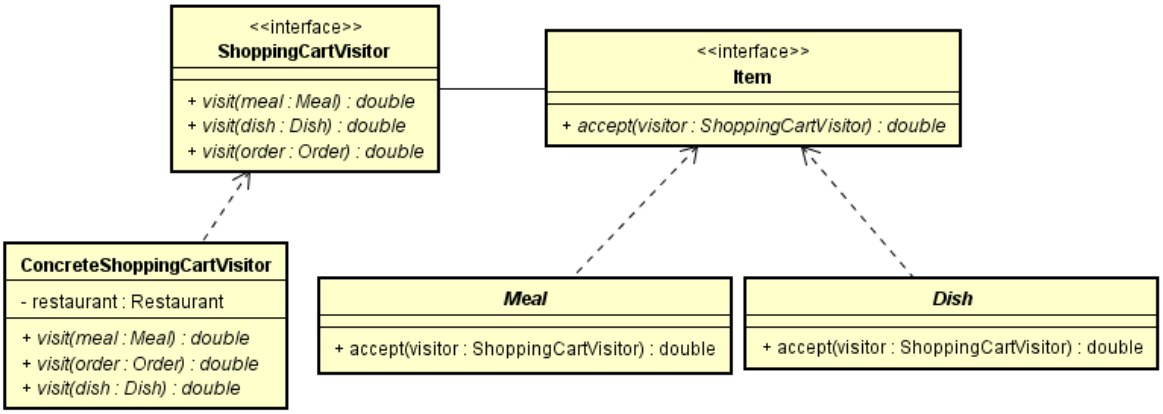
MyFoodora should be able to dynamically compute the total price to pay for a customer and consider whether he/she has a Point Fidelity card or a Lottery Card. The Fidelity

card system should should for adding new implementations of the cards and selecting a new at the run time. That's why, a Strategy pattern is used for fidelity cards.



10. 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).



V. Restaurant functionalities explained

Here we will extend on some key functionalities that we think are worth explaining more in detail.

1. Pricing a meal/order

The price of a meal/order is automatically computed. The price of the meal is calculated by the ShoppingCartVisitor by extracting the correct discount factor from the meal's restaurant variable (generic discount factor for the half-meal and the special discount factor for the special-meal). The price of the order is given by the ShoppingCartVisitor when it is added to the ShoppingCart.

2. Ordering a meal

The ordering of a meal/dish is implemented by CustomerService.java. When a customer places an order, the corresponding restaurant invokes its abstract factory and produces the meal/dish. Customer, restaurant and meal/dish are then placed inside an order that is added to the shopping cart of the customer.

3. Payment and parsing of orders

When the customer pays, his/her fidelity card is used to pay the price of his/her shopping cart, and at the same time the fidelity card policies are applied (points added to the balance of a point card / possibility to obtain a free meal for a lottery card).

His/her shopping cart is then cleared and the orders in the shopping cart are parsed to a courier and a message displays, indicating the courier who has been assigned according to the shipping policy. If the courier accepts (by logging in and using the command accept) the order is complete and registered into the History of MyFoodora and the concerned Restaurant.

Teamwork

The project is the result of teamwork between us.

We used GitHub to put together our works: github.com/raym96/raymxiaoan

Team work chart

Coding & Testing

TASK	CODING	TESTING
Restaurant MVC	Xiaoan & Raymond	Xiaoan
Customer MVC	Raymond	Xiaoan
Manager MVC	Xiaoan	Raymond
Courier MVC	Xiaoan	Raymond
MyFoodora MVC	Xiaoan	Raymond
Package system	Xiaoan & Raymond	Xiaoan
Package restaurant	Raymond	Xiaoan
Policies	Xiaoan & Raymond	Xiaoan
Initialization	Raymond	Xiaoan
Use Case	Xiaoan & Raymond	Raymond
Custom Scenario	Raymond	Xiaoan
Javadoc	Xiaoan	----------
UML	Xiaoan	----------

Report writing

TASK	Author
Introduction	Raymond
Structure	Xiaoan
Design Patterns	Xiaoan & Raymond
Initialization	Raymond
Tests	Xiaoan
Scenarios	Raymond
Conclusion	Raymond

Initialization

We created a .ini file reader class which can automatically import data from a ini file and enter it into the MyFoodora system. This way we can optimally set up initial scenarios and conduct tests based on given initial scenarios. It will be heavily used for the JUnit4 Test part and the Scenario Test part.

For this feature, we used an external package ini4j which is a simple Java API for handling configuration files in Windows .ini format. It is included in the .zip file of our project but it can also be download here:

http://ini4j.sourceforge.net/download.html

The classes needed for this feature are stored in the initialization package:

- InitialScenario, which contains methods to load from given .ini files respecting a given format.
- InitialScenarioGenerator, which given the number of restaurants, customers, managers, couriers, dishes per restaurant, meals per restaurant or the number of orders in history, can generate the corresponding text that can be filled and copy-pasted into a .ini file.

We will be primarily working on 2 ini files:

- Init.ini which hat 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 100 past orders in the history. It is the basic set up for Use Case tests.
- Scenario_test_service.ini which is a simplified version of the initial scenario, containing 2 restaurants, 1 manager, 3 couriers, 2 customers, 6 dishes per restaurant, 5 meals (2 half-meal, 2 full-meal, 1 special-meal) per restaurant and 30 past orders in the history. It is the basic set up for JUnit4 Tests of Controller layers of the key user classes.

Tests

JUnit simple tests

Software testing is mandatory because we all 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.

In java, the basic tests will be conducted using the JUnit4 package. Explicitly, different Junit annotations and statements are used for testing:

@BeforeClass –test method with it is executed when the Class is loaded, so generally it's an initialization, such as contractors of classed.

@AfterClass – test method with it executed at last of the working flow, so generally it's summaries, such as *toString*(), display methods and closure of files and streams.

@Test – normal test methods

*@Test(expected = ***Exception.class)* – when a kind of exceptions is expected and tested

assertNotNull() – statements which are used to test initializations of a class and fields inside such as an ArrayList.

assertTrue(),assertFalse(),assertEquals() – statements which are used to test Intermediate results.

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

Therefore, the **key** JUnit tests are the ten tests included in the **test.user** package.

We listed below some of the errors we had been able to detect thanks to testing:

- Logical errors

  We made several crucial logical errors during the coding phase of the project, including for example the Fidelity Card system.

- Structural problem: Inconsistencies with special meal

  At first, we created a SpecialMeal.java class which would be the meals declared "special-offers". However, we realized thanks to tests that it would lead to

inconsistencies and problems, as a SpecialMeal can be as well as a HalfMeal or a FullMeal.

- Structural problem: History and orders sorting

  At the beginning orders were stored in the history under the form of an ArrayList<Order>. However, tests proved necessary that a History.java class needed to be implemented so the system would be less dependent on the Order.java class, and the HashMap interface to be used to store the frequency of an order.

- Structural problem: Courier allocating

  When the system allocates an order to a courier, a courier needs to either accept the order or refuse it. If he refuses, the system needs to automatically allocate another courier.

Use case scenario

After isolating each class and testing them using JUnit4 package, we need now to test the whole business process through given use cases of MyFoodora.

The use case scenario tests a located in the package test.usecase.

Use case scenarios are based on services of users and MyFoodora. All use case scenario should depend on only the Controller layer of users and MyFoodora, if some functions in the Model layer and some extra functions are needed, then they should be declared as new service in correspondent ***Service interface and be implemented in correspondent ***ServiceImpl.class.

In other words, any client should be able to fully benefit from functionalities provided by MyFoodora using **only** methods provided in the Controller layer of Customer, Manager, Courier, Restaurant and MyFoodora.

1. StartupScenario

In all other use case scenario, the system should be initialized at first, so a preprocessing scenario is called using the Junit annotation *@BeforeClass.*

In StartupScenario, 2 managers (the CEO, and his deputy), 5 restaurants and per restaurant 4 full-meals, 2 couriers, 7 customers are initialized and customers are asked whether they agree to be notified of special offers, using the Observer pattern SpecialOfferBoard/SpecialOfferObserver.

2. LoginUserTest

The loginUserTest.java simulate a user who wants to log into the system. Users input their username and password following prompts. If username and password are correct, then they log in successfully.

Whenever a user logs in, the MyFoodora system is informed using the Observer Pattern Message/MessageBoard.

3. RegisterAUserTest

Users can register on MyFoodora as a customer, a restaurant or a courier by inputting personal information related to their category.
Then, the user will be registered on MyFoodora by a ManagerService automatically generated by the system.

4. InsertMealTest

Simulation of the addition of a new dish/meal to a restaurant's menu. The client is a restaurant. If it is a new dish, the client inputs the name, price and category the dish and the latter is added to the menu. If it is a new meal, the client inputs the name, the category(halfmeal/fullmeal) of the meal and its components. The price of the new meal is automatically computed by the meal menu, taking into account the discount factor of the restaurant. The operator is the RestaurantService.

5. OrderingAMealTest

Simulation of the ordering of a meal by a customer in a restaurant. The client is a customer who chooses a restaurant then inputs the category and name of a dish/meal, and the operator is the CustomerService.

6. AddSpecialOfferTest

Simulation of the addition of special offers in the menu of a restaurant. The client is a restaurant who chooses several dishes/meals as its special meals, and the operator is the RestaurantService.

Whenever a special meal is added to the special meal-menu, all registered customers who agreed to receive notification will be this notified using the Observer pattern SpecialOfferBoard/SpecialOfferObserver. The operator is MyFoodoraService.

7. RemoveSpecialOfferTest

Simulation of the removal of special offers in a restaurant. The client is a restaurant who can choose a meal from its special meal menu, and the operator is the RestaurantService.

Conclusion

This first part of the project has been very interesting and allowed us to sharply improve ourselves in programming.

However, there are still several improvements that we could make on the first part of the project that we regret not to have enough time to do:

- Orders from the same restaurant should be packed together and delivered by the same courier.
- Pricing of the order/meal/dish by the ShoppingCartVisitor can be more consistent.
- Customer should not have to specify whether he wants to order a Half-meal or a Full-meal, ideally, he would just need to give the name of the meal.
- Improvements on user display.

We are optimistic and highly motivated for the second part with the user interface.