CentraleSupélec

Final Report

MyFoodora project

Arturo Garrido and Guy Tayou

Ecole CentraleSupelec, January 8, 2017

Contents

Chapter 1

Introduction

This is the final written report on the design and implementation of our project. We decided to use the LaTeX software to write the report, in order to give it a more professional appearance. We will go through a few points in the introduction, concerning the main design of the project and the report, and we will then get started with the project itself.

1.1 Structure of the report

As you can see in the content index, we will structure the main part of the report in packages. For each of the eight packages, we will first do a brief description about the general aspects of the package (the design decisions, the classes that the package contains, etc.). We will then talk about the different tasks that the package required. Usually, each class will be a task. For each task we will also do some general comments at first, and then talk about the main characteristics of the class. Throughout the description of the class, we will also talk about the approach that we used, and the problems and difficulties that we found.

After talking about the core of the system, we will describe the Command Line User Interface. We modeled the interface as a package, named clui, so we will talk about the two classes included in this package. We will also describe the modifications done with reference to the clui commands given by the project documentation. In the next chapter, we will talk about the three test scenarios that we created. We will then describe the UML diagram of our code, and show images of it. Finally, we will wrap up the rapport with a brief conclusion, in which we will talk about how we felt about the project and what we learned from it.

1.2 Design Patterns

We tried to use as many design patterns as possible, in order to comply with the Open-Close principle. However, we didn't want to overfit the code with useless patterns that wouldn't really contribute to the correct design of the application. We realized that following this Open-Close principle was very useful when doing further modifications of the code. To give an example, when we decided that a change in the special discount factor or the generic discount factor would also be considered as a special offer, as we had used an observer pattern to model these notifications, we only had to do a couple of changes in the code of the update() method of the observer, and a call to the notify function when the descount functions were modified.

1.2.1 Notifications: Observer Pattern

To make use of this package, we had two options: we could use the Observable class and Observer interface provided by the Java JDK, or we could create our own implementation of the pattern. We decided to do it by ourselves. Therefore, we created both the Observable and the Observer interfaces, each of them with their respective methods. The observers would be the customers (those who gave consensus to receive these notifications), and the observable is the core system (which manages the action of notifying the customers). We will further explain this on 2.6.

1.2.2 Orders: Factory Pattern

We used a factory pattern to create the orders, to separate the logic of the creation of an order. Therefore, we created a class OrderFactory, which has the method createOrder() that returns a created order. When a customer places an order, he will call this createOrder method, instead of creating the order by himself with the constructor.

1.2.3 Meals: Factory Pattern

In the same way, the meals are created through a class called MealFactory. This factory will create a meal with its corresponding type and name, and then the meal class will have a method to prepare the meal (that is, choose its ingredients). We will further explain this on 2.5

1.2.4 Delivery Policies: Strategy Pattern

There are different delivery policies, which can be seen as different strategies. Therefore we created an interface, DeliverPolicy, with a unique method allocateCourier(). This method will be applied in a different way by the classes that implement this interface (each one with its own strategy). We will further explain this on 2.3

1.2.5 Fidelity Cards: Strategy Pattern

In the same way, each type of fidelity card can be seen as a behavoir. We created an interface, FidelityCard, that has a unique method use(). This method will be defined in different ways by the classes that implement this interface (each one with its own strategy or behavoir). We will further explain this on 2.4.

1.3 JUnit tests structure

For each class we tested each method. We created and object attribute which instantiates the tested class. We also created objects attributes that we used with the methods. Each test class has a setUp() method handling the initializations of objects and a tearDown method to reset the attributes. For example, we used the tearDown() method to clear the list of identifiers and usernames when users were involved. These methods called automatically respectively before and after each test. Therefore, we dont have to repeat initialization and resetting inside the test methods. In order to use the setUp() and tearDown(), the test class has to extend TestCase.

Chapter 2

Packages

2.1 System

2.1.1 Description

Package representing the MyFoodora core. It includes one class, called myfoodora (the core) with its respective test. The design of this package started the same day we started the project, and was completed progressively throughout our entire work, as it contains methods that affect almost all of the other packages. Therefore, some functionalities weren't added until certain different classes were completed.

Content

- class MyFoodora

2.1.2 Task 1: MyFoodora class → Arturo Garrido

Design decisions

We decided to include a big part of the logic and calculations of the program into this class. Most of the users functionalities are done through the MyFoodora core. For example, a Manager has the capability to compute the average income per customer over a time period. However, in real life, all he would do is type the command into the system, and the system would return him the value (he wouldnt calculate it by himself with pen and pencil). Therefore, the Managers method for computing the average income per customer will only be a call to the MyFoodora method that does this calculation and returns the result. As a result of this decision, MyFoodora will have more capabilities and methods than the five ones defined in the project text.

To make the code simpler, the superclass User will have a MyFoodora attribute, and the constructors of all users (except for Courier) will have a MyFoodora as an argument. The instance of MyFoodora will be sent to them when constructed, so that we dont have to send it as an argument for every method that calls the MyFoodora system.

Regarding the design patterns, the MyFoodora class is the "Observable" object of the Observer pattern that we used to model the notifications of special offers sent to the customers that gave consensus to receive them. Therefore, it contains a list of the register observers (which will be Customers), and the corresponding methods for registering and removing observers as well as notifying them.

Main Characteristics

- Stores the information of all Users. It can add or remove any kind of user:
  - Getters and setters of restaurants, customers, managers and couriers.
  - **public void** removeRestaurant(Restaurant restaurant)
  - **public void** removeCustomer(Customer customer)
  - **public void** removeCourier(Courier courier)
  - **public void** removeManager(Manager manager)
  - **public void** addRestaurant(Restaurant restaurant)
  - **public void** addCustomer(Customer customer)
  - **public void** addCourier(Courier courier)
  - **public void** addManager(Manager manager)
- Stores the service fee, markup percentage and delivery cost values. It can change them to any value, and also change them in order to reach a target profit.
  - Getters and setters of service fee, markup percentage and delivery cost.
  - **public void** targetProfit_DeliveryCost(double targetprofit)
  - **public void** targetProfit_ServiceFee(double targetprofit)
  - **public void** targetProfit_Markup(double targetprofit)
- Stores the delivery policy of the system, and it can change it.
  - Getters and setters of delivery policy
- Stores the information of all completed orders. It also processes an order: allocates the courier of the order (depending on the delivery policy), gets the customers fidelity card and sets the price, sets the date, adds the order to the archive of orders, ect.
  - **public void** processOrder(Order order)
- Computes the total income, total profit and average income per customer in a given period of time. It also calculates the average price of the orders of the last month, in order to set the target profit policies. Finally, we also created a method that computes the total income of all orders processes, without needing to introduce any specific dates.
  - **public double** ComputeTotalIncome(Date initial_date, Date final_date)
  - **public double** ComputeTotalProfit(Date initial_date, Date final_date)
  - **public double** ComputeAverageIncome(Date initial_date, Date final_date)
  - **private double** priceAVG()
  - **public double** ComputeTotalProfit()
- Computes the most/least selling restaurants as well as the most/least active couriers. The main approach was to go through all the stored orders in the system, and collect the information of how many times each restaurant/courier was involved in these orders. However, to make the algorithm less complex, we decided to create an attribute (integer) in both classes that stored the number of times a restaurant received an order / the number of times a courier delivered an order. This attribute is updated each time an order is processed.

- **public** Restaurant mostSellingRestaurant()
- **public** Restaurant leastSellingRestaurant()
- **public** Courier mostActiveCourier()
- **public** Courier leastActiveCourier()

- Allocates a courier to an order placed by a customer. During the order processing, as explained before, this method will be called. The operations for choosing a Courier are actually done in the deliveryPolicy class, so this method will only call that operations, and then set the obtained Customer to the order.
  - **public** Courier allocateCourier(Order order)
- It also supports the shipped order sorting policies: sorts the items and half meals with reference to the number of times ordered (only -la-carte for items). We first tried to use a special type of Collection, a SortedSet, that compared the restaurants / couriers based on their attribute number_orders. However if two restaurants had processed the same number of orders (or two couriers the same number of deliveries), the Set treated them as the same object, an one of them wasnt saved. Therefore, we finally adopted the strategy of iterating over the restaurants / couriers stored in the system, and introduce them into a sorted ArrayList.
  - **public** ArrayList<HalfMeal> sortOrderedHalfMeals()
  - **public** ArrayList<Item> sortOrderedItems()
- Register and remove observers so that they get notifications of the special offers set by restaurants. It also notifies the registered observers.
  - Getters and setters of the observers.
  - **public void** registerObserver(Observer observer)
  - **public void** removeObserver(Observer observer)
  - **public void** notifyObservers(Restaurant restaurant, Offer offer)

2.1.3 JUnit Tests

This was probably the longest JUnit Test. As the MyFoodora class is the core of the whole system, we had to create instances of all users, a lot of items, some meals (full and half), etc. We did this in order not to have "NullPointerExceptions", which were the main problem throughout the tests. For example, at first I created the items with null attributes, as I didn't think it was necessary to create them with detail when testing the MyFoodora class. However, in the test of the "processOrder" method, it calculates the price of the order, so it gave us a NullPointerException because the attribute price was null. I mainly used *assertTrue*, *assertEquals* and *assertNonNull*.

2.2 Items

2.2.1 Description

This package has 4 classes representing the different kinds of items available in the restaurants. Each class representing a concrete kind of Item inherits the Item class. We could have declared a kind attribute in Item in order to differentiate the items but as we will see this structure allows us to create and manipulate meals objects easily.

Content

- class Item
- class Starter
- class MainDish
- class Dessert

2.2.2 Task 1: Item class → responsible: Guy Tayou

Design decisions

Each item has a name and a price. Therefore, we included two attributes to store their value. We introduced an attribute allowing us to know whether or not an item is gluten free. Besides an Item can have one of the following types: standard or vegetarian. Instead of using a string attribute to specify the type, we created an enumeration declaring the possible types of items and Meal. We created an attribute to store the value of the type. To make the code simpler, the class has a reference to the owner restaurant, which is null by default and a reference to the orders which contain it. Besides each attribute is protected for inheritance.

In order to instantiate an item object, we need to specify its name, its price, its type and the value of its glutenFree attribute.

Main Characteristics

- We created the getters and setters for all attributes.
- We wanted to be able to know if two items were equal. We re-implemented the methods hashCode, equals and toString. We consider that two items are equal if the attributes name, price, type and glutenFree are equal.
  - **public boolean** equals(Object obj)
  - **public int** hashCode()
  - **public** String toString()

2.2.3 Task 2: Starter MainDish and Dessert classes → responsible: Guy Tayou

Theses classes inherit the Item class. They do not add any attribute or method. They are used to represent a starter. Their constructor takes the same parameters as the constructor of the superclass.

2.2.4 JUnit Tests

We created an attribute item and an attribute restaurant for the tests of the item class. We used the method assertEquals in order to the test getters and setters the method assertNotNull to test the constructor. For Starter, MainDish and Dessert we only tested the constructors because theses classes dont add any methods.

2.3 DeliveryPolicies

2.3.1 Description

This package has one interface and two classes which allow us handle the delivery policies. We needed a way to allocate a courier to an order to be delivered. The allocation had to support the following policies:

- fastest delivery: the courier which has the shortest distance to cover to retrieve the order from the chosen restaurant and delivering it to the customer is chosen.
- fair-occupation delivery: the courier with the least number of completed delivery is chosen.

Design decisions

We chose to use a strategy pattern. The interface DeliveryPolicy provides a unique method allocateCourier(Order order, ArrayList<Courier> couriers) for the allocation. We use one class for each policy. Each class implements this method with the required behavior.

Content

- interface DeliveryPolicy
- class FairOccupationDelivery
- class FastestDelivery

2.3.2 Task 1: DeliveryPolicy interface → responsible: Guy Tayou

Main Characteristics

- The first parameter is the order we have to deliver and the second parameter is the list of courier wherein we had to select a courier.
  - Courier allocateCourier(Order order, ArrayList<Courier> couriers)

2.3.3 Task 2: FairOccupationDelivery class → responsible: Guy Tayou

Main Characteristics

- Looks over the list of couriers given as parameter. It selects and returns the on-duty courier who delivered the minimum of orders. A courier possesses an attribute which corresponds to number of orders he delivered, so the algorithm isnt very complex.
  - **public** Courier allocateCourier(Order order, ArrayList<Courier> couriers)

2.3.4 Task 3: FastestDelivery class → responsible: Guy Tayou

Main Characteristics

- Looks over the list of couriers given as parameter. It selects and returns the closest on-duty courier to the restaurant which received the order.
  - **public** Courier allocateCourier(Order order, ArrayList<Courier> couriers)

2.3.5 JUnit Tests

We tested the classes FairOccupationDelivery and FastestDelivery. We used an order, a list of courier as additional attributes.

2.4 FidelityCards

2.4.1 Description

This package possesses one abstract class and three concrete classes. Each class represents a fidelity card that a customer may have.

The pricing policy depends on the kind of fidelity card that a user owns. MyFoodora handles three types of cards: basic fidelity card, point fidelity card and lottery fidelity card. The final price is calculated according to some rules (see the classes descriptions below).

Design decisions

We chose to use a strategy pattern to solve the problem. We created the abstract class FidelityCard providing a unique method use for the calculus of a price. Each concrete class represents a kind of fidelity card and implements the method to compute et set the price of an order. Each class declares the attributes it needs to work correctly.

Content

- FidelityCard abstract class
- BasicFidelityCard class
- LotteryFidelityCard class
- PointFidelityCard class

2.4.2 Task 1: FidelityCard class → responsible: Guy Tayou

Main Characteristics

The parameter is the order to which we have to set the price.

- **public abstract void** use(Order order)

2.4.3 Task 2: BasicFidelityCard class → responsible: Arturo Garrido

Main Characteristics

This class represents the fidelity card given by default, at registration, to any customer. A basic fidelity card simply allows to access to special offers that are provided by the restaurant (see MyFoodora).

- Computes and sets the price of an order. Adds to the order its associated items and meals he algorithm of the use method goes through all the meals and the items of the orders to compute the total price.
  - **public void** use(Order order)

2.4.4 Task 3: LotteryFidelityCard class → responsible: Guy Tayou

Main Characteristics

It includes the attributes chance and max. With a LotteryFidelityCard a user has a probability of chance/max to order for free.

- **public void** use(Order order)
- Getters and setters for chance and max

2.4.5 Task 4: PointFidelityCard class → responsible: Guy Tayou

Main Characteristics

It includes the attributes points and reached. The first one is the amount of points a card possess and the second one allow us to know whether or not 100 points were reached with the previous order. The number of points of the card increases by one for each 10 euros spent in a restaurant. When this number reaches 100 points it will receive a 10

- **public void** use(Order order)
- Getters and setters for points and reached

2.4.6 JUnit Tests

We didnt test the class FidelityCard because it is abstract and do not implement any method. For each subclass we created an associated attribute fidelity card. We used the method assertEquals for the potential tests of getters and setters and the method assertNotNull for the test of the constructor. The main differences are the implementations of testUse.

2.5 Meals

2.5.1 Description

This package has classes allowing to deal with the meal a user may order to a restaurant.

Design decisions

We used a factory pattern for the creation of meals. Therefore we created an abstract Meal class, concrete classes which inherit the abstract class and the class MealFactory to handle the creation of meal objects. The Meal class possesses the abstract method price and prepare used by the subclasses. The first one compute and return the price of the meal and second sets the items attributes according to the meal type and nature.

Content

- Meal abstract Class
- FullMeal class
- HalfMeal class
- MealFactory class
- InvalidItemException class

2.5.2 Task 1: Meal abstract class → responsible: Guy Tayou

Each meal has a name and a price. Therefore, we included an attribute to store its value. We introduced an attribute allowing us to know its type. It corresponds to the type of the items it can possess. A meal can have a starter, a dessert and a main dish. Thats why we created three null by default attributes. In order to simplify the code, the class has a null by default reference to the owner restaurant and a reference to the orders which contain it. Each attribute is protected for inheritance.

Main Characteristics

- Getters and setters for all attributes.
- Methods to add and remove an order to the arrayList of orders associated with the meal.
  - **public void** removeOrder(Order order)
  - **public void** removeOrder(Order order)
- Returns the discount factor of the restaurant we need to use the compute the price of the meal
  - **protected double** discountFactor()
- We wanted to be able to know if two meals were equal. We re-implemented the methods hashCode, equals and toString. We consider that two meals are equal if the attributes type, starter mainDish and dessert are equal.
  - **public boolean** equals(Object obj)
  - **public int** hashCode()
  - **public String** toString()
- **public abstract double** price()
- **public abstract void** prepare(Starter starter, MainDish mainDish, Dessert dessert) **throws** InvalidItemException

2.5.3 Task 2: FullMeal class → responsible: Guy Tayou

The FullMeal class inherits the Meal class. It doesnt add any attribute or method. It is used to represent a HalfMeal. Its constructor takes the same parameters as the constructor of the superclass.

Main Characteristics

- Computes the price of a meal by taking into account its starter, its main dish and its dessert.
  - **public double** price()
- Set the items of the meal and throws an exception if one of the given items in null or does not match the type of the meal.
  - **protected void** prepare(Starter starter, MainDish mainDish, Dessert dessert) **throws** InvalidItemException

2.5.4 Task 3: HalfMeal class → responsible: Guy Tayou

The HalfMeal class inherits the Meal class. It doesnt add any attribute or method. It is used to represent a HalfMeal. Its constructor takes the same parameters as the constructor of the superclass.

Main Characteristics

- Computes the price of a meal by taking into account the right items.
  - **public double** price()
- Set the items of the meal and makes sure the meal meal each half at the end of the operation. It throws an exception if one of the given items in null or does not match the type of the meal.
  - **protected void** prepare(Starter starter, MainDish mainDish, Dessert dessert) **throws** InvalidItemException

2.5.5 Task 4: InvalidItemException class → responsible: Guy Tayou

This class is used by meal classes. See task 3 for more informations.

2.5.6 JUnit Tests

The Meal class only implements a constructor and getters and setters. We did not test it but tested halfMeal and FullMeal. We didnt test the getters and setters as they are few complex methods and not the most significant here. We add Starter, MainDish and Dessert attributes. We only test the constructors for both classes. We tested MealFactory by testing the mehod createdMeal. We checked that it returned a Meal of the right class with the right type.

2.6 Notifications

2.6.1 Description

As previously explained, we decided to model the notifications of special offers with an Observer pattern. We had two options: we could use the Observable class and Observer interface provided by the Java JDK, or we could create our own implementation of the pattern. We chose the second option. The Observable interface will then be implemented by MyFoodora, and the Observer interface will be implemented by Customer. We decided that there can be three types of special offers: a new meal of the week, a change in the generic discount factor and a change in the special discount factor. Therefore, we decided to create an enum called Offer, with the three options.

Content

- interface Observable
- interface Observer
- enum Offer

2.6.2 Task 1: whole package → responsible: Arturo Garrido

Main Characteristics

- The Observable interface allows to register and remove observers (that will be, the customers that gave consensus to receive special offers), as well as notifying the observers (that will be, when there is a new special offer).
  - **public void** registerObserver(Observer observer);
  - **public void** removeObserver(Observer observer);
  - **public void** notifyObservers(Restaurant restaurant, Offer offer);
- The Observer interface contains the update method, that will receive the type of special offer as well as the restaurant that added this special offer and then notify the observers.
  - **public void** update(Restaurant restaurant, Offer offer);

2.7 Orders

2.7.1 Description

Package representing the orders that will be done by a customer, provided by a restaurant and delivered by a courier. We decided that an order would have a list of items, a list of meals, a restaurant, a courier, a customer, a price and a date as attributes. However, when doing the second part of the project, the CLUI, we realized that we also had to add a name to the order.

Content

- class Order
- class OrderFactory

2.7.2 Task 1: whole package → responsible: Arturo Garrido

Design decisions

We decided to use a factory pattern, in order to separate the logic of the creation of the order. Therefore, when a customer orders, he will call the method createOrder from the order factory, insted of creating it by himself with the constructor.

Main Characteristics

- The class Order has getters and setters for all of its attributes
- It can add an item or a meal to an already created order
  - **public void** addItem(Item item)
  - **public void** addMeal(Meal meal)
- It processes the order once it has been completed. To do this, it calls the function processOrder of the MyFoodora system
  - **public void** process(MyFoodora myfoodora)

- Finally, the Order Factory has the method create order, that returns the created order (but still not processed, as further items and meals can be added).
  - **public** Order CreateOrder(String name, Customer customer, Restaurant restaurant, ArrayList<Item> items, ArrayList<Meal> meals)

2.7.3 JUnit Tests

Once again, in order to test the getters and setters of all attributes, as well as the processOrder method, we had to create instances of almost all classes of the system (in order not to get NullPointerExceptions). We mainly used *assertTrue* with a comparison inside to test if the components of the order had been set correctly. To test processOrder, we used *assertNonNull*, as the only thing for which this method is responsible is to call the processOrder method of the MyFoodora class, so we only have to check if the order was processed, and not if it was processed with the correct values (we do that in the JUnit test of MyFoodora).

2.8 Users

2.8.1 Description

This package contains a class for each different user that can use the system. We decided to create a class User, that is then extended by each specific user (courier, customer, restaurant and manager). The main difficulty that we faced here was how to treat the usernames and the IDs, which had to be unique. We first decided that they would be unique for each kind of user. In other words, there can't be two restaurants with the same username, but there can be a restaurant with the same username as a customer. On the other hand, the ID is generated automatically by the system.

Content

- class User
- class Courier
- class Customer
- class ExistingUserException
- class Manager
- class Restaurant

2.8.2 Task 1: User class → responsible: Arturo Garrido

Design decisions

It has all the attributes that are common to all users (name, username, password, ID). We decided to also include a MyFoodora object as an attribute of a user. Therefore, the system only has to be sent as an input once, when instantiating the user. We declared all attributes as protected, so that the subclasses would have easy access to them.

Main Characteristics

- It only has getters and setters of all the attributes.

2.8.3 Task 2: Courier class → responsible: Guy Tayou

Design decisions

This class represents a courier. It extends the User class. As a restaurant, a courier has the common characteristic of a user. A courier also have a surname, a phone number and a position. We created two attributes to store the name and the phone number. For compatibility with the Restaurant class we created a Point2D.Double attribute to store both coordinates of the position.

In addition, a courier has a state. We created an enumeration and an attribute to specify whether or not a courier is on-duty.

We first though of using an arraylist to store the orders that each courier had done. However, we realized that we only needed the number of orders that he had delivered (to be able to sort the couriers with reference to the number of orders), and not any information about the orders themselves. That is why we decided to use an attribute as a counter of the orders delivered by the courier: int deliveredOrders.

Main Characteristics

- Getters and setters for all atributes.
- Inherited methods. It doesn't add any other methods.

JUnit Tests

One of the simpler class to test. We only used a Courier attribute and tested the constructor, getters and setters. We only used the assertTrue method.

2.8.4 Task 3: Customer class → responsible: Arturo Garrido

Design decisions

This class represents a restaurant. It extends the User class. It has all the common attributes of a user, and adds an adress, a fidelity card, contact information (phone and email adress)... We decided to also create an ArrayList of Orders, in which the orders done by the customer will be stored.

Regarding the design patterns, the customer represents the observer of the Observer pattern used to model the notification system. Therefore, it implements the interface Observer, and overrides the update function. To model the notifications system, we introduced an attribute, Boolean consensus, to set the consensus of the customer to receive the special offer notifications. Each time a customer gives consensus to receive the offers, he will be added to the list of observers of the observable object (MyFoodora class); in the same way, each time a customer removes consensus to receive the notifications, he is removed from the list of observers. We also created an enum to set the channel through which the customer wants to receive these notifications, which can be either phone or email.

Main Characteristics

- It can place an order. The order will be created (using the Factory pattern) but it will not be processed yet.
  - **public void** order(String name, Restaurant restaurant, ArrayList<Item> items, ArrayList<Meal> meals)
- Receives a notification on the phone or email (depending on what he selected) of a special offer, through the update function.
  - **public void** update(Restaurant restaurant, Offer offer)

JUnit Tests

As the customer interacts with the orders (through the order method), we had to also instantiate some items, meals and restaurants. In both testHistoryOrders() and testOrder(), we first use an *assertNonNull* to test if the order was created (if it exists), and then an *assertTrue* to see if a certain attribute of the order, for example the customer, is correct. In the testUpdate(), there isn't actually any assert call: the test will be correct if the prints the notification in the console.

2.8.5 Task 4: Manager class → responsible: Arturo Garrido

Design decisions

This class represents a restaurant. It extends the User class. This class has a lot of interaction with the MyFoodora core. As we explained before, we decided to integrate all the logic of the operations in the MyFoodora class instead of here, as it is a more realistic approach (when a manager wants to, lets say, calculate the total profit, he does it through the MyFoodora core, and not by himself). Therefore, a lot of the managers' capabilities are representing through methods that simply are a call to a method from the MyFoodora system.

Main Characteristics

- Add/remove any kind of user to/from the system. The method receives a string with the type of User, and the User object.
  - **public void** remove_user(String type, User user) **throws** ExistingUserException
  - **public void** add_user(String type, User user)
- Changing the service-fee percentage, the markup percentage or the delivery-cost. This methods will call MyFoodora, that will perform the operations.
  - **public void** setServiceFee(**double** fee)
  - **public void** setMarkupPercentage(**double** percentage)
  - **public void** setDeliveryCost(**double** delivery)
- Computing the total income and profit over a time period, the total profit of the system and the average income per customer over a time period. Once again, the methods use the MyFoodora system.
  - **public double** ComputeTotalIncome(Date initial_date, Date final_date)

  - **public double** ComputeTotalProfit(Date initial_date, Date final_date)
  - **public double** ComputeTotalProfit()
  - **public double** ComputeAverageIncome(Date initial_date, Date final_date)

- Determining either the service-fee, markup percentage or the delivery-cost so to meet a target-profit. Once again, the methods use the MyFoodora system.
  - **public void** targetProfit_DeliveryCost(**double** targetprofit)
  - **public void** targetProfit_ServiceFee(**double** targetprofit)
  - **public void** targetProfit_Markup(**double** targetprofit)

- Determining the most/least selling restaurant, as well as the most/least active courier. Once again, the methods use the MyFoodora system.
  - **public** Restaurant mostSellingRestaurant()
  - **public** Restaurant leastSellingRestaurant()
  - **public** Courier mostActiveCourier()
  - **public** Courier leastActiveCourier()

- Setting the delivery policy of the system, to determine in which way couriers are assigned.
  - **public void** setDeliveryPolicy(DeliveryPolicy deliveryPolicy)

- Activating and disactivating any kind of user from the system. In order to do this, we created an attribute in user, boolean enabled, with its respective methods isEnabled() and setEnabled().
  - **public void** activateUser(User user)
  - **public void** disactivateUser(User user)

JUnit Tests

One of the largest tests. As most of the manager's methods are a call to a MyFoodora system, the only thing that had to be tested here was if the order had been processes, and not if it had been processed in the correct way (that was done in the MyFoodoraTest). Therefore, we mostly used *assertNotNull*.

2.8.6 Task 5: Restaurant class → responsible: Guy Tayou

Design decisions

This class represents a restaurant. It extends the User class. A restaurant has all common characteristics of a user, a name and an address, so we added an attribute to store the name and a Point2D.Double attribute to store both coordinates of the address.

We used an arraylist to store all associated items. The system will go through this list to order a-la-carte. We created an arraylist of the associate meals for the selection of pre-compiled meals and another for the selection of meals of the week.

Besides, a restaurant has a generic discount factor and a special discount factor used to compute respectively the price of common meals and meals of the week special offers. Therefore, we created two attributes to store the value of the discount factor with their respective default

value. We created an attribute which stores the number of orders created for the restaurant. Its value is incremented by the system each time an associated order is processed.

Finally, regarding the orders prepared by the restaurant, we had the same issue as with the Courier class. We first though of using an arraylist to store the orders that each restaurant had processed. However, we realized that we only needed the number of orders prepared (to be able to sort the restaurants with reference to the number of orders sold), and not any information about the orders themselves. That is why we decided to use an attribute as a counter of the orders prepared by a restaurant: int number_orders.

Main Characteristics

- Stores the items and meals. It can add and remove any king of orderable object.
  - **public void** addItem(Item item)
  - **public void** removeItem(Item item)
  - **public void** addMeal(Meal meal)
  - **public void** removeMeal(Meal meal)
  - **public void** addMealOfTheWeek(Meal meal)
  - **public void** removeMealOfTheWeek(Meal meal)
- Getters and setters for all attributes.

JUnit Tests

A quite long JUnit test. We created a Restaurant attribute. We also created attributes for MyFoodora, Item and Meal because Restaurant needs them to works correctly. We mostly used the method assertTrue. We tested all methods of Restaurant.

Chapter 3

Command Line User Interface: CLUI

3.1 Introduction

We will now explain how we modeled the CLUI of the system. First of all, we did a couple of changes with reference to the commands given in the project documentation:

- login <userType> <username> <password>: we added the argument userType. This is because we considered that the usernames are unique for each kind of user, so there can be a courier and a customer with the same username.
- createMeal <mealName> <foodType> <mealType>: we though it was necessary to add the arguments foodType (which gives us the type of food in the meal, for example vegetarian) and mealType (a half meal or a full meal).
- setProfitPolicy <ProfitPolicyName> <target>: we added the target profit that has to be achieved by applying a profit policy.
- setCustomerInfo <phone> <mail>: new command. For the currently logged on customer to add his phone and email address.
- setConsensus <consensus>: new command. For the currently logged on customer to give or remove consensus to receive special offers.

Concerning the structure, we created a package named clui. At first, we wrote all the code in a single class, but we eventually decided to create another class with some of the functions that are not directly related to a command.

3.2 Class Application

We though it was logical to create a method for each command. We included all these methods in this class, as well as other methods that we will now explain.

3.2.1 Main characteristics

Attributes

- The system
  - MyFoodora myFoodora
- A list containing the meals created during the execution of the application.
  - ArrayList¡Meal¿ meals
- An object which provide a set of methods to look for the users created by given ther username or name
  - Finder finder
- The logged user, null by default.
  - User user

Methods

- The command methods
  - For each command, we defined a method with the same name as the command. This method contains the required instructions and takes the same number of arguments as the command. Besides the orders of the arguments of the methods match those of the commands. Inside each method we check that the logged user has the right to execute the associate command. For example, only a logged manager can use the command registerRestaurant.
- The execute methods
  - To call these methods we created two functions. They are both named execute but take different arguments. They make the link between a command line and the associated method.
- The converter methods
  - We created two methods for conversion. stringToDate converts a string date with the format "DD/MM/YYYY" into a Date object. stringToPoint converts a bi-dimensional string point with the format "x,y" (ex "12,6") into a Point2D.Double object
- The main method
  - The static method to run when we want to use the CLUI. It presents the commands help and exit. It possesses a loop, to make sure we can execute as many command as we want before closing the application.
- The help method
  - It reads a file in the folder help and displays its content

The execution of commands

The function execute(String commandLine) splits given the command line into an array of String objects. The split method uses a regex (regular expression). For example, it makes sure that the String "registerCourier \"Klay\" "Thompson\" \"K T\" \"0,0\" \"shoot\" becomes the array {"registerCourier", "Klay", "Thompson", "K T", "0,0", "shoot"}. We assume that the first element of the array is the name of the command and the others are the arguments. Then the function calls the execute(String commandName, String[] args) function. It passes the first element of the array as the first argument and the remaining subarray as the second argument. The execute(String commandName, String[] args) is quite simple. It associates a function to a command name and executes it. The task is fulfilled with a switch.

3.3 Class Finder

In this class we introduced a set of methods to look for a created object, by either its username or its name. We decided to create these methods in order to make the code on the commands simpler. We can find:

- A meal from a specific restaurant
- An order from the history of orders of a customer
- An order from the myfoodora system
- An item from a specific restaurant
- A registered courier
- A registered customer
- A registered restaurant

3.4 JUnit Tests

Even if the best way to test the CLUI is to write commands in the terminal, we decided to do JUnit tests for both the Application and the Finder, in order to check beforehand that they would work correctly. For the application, we had to write a really long test, in order to test all of the commands. We mainly used *assertNotNull* and *assertTrue*. The FinderTest was also long, we had to instantiate every object that could be searched (a couple of each type, in order to have a group from which to select). We used *assertTrue* to check that the found object was the correct one.

Chapter 4

Test Scenarios

4.1 Test Scenario 1 → responsible: Arturo Garrido

Command: runTest "TestScenario/testScenario1.txt" This is the mandatory test-scenario, asked for in the project documentation. It calls absolutely all commands, except for runTest. It develops the scenario described on the project documentation, but in a more complete way (in order to use all commands). It does the following:

- logon as manager (the created by default manager of the system, with username "ceo" and password "123456789")
- register two restaurants: the "Bonheur Antony" and the "Burger Lobby"
- register two customers: "Dominique Wilkins" and "Steve Nash"
- register two couriers: "Jason Kidd" and "Pau Gasol"
- display the available restaurants with reference to the orders processed (here they haven't processed any orders yet)
- display the customers registered in the system
- set the delivery policy of the system as "fastest delivery policy"
- associate a fidelity card, "LotteryFidelityCard", to the customer "Dominique"
- logout manager
- "Bonheur Antony" logs in as restaurant
- add four dishes to it's menu: "Soup" "Chop Suey" "Spring Rolls" and "Chocolate Cake"
- create a full meal called "Pekinese lunch"
- add a starter, main and dessert from the menu to the created meal
- save the created meal
- set the meal "Pekinese lunch" as a special offer
- show the meal "Pekinese lunch"

- logout restaurant
- ”Steve” logs in as customer
- gives consensus to receive special offers
- sets his contact information: phone and email adress
- logout customer
- ”Burguer Lobby” logs in as restaurant
- add four dishes to it’s menu: ”Salad” ”Chips” ”Beef burguer” and ”Fruit Mix”
- create a half meal called ”Burger lovers”
- add a starter and main from the menu to the created half meal
- save the created meal
- set the meal ”Burger lovers” as a special offer
- create a full meal called ”Burger lovers XL”
- add a starter, main and dessert from the menu to the created meal
- save the created meal
- set the meal ”Burger lovers XL” as a special offer
- remove the meal ”Burger lovers” from the special offer list
- logout restaurant
- ”ceo” logs in as manager
- displays the menu of ”Bonheur Antony”
- displays the menu of ”Burger Lobby”
- logout manager
- ”Dominique” logs in as customer
- creates an order called ”Friday dinner” from the restaurant ”Bonheur Antony”
- adds a meal to the order
- adds an item to the order
- ends order setting the date
- logout customer
- call to the findDeliverer method, to verify that the correct courier is assigned. ”Pau Gasol” is assigned to the order
- ”Pau Gasol” logs in as courier

- set his state as off duty
- logout courier
- findDeliverer method called once again. Now, as "Pau Gasol" is off duty, "Jason Kidd" is assigned the delivery
- "Pau Gasol" logs in as courier
- set his state as on duty
- logout courier
- "Steve" logs in as customer
- creates an order called "Netflix and chill" from the restaurant "Burger Lobby"
- adds a meal to the order
- adds two items to the order
- ends order setting the date
- logout customer
- "Steve" logs in as customer
- creates an order called "Lunch with friends" from the restaurant "Bonheur Antony"
- adds three meals to the order
- adds four items to the order
- ends order setting the date
- logout customer
- "ceo" logs in as manager
- shows total profit
- shows profit of the last month, giving the dates as inputs. The profit is of 4.95
- sets the profit policy, with a target profit of 5
- shows profit of the last month, giving the dates as inputs. The profit is now of 5, as expected
- shows couriers ordered with reference to the deliveries done. "Pau" will be first, as he did two deliveries and then "Jason" with only one.
- show customers
- show restaurants ordered with reference to the number of orders processed. "Bonheur Antony" will be first, as he processed two orders, and then "Burger Lobby"
- logout manager
- exit the system

4.2 Test Scenario 2 → responsible: Guy Tayou

Command: runTest "TestScenario/testScenario2.txt" An optional shorter test-scenario calling some commands:

- logon as manager (the created by default manager of the system, with username ceo and password 123456789)
- register three customers: Ronaldo Luis Nazrio de Lima, Jenifer Lopez J-Lo, Mboma Patrick
- register one restaurant: Detroit Restaurant
- register three couriers: Ryu, Kenand Chun-Li
- associate a fidelity card, LotteryFidelityCard, to the customer Jennifer
- logout manager
- Detroit Restaurant logs in as restaurant
- add six dishes to its menu: Ndol Poulet DG Miondo Poisson brais Beignet de mais and Guiness
- logout restaurant
- login as customer Ronaldo
- simulate placing of three orders by for Ronaldo and Detroit Restaurant
- sets the profit policy, with a target profit
- compute/display the total profit of the system since creation shows couriers ordered with reference to the deliveries done.
- shows couriers ordered with reference to the deliveries done
- show restaurants ordered with reference to the number of orders processed.
- logout manager
- exit the system

4.3 Test Scenario 3→ responsible: Guy Tayou

Command: runTest "TestScenario/testScenario3.txt" An optional shorter test-scenario calling some commands:

- logon as manager (the created by default manager of the system, with username ceo and password 123456789)
- registers two restaurants: Kentucky Fried Chicken and Le Pont d'Istanbul
- registers three customers: Bismack Biyombo, Kevin Garnett and Kawhi Leonard
- registers one courier: Klay Thompson
- sets the delivery policy of the system as FairOccupationDelivery

- logout manager
- Kentucky Fried Chicken logs in as restaurant
- adds three dishes to its menu and creates a meal containing all of them
- logout restaurant
- Le Pont dIstanbul logs in as restaurant
- adds two dishes to its menu
- logout restaurant
- login as customer Kevin
- simulates placing of one order for Kevin and Kentucky Fried Chicken
- login as customer Bismack
- simulates placing of one order for Kevin and Le Pont d'Istanbul
- login as customer Kawhi
- simulates placing of two orders for Kawhi and both restaurants
- sets the profit policy, with a target profit
- computes/displays the total profit of the system since creation shows couriers ordered with reference to the deliveries done
- shows couriers ordered with reference to the deliveries done
- shows customers
- shows restaurants ordered with reference to the number of orders processed
- logout manager
- exit the system

Chapter 5

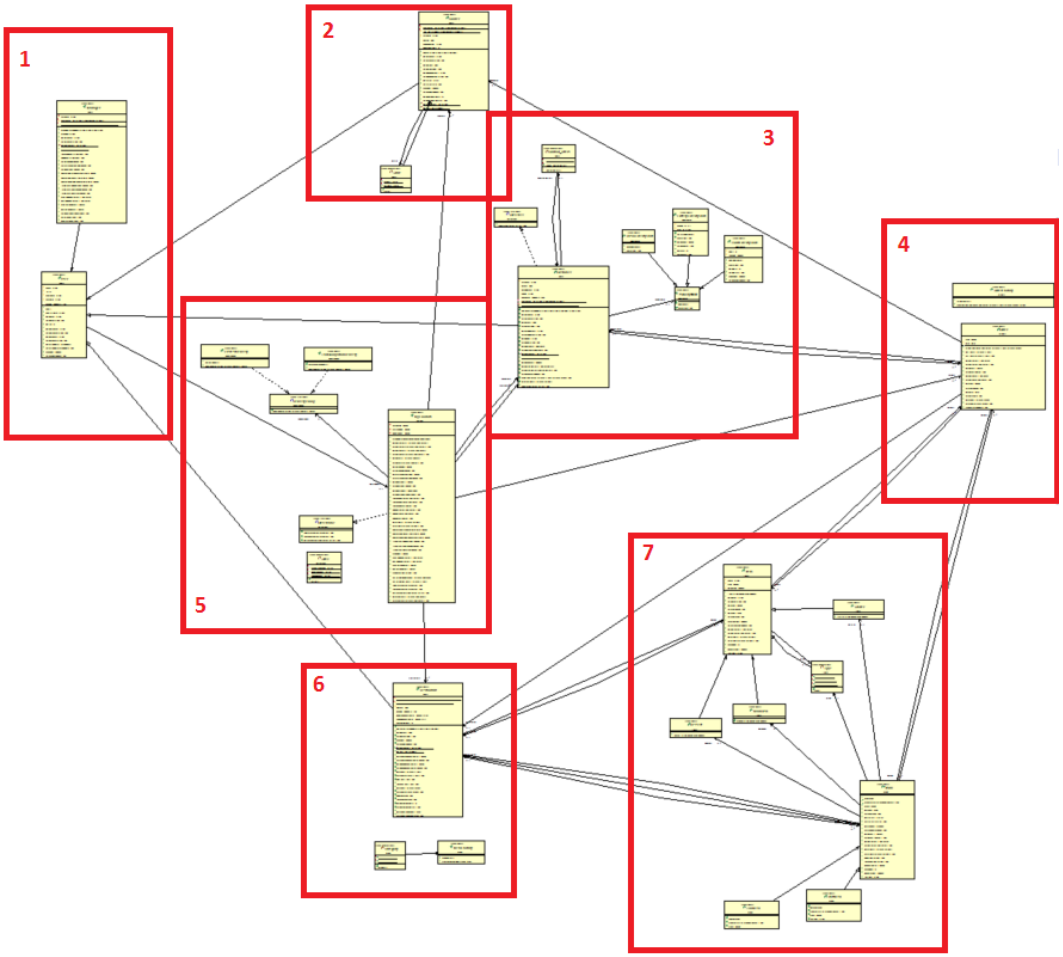
UML Diagram

5.1 Description

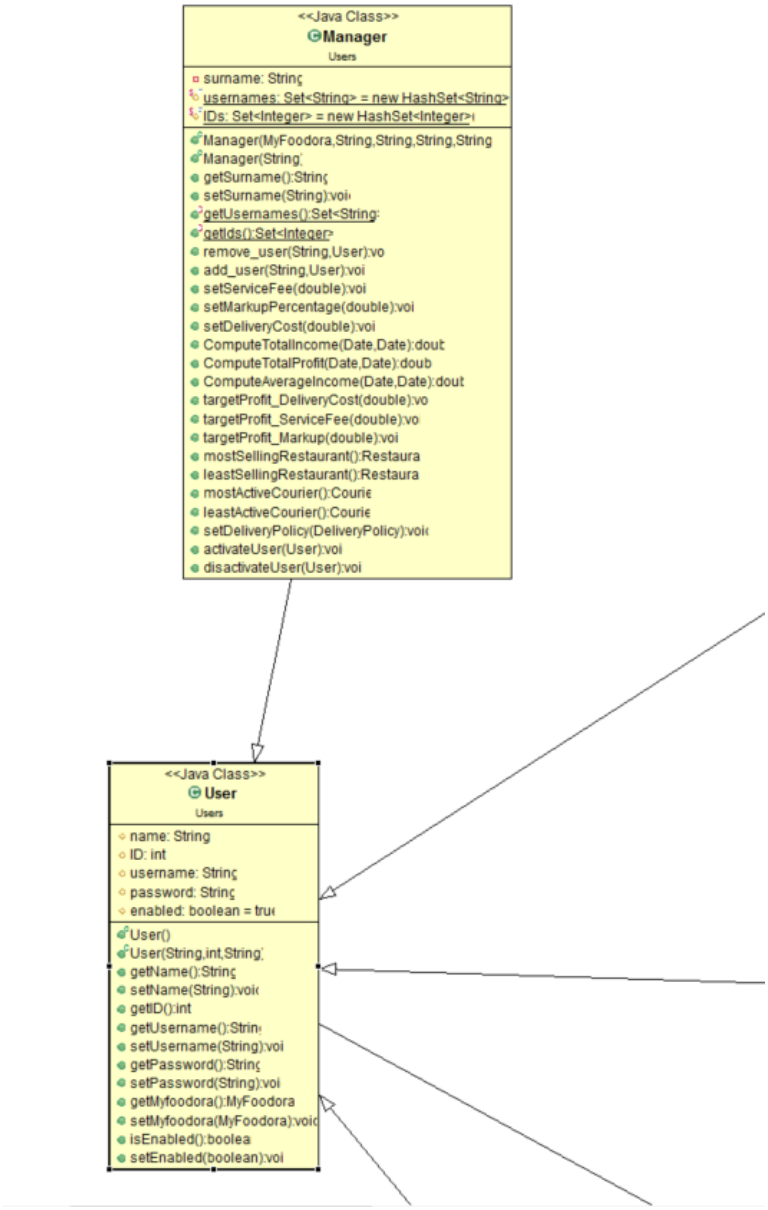
In order to design de UML diagram, we used an Eclipse plugin: ObjectAid. We read in several forums that it is the most effective and intuitive tool to design UML diagrams, and we thought it could be interesting to learn how to use a new one, so we decided to use it. Therefore, in the Eclipse file of our project, there is a package called Model that contains a class called model.ucls, where the diagram is represented (you can zoom and move around it). We decided to also create this section with screenshots of the diagram. We will first present the big picture, in which details cant be read. Then, we will show each zoomed part in detail.

5.2 Diagram

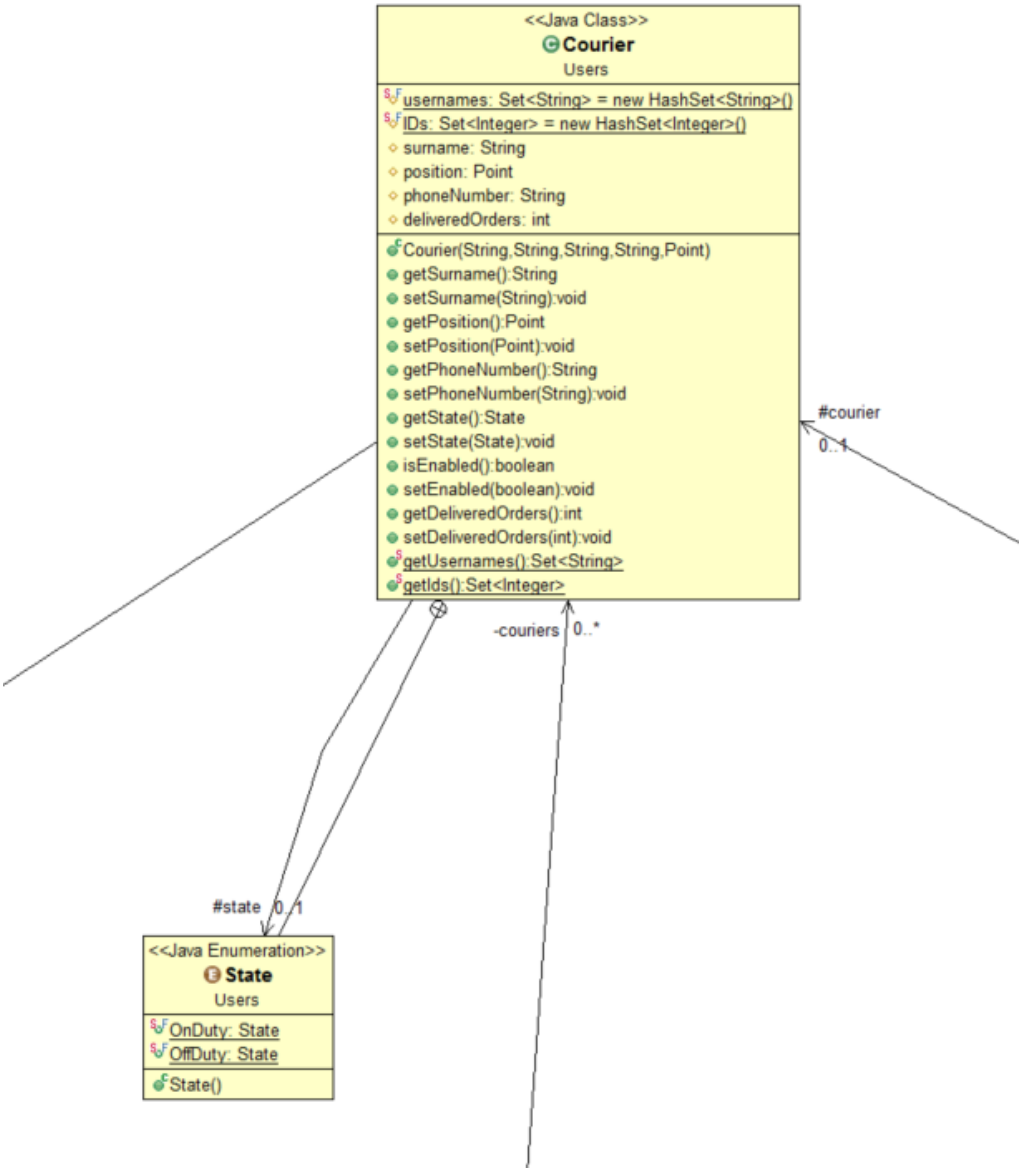
Big Picture



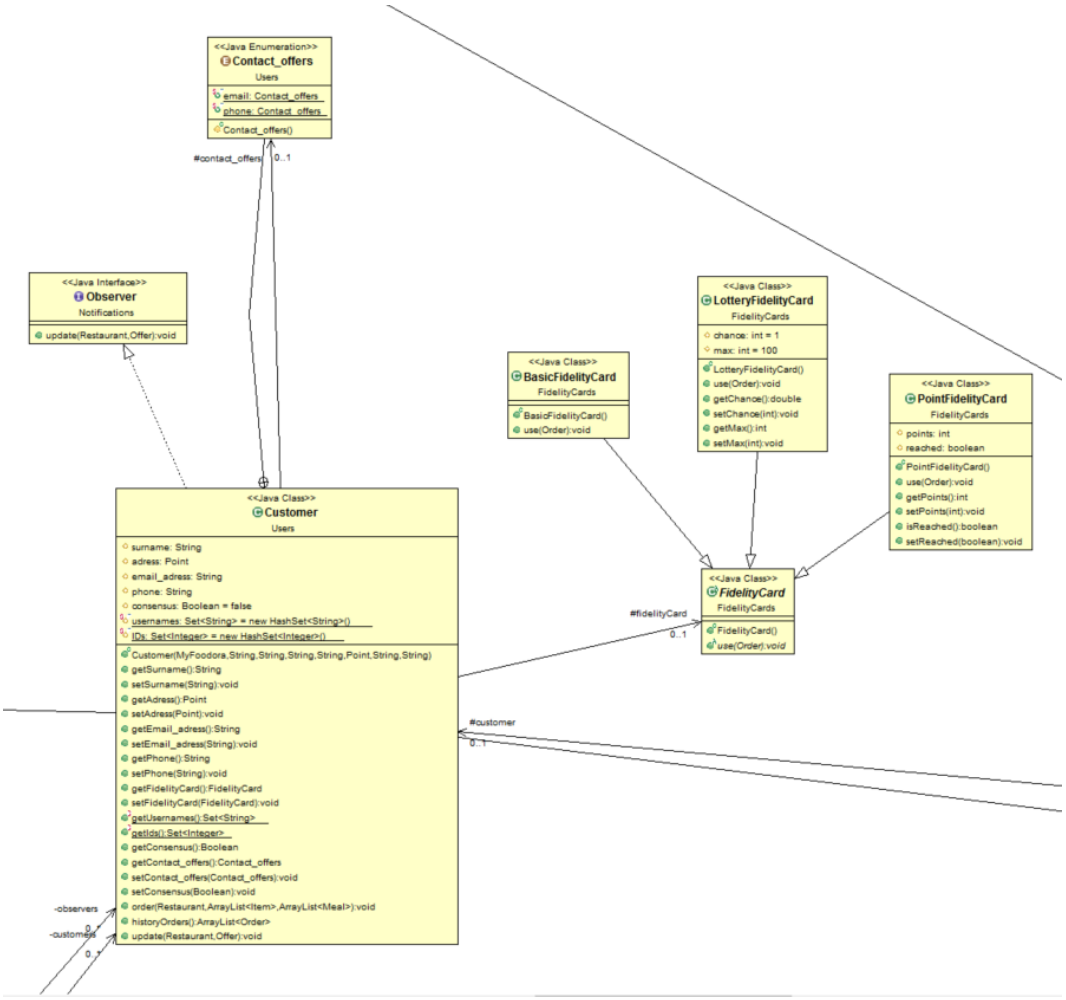
Square 1


<<Java Class>>
Manager
Users
surname: String
usernames: Set<String> = new HashSet<String>
IDs: Set<Integer> = new HashSet<Integer>
Manager(MyFoodora,String,String,String,String
Manager(String;
getSurname():String
setSurname(String):voi
getUsernames():Set<String>
getIds():Set<Integer>
remove_user(String,User):vo
add_user(String,User):voi
setServiceFee(double):voi
setMarkupPercentage(double):voi
setDeliveryCost(double):voi
ComputeTotalIncome(Date,Date):dout
ComputeTotalProfit(Date,Date):doub
ComputeAverageIncome(Date,Date):dout
targetProfit_DeliveryCost(double):vo
targetProfit_ServiceFee(double):vo
targetProfit_Markup(double):voi
mostSellingRestaurant():Restaura
leastSellingRestaurant():Restaura
mostActiveCourier():Courie
leastActiveCourier():Courie
setDeliveryPolicy(DeliveryPolicy):voi
activateUser(User):voi
disactivateUser(User):voi
<<Java Class>>
User
Users
name: String
ID: int
username: String
password: String
enabled: boolean = tru
User()
User(String,int,String;
getName():String
setName(String):voi
getID():int
getUsername():Strin
setUsername(String):voi
getPassword():String
setPassword(String):voi
getMyfoodora():MyFoodora
setMyfoodora(MyFoodora):voi
isEnabled():boolea
setEnabled(boolean):voi

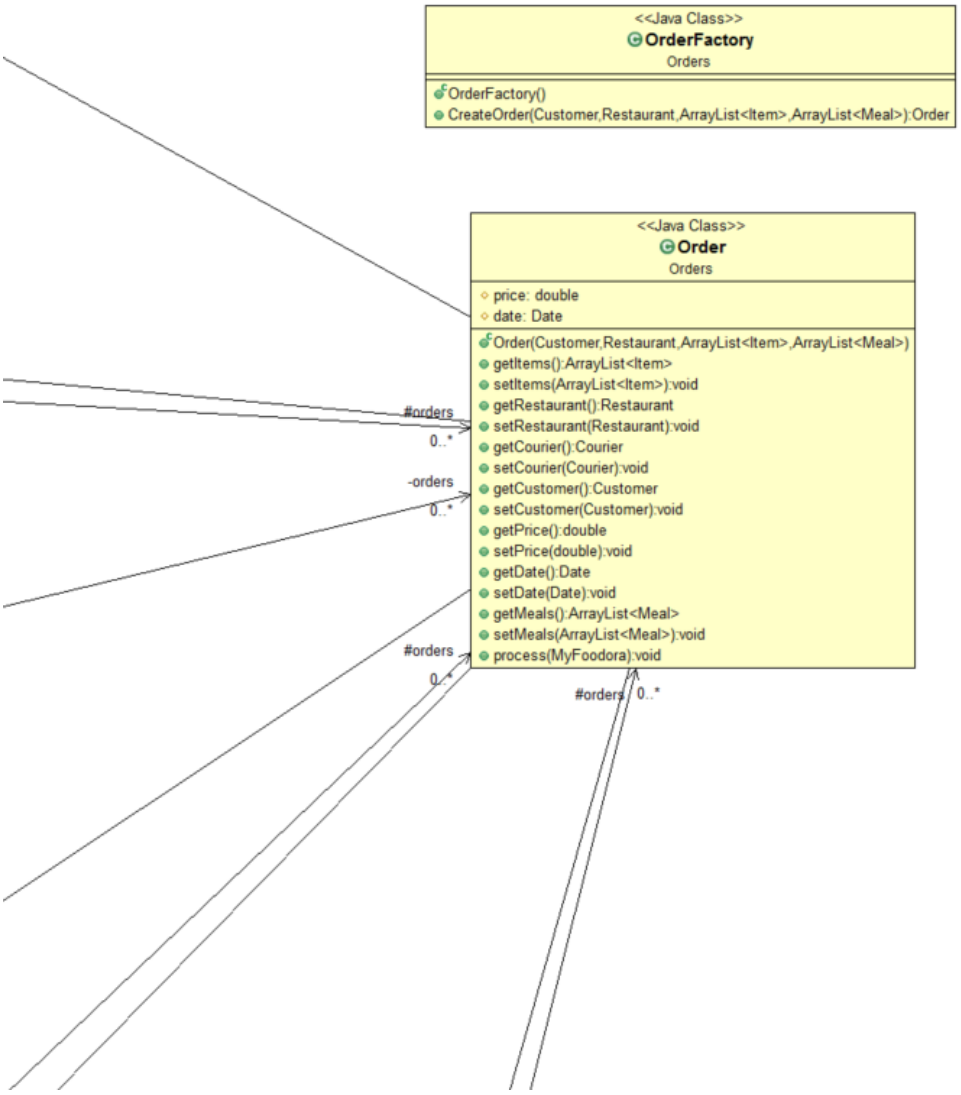
Square 2


<<Java Class>>
Courier
Users
usernames: Set<String> = new HashSet<String>()
IDs: Set<Integer> = new HashSet<Integer>()
surname: String
position: Point
phoneNumber: String
deliveredOrders: int
Courier(String,String,String,String,Point)
getSurname():String
setSurname(String):void
getPosition():Point
setPosition(Point):void
getPhoneNumber():String
setPhoneNumber(String):void
getState():State
setState(State):void
isEnabled():boolean
setEnabled(boolean):void
getDeliveredOrders():int
setDeliveredOrders(int):void
getUsernames():Set<String>
getIds():Set<Integer>
#courier
0..1
-couriers
0..*
#state
0..1
<<Java Enumeration>>
State
Users
OnDuty: State
OffDuty: State
State()

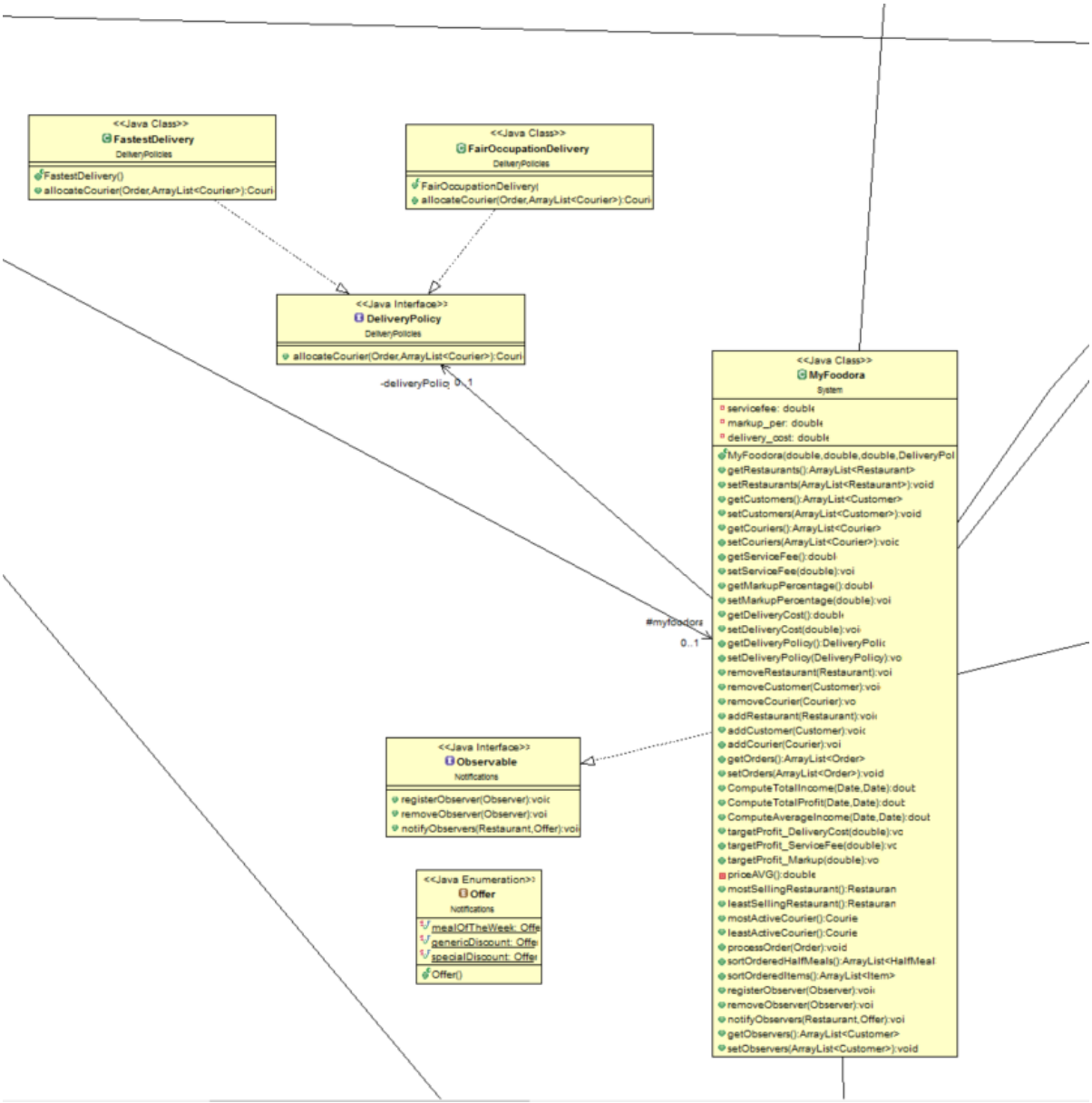
Square 3


<<Java Enumeration>>
Contact_offers
Users
email: Contact_offers
phone: Contact_offers
Contact_offers()
#contact_offers
0..1
<<Java Interface>>
Observer
Notifications
update(Restaurant,Offer):void
<<Java Class>>
LotteryFidelityCard
FidelityCards
chance: int = 1
max: int = 100
LotteryFidelityCard()
use(Order):void
getChance():double
setChance(int):void
getMax():int
setMax(int):void
<<Java Class>>
BasicFidelityCard
FidelityCards
BasicFidelityCard()
use(Order):void
<<Java Class>>
PointFidelityCard
FidelityCards
points: int
reached: boolean
PointFidelityCard()
use(Order):void
getPoints():int
setPoints(int):void
isReached():boolean
setReached(boolean):void
<<Java Class>>
Customer
Users
surname: String
adress: Point
email_adress: String
phone: String
consensus: Boolean = false
usernames: Set<String> = new HashSet<String>()
IDs: Set<Integer> = new HashSet<Integer>()
Customer(MyFoodora,String,String,String,String,Point,String,String)
getSurname():String
setSurname(String):void
getAdress():Point
setAdress(Point):void
getEmail_adress():String
setEmail_adress(String):void
getPhone():String
setPhone(String):void
getFidelityCard():FidelityCard
setFidelityCard(FidelityCard):void
getUsernames():Set<String>
getIds():Set<Integer>
getConsensus():Boolean
getContact_offers():Contact_offers
setContact_offers(Contact_offers):void
setConsensus(Boolean):void
order(Restaurant,ArrayList<Item>,ArrayList<Meal>):void
historyOrders():ArrayList<Order>
update(Restaurant,Offer):void
<<Java Class>>
FidelityCard
FidelityCards
FidelityCard()
use(Order):void
#fidelityCard
0..1
#customer
0..1
-observers
0..*
-customers
0..*

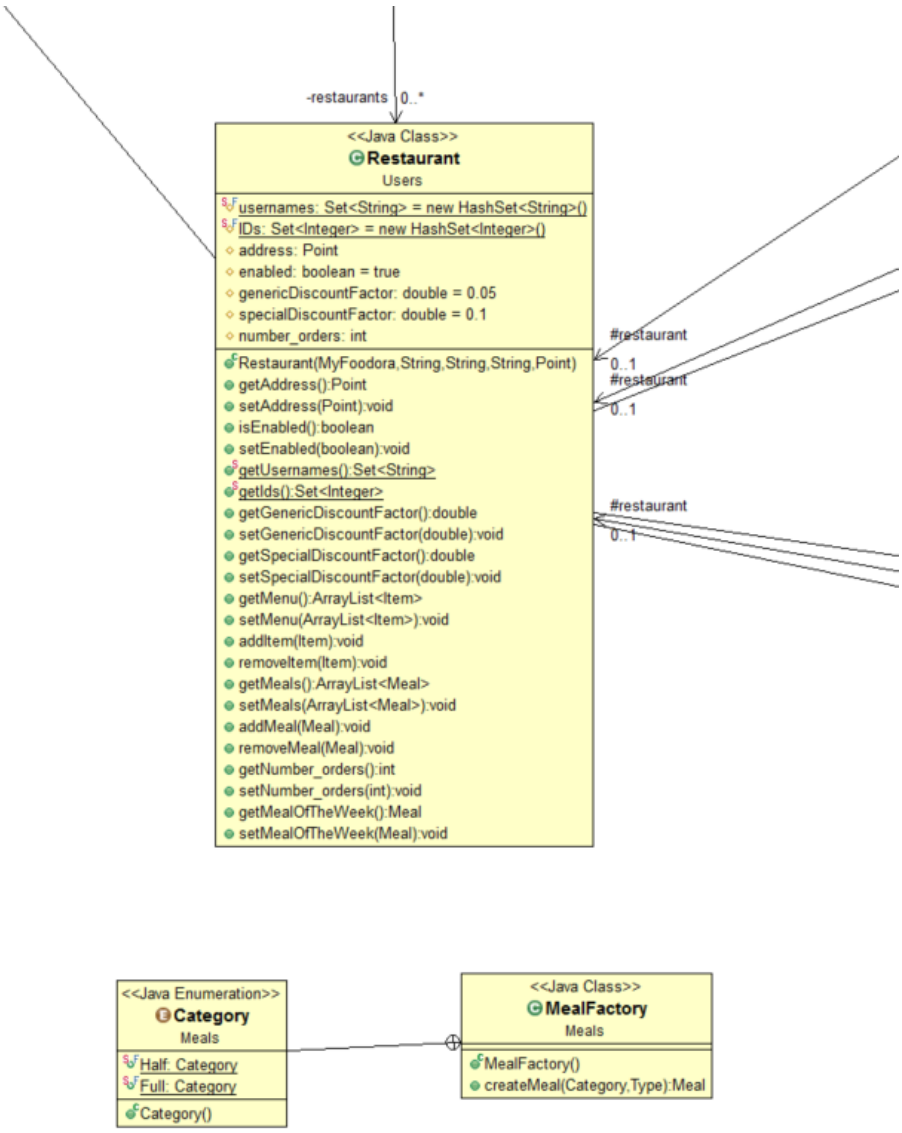
Square 4


<<Java Class>>
OrderFactory
Orders
OrderFactory()
CreateOrder(Customer,Restaurant,ArrayList<Item>,ArrayList<Meal>):Order
<<Java Class>>
Order
Orders
price: double
date: Date
Order(Customer,Restaurant,ArrayList<Item>,ArrayList<Meal>)
getItems():ArrayList<Item>
setItems(ArrayList<Item>):void
getRestaurant():Restaurant
setRestaurant(Restaurant):void
getCourier():Courier
setCourier(Courier):void
getCustomer():Customer
setCustomer(Customer):void
getPrice():double
setPrice(double):void
getDate():Date
setDate(Date):void
getMeals():ArrayList<Meal>
setMeals(ArrayList<Meal>):void
process(MyFoodora):void
#orders
0..*
-orders
0..*
#orders
0..*
#orders
0..*

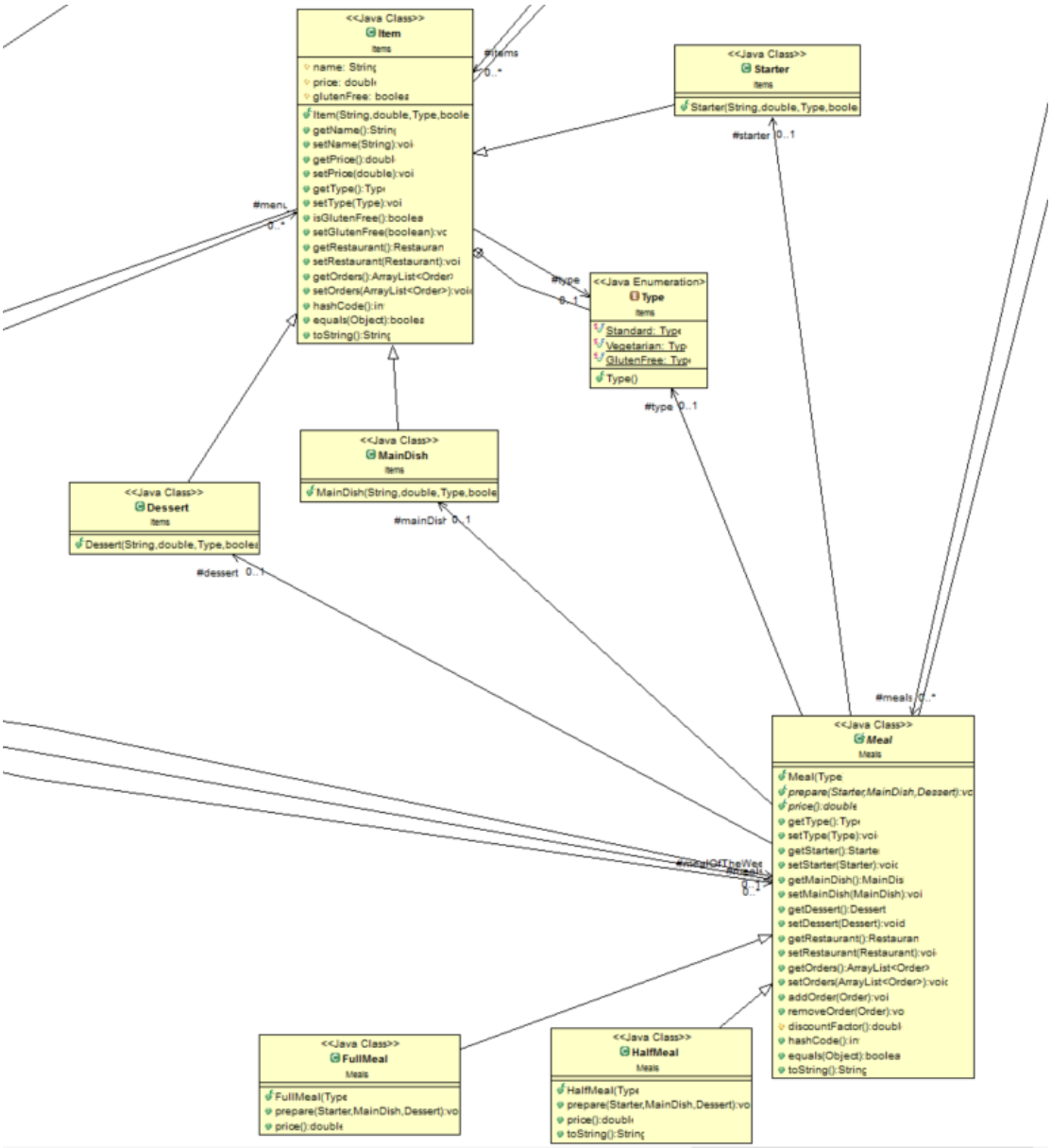
Square 5


<<Java Class>>
FastestDelivery
DeliveryPolicies
FastestDelivery()
allocateCourier(Order,ArrayList<Courier>):Couri
<<Java Class>>
FairOccupationDelivery
DeliveryPolicies
FairOccupationDelivery(
allocateCourier(Order,ArrayList<Courier>):Couri
<<Java Interface>>
DeliveryPolicy
DeliveryPolicies
allocateCourier(Order,ArrayList<Courier>):Couri
-deliveryPolicy 0..1
<<Java Class>>
MyFoodora
System
servicefee: doubl
markup_per: doubl
delivery_cost: doubl
MyFoodora(double,double,double,DeliveryPol
getRestaurants():ArrayList<Restaurant>
setRestaurants(ArrayList<Restaurant>):void
getCustomers():ArrayList<Customer>
setCustomers(ArrayList<Customer>):void
getCouriers():ArrayList<Courier>
setCouriers(ArrayList<Courier>):voic
getServiceFee():doubl
setServiceFee(double):voi
getMarkupPercentage():doubl
setMarkupPercentage(double):voi
getDeliveryCost():doubl
setDeliveryCost(double):voi
getDeliveryPolicy():DeliveryPolic
setDeliveryPolicy(DeliveryPolicy):vo
removeRestaurant(Restaurant):voi
removeCustomer(Customer):voi
removeCourier(Courier):vo
addRestaurant(Restaurant):voi
addCustomer(Customer):voi
addCourier(Courier):voi
getOrders():ArrayList<Order>
setOrders(ArrayList<Order>):void
ComputeTotalIncome(Date,Date):dout
ComputeTotalProfit(Date,Date):dout
ComputeAverageIncome(Date,Date):dout
targetProfit_DeliveryCost(double):vc
targetProfit_ServiceFee(double):vc
targetProfit_Markup(double):vo
priceAVG():double
mostSellingRestaurant():Restauran
leastSellingRestaurant():Restauran
mostActiveCourier():Courie
leastActiveCourier():Courie
processOrder(Order):void
sortOrderedHalfMeals():ArrayList<HalfMeal
sortOrderedItems():ArrayList<Item>
registerObserver(Observer):voi
removeObserver(Observer):voi
notifyObservers(Restaurant,Offer):voi
getObservers():ArrayList<Customer>
setObservers(ArrayList<Customer>):void
#myfoodora
0..1
<<Java Interface>>
Observable
Notifications
registerObserver(Observer):voi
removeObserver(Observer):voi
notifyObservers(Restaurant,Offer):voi
<<Java Enumeration>>
Offer
Notifications
mealOfTheWeek: Offe
genericDiscount: Offe
specialDiscount: Offe
Offer()

Square 6


-restaurants
0..*
<<Java Class>>
Restaurant
Users
usernames: Set<String> = new HashSet<String>()
IDs: Set<Integer> = new HashSet<Integer>()
address: Point
enabled: boolean = true
genericDiscountFactor: double = 0.05
specialDiscountFactor: double = 0.1
number_orders: int
Restaurant(MyFoodora,String,String,String,Point)
getAddress():Point
setAddress(Point):void
isEnabled():boolean
setEnabled(boolean):void
getUsernames():Set<String>
getIds():Set<Integer>
getGenericDiscountFactor():double
setGenericDiscountFactor(double):void
getSpecialDiscountFactor():double
setSpecialDiscountFactor(double):void
getMenu():ArrayList<Item>
setMenu(ArrayList<Item>):void
addItem(Item):void
removeItem(Item):void
getMeals():ArrayList<Meal>
setMeals(ArrayList<Meal>):void
addMeal(Meal):void
removeMeal(Meal):void
getNumber_orders():int
setNumber_orders(int):void
getMealOfTheWeek():Meal
setMealOfTheWeek(Meal):void
#restaurant
0..1
#restaurant
0..1
#restaurant
0..1
<<Java Enumeration>>
Category
Meals
Half: Category
Full: Category
Category()
<<Java Class>>
MealFactory
Meals
MealFactory()
createMeal(Category,Type):Meal

Square 7


<<Java Class>>
Item
Items
name: String
price: double
glutenFree: boolea
Item(String,double,Type,boole
getName():Strin
setName(String):voi
getPrice():doubl
setPrice(double):voi
getType():Typ
setType(Type):voi
isGlutenFree():boolea
setGlutenFree(boolean):vc
getRestaurant():Restauran
setRestaurant(Restaurant):voi
getOrders():ArrayList<Order
setOrders(ArrayList<Order>):voi
hashCode():in
equals(Object):boolea
toString():Strin
#items
0..*
<<Java Class>>
Starter
Items
Starter(String,double,Type,boole
#starter 0..1
#menu
0..*
#type
0..1
<<Java Enumeration>
Type
Items
Standard: Typ
Vegetarian: Typ
GlutenFree: Typ
Type()
#type 0..1
<<Java Class>>
MainDish
Items
MainDish(String,double,Type,boole
#mainDish 0..1
<<Java Class>>
Dessert
Items
Dessert(String,double,Type,boolea
#dessert 0..1
#meals 0..*
<<Java Class>>
Meal
Meals
Meal(Type
prepare(Starter,MainDish,Dessert):vc
price():double
getType():Typ
setType(Type):voi
getStarter():Starte
setStarter(Starter):voi
getMainDish():MainDis
setMainDish(MainDish):voi
getDessert():Dessert
setDessert(Dessert):void
getRestaurant():Restauran
setRestaurant(Restaurant):voi
getOrders():ArrayList<Order>
setOrders(ArrayList<Order>):voi
addOrder(Order):voi
removeOrder(Order):vo
discountFactor():doubl
hashCode():in
equals(Object):boolea
toString():Strin
0..1
<<Java Class>>
FullMeal
Meals
FullMeal(Type
prepare(Starter,MainDish,Dessert):vo
price():double
<<Java Class>>
HalfMeal
Meals
HalfMeal(Type
prepare(Starter,MainDish,Dessert):vo
price():doubl
toString():Strin

Chapter 6

Conclusion

This project was, with no doubt, the heaviest workload of the course. It required much more time than studying for the exam, for example. However, it was a very satisfying experience. It is important to learn the theory behind Object Oriented Programming and the Java language, and of course the PCs are a key element for learning how to apply this knowledge. However, the fact of facing a real job that a software developer would perform, is the most interesting thing of the course. Even if we could have done some things better, we are very proud of the work we have done, and the fact of having a tangible outcome is very satisfying. Thank you very much for taking the time to read the report.