myFoodora Project - Report

BONATTO Alisson
ADIB Aymane Chaoui

Object Oriented Programming Course
Engineering Cycle - 2nd year
SG8

May 2025

Introduction

In the context of the Object-Oriented Software Engineering course at CentraleSupélec, this project aimed to design and implement a food delivery platform named myFoodora, inspired by real-world services such as Foodora or Deliveroo. The main objective was to develop a modular and extensible Java-based system capable of managing the interactions between customers, restaurants, couriers, and managers, while incorporating key software engineering principles such as encapsulation, abstraction, and the use of design patterns.

The project was divided into two main parts: the development of the core infrastructure of the myFoodora system and the implementation of a command-line user interface (CLUI) to interact with it. The core includes functionalities such as user registration, order management, delivery coordination, and profit calculation, all while maintaining flexibility through the use of customizable policies for pricing, delivery, and profit optimization. The CLUI allows users to interact with the system using structured commands, enabling end-to-end testing and simulation of real-life scenarios.

This report presents the design choices, implementation details, and testing strategy adopted throughout the project. It highlights the structure of the system, the application of object-oriented principles, and the design patterns used to ensure a scalable and maintainable architecture. Additionally, it includes a description of test scenarios and how to reproduce them to validate the correctness and robustness of our solution.

Contents

1 Core

This section aims to explain the design choices made throughout the implementation of the core of our application. As one can see in the subsections' division, our core was implemented through a number of connected packages that gather the different classes that interact the most with each other and that constitute a compact set of commands with high interdependence.

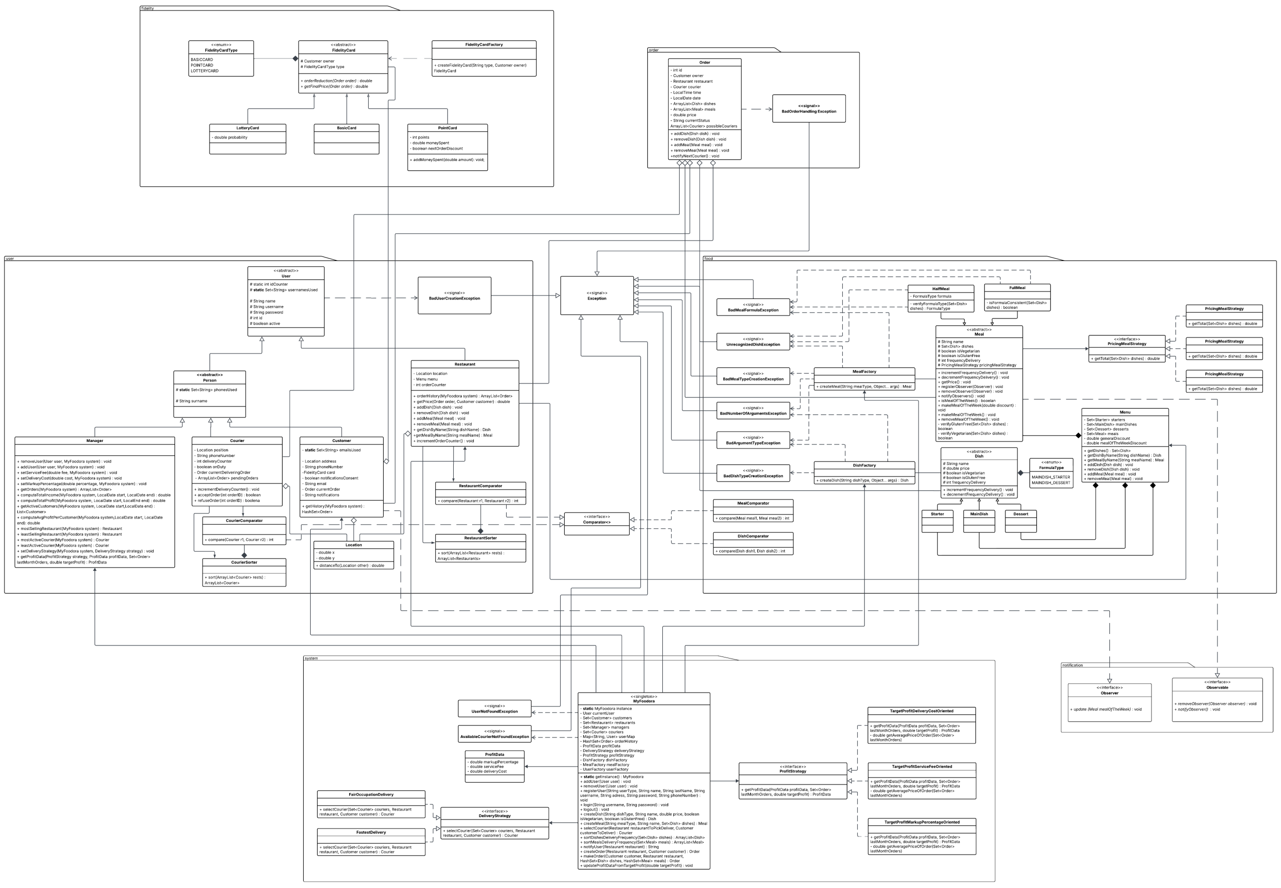


Figure 1: UML Diagram look

This above represents the general alure of the diagram we came up with. For a more readable document, you could check the uml_diagrams folder in the project where you will find it in pdf. For the rest of this report, we will be using package-sized diagrams for explanations.

1.1 Food Package

Developped by Bonatto Alisson

The food package contains all entities related to food, such as the dishes, meals, menu in addition to the different factories, strategies and exceptions. Figure 8 shows the class diagram in UML of the food package.

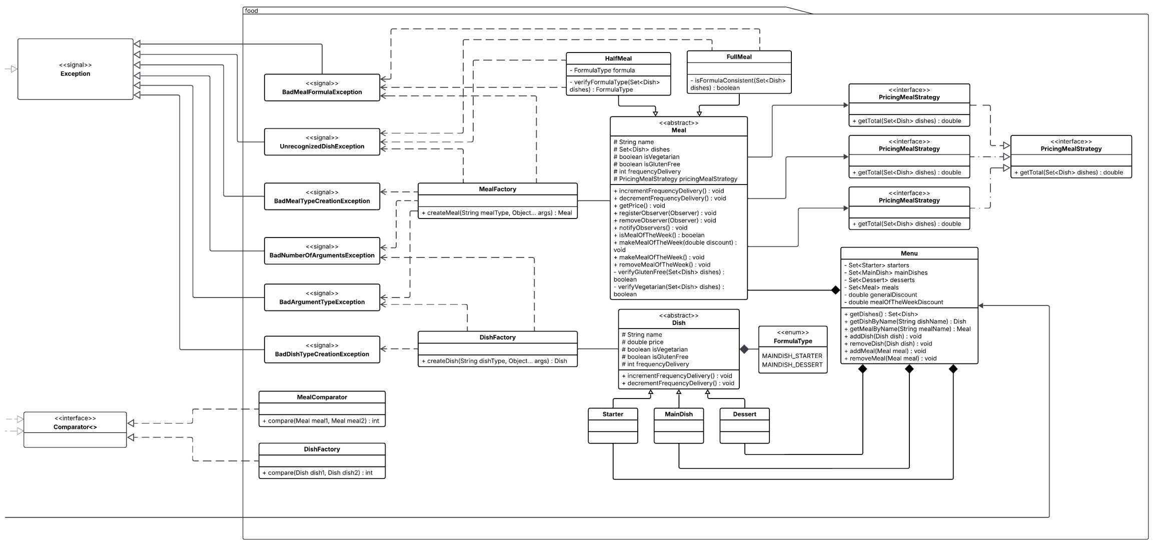

food
<<signal>>
Exception
<<signal>>
BadMealFormulaException
<<signal>>
UnrecognizedDishException
<<signal>>
BadMealTypeCreationException
<<signal>>
BadNumberOfArgumentsException
<<signal>>
BadArgumentTypeException
<<signal>>
BadDishTypeCreationException
HalfMeal
- FormulaType formula
- verifyFormulaType(Set<Dish> dishes) : FormulaType
FullMeal
- isFormulaConsistent(Set<Dish> dishes) : boolean
<<abstract>>
Meal
# String name
# Set<Dish> dishes
# boolean isVegetarian
# boolean isGlutenFree
# int frequencyDelivery
# PricingMealStrategy pricingMealStrategy
+ incrementFrequencyDelivery() : void
+ decrementFrequencyDelivery() : void
+ getPrice() : void
+ registerObserver(Observer) : void
+ removeObserver(Observer) : void
+ notifyObservers() : void
+ isMealOfTheWeek() : booelan
+ makeMealOfTheWeek(double discount) : void
+ makeMealOfTheWeek() : void
+ removeMealOfTheWeek() : void
- verifyGlutenFree(Set<Dish> dishes) : boolean
- verifyVegetarian(Set<Dish> dishes) : boolean
MealFactory
+ createMeal(String mealType, Object... args) : Meal
DishFactory
+ createDish(String dishType, Object... args) : Dish
<<interface>>
PricingMealStrategy
+ getTotal(Set<Dish> dishes) : double
<<interface>>
PricingMealStrategy
+ getTotal(Set<Dish> dishes) : double
<<interface>>
PricingMealStrategy
+ getTotal(Set<Dish> dishes) : double
<<interface>>
PricingMealStrategy
+ getTotal(Set<Dish> dishes) : double
Menu
- Set<Starter> starters
- Set<MainDish> mainDishes
- Set<Dessert> desserts
- Set<Meal> meals
- double generalDiscount
- double mealOfTheWeekDiscount
+ getDishes() : Set<Dish>
+ getDishByName(String dishName) : Dish
+ getMealByName(String mealName) : Meal
+ addDish(Dish dish) : void
+ removeDish(Dish dish) : void
+ addMeal(Meal meal) : void
+ removeMeal(Meal meal) : void
<<abstract>>
Dish
# String name
# double price
# boolean isVegetarian
# boolean isGlutenFree
# int frequencyDelivery
+ incrementFrequencyDelivery() : void
+ decrementFrequencyDelivery() : void
<<enum>>
FormulaType
MAINDISH_STARTER
MAINDISH_DESSERT
Starter
MainDish
Dessert
<<interface>>
Comparator<>
MealComparator
+ compare(Meal meal1, Meal meal2) : int
DishFactory
+ compare(Dish dish1, Dish dish2) : int

Figure 2: UML Diagram for the Food Package.

1.1.1 Dish Class

The Dish abstract class is the parent class of all types of dishes. It contains all necessary dish information, such as name, price, the frequency that this dish was delivered and two boolean variables indicating if the dish is vegetarian or gluten free. The delivery frequency can be configured directly via a setter or it can be incremented or decremented via the methods `incrementFrequencyDelivery()` and `decrementFrequencyDelivery()`.
The method `equals()` was overridden in order to establish that two dishes are equal if they have the same name. The method `hasCode()` was overridden in order to establish an useful hash code to be used in a search of a hash set of dishes. The new hash code is the hash code of the string defining the name of the dish.

1.1.2 Starter, MainDish and Dessert Classes

The Starter, MainDish and Dessert classes are concrete implementations of the parent class Dish. These classes don't have a particular attribute or method, but they were made to be able to distinguish between the different types of dishes. This design choice is useful when we are dealing with a set of different types of dishes using polymorphism and also to distinguish the dish type based on its class.
These three classes can be part of a meal (depending on the type o the meal) and also are part of a Menu. These particularities will be explain throughout this section.

1.1.3 Meal Class

The Meal abstract class is the parent class of the two types of meals (i.e. two sub-classes): FullMeal and HalfMeal. A meal is composed of a set of dishes, a name, the frequency that this meal was delivered, two boolean variables indicating if the meal is vegetarian or gluten free, and a pricing strategy that defines if the meal will have the basic discount or will have the special discount once that this meal is a meal of the week.
The price is not an attribute of the class: it's necessary to use the `getPrice()` method that returns the price of the meal. This method returns the retun of the `getTotal()` method of the pricing strategy chosen: the final price is composed of the sum of all dishes of the meal and a discount. This design design choice was made in order to be possible to chose between different ways of computing the price based on the chosen strategy. The restaurant can chose between the generic discount strategy, which implements the basic discount on the meal (the standard value of this discount is 5%), the meal of the week strategy, which implements a bigger discount on the meal (the standard value of this discount is 10%), and a no discount strategy, which implements no discount. For each Menu, there exists at least one meal of the week.
Nevertheless, the class has getters and setters in addition to the following methods:

- **incrementFrequencyDelivery() and decrementFrequencyDelivery()**: responsible for increment and decrement the delivery frequency.
- **verifyGlutenFree(Set<Dish> dishes) and verifyVegetarian(Set<Dish> dishes)**: private methods responsible for checking if the meal is gluten free or vegetarian based on the dishes passed to the constructor. These methods are called in the constructor in order to fill the two respective boolean variables. A meal is gluten free if all the dishes of the meal are gluten free and a meal is vegetarian is all the dishes of the meal are vegetarian.
- **makeMealOfTheWeek() and makeMealOfTheWeek(double discount)**: method overloading responsible for making the meal a meal of the week. One of the methods can receive a discount factor if the user wants to implement a discount factor different of the standard one.

This class also implements a notification system using the observer design pattern (further explained in Subsection 1.3) to notify customers who have consented to receive notifications when a new meal of the week is created. To do so, a static attribute with a list of observers was implemented in addition to several methods to register/remove an observer and to notify them: `registerObserver(notification.Observer observer), removeObserver(notification.Observer observer), and notifyObservers()`. The method `makeMealOfTheWeek()` calls the notification method to notify the customers.
The Meal class contains two constructors: `Meal(String name, Set<Dish> dishes)` where the generic discount is applied; `Meal(String name, Set<Dish> dishes, PricingMealStrategy pricingStrategy)` where it's possible to chose a different pricing strategy during the creation of the object, and if its pricing strategy is a meal of the week, the customers are notified.
The method `equals(Object other)` was overridden to specify that two meals are equal if they have the same name. The method `hashCode()` was overridden to specify that the hash code of this class is the hash code of the meal's name.

1.1.4 HalfMeal and FullMeal Classes

A half meal is a meal composed of two dishes: a main dish and a dessert or a main dish and a starter. A full meal is a meal composed of three dishes: a starter, a main dish and a dessert. The HalfMeal and FullMeal classes are the child concrete classes of the Meal class, representing a half meal and a full meal.
In addition to the attributes and methods of the parent class, the HalfMeal class has an attribute called `formula` of the type `FormulaType`, which is an enum created to specify the formula of the meal, being possible to have the value `MAINDISH_STARTER` or `MAINDISH_DESSERT`. A method called `verifyFormulaType(Set<Dish> dishes)` was implement to verify which formula is being created, and if it is an invalid formula (two main dishes or a starter and a dessert, for example), the method throws an exception called `BadMealFormulaException`. If one of the dishes that are begin verified doesn't belong to any know class, the method throws the excpetion `UnrecognizedDishException`. This method returns the formula type to fill the `formula` attribute in the constructor.
Similarly, the FullMeal class has a method called `isFormulaConsistent(Set<Dish> dishes)` which verifies the consistency of the meal and throws an exception if necessary. The difference is that this method doesn't return anything, once that we already know the formula of a full meal: a starter, a main dish and a dessert.

1.1.5 Menu Class

A menu is composed of: a set of starters, a set of main dishes, a set of desserts, a set of meals and a generic discount an a meal of the week discount. This class contains an overload of three constructors: a constructor that receives all the mentioned attributes; a constructor that receives only the discount factors; a constructor that doesn't receive any argument. All the constructors that receives the discount factors, and the respective setters as well, throws the exception `IllegalArgumentException` if the discount is less than 0 or higher than 1. When these discounts are not specified, the standard values are used.
A dish and a meal can be added or removed from the menu via the methods `addMeal(Meal meal), removeMeal(Meal meal),addDish(Dish dish)` and `removeDish(Dish dish)`. It's important to notice the use of the polymorphism in the dish related methods, which facilitates the implementation of a single method to all type of dishes.

Other two methods were implemented in order to help in further implementations, the `getDishByName(String dishName)` and `getMealByName(String mealName)` that returns a dish and a meal by passing its name, respectively.

1.1.6 DishComparator and MealComparator Classes

The DishComparator and MealComparator are classes that implements the `Comparator` interface. Both comparators were made in order to compare two dishes (and two meals) by their delivery frequencies. These comparators are used to sort a list of dishes and a list of meals by the delivery frequency.

1.1.7 DishFactory Class

In order to detach the implementation of the classes in this package of the usage of them, factories were made, including a factory for the dish child classes. The DishFactory class returns a instance of a dish with the method `createDish(String dishType, Object... args)`. The first argument is the dish type, i.e. "STARTER", "MAINDISH" or "DESSERT", while the args are the arguments used to construct a dish. The arguments are of the generic type Object, but each one of them are checked to verify if it is a instance of the expected class to pass as argument to the correct constructor.
Three different types of exceptions can be thrown by the mentioned method depending of the occurred error:

- **BadNumberOfArgumentsException**, if an incorrect number of arguments was received.
- **BadDishTypeCreationException**, if the dish type wasn't recognized.
- **BadArgumentTypeException**, if one of the arguments is not of the correct type.

1.1.8 MealFactory Class

Similarly to the DishFactory, the MealFactory is responsible to create meals through the method `createMeal(String mealType, Object... args)`. The first argument is the meal type, i.e. "FULLMEAL" or "HALFMEAL" and the args are the the arguments to construct the meal. The following exceptions can be thrown:

- **BadNumberOfArgumentsException**, if an incorrect number of arguments was received.
- **BadArgumentTypeException**, if one of the arguments is not of the correct type.
- **UnrecognizedDishException**, if any dish in the set is not recognized as Starter, MainDish or Dessert.
- **BadMealTypeCreationException**, if the meal type is unrecognized.
- **BadMealFormulaException**, if the meal composition is invalid.

1.1.9 PricingMealStrategy Interface and its Strategy Classes

The PricingMealStrategy interface is responsible for ensure the implementation of the method `getTotal(Set<Dish> dishes)` which is responsible for getting the total price of a meal. The three different classes and strategies implements the PricingMealStrategy:

- **NoDiscountMeal Class**, if the meal doesn't have a discount - then the total price of a meal is just the sum of the price of each dish.
- **GeneralDiscountMeal**, if the meal has the generic discount applied (the standard value is 5% is used if no discount is passed to the constructor).
- **MealOfTheWeekDiscount**, if the meal is a meal of the week and applies the special discount (the standard value is 10% is used if no discount is passed to the constructor).

1.2 User Package

Developped by Adib Aymane

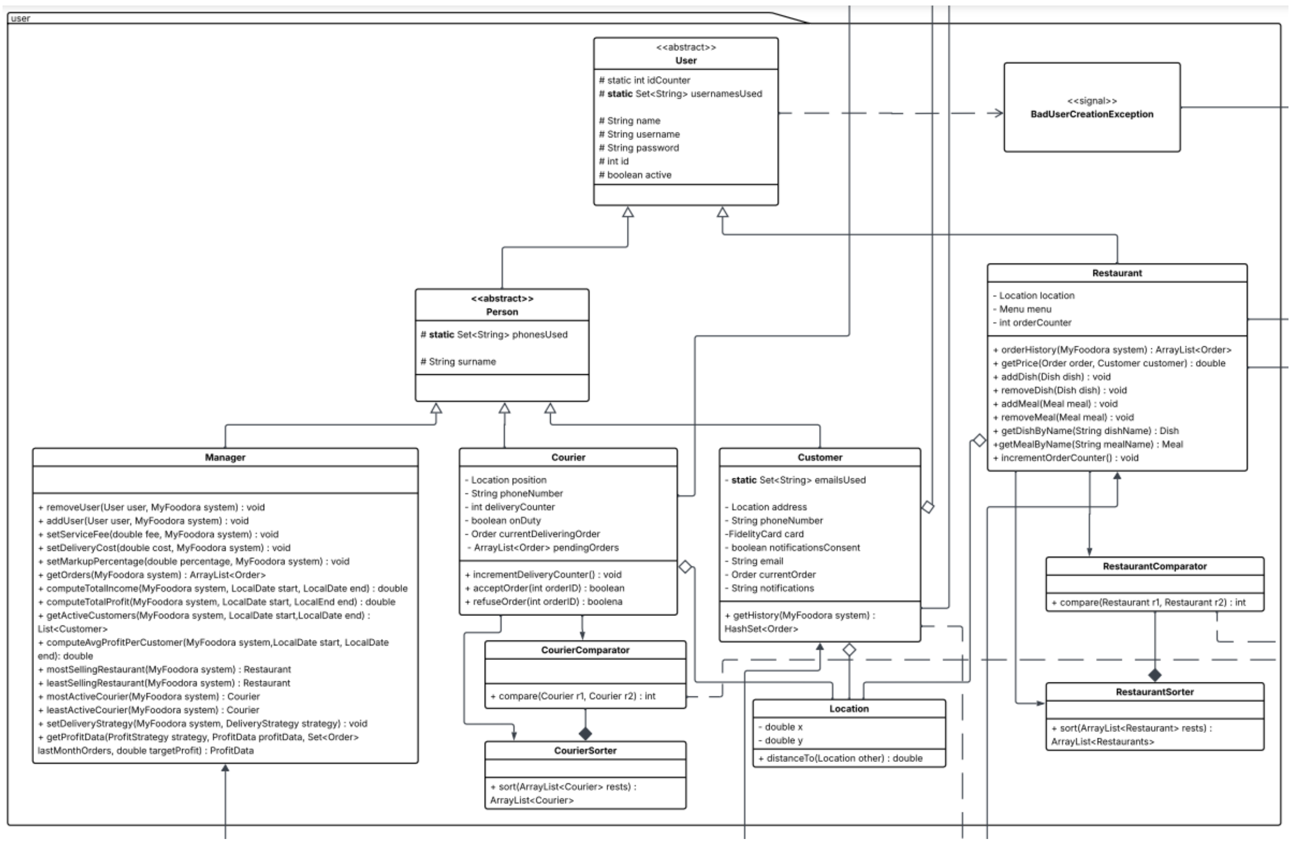

user
<<abstract>> User
# static int idCounter
# static Set<String> usernamesUsed
# String name
# String username
# String password
# int id
# boolean active
<<signal>> BadUserCreationException
<<abstract>> Person
# static Set<String> phonesUsed
# String surname
Restaurant
- Location location
- Menu menu
- int orderCounter
+ orderHistory(MyFoodora system) : ArrayList<Order>
+ getPrice(Order order, Customer customer) : double
+ addDish(Dish dish) : void
+ removeDish(Dish dish) : void
+ addMeal(Meal meal) : void
+ removeMeal(Meal meal) : void
+ getDishByName(String dishName) : Dish
+getMealByName(String mealName) : Meal
+ incrementOrderCounter() : void
Manager
+ removeUser(User user, MyFoodora system) : void
+ addUser(User user, MyFoodora system) : void
+ setServiceFee(double fee, MyFoodora system) : void
+ setDeliveryCost(double cost, MyFoodora system) : void
+ setMarkupPercentage(double percentage, MyFoodora system) : void
+ getOrders(MyFoodora system) : ArrayList<Order>
+ computeTotalIncome(MyFoodora system, LocalDate start, LocalDate end) : double
+ computeTotalProfit(MyFoodora system, LocalDate start, LocalEnd end) : double
+ getActiveCustomers(MyFoodora system, LocalDate start,LocalDate end) : List<Customer>
+ computeAvgProfitPerCustomer(MyFoodora system,LocalDate start, LocalDate end): double
+ mostSellingRestaurant(MyFoodora system) : Restaurant
+ leastSellingRestaurant(MyFoodora system) : Restaurant
+ mostActiveCourier(MyFoodora system) : Courier
+ leastActiveCourier(MyFoodora system) : Courier
+ setDeliveryStrategy(MyFoodora system, DeliveryStrategy strategy) : void
+ getProfitData(ProfitStrategy strategy, ProfitData profitData, Set<Order> lastMonthOrders, double targetProfit) : ProfitData
Courier
- Location position
- String phoneNumber
- int deliveryCounter
- boolean onDuty
- Order currentDeliveringOrder
- ArrayList<Order> pendingOrders
+ incrementDeliveryCounter() : void
+ acceptOrder(int orderID) : boolean
+ refuseOrder(int orderID) : boolena
Customer
- static Set<String> emailsUsed
- Location address
- String phoneNumber
-FidelityCard card
- boolean notificationsConsent
- String email
- Order currentOrder
- String notifications
+ getHistory(MyFoodora system) : HashSet<Order>
RestaurantComparator
+ compare(Restaurant r1, Restaurant r2) : int
CourierComparator
+ compare(Courier r1, Courier r2) : int
Location
- double x
- double y
+ distanceTo(Location other) : double
RestaurantSorter
+ sort(ArrayList<Restaurant> rests) : ArrayList<Restaurants>
CourierSorter
+ sort(ArrayList<Courier> rests) : ArrayList<Courier>

Figure 3: UML Diagram for the User Package

Disclaimer : *Due to the complexity of this package's diagram, external dependencies may have not been completely represented (such as the presence of Order objects or FidelityCard objects in the Customer Class).*

The user package regroups all of the different users of the MyFoodora system, alongside the different classes that interect directly with them (Comparators, Sorter, Exception), with the Location class.

1.2.1 Location Class

This class was developped due to the need for a representation for the different types of locations in the app ; namely the Courier's position, the Customer's address and the Restaurant's location. It is composed of an X and Y coordinate represented as private double attributes, with their own setters and getters.
The only other method developped (apart from the setters/getters) was the distanceTo(Location other) that computes the distance from our location to another. This will be helpful in the Courier assignement strategy where it is needed to calculate the distance between the courier and the restaurant and the one between the restaurant and the customer's address.

1.2.2 User Class

The parent class for all the different users of the MyFoodora system, implemented as an abstract class. It contains all the common attributes of the different users of the app, such as a name, a unique username to ensure a safe login, a password, a unique id and a boolean that represents whether the account is active or not. In order to ensure that the id and the username are unique, we implemented static variables : an idCounter (int) to keep track of the latest id attributed to users and a usernamesUsed (Set<String>) that contain all the different usernames already attributed to an account. The creation of an object of a subclass of User with a

given username will check if the username is already used, and will throw a BadUserCreationException if that is the case.
The abstract class does not have any concrete nor abstract method (beside getters/setters), as there is no action in the app that may be common to all the different users. We put 2 constructors in place:

- **User(String name, String username, String password) :** This one is for when the user wants to create account while customizing all his information. The default active value will be true, since it does not make sense to create an account while not being active.
- **User(String password) :** if the user does not care about personnalization, or cares about privacy. The default name will be *"User "+id* while the default username will be *"user_"+id*.

We could go further and define constructors with combinations of keeping/leaving either the name or the username, but for the sake of simplicity it was decided to only keep those two.

1.2.3 Restaurant Class

A restaurant can have an account on the app, and its information will be stored in the Restaurant class that inherits directly from the User class. Its attribute, apart from the ones inherited, are the location (a Location object), the menu (a food.Menu object) and the orderCounter (int).

Each restaurant has only one menu, and each menu is linked to only one restaurant. That is why many methods may be redundant between the two classes(add/removeDish, add/removeMeal,setGeneral/SpecialDiscount...), but the implementation of most of those methods use the ones implemented in the Menu classes. We decided to make the "real" implementation in the Menu class since those methods mainly manipulate attributes of the Menu, while we re-implemented those methods in Restaurant since restaurant objects are the one who will be mainly manipulated through the CLI, so it is easier to make "dummy" methods that call Menu's methods directly from the Restaurant objects.

The orderCounter attribute keeps track of the number of orders made through the app. It will serve us for comparing between restaurants to get the most and least popular restaurants.

A number of methods were implemented in this classes.

- **orderHistory :** takes the system as an argument and returns the list of orders that were passed in this restaurant,
- **getPrice :** takes as argument the order (and not the customer since it is already an attribute of the order, the UML Diagram contains a typo), and returns its prices. The actual implementation is done in the FidelityCard class, this methods only calls the already implemented method. We made the choice to re-implement it here since it makes sense to have the restaurant give the price in the actual usage of the app later,
- **addDish/addMeal :** takes a Meal/Dish and adds it to the restaurant's menu,
- **removeDish/removeMeal :** takes a Meal/Dish and removes it from the menu. Look up the implementation from the food package for the exception thrown,
- **getDishByName/getMealByName :** takes the name of a Dish/Meal and returns the corresponding object. These methods were implemented to make it easier for the user to interact with meals and dishes by only referring to them by their names (for a customer to add one to an order or for a restaurant to add one to their menu for example),
- **incrementOrderCounter :** to increment the orderCounter.

I implemented two constructors, each one calling for one of the constructors of user. They both create an empty menu : the restaurant can not create their account all while creating their menu, it makes more sense to have them add bit by bit items to it (through the methods implemented). This justifies the fact that both constructors do not take a menu as argument. The location tho is present in both cases.

1.2.4 RestaurantComparator and RestaurantSorter

They are two classes I implemented to compare between the restaurants based on their orderCounter.

- **RestaurantComparator :** a class implementing the Comparator<Restaurant> interface containing the compare method. This method returns an integer whose sign represents who is better/more popular.
- **RestaurantSorter :** a class containing a single method which is sort. This method sorts an ArrayList of Restaurants based on the RestaurantComparator in a Descending manner.

1.2.5 Person Class

This is an abstract class that inherits from the User class and that regroups all the other types of users. I implemented this class to regroup the characteristics of a person that a Restaurant would not have, such as a surname. It also helps with modularity since if the app wants to support later on more options regarding of people (a payment method or an IBAN for example), it could directly be implemented here.
Its only attributes are the surname and a static Set<String> that contains the phoneNumbers used. Indeed, customers and couriers, which are implemented as subclasses of Person, both use phone numbers and therefore we need to ensure that those phone numbers are unique for each account, even across the Courier-Customer classes. That is why the repository for the phone numbers is implemented in the Person class their direct parent class.
Same as the restaurant class, the constructors here (two of them) each take either all the information and assigns them to the corresponding attribute, or only take a password and fills the rest with user+id.

1.2.6 Courier Class

This class inherits from the Person abstract class. It takes as attributes the position of the courier (a Location object), a unique phoneNumber (String), a deliveryCounter (int) that helps comparing between couriers and a boolean representing whether the courier is on or off duty (able to receive orders).
Apart from the setters/getters, it's implemented the method incrementDeliveryCounter, that does what its name suggests.
The two constructors only differ by the super constructor they call, as they both force the courier to provide a position and a phone number.

Accepting and refusing an order

A courier is also capable to accept/refuse an order through the `acceptOrder` and `refuseOrder` methods, following the bellow logic order:

1. A customer creates an order, add items to it and then ends this order.
2. The order changes its state to "COMPLETED AND WAITING FOR ACCEPTANCE OF A COURIER".
3. The system searches for the available couriers and sort them based on the chosen strategy (by the least occupied courier or the one that minimizes the total distance that needs to be covered), returning a list of courier sorted by a preference order.
4. The system adds this order to the first courier of the list as an "pending order" that needs to be accepted or refused. For the sake of simplification of the system, it's not possible to create e new order while the pending order is not resolved.
5. The current user needs to logout and the mentioned courier needs to login to accept or refuse the order. When the courier logs into the system, he must make this decision before choosing any other command of the system.
6. If the courier accepts the order, the order change its status to "ACCEPTED AND DELIVERING", the couriers change his status to off duty and there's no pending order anymore in the system. The list of possible couriers to deliver the order is cleared. Accepting a order makes the courier refuses all other pending orders (once that it's possible only have one pending order in the system, there's gonna be only order to accept or refuse each time, but this behavior was chosen in order to facilitate further implementations).

7. If the courier refuses the order, the next courier of the list receives it as an "pending order" and he needs to accept or refuse the order, and so on.
8. If all available couriers refuses the order, the order changes its status to "INCOMPLETE, NO COURIER FOUND" and there's no pending order anymore in the system.

1.2.7 CourierComparator and CourierSorter

They are two classes I implemented to compare between couriers based on their orderCounter.

- **CourierComparator :** a class implementing the Comparator<Courier> interface containing the compare method. This method returns an integer whose sign represents who is better/more active.
- **CourierSorter :** a class containing a single method which is sort. This method sorts an ArrayList of Couriers based on the CourierComparator in a Descending manner.

1.2.8 Customer Class

This class inherits from the Person abstract class. It takes as attributes the address of the customer (a Location object), a unique phoneNumber (String), a fidelityCard (FidelityCard object), a boolean representing whether the customer expressed consent to notifications, a unique email (String) thath is enforced as unique thanks to a static Set<String> of emailsUsed, a currentOrder (Order object) that represents the unfinished order currently being made through the CLI, and a notification (String) that stores the notifications about MealOfTheWeek-s. The only method implemented apart from the setters/getters would be the getHistory() that takes as an argument the MyFoodora system and that returns a HashSet<Order> of all the orders this courier has delivered.

1.2.9 Manager Class

This class represents the Manager, it inherits from Person and does not add any class-specific attributes of its own. In counterpart, this class implements a number of methods, as instructed in the subject of this project. The Manager interacts with the system and should be able to call on a number of statistics regarding the system and even make changes in it. Here are its methods :

- addUser/removeUser,
- setServiceFee,
- setDeliveryCost,
- setMarkupPercentage,
- getOrders,
- computeTotalIncome,
- computeTotalProfit,
- getActiveCustomers,
- computeAverageProfitPerCustomer,
- mostSellingRestaurant,
- leastSellingRestaurant,
- mostActiveCourier,
- leastActiveCourier,
- setDeliveryStrategy,
- getProfitData.

1.3 Notification Package

Developped by Adib Aymane

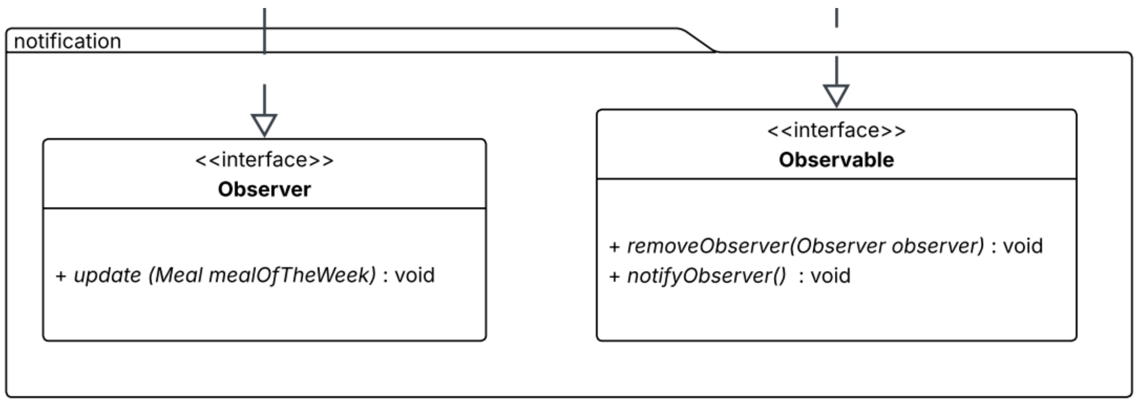

notification
<<interface>>
Observer
+ update (Meal mealOfTheWeek) : void
<<interface>>
Observable
+ removeObserver(Observer observer) : void
+ notifyObserver() : void

Figure 4: UML Diagram for the Notification Package

This notification package serves as an implementation of the Observer Pattern. We decided to implement the Observer and Observable class ourselves instead of using the one already implemented in Java since this is a coding project and the goal is to develop the project from scratch.

1.3.1 Observer Class

The interface representing the observer, who will be implemented by the Customer class since he needs to be notified of every new MealOfTheWeek offer if he gives consent to it.
It contains the update method that takes the mealoftheweek as an argument to append to the notification (String) attribute the info about this new MOTW.

1.3.2 Observable Class

The interface representing the observable, who will be implemented by the Meal class since it is meals that are tracked in order to notify the obeservers of special offers.
It contains the removeObserver method to remove one from the list of observers, and a notifyObserver that gets called each time a special offer is set.

1.4 Order Package

Developped by Adib Aymane

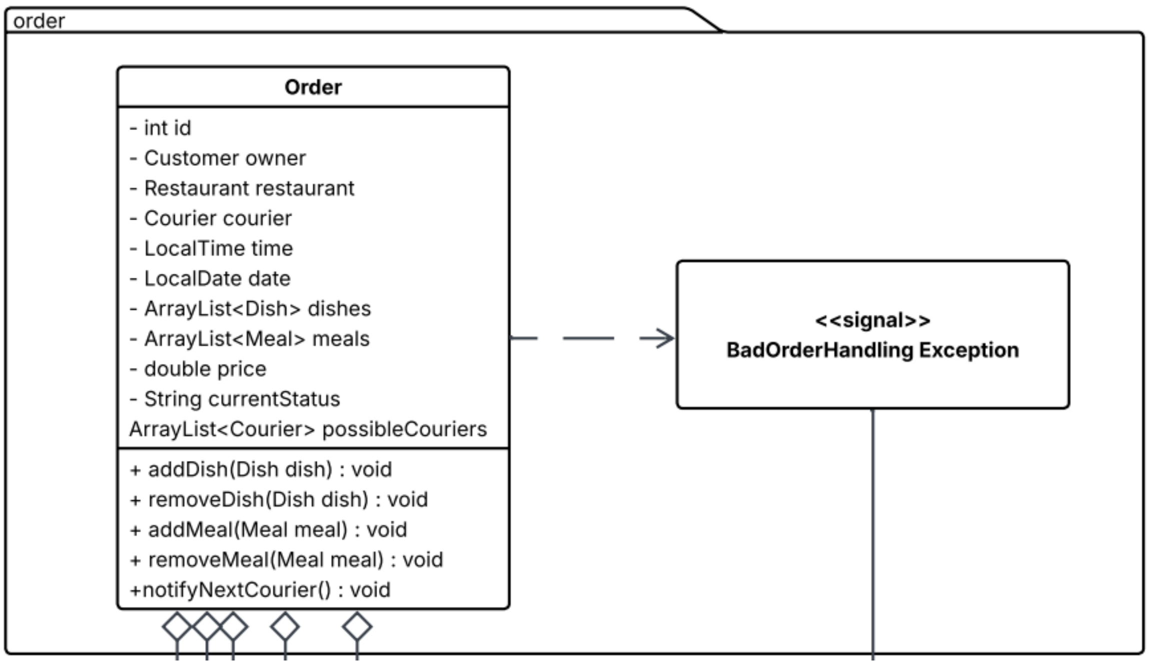

order
Order
- int id
- Customer owner
- Restaurant restaurant
- Courier courier
- LocalTime time
- LocalDate date
- ArrayList<Dish> dishes
- ArrayList<Meal> meals
- double price
- String currentStatus
ArrayList<Courier> possibleCouriers
+ addDish(Dish dish) : void
+ removeDish(Dish dish) : void
+ addMeal(Meal meal) : void
+ removeMeal(Meal meal) : void
+notifyNextCourier() : void
<<signal>>
BadOrderHandling Exception

Figure 5: UML Diagram for the Order Package

We decided to implement this class in its own package since it is at an intersection between the user package (where it takes 3 types of users as an attribute) and the food package (where it has a set of dishes and meals as attributes).

1.4.1 Order Class

An Order object has the following attributes :

- int id : a unique id,
- static int idCounter : a counter that keeps track of the latest id given,
- Customer owner : the customer making the order,
- Restaurant resto : the restaurant preparing the order,
- Courier courier : the courier bringing the order,
- LocalTime time : the time of the order,
- LocalDate date : the date of the order,
- ArrayList<Dish> dishes : the dishes ordered,
- ArrayList<Meal> meals : the meals ordered,
- double price : the price of the ordered. This included the reductions from special offers, but does not include the reduction from fidelityCard. The price as an attribute of Order is **not the final price paid by the customer.**
- ArrayList<Courier> possibleCouriers: ordered list of possibles couriers that can deliver this order.

The methods implemented are the adding and removing of dishes and meals for the customer to interact with the order through the CLI. The order is created with no dishes and meals and no couriers, only after the customer adds the elements he wants, he finalizes the command. The system then finds it a courier.

1.5 Fidelity Package

Developped by Adib Aymane

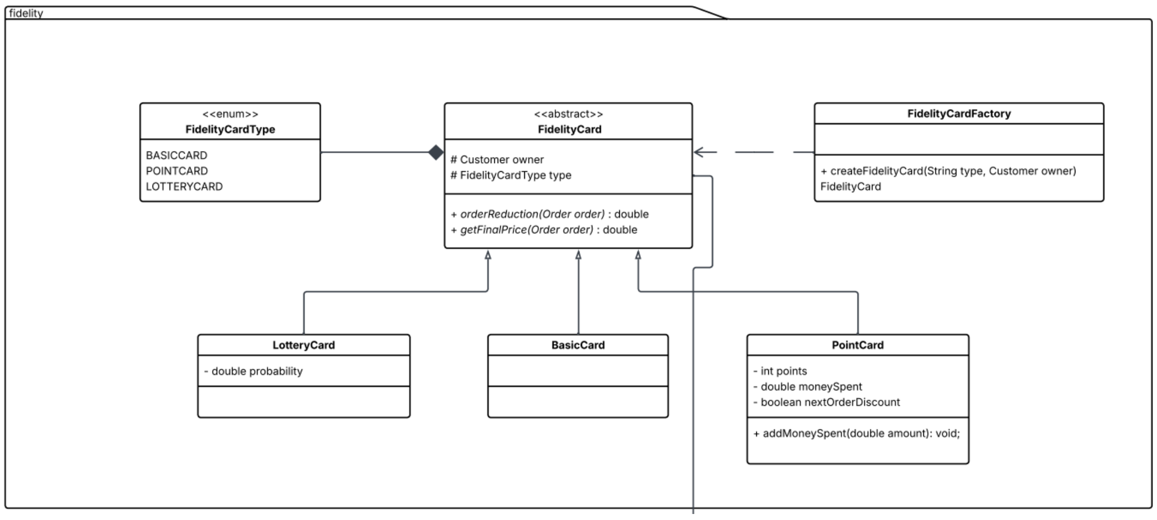


Figure 6: UML Diagram for the Fidelity Package

This package manages the implementation of the fidelity card system in *myFoodora*. Each customer may own one of three different types of fidelity cards that influence the final price of their orders. These cards are assigned and managed through the manager and customer classes, and pricing is handled using polymorphism via the `FidelityCard` interface and its implementations. The design follows the Open/Closed Principle to allow easy extension of new card types without modifying existing code.

1.5.1 FidelityCard Class

The `FidelityCard` class is an `abstract` class that defines the common structure and behavior of all fidelity card types. It contains a method:

- `getFinalPrice(double rawPrice)`: computes the final price to be paid after applying the card-specific discount or rule.

This method is abstract and must be overridden in each subclass to apply the appropriate pricing policy.

1.5.2 FidelityCardType Enum

This `enum` defines the three types of cards supported in the app: `BASIC`, `POINT`, and `LOTTERY`. It is used in the `FidelityCardFactory` to instantiate the correct card based on user input or system logic. It allows decoupling the card creation from the client code, respecting the Single Responsibility Principle.

1.5.3 FidelityCardFactory Class

This class uses the Factory design pattern to instantiate fidelity cards. It exposes one method:

- `createCard(FidelityCardType type)`: returns an instance of the appropriate card subtype based on the enum value provided.

This ensures the card creation logic is centralized, clean, and easily maintainable.

1.5.4 BasicCard Class

This is the default fidelity card automatically associated with any new customer. It allows access to special offers set by restaurants (e.g. Meal of the Week), but does not apply any additional discounts or point mechanisms. The `getFinalPrice` method simply returns the price provided, unchanged.

1.5.5 LotteryCard Class

This card introduces a probabilistic element. At the moment of order finalization, the system simulates a lottery where the user has a chance to get the order for free. The probability is configurable through a probaiblity attribute (double). Only a manager will have access to change that later on in the CLI. In our current implementation, the outcome is determined at system startup for simplicity. If the user "wins", the final price is 0; otherwise, the base price is returned.

1.5.6 PointCard Class

This card allows customers to accumulate points as they make purchases. For every 10 euros spent, one point is earned. Upon reaching 100 points, the user automatically gets a 10% discount on their next order, and points are reset. The internal logic tracks accumulated points and applies discounts conditionally based on the counter.
To represent this we use as attributes the points as int, the moneySpent as a double and a boolean to track whether the next order will be discounted. This has been done in order to ensure that the points are correctly calculated.
The method addMoneySpent gets called each time the user passes an order, to compute the new point tally he has.

This design makes it easy to add new types of fidelity programs in the future while respecting the principles of polymorphism and encapsulation. Pricing logic is entirely decoupled from the user class and isolated within the fidelity package. If a new type of fidelitycard is added with its own rule of calculating the final price, one simply needs to develop the class with its getPrice method. Since in the app the pricing method is strongly linked to the fidelity card the customer uses, a Strategy or Visitor Pattern was deemed unnecessary by this developer, and a simple overriding of the method getPrice provides the modularity needed in this particular setting.

1.6 System Package

Developped by Bonatto Alisson

The system package is responsible for implementing the core where it contains all functionalities of the system (login, logout, register an user, make an order, etc.) and where all date are stored (users, orders, etc.). Figure X shows the UML class diagram for the system package.

1.6.1 ProfitData Class

This class concatenates the profit information of the system: the markup percentage, the service fee and the delivery cost. It contains two constructors, one initializing each one of the attributes via its arguments, and another one that initializes all data with zero (to further initialization via setters). Even tough this class doesn't provides any particular method besides the getters and setters, it is particularly useful to concatenate the profit data of the system.

1.6.2 MyFoodora Class

The MyFoodora class is the principal class of the system package. It is a singleton class, which means that its constructor is private and its instantiation is provided via a static method `getInstance()` which return an instance of the MyFoodora class and save it in the static attribute `instance`. This design pattern ensures that only one instance of this class can be created - when used in the interface, for example. This class contains several attributes, such as: a set of customers, a set of managers, a set of restaurants, a set of couriers, a hash map linking the username of an user with the respective user, the current user logged into the system, a set of orders, the profit data (instance of the class ProfitData), a delivery policy, a profit policy, and a dish factory, a meal factory and a user factory.
In addition to some getters and setters, this class has several methods to implement different functionalities, such as:

- **addUser(User user)**: adds a User to the system.

- **removeUser(User user)**; removes a user from the system. Throws `UserNotFoundException` if user does not exist.
- **registerUser(String userType, String name, String lastName, String username, String adress, String password, String phoneNumber)**: register a new user in the system. This method uses the user factory.
- **login(String username, String password)**: logs a user into the system.
- **logout()**: Logs out the currently logged-in user.
- **createDish(String dishType, String name, double price, boolean isVegetarian, boolean isGlutenFree)**: Creates a new dish using the dish factory. Can throw `BadNumberOfArgumentsException`, `BadDishTypeCreationException` and `BadArgumentTypeException` as described in Subsection 1.1.
- **createMeal(String mealType, String name, Set<Dish> dishes, PricingMealStrategy pricingStrategy)**: creates a new meal using the meal factory. Can throw `UnrecognizedDishException`, `BadMealFormulaException`, `BadMealTypeCreationException`, `BadNumberOfArgumentsException`, `BadArgumentTypeException` as described in Subsection 1.1.
- **selectCourier(Restaurant restaurantToPickDeliver, Customer customerToDeliver)**: Selects a courier to deliver an order based on current delivery police. This design pattern allows to chose between different strategies to select a courier during the exection of the system. Throws `AvailableCourierNotFoundException` if no courier is available
- **sortDishesDeliveryFrequency(Set<Dish> dishes)**: sorts list of dishes based on their delivery frequency using the dish comparator.
- **sortMealsDeliveryFrequency(Set<Meal> meals)**: sorts list of meals based on their delivery frequency using the meal comparator.
- **notifyUser(Restaurant restaurant)**: Notifies the current user with the meal of the week, if they gave consent.
- **createOrder(Restaurant restaurant, Customer customer)**: Creates a new order for the specified restaurant and customer. Initializes the order with no dishes or meals but the order is stored as the current order of the customer.
- **makeOrder(Customer customer, Restaurant restaurant, HashSet<Dish> dishes, HashSet<Meal> meals)**: Creates and processes a new order for the specified customer and restaurant, including the selected dishes and meals. A courier is automatically assigned based on the current delivery strategy. The order is only processed if the current user is an active customer and there is at least one dish or meal. Throws `AvailableCourierNotFoundException` if no available courier can be assigned to the order.
- **updateProfitDataFromTargetProfit(double targetProfit)**: updates the platform's profit data to meet a target profit over the last month. It uses the current profit policy (strategy design pattern) of the system.

1.6.3 ProfitStrategy interface and its Strategy Classes

The ProfitStrategy interface is responsible for ensure the implementation of the method `getProfitData (ProfitData profitData, Set<Order> lastMonthOrders, double targetProfit)` which returns the profit data adjusted to meet a target profit and it uses the last month orders to computes an average profit in order to compute the desired factor using the formula: $profitForOneOrder = orderPrice{\cdot}markupPercentage + serviceFeedeliveryCost$. The manager can chose between different policies (i.e. strategy classes) to adjust the parameters:

- **TargetProfitDeliveryCostOriented** Class: Calculates new profit data by adjusting the delivery cost to meet the target profit.
- **TargetProfitMarkupPercentageOriented** Class: Calculates new profit data by adjusting the markup percentage to meet the target profit.

- **TargetProfitServiceFeeOriented** Class: Calculates new profit data by adjusting the service fee to meet the target profit.

As already mentioned, this design pattern allows the manager to chose between different ways of computing the profit data to meet a profit target during run time.

1.6.4 DeliveryStrategy interface and its Strategy Classes

The DeliveryStrategy interface is responsible for ensure the implementation of the method `selectCourier (Set<Courier> couriers, Restaurant restaurant, Customer customer)` which returns the selected courier based on a chosen delivery policy (strategy classes). The delivery policies are:

- **FairOccupationDelivery** Class: Selects the least occupied courier who is on duty from the provided set of couriers. The sort of couriers is done via the courier comparator.
- **FastestDelivery**: Selects the courier with the minimum total delivery distance (courier to restaurant plus restaurant to customer) among those who are on duty.

2 Core - Testing

In order to test the developed Core of our application we developed multiple JUnit tests going through each of our methods and classes. As advised by Mr. Lapitre in the Lab classes, the testing of each package was done by the Developer that did not work on it. All the following tests were grouped in a single package, called "test". The total number of tests was done was 129.

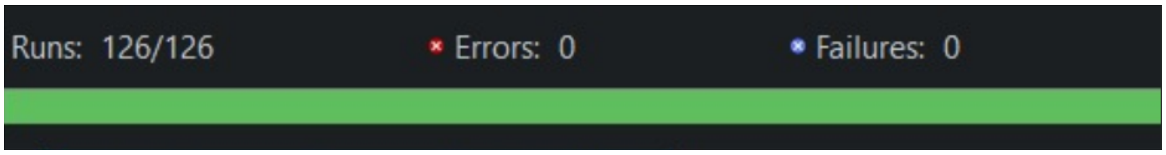

Runs: 126/126 Errors: 0 Failures: 0

Figure 7: JUnit Testing : 126 successful tests

Here were the test cases :

2.1 Food Package

Developped by Adib Aymane

These tests were done in 3 different classes, regrouping the tests of a class and its subclasses together.

2.1.1 TestDish

This class contains 15 different tests. We start by testing the creating of the different instances of Dish (Dessert, MainDish and Starter) and testing their attributes to check if they were correctly initialised.
We then try updating the price we put on a given dish, first with a valid price and then with a negative value. The expected exception (IllegalArgumentException) was raised.
We continue with updating tests with the update of the the isVegetarian, isGlutenFree and the name.
The overrident equals method for dishes is then tested by creating two instances of the same dish (of course with 2 different new Desssert(args...)) and we assert the equality of those dishes. A comparaison test is also done to check which dish has been ordered the most. And for that test to be reliable we performed a test on the getter of the frequency delivery.
Finally, we made the dishf factory go through various tests ; from real instances of dish creation to defectious ones with either not the good number of arguments, unknown type of dish type or with bad arguments. Each time the expected exception was thrown.

2.1.2 TestMeal

This class contains 9 tests. For this, we start by a BeforeClass (the project is on JUnit 4) where we start by creating dishes and meals we use in tests. We also define a tearDown method (through an AfterClass) to make them all go null to avoid eventual conflicts with other tests.
Same as before, we start by checking the creation of FullMeals and HalfMeals along with their attributes. We also try to create a meal with the wrong formula (a half meal with 3 dishes) and obtain the corresponding exception.
We test the meal factory by using a correct setting of dishes, then we test an nonexistent type of meal and a bad formula.
We also tested changing the pricing, whether the meal is MealOfTheWeek or not, but also changing the different factors that go into these pricing strategies.
Finally, we successfully test the overridden equals method.

2.1.3 TestMenu

The TestMenu class contains 17 tests. We begin by testing the menu's creation and then the adding of dishes and meals to it. We also test the removal of those items from the menu.

For the removals, we test removing non existant items (meals and dishes) and we get the relevant exceptions from those tests.
We test the getter of meals and dishes, and also the ones related to the discounts. For their setters, we test negative discount and discounts above 1, both successfully.

2.2 Order Package

Developped by Bonatto Alisson
The TestOrder class contains 8 tests. We tested the addition and the removal of dishes and meals to the order, in addition to the situations where the Order class can throw an exception when we are trying to remove a meal or a dish that is not present in the order. We also tested the price computation of the order: it must be the sum of the price of each dish plus the price of the meal.

2.3 User Package

Developped by Bonatto Alisson
For testing the user package, one test class for each type of user was created.
The TestCustomer class contains 12 tests. It contains tests for the creation of a customer using the Customer class constructor and the UserFactory as well. It also contains different tests for bad user creation, i.e. when someone tries to create an user with an already used username, or already used email, or already used phone number, analysing the expected thrown exception. The notification system in the notification package was also tested: a meal of the week is created, then we check the notifications string of the consented notification user.
The TestRestaurant class contains 9 tests. The creation of a restaurant is tested via its constructor and via the UserFactory. It also contains different tests for bad user creation, i.e. when someone tries to create an user with an already used username. The addition and removal of dishes and meals are also tested. Through the usage of different fidelity cards of a customer, this class tests the `getPrice (Order order, Customer customer)` method.
The TestManager class contains 13 tests. Besides the Manager creation and its possible exceptions, this class tests the manager functionalities, such as: adding and removing users from the system, the sorting of restaurants by its order counters, the sorting of couriers by their delivery counters, and the computation of total income and profit.
the TestCourier class contains 7 tests. Similarly we test a courier creation with and without exceptions, in addition to tests related to the delivery counter of the courier.

2.4 Fidelity Package

Developped by Bonatto Alisson
The TestFidelityCards contains 10 tests, testing the creation of different fidelity cards (basic card, lottery card and point card), and the get price of each type of card. The FidelityCardFactory was also tested

2.5 System Package

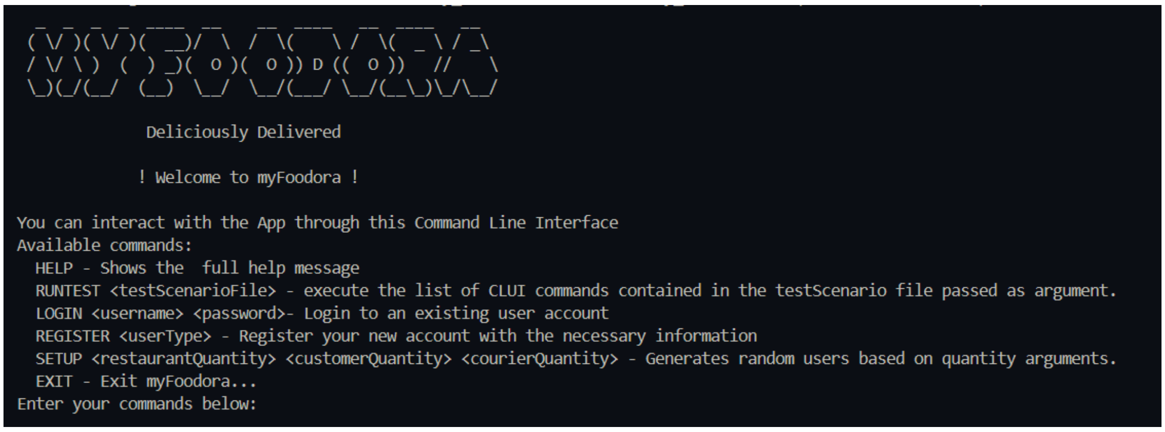
Developped by Adib Aymane

All the system package was tested in a single class (the TestSystem) since it was all strongly linked. It contains 25 tests. This developer realizes that the package could lead to many more tests, but due to the lack of time, elaborate test functions for methods such as getProfitData and others were dismissed.
The setup beforeclass was done by initializing the system and creating its different users while also creating 3 dishes to manipulate throughout the tests.
We start by testing the instance of system existing. We test if the users are existing within the system as well.
We test the getters for all the lists of users as well, and other attributes of the system.
We test the removal of some users, while of course testing the removal of nonexistent users.
We test the login/logout routine, while testing to login with nonexistent accounts or with a wrong password.

We are also testing the creation of dishes and meals through the system, with all the exceptions possible surrounding that.
We test the selection of couriers through different selection strategies, but also testing the making of orders (correct and invalid ones). This involves ones with no courier available, without being logged in or empty orders.
We make an AfterClass to tear down the static attributes we created.

3 Command Line Interface

3.1 General Description



```
  _  _  _  _  ____  __    __  ____   __  ____   __
 ( \/ )( \/ )(  __)/  \  /  \(    \ /  \(  _ \ / _\
 / \/ \ )  (  ) _)(  O )(  O )) D ((  O ))   //    \
 \_)(_/(__/  (__)  \__/  \__/(____/ \__/(__\_)\_/\_/

            Deliciously Delivered

           ! Welcome to myFoodora !

You can interact with the App through this Command Line Interface
Available commands:
  HELP - Shows the  full help message
  RUNTEST <testScenarioFile> - execute the list of CLUI commands contained in the testScenario file passed as argument.
  LOGIN <username> <password>- Login to an existing user account
  REGISTER <userType> - Register your new account with the necessary information
  SETUP <restaurantQuantity> <customerQuantity> <courierQuantity> - Generates random users based on quantity arguments.
  EXIT - Exit myFoodora...
Enter your commands below:
```

Figure 8: Welcome message - Terminal

As instructed, the group developed a CLI to interact with the app. We developed the main function that displays a welcome message (look above) and initializes the system through a my_foodora.ini file you can find in the eval folder. Through this inisialization we create 2 managers, 2 customers, 2 restaurants and 2 couriers. This ensures that the app is never empty for a user. The main method then takes your input from the terminal, splits it by spaces and sends those commands to the handleCommand method. The relevant commands are available by typing help, which shows you the **Common Commands** that can be executed by any user, and the **Class-specific Commands** of the logged in user's class. Here is a description of what they are:

3.2 Common Commands

Developed by Bonatto Alisson

3.3 Restaurant-specific Commands

Developed by Adib Aymane

Here are the commands available :

- **CREATEDISH** <dishType> <dishName> <unitPrice> <isVege [y/n]> <glutenFree [y/n]> - Add a dish to the restaurant's menu.
- **CREATEMEAL** <mealType> <mealName> <dish1Name> <dish2Name> [<dish3Name>] - Create a meal with specified dishes and add it to your menu.
- **SHOWMEAL** <mealName> - Show details of a specific meal from your menu.
- **SHOWDISH** <dishName> - Show details of a specific dish from your menu.
- **REMOVEMEAL** <mealName> - Remove a specific meal from your menu.
- **REMOVEDISH** <dishName> - Remove a specific dish from your menu.
- **SETSPECIALOFFER** <mealName> - Set a special offer for a meal.
- **REMOVEFROMSPECIALOFFER** <mealName> - Remove a meal from the special offer.

- **SHOWMENUITEMS** - Show details of your menu.
- **SETPRICE** <dishName> <newPrice> - Changing the price of an existing dish in your menu.
- **SETGENERICDISCOUNTFACTOR** <discountFactor> - Set the generic discount factor for your restaurant.
- **SETSPECIALDISCONTFACTOR** <discountFactor> - Set the special discount factor for your restaurant.

3.4 Customer-specific Commands

Developed by Adib Aymane

Here are the commands available :

- **CREATEORDER** <restaurantName> - Create a new order with the specified restaurant.
- **CREATEORDER** <restaurantName> - Create a new order with the specified restaurant.
- **ADDITEM2ORDER** <itemType> <itemName> - Add an item to an existing order.
- **ENDORDER** - End the current order, finalizing it and processing it.
- **ASSOCIATECARD** <cardNumber> - Associate a fidelity card with your account.
- **SHOWMENUITEMS** <restaurantName> - Show details of a specific restaurant's menu.
- **SHOWRESTAURANTS** - Show a list of all available restaurants.
- **SHOWORDERS** [<restaurantName>] - Show a list of your past orders, can be to a specific restaurant.
- **SHOWPOPULARRESTAURANTS** - Show a list of the most popular restaurants.
- **SHOWMONEYSPENT** [<restaurantName>] - Show the total amount of money spent on orders, can be to a specific restaurant.
- **CHANGEADDRESS** <x> <y> - Change your delivery address.
- **CHANGEPHONENUMBER** <newPhoneNumber> - Change your phone number.
- **CONSENTNOTIFICATIONS** <yes/no> - Set your consent for receiving notifications.
- **DISPLAYFIDELITYCARD** - Display your fidelity card information.

3.5 Courier-specific Commands

Developed by Adib Aymane

Here are the commands available :

- **ONDUTY** - Mark yourself as on duty to accept orders.
- **OFFDUTY** - Mark yourself as off duty to stop accepting new orders.

3.6 Manager-specific Commands

Developed by Bonatto Alisson

4 Starting the App - Guide

This Chapter has the objective of give a guide to the execution of the system. The project was made using Eclipse and VS Code IDE's organizing the files as stipulated in the project description.

4.1 Downloading and executing the system

You can download the project through the GitHub repository [1], executing the following command in the dependency where you want to clone the project:

```
git clone https://github.com/aymaneadib/my-Foodora.git
```

The **main function is in the `CLI.java` class of `cli` package**, inside the `src` folder, as shown in Figure 9. You can execute it directly from a IDE like Eclipse or through the following command:

```
cd cs.fr.groupB.myFoodora
javac -d bin -sourcepath src src/cli/CLI.java
java src/cli/CLI.java
```

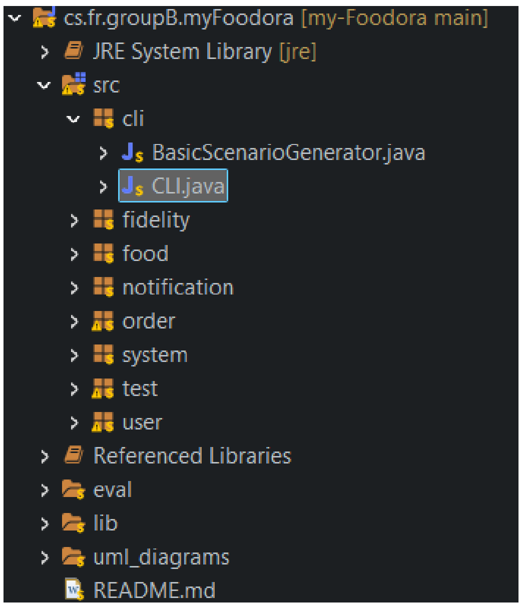


Figure 9: Folder tree of the project, highlighting the CLI.java file

And it's done! Now you can use the terminal interface of the MyFoodora system. You can type `HELP` to have more information about the available commands.

4.2 Running a text file scenario of commands

You can, instead of running the commands one by one in the terminal, put all the commands to be executed in a `.txt` file and execute it via the command `runTest <file_path.txt>`. The provided test scenarios are located in the `eval` folder. So, to execute the `./eval/test_scenario_1.txt` file, you must run the following command during the execution of the interface:

```
> runTest ./eval/test_scenario_1.txt
```

Be careful to use the correct path to the file to be executed, because the relative path to the test scenario text file depends on where you executed the interface. If you're executing the CLI.java file from the root of the project like described in Subsection 4.1 or through Eclipse, the correct path to the test scenario number one is indeed `./eval/test_scenario_1.txt`.

An useful feature is that you can add comments that will not be executed in these text files beginning a line with the "//" characters. If you add empty lines to the document they will be ignored as well.

4.3 Configuring the initialization file

At the beginning of the execution of the interface, the system loads in its memory the content of the file `./eval/my_foodora.ini`. As ant `.ini` file, each entity is divided into sections following a list of key-value pairs. Here each section is designed to represents a user to be loaded into the system, thus the possible sections are: Manager, Customer, Courier, Restaurant. You can modify this file to add as many user as you want, but you have to keep the correct sequence of the pairs key-value as indicated at the beginning of the file through comments.

4.4 Adding random users through SETUP command

Another feature that affects directly the initialization of the system is the command `SETUP <nRestaurants> <nCustomers> <nCouriers>` which generates the specified number of restaurants, customer and couriers with random attributes (names, surnames, phone number, username, menu, etc.) and also a Manager which the username is `admin` and the password is `admin`. Through the admin user you can display the list of all other users to see the results of the initialization. It's ensured that the menu of each restaurant is filled with at least three dishes (one of each type) an one meal.

5 Test Scenarios

As described previously, the system is also capable of executing a pipeline of commands written in a text file, line per line, through the command `runTest <file_path.txt>`. The objective of this chapter is to describe the test scenarios created and the obtained results.

5.1 Scenario 1

This scenario is designed to show the possible ways of initializing the system: through the standard initialization with the `my_foodora.ini` file but also with the command `SETUP`.

5.1.1 Objectives

The objective of this test scenario is to ensure that the initialization of the system is properly done:

- Verify if all the users defined in `my_foodora.ini` are present in the system.
- Verify if the command `SETUP` is error proof, i.e. is capable of show a message error if the user tries to pass a negative number or a character as argument.
- Verify if the command `SETUP` is indeed creating the number of each user specified in the arguments.
- Additionally, test the registration of users.

5.1.2 Execution of the pipeline and results

In the section, some parts of the execution of the pipeline and the obtained resulted will be demonstrated and explained.

The first set of instructions are related to the verification of the users created via `my_foodora.ini` file:

```
> runtest ./eval/test_scenario_1.txt
EXECUTING TEST FILE ./eval/test_scenario_1.txt ...

> login alissonbonatto bonatto1234
(alissonbonatto) User alissonbonatto logged in.

> SHOWCUSTOMERS
Nikola Tesla <nikolatesla@email.com> ( nikolatesla - Customer )
Alan Turing <alanturing@email.com> ( alanturing - Customer )

> SHOWRESTAURANTTOP
ChezJacques with 0 orders.
PetitParis with 0 orders.

> SHOWCOURIERDELIVERIES
Ada Lovelace ( adalovelace - Courrier with 0 deliveries )
Isaac Newton ( isaacnewton - Courrier with 0 deliveries )

> logout
(alissonbonatto) User alissonbonatto logged out.

> login aymaneadib adib1234
(aymaneadib) User aymaneadib logged in.

> logout
(aymaneadib) User aymaneadib logged out.
```

We can see that all users created in the initialization file are uploaded into the system. Next the test scenario executes the `SETUP` command, first with bad arguments showing message errors, then with a correct set of arguments.

```
> SETUP a 2 3
Error: invalid number type.
```

```
Usage: SETUP <restaurantQuantity> <customerQuantity> <courierQuantity> - Generates random users
    based on quantity arguments.

> SETUP 5 -8 5
Error: Select quantities between 1 and 100.

> SETUP 3 4 5
Created 3 restaurants, 4 customers and 5 couriers.
```

The system correct verified that the character "a" is an invalid input, as well the value -8. Then the next commands are executed in order to see if the system indeed created the specified number of users:

```
> login admin incorrectPassword
Error: Username and password do not match.

> login admin admin
(admin) User admin logged in.

> SHOWCUSTOMERS
customerName3 customerSurname3 <customerUsername3@email.com> ( customerUsername3 - Customer )
customerName0 customerSurname0 <customerUsername0@email.com> ( customerUsername0 - Customer )
customerName2 customerSurname2 <customerUsername2@email.com> ( customerUsername2 - Customer )
customerName1 customerSurname1 <customerUsername1@email.com> ( customerUsername1 - Customer )

> SHOWRESTAURANTTOP
restaurantName0 with 0 orders.
restaurantName1 with 0 orders.
restaurantName2 with 0 orders.

> SHOWCOURIERDELIVERIES
courierName1 courierSurname1 ( courierUsername1 - Courrier with 0 deliveries )
courierName4 courierSurname4 ( courierUsername4 - Courrier with 0 deliveries )
courierName2 courierSurname2 ( courierUsername2 - Courrier with 0 deliveries )
courierName3 courierSurname3 ( courierUsername3 - Courrier with 0 deliveries )
courierName0 courierSurname0 ( courierUsername0 - Courrier with 0 deliveries )

> logout
(admin) User admin logged out.
```

The command has correctly created 3 restaurants, 4 customers and 5 couriers, in addition to the manager admin that is created by default. The next commands have the goal of showing that the menu of restaurants are filled with some dishes and meals. To do so, one created restaurant logs into the system and shows its menu:

```
> login restaurantUsername1 restaurantUsername1
(restaurantUsername1) User restaurantUsername1 logged in.

> SHOWMENUITEMS
------------ STARTERS -----------
Starter standardSTARTER - 103.07€
Starter dishName29restaurantName1 - 189.06€
// ...
---------- MAIN DISHES ----------
Main Dish dishName32restaurantName1 - 15.54€
Main Dish dishName18restaurantName1 - 266.86€
Main Dish standardMAINDISH - 139.46€
// ...
------------ DESSERTS -----------
Dessert dishName13restaurantName1 - 151.62€
Dessert standardDESSERT - 21.20€
// ...
-------------- MEALS ------------
Full Meal meal2restaurantName1 - 771.28€ - Composed of: dishName4restaurantName1,
    dishName16restaurantName1, dishName2restaurantName1.
Half Meal meal7restaurantName1 - 34.90€ - Composed of: dishName32restaurantName1, standardDESSERT.
// ...
> logout
(restaurantUsername1) User restaurantUsername1 logged out.
```

In the above piece of execution, a part of the prompt result was suppressed in this document ("//..." lines) for the sake of a good visualization. We can see that dishes and meals were added into restaurant's menu. Random names and prices are chosen, with the exception of the standardSTARTER, standardMAINDISH and standardDESSERT, that are hard coded dishes to ensure that at least 1 dish of type is created.

After showing the menu of another restaurant, the test scenario tests the registration of a new user and the possible errors:

```
> REGISTER CUSTOMER
Enter the Customer information.
Usage: <name> <surname> <username> <password> <phoneNumber> <email> <addresX> <addresY>
    <consentNotifications yes/no>

> Carl Sagan carlsagan sagan1234 +3312456789 sagan@email.com 12.5 -7
User carlsagan registered successfully!
To use the new user, you need to login.

> REGISTER CUSTOMER
Enter the Customer information.
Usage: <name> <surname> <username> <password> <phoneNumber> <email> <addresX> <addresY>
    <consentNotifications yes/no>

> NewUserName NewUserSurname carlsagan password +phoneNumber newUser@email.com 0 0
Error: Username already used: carlsagan

> REGISTER CUSTOMER
Enter the Customer information.
Usage: <name> <surname> <username> <password> <phoneNumber> <email> <addresX> <addresY>
    <consentNotifications yes/no>

> NewUserName NewUserSurname newUsername password +phoneNumber sagan@email.com 0 0
Error: Email already used by another account: sagan@email.com

> REGISTER CUSTOMER
Enter the Customer information.
Usage: <name> <surname> <username> <password> <phoneNumber> <email> <addresX> <addresY>
    <consentNotifications yes/no>

> NewUserName NewUserSurname newUsername password +3312456789 newUser@email.com 0 0
Error: Phone number already used by another account: +3312456789

END OF EXECUTION OF ./eval/test_scenario_1.txt TEST FILE.
```

As we can see, the system successfully register the Customer Carl Sagan. Then, a new customer with same username tries to be registered resulting in an error. The same happens when a customer with the same email and phone number tries to be created: an error, because the username, emails and phone numbers must be unique.

With this test scenario we can show that the: system is properly initiated; it's possible to create random users; it's possible to register an user and all this features are error proof.

5.2 Scenario 2

This scenario is designed to evaluate the system's behavior in abnormal, erroneous, or uncommon use cases. It includes around 100 commands and covers a wide range of functionalities such as order management, user interaction, and policy configuration. The purpose is to assess the robustness of the platform under adverse or unintended conditions.

5.2.1 Objectives

The main objectives of this test scenario are:

- To verify that the system handles invalid commands or inputs gracefully (e.g., setting a negative discount, malformed phone numbers).
- To ensure that dish and meal removal correctly impacts menu visibility and availability.

- To test order assignment logic in scenarios where all couriers initially reject the order.
- To assess the delivery system's ability to assign remaining pending orders once a new eligible courier becomes available.
- To confirm the system respects the state of couriers (on duty / off duty) during order acceptance.

5.2.2 Execution of the pipeline and results

In the section, some parts of the execution of the pipeline and the obtained resulted will be demonstrated and explained.

At first, we ensure that the `help` command displays the corresponding commands to the user :

```
> login aymaneadib adib1234
(aymaneadib) User aymaneadib logged in.

> help
----------------------------------------

Generic Commands Available :
   - HELP - Show this help message
   - RUNTEST <testScenarioFile> - execute the list of CLUI commands contained in the testScenario
     file passed as argument.
   - LOGIN <username> <password> - Log in with the specified username and password.
   - REGISTER <userType> - Register a new user account. User types can be: CUSTOMER, RESTAURANT,
     COURIER.
   - LOGOUT - Log out of the current session.
   - EXIT - Exit myFoodora...

----------------------------------------

Manager Commands Available :
   - SHOWMENUITEMS <restaurantName> - Show details of a specific restaurants menu.
   - ASSOCIATECARD <cardNumber> <customerUsername> - Associate a fidelity card with a customer
     account.
   - REGISTER <userType> - Register a new user account. User types can be: MANAGER, CUSTOMER,
     RESTAURANT, COURIER.
   - SHOWCOURIERDELIVERIES - Display the list of couriers sorted in decreasing order w.r.t. the
     number of completed deliveries.
   - SHOWRESTAURANTTOP - Display list of restaurants sorted in decreasing order w.r.t. the number
     of delivered orders.
   - SHOWCUSTOMERS - Display the list of customers.
   - SHOWTOTALPROFIT <startDate YYYY-MM-DD> <endDate YYYY-MM-DD> - Show the total profit of the
     system. Time interval is optional.
   - SETDELIVERPOLICY <delPolicy> - set the delivery policy of the system : FairOccupationDelivery,
     FastestDelivery.
   - SETPROFITPOLICY <profitPolicy> - set the profit policy of the system : DeliveryCostOriented,
     MarkupPercentageOriented, ServiceFeeOriented.

> logout
(aymaneadib) User aymaneadib logged out.
```

We can see that the `help` command correctly provides only the generic and manager commands as intended. Next, we enter the variables needed for initialization, so that we can produce a richer scenario smoothly. We also test the registring process of all types of users at the same time, and we notice that it all works properly :

```
> register Restaurant
Enter the Restaurant information.
Usage: <name> <username> <password> <addresX> <addresY>

> BurgersAndFries bnf secret 100.5 100.5
User bnf registered successfully!
To use the new user, you need to login.

> register Restaurant
Enter the Restaurant information.
Usage: <name> <username> <password> <addresX> <addresY>
```

```
> TacoTown tacotown taco123 50.0 50.0
User tacotown registered successfully!
To use the new user, you need to login.

> register Customer
Enter the Customer information.
Usage: <name> <surname> <username> <password> <phoneNumber> <email> <addresX> <addresY>
    <consentNotifications yes/no>

> Alice Cooper acooper pass123 +1234 alice@domain.com 120.0 120.0
User acooper registered successfully!
To use the new user, you need to login.

> register Customer
Enter the Customer information.
Usage: <name> <surname> <username> <password> <phoneNumber> <email> <addresX> <addresY>
    <consentNotifications yes/no>

> Bob Marley bmarley oneLove +128834 bob@domain.com 121.0 121.0
User bmarley registered successfully!
To use the new user, you need to login.

> register Courier
Enter the Courier information.
Usage: <name> <surname> <username> <password> <phoneNumber> <addresX> <addresY>

> Courier One c1 pw +555 123.0 123.0
User c1 registered successfully!
To use the new user, you need to login.

> register Courier
Enter the Courier information.
Usage: <name> <surname> <username> <password> <phoneNumber> <addresX> <addresY>

> Courier Two c2 pw +66666 124.0 124.0
User c2 registered successfully!
To use the new user, you need to login.

> register Courier
Enter the Courier information.
Usage: <name> <surname> <username> <password> <phoneNumber> <addresX> <addresY>

> Courier Three c3 pw +77777 125.0 125.0
User c3 registered successfully!
To use the new user, you need to login.
```

We then switch the status of all courriers to `onduty` so that they can deliver orders. We can notice that it worked as intended later since the courriers received the delivery requests.

```
> login c1 pw
(c1) User c1 logged in.

> onduty
(c1) You are now on duty. You can accept new orders.

> logout
(c1) User c1 logged out.

> login c2 pw
(c2) User c2 logged in.

> onduty
(c2) You are now on duty. You can accept new orders.

> logout
(c2) User c2 logged out.

> login c3 pw
(c3) User c3 logged in.
```

```
> onduty
(c3) You are now on duty. You can accept new orders.

> logout
(c3) User c3 logged out.
```

The following commands aims to create new dishes and meals, and we also test if the CLI reacts correctly when arguments of a command don't match its intended usage :

```
> login tacotown taco123
(tacotown) User tacotown logged in.

> createDish mainDish beefTaco 9.0 true true
(tacotown) Dish added successfully to the Menu

> createDish dessert chocoChurros 4.0 false true
(tacotown) Dish added successfully to the Menu

> createDish starter nachos 5.0 true true
(tacotown) Dish added successfully to the Menu

> createMeal fullmeal MealTaco beefTaco nachos chocoChurros
(tacotown) Full meal created successfully.

> setSpecialOffer MealTaco
(tacotown) Special offer set for meal: MealTaco

> setPrice beefTaco 12.0
(tacotown) Price updated successfully for dish: beefTaco

> setGenericDiscountFactor -0.1
(tacotown) Error: Invalid discount factor. Please provide a valid number.

> setGenericDiscountFactor 0.05
(tacotown) Generic discount factor set to: 0.05

> removeDish nonExistentDish
(tacotown) Dish not found in your menu.

> removeFromSpecialOffer NotInOffer
(tacotown) Meal not found in your menu.

> showMenuItems
(tacotown)
------------ STARTERS -----------
Starter nachos - 5,00?
---------- MAIN DISHES ----------
Main Dish beefTaco - 12,00?
------------ DESSERTS -----------
Dessert chocoChurros - 4,00?
-------------- MEALS ------------
Full Meal MealTaco - 18,90? - Composed of: beefTaco, nachos, chocoChurros.


> logout
(tacotown) User tacotown logged out.
```

testing pipeline dial making order nwsef lmarahil verifizw oder dart

Moving on, we test the pipeline of making an order, and ensure that the following commands work properly : associating a fidelity card to the client's account, displaying it, creating an order and adding dishes, displaying the current order, comfirming the order then showing it. We also make sure the the order has been successfully issued.

```
> login acooper pass123
(acooper) User acooper logged in.

> associateCard lottery
```

```
(acooper) Fidelity card associated successfully with your account.

> displayFidelityCard
(acooper) Fidelity Card Details: LotteryCard

> createOrder TacoTown
(acooper) Order created successfully with ID: 1

> addItem2Order dish beefTaco
(acooper) Item added to order successfully.

> addItem2Order meal MealTaco
(acooper) Item added to order successfully.

> displayCurrentOrder
(acooper) ---------- Dishes in your order ----------
(acooper) Main Dish beefTaco - 12,00?
(acooper) ---------- Meals in your order -----------
(acooper) Full Meal MealTaco - 18,90? - Composed of: beefTaco, nachos, chocoChurros.
(acooper) ------------------------------------------
(acooper) Order status: WAINTING FOR COMPLETION

> endOrder
(acooper) Order ended successfully. Order ID: 1
(acooper) We will find a courier for you order.

> showOrders
(acooper) Your Past Orders:
(acooper)  - Order ID: 1, Restaurant: TacoTown

> logout
(acooper) User acooper logged out.
```

We test in the following lines what happens when a courier refuses an order. We can clearly see it transferring over to the next courier with the strategy chosen. If everyone of them refuses, the order is simply refused.

```
> login c1 pw
(c1) User c1 logged in.
(c1) -------- You must accept or refuse the following orders --------
(c1) Order ID 1 from TacoTown to Alice

> refuseOrder 1
(c1) Order ID 1 refused.

> logout
(c1) User c1 logged out.

> login c3 pw
(c3) User c3 logged in.
(c3) -------- You must accept or refuse the following orders --------
(c3) Order ID 1 from TacoTown to Alice

> refuseOrder 1
(c3) Order ID 1 refused.

> logout
(c3) User c3 logged out.

> login adalovelace adalovelace1234
(adalovelace) User adalovelace logged in.
(adalovelace) -------- You must accept or refuse the following orders --------
(adalovelace) Order ID 1 from TacoTown to Alice

> refuseOrder 1
(adalovelace) Order ID 1 refused.

> logout
(adalovelace) User adalovelace logged out.

> login isaacnewton isaacnewton1234
```

```
(isaacnewton) User isaacnewton logged in.
(isaacnewton) -------- You must accept or refuse the following orders --------
(isaacnewton) Order ID 1 from TacoTown to Alice

> refuseOrder 1
(isaacnewton) Order ID 1 refused.

> logout
(isaacnewton) User isaacnewton logged out.

> login c2 pw
(c2) User c2 logged in.
(c2) -------- You must accept or refuse the following orders --------
(c2) Order ID 1 from TacoTown to Alice

> refuseOrder 1
(c2) Order ID 1 refused.

> logout
(c2) User c2 logged out.
```

We can also test accepting an order from a courier, which happens succefully also.

```
> login c3 pw
(c3) User c3 logged in.
(c3) -------- You must accept or refuse the following orders --------
(c3) Order ID 3 from BurgersAndFries to Alice

> acceptOrder 3
(c3) Order ID 3 accepted.

> logout
(c3) User c3 logged out.
```

For the rest of the scenario, you can check it in the eval folder where you will have the results of the scenario_2.

Conclusion

The myFoodora project provided a comprehensive opportunity to apply object-oriented programming principles in the development of a complete and modular software system. Through careful analysis of requirements, systematic design using UML diagrams, and rigorous implementation in Java, we built a fully functional food delivery platform that models realistic user interactions and business logic.

Throughout the project, we ensured that the system was structured to respect the Open/Closed Principle and leveraged relevant design patterns to improve code maintainability and extensibility. The command-line user interface allowed us to test the system effectively, simulate typical usage scenarios, and validate the proper functioning of core features such as user registration, order processing, delivery assignment, and profit computation.

In addition to implementing the required functionalities, we placed significant emphasis on testing through JUnit, ensuring that each component behaved as expected in isolation. This emphasis on modularity and testing not only strengthened the robustness of our solution but also demonstrated good software engineering practices.

As future prospects for the project, it remains to implement a more user-friendly graphical interface. Going further, the system can be integrated with a server where users can actually register and place orders through the system. For all this, many modifications would be necessary, notably the system for assigning deliveries to couriers, which, for this project, was greatly simplified.

Overall, the myFoodora project was a valuable experience in developing a real-world-inspired application from scratch, reinforcing both technical and collaborative skills. The final system is flexible, extensible, and well-documented, laying a solid foundation for future enhancements or integration with more advanced user interfaces.

References

[1] ADIB, Aymane Chaoui, BONATTO, Alisson. (202025). *my-Foodora*. GitHub repository. Link: https://github.com/aymaneadib/my-Foodora