CentraleSupélec

myFoodora - Food Delivery System

Hala CHAFIK
Salma MAAZOUM

03/05/2024 - 09/06/2024

Supervisors :

Mr. Paolo BALLARINI
Mr. Arnault LAPITRE

Contents

1 Introduction **2**

2 myFoodora core **3**

2.1 *Menus and Meals* . . . 3

2.1.1 MenuComponent class . . . 4

2.1.2 FoodItem class . . . 4

2.1.3 Meal abstract class . . . 4

2.1.4 FullMeal and HalfMeal class . . . 5

2.1.5 Menu class . . . 5

2.1.6 AddItemVisitor interface . . . 5

2.2 *Fidelity Cards* . . . 6

2.2.1 FidelityCard abstract class . . . 6

2.2.2 BasicFidelityCard class . . . 6

2.2.3 PointFidelityCard . . . 7

2.2.4 LotteryFidelityCard . . . 7

2.3 *Delivery Policy* . . . 8

2.3.1 Enhancing the Open/Closed Principle . . . 8

2.3.2 Design Limitations . . . 9

2.3.3 DeliveryPolicy Interface . . . 9

2.3.4 FastestDeliveryPolicy class . . . 9

2.3.5 FairOccupationDeliveryPolicy class . . . 10

2.4 *Target Profit Policy* . . . 10

2.5 *Users of myFoodora system* . . . 11

2.5.1 Restaurant . . . 11

2.5.2 Customer . . . 13

2.5.3 Courrier . . . 13

2.5.4 Manager . . . 14

2.6 *Notification Service* . . . 16

2.6.1 NotificationService class . . . 17

2.7 *Order* . . . 18

2.8 *Exceptions* . . . 19

2.9 *AppSystem - myFoodora core class* . . . 19

2.10 *AppSystemCLI - Command Line Interface* . . . 20

3 Test Scenario **22**

4 Strating the app **23**

5 Task Distribution **23**

1 Introduction

This document details the design and implementation of **myFoodora**, a Java-based framework engineered to emulate popular food delivery services, such as Deliveroo and UberEats. The development is split into two main sections: the **1st section** constructs the core infrastructure required to support complex operations within the platform, and the **2nd section** designs a command-line interface that caters to user interactions with myFoodora.

The primary goal of the `myFoodora` project is to establish a robust framework capable of facilitating seamless interactions between customers and restaurants. This includes managing diverse restaurant offerings, processing customer orders, and ensuring efficient delivery systems. The project aims to combine essential functionalities into a cohesive system implemented within a user-friendly interface, enhancing user experience.

2 myFoodora core

The first part of the project involves the design and development of the core Java infrastructure for the `myFoodora` system. This core infrastructure encompasses the following requirements :

2.1 *Menus and Meals*

Restaurants are one of the four basic users of the `myFoodora` system (see following section). They offer dishes to customers, which are listed in a menu. Additionally, these dishes can be grouped into specially priced meal combinations, providing an alternative to ordering individual dishes à la carte.

The classes mentioned in this section were consolidated into a single package named `food`.

The subsequent UML diagram illustrates the organization and distribution of the major constituent classes within the `food` package.

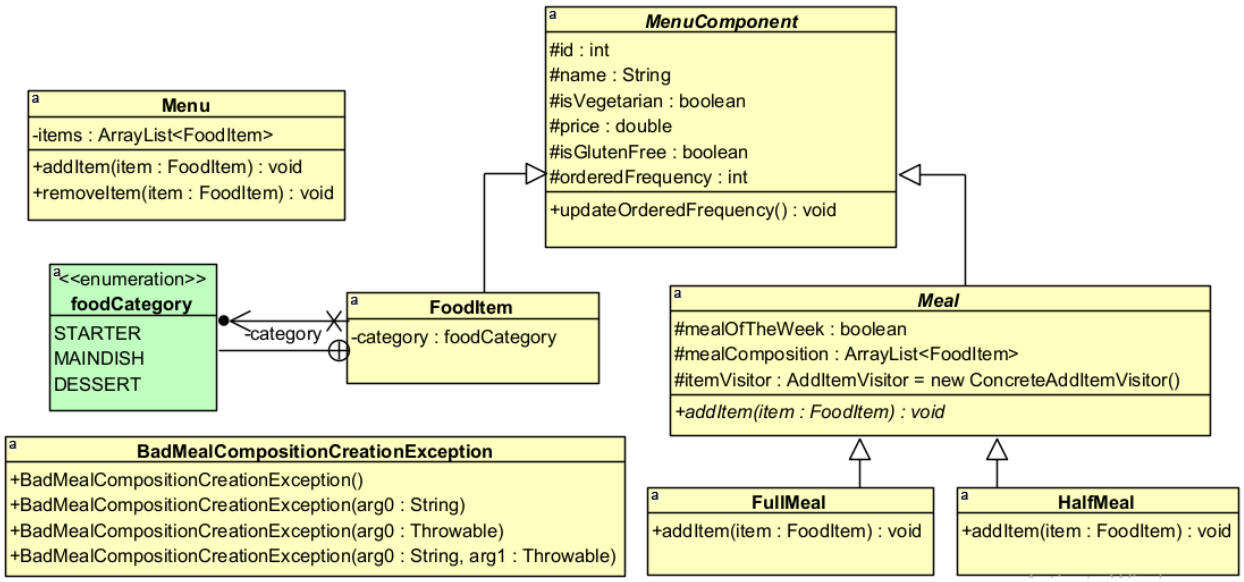


Figure 1: Partial UML Diagram for Menu and Meals

As described in the UML diagram, this section of `myFoodora` infrastructure comprises classes which handle different aspects of menu and meal offerings. The structured relationships and functionalities are elaborated as follows:

2.1.1 MenuComponent class

Attributes: Contains common identifiers and attributes :

- `int id`
- `String name`
- `boolean isVegetarian`
- `double price`
- `boolean isGlutenFree`
- `int orderedFrequency` : this attribute reflects how frequently an item has been ordered.

Methods:

- Getters and Setters
- `public void updateOrderedFrequency()` : updates the `orderedFrequency` whenever an order containing `this` menu component is submitted.

2.1.2 FoodItem class

Attributes: Inherits attributes from `MenuComponent` and integrates category, a designation based on the *`foodCategory`* enumeration which includes the three main categories: STARTER, MAINDISH, and DESSERT.

2.1.3 Meal abstract class

Attributes:

- Inherits attributes from `MenuComponent`.
- `boolean mealOfTheWeek` : boolean indicator of whether the meal is featured as a special for the week.
- `ArrayList<FoodItem> mealComposition`: lists the ingredients or components of the meal.
- `addItemVisitor visitor` (discussed in the following paragraph).

Methods:

- Getters and Setters

- `public abstract void addItem(item: FoodItem) throws BadMealCompositionCreationException` : allowing for the the addition of a dish to a certain meal, respecting its size and the requirements regarding its category (whether it is vegetarian/ gluten-free).

2.1.4 FullMeal and HalfMeal class

Derive from `Meal` and adapt the `addItem` method to meet specific requirements for complete and partial meal configurations.

Methods:

- `public abstract void addItem(item: FoodItem) throws BadMealCompositionCreationException`

2.1.5 Menu class

Stores the dishes in an ArrayList, and performs the operation of adding and removing a dish to a restaurant's menu.

Attributes:

- `ArrayList<FoodItem> items`

Methods:

- `public abstract void addItem(item: FoodItem)`
- `public abstract void removeItem(item: FoodItem)`

2.1.6 AddItemVisitor interface

In order to distinguish between the two different ways of adding a dish to a meal, depending on whether it is a half-meal of a full-meal, we implemented the **Visitor** design pattern. The `AddItemVisitor` interface, realized by the `ConcreteAddItemVisitor` class, allows operations on both `FullMeal` and `HalfMeal` without modifying their structures.

The following UML describes the implemented pattern along with the two corresponding methods.

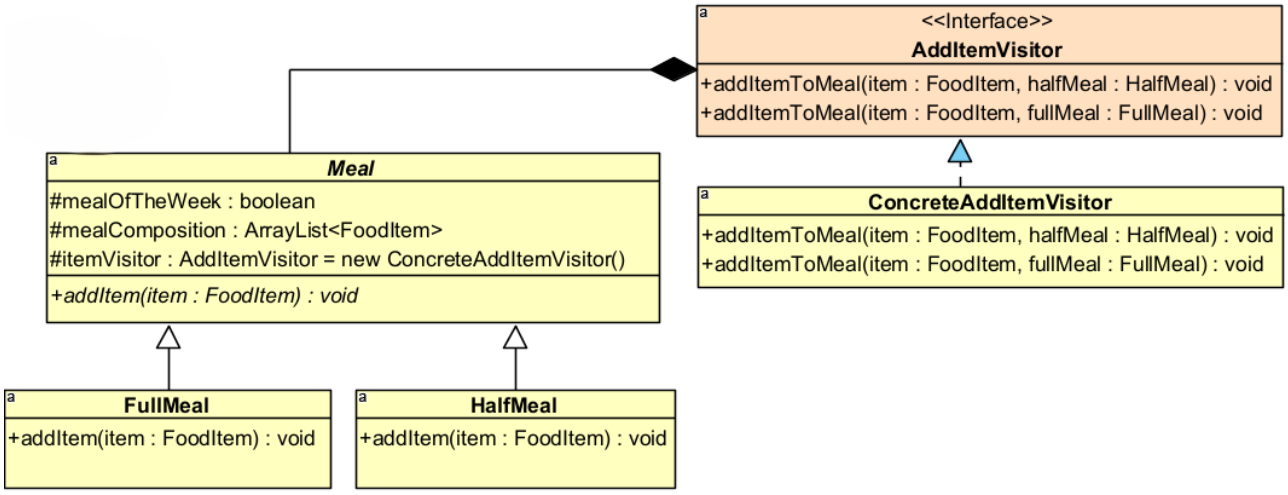


Figure 2: Adding dish to meals' visitor pattern UML Diagram

2.2 *Fidelity Cards*

The `FidelityCards` package contains one abstract class `FidelityCard`, three concrete classes; `BasicFidelityCard`, `PointFidelityCard` and `LotteryFidelityCard`, one `FidelityCardFactory` (**Factory** design pattern) to create a new fidelity card; the input being a `String : cardType`.
The pricing of a customer's order depends on the type of fidelity card he/she detains; for that reason, we have decided to implement a **Strategy** design pattern to compute the reduction and therefore the price with respect to the card's type.

2.2.1 FidelityCard abstract class

Attributes :

- *FidelityCardType* `cardType`

Methods :

- `public abstract double computeOrderReduction(Order order)`
- `public abstract double computeOrderPrice(Order order)`

2.2.2 BasicFidelityCard class

This class extends `FidelityCard`, it 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.
It doesn't introduce additional attributes, and the method `computeOrderReduction(Order order)` always returns 0.0 .

2.2.3 PointFidelityCard

Attributes :

- `int points`: the number of points accumulated so far.
- `double amountReduction = 10`: the price to reach in an order to receive 1 point.
- `int targetPoints = 100`: the number of points to reach in order to get a discount in the next order.
- `double discountFactor = 0.1`: the rate of the discount applied when the targetPoints value is reached.

Main methods :

- `public double computeOrderReduction(Order order)`
- `public double computeOrderPrice(Order order)`
- `public void updatePoints()`

2.2.4 LotteryFidelityCard

Attributes :

- `double probability = 0.02`
- `Calendar lastTimeUsed = null`

Main methods :

- `public double computeOrderReduction(Order order)`
- `public double computeOrderPrice(Order order)`
- `public boolean areDifferentDays(Calendar d1,Calendar d2)` : to check whether two `Calendar` dates are the same or not, seeing that the Lottery Fidelity card is only applicable once per day.

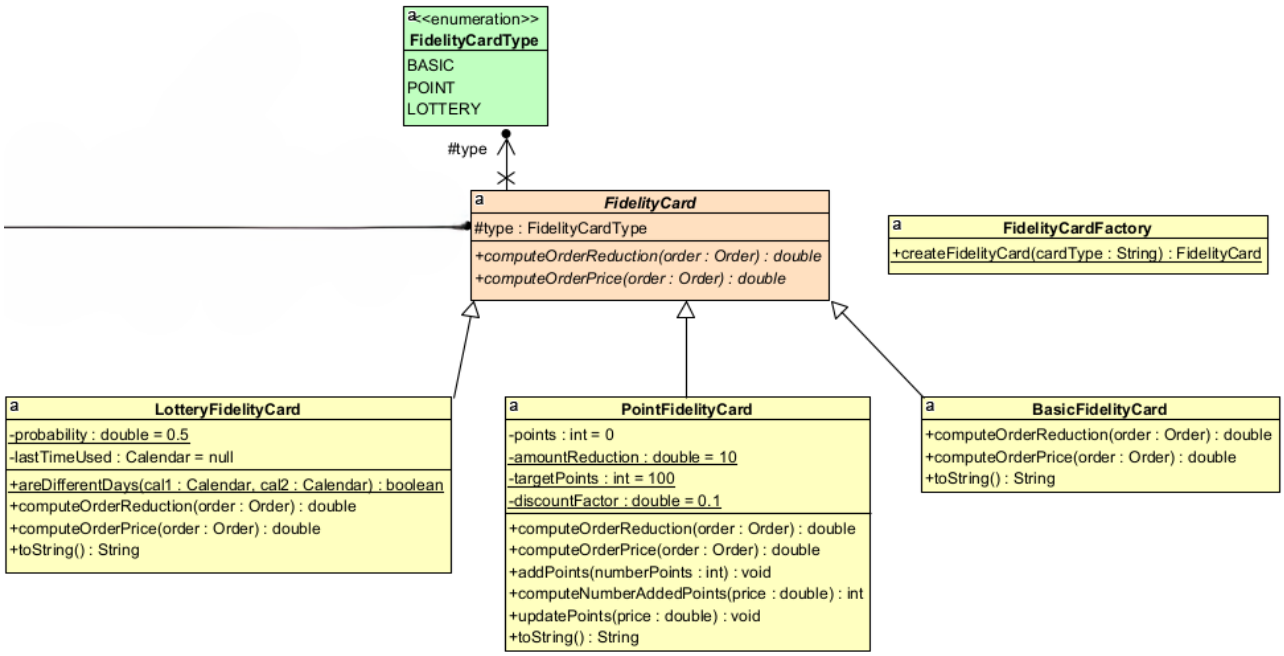


Figure 3: UML Diagram of Fidelity Card classes

2.3 *Delivery Policy*

The `delivery` package contains one interface `DeliveryPolicy`, two concrete classes implementing the latest; `FairOccupationDeliveryPolicy` and `FastestDeliveryPolicy`, one `DeliveryPolicyFactory` (**Factory** design pattern) to create a new DeliveryPolicy; the input being a `String : DeliveryType`. This choice was made in order to abstract the creation of objects logic from the Manager class which for now only detains the Factory and asks for a given policy to be returned.
The allocation of an off-duty Courier to order depends on the policy defined by the Manager. Thus, we decided to use a **Strategy** design pattern to implement this allocation, with each class defining its own `allocateCourierToOrder` method to meet the given constraint for choosing a courier to deliver a given order. This package usage takes place when a customer ends his order.

2.3.1 Enhancing the Open/Closed Principle

Our design enhances the Open/Closed Principle (OCP) by using the Strategy and Factory design patterns. The OCP states that software entities should be open for extension but closed for modification. Here's how our design achieves this:

- **Strategy Pattern**: By defining the `DeliveryPolicy` interface and having multiple implementations (`FastestDeliveryPolicy` and `FairOccupationDeliveryPolicy`), we can introduce new delivery policies without modifying the existing code. This allows the system to be easily extensible.

- **Factory Pattern**: The `DeliveryPolicyFactory` class abstracts the creation logic of the delivery policies. This means that the `Manager` class does not need to change when new delivery policies are added; it simply requests a policy based on a string identifier. This encapsulation supports OCP by localizing changes related to the instantiation of delivery policies.

By adhering to these patterns, we ensure that the system can grow and adapt to new requirements without disrupting the existing functionality, thereby maintaining robustness and reducing the risk of introducing bugs.

2.3.2 Design Limitations

Despite its advantages, our design has some limitations:

- **Potential Performance Overhead**: Using a factory and strategy pattern introduces additional layers of abstraction. While this enhances flexibility and adherence to design principles, it might also introduce slight performance overhead, especially in a high-load system where policy creation and switching occur frequently.
- **Dependency on String Identifiers**: The current implementation of `DeliveryPolicyFactory` relies on string identifiers to create policies. This approach can be error-prone, as any typo or mismatch in the string could lead to runtime errors. A more robust solution could involve using enums or constants to represent policy types.

2.3.3 DeliveryPolicy Interface

Method :

- `public abstract void allocateCourierToOrder(List<Courier> couriers, Order order)`

2.3.4 FastestDeliveryPolicy class

This class implements the `DeliveryPolicy` interface, hence overrides its `allocateCourierToOrder` method. Given the list of couriers, this method checks for an available courier his distance to the restaurant where he would be getting the order and keeps track of the closest one so far. Once the closest courier to the restaurant is found, the method allocates the order and performs the necessary changes. If no courier is found an Exception is raised which will be handled by the system.

2.3.5 FairOccupationDeliveryPolicy class

Similar to the previous class, the only difference relies on the allocation method which uses this time the number of completed orders by the courier.

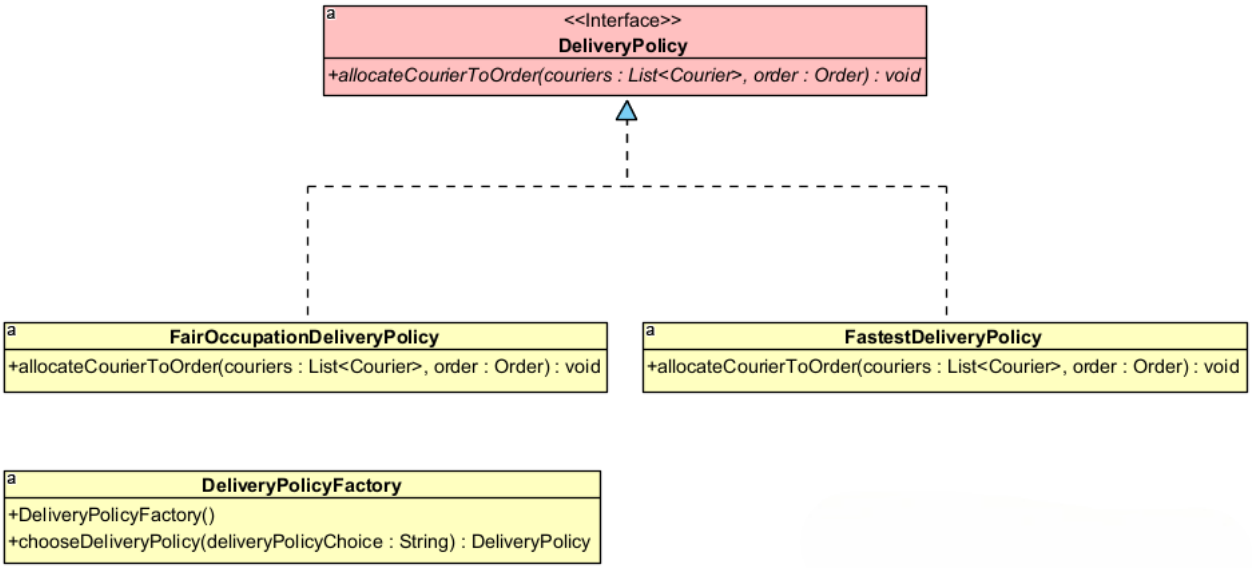


Figure 4: UML Diagram of the DeliveryPolicy feature

2.4 *Target Profit Policy*

The `targetProfit` package contains one interface `TargetProfitPolicy`, three concrete classes implementing the latest; `MarkupTargetPolicy`, `DeliveryCostServicePolicy` and `ServiceFeePolicy`, one `TargetProfitPolicyFactory` (**Factory** design pattern) to create a new TargetProfitPolicy; the input being a `String : TargetProfitType`. This choice was made in order to abstract the creation of objects logic from the Manager class which for now only detains the Factory and asks for a given policy to be returned.
Also, in order to address the ability to adjust parameters to achieve a certain profit. We decided to use a **Strategy** which allows us to compute the new parameter's value given the chosen policy by the manager and incorporate it in the AppSystem.

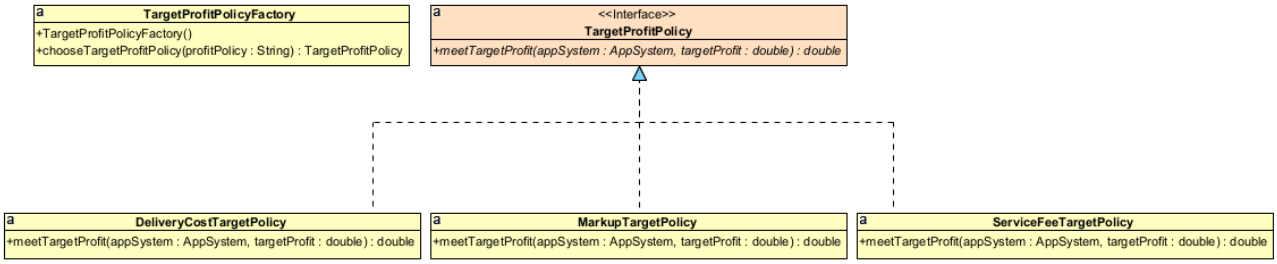


Figure 5: UML Diagram of the TargetProfitPolicy feature

2.5 *Users of myFoodora system*

The `User` class is the main class which represents the parent class of the four different users of `myFoodora` system.
Attributes : It has four main attributes:

- `int id`
- `String name`
- `String username`
- `String password`

The unique `id` is generated using a **Singleton** instance of the `UserIdGenerator` class, which sequentially increments a counter each time a new ID is requested.

For users with the attribute sat using composition from the Coordinate class, this attribute uses a random function to assign x and y coordinates within a rectangle of a given side length.

2.5.1 Restaurant

It extends the `User` class. A restaurant has all common characteristics of a user, in addition to the following attributes:
Attributes :

- `Coordinate address` : initialized randomly within a certain bounding box.
- `Menu menu`
- `double genericDiscount`
- `double specialDiscount`
- `ArrayList<Meal> meals`

- `MealPriceStrategy mealPriceStrategy`

- `FoodFactory foodFactory`

Main methods :

- Getters and Setters

- `public Meal createMeal(String mealName, String mealType)`

- `public void addDish2Meal(String mealName, String dishName) throws NotFoundException, BadMealCompositionCreationException`

- `public void addDishRestaurantMenu(FoodItem item)`

- `public void removeItemMenu(FoodItem item)`

- `public void addMeal(Meal meal)`

- `public void removeMeal(Meal meal)`

- `public void setPriceStrategy(MealPriceStrategy strategy)` : set Pricing strategy used to compute the meal's price, in the case of the project, it's only a discount based strategy.

- `public void setMealPrice(Meal meal)` : sets the price of a certain meal (based on MealOfTheWeek attribute).

- `public void setSpecialOffer(Meal meal)`

- `public void removeFromSpecialOffer(FoodItem item)`

- `public void showSortedHalfMeals()`

- `public void showSortedDishes()`

N.B: Since we are implementing a command line interface, the commands we receive are in string format. Consequently, most methods in the Restaurant class accept strings as arguments. We utilize methods like findDishUsingName or findMealUsingName to identify and execute tasks based on dish or meal names. This list highlights only a selection of available methods to maintain brevity.

2.5.2 Customer

Attributes :

- `String surname`
- `Coordinate address`: initialized randomly within a certain bounding box.
- `String email`
- `String phone`
- `ArrayList<Order> orderHistory`
- `FidelityCard fidelityCard`
- `boolean consensus`: the consensus of the customer to receive the special offer notifications.
- *`Contact_offers`* `contact_offers = Contact_offers.email`: the contact through which the customer receives the special offers.

Main methods:

- `public void endOrder()`: submit the order.
- `public void payOrder()`
- `public void registerFidelityCard()`

In order to notify the customers of the special offers, we decided to implement an **Observer** design pattern. Therefore, the customer class implements the `Observer` interface, and overrides the `update` function, which displays a successfully sent phone or email notification message.

- `public void update(Restaurant restaurant, Offer offer,Meal meal)` +

2.5.3 Courrier

It extends the `User` class. A courier has all common characteristics of a user, in addition to the following attributes:

Attributes:

- `String surname`
- `Coordinate location`: initialized randomly within a certain bounding box.

- `String phone`

- `int deliveryCounter`

- `boolean OnDuty`

Main methods:

- Getters and setters are implemented to manage the class's attributes.

- `public void setOnDuty(boolean OnDuty)`

- `public void incrementDeliveryCounter()`

2.5.4 Manager

It extends the `User` class. A manager has all common characteristics of a user, in addition to the following attributes:

Attributes :

- `AppSystem appSystem`

  The single instance of MyFoodora, we decided to add it as an attribute in the Manager class since the latter performs several changes on the system as part of his responsibility.

- `Menu menu`

- `String surname`

- `DeliveryPolicyFactory deliveryPolicyFactory`
  The manager orders the creation of a new DeliveryPolicy object to be handed over to the AppSystem class and makes the new delivery policy used for allocating couriers to an order.

- `TargetProfitPolicyFactory targetProfitPolicyFactory`

  For the same reason as above we have decided to add the factory as an attribute to the manager.

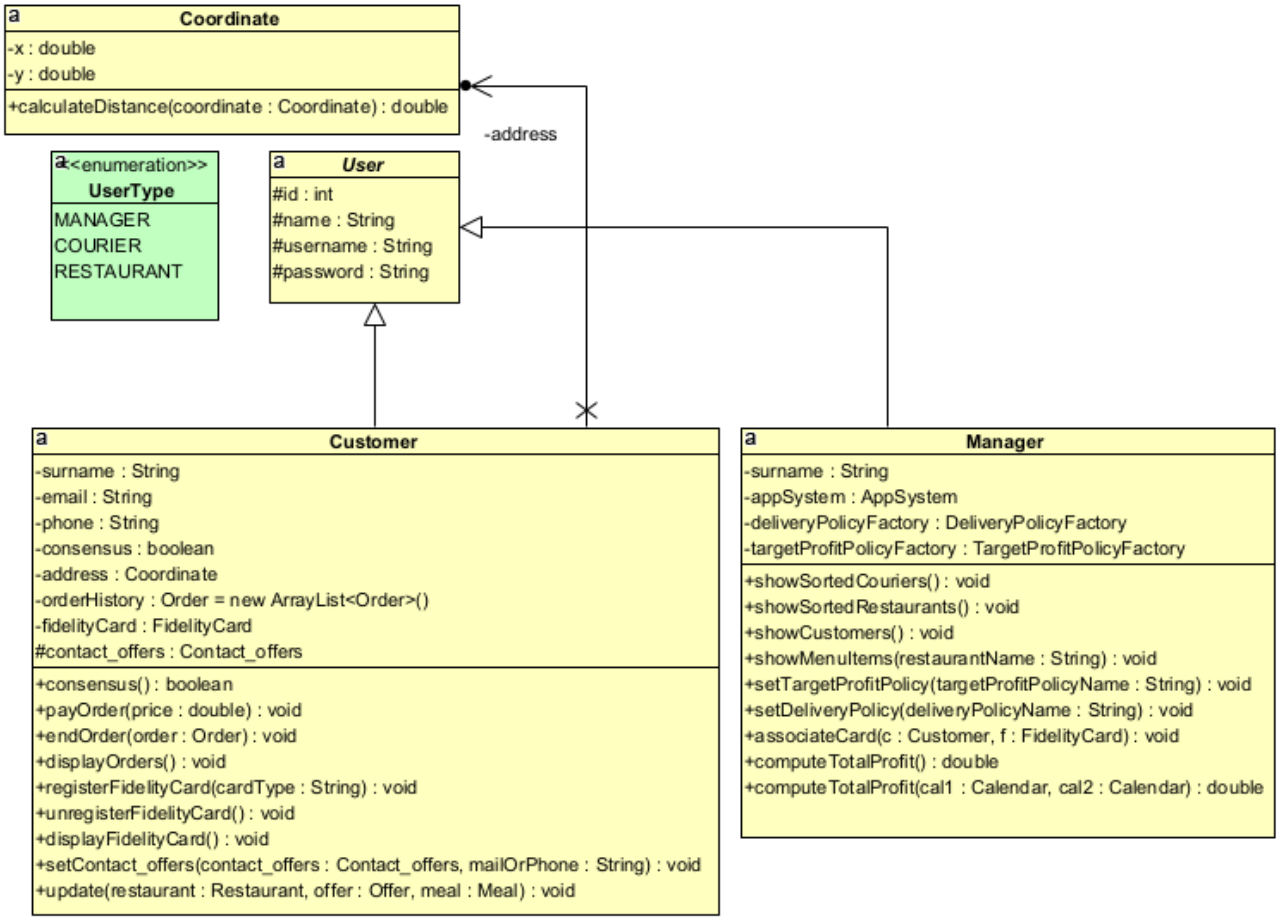
Main methods :

- Getters and setters are implemented only when necessary to manage the class's attributes. For this class, only 'surname' has them.

- `public void showSortedCouriers()`

- `public void showSortedRestaurants()`

- `public void showCustomers()`
- `public void showMenuItems(String restaurantName)`
- `public void setTargetProfitPolicy(String targetProfitPolicyName)throws unknownProfitPolicyException`
- `public void setDeliveryPolicy(String deliveryPolicyName) throws unknownDeliveryPolicyException`
- `public void associateCard(Customer c, FidelityCard f)`
- `public double computeTotalProfit()`
- `public double computeTotalProfit(Calendar cal1, Calendar cal2)`



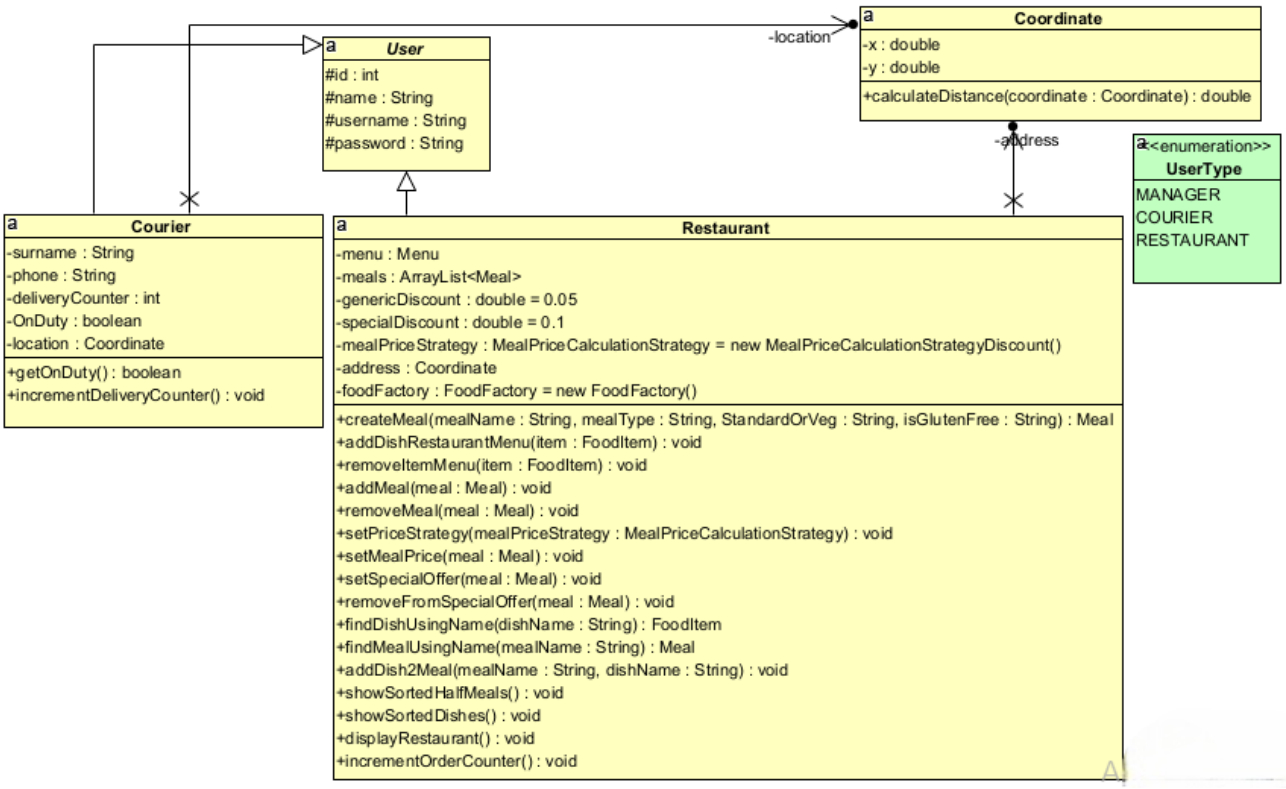


Figure 6: UML Diagram of the User related classes

N.B: To adhere to the separation of concerns principle, we have encapsulated all manager-related tasks within the Manager class. When a user selects a command related to manager functionality in the CLI, the command is routed to the AppSystem. The AppSystem verifies that the current user has the appropriate permissions to execute manager actions. If the user is authorized, the command is forwarded to the Manager class for processing. We have also chosen to handle the registration of new customers initiated by a manager within the AppSystem class. This decision is based on the fact that no additional processing is required by the Manager class; the only necessary action is the verification of proper authorization, which is already managed by the AppSystem class.

2.6 *Notification Service*

When a customer gives its consent about receiving special offers shared by restaurants, he/she gets notified whenever an offer is set. This is what the **Observer** design pattern is all about. In the package `NotificationService`, we defined the two interfaces; `Observable` and `Observer`, along with the `Offer` enumeration, seeing that an offer can either be, a meal which is set as mealOfTheWeek, a new generic discount or a new special discount.

2.6.1 NotificationService class

The `NotificationService` class is designed as a **Singleton** which implements the `Observable` interface. This design pattern ensures that only one instance of the NotificationService can exist throughout the lifespan of an application.

Attributes :

- `NotificationService instance = null`
- `ArrayList<Observer> subscribers = new ArrayList<Observer>()`
- `boolean changed`

Methods :

- `public void registerObserver(Observer c)`
- public void removeObserver(Observer c)
- `public void notifyObservers(Meal meal, Restaurant restaurant, Offer offer)`: Notifies all subscribed observers by calling their update method, provided there has been a change.
- `public void setOffer(Meal meal, Restaurant restaurant, Offer offer)`: Marks the state as changed and notifies all observers about the new offer.

When a restaurant releases one of the mentioned offers, it calls the `NotificationService` instance to set the offer and notify the subscribers.

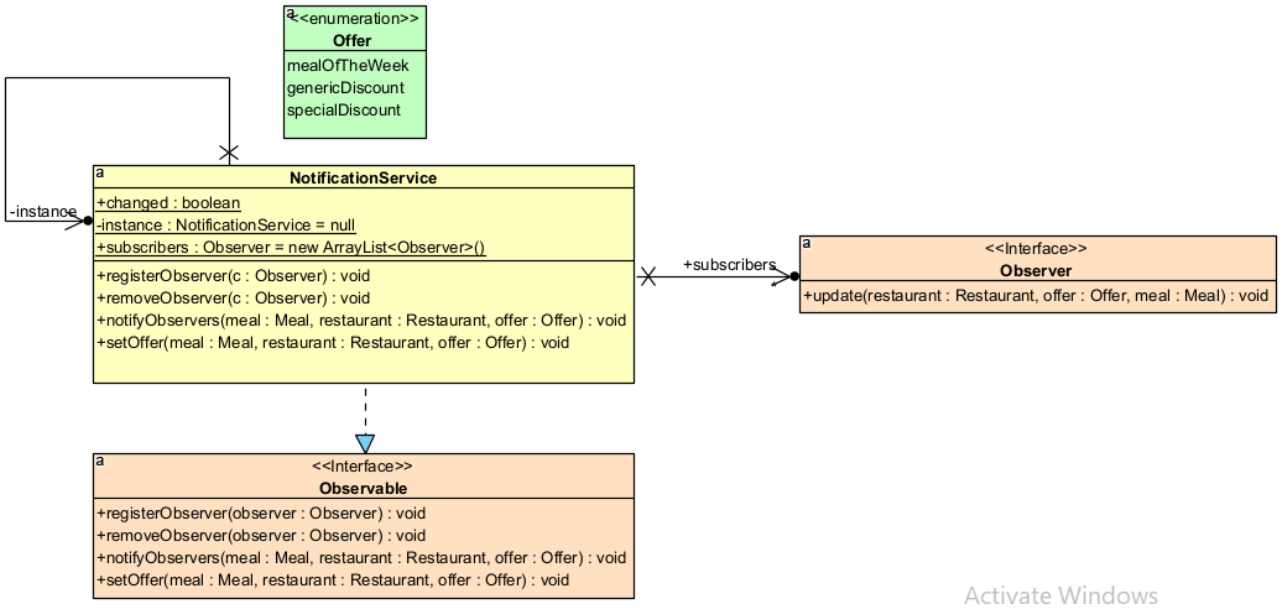


Figure 7: UML Diagram of Notification Service

2.7 *Order*

Attributes :

- `Restaurant restaurant`
- `AppSystem appSystem` - The myFoodora system class, defined further below.
- `int id`
- `Calendar orderDate`
- `Customer customer`
- `Courier courier`
- `Map<FoodItem, Integer> orderItems = new HashMap<FoodItem, Integer>()` - A map storing each food item and its corresponding quantity in the order.
- `Map<Meal, Integer> orderMeals = new HashMap<Meal, Integer>()` - A map storing each meal and its corresponding quantity in the order.
- `double price` - The price of the order before adding any extra charges.
- `double profit` - The profit made by myFoodora from the order. It is computed in this stage to take into consideration the current profit parameters; serviceFee, deliveryCost and markupPercentage.
- `double totalPrice` - The total price of the order including the service fee and the markup percentage.

Main methods :

- `public void addItem(String foodName, int quantity) throws NotFoundException` - Adds an item or meal to the order by name, updating the quantities accordingly.
- `public double getFirstPrice()` - Calculates the initial price of the order without any discounts by summing the product of each item or meal's price and its corresponding quantity.
- `public double getFinalPrice()` - Calculates the final price of the order considering any applicable discounts through the customer's fidelity card.
- `public void submitOrder(double price)` - Submits the order, adds it to the customer's order history, updates the ordered frequency of items, computes the total price which the customer has to pay, computes the profit made by myFoodora system and displays the order details and final price.

2.8 *Exceptions*

- **BadMealCompositionCreationException:** a custom exception class intended to handle errors during the meal composition pro- cess, e.g: attempting to add a third item to a half-meal, attempting to add a standard dish to a vegetarian meal.

- **NonReachableTargetProfitException:** given a target profit, the current income over the previous month and other profit parameters (say ServiceFee and markupPercentage), we want to compute the corresponding profit parameter (say deliveryCost) value in order to reach a certain target profit. If it is not possible, then this exception is thrown.

- **NoPermissionException :** a custom exception class used to indicate that a user lacks the necessary permissions to execute a command. It is thrown when a user tries to run a given command which is not accessible to his/her user type.

- **NotFoundException :** this exception is thrown whenever we attempt to use or look for something (a food item, a meal, a restaurant, ..) which is not stored in

2.9 *AppSystem - myFoodora core class*

The `AppSystem` class serves as a central component in this project framework, designed to facilitate the management of users, orders, and interactions within a service-oriented environment. Employing the **Singleton** design pattern, it ensures that only one instance of the class is created, guaranteeing a consistent state across the application.

Attributes :

- `private static AppSystem instance = null` - Singleton instance of the AppSystem class.

- `List<Manager> managers = new ArrayList<>()`

- `List<Customer> customers = new ArrayList<>()`

- `List<Courier> couriers = new ArrayList<>()`

- `List<Restaurant> restaurants = new ArrayList<>()`

- `List<Order> orders = new ArrayList<>()` - List to store orders.

- `private static Optional<User> currentUser = Optional.empty()` - Optionally holds the currently logged-in user.

- `private static Optional<UserType> currentUserType = Optional.empty()`
  - Optionally holds the type of the currently logged-in user.

The `AppSystem` class is highly correlated to the user interface, as it stores the currently logged-in user as an attribute. Each method within this class first verifies if the type of the logged-in user aligns with the required user type for the requested action. If the user type matches, the class delegates the operation to the appropriate method previously defined in the user-specific class (such as `Manager, Customer`, etc.). This design ensures that the logic pertinent to each user type is encapsulated within its respective class. Conversely, if there is a mismatch between the user type needed for a method and that of the current user, the system enforces security by throwing a `NoPermissionException`.
The Constructor of the `AppSystem` class sets two initial managers, who get to initialize the application.

Additional methods :

- `public boolean login(String username, String password)` - Attempts to log a user into the system.
- `public void logout()` - Logs the current user out of the system.

2.10 *AppSystemCLI - Command Line Interface*

The AppSystemCLI is a command-line interface (CLI) for myFoodora. This class enables users to interact with the system via text commands. It also serves as an entry to our app. Thus, to turn MyFoodora on this file needs to be run. The CLI is designed to manage various user interactions such as login, registration, order creation, and administrative tasks. Here is a description of the supported commands:

- **User Authentication Commands**
  - `login <username> <password>`: Authenticates a user and displays a list of available restaurants for customers.
  - `logout`: Logs out the current user and resets customer orders.
- **Registration Commands**
  - `registerCustomer <firstName> <lastName> <username> <password>`: Registers a new customer.
  - `registerRestaurant <name> <username> <password>`: Registers a new restaurant.
  - `registerCourier <firstName> <lastName> <username> <password>`: Registers a new courier.

- **Order Management Commands**
  - `createOrder <restaurantName> <orderName>`: Creates a new order for a specified restaurant.
  - `addItem2Order <orderName> <itemName>`: Adds an item to an existing order.
  - `endOrder <orderName>`: Finalizes an order.
  - `historyOfOrders`: Displays the history of orders for the current customer.
- **Administrative Commands**
  - `showCourierDeliveries`: Displays sorted couriers.
  - `showRestaurantsTop`: Displays sorted restaurants.
  - `showMenuItem <restaurantName>`: Displays the menu of a specified restaurant.
  - `setDeliveryPolicy <delPolicyName>`: Sets the delivery policy.
  - `setProfitPolicy <ProfPolicyName>`: Sets the profit policy.
  - `associateCard <userName> <cardType>`: Associates a fidelity card with a user.
  - `showTotalProfit`: Displays the total profit, with optional date range parameters.
- **Restaurant Management Commands**
  - `showMenu`: Displays the menu of the current restaurant.
  - `setSpecialDiscountFactor <specialDiscountFactor>`: Sets a special discount factor.
  - `setGenericDiscountFactor <genericDiscountFactor>`: Sets a generic discount factor.
  - `addDishRestaurantMenu <dishName> <dishCategory> <foodType> <glutenFree> <unitPrice>`: Adds a dish to the restaurant menu.
  - `createMeal <mealName> <mealType>`: Creates a new meal.
  - `addDish2Meal <dishName> <mealName>`: Adds a dish to an existing meal.
  - `showMeal <mealName>`: Displays information about a specified meal.
  - `setSpecialOffer <mealName>`: Sets a meal as a special offer.
  - `removeFromSpecialOffer <mealName>`: Removes a meal from special offers.

  - `showSortedHalfMeals`: Displays sorted half meals.
  - `showSortedDishes`: Displays sorted dishes.

- **Courier Management Commands**
  - `onDuty <username>`: Sets a courier's status to on-duty.
  - `offDuty <username>`: Sets a courier's status to off-duty.

- **Help Commands**
  - `help <>`: Displays available commands and their descriptions.
  - `runTest <file-name>`: Run tests from given textfile path.

3 Test Scenario

In this simulation, the manager, denoted as Salma, initiates the test by authenticating into the system and proceeds to configure the environment. This setup includes the registration of two restaurants, NoodleHouse and BurgerKing, alongside the enrollment of other users including customers Alice Waters and Thomas Keller, and couriers Michael Johnson, Carl Lewis, and Flash Gordon. After registering, Salma queries the system to display details about customers, courier deliveries, and top-performing restaurants before logging out.

Subsequently, the NoodleHouse account is accessed to expand its menu offerings, including dishes such as ramen pork and vegetarian sushi rolls, alongside the creation of meal combinations like ComboA and ComboB. Special offers are set for ComboA, and a generic discount factor is applied to enhance customer attraction. Afterward, the account logs out.

Alice Waters, a registered user, logs in to update her contact information, specifically her email address, to set the consensus about receiving notifications about the restaurants' special offers.

BurgerKing follows a similar pattern, enriching its menu with items like cheeseburgers and veggie burgers, and crafting meal deals such as MegaMeal and MiniMeal. A special offer is designated for MiniMeal and the menu is displayed with updated pricing following a set discount factor.

The scenario progresses with customer interactions. New customer Remy Linguini registers and immediately leverages a Point fidelity card to place an order at BurgerKing, selecting a variety of meals and dishes. Similarly, Alice Waters, another customer, uses her fidelity card benefits to order from NoodleHouse,

including the special ComboA and ComboB meals. Throughout these interactions, various system functions such as displaying fidelity card information, reviewing order histories, and updating system policies (like delivery and profit strategies) are tested.

In parallel, courier activities are managed, with Carl Lewis and Flash Gordon toggling their duty status, ensuring the delivery operations can be dynamically adjusted according to real-time needs.

The testing culminates with Manager Salma reviewing the total profit and adjusting profit strategies to ensure financial targets are met. The scenario concludes with Salma reviewing system-wide statistics on profits, customer engagements, and restaurant performance, thus thoroughly examining every facet of the restaurant management system's capabilities and its response to diverse operational demands.

4 Strating the app

To launch the CLUI, navigate to the package CLI, and go to the file. Then you can run the file by clicking on the run button. Once launched, the CLUI will display a prompt symbol with the app name indicating that it is ready to receive commands. To run tests please type in the following command:

runTest testScenario.txt

5 Task Distribution

Task	**Assigned To**
Users of myFoodora	Salma
Menus and Meals	Hala
Fidelity Cards	Hala
Delivery Policy	Salma
Notification Service	Hala
Target Profit Policy	Salma
JUnit	Hala
AppSystem and CLI	Hala and Salma

Table 1: Task Distribution