CentraleSupélec

MyFoodora : Food delivery service

OUCHNA Yassine
YARTAOUI Farouk

May 2024 - June 2024

Table des matières

1 Introduction

This report describes in detail the code and the design thinking behind the development of a delivery service in Java called *MyFoodora*, the code of which you can find here Github link.
This project is part of an assignment of the *Object Oriented Software Engineering* course at CentraleSupelec (2024).

1.1 Structure of the project

In any Java project, it's best to organize the defined classes in different packages. This helps a lot in term of organization and encapsulating different parts of the system in specified folders.

The following is a brief overview of the packages :
- **core** : Core functionalities of the *MyFoodora* app.
  - **users** : Package for managing different type of users like Restaurants, Customers and Managers of the *MyFoodora* app.
  - **comparators** : Package for implementing comparators for MenuItems and Restaurants.
  - **enums** : Relevant Enums.
  - **exceptions** : Relevant custom exceptions.
  - **fidelityCards** : Fidelity cards for customers.
  - **food** : Everything food related from Dishes, Menus, MenuItems and Meals.
  - **orders** : Package managing Orders.
  - **policies** : Package managing different policies for delivery and pricing.
- **test** : Package holding JUnit tests for the whole *MyFoodora* app including the core functionalities and the interface.
  - **core** : Tests for the core of the app.
  - **interfac** : Tests for the interface.
- **MyFoodora.java** : The main file of the system containing the relationships between its different functionalities.
- **MyFoodoraClient.java** : The CLI interface that takes care of the interaction with the users.

2 The Core Packages

2.1 The `core.food` Package

The `core.food` package is a fundamental component of the *MyFoodora* application, encapsulating the core functionalities related to menu items, dishes, and meals.

This package plays a crucial role in defining and managing the food items offered by restaurants within the system. The main classes in this package are `MenuItem`, `Dish`, `Meal`, and the `Menu` class that manages the collection of these items.

2.1.1 The `MenuItem` Class

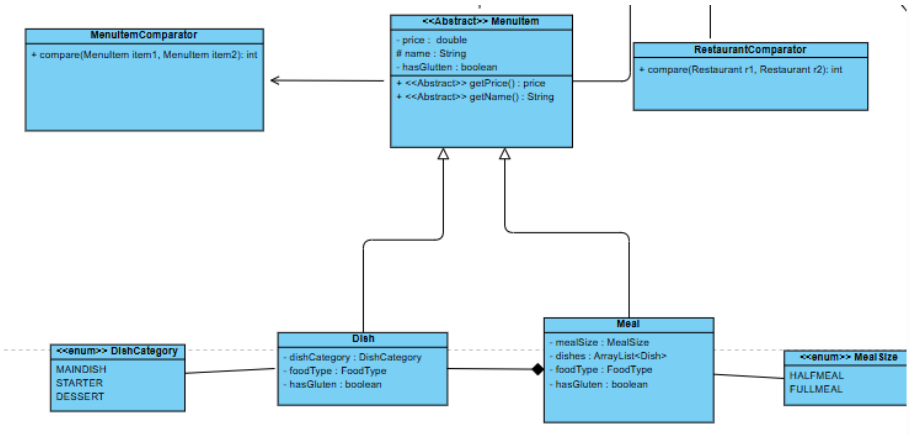


FIGURE 2 – MenItem Class UML

At the heart of the `core.food` package is the abstract `MenuItem` class. This class serves as the base class for all food items in the system. A `MenuItem` includes essential attributes such as :

— `name` : The name of the menu item.
— `price` : The price of the menu item.
— `orderFrequency` : A counter to track the frequency of orders for the item.

The `MenuItem` class provides getter methods for these attributes, ensuring that derived classes can access and manipulate these values appropriately.

2.1.2 The `Dish` Class

The `Dish` class extends `MenuItem` and represents the atomic components of a restaurant's menu. In addition to the inherited attributes, a `Dish` includes specific details such as :

— `hasGluten` : A boolean indicating whether the dish contains gluten.
— `foodType` : An enum specifying the type of food (e.g., `VEGETARIAN`, `STANDARD`).
— `dishCategory` : An enum categorizing the dish (e.g., `MAIN`, `DESSERT`).

The `Dish` class allows for detailed descriptions and precise categorization of individual dishes, enabling restaurants to define their menu items accurately.

2.1.3 The `Meal` Class

The `Meal` class also extends `MenuItem` and represents a combination of multiple `Dish` instances. A `Meal` includes :

- `dishes` : A list of `Dish` objects that make up the meal.
- `mealSize` : An enum indicating the size of the meal (e.g., `FULLMEAL`, `HALFMEAL`).
- `hasGluten` and `foodType` : Attributes determined based on the constituent dishes.

The `Meal` class aggregates several dishes and calculates the total price and other properties, offering a comprehensive representation of complex menu offerings.

2.1.4 The `Menu` Class

The `Menu` class manages a collection of `MenuItem` objects, providing methods to add, remove, and retrieve items. It also handles special offers by allowing restaurants to set a specific `Meal` as a special offer. The `Menu` class ensures that all items are properly categorized and available for customer orders.

The `Menu` class allows adding items of type `DISH` or `MEAL` to the menu by verifying descriptions and applying the generic discounts defined by the restaurant. It also allows retrieving and removing items from the menu while managing a special offer that can be set by the restaurant.

The classes in the `core.food` package collectively form the foundation of food-related functionalities in the *MyFoodora* application, enabling detailed and flexible management of menu items, dishes, and meals offered by restaurants.

2.1.5 Design Pattern Considerations

Initially, we aimed to implement a design pattern to handle the creation of `MenuItem` instances, with the `Restaurant` or `Menu` acting as the client. We considered both the simple factory pattern and the abstract factory pattern. However, we encountered some challenges.

The simple factory pattern involves a single factory class with a method to create different types of `MenuItem` objects based on input parameters. This approach seemed suitable given the limited number of `MenuItem` subclasses (`Dish` and `Meal`). However, it introduced unnecessary complexity since the creation of `MenuItem` objects is straightforward and does not require hiding or encapsulating complex creation logic.

Similarly, the abstract factory pattern, which involves creating families of related or dependent objects, was considered but ultimately deemed overly complex for our needs. The `core.food` package does not have a large number of concrete `MenuItem` types, and the additional abstraction did not provide significant benefits.

In conclusion, we decided against implementing a factory pattern for `MenuItem` creation, opting instead for direct instantiation within the `Menu` class. This approach

simplifies the codebase and maintains clarity, given the straightforward nature of our `MenuItem` hierarchy.

2.2 The `core.users` Package

2.2.1 Overview

The `core.users` package contains classes representing the users of the MyFoodora system. These classes include `User`, `Restaurant`, `Customer`, `Courier`, and `Manager`. Each specific subclass of `User` has special functionalities. Additionally, we have implemented an observer pattern where the `Customer` is the observer and the `MyFoodora` system is the observable.

2.2.2 User Class

Overview The `User` class is the base class for all user types in the system. It includes common attributes and methods for all users.

Properties

- `private String surname;`
- `private String name;`
- `private String phoneNumber;`
- `private String email;`
- `private boolean isActive;`
- `private int id;`
- `private static int nextId = 0;`
- `private String username;`
- `private int hashedPassword;`

Functionalities

- Users can be activated or deactivated.
- Each user has a unique identifier (`id`), and their login information includes a username and hashed password.
- Users have methods for retrieving and setting their personal details.

Key Methods

- `login()` : Placeholder for user login functionality.
- `setActive(boolean isActive)` : Activates or deactivates a user.
- `setName(String name)`, `setSurname(String surname)`, etc. : Methods for setting user details.

2.2.3 Restaurant Class

Overview The `Restaurant` class inherits from `User` and includes additional properties and functionalities specific to restaurants.

Additional Properties
- `private double[] location;`
- `private Menu menu;`
- `private double genericDiscount;`
- `private double specialDiscount;`
- `private int numDeliveredOrders;`

Functionalities
- Restaurants manage a menu and can set special offers.
- They track the number of delivered orders and can notify subscribers of special offers.

Key Methods
- `addMenuItem(String itemType, String[] description)` : Adds an item to the restaurant's menu.
- `removeItem(String itemName)` : Removes an item from the menu.
- `setSpecialOffer(Meal specialOffer)` : Sets a meal as a special offer and notifies subscribers.

2.2.4 Manager Class

Overview The `Manager` class inherits from `User` and includes additional functionalities specific to managing the system.

Additional Functionalities
- Managers can manage users, set service fees, compute profits, and set policies for the MyFoodora application.
- They can activate or deactivate users and manage delivery and profit policies.

Key Methods
- `addUser(User u)`, `removeUser(User u)` : Methods for managing users in the system.
- `activateUser(User u)`, `desactivateUser(User u)` : Methods for activating or deactivating users.
- `totalProfit(Date start, Date end)`, `totalIncome(Date start, Date end)` : Methods for calculating financial metrics.

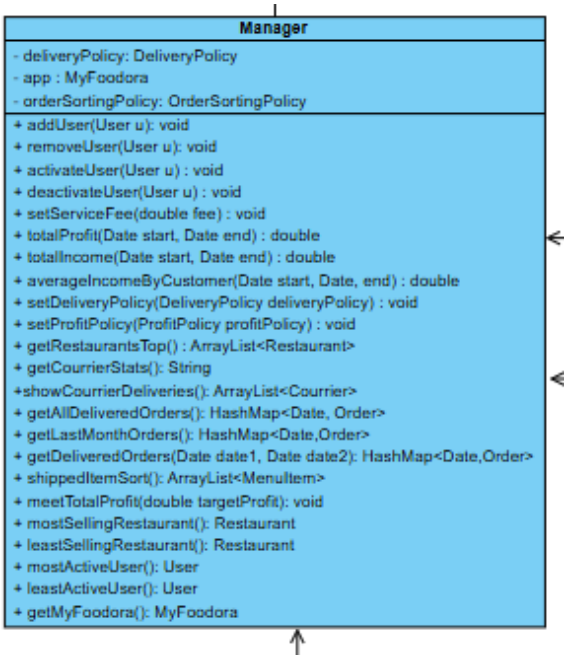


FIGURE 3 – Manager Class UML

— `meetTargetProfit(double targetProfit)` : Adjusts parameters to meet the target profit.
— `showRestaurantTop()` : Returns a list of top-performing restaurants.

2.2.5 Customer Class

Overview The `Customer` class inherits from `User` and includes additional properties and functionalities specific to customers.

Additional Properties

— `private double[] address;`
— `private boolean notificationsOn;`
— `private FidelityCard fidelityCard;`
— `private ArrayList<String> receivedEmails;`

Functionalities

— Customers can subscribe to special offers, manage their fidelity card, and view their order history.
— They can receive notifications for special offers if subscribed.

Key Methods

— `updateSubscriber(String restaurantName, Meal specialOffer)` : Receives updates on special offers from subscribed restaurants.
— `registerFidelityCard(FidelityCard fidelityCard)`, `unregisterFidelityCard()` : Methods for managing fidelity cards.
— `receiveMail(String message)` : Simulates receiving an email about a special offer.

2.2.6 Courier Class

Overview The `Courier` class inherits from `User` and includes additional properties and functionalities specific to couriers.

Additional Properties
— `private int completedDeliveries;`
— `private double[] position;`
— `private boolean onDuty;`
— `private ArrayList<Order> board;`

Functionalities
— Couriers manage their delivery tasks, accept or decline delivery calls, and track their delivery history.

Key Methods
— `acceptDeliveryCall(boolean decision, Order waitingOrder)` : Accepts or refuses a delivery call for a waiting order.
— `increaseCounter()` : Increases the counter of completed deliveries.
— `findOrderIdInBoard(int id)` : Finds an order in the courier's board by its ID.

2.2.7 SubscriberObserver & SubscriberObservable Interfaces

SubscriberObserver
— **Purpose** : Defines a method for receiving updates.
— **Key Method** : `updateSubscriber(String restaurantName, Meal specialOffer)` : Receives updates from observable entities (e.g., restaurants).

SubscriberObservable
— **Purpose** : Defines methods for managing subscribers.
— **Key Methods** :
  — `addSubscriber(SubscriberObserver o)` : Adds a subscriber.
  — `removeSubscriber(SubscriberObserver o)` : Removes a subscriber.
  — `notifySubscribers(String restaurantName, Meal specialOffer)` : Notifies subscribers about special offers.

2.3 `core.orders` Package

2.3.1 Order Class

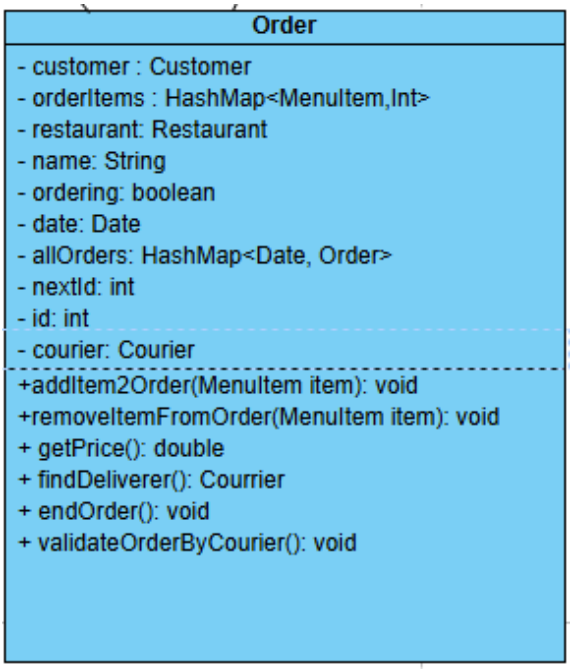


FIGURE 4 – Classe Order UML

Overview The `Order` class is a fundamental part of the `core.orders` package, representing an order placed by a customer in the MyFoodora system. It includes essential details such as the customer, restaurant, items ordered, and the courier responsible for delivery. The class provides functionalities to manage the order lifecycle, from creation to completion.

Properties

- `private static int nextId = 0;`
- `private int id;`
- `private Customer customer;`
- `private String name;`
- `private Restaurant restaurant;`
- `private Map<MenuItem, Integer> orderItems;`
- `private boolean ordering = true;`
- `private Date date;`
- `private Courier courier;`

Functionalities

- **Order Creation and Initialization :** Orders are initialized with unique IDs, linked to a restaurant, and optionally assigned to a customer.
- **Adding and Removing Items :** Items can be added to or removed from the order while it is active, ensuring the items exist in the restaurant's menu.
- **Price Calculation :** Computes the total price of the order based on the items and their quantities.
- **Order Finalization :** Finalizes the order, applying any customer discounts and updating order frequencies for menu items.
- **Delivery Management :** Assigns a courier to the order and tracks the order's delivery status.

Key Methods

- `public Order(Restaurant restaurant, String name)` : Constructor initializing the order with a restaurant and a name.
- `public Order(Restaurant restaurant, String name, Customer customer)` : Constructor initializing the order with a restaurant, a name, and a customer.
- `public void addItem2Order(MenuItem item)` : Adds an item to the order, checking its availability in the restaurant's menu.
- `public void removeItemFromOrder(MenuItem item)` : Removes an item from the order, ensuring it exists in the current order.
- `public double getPrice()` : Computes and returns the total price of the order.
- `public void endOrder()` : Ends the order, making it unmodifiable, applying discounts, and updating item frequencies.
- `public void setCourier(Courier courier)` : Assigns a courier to the order and adds the order to the courier's board.
- `public void validateOrderByCourier()` : Validates the order by the courier, marking it as completed and updating relevant records.

Additional Details

- **Date Management :** Each order is timestamped with a date of creation.
- **Integration with Other Components :** The `Order` class interacts with the `Customer`, `Restaurant`, and `Courier` classes, as well as the `MyFoodora` system for maintaining records of completed orders.

Exception Handling

- `ItemNotInMenuException` : Thrown when attempting to add an item not available in the restaurant's menu.
- `ItemNotInOrderException` : Thrown when attempting to remove an item not present in the current order.

String Representation The `Order` class includes a `toString()` method that provides a comprehensive string representation of the order details, including the date, customer, restaurant, items, price, courier, and delivery address.

2.4 `core.policies` package

In MyFoodora, policies play a crucial role in shaping various aspects of the platform's operations, ranging from profit management to order allocation and sorting. These policies are implemented using the Strategy Pattern, allowing for dynamic selection and interchangeability.

2.4.1 Target Profit Policies

Target profit policies enable managers to strategize and meet predefined profit targets efficiently. MyFoodora supports three distinct target profit policies :

— **targetProfit_DeliveryCost :** This policy computes the delivery cost necessary to achieve a given target profit. It factors in the last month's total income (based on the number of completed orders), a specified service fee, and a markup percentage.

— **targetProfit_ServiceFee :** In contrast, this policy calculates the service fee required to meet the target profit. It considers the previous month's total income, a designated markup percentage, and the current delivery cost.

— **targetProfit_Markup :** This policy determines the markup percentage needed to attain the desired profit level. It takes into account the previous month's total income, a specific service fee, and the current delivery cost.

2.4.2 Delivery Policies

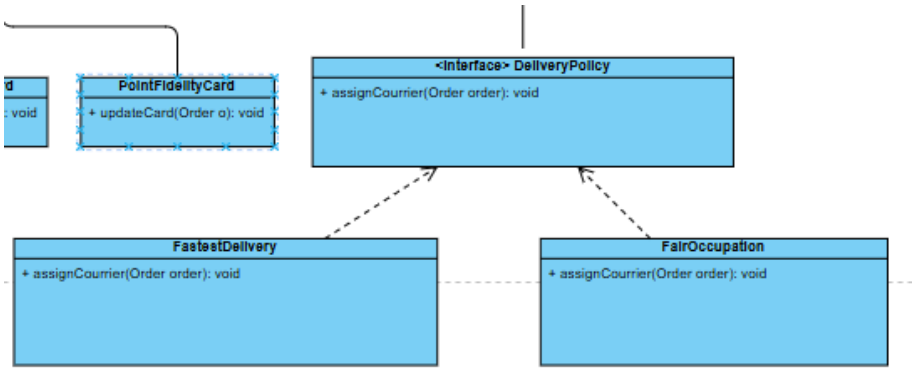


FIGURE 5 – Delivery classes UML

Delivery policies govern the allocation of orders to couriers, ensuring efficient and fair order distribution. MyFoodora implements two primary delivery policies :

— **Fastest Delivery :** This policy selects the courier with the shortest distance to cover for retrieving the order from the chosen restaurant and delivering it to the customer. It optimizes delivery time and enhances customer satisfaction.
— **Fair Occupation Delivery :** In this policy, the courier with the least number of completed deliveries is chosen for order allocation. This strategy promotes fairness among couriers and balances their workload.

2.4.3 Shipped Order Sorting Policies

Shipped order sorting policies facilitate the sorting of shipped orders based on specific criteria, enhancing accessibility and organization. MyFoodora offers the following sorting policies :
— **Most/Least Ordered Half-Meal :** This policy displays all half-meals sorted according to the number of shipped half-meals. It provides insights into popular half-meal choices among customers.
— **Most/Least Ordered Item à la Carte :** In this policy, all menu items are sorted based on the number of times they have been selected à la carte. It offers visibility into customer preferences for individual menu items.

The implementation of these policies using the Strategy Pattern ensures flexibility, scalability, and maintainability within the MyFoodora platform.

2.5 `core.fidelityCards` package

The Fidelity Cards subsystem within MyFoodora is designed to enhance customer engagement and loyalty through various incentives and rewards. Here's a detailed overview of each type of fidelity card supported by the system :

1. **Basic Fidelity Card** : Every user receives a Basic Fidelity Card by default upon registration with MyFoodora. This card provides access to special offers and promotions offered by participating restaurants. Customers can enjoy these perks without the need for any additional actions or points accumulation.
2. **Point Fidelity Card** : Customers have the option to upgrade to a Point Fidelity Card, which rewards them with points for each €10 spent on orders. These points accumulate over time, and once a customer reaches a threshold of 100 points, they become eligible for a 10% discount on their next order. This incentivizes customers to make repeat purchases and increases their loyalty to the platform.
3. **Lottery Fidelity Card** : For customers seeking a more exciting and unpredictable experience, MyFoodora offers the Lottery Fidelity Card. While holders of this card do not receive traditional discounts or points, they have the opportunity to win their meal for free each day. This adds an element of chance and excitement to their dining experience.

2.6 `core.Exceptions` package

This package contains the exceptions thrown by different methods of the system :

Exception	Description
`AccountDesactivatedException`	Exception thrown when attempting to access a deactivated account.
`FoodItemNotFoundException`	Exception raised when a food item is not found in the system.
`IncorrectIdentificationException`	Exception indicating incorrect identification information.
`InvalidItemDescription`	Exception thrown when creating a menu item with an invalid description.
`InvalidUserException`	Exception raised when an invalid user object is used as input to a method.
`ItemNotInMenuException`	Exception thrown when attempting to remove a non-existing item from a menu.
`ItemNotInOrderException`	Exception raised when an item is not found in an order.
`NoPlaceInMealException`	Exception indicating that there is no place in a meal to add additional items.
`OrderNotFoundException`	Exception thrown when an order is not found in the system.
`ProfitUnreachableException`	Exception indicating that a target profit is unreachable.
`SubscriberAlreadyExistsException`	Exception raised when attempting to add a subscriber that already exists.
`SubscriberNotFoundException`	Exception thrown when a subscriber is not found in the system.
`UserNotFoundException`	Exception indicating that a user is not found in the system.

2.7 `core.comparators` Package

This package contains classes that implement the `Comparator` interface to define custom comparison logic for specific types of objects.

— `MenuItemComparator` : Compares `MenuItem` objects based on their `orderFrequency` attribute.

— `RestaurantComparator` : Compares `Restaurant` objects based on the number of delivered orders.

2.8 `core.enums` Package

This package contains enums that define fixed sets of constants for specific types of attributes.

— `DishCategory` : Represents categories of dishes, such as main dishes, starters, and desserts.
— `FoodType` : Represents types of food, such as vegetarian or standard.
— `MealSize` : Represents the sizes of meals, such as half meals or full meals.

2.9 Testing Package

In the `test` package, JUnit tests are meticulously crafted to validate both core and interface functionalities of the application, by testing every imporant method of each class. These tests are designed to assess various aspects of the software, including its core logic, data processing algorithms, user interface interactions, and overall system behavior. By employing JUnit, the test suite ensures that each component of the system performs as expected, producing reliable results and adhering to specified requirements.

2.10 The `MyFoodora` Class

The `MyFoodora` class serves as the central hub for managing the entire MyFoodora system. It implements the `SubscriberObservable` interface to handle customer subscriptions for special offers. This class encapsulates various functionalities, including user management, order processing, policy application, and notification systems. Here are the key elements and their descriptions :

— **Attributes :**
  — `userLastID`: Stores the last ID value of users between save and load operations.
  — `users`: An `ArrayList` of `User` objects, representing all users in the system.
  — `hashedPasswords`: A `HashMap` mapping user IDs to their hashed passwords.
  — `completedOrders`: An `ArrayList` of `Order` objects, containing all the completed orders in the system.
  — `deliveryPolicy`: A `DeliveryPolicy` object that dictates how deliveries are handled.
  — `profitPolicy`: A `TargetProfitPolicy` object that determines the profit calculation strategy.
  — `subscribedCustomers`: An `ArrayList` of `SubscriberObserver` objects, representing customers subscribed to notifications.
  — `markupPercentage, deliveryCost, serviceFee`: Double values representing the financial parameters of the system.

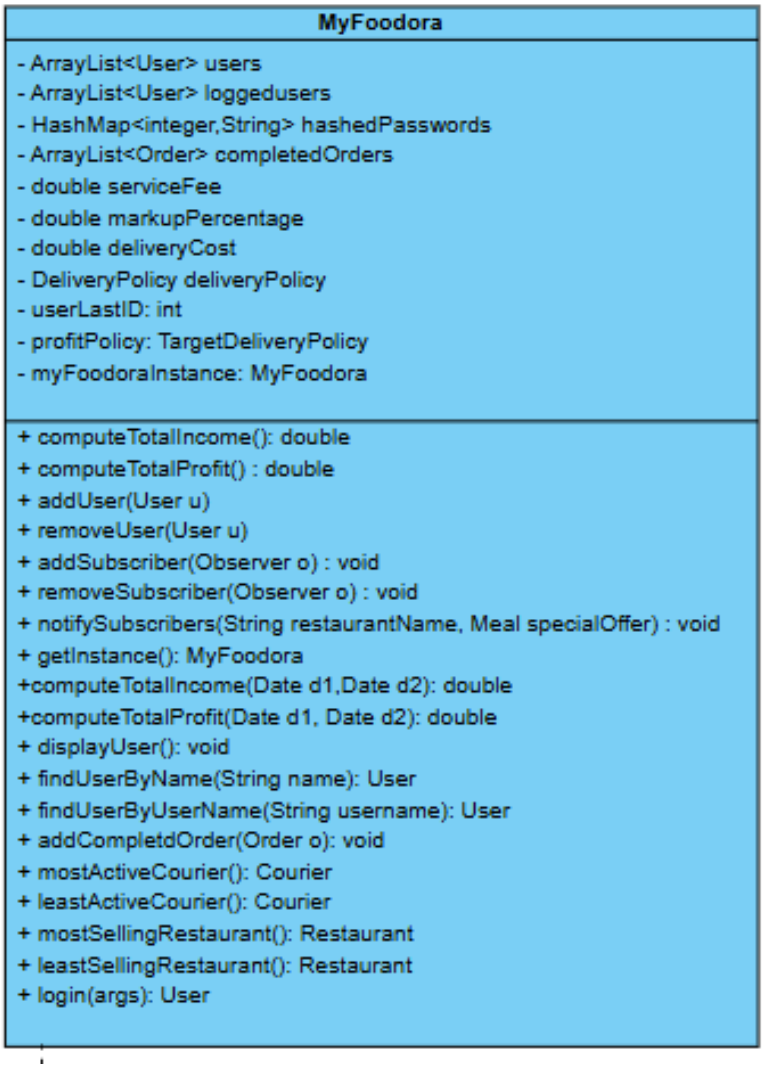


FIGURE 6 – Classe MyFoodora UML

- `myFoodoraInstance`: A static instance of `MyFoodora`, ensuring the singleton pattern.

— **Methods :**

- `getInstance()`: Returns the singleton instance of `MyFoodora`.
- `addUser(User u)`: Adds a user to the system and stores their hashed password.
- `removeUser(User u)`: Removes a user from the system and their hashed password, throwing `UserNotFoundException` if the user doesn't exist.
- `computeTotalIncome()`: Computes the total income from all completed orders.
- `computeTotalIncome(Date date1, Date date2)`: Computes the total

income within a specified date range.
- `computeTotalProfit()`: Computes the total profit using the markup percentage, service fee, and delivery cost.
- `computeTotalProfit(Date date1, Date date2)`: Computes the total profit within a specified date range.
- `setMarkupPercentage(double markupPercentage)`: Sets the markup percentage.
- `setDeliveryCost(double deliveryCost)`: Sets the delivery cost.
- `setServiceFee(double serviceFee)`: Sets the service fee.
- `displayUsers()`: Prints the list of users to the console.
- `findUserByName(String name)`: Finds a user by their name, throwing `UserNotFoundException` if the user doesn't exist.
- `findUserByUserName(String username)`: Finds a user by their username, throwing `UserNotFoundException` if the user doesn't exist.
- `addCompletedOrder(Order order)`: Adds a completed order to the list.
- `setDeliveryPolicy(DeliveryPolicy deliveryPolicy)`: Sets the delivery policy.
- `setProfitPolicy(TargetProfitPolicy profitPolicy)`: Sets the profit policy.
- `addSubscriber(SubscriberObserver o)`: Adds a subscriber to the notification list, throwing `SubscriberAlreadyExistsException` if the subscriber already exists.
- `removeSubscriber(SubscriberObserver o)`: Removes a subscriber from the notification list, throwing `SubscriberNotFoundException` if the subscriber doesn't exist.
- `notifySubscribers(String restaurantName, Meal specialOffer)`: Notifies all subscribers of a new special offer.
- `mostActiveCourier()`: Calculates and returns the courier with the most completed deliveries.
- `leastActiveCourier()`: Calculates and returns the courier with the fewest completed deliveries.
- `mostSellingRestaurant()`: Calculates and returns the restaurant with the most completed orders.
- `leastSellingRestaurant()`: Calculates and returns the restaurant with the fewest completed orders.
- `login(String username, String password)`: Authenticates a user based on their username and password, throwing `IncorrectIdentificationException` or `AccountDesactivatedException` for invalid credentials or deactivated accounts, respectively.

The `MyFoodora` class is designed to encapsulate all the core functionalities required for managing the MyFoodora system. It supports user authentication, order processing, financial calculations, policy management, and customer notifications.

The singleton pattern ensures that only one instance of the class is created, maintaining a consistent state across the application.

3 Design Patterns

3.1 Strategy Pattern

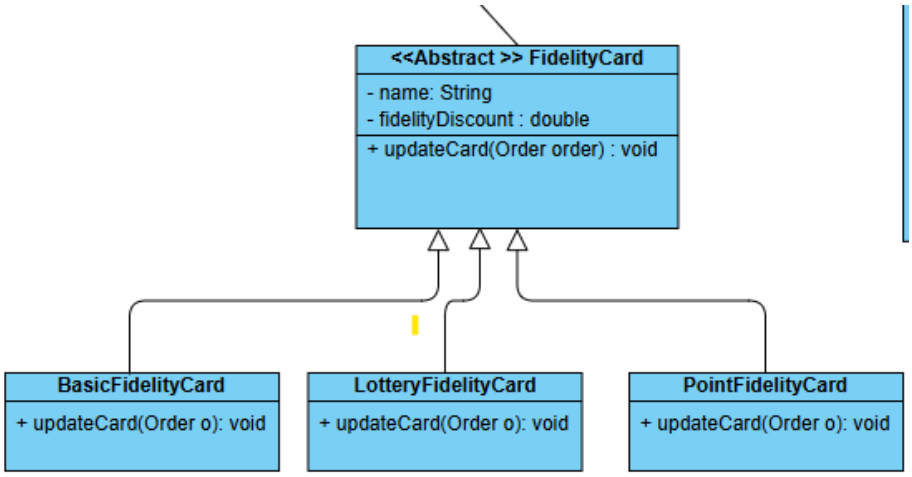


FIGURE 7 – Fidelity Card Strategy Pattern UMl

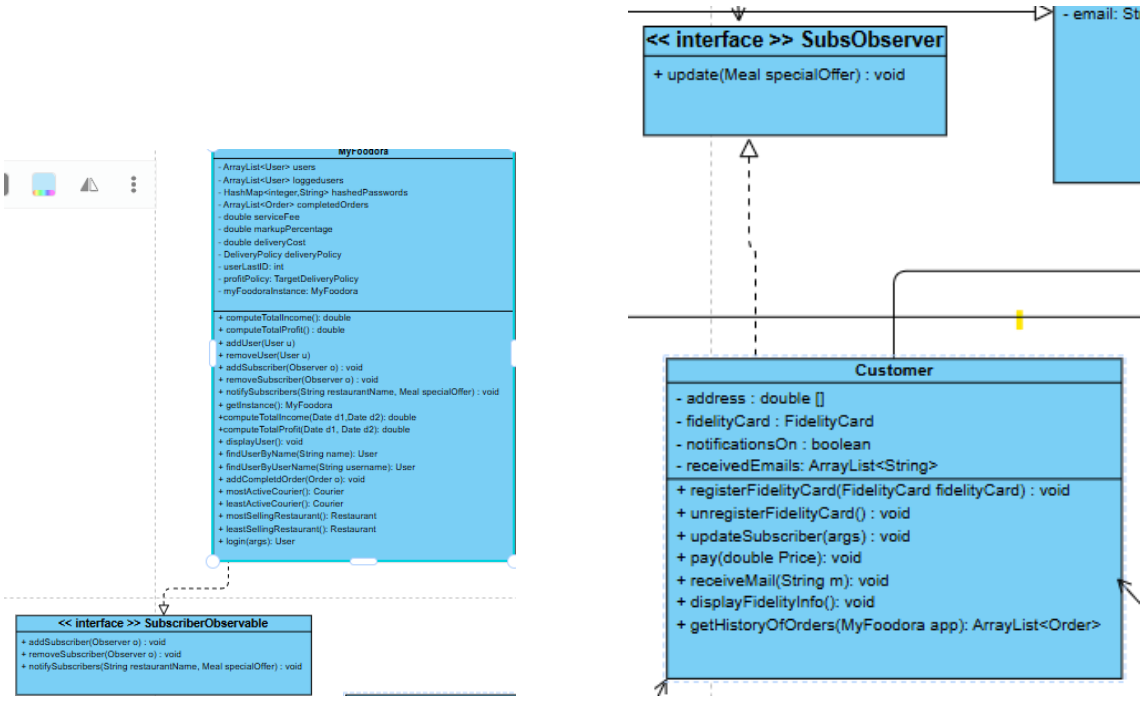
The Strategy pattern is used to define a family of algorithms, encapsulate each one, and make them interchangeable. In MyFoodora, we use this pattern for delivery and profit calculation policies.

— **DeliveryPolicy** : An interface that defines the method for assigning a courier to an order.
— **FastestDeliveryPolicy** : A class that implements DeliveryPolicy, assigning the closest available courier.
— **FairDeliveryPolicy** : A class that implements DeliveryPolicy, assigning the courier with the fewest deliveries.
— **ProfitCalculationPolicy** : An interface that defines the method for calculating profit.
— **TargetProfitPolicy** : A class that implements ProfitCalculationPolicy, adjusting delivery fees to meet a target profit.
— **ServiceFeePolicy** : A class that implements ProfitCalculationPolicy, adjusting service fees to optimize profit.

3.2 Observer Pattern

The Observer pattern is used when there is a one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. In MyFoodora, we use this pattern to notify customers about special offers.

— **Observable** : An interface or abstract class that represents the entity being observed (e.g., a Restaurant).
— **Observer** : An interface that represents the observers (e.g., Customers) who need to be notified of changes.
— **Restaurant** : Implements Observable to notify customers about special offers.
— **Customer** : Implements Observer to receive notifications about special offers.



(a) Observable UML (b) Observer UML

FIGURE 8 – UML Diagrmas for Observable and Observer

3.3 Singleton Pattern

The Singleton pattern ensures that a class has only one instance and provides a global point of access to it. In MyFoodora, the main system class, MyFoodora, is implemented as a singleton.

— **MyFoodora** : The singleton class that represents the entire system. It ensures that there is only one instance of MyFoodora throughout the application.

4 Command-Line Interface (CLI)

The Command-Line Interface (CLI) serves as the primary interaction point between users and the MyFoodora system. It employs a tokenizer to parse user input and execute corresponding commands. Below is a non exhaustive list of available commands along with their descriptions, implemented in `MyFoodoraClient.java` module :

— **login <username> <password> :** Allows a user to perform login authentication. A MyFoodora manager user with the username "ceo" and password "123456789" is assumed to exist.
— **logout <> :** Enables the currently logged-in user to log off from the system.
— **registerRestaurant <name> <address> <username> <password> :** Allows the currently logged-in manager to add a restaurant with the given name, address (bi-dimensional coordinate), username, and password to the system.
— **registerCustomer <firstName> <lastName> <username> <address> <password> :** Permits the currently logged-in manager to add a customer to the system, providing the first name, last name, username, address, and password.
— **registerCourier <firstName> <lastName> <username> <position> <password> :** Enables the currently logged-in manager to add a courier to the system, specifying the first name, last name, username, position, and password. By default, newly registered couriers are considered on-duty.

Users can interact with the CLI by entering commands followed by their respective parameters. The CLI offers a comprehensive array of functionalities for managing restaurants, customers, couriers, orders, and system configurations.

5 Test Scenario

Our test scenario for shortage of time is quite simple, it does the following : 1. registerManager managerName manager1 manager1 : Registers a manager with the name managerName and the username and password manager1. This command creates a new manager account in the system.

2. registerCustomer customerName customerLastName customerUsername customerPass taboun@gmail.fr 0661674121 2,6 : Registers a customer with the provided details, including their name, last name, username, password, email, phone number, and loyalty card information. This command creates a new customer account in the system.

3. y : This command answers the prompt : "do you want to have notifications on ?".

4. registerRestaurant restaurant_name restaurantUsername restaurantPass 12,15 : Registers a restaurant with the given name, username, password, and delivery fee range. This command creates a new restaurant account in the system.

5.login restaurantUsername restaurantPass : Logs into the system using the provided restaurant username and password.

6. logout : Logs out of the currently logged-in account, terminating the session.

7. login customerUsername customerPass : Logs into the system using the provided customer username and password.

8. createOrder restaurant_name : Initiates the process of creating a new order with the specified restaurant. This command would likely prompt further steps to complete the order.

9.logout : Logs out of the currently logged-in account, terminating the session.

10. login manager1 manager1 : Logs into the system using the provided manager username and password.

11. showCustomers : Displays a list of customers registered in the system. This command shows details such as customer names, usernames, and other relevant information.

12. logout : Logs out of the currently logged-in account, terminating the session.

close : Closes or exits the system, ending the simulation or terminating the application.

6 Workload split between group members

Group Member	**Tasks Done**
Yassine	— Food Classes and their tests — Restaurant and Menu classes and their tests — parts of Manager and MyFoodora — TargetProfit policies and their tests — Fidelity cards and their tests — Command Line Interface (CLI) — CLI test scenario — System Singleton pattern
Farouk	— Order Class and its tests — Delivery and OrderSorting Policies and their tests — Subscriber Observer Pattern — parts of Manager and MyFoodora — Comparators and their tests — CLI

TABLE 1 – Workload distribution among group members

7 Advantages and Disadvantages of Our Solution

Our solution comes with several notable advantages and some disadvantages, which are detailed below :

7.1 Advantages

— **Adherence to the Open/Closed Principle :** Our design respects the Open/Closed Principle to the maximum extent possible. The system is designed to be easily extensible without modifying existing code, allowing new features to be added with minimal impact on the existing system.
— **Good Modularity :** The solution is highly modular, with clear separation of concerns across different components. This modularity enhances maintainability, readability, and testability of the codebase.
— **Singleton Pattern for Core Management :** The use of the Singleton pattern in the 'MyFoodora' class ensures that there is a single point of control for the system's core operations, which simplifies resource management and state consistency.
— **Flexible Command-Line Interface (CLI) :** The CLI implementation is flexible and robust, utilizing a tokenizer to parse and execute commands. This makes it easy to add new commands and functionalities without significant refactoring.
— **Observer Pattern for Notifications :** The use of the Observer pattern for notifying customers about special offers ensures that updates are efficiently communicated to all subscribed customers, improving user engagement and satisfaction.
— **Comprehensive Policy Management :** The system includes a robust framework for managing different policies, such as delivery and profit policies. This flexibility allows the system to adapt to varying business requirements and optimize operations.
— **Encapsulation of Business Logic :** The business logic is well-encapsulated within the core classes, such as 'MyFoodora', 'Order', and 'User'. This encapsulation ensures that business rules are consistently applied and makes it easier to update the logic when requirements change.
— **Extensible Meal and Order Management :** The design allows for easy extension and management of meals and orders, enabling restaurants to efficiently manage their offerings and customers to conveniently place and track orders.
— **Clear Role-Based Access Control :** The system implements clear role-based access control, ensuring that only authorized users can perform specific actions, which enhances security and data integrity.

7.2 Disadvantages

— **Lack of MVC Pattern :** One of the main disadvantages is that the interface does not utilize the Model-View-Controller (MVC) pattern. This can make the code less intuitive and harder to manage, especially as the system grows and requires more complex interactions between components.
— **Error Handling and Validation :** While there is some error handling, it could be more comprehensive. For instance, many commands do not fully validate inputs or handle edge cases gracefully, which could lead to unexpected behavior or crashes.
— **Scalability Issues :** The current design might face scalability issues as the system grows. For example, storing all users and orders in memory could become a bottleneck, and a more scalable solution would involve persistent storage and efficient querying mechanisms.
— **Single Responsibility Principle Violations :** Some classes, especially the 'MyFoodora' class, may have too many responsibilities. This violates the Single Responsibility Principle (SRP), making the system harder to maintain and extend.
— **Testing and Debugging Challenges :** The tight coupling between certain components can make testing and debugging more challenging. More reliance on dependency injection and interface-based design could improve testability.
— **Limited Documentation and Comments :** The solution lacks thorough documentation and comments, which are essential for understanding the system's workings and for onboarding new developers.

8 Conclusion

In conclusion, our solution for managing the MyFoodora system presents a robust framework that emphasizes modularity, extensibility, and adherence to key software design principles. The implementation showcases the effective use of design patterns such as Singleton and Observer, which contribute to a well-organized and maintainable codebase. The Command-Line Interface (CLI) provides a powerful tool for interacting with the system, making it easy to add new functionalities without extensive modifications to existing code.

However, there are areas for improvement, particularly in adopting the Model-View-Controller (MVC) pattern to enhance the separation of concerns and improve interface management. Addressing error handling and validation comprehensively will also be crucial to ensure the system's robustness. Furthermore, scalability considerations and adherence to the Single Responsibility Principle (SRP) should be taken into account to prepare the system for future growth and complexity.

Overall, the solution lays a strong foundation for MyFoodora, balancing flexibility and control while highlighting the importance of maintaining a clear and

organized architecture. Future iterations should focus on refining these aspects to achieve an even more resilient and efficient system.

9 Additional Documentation

Detailed API documentation and further explanations about the system architecture can be found at the provided GitHub repository link.