MyFoodora

TAP, ORDER, EAT.

MyFoodora - Food Delivery System

Project report

CentraleSupélec

Alami Nabil

ETUNUM : 2200762

Elias Al Bouzidi

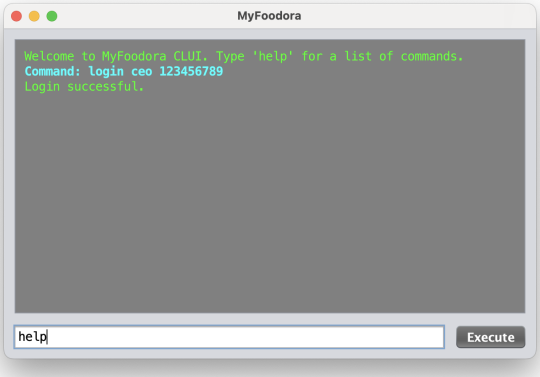
ETUNUM : 2200761

Table des matières

1 Introduction

This report presents the development of MyFoodora system, similar to popular apps such as UBER and DELIVEROO, a Java framework designed to manage a food delivery system inspired by successful implementations of popular food delivery services. The project is divided into two parts : Part 1 focuses on developing the core infrastructure, while Part 2 is dedicated to designing and developing a command line user-interface for the MyFoodora system.

The primary objective of this project is to create a Java framework, MyFoodora, that meets the requirements for managing a food delivery system. The framework will support essential functionalities such as restaurant and menu management, user registration and interaction, and order processing. Furthermore, a user-friendly interface will be designed to enhance the user experience and facilitate seamless interaction with the MyFoodora system, as shown below.



2 MyFoodora System

The first part of this project is dedicated to designing and coding the core structure of the system, which encompasses key requirements detailed in the subsections bellow.

2.1 Myfoodora

The `MyFoodoraSystem` package serves as the central component of the MyFoodora application. It manages users, orders, delivery policies, and computes various statistics related to the operation of the food delivery system.

— `restaurants` : A list of all registered restaurants.

— `customers` : A list of all registered customers.

— `couriers` : A list of all registered couriers.

— `orders` : A list of all placed orders.

— `incompleteOrders` : A list of orders that are not yet completed.

— `managers` : A list of all registered managers.

— `users` : A list of all registered users.

— `serviceFee` : The service fee percentage charged to customers.

— `markupPercentage` : The markup percentage added to the total price of orders.

— `deliveryCost` : The delivery cost charged to customers.

— `deliveryPolicy` : The current delivery policy used for assigning couriers to orders.

— `shippedOrderSortingPolicy` : the current shipped order sorting policy to display the most ordered meals/items.

— `profit` : The list of profits earned from completed orders.

It's methods are :

— `addUser(User user)` : Adds a new user to the system.

— `login(String username, String password)` : Logs in a user with the provided username and password.

— `removeUser(User user)` : Removes a user from the system.

— `getUsers()` : Returns a list of all registered users.

— `addIncompleteOrder(Order order)` : Adds an incomplete order to the system.

— `updateIncompleteOrders(Order order)` : Updates an incomplete order in the system.

— `getIncompleteOrders()` : Returns a list of incomplete orders.

— `getServiceFee()` : Returns the current service fee percentage.

— `getMarkupPercentage()` : Returns the current markup percentage.

— `getDeliveryCost()` : Returns the current delivery cost.

— `getProfit()` : Returns a list of profits earned from completed orders.

— `getDeliveryPolicy()` : Returns the current delivery policy.

— `setServiceFee(double serviceFee)` : Sets the service fee percentage.

— `setMarkupPercentage(double markupPercentage)` : Sets the markup percentage.

— `setDeliveryCost(double deliveryCost)` : Sets the delivery cost.

— `getNumberOfOrdersPeriod(Time startTime, Time endTime)` : Returns the number of orders placed within a specified time period.

— `getNumberOfOrdersForMonth(int year, int month)` : Returns the number of orders placed in a specified month.

— `getNumberOfOrdersForYear(int year)` : Returns the number of orders placed in a specified year.

— `getNumberOfOrdersLastMonth()` : Returns the number of orders placed in the previous month.

— `computeTotalIncome()` : Computes the total income earned from all orders.

— `computeIncomePeriod(Time startTime, Time endTime)` : Computes the income earned within a specified time period.

— `computeIncomeForMonth(int year, int month)` : Computes the income earned in a specified month.

— `computeIncomeForYear(int year)` : Computes the income earned in a specified year.

— `computeIncomeLastMonth()` : Computes the income earned in the previous month.

— `computeTotalProfit()` : Computes the total profit earned from completed orders.

— `calculateProfitPeriod(Time startTime, Time endTime)` : Calculates the profit earned within a specified time period.

— `calculateProfitForYear(int year)` : Calculates the profit earned in a specified year.

— `calculateProfitForMonth(int year, int month)` : Calculates the profit earned in a specified month.

— `calculateProfitLastMonth()` : Calculates the profit earned in the previous month.

— `computeAverageIncomePerCustomerPeriod(Time startTime, Time endTime)` : Computes the average income per customer within a specified time period.

— `computeAverageIncomePerCustomer()` : Computes the average income per customer.

— `getMostSellingRestaurant()` : Returns the restaurant with the most sales.

— `getLeastSellingRestaurant()` : Returns the restaurant with the least sales.

— `determineMostActiveCourier()` : Determines the most active courier.

— `determineLeastActiveCourier()` : Determines the least active courier.

— `setCurrentDeliveryPolicy(DeliveryPolicy policy)` : Sets the current delivery policy.

— `placeOrder(Order order)` : Places an order in the system.

— `getOrders()` : Returns a list of all placed orders.

— `getRestaurants()` : Returns a list of all registered restaurants.

— `getCustomers()` : Returns a list of all registered customers.

— `getCouriers()` : Returns a list of all registered couriers.

— `getManagers()` : Returns a list of all registered managers.

— `setDeliveryPolicy(DeliveryPolicy deliveryPolicy)` : Sets the delivery policy.

— `getHistory(User user)` : Returns the order history for a specified user.

Fig. 1 – UML diagram of MyFoorodra Class

2.2 User

The User package contains all the user involoved in the myfoodora application ;

2.2.1 Courier

The `Courier` class represents a courier in the MyFoodora system. Couriers are responsible for delivering orders to customers.

Attributes :

— `name` : The first name of the courier.

— `surname` : The last name of the courier.

— `id` : The unique identifier of the courier.

— `username` : The username used by the courier for authentication.

— `password` : The password used by the courier for authentication.

— `position` : The current position of the courier.

— `phoneNumber` : The phone number of the courier.

— `deliveredOrders` : The number of orders delivered by the courier.

— `onDuty` : A boolean indicating whether the courier is currently on duty.

Methods :

— `setOnDuty(boolean onDuty)` : Sets the duty status of the courier.

— `setPosition(Position newPosition)` : Sets the position of the courier.

— `acceptDelivery()` : Marks a delivery as accepted by the courier.

— `refuseDelivery()` : Logic for refusing a delivery (not implemented).

— `isOnDuty()` : Checks if the courier is currently on duty.

— `getDeliveredOrders()` : Returns the number of orders delivered by the courier.

— `getSurname()` : Returns the surname of the courier.

— `getPosition()` : Returns the current position of the courier.

— `getPhoneNumber()` : Returns the phone number of the courier.

Additional Notes :

— The `toString()` method is overridden to provide a string representation of the courier object, including name, surname, ID, phone number, delivered orders, and duty status.

2.2.2 Customer

The `Customer` class represents a customer in the MyFoodora system. Customers can place orders, manage their profiles, and receive notifications about special offers.

Attributes :

— `name` : The first name of the customer.

— `surname` : The last name of the customer.

— `id` : The unique identifier of the customer.

— `username` : The username used by the customer for authentication.

— `password` : The password used by the customer for authentication.

— `address` : The address of the customer.

— `email` : The email address of the customer.

— `phoneNumber` : The phone number of the customer.

— `fidelityCard` : The fidelity card associated with the customer.

— `notifySpecialOffers` : A boolean indicating whether the customer has consented to receive notifications about special offers.

— `notifications` : A list of all notifications received by the customer.

— `unreadNotifications` : A list of unread notifications.

Methods :

— `getAddress()` : Returns the address of the customer.

— `unregisterFidelityCard()` : Unregisters the fidelity card associated with the customer. It is set to basic fidelity card.

— `getFidelityCard()` : Returns the fidelity card associated with the customer.

— `getPoints()` : Returns the number of points accumulated by the customer (if applicable).

Additional Notes :

— The `update(String message)` method is implemented from the `Observer` interface and is used to notify the customer about special offers.

— The `toString()` method is overridden to provide a string representation of the customer object, including name, surname, ID, email, phone number, address, and special offers notification status.

2.2.3 Manager

The `Manager` class represents a manager in the MyFoodora system. Managers are responsible for overseeing the system operations, including managing users, setting financial parameters, and monitoring performance metrics.

Attributes :

— `name` : The first name of the manager.

— `surname` : The last name of the manager.

— `id` : The unique identifier of the manager.

— `username` : The username used by the manager for authentication.

— `password` : The password used by the manager for authentication.

— `myfoodorasystem` : A reference to the `MyFoodoraSystem` instance.

Methods :

— `addUser(User user)` : Adds a new user to the MyFoodora system.

— `removeUser(User user)` : Removes an existing user from the MyFoodora system.

— `activateUser(User user)` : Activates a user account in the MyFoodora system.

— `deactivateUser(User user)` : Deactivates a user account in the MyFoodora system.

— `setServiceFee(double serviceFee)` : Sets the service fee for the MyFoodora system.

— `setMarkupPercentage(double markupPercentage)` : Sets the markup percentage for the MyFoodora system.

— `setDeliveryCost(double deliveryCost)` : Sets the delivery cost for the MyFoodora system.

— `computeTotalIncome()` : Computes the total income of the MyFoodora system.

— `computeTotalProfit()` : Computes the total profit of the MyFoodora system.

— `computeAverageIncomePerCustomer()` : Computes the average income per customer.

— `determineMostActiveCourier()` : Determines the most active courier in the MyFoodora system.

— `determineLeastActiveCourier()` : Determines the least active courier in the MyFoodora system.

— `setCurrentDeliveryPolicy(DeliveryPolicy policy)` : Sets the current delivery policy for the MyFoodora system.

— `getMostSellingRestaurant()` : Returns the most selling restaurant in the MyFoodora system.

— `getLeastSellingRestaurant()` : Returns the least selling restaurant in the MyFoodora system.

— `showOrderedShippedOrder()` : This method prints the sorted list of name of restaurant and item/meal ordered and the number of shipping based on the policy, see ShippedOrderSortingPolicy

Additional Notes :

— The `Manager` class provides static methods to retrieve lists of restaurants, customers, couriers, and orders from the `MyFoodoraSystem`.

— The manager can manage various aspects of the MyFoodora system, including financial parameters and user activity.

— The `toString()` method is inherited from the `User` class and is not overridden in the `Manager` class.

2.2.4 Restaurant

The `Restaurant` class represents a restaurant within the MyFoodora system. Restaurants manage their location, menu, and meals, and they can set discount factors and notify customers of special offers.

Attributes :

— `location` : The location of the restaurant, represented by a `Position` object.

— `menu` : The menu of the restaurant, represented by a `Menu` object.

— `meals` : A list of meals offered by the restaurant.

— `menuFactory` : A factory object for creating menu items.

— `mealFactory` : A factory object for creating meals.

— `genericDiscountFactor` : The discount factor for generic meals.

— `specialDiscountFactor` : The discount factor for the special meal of the week.

— `mealOfTheWeek` : The current meal of the week.

— `myFoodoraSystem` : A reference to the `MyFoodoraSystem` instance.

Methods :

— `addMenuItem(String type, String name, double price, boolean isVegetarian, boolean isGlutenFree)` : Adds a menu item to the restaurant's menu.

— `createMeal(String name, String mealType, MenuItem... items)` : Creates a meal and adds it to the restaurant's list of meals.

— `removeMenuItem(String name)` : Removes a menu item by name from the restaurant's menu.

— `removeMenuItem(MenuItem item)` : Removes a specific menu item from the restaurant's menu.

— `removeMeal(Meal meal)` : Removes a meal from the restaurant's list of meals.

— `setGenericDiscountFactor(double discountFactor)` : Sets the generic discount factor and updates all meals with this factor.

— `setSpecialDiscountFactor(double discountFactor)` : Sets the special discount factor and updates the meal of the week.

— `setMealOfTheWeek(Meal meal)` : Sets the meal of the week and notifies observers.

— `getMealOfTheWeek()` : Returns the current meal of the week.

— `getMenuItems()` : Returns a list of all menu items.

— `getMeals()` : Returns a list of all meals.

— `getGenericDiscountFactor()` : Returns the generic discount factor.

— `getSpecialDiscountFactor()` : Returns the special discount factor.

— `printMenu()` : Prints the restaurant's menu to the console.

— `printMeals()` : Prints the restaurant's meals to the console.

— `printMealOfTheWeek()` : Prints the meal of the week to the console.

— `notifyObservers(String offer)` : Notifies customers who have opted in for special offers about the current offer.

— `toString()` : Returns a string representation of the restaurant, including its name, location, menu items, meals, and meal of the week.

— `showOrderedShippedOrder()` : This method prints the sorted list of name of restaurant and item/meal ordered and the number of shipping based on the policy, see ShippedOrderSortingPolicy

Additional Notes :

— The `Restaurant` class implements the `Observable` interface, allowing it to notify customers about special offers.

— The class uses factory objects (`MenuFactory` and `MealFactory`) to create menu items and meals.

— The discount factors allow the restaurant to offer special pricing for both generic and special meals (which can be updated dynamically).

— The `notifyObservers` method uses the MyFoodora system to retrieve customers who are interested in special offers and updates them with the current offer.

The user package contains also a `user` abstract class, that shares the common attributes of all the Myfoodora user. Additionaly, The user class has 2 other classes : `Observable, Observer` for notification management. For exemple each time a restaurant updates it's menu, the user receives a notification alert, which will be displaced when he loggs in.

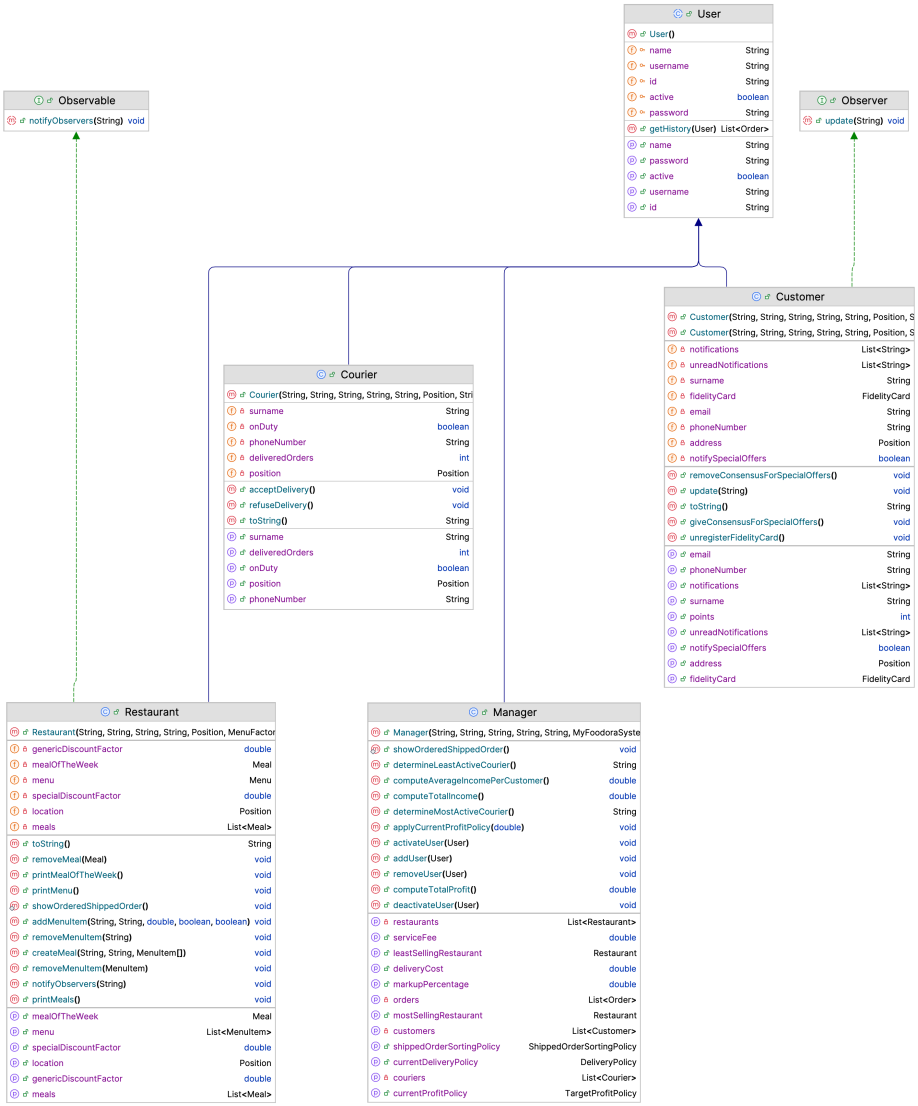


Fig. 2 – UML diagram of User Class

2.3 The Menu

The `Menu` package is responsible for managing and organizing the various menu items offered by a restaurant. It contains classes for managing menu items in the myFoodora application :

- `MenuItem` : Abstract class for menu items.
- `Menu` : Manages collections of menu items.
- `MenuFactory` : Interface for creating menu items.
- `ConcreteMenuFactory` : Implements MenuFactory for creating specific menu items.
- `Starter, MainDish, Dessert` : Classes representing different types of menu items.

Details of the classes are mentionned below :

2.3.1 Menu

— `List<MenuItem> starters` : A list to store all starter items.

— `List<MenuItem> mainDishes` : A list to store all main dish items.

— `List<MenuItem> desserts` : A list to store all dessert items.

Constructor

— `Menu()` : Initializes the `starters`, `mainDishes`, and `desserts` lists.

Methods

— `void addMenuItem(MenuItem item)` : Adds a menu item to the appropriate list based on its type (starter, main dish, or dessert).

— `List<MenuItem> getStarters()` : Returns the list of starters.

— `List<MenuItem> getMainDishes()` : Returns the list of main dishes.

— `List<MenuItem> getDesserts()` : Returns the list of desserts.

— `List<MenuItem> getAllItems()` : Returns a list of all menu items by combining the starters, main dishes, and desserts lists.

— `void printItems()` : Prints all the menu items categorized into starters, main dishes, and desserts.

Items are added to the menu through the `addMenuItem` method, and the items are categorized into starters, main dishes, and desserts. The `getAllItems` method provides a way to retrieve all items at once, while the `printItems` method is useful for displaying the menu in a readable format.

2.3.2 MenuItem

The `MenuItem` class is an abstract class that serves as a blueprint for items on a menu. This class encapsulates common properties and behaviors that menu items share.

Attributes :

— `name` : A `String` representing the name of the menu item.

— `price` : A `double` representing the price of the menu item.

— `isVegetarian` : A `boolean` indicating whether the menu item is vegetarian.

— `isGlutenFree` : A `boolean` indicating whether the menu item is gluten-free.

Constructor :

— The constructor initializes the `name`, `price`, `isVegetarian`, and `isGlutenFree` attributes.

Methods :

— `getName()` : Returns the name of the menu item.

— `getPrice()` : Returns the price of the menu item.

— `isVegetarian()` : Returns whether the menu item is vegetarian.

— `isGlutenFree()` : Returns whether the menu item is gluten-free.

— `toString()` : Provides a string representation of the menu item, formatted to display the name, price, and dietary information.

2.3.3 MenuFactory and ConcreteMenuFactory

The `MenuFactory` interface defines a contract for creating different types of menu items. It provides methods for creating starters, main dishes, and desserts with specified attributes such as name, price, and dietary information.

Methods :

— `createStarter(String name, double price, boolean isVegetarian, boolean isGlutenFree)` : Creates a starter menu item with the given attributes.

— `createMainDish(String name, double price, boolean isVegetarian, boolean isGlutenFree)` : Creates a main dish menu item with the given attributes.

— `createDessert(String name, double price, boolean isVegetarian, boolean isGlutenFree)` : Creates a dessert menu item with the given attributes.

The `ConcreteMenuFactory` class implements the `MenuFactory` interface, providing concrete implementations for creating different types of menu items. It overrides the methods defined in the interface to create instances of specific menu item types, such as starters, main dishes, and desserts, with the given attributes.

Methods :

— `createStarter(String name, double price, boolean isVegetarian, boolean isGlutenFree)` : Creates and returns a new instance of a starter menu item with the specified attributes.

— `createMainDish(String name, double price, boolean isVegetarian, boolean isGlutenFree)` : Creates and returns a new instance of a main dish menu item with the specified attributes.

— `createDessert(String name, double price, boolean isVegetarian, boolean isGlutenFree)` : Creates and returns a new instance of a dessert menu item with the specified attributes.

Finally, the following classes represent different types of menu items within the `myFoodora` system :

— **Dessert** : Represents a dessert menu item, inheriting attributes and methods from the `MenuItem` class.

— **Starter** : Represents a starter menu item, also inheriting attributes and methods from the `MenuItem` class.

— **MainDish** : Represents a main dish menu item, inheriting from the `MenuItem` class as well.

Each class has a constructor that initializes the menu item with a name, price, and information about whether it's vegetarian and/or gluten-free.

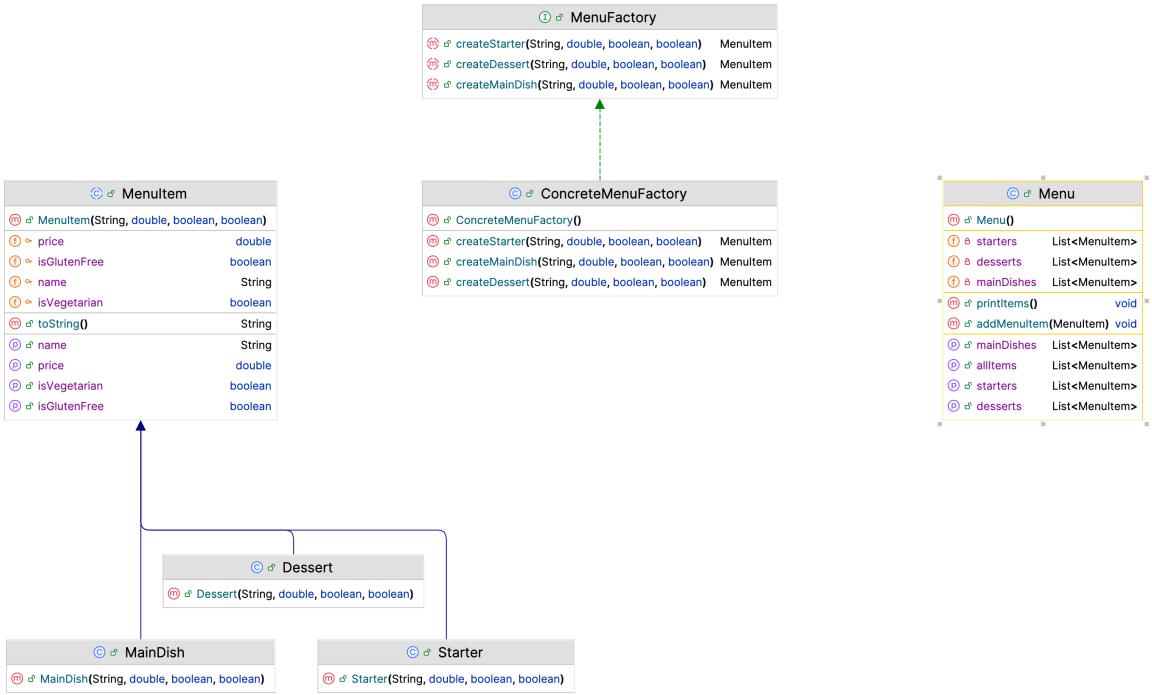


Fig. 3 – UML diagram of Menu Class

2.4 Meal

Each of Meal classes plays a role in managing meals within the myFoodora application. Here's a brief overview :

- `Meal` : Abstract class representing a meal composed of one or more menu items.
- `HalfMeal` : Represents a half meal consisting of two menu items. The only combinaisons allowed are : (starter - maindish) and (maindish - desert) with the same type (glutenfree and/or vegetarian)
- `FullMeal` : Represents a full meal consisting of three menu items.The only combinaison allowed is : (starter - maindish - desert) with the same type (glutenfree and/or vegetarian)
- `MealFactory` : Interface for creating different types of meals.
- `ConcreteMealFactory` : Implements MealFactory for creating specific types of meals.
- `SpecialMeal` : Represents a special meal, often with a discount, usually designated as the "meal of the week" in the application.

2.4.1 Meal

The Meal class represents a meal that consists of multiple menu items. Here's a breakdown of its key components :

1. **Name :** Represents the name of the meal.
2. **Items :** A list of menu items that compose the meal.
3. **Discount Factor :** Represents the discount factor applied to the total price of the meal.
4. **Vegetarian and/or Gluten-Free Flags :** Indicates whether the meal is vegetarian and/or gluten-free based on its constituent items.

Constructor :

— Initializes a 'Meal' object with a name, a list of items, and a discount factor.

— Determines the vegetarian and gluten-free status of the meal based on its first item.

Methods :

— `getName()`: Returns the name of the meal.

— `getPrice()`: Calculates and returns the total price of the meal after applying the discount factor.

— `getItems()`: Returns the list of items in the meal.

— `isVegetarian()`: Checks if the meal is vegetarian.

— `isGlutenFree()`: Checks if the meal is gluten-free.

— `setDiscountFactor(double discountFactor)`: Sets the discount factor of the meal.

— `setVegetarian(boolean vegetarian)`: Sets the vegetarian flag of the meal.

— `setGlutenFree(boolean glutenFree)`: Sets the gluten-free flag of the meal.

— `toString()`: Generates a string representation of the meal, including its items and total price after discount.

2.4.2 HalfMeal and FullMeal

The `HalfMeal` and `FullMeal` classes represent different types of meals offered by the system.

1. **Name :** The name of the meal.
2. **Items :** The menu items that constitute the meal.
3. **Discount Factor :** A fixed discount factor applied to the total price of the meal.

HalfMeal Constructor :

— Initializes a ‘HalfMeal‘ object with a name and two menu items.

— Sets the discount factor to 5

FullMeal Constructor :

— Initializes a ‘FullMeal‘ object with a name, starter, main dish, and dessert.

— Sets the discount factor to 5

The `SpecialMeal` class represents a special type of meal offered by the system.

2.4.3 ConcreteMealFactory and MealFactor

the ConcreteMealFactory class is a specific implementation of the MealFactory interface, providing the actual implementation for creating meals, while the MealFactory interface defines the contract that concrete factories must adhere to for creating meals.

Fig. 4 – UML diagram of Meal Class

2.5 Order

The Order class Represents an order made by a customer to a restaurant in the MyFoodora system.

Attributes :

— `name` : Name of the order.

— `customer` : Customer who placed the order.

— `restaurant` : Restaurant from which the order was placed.

— `items` : List of individual menu items included in the order.

— `meals` : List of meals included in the order.

— `totalPrice` : Total price of the order.

— `courier` : Courier responsible for delivering the order.

— `time` : Time when the order was made.

Methods :

— `addItem(MenuItem item)` : Adds a menu item to the order and updates the total price.

— `addMeal(Meal meal)` : Adds a meal to the order and updates the total price.

— `calculateTotalPrice()` : Calculates the total price of the order.

— `setTotalPrice(double price)` : Sets the total price of the order.

— Accessor methods to retrieve various attributes of the order.

— `toString()` : Generates a string representation of the order, including details such as the restaurant, customer, delivery time, total price, and items.

This class facilitates the management and tracking of orders within the MyFoodora system.

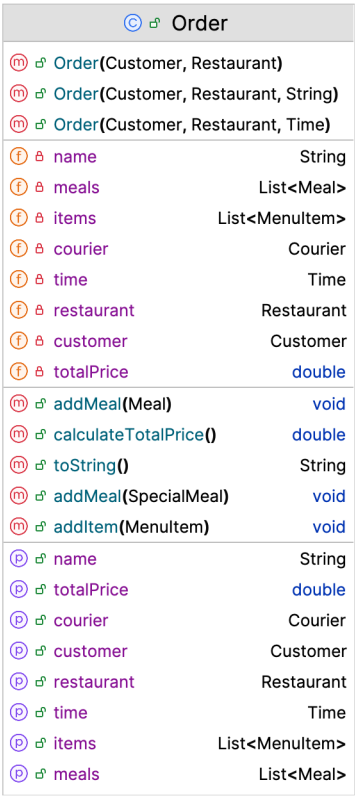


FIG. 5 – UML diagram of Order Class

2.6 Fidelity Card

The fidelity card package contains four classes that represent different types of fidelity cards within the MyFoodora system.

— `BasicFidelityCard` : Implements the `FidelityCard` interface but does not provide any discount, returning the total price unchanged.

— `LotteryFidelityCard` : Also implements the `FidelityCard` interface, providing a 1% chance for the customer to get the meal for free. Otherwise, it returns the total price unchanged.

— `PointFidelityCard` : Another implementation of the `FidelityCard` interface, this card accumulates points based on the total price of the orders. When the customer accumulates 100 points, they receive a 10% discount on their next order.

— `FidelityCard` : An interface defining the contract for fidelity card implementations, requiring them to implement the `applyDiscount` method to calculate the discount on the total price.

The implementation follows the Strategy Pattern, where the FidelityCard interface defines a common method, applyDiscount, and the concrete classes (BasicFidelityCard, LotteryFidelityCard, PointFidelityCard) provide specific discount strategies.

These classes allow for different fidelity card functionalities, such as basic no-discount cards, lottery-based discounts, and points-based discounts. They offer flexibility in providing benefits to loyal customers based on their preferences and behaviors.

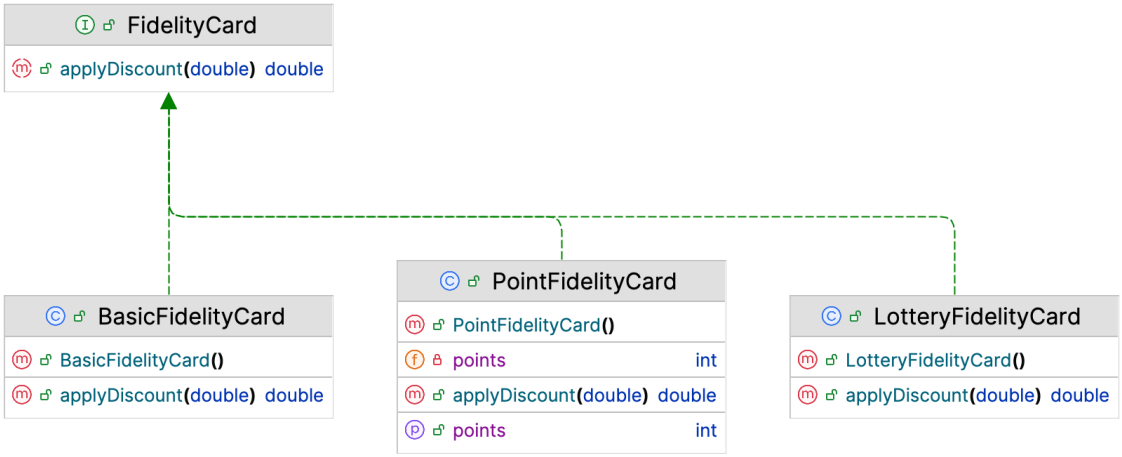


Fig. 6 – UML diagram of FidelityCard Class

2.7 DileveryPolicy

The delivery policy package contains three classes that represent different delivery policies used by the MyFoodora system.

— `FairOccupationDelivery` : Implements the `DeliveryPolicy` interface, which assigns the order to the courier with the least number of delivered orders, ensuring a fair distribution of orders among couriers. It retrieves the list of couriers from the `MyFoodoraSystem`, filters them to select only those on duty, and then determines the courier with the minimum number of delivered orders.

— `FastestDelivery` : Also implements the `DeliveryPolicy` interface, but this policy assigns the order to the courier who can reach the customer from the restaurant in the shortest amount of time. It calculates the distance between the restaurant and the customer for each courier, considering their current positions, and selects the courier with the minimal total distance.

— `DeliveryPolicy` : This is an interface defining the contract for delivery policy classes. It declares a single method `assignCourier()` that must be implemented by concrete delivery policy classes to assign a courier to an order.

These classes provide different strategies for assigning couriers to orders, allowing MyFoodora to optimize delivery efficiency based on factors such as fairness among couriers or minimizing delivery time.

These classes provide different strategies for assigning couriers to orders, allowing MyFoodora to optimize delivery efficiency based on factors such as fairness among couriers or minimizing delivery time.

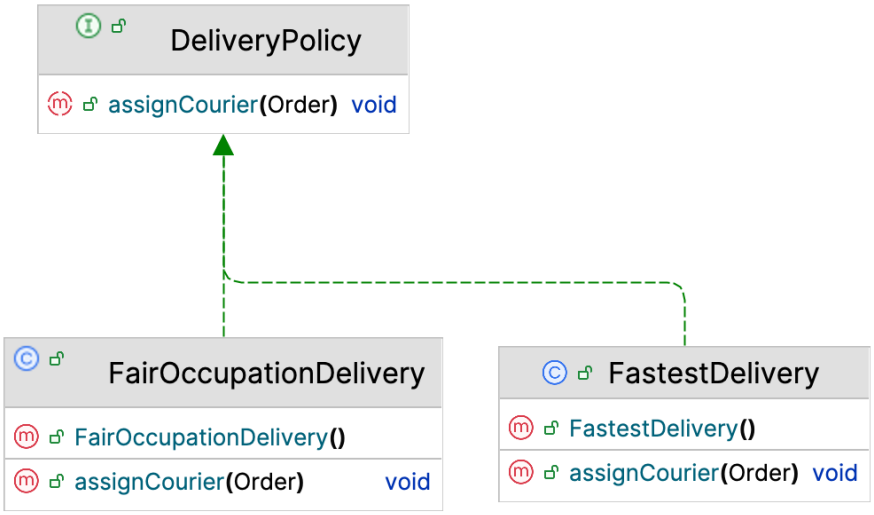


Fig. 7 – UML diagram of DileveryPolicy Class

2.8 Space-Time Coordinates

The space-time coordinates package contains two classes that handle position and time-related functionalities. They facilitate operations such as distance calculation, time manipulation, and temporal comparisons.

— `Position` : Represents a position in two-dimensional space with coordinates `x` and `y`. It provides a method `distanceTo()` to calculate the Euclidean distance between two positions and a `toString()` method for displaying position information.

— `Time` : Represents a specific point in time, including date and time components such as day, month, year, hour, minute, and second. It offers methods to retrieve the current time, calculate the start and end of the previous month, compare two time instances, and generate formatted string representations of time. Additionally, it includes methods to check if a time instance is before, after, or equal to another time instance.

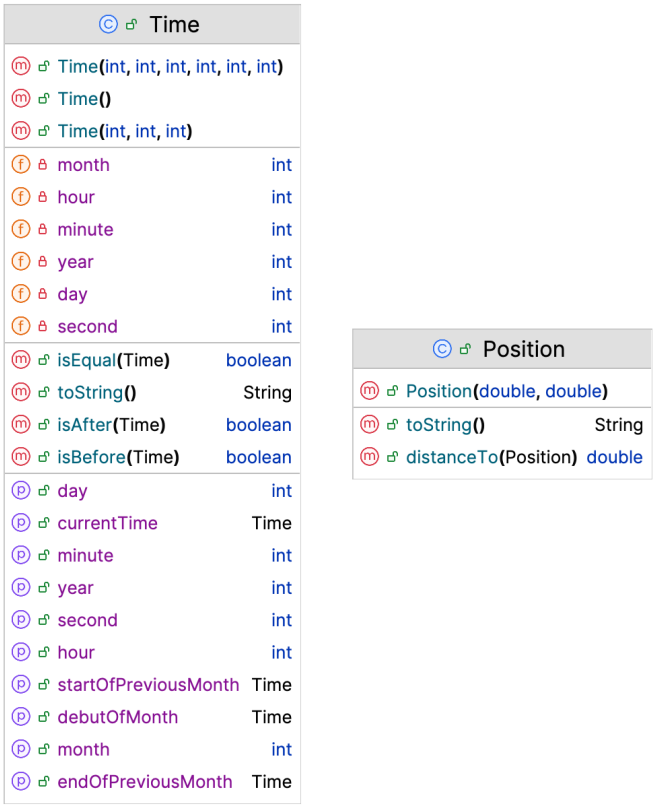


Fig. 8 – UML diagram of Space-Time Class

2.9 Target Profit Policy

The target profit policy package includes four classes responsible for applying different strategies to achieve a target profit within the MyFoodora system. Is is based on relevant statistics (total income of the last month, number of orders) and myFoodora parameters of the last month.
Based on those informations, the method apply allow the manager to automatically adjusts the myfoodora parameters depending on the TargetProfitPolicy to acheive a specific target profit for the next month.

— `TargetProfitDeliveryCost` : Implements the `TargetProfitPolicy` interface and applies a strategy to adjust the delivery cost based on the target profit. It calculates the delivery cost per order by considering the total income, service fee, markup percentage, and the desired target profit.

— `TargetProfitMarkup` : Also implements the `TargetProfitPolicy` interface and focuses on adjusting the markup percentage to reach the target profit. It computes the markup percentage by taking into account the total income, service fee, delivery cost, and the desired target profit.

— `TargetProfitServiceFee` : Another implementation of the `TargetProfitPolicy`

interface, this class modifies the service fee to achieve the target profit. It calculates the service fee per order by considering the total income, markup percentage, delivery cost, and the desired target profit.

— `TargetProfitPolicy` : An interface defining a common method `apply()` that must be implemented by all target profit policy classes. This method allows applying the target profit strategy to the MyFoodora system.

These classes provide flexibility in adjusting various parameters such as delivery cost, markup percentage, and service fee to meet the target profit goals of the MyFoodora system.

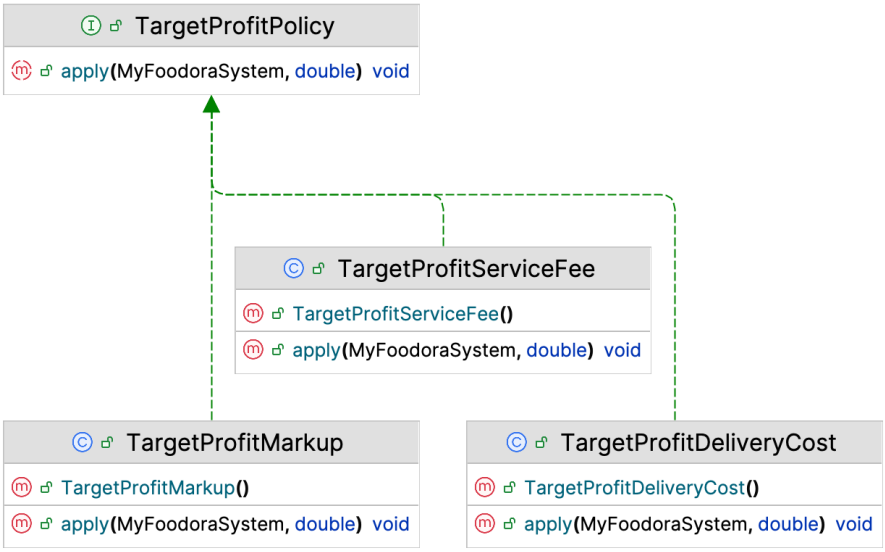


Fig. 9 – UML diagram of TargetProfitPolicy Class

2.10 ShippedOrderSortingPolicy

Finally, the `shippedOrderSortingPolicy` package includes three classes responsible for applying different strategies to sort shipped orders within the MyFoodora system. These strategies are based on various criteria, such as the most ordered items and half meals.

— `ShippedOrderSortingPolicy` is an interface defining a common method `show()` that must be implemented by all shipped order sorting policy classes. It prints the sorted list of name of restaurant and item/meal ordered and the number of shipping based the policy.

— `MostOrderedHalfMealPolicy` Implements the `ShippedOrderSortingPolicy` interface and applies a strategy to sort orders based on the most ordered half meals. It calculates the frequency of each half meal ordered and displays the sorted list.

— `MostOrderedItemMenuPolicy` : Also implements the `ShippedOrderSortingPolicy` interface and focuses on sorting orders based on the most ordered items from the menu. It counts the occurrences of each item ordered and prints the sorted list.

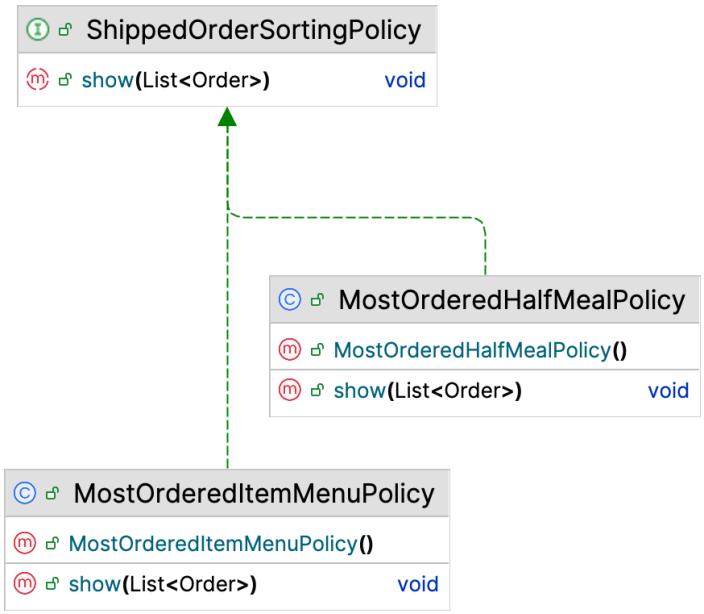


FIG. 10 – UML diagram of ShippedOrderSortingPolicy

3 MyFoodora interface

3.1 Initialization

The constructor initializes the `MyFoodoraSystem`, the main system object for managing users, orders, and policies.

The `setup` method reads configuration data from the file "my_foodora.ini" to set up the initial state of the system, including creating restaurants, customers, and couriers based on the specified numbers in the configuration file.

The `start` method initiates the command-line interface (CLI), allowing users to interact with the system. It begins by creating a CEO manager with the username `ceo` and password `123456789`. Additionally, it creates a second manager, corresponding to the deputy, with the same username `deputy` and password `123456789`. Following this, the method enters a loop to continually prompt the user for input commands.

3.2 User Authentication

The `login` method handles user authentication by verifying the provided username and password against the stored user credentials. Upon successful login, it sets the current user to the authenticated user.

The `logout` method logs out the current user, resetting the current user to null.

3.3 Notification Management

The methods `giveConsensusNotification`, `removeConsensusNotification`, and `showNotification` collectively handle the management of notifications for customers.

The `giveConsensusNotification` method allows customers to give their consensus for receiving special offer notifications. It checks if the current user is a customer and updates their notification settings accordingly.

The `removeConsensusNotification` method enables customers to withdraw their consensus for receiving special offer notifications. Similar to the previous method, it checks if the current user is a customer and adjusts their notification preferences.

The `showNotification` method displays the latest notifications for the current customer. It first checks if a customer is logged in and then retrieves and prints their unread notifications. If there are no notifications, it prompts the customer to give their consensus to start receiving notifications.

3.4 Restaurant and Order Management

3.4.1 Managing the ordres

The `registerRestaurant` method allows a manager to register a new restaurant in the system. It verifies if the current user is a manager and expects four parameters : the restaurant's name, address, username, and password. The address is split into latitude and longitude coordinates, and a new `Restaurant` instance is created with the provided information. Finally, the restaurant is added to the system.

The `createOrder` method enables customers to create a new order by specifying the restaurant's name and the order name. It checks if the current user is a customer and searches for the restaurant with the given name.

The `addItem2Order` method permits customers to add items to their order. It first verifies if the current user is a customer, then searches for the specified order, and finally adds the item to the order if found.

The `endOrder` method allows customers to finalize an order by providing the order name and the date of completion. It validates the format of the date and ensures that the order belongs to the current customer before finalizing it.

The `removeSpecialOffer` method allows a restaurant to remove the special offer by setting the meal of the week to null. It verifies if the current user is a restaurant before proceeding.

The `showMenuItem` method displays the menu items for a specified restaurant. It searches for the restaurant with the given name and prints its menu items along with their prices.

3.4.2 Restaurant numbers

The `showTotalProfit` method allows managers to view the total profit of the MyFoodora system. It verifies if the current user is a manager and prints the total profit calculated by the system.

The `showTotalProfitPeriod` method enables managers to see the total profit within a specified time period. It validates the manager's status and expects the start and end dates as input. Then, it calculates and prints the total profit within the specified period.

The `showRestaurantTop` method displays statistics about restaurants based on their sales. It retrieves the list of orders from the system and counts the number of delivered orders for each restaurant. Then, it sorts the restaurants by the number of sales in decreasing order and displays them. Additionally, it shows the remaining restaurants alphabetically sorted without any orders.

3.5 Customer Management

The `registerCustomer` method enables a manager to register a new customer. Similar to registering a restaurant, it checks if the current user is a manager and expects five parameters : the customer's first name, last name, username, address, and password. The address is split into latitude and longitude coordinates, and a new `Customer` instance is created with the provided information. Finally, the customer is added to the system.

The `showCustomers` method retrieves the list of customers from the system and prints their names in a clear format.

The `showPoints` method displays the number of points on a customer's fidelity

card. It checks if the current user is a customer and prints either the points or the type of fidelity card if no points are available.

The `showAverageIncomePerCustomer` method allows managers to view the average income per customer in the MyFoodora system. It verifies the manager's status and prints the average income per customer calculated by the system.

The `showAverageIncomePerCustomerPeriod` method enables managers to see the average income per customer within a specified time period. It validates the manager's status and expects the start and end dates as input. Then, it calculates and prints the average income per customer within the specified period.

3.6 Courier Management

The `registerCourier` method allows a manager to register a new courier. It verifies if the current user is a manager and expects five parameters : the courier's first name, last name, username, position (in the format x,y), and password. The position is split into latitude and longitude coordinates, and a new `Courier` instance is created with the provided information. Finally, the courier is added to the system.

The `onDuty` and `offDuty` methods allow couriers to set themselves on or off duty, respectively. They verify if the current user is a courier before proceeding and update the courier's on-duty status accordingly.

The `changeLocation` method enables couriers to update their current location. It verifies if the current user is a courier and expects latitude and longitude coordinates as input to set the new location.

The `findDeliverer` method allows restaurants to allocate orders to a deliverer. It verifies if the current user is a restaurant and searches for the specified order in incomplete orders. Then, it allocates a courier to the order based on the current delivery policy.

For the deliveries, the `showCourierDeliveries` method enables managers to view the deliveries completed by couriers. It retrieves the list of orders from the system and counts the number of completed deliveries for each courier. Then, it sorts and displays the couriers with completed deliveries and those without completed deliveries separately.

3.7 Policy

The `setDeliveryPolicy` method allows managers to set the delivery policy for the MyFoodora system. It verifies if the current user is a manager and expects the name of the policy as input, then sets the appropriate delivery policy.

The `setProfitPolicy` method enables managers to set the profit policy for the MyFoodora system. Similar to setting the delivery policy, it verifies the user's manager status and expects the name of the profit policy as input, then sets the appropriate profit policy.

And the `associateCard` method allows managers to associate a fidelity card with a user. It verifies if the current user is a manager and expects the username and type of the fidelity card as input. Then, it associates the specified fidelity card with the corresponding user.

3.8 runTest command

Finally, the `runTest` command executes a predefined suite of test cases in a text file scenario. By executing these test scenarios, users can evaluate the Myfoodora system's behavior.

Fig. 11 – UML diagram of Interface Class

4 Evaluation and tests

To thoroughly test our program's functionalities, we created several test cases. These test cases execute commands grouped within specific text files. Additionally, we implemented an initial configuration file named **my_foodora.ini**. This file enables automatic initialization of a default configuration that includes :

— 5 restaurants

— 4 customers

— 3 couriers

4.1 Test scenario 1

1. The CEO logs in.
2. The CEO registers two customers, Elias and Nabil.
3. The CEO logs out.
4. Elias logs in and gives his consent to be notified, then logs out.
5. Restaurant1 logs in, sets the Fullmeal to a special offer, then logs out.
6. Elias logs in, receives a notification about a new meal of the week at restaurant1.
7. Elias creates an order at restaurant1, adds items, includes the special meal, finalizes the order, and logs out.
8. Nabil logs in and does not receive any notifications since he didn't give consent.
9. Nabil creates an order at restaurant2, adds some items and a meal and logs out.
10. The CEO logs in again and access to :
    — order history
    — The average income per customer
    — The total profit generated
    — The courier deliveries
    — The top-selling restaurants.

This scneario was created to test the functionality of the system related to restaurant offers, customer orders, and CEO-level access to business metrics. Also, this scenario shows how the system notification management for the customers. Additionally, the scenario assesses the system's robustness in managing various user roles and permissions. It demonstrates the differentiation between CEO, restaurant, and customer roles, with each having distinct privileges and interactions within the platform.

4.2 Test scenario 2

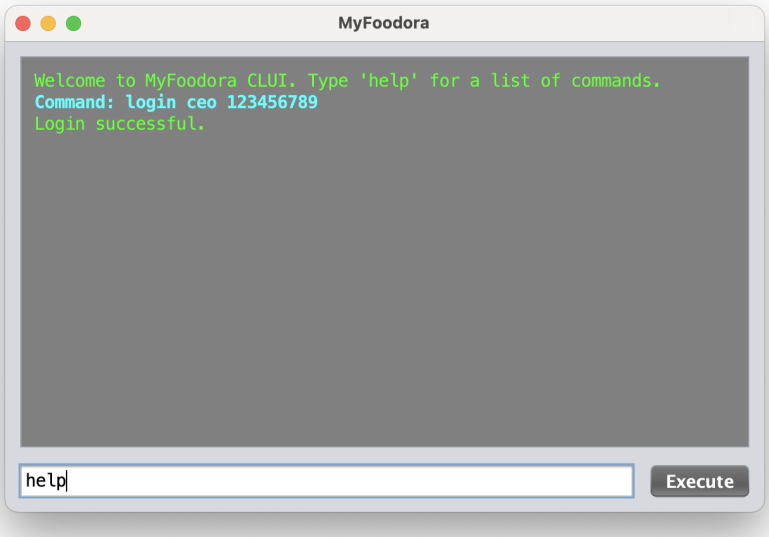
1. The CEO logs in.
2. The CEO registers two customers, Elias and Nabil.
3. The CEO logs out.
4. Elias logs in and gives his consent to be notified, then logs out.
5. Restaurant1 logs in, sets the Fullmeal to a special offer, then logs out.
6. Elias logs in, receives a notification about a new meal of the week at restaurant1.
7. Elias creates an order at restaurant1, adds items, includes the special meal, finalizes the order (May), and logs out.
8. Nabil logs in and does not receive any notifications since he didn't give consent.
9. Nabil creates an order at restaurant2, adds some items and a meal, finalizes the order (May), and logs out.
10. The CEO logs in again and accesses :
    - The total profit generated
    - The courier deliveries
11. The CEO sets the delivery policy to FastestDelivery.
12. The CEO sets the profit policy to TargetProfitServiceFee.
13. The CEO applies the profit policy with a value of 20 aiming for $20 in profit in June.
14. The CEO logs out.
15. Elias and Nabil log in successively and make the same orders as in May but finalize them in June, and log out.
16. The CEO logs in again and accesses :
    - The total profit generated in June. It is $20 as aimed.
    - The courier deliveries

This test scenario evaluates the manager's ability to influence MyFoodora's parameters to achieve a specific profit target. Similar to the previous scenario, two customers are created who will place orders in May, generating a total profit of $15. Based on this history, the manager will aim for a $20 profit in the following month by utilizing the targetPolicy, which automatically adjusts the service fee to achieve this goal.

5 Guide and tips for the interface

We designed the interface so that it can be as easy to use as possible. To make it also more appealing and clear, we displayed the interface on a window as shown below.

The corresponding file is located at the **fr.cs.Myfoodora.userInterface** package, where it can be executed in the **MyFoodoraGUI** class, that displas the window one the code is runned. Initially, as mentionned before, an initial configuration is launched



Besides the ceo, any user can register using the appropriate command. Here are the essential commands to follow to interact with the interface ;

— **Registration** :
  • As a **Restaurant Owner**, use : `registerRestaurant <name> <address> <username> <password>`
  • As a **Customer**, use : `registerCustomer <firstName> <lastName> <username> <address> <password>`
  • As a **Courier**, use : `registerCourier <firstName> <lastName> <username> <x,y> <password>`

— **Menu Management** :
  • **Adding Dishes to Menu** : `addDishRestaurantMenu <name> <type> <price> [<isVegetarian> <isGlutenFree>]`
  • **Creating a Meal** : `createMeal <name> <mealType> <item1> <item2> [<item3>]`
  • **Setting Special Offers** : `setSpecialOffer <name>`

— **Order Handling** :
  • **Creating an Order** : `createOrder <restaurantName> <orderName>`
  • **Adding Items to Order** : `addItem2Order <orderName> <itemName>`
  • **Finalizing an Order** : `endOrder <orderName> <day> <month> <year>`

— **Delivery Management** :
  • **Setting Delivery Policy** : `setDeliveryPolicy <policyName>`

— **Profit Policy** :
  • **Setting Profit Policy** : `setProfitPolicy <policyName>`

— **Fidelity Cards** :
  • **Associating Fidelity Cards with Customers** : `associateCard <username> <cardType>`

— **Viewing Reports** :
  • **Viewing Courier Deliveries** : `showCourierDeliveries`
  • **Viewing Restaurant Top Orders** : `showRestaurantTop`

— **Test Script Execution** :
  • **Running Test Script** : `runTest <filepath> or <filename>`, in our case **testScenario1.txt** or **testScenario2.txt**

`help` displays a comprehensive list of commands for interacting with the food delivery system interface.

6 Splitting work

Name	Work
Elias	core of the system (with meals, menu, user, policies and UML diagrams)
Nabil	core of the system (with space time coordinates, order, CLUI interface, GUI and JUnit tests)

7 Advantages and limitations

We built the MyFoodora system in a way that it can responds to all the technical requirements. Our MyFoodora system has been designed with several key advantages, ensuring flexibility, extensibility, and robustness. The most important principle that we focused on is the open close principle. That's why we implemented various interfaces and abstract classes to apply robust patterns that allow update for the application without having to modify all the classes.

We implemented the Abstract Factory Pattern for meals and menu to enhance the system's flexibility by enabling the creation and addition of new menu items like drinks and new way to create meals without modifying the core system. This approach allows MyFoodora to expand its offerings and adapt to changing market demands effortlessly.

Similarly, we implemented the Strategy Pattern for fidelity cards. We can thus introduce new discount strategies seamlessly, providing customizable benefits to our

customers. We also implemented the Strategy Pattern for the shipped order sorting policies, delivery policies and target profit policies.

We implemented the abstract class user that represents various users allowing us to add potentiel new type of users to the application.

Moreover, we implemented the observer pattern, where the customer is the observer and the restaurant the observable. That allowed us to manage effiently the notifications and to update the interested customers each time they login into the application.

A GUI interface was implemented ensuring an intuitive and user-friendly experience for both customers and system administrators, making interactions with the system smooth and efficient. We have also dealt with error handling throughout the system, maintaining the system's stability and providing clear feedback to users.

However, there are a few limitations that should be acknowledged. for instance, the GUI interface was designed to be user-friendly and intuitive, but there is always room for enhancement. Adding more interactive features, improving the visual design, and incorporating user feedback can further improve the user experience. Advanced functionalities such as drag-and-drop meal customization or real-time order tracking could also be valuable additions.

8 Conclusion

The MyFoodora project successfully delivered a robust and flexible food delivery management system, overcoming challenges to create a user-friendly interface that allows for the creation, management, and utilization of various entities within the food delivery ecosystem. This system effectively simulates real-world food delivery app operations.