CentraleSupélec

myFoodora - Food Delivery System

Group n°5

Work carried out by the group:
Farah MESSIOUI
Imad AIT BEN SALAH

Supervisors:
Mr. Paolo BALLARINI
Mr. Arnault LAPITRE

June 5, 2025

Contents

1 Introduction

This document explains the design and development of myFoodora, a Java-based project that simulates popular food delivery apps such as Deliveroo. The work is divided into two parts: the first builds the main system needed to handle the platform's key functions, and the second creates a command-line interface for users to interact with the system.

The main purpose of myFoodora is to create a strong and easy-to-use platform that connects customers with restaurants. It handles restaurant menus, takes customer orders, and supports smooth delivery management. The project brings all these features together into one simple system designed to offer a good 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. 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 `menuMeals`.

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

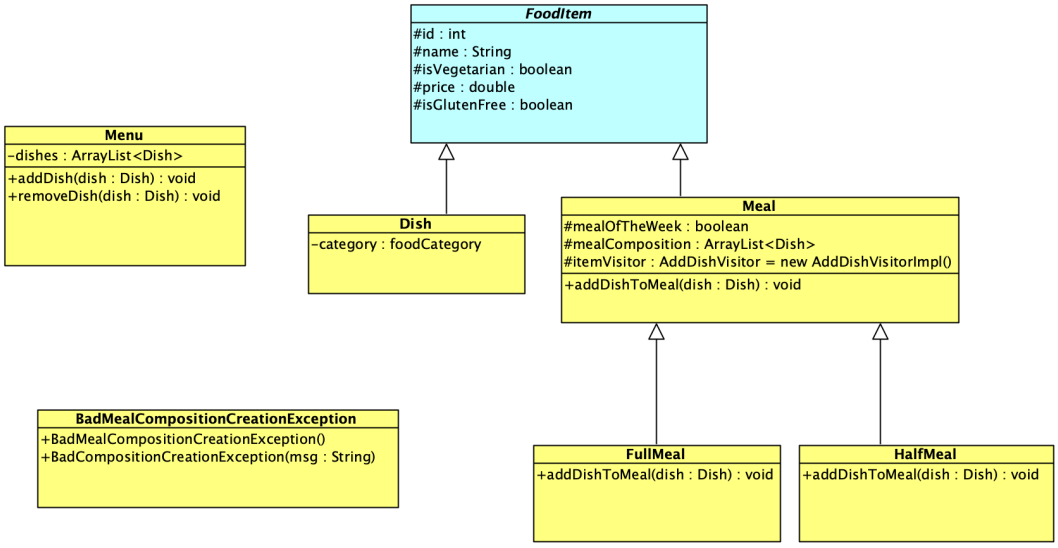

Figure 1: UML Diagram of the `menuMeals` package classes

2.1.1 `FoodItem` Class

At the core of the `menuMeals` package lies the abstract class `FoodItem`, which serves as the foundation for all food-related components offered by restaurants.

The abstract class `FoodItem` contains the following key attributes:

Attributes:

- **id (int)**: A unique identifier automatically generated for each food item.
- **name (String)**: The name of the food item.
- **isVegetarian (boolean)**: Indicates whether the food item is vegetarian.
- **price (double)**: The price of the food item.
- **isGlutenFree (boolean)**: Indicates whether the food item is gluten-free.
- **orderedFrequency (int)**: Tracks how many times the food item has been ordered.

Constructors:

- `FoodItem(String name, boolean isVegetarian, double price, boolean isGlutenFree)`: Full initialization with price.
- `FoodItem(String name, boolean isVegetarian, boolean isGlutenFree)`: Initialization without price.
- `FoodItem(String name)`: Initialization with only the name.

Methods:

- `Getters and Setters` for all attributes to access and modify private fields.
- `public void incrementOrderedFrequency()`: Increments the number of times this item was ordered, useful for statistics.

2.1.2 `Dish` Class

The `Dish` class extends the abstract `FoodItem` class to represent a single dish on a restaurant's menu. Unlike composite meals, a dish is a basic item without sub-elements and is categorized as a starter, main dish, or dessert.

Attributes:

- Inherits several attributes from its superclass `FoodItem`
- **category (DishCategory)**: possible values are `STARTER`, `MAINDISH`, or `DESSERT`.

Constructors:

- `Dish(String name)`: Default constructor initializing only the name.
- `Dish(String name, boolean isVegetarian, double price, boolean isGlutenFree, DishCategory category)`: Full constructor initializing all properties.
- `Dish(String name, boolean isVegetarian, boolean isGlutenFree)`: Partial constructor without price or category.

Methods:

- `DishCategory getCategory()`: Returns the dish category.
- `void setCategory(DishCategory category)`: Sets the dish category.
- `String toString()`: Provides a string representation of the dish, including all its properties.

2.1.3 `Meal` Class

The abstract class `Meal`, part of the `menuMeals` package, represents a complex food item composed of multiple `Dish` objects. It extends the abstract class `FoodItem`, inheriting its common attributes.

Attributes:

- **mealOfTheWeek (boolean)**: Indicates if the meal is designated as the "meal of the week".
- **mealComposition (ArrayList<Dish>)**: A list containing the dishes that make up the meal.
- **itemVisitor (AddDishVisitor)**: An instance of the visitor pattern implementation used to validate or add dishes to the meal.

Constructors:

- `Meal(String name)`
- `Meal(String name, boolean isVegetarian, boolean isGlutenFree)`

Methods:

- `public abstract void addDishToMeal(Dish dish) 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).
- Getters and setters

2.1.4 FullMeal and HalfMeal Classes

These classes inherit from the abstract `Meal` class and implement the `addDishToMeal` method to enforce specific rules for full meal and half meal compositions respectively.

Methods:

- `public abstract void addDishToMeal(Dish dish) throws BadMealCompositionCreationException`: Abstract method to add a dish to the meal with validation.

2.1.5 `Menu` Class

The `Menu` class manages a collection of dishes available in a restaurant's menu. It provides methods to add and remove dishes while ensuring no duplicates exist.

Attributes:

- **items (HashMap<String, Dish>)**: A hashmap storing dishes with their names as keys.

Constructors:

- `Menu()`: Default constructor initializing an empty menu.
- `Menu(HashMap<String, Dish> items)`: Constructor initializing the menu with a predefined set of dishes.

Methods:

- Getters and setters for `items`
- `void addDish(Dish item) throws ItemAlreadyExistsException`: Adds a dish to the menu if it does not already exist.
- `void addDish(String itemName) throws ItemAlreadyExistsException`: Adds a new dish with the given name if it does not already exist.
- `void removeDish(FoodItem item) throws ItemNotFoundException`: Removes a dish from the menu if it exists.
- `void removeDish(String itemName) throws ItemNotFoundException`: Removes a dish by name if it exists.
- `Dish getDish(String itemName) throws ItemNotFoundException`: Retrieves a dish by its name if it exists.
- `String toString()`: Returns a string representation of the menu listing all dishes.

2.1.6 `AddDishVisitor` 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 `AddDishVisitor` interface, realized by the `AddDishVisitorImpl` 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.

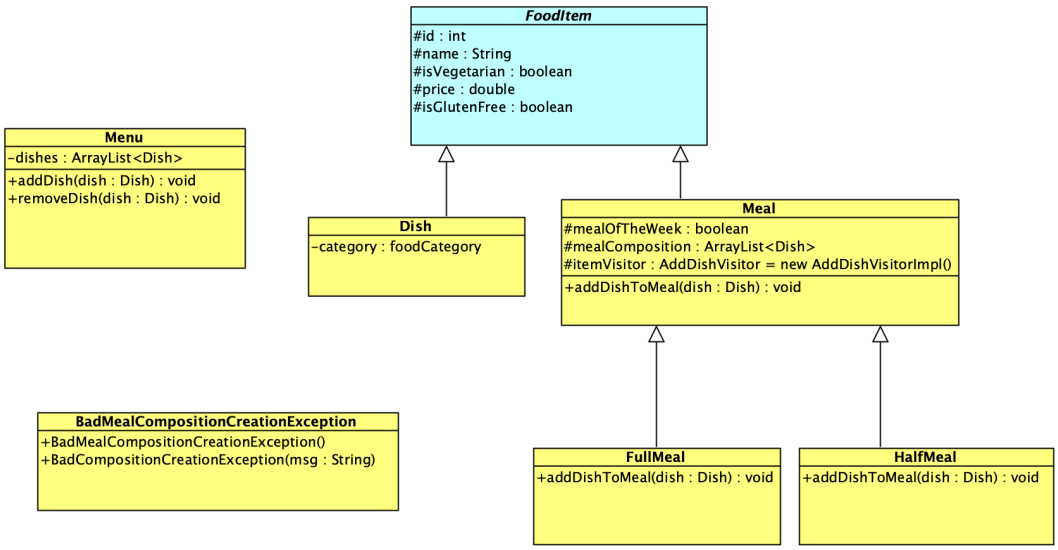

Figure 2: Adding dish to meals' visitor pattern UML Diagram

Methods:

- `void visit(Dish dish, FullMeal meal) throws BadMealCompositionCreationException`: Adds a dish to a full meal with validation.
- `void visit(Dish dish, HalfMeal meal) throws BadMealCompositionCreationException`: Adds a dish to a half meal with validation.

2.1.7 Meal Price Calculation

In order to provide a flexible mechanism for computing the price of a meal especially when discounts are involved we introduced the Strategy design pattern through the `MealPriceCalculation` interface.

Interface `MealPriceCalculation` This interface defines the contract for calculating the total price of a meal:

- `double calculatePrice(Meal meal, double discount)`: computes the total price of a meal after applying a discount.

Concrete Implementation: `MealPriceCalculationImpl` This class performs the calculation by summing the prices of all dishes in the meal and applying the discount.

2.2 FidelityCards Package

The `FidelityCards` package is responsible for managing the pricing strategy depending on the fidelity card associated with the customer. It contains:

- One **abstract class** `FidelityCard`,
- Three **concrete classes**:
  - `BasicFidelityCard`
  - `PointFidelityCard`
  - `LotteryFidelityCard`
- One **FidelityCardFactory** (following the Factory design pattern), which is responsible for creating fidelity cards given a string input (`cardType`).

The pricing of a customer's order depends on the type of fidelity card he/she possesses. To manage this dynamic behavior, we implemented the Strategy design pattern: each type of fidelity card defines its own logic for computing reductions and final prices. This makes the system flexible and extensible for future types of fidelity cards.

2.2.1 FidelityCard Class

An **abstract class** representing a fidelity card used to apply reductions and compute final prices for orders.

Attributes:

- `FidelityCardType type`: the type of the fidelity card.

Constructors:

- `FidelityCard()`: default constructor.
- `FidelityCard(FidelityCardType type)`: constructs a fidelity card with the specified type.

Methods:

- Getters and setters
- `public abstract double computeOrderReduction(Order order)`: calculates the reduction for a given order.
- `public abstract double computeOrderPrice(Order order)`: calculates the final price of the order after reduction.

2.2.2 Basic Fidelity Card

This class represents the most elementary fidelity card, offering no discounts to customers.

Methods:

- `computeOrderReduction(Order order)`: always returns 0, as no reduction is applied.
- `computeOrderPrice(Order order)`: simply returns the original order price without any modification.

2.2.3 Point Fidelity Card

This fidelity card allows customers to accumulate points based on the value of their previous orders. Once a certain threshold is reached, a discount is applied to the next order.

Attributes:

- Points are earned at a rate of 1 point per `10€` spent (modifiable via `amountReduction`).
- When `100 points` (parameter `targetPoints`) are accumulated, the customer receives a `10%` discount (parameter `discountFactor`) on their next order.
- After using the discount, `100 points` are subtracted from the customer's balance.

Methods:

- `computeOrderReduction(Order order)`: returns the discount if the target number of points is reached.
- `computeOrderPrice(Order order)`: applies the discount if available, then deducts the corresponding points.
- `updatePoints(double price)`: updates the point balance after an order.

This implementation supports **dynamic reward logic** and is consistent with the Strategy design pattern used in the `FidelityCards` package.

2.2.4 LotteryFidelityCard

This fidelity card gives customers a daily chance to win their most expensive ordered meal for free, subject to a probability.

Attributes:

- `static double probability`: chance (default 0.05) to win a free meal each day.
- `Calendar lastTimeUsed`: tracks the last date when a free meal was awarded.

Constructors:

- `LotteryFidelityCard()`

Methods:

- `static boolean areDifferentDays(Calendar cal1, Calendar cal2)`: checks if two dates fall on different days.
- `double computeOrderReduction(Order order)`: if unused today and a random draw (using `probability`) succeeds, awards a free meal equal to the highest-priced meal in the order and updates `lastTimeUsed`.
- `double computeOrderPrice(Order order)`: returns `order.computePrice()` minus the reduction (if any).
- `Calendar lastTimeUsed()` and `void setLastTimeUsed(Calendar lastTimeUsed)`: getters/setters for the last-used date.
- `String toString()`: returns a string showing "Last time used" (or "Never used").

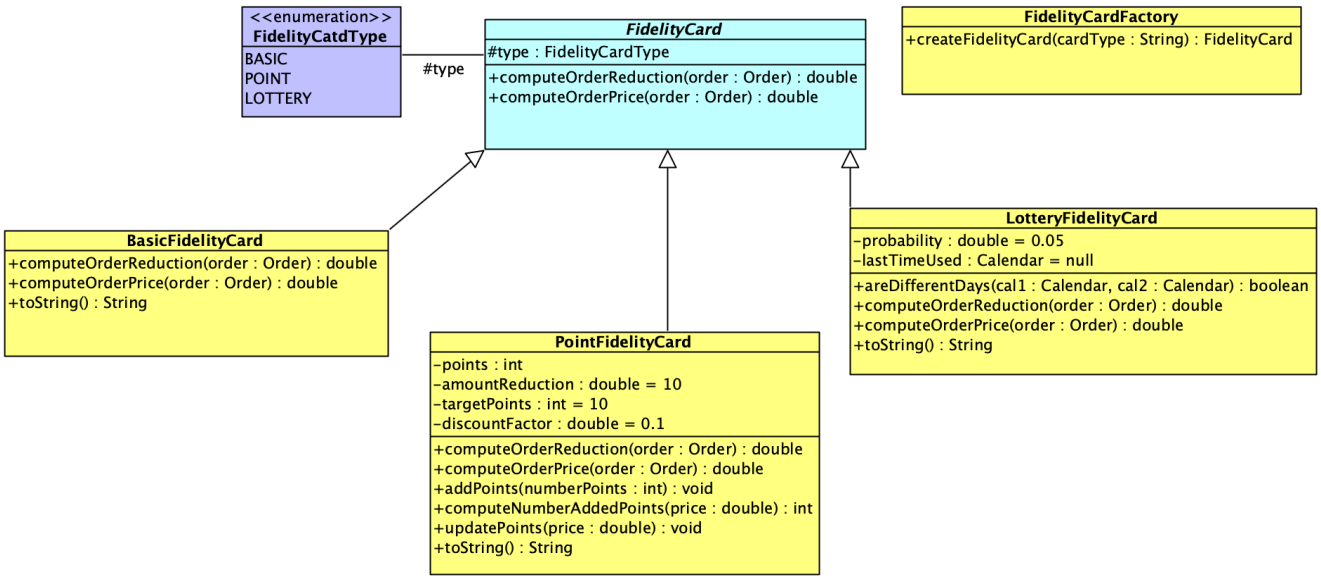

Figure 3: UML Diagram of Fidelity Card classes

2.3 Delivery Policy

The `DeliveryPolicy` package is designed with extensibility and modularity in mind. It contains:

- An interface: `DeliveryPolicy`
- Two concrete implementations: `FairOccupation` and `Fastest`
- A factory class: `DeliveryPolicyFactory`
- An enumeration: `DeliveryPolicyType`

This package uses the **Strategy Design Pattern** to define different algorithms for assigning couriers to delivery orders. Each policy encapsulates its own strategy for selecting a courier:

- `FairOccupation`: selects the courier with the fewest deliveries.
- `Fastest`: selects the courier closest to the restaurant.

To decouple the policy creation from the system's main logic (e.g., the Manager class), we employ the **Factory Design Pattern**. The `DeliveryPolicyFactory` returns the appropriate policy implementation based on a user-provided type, encapsulating the instantiation logic and avoiding changes in the consumer class when new policies are added.

Enhancing the Open/Closed Principle: Our design strictly follows the **Open/Closed Principle (OCP)**:

- **Strategy Pattern:** By defining a common interface and multiple implementations, we can add new policies without modifying existing ones.
- **Factory Pattern:** The factory handles object creation, allowing the system to be extended with new policy types while maintaining the same interface.

This structure ensures that the system is open to extension but closed to modification, increasing maintainability and robustness.

2.3.1 Delivery Policy interface

`DeliveryPolicy` **Interface** defines the contract that any delivery policy must implement.

- **Methods:**
  - `public abstract void allocateCourier(Order order)` : allocates a courier to the given `Order`. This method may throw a `NoCourierIsAvailableException` if no suitable courier is found.

2.3.2 FairOccupation Class

`FairOccupation` Class implements DeliveryPolicy

- **Methods:**
  - `allocateCourier(Order order)` : selects the on-duty courier with the lowest delivery count. assigns the courier to the order and increments their delivery counter. Throws `NoCourierIsAvailableException` if no courier is available.

2.3.3 Fastest Class

`Fastest` Class implements DeliveryPolicy

- **Methods:**
  - `allocateCourier(Order order)` : selects the closest on-duty courier to the restaurant's location. Assigns them to the order and increments their delivery counter. Throws `NoCourierIsAvailableException` if no courier is available.

2.3.4 `DeliveryPolicyFactory`

This class is responsible for instantiating a concrete implementation of the `DeliveryPolicy` interface based on the given `DeliveryPolicyType`. It plays a key role in applying the **Factory Design Pattern**.

Constructor:

- `public DeliveryPolicyFactory()`: initializes an instance of the factory.

Method:

- `public DeliveryPolicy createDeliveryPolicy(DeliveryPolicyType deliveryPolicy) throws UndefinedPolicyException:`
  Creates and returns a concrete delivery policy implementation based on the input parameter. The mapping is as follows:
  - `FASTEST` → `FairOccupation`
  - `FAIR` → `Fastest`

  If the given policy type is not recognized, an `UndefinedPolicyException` is thrown.

2.3.5 `DeliveryPolicyType` Enum

- **Values:**
  - `FASTEST`
  - `FAIR`

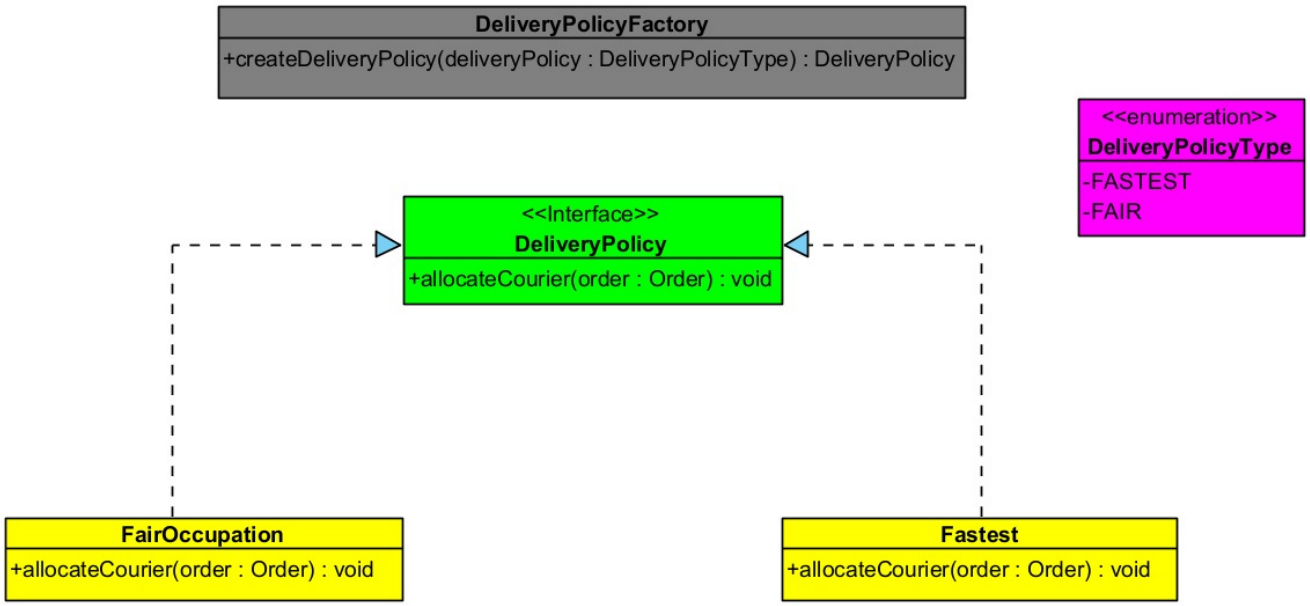

Figure 4: UML Diagram of the DeliveryPolicy feature

2.4 targetProfitPolicy

The `targetProfitPolicy` package implements two well-known design patterns to manage the selection and application of profit-targeting strategies: the **Factory Design Pattern** and the **Strategy Design Pattern**.

- The **Strategy Pattern** is used to define a family of algorithms, encapsulate each one, and make them interchangeable. The interface `TargetProfitPolicy` declares two main methods: `computeTargetProfitPolicyParam()` and `setParam()`, both of which are implemented differently in each concrete strategy class.
- The **Factory Pattern** is employed through the `TargetProfitPolicyFactory` class, which centralizes the creation logic of policy objects. Depending on the `TargetProfitPolicyType` provided, it returns the corresponding policy instance, fully decoupling object instantiation from client classes like `Manager`.

The `targetProfitPolicy` package contains:

- An interface: `TargetProfitPolicy`
- Three concrete implementations:
  - `TargetProfitMarkUp`: adjusts the markup percentage.
  - `TargetProfitDeliveryCost`: adjusts the delivery cost.
  - `TargetProfitServiceFee`: adjusts the service fee.
- A factory class: `TargetProfitPolicyFactory` to centralize the creation of policy objects.
- An enumeration: `TargetProfitPolicyType` representing available strategies.

Each policy implements the logic to compute a parameter based on last month's statistics (income, number of orders) and raises a dedicated exception if the target is deemed unreachable. The Manager can easily switch policies without modifying the internal logic, only needing to request the appropriate instance from the factory.

Design Principles This package upholds the **Open/Closed Principle** by enabling the addition of new strategies (e.g., modifying other parameters) without touching existing code. The **Strategy Pattern** provides the flexibility to encapsulate different algorithms independently, while the **Factory Pattern** ensures that instantiation logic is isolated and easily extendable.

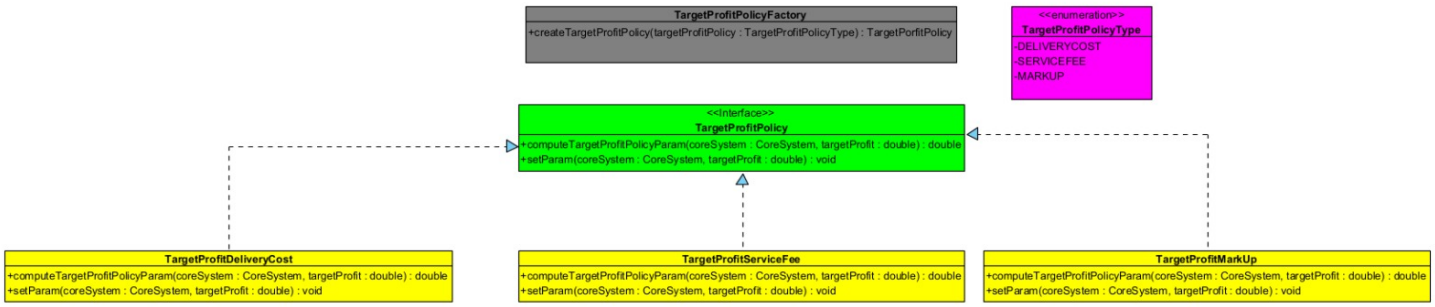

Figure 5: UML Diagram of the targetProfitPolicy feature

2.5 Users of myFoodora system

The `users` package represents the core of the user management system in `myFoodora`. It encapsulates all types of users, along with the logic required to generate user identifiers and manage secure authentication through password hashing.

Architecture This package is designed around an abstract base class, `User`, which serves as the parent class for all user roles in the system:

- `Customer`
- `Courier`
- `Restaurant`
- `Manager`

Design Patterns Used

- **Singleton Pattern**: Implemented in the `UserIdGenerator` class, which ensures that user IDs are generated uniquely and consistently across the system through a single, globally accessible instance.
- **Inheritance and Polymorphism**: The base class `User` defines shared attributes and methods for all user types, facilitating code reuse and modular design.

Security Considerations The package promotes good security practices through the use of SHA-256 hashing via the `PasswordHasher` class.

Extensibility The system is easily extendable: new user types can be added by creating a new class that inherits from `User` and is included in the `UserType` enum. This design complies with the Open/Closed Principle.

2.5.1 User class

The `User` class is the abstract parent class of all users in the `myFoodora` system. It provides shared attributes and behaviors across the various user types (Customer, Courier, Restaurant, Manager).

Attributes:

- **id (int)**: A unique identifier for the user, generated automatically via the singleton `UserIdGenerator`.
- **name (String)**: The full name of the user.
- **username (String)**: The unique username used for login authentication.
- **hashedPassword (String)**: The password of the user, hashed using SHA-256.

Constructor:

- `User(String name, String username, String password)`: Initializes a new user with the given name, username, and plaintext password. The password is hashed using the `PasswordHasher` class, and the ID is assigned via `UserIdGenerator`.

Methods:

- `public int getId()`: Returns the unique ID of the user.
- `public String getName()`: Retrieves the name of the user.
- `public void setName(String name)`: Updates the name of the user.
- `public String getUsername()`: Retrieves the username of the user.
- `public void setUsername(String username)`: Updates the username.
- `public String getHashedPassword()`: Returns the hashed password.
- `public void setHashedPassword(String password)`: Updates the user's password (hashes the new password).

2.5.2 Restaurant

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

Attributes:

- **location (Coordinate)**: The geographical location of the restaurant.
- **menu (Menu)**: The menu containing a collection of dishes.
- **meals (HashMap<String, Meal>)**: A map linking meal names to Meal objects.
- **foodFactory (FoodFactory)**: Factory used to create meals.

- **mealPriceStrategy (MealPriceCalculation)**: Strategy used to compute the price of a meal.
- **genericDiscount (double)**: Generic discount applicable to all meals.
- **specialDiscount (double)**: Special discount for meals of the week.
- **orderCounter (int)**: Tracks the number of orders made at the restaurant.

Constructors:

- `Restaurant(String name, String username, String password, Coordinate location, Menu menu, HashMap<String, Meal> meals, double genericDiscount, double specialDiscount)`: Full initialization of the restaurant.
- `Restaurant(String name, String username, String password, Coordinate location, double genericDiscount, double specialDiscount)`: Initialization with location and discounts, menu and meals initialized empty.
- `Restaurant(String name, String username, String password)`: Minimal initialization with default location, empty menu and meals.

Methods:

- `Getters and Setters`: For all attributes.
- `public void incrementOrderCounter()`: Increments the order counter.
- `public Meal createMeal(String name, MealType type)`: Creates a basic meal.
- `public Meal createMeal(String name, MealType type, boolean isVegetarian, boolean isGlutenFree)`: Creates a meal with additional dietary info.
- `public void addMeal(...)`: Adds a meal to the restaurant with various overloads.
- `public void removeMeal(...)`: Removes a meal from the restaurant.
- `public void showMeal(String mealName)`: Displays a meal by name.
- `public void addDish(...)`: Adds a dish to the menu with various overloads.
- `public void removeDish(...)`: Removes a dish from the menu with various overloads.
- `public void setMealPrice(Meal meal)`: Sets the price of a meal based on discount.
- `public void addDishToMeal(String mealName, String dishName)`: Adds a dish from the menu to a given meal.
- `public void showSortedHalfMeals()`: Displays sorted half meals based on ordered frequency.
- `public void showSortedFullMeals()`: Displays sorted full meals based on ordered frequency.
- `public void showSortedDishes()`: Displays sorted dishes based on ordered frequency.
- `public void displayRestaurant()`: Displays the menu and meals of the restaurant.
- `public String toString()`: String representation of the restaurant.

2.5.3 Customer

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

Attributes:

- **surname (String)**: The surname of the customer.
- **email (String)**: The customer's email address.
- **phone (String)**: The customer's phone number.
- **address (Coordinate)**: The physical address of the customer.
- **orderHistory (List<Order>)**: A list of past orders made by the customer.
- **consensus (boolean)**: A flag to store user consent (e.g., for receiving offers).

Constructors:

- `Customer(String name, String username, String password, String surname, String email, String phone, Coordinate address)`: Full constructor initializing all attributes.
- `Customer(String name, String username, String password, String surname)`: Minimal constructor with only surname and a default address.

Methods:

- `Getters and Setters`: For all attributes.
- `public void placeOrder()`: Allows the customer to place an order.
- `public void registerFidelityCard()`: Registers a fidelity card for the customer.
- `public void unregisterFidelityCard()`: Unregisters the fidelity card.
- `public void accessOrderHistory()`: Accesses the customer's order history.
- `public void accessFidelityPoints()`: Displays the customer's fidelity points.
- `public void displayOrders()`: Prints out the customer's completed orders with dates and totals.
- `public String toString()`: Returns a string representation of the customer.

2.5.4 Courier

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

Attributes:

- **surname (String)**: The surname of the courier.
- **position (Coordinate)**: The current geographical position of the courier.
- **phone (String)**: The courier's phone number.
- **deliveryCounter (int)**: The number of deliveries completed by the courier.
- **onDuty (boolean)**: Indicates whether the courier is currently on duty.

Constructors:

- `Courier(String name, String username, String password, String surname, Coordinate position, String phone)`: Full initialization with position.
- `Courier(String name, String username, String password, String surname, String phone)`: Initialization with default position.

Methods:

- `Getters and Setters`: For all attributes.
- `public void incrementDeliveryCounter()`: Increments the delivery counter by 1.
- `public String toString()`: String representation of the courier.

2.5.5 Manager

This class extends the `User` class and represents a manager within the system. A manager can manage users, configure policies, and compute global statistics on orders.

Attributes:

- **surname (String)**: The manager's surname.
- **coreSystem (CoreSystem)**: Singleton instance of the main system.
- **TPPFactory (TargetProfitPolicyFactory)**: Factory to create target profit policies.
- **DPFactory (DeliveryPolicyFactory)**: Factory to create delivery policies.

Constructor:

- `Manager(String name, String username, String password, String surname)`: Initializes a manager with name, credentials, and surname.

Main Methods:

- `Getters and Setters`
- `addCustomer(Customer)`: Adds a customer; throws exception if username already exists.
- `addRestaurant(Restaurant)`: Adds a restaurant; throws exception if username already exists.
- `addCourier(Courier)`: Adds a courier; throws exception if username already exists.
- `removeCustomer(Customer)`: Removes a customer; throws exception if user does not exist.
- `removeRestaurant(Restaurant)`: Removes a restaurant; throws exception if user does not exist.
- `removeCourier(Courier)`: Removes a courier; throws exception if user does not exist.
- `computeTotalIncome()`: Computes total income from all orders.
- `computeTotalIncome(Calendar, Calendar)`: Computes total income between two dates.
- `computeTotalProfit()`: Computes total profit from all orders.

- `computeTotalProfit(Calendar, Calendar)`: Computes total profit between two dates.
- `computeAverageIncomePerCostumer()`: Computes average income per customer.
- `computeAverageIncomePerCostumer(Calendar, Calendar)`: Computes average income per customer between two dates.
- `computeNumberOfOrders(Calendar, Calendar)`: Counts number of orders between two dates.
- `toString()`: Returns a string representation of the manager.

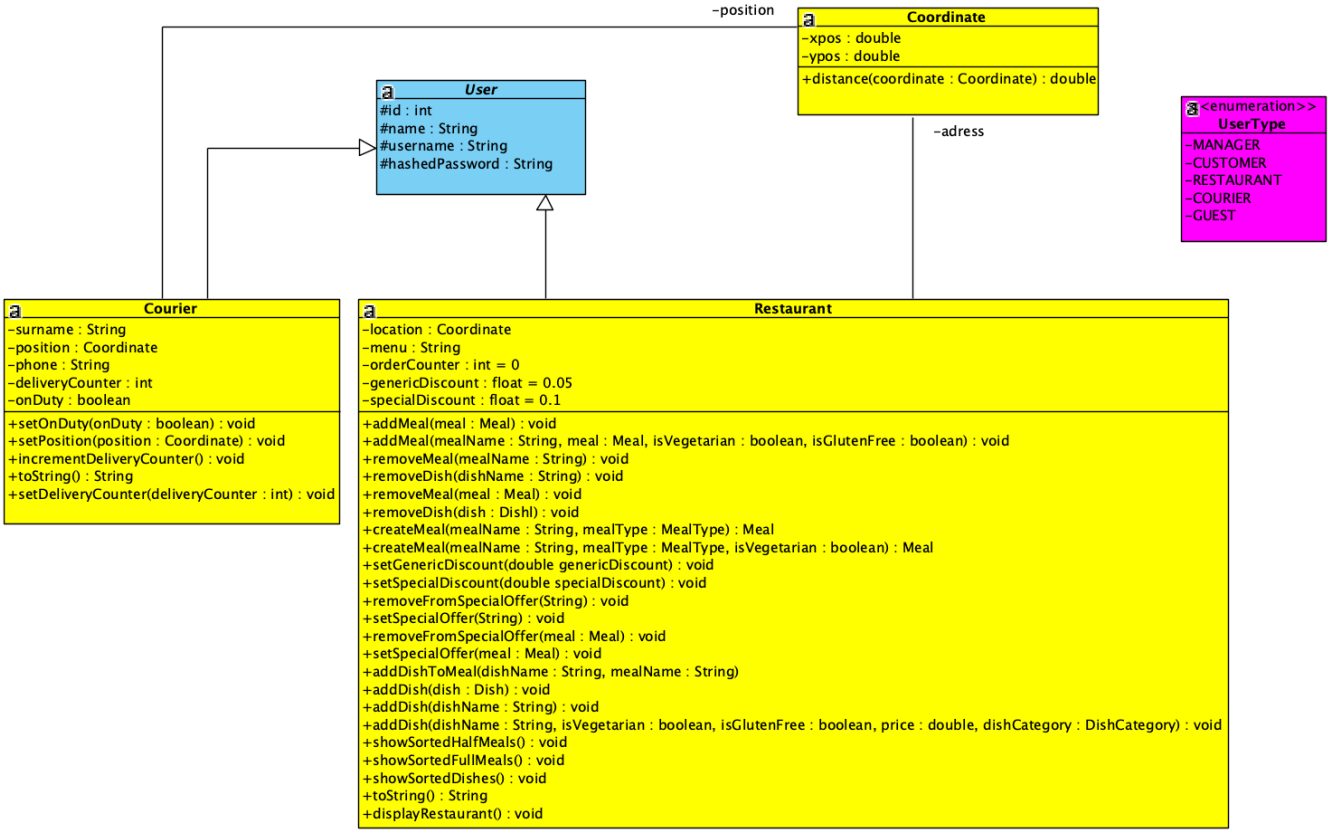

Figure 6: UML Diagram of the User related classes 1/2

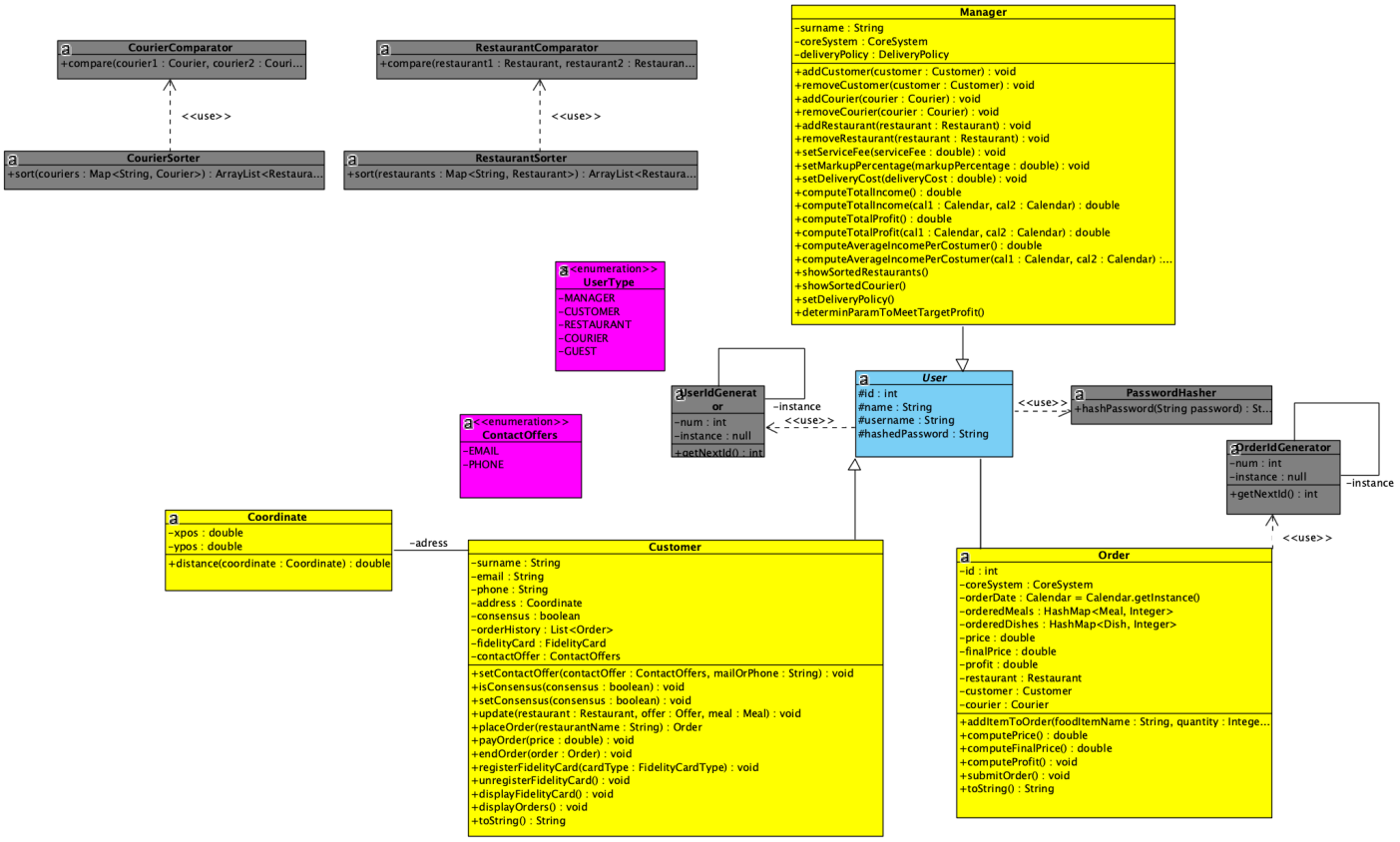

Figure 7: UML Diagram of the User related classes 2/2

2.6 Order

This class represents an order placed by a customer at a restaurant. It manages ordered meals and dishes, calculates prices and profits, assigns couriers, and keeps track of order details.

Attributes:

- **id (int)**: Unique identifier of the order.
- **coreSystem (CoreSystem)**: Singleton instance of the main system.
- **orderDate (Calendar)**: Date and time when the order was created.
- **orderedMeals (HashMap<Meal, Integer>)**: Meals in the order with their quantities.
- **orderedDishes (HashMap<Dish, Integer>)**: Dishes in the order with their quantities.
- **price (double)**: Total price before fees and markup.
- **finalPrice (double)**: Final price including fees and markup.
- **profit (double)**: Profit made from this order.
- **restaurant (Restaurant)**: The restaurant fulfilling the order.
- **customer (Customer)**: Customer who placed the order.
- **courier (Courier)**: Courier assigned to deliver the order.

Constructor:

- `Order(Customer customer, Restaurant restaurant)`: Initializes a new order with a unique ID, date, customer, and restaurant references.

Main Methods:

- `addItemToOrder(String foodItemName, Integer quantity)`: Adds a dish or meal by name and quantity to the order. Throws `ItemNotFoundException` if item does not exist.
- `computePrice()`: Computes the total price of all ordered meals and dishes.
- `computeProfit()`: Calculates the profit considering markup, service fees, and delivery costs.
- `submitOrder(double totalPrice)`: Allocates a courier, adds the order to system and customer history, increments order frequencies, and prints order details. Throws `NoCourierIsAvailableException` if no courier is available.
- `toString()`: Returns a formatted string representation of the order details including items and prices.

2.7 NotificationService

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.

2.7.1 NotificationService class

This class is a **Singleton** implementing the `Observable` interface. It manages a list of observers (subscribers), typically customers, who get notified when a restaurant creates a new special offer, such as a meal-of-the-week or special discounts.

Attributes:

- **instance (NotificationService)**: Singleton instance of the service.
- **subscribers (ArrayList<Observer>)**: List of registered observers.
- **changed (boolean)**: Flag indicating if there has been a change triggering notifications.

Constructor:

- `NotificationService()`: Private constructor to prevent multiple instances; initializes subscribers list and the changed flag.

Main Methods:

- `registerObserver(Observer)`: Adds an observer if not already subscribed.
- `removeObserver(Observer)`: Removes an observer if it exists in the list.
- `notifyObservers(Meal, Restaurant, Offer)`: Notifies all subscribed observers with the new offer details if there was a change.
- `setOffer(Meal, Restaurant, Offer)`: Sets the changed flag to true and triggers notification of observers.

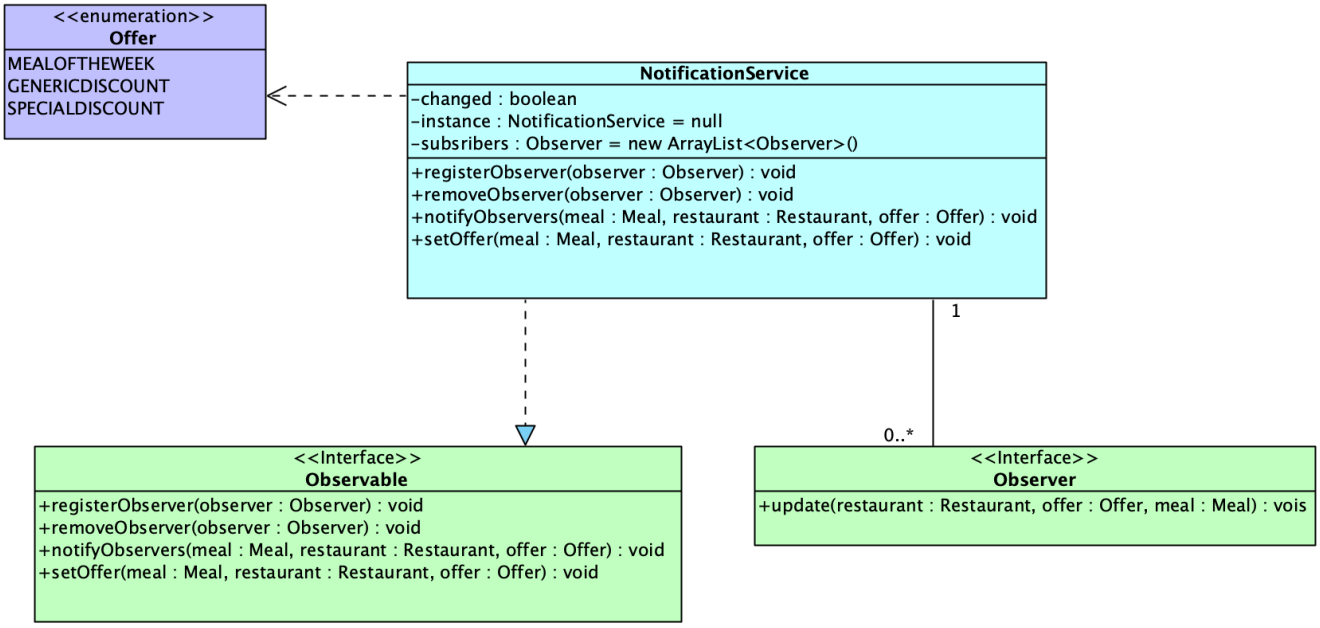

Figure 8: UML Diagram of Notification Service

2.8 Exceptions

- **BadMealCompositionCreationException**: Thrown when a meal is composed incorrectly, for example with missing or incompatible components.
- **ItemAlreadyExistsException**: Thrown when attempting to add an item that already exists in the system (e.g., duplicate products).
- **ItemNotFoundException**: Thrown when an expected item (such as a dish or meal) is not found in the system.
- **NoCourierIsAvailableException**: Thrown when there are no available couriers to assign to an order.
- **NoMatchingCredentialsException**: Thrown when login credentials do not match any user in the system.
- **NoUserExistsException**: Thrown when trying to access or remove a user who does not exist in the system.
- **PermissionDeniedException**: Thrown when a user attempts to perform an action without the required permissions.
- **UndefinedPolicyException**: Thrown when trying to apply or reference a policy that has not been defined.
- **UndefinedTypeException**: Thrown when encountering an unrecognized or unsupported type (e.g., for items or policies).
- **UnreachableTargetProfitException**: Thrown when it is impossible to reach a specified target profit under current conditions.
- **UserAlreadyExistsException**: Thrown when trying to add a user with a username that already exists.

2.9 CoreSystem - myFoodora core class

The `CoreSystem` class represents the core backend of the *myFoodora* application. It is implemented as a **singleton**, ensuring that only one instance of the system exists at runtime. It stores and manages the state of users, restaurants, orders, and application-wide policies like delivery and profit strategies.

Attributes:

- **instance (CoreSystem)**: Singleton instance of the system.
- **customers (Map<String, Customer>)**: Stores registered customers.
- **managers (Map<String, Manager>)**: Stores registered managers.
- **couriers (Map<String, Courier>)**: Stores registered couriers.
- **restaurants (Map<String, Restaurant>)**: Stores registered restaurants.
- **orders (List<Order>)**: Stores all orders in the system.
- **customerOrders (List<Order>)**: Stores orders placed by customers.
- **currentUser (Optional<User>)**: The currently connected user.
- **currentUserType (Optional<UserType>)**: Type of the current user.
- **targetProfitPolicy (TargetProfitPolicy)**: Configurable strategy for profit.
- **deliveryPolicy (DeliveryPolicy)**: Strategy for assigning deliveries.
- **serviceFee (double)**: Fixed fee applied to each order.
- **markUpPercentage (double)**: Price markup percentage.
- **deliveryCost (double)**: Base cost of deliveries.

Constructor:

- `CoreSystem()`: Private constructor to enforce the singleton pattern.

Initialization:

- `getInstance()`: Returns the singleton instance, creating it if needed.
- `addDefaultManagers()`: Initializes default privileged users.

Session Management:

- `login(String username, String password)`: Authenticates user and sets session.
- `logout()`: Resets the current session.

Access Control:

- `checkManagerPrivileges()`: Ensures only managers perform sensitive operations; throws `PermissionDeniedException` otherwise.

User Management:

- `addUser(User user)`: Adds a new user to the appropriate collection.
- `removeUser(User user)`: Removes a user.

Order Policy Management:

- `addOrder(Order order)`: Adds a new order to the system.
- `setDeliveryPolicyType(...)`: Configures the delivery strategy.
- `setTargetProfitPolicyType(...)`: Configures the profit policy.

Operational Analytics:

- `computeTotalIncome(...)`: Computes income over a period or globally.
- `computeTotalProfit(...)`: Computes system profit.
- `computeAverageIncomePerCustomer(...)`: Income metric per customer.
- `computeNumberOfOrdersLastMonth()`: Order count over the last month.

Insights and Listings:

- `showSortedCouriers()`: Lists couriers ranked by activity.
- `showSortedRestaurants()`: Ranks restaurants by performance.
- `showCustomers()`: Lists all customers.
- `showMenuItem(String restaurantName)`: Displays menu of a restaurant.
- `showSortedHalfMeals(String restaurantName)`: Ranks half meals of a restaurant.

Courier Handling:

- `getOnDutyCouriers(Map<String, Courier>)`: Returns couriers currently on duty.

Fidelity System:

- `associateCard(String username, FidelityCardType cardType)`: Assigns a fidelity card to a customer using the `FidelityCardFactory`.

The `CoreSystem` class acts as the central hub of the application, maintaining the state of all users, restaurants, orders, and policies. It keeps track of the currently logged-in user and their role, ensuring that every operation first verifies the user's privileges before execution. When a method is called, the system checks if the user has the necessary permissions if so, it delegates the task to the appropriate component or user-specific class. This structure guarantees that responsibilities and logic are clearly separated according to user roles, while unauthorized actions are prevented by raising permission exceptions: `PermissionDeniedException`.

2.10 CoreSystemCLI - Command Line Interface

The `CoreSystemCLI` is a command-line interface (CLI) for *myFoodora*. This class enables users to interact with the system through text-based commands. It acts as the entry point to the application, meaning that running this class starts the system.

The CLI is designed to handle a variety of user interactions, such as logging in, registering, placing orders, and performing administrative operations. It is a key component for testing and demonstrating the core functionalities of the system.

Here is a detailed description of the supported commands:

3 Supported Commands

Below is a detailed description of the available system commands, organized by user roles.

General Commands

- `runTest <testScenarioFile>`
  Execute a test scenario by reading and running commands from a specified file.
- `login <username> <password>`
  Log in using the specified credentials.
- `logout`
  Log out of the current session.
- `help`
  displays the available commands.

Manager Commands

- `registerCustomer <firstName> <lastName> <username> <password>`
  Register a new customer in the system.
- `registerRestaurant <name> <username> <password>`
  Add a new restaurant to the system.
- `registerCourier <firstName> <lastName> <username> <password>`
  Register a new courier for deliveries.
- `showCourierDeliveries`
  Display all deliveries assigned to each courier.
- `showRestaurantsTop`
  Show a ranked list of the top-performing restaurants.
- `showMenuItem <restaurant-name>`
  Display the menu items for the specified restaurant.
- `setDeliveryPolicy <delPolicyName>`
  Set the global delivery policy (e.g., `FASTEST`, `FAIR`).
- `setProfitPolicy <ProfitPolicyName>`
  Set the target profit policy for the system.

- `associateCard <userName> <cardType>`
  Associate a fidelity card to a specific user.
- `showTotalProfit [<startDate> <endDate>]`
  Show the total profit, optionally between specific dates.

Customer Commands

- `createOrder <restaurantName> <orderName>`
  Create a new order at the specified restaurant.
- `addItem2Order <orderName> <itemName>`
  Add an item to an existing order.
- `endOrder <orderName>`
  Finalize and place the order.
- `removeConsensus`
  Remove all previously set consensus communication methods.
- `setConsensusPhone <phone>`
  Set a phone number for receiving SMS notifications.
- `setConsensusMail <mail>`
  Set an email address for receiving notifications.
- `historyOfOrders`
  Display a history of all previous orders.
- `registerFidelityCard <cardType>`
  Register a fidelity card to accumulate rewards.
- `unregisterFidelityCard`
  Remove the registered fidelity card.
- `displayFidelityCardInfo`
  Show details and benefits of the registered fidelity card.

Restaurant Commands

- `showMenu`
  Display the current menu of the restaurant.
- `setSpecialDiscountFactor <specialDiscountFactor>`
  Set the discount factor for special meals.
- `setGenericDiscountFactor <genericDiscountFactor>`
  Set a general discount factor for all meals.
- `addDishRestaurantMenu <dishName> <dishCategory> <foodType> <glutenFree> <unitPrice>`
  Add a new dish to the restaurant's menu with specified properties.
- `createMeal <mealName> <mealType> [<standardOrVegetarian> <GlutenFree>]`
  Create a meal composed of multiple dishes.

- `addDish2Meal <dishName> <mealName>`
  Add a specific dish to an existing meal.
- `showMeal <mealName>`
  Show detailed information about a meal.
- `setSpecialOffer <mealName>`
  Mark a meal as a special offer.
- `removeFromSpecialOffer <mealName>`
  Remove a meal from the list of special offers.
- `showSortedHalfMeals`
  Display a sorted list of meals consisting of only two dishes.
- `showSortedDishes`
  Display all dishes sorted by their name or price.

Courier Commands

- `onDuty <username>`
  Set the courier's status to available for deliveries.
- `offDuty <username>`
  Set the courier's status to unavailable.

4 Test Scenario : DeliveryPolicy

This test focuses on evaluating the system's delivery strategies and the Target Profit Policy.

The scenario begins with the manager (CEO) logging into the system to register a new courier named *Aymen Maftouh*. After completing this task, the CEO logs out.

Next, a customer named `Hmad_lvista` logs in and places three orders. Notably, the first two orders are placed simultaneously, each maintaining its own composition correctly. This demonstrates the system's capability to handle concurrent orders with independent modifications. After placing a third and final order, the customer logs out.

The CEO then logs back in to review the list of couriers and observes that all deliveries were handled by the same courier, which aligns with the default FASTEST delivery policy—assigning deliveries based on proximity. To ensure fair distribution of workload, the CEO changes the delivery policy to FAIR, then logs out.

Subsequently, another customer named Meryam logs in. She updates her preferences by setting a consensus phone number to receive notifications. She then places an order and registers a Point-based fidelity card. Later, she places a second order and successfully earns reward points, thanks to her loyalty membership. Meryam then logs out.

The CEO logs in once more to verify the updated courier assignments and confirms that the workload is now evenly distributed, validating the effectiveness of the FAIR policy. He also checks the system's total profit, which is currently 108. To improve profitability, he updates the Target Profit Policy to aim for a profit of 110, resulting in a calculated markup rate of approximately 0.176.

After logging out, the restaurant BurgerPlace logs in to update its special discount factor and sets *BurgerAndFries* as the new meal of the week. These changes trigger SMS notifications to eligible customers like Meryam, based on her phone notification settings.

When Meryam logs in again, she receives a reminder notification about the current offers. After reviewing the promotions, she logs out.

Finally, the CEO logs in one last time to review the list of restaurants and registered customers, successfully concluding the test scenario.

5 Test Scenario : RestaurantAndMeal

For this test, we wanted to evaluate how the system handles the creation of a new restaurant that is building its menu from scratch.

First, the manager (CEO) logs in and adds `SushiPlace` as a restaurant and `Khozozo` as a customer. Then, they attempt to use the `associateCard` function on a non-existent user, which results in a `UserNotFound` message from the system. Realizing the mistake, the manager corrects the user to `Khozozo`, and the association goes through successfully.

Next, the manager tries to create an order—a command they don't have access to. The system correctly denies the action, stating that only customers have permission to perform it. The manager logs out.

Then, `Frida` logs in, sets her consensus with an email, and logs out.

After that, `SushiPlace` logs back in and starts adding items to the menu: 4 vegetarian dishes, 2 desserts (one gluten-free and one not), a gluten-free main dish, and a gluten-free starter. The restaurant then creates a full meal, marks it as vegetarian and gluten-free, and begins adding dishes to it. However, when attempting to add the non-gluten-free dessert, the system blocks it due to the meal's gluten-free constraint. The restaurant realizes the issue and replaces it with the gluten-free dessert instead. It then displays the meal to verify everything is correct.

Next, the restaurant creates a half-meal using default parameters (standard + non-gluten-free), adds a dessert, and attempts to add a second dessert—something the system doesn't allow. It then tries to add a starter, which is again denied because a half-meal must first have a main dish. The restaurant adds the main dish last and displays the meal to confirm its contents.

Then, it sets the half-meal as the *Meal of the Week*, prompting `Khozozo` to receive a notification email. The restaurant also changes the special discount factor, which `Khozozo` is notified about as well. The menu is displayed again, and the restaurant logs out.

`Khozozo` logs in, receives all pending notifications, and proceeds to place orders. Afterwards, he checks his fidelity card information and logs out.

Finally, `SushiPlace` logs in to see how the menu performed. It views the dishes, half-meals, and full meals—sorted. Then it logs out.

Lastly, the CEO logs in to check profits, restaurant rankings, and couriers.

6 How to Launch MyFoodora

To start the `MyFoodora` CLI application, simply navigate to the `interface` package and run the `Main` class. This class contains the `main` method that initializes the application and launches the interactive console.

Alternatively, you can directly run any compiled class that includes the CLI entry point, and it will open the same interactive session.

Once launched, you can freely explore and use the application via the command-line interface.

To run predefined test scenarios in the CLI, simply use the following command in the console:

```
runTest TestScenario.txt
```

or

```
runTest TestScenario1.txt
```

depending on which scenario you would like to execute. These commands will automatically run through the scripted interactions to simulate and validate key features of the application.

7 Task Distribution

Task	**Assigned To**
Users of myFoodora	Imad
Menus and Meals	Farah
Fidelity Cards	Farah
Delivery Policy	Imad
Notification Service	Farah
Target Profit Policy	Imad
JUnit	Imad
CoreSystem	Imad
CLI	Farah and Imad
Porject's Report	Farah and Imad

Table 1: Task Distribution Table