CentraleSupélec

Object Oriented Project - Foodora

Chaeyeon Lee, Uijin Lee

2EL1580 Artificial Intelligence

Table des matières

1 Introduction

This project presents myFoodora, a food delivery platform inspired by services like Foodora and Deliveroo. The system connects customers with a network of restaurants, allowing them to place orders either à-la-carte or through predefined meal selections. Customers are charged a total fee comprising the restaurant-set order price, a service fee defined by the myFoodora manager, and a markup percentage retained by myFoodora to ensure revenue.

Once an order is placed, the system manages its delivery by assigning it to an available courier. The platform also supports restaurant-defined special offers, customer fidelity plans with discounts, and automated revenue distribution to restaurants and couriers.

myFoodora simulates a complete food delivery workflow with a focus on modular design, extensibility, and realistic business logic.

2 Main Characteristics

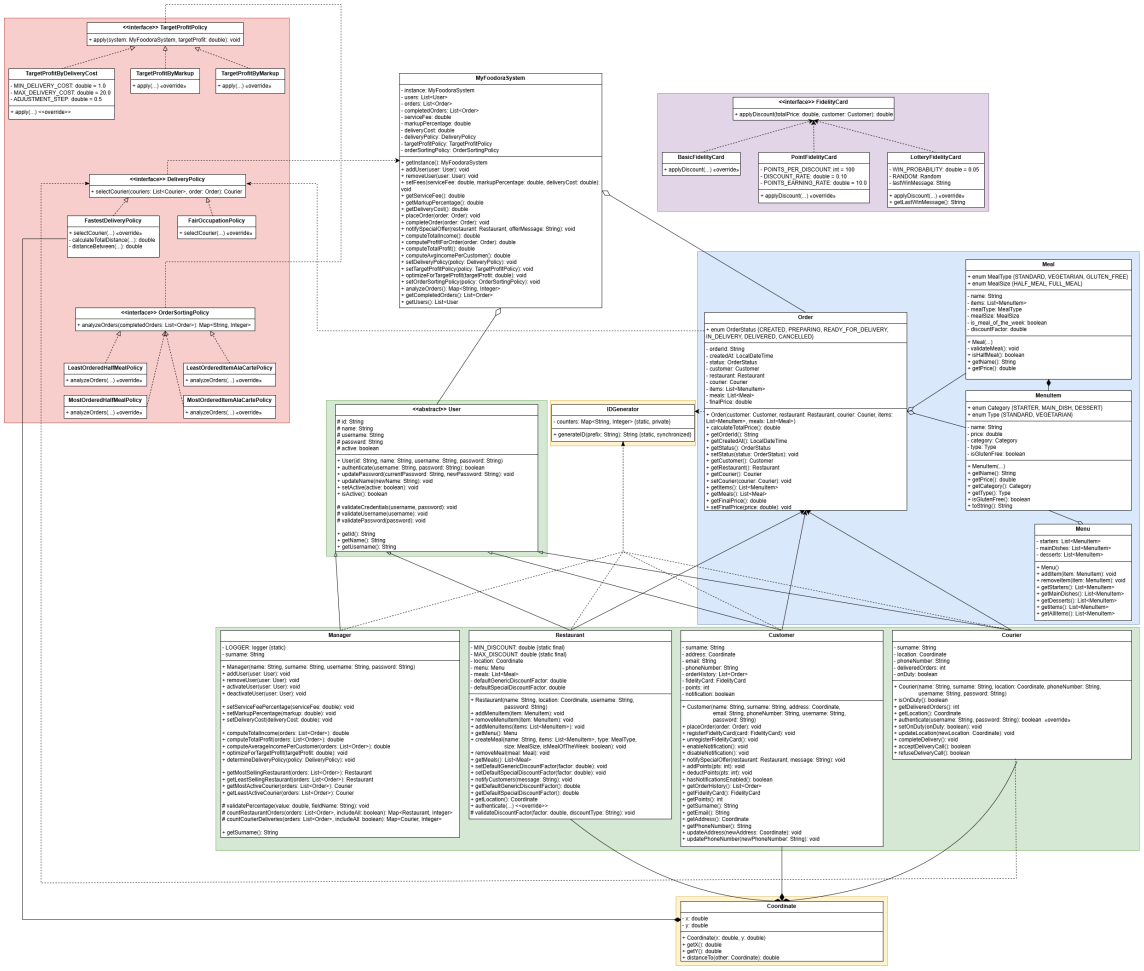

FIGURE 1 – Foodora diagram

The following diagram represents the entire Java project we designed based on our understanding and analysis of the assignment.

2.1 Class Design Explanation

2.1.1 User Module Design Overview

Inheritance Structure The `User` class is an abstract superclass that defines common fields and methods for all users in the `myFoodora` system. It is extended by the following concrete subclasses :

— `Customer` – end user who places orders
— `Courier` – delivery personnel
— `Restaurant` – food providers who manage menus

— `Manager` – system administrators

Shared Functionality (Provided by `User`) All user types inherit the following functionality from the `User` superclass :

— **Identity and Login**
  — Fields : `id` (immutable), `name`, `username`, `password`
— **Authentication and Account Control**
  — `authenticate()` : verifies username and password
  — `updatePassword()`, `updateName()` : update user profile
  — `setActive()`, `isActive()` : activate or deactivate account
  — Internal validation for username (length 3) and password (length 6)

Role-Specific Responsibilities

Customer
— `placeOrder(Order)` : Calculates total price from both meals and items, applies fidelity discount, and stores the order in history.
— `registerFidelityCard(FidelityCard)` : Assigns a fidelity card to the customer (Strategy pattern).
— `unregisterFidelityCard()` : Reverts to a `BasicFidelityCard`, resets loyalty points.
— `addPoints()`, `deductPoints()` : Updates loyalty point balance ; includes validation for non-negative values.
— `notifySpecialOffer(Restaurant, String)` : If notifications are enabled, displays a message about offers.

Courier
— `setOnDuty(boolean)` : Indicates if the courier is available for delivery.
— `updateLocation(Coordinate)` : Sets courier's position for delivery policy calculation.
— `completeDelivery()` : Increments the internal delivery counter.
— `acceptDeliveryCall()`, `refuseDeliveryCall()` : Placeholder for future delivery decision logic.

Restaurant
— `addMenuItem()`, `removeMenuItem()` : Manages dishes in the restaurant's `Menu` ; triggers customer notifications.
— `createMeal()` : Builds a `Meal` object ; validates number and category of items.
— `setDefaultGenericDiscountFactor()`, `setDefaultSpecialDiscountFactor()` : Applies validation to keep discounts between 0%–50%.

— `notifyCustomers(String)` : Sends promotional messages to subscribed customers via system broadcast.

Manager

— `addUser()`, `removeUser()` : Interact with `MyFoodoraSystem` to register or deregister users. Log messages are recorded for each operation.
— `activateUser()`, `deactivateUser()` : Modify user status by invoking the inherited `setActive()` method.
— `setServiceFeePercentage()`, `setMarkupPercentage()`, `setDeliveryCost()` : Update system-wide parameters. All inputs are validated for valid ranges (e.g., percentages between 0 and 1, positive delivery cost).
— `computeTotalIncome()`, `computeTotalProfit()`,

`computeAverageIncomePerCustomer()` : Query metrics from the central system using existing order data.
— `optimizeForTargetProfit()` : Calls the assigned `TargetProfitPolicy` strategy to adjust fee parameters (e.g., service fee, markup) to meet a profit goal.
— `determineDeliveryPolicy()` : Assigns a delivery selection algorithm (e.g., `FastestDeliveryPolicy`).
— `getMostSellingRestaurant()`, `getMostActiveCourier()`, etc. : Use internal helper methods to analyze completed orders and return top-performing restaurants and couriers.

2.1.2 Model Module Design Overview

1. Class Structure and Relationships The `model` package defines the core data entities of the `myFoodora` system. It models food items, menus, meals, and orders as Java objects. Although there is no class inheritance among them, they form a strong compositional structure :

— `MenuItem` is the basic atomic food element used in both `Meal` and `Order`.
— `Meal` is a composed entity containing multiple `MenuItem` objects.
— `Menu` aggregates all available `MenuItems` belonging to a `Restaurant`.
— `Order` represents a customer request, composed of both `Meals` and individual `MenuItems`.

These relationships are realized through containment and association rather than inheritance.

2. Class Responsibilities and Key Functionalities

2.1 `MenuItem`

— Role : Represents a single food item.
— Fields : `name`, `price`, `category`, `type`, `isGlutenFree`.

— Methods :
  — `getPrice()`, `getCategory()`, `isGlutenFree()` — basic accessors.
  — `toString()` : Formats the item's name, category, type, price, and gluten info.

2.2 `Meal`
— Role : Represents a fixed meal composed of multiple `MenuItem`s.
— Validated Properties :
  — Must have 2 or 3 items depending on meal size.
  — Must include a correct mix of categories (starter, main, dessert).
  — All items must be compatible with the declared meal type (e.g., vegetarian).
— Methods :
  — `validateMeal()` : Throws exceptions if the structure or type conditions are violated.
  — `getPrice()` : Sums up item prices and applies discount factor.
  — `isHalfMeal()` : Returns true if the meal is a half-sized meal.

2.3 `Menu`
— Role : Contains all categorized `MenuItem`s for a restaurant.
— Structure :
  — Internally stores starters, main dishes, and desserts in separate lists.
— Methods :
  — `addItem()`, `removeItem()` : Automatically categorizes items by type.
  — `getStarters()`, `getMainDishes()`, `getDesserts()` : Returns respective item categories.
  — `getAllItems()` : Aggregates all items into one immutable list.

2.4 `Order`
— Role : Represents a customer order containing meals and individual items.
— Associated Entities : `Customer`, `Restaurant`, `Courier`, `MenuItem`, `Meal`.
— Methods :
  — `calculateTotalPrice()` : Computes total cost by summing item and meal prices.
  — `setStatus()`, `getStatus()` : Updates and retrieves order status (`OrderStatus` enum).
  — `setFinalPrice()`, `getFinalPrice()` : Records final price after applying discounts.
  — `getItems()`, `getMeals()`, `getCourier()` : Accessors for order contents and participants.

2.1.3 Policy Module Design Overview

1. Module Purpose The `policy` package defines strategy-based behaviors for the `myFoodora` system. It is designed using the Strategy Pattern and provides flexible, interchangeable policies for :

— Selecting couriers (`DeliveryPolicy`)
— Analyzing order frequency (`OrderSortingPolicy`)
— Optimizing for target profit (`TargetProfitPolicy`)

2. Inheritance Structure All policy types are interfaces implemented by several classes. Each interface declares a single method to be implemented :

— `DeliveryPolicy.selectCourier(List<Courier>, Order)`
— `OrderSortingPolicy.analyzeOrders(List<Order>)`
— `TargetProfitPolicy.apply(MyFoodoraSystem, double)`

3. Delivery Policies

FairOccupationPolicy The `selectCourier(List<Courier> couriers, Order order)` method filters on-duty couriers using `courier.isOnDuty()`. It then selects the courier with the fewest completed deliveries using

`Comparator.comparingInt(Courier::getDeliveredOrders)`. This approach promotes a balanced workload among couriers.

FastestDeliveryPolicy The `selectCourier()` method identifies the courier with the shortest total delivery path. It calculates the Euclidean distance from the courier to the restaurant and then to the customer using the helper method `distanceBetween()`. The courier with the lowest total distance is selected, enhancing delivery efficiency.

4. Order Sorting Policies

MostOrderedHalfMealPolicy

The method `analyzeOrders(List<Order> completedOrders)` filters meals using `meal.isHalfMeal()` and counts them with a `HashMap<String, Integer>`. It sorts the map entries in descending order by frequency using `Stream.sorted()` and returns a `LinkedHashMap` to preserve order.

LeastOrderedHalfMealPolicy This class performs the same filtering and counting as `MostOrderedHalfMealPolicy`, but sorts the entries in ascending order. It is useful for identifying unpopular or underperforming half-meals.

MostOrderedItemAlaCartePolicy This class analyzes individual `MenuItems` ordered à la carte. The method iterates over all items in each order, counts their names using a `HashMap`, sorts them in descending order, and returns a `LinkedHashMap`.

LeastOrderedItemAlaCartePolicy This class implements the same logic as `MostOrderedItemAlaCartePolicy` but sorts the map entries in ascending order of frequency. It is used to identify rarely ordered individual items.

5. Target Profit Policies

TargetProfitByServiceFee The method `apply(MyFoodoraSystem system, double targetProfit)` computes the required service fee per order to reach the target profit. It uses total income, number of orders, average price, markup, and delivery cost. If the resulting service fee is negative, it throws an exception. The fee is then applied to the system using `setFees()`.

TargetProfitByMarkup This class keeps the service fee and delivery cost constant and adjusts the markup percentage. The required markup is calculated based on average price and profit per order. If the result is negative, an exception is thrown.

TargetProfitByDeliveryCost The `apply()` method adjusts delivery cost in increments of ±0.5€ depending on whether the current profit is below or above the target. It maintains delivery cost within bounds of 1.0€ to 20.0€, and updates the system using `setFees()`.

2.1.4 Fidelity Program Module Design Overview

1. Module Purpose The `fidelity` package defines a set of loyalty card strategies that determine how discounts are applied when a customer places an order. Using the Strategy Pattern, this module allows the system to assign different discount behaviors without altering the core customer logic.

2. Inheritance Structure All fidelity strategies implement the following interface :

— `double applyDiscount(double totalPrice, Customer customer)`

Inheritance Diagram :

— `FidelityCard` → `BasicFidelityCard`, `PointFidelityCard`,

`LotteryFidelityCard`

3. Fidelity Card Implementations

BasicFidelityCard

— `applyDiscount(double totalPrice, Customer customer)` :
This default implementation applies no discount. The method simply returns the original total price. It is used when no loyalty benefit is assigned to the customer.

PointFidelityCard

— `applyDiscount(double totalPrice, Customer customer)` :
This class implements a point-based reward system. For every 10€ spent, the customer earns 1 point. If the customer accumulates 100 points or more, a 10% discount is applied, and 100 points are deducted. Internally :
  — Calculates earned points : `int earnedPoints = (int) (totalPrice / 10.0)`
  — Adds points via `customer.addPoints(earnedPoints)`
  — If `customer.getPoints() >= 100`, it applies a 10% discount and calls `customer.deductPoints(100)`

LotteryFidelityCard

— `applyDiscount(double totalPrice, Customer customer)` :
This class gives the customer a 5% chance of receiving a free meal. The method generates a random number using a shared `Random` object. If the number is below 0.05, the total price becomes zero. Additionally :
  — A win message is generated and stored in `lastWinMessage`
  — The message can be accessed through `getLastWinMessage()` for UI display

2.1.5 MyFoodoraSystem : Central System Controller

1. Purpose The `MyFoodoraSystem` class is the central coordinator of the myFoodora application. It applies the Singleton pattern to maintain a unique instance throughout the runtime. The system manages users, orders, global policies, and performs financial and strategic analysis. It serves as a bridge between the `user`, `model`, `policy`, and `fidelity` modules.

2. System Integration This class orchestrates the following components :
— **User Management** : Manages all system users including customers, couriers, restaurants, and managers.
— **Order Processing** : Handles placement and completion of orders with courier assignment.
— **Policy Application** : Applies delivery, sorting, and profit optimization policies via interfaces.
— **Financial Calculation** : Computes profit, income, and customer metrics.

— **Notification System** : Sends offers from restaurants to eligible customers.

3. Key Method Descriptions

Singleton Pattern `getInstance()` : Ensures only one instance exists throughout the application using lazy initialization.

User Management `addUser()`, `removeUser()` : Adds or removes a user from the global list.

Fee Management `setFees()` : Updates core parameters : service fee, markup percentage, and delivery cost. `getServiceFee()`, `getMarkupPercentage()`, `getDeliveryCost()` : Accessors.

Order Placement and Completion `placeOrder(Order)` :
— Gathers all couriers from the user list.
— Applies the active `DeliveryPolicy` to select a courier.
— Adds the order to the active list ; throws an error if no courier is available.
`completeOrder(Order)` :
— Moves an order to the completed list.
— Increments the assigned courier's delivery count.

Notifications `notifySpecialOffer(Restaurant, String)` : Filters customers with enabled notifications and sends a message through their `notifySpecialOffer()` method.

Financial Computation `computeTotalIncome()` : Sums up the `finalPrice` of all completed orders.
`computeProfitForOrder(Order)` :

$$\text{profit} = \text{finalPrice} \times \text{markupPercentage} + \text{serviceFee} - \text{deliveryCost}$$

`computeTotalProfit()` : Aggregates all profits from completed orders.
`computeAvgIncomePerCustomer()` : Groups completed orders by customer and returns the average.

Policy Configuration `setDeliveryPolicy()`, `setTargetProfitPolicy()`,

`setOrderSortingPolicy()` : These methods assign new strategies to the system.
`optimizeForTargetProfit(target)` : Calls the current `TargetProfitPolicy` with the target value.
`analyzeOrders()` : Executes the active `OrderSortingPolicy` on all completed orders.

3 Design Decisions

3.1 Overview

The design of the Foodora system was guided by the principles of modularity, extensibility, maintainability, and real-world fidelity. These goals influenced the system architecture, class hierarchies, use of design patterns, and component interactions.

3.2 Key Architectural Goals

— **Single Point of Control** : Ensure centralized management of system-wide data and operations, including user accounts, orders, and policies.
— **Extensibility via Interfaces** : Facilitate the addition of new business logic (e.g., fidelity cards, profit strategies) without altering existing code.
— **Robust User Management** : Clearly differentiate roles and responsibilities among various user types (e.g., customers, couriers, restaurants, managers).
— **Realistic System Simulation** : Reflect real-world relationships, such as customer-to-order links or restaurant-to-menu ownership.
— **Clear Lifecycle Management** : Model system state changes explicitly (e.g., order creation to delivery) to support process tracking and automation.

3.3 Why Hierarchical Class Structures Were Used

The system models diverse user roles, each with unique responsibilities. To avoid duplication and support polymorphism, we defined a base abstract class `User` and extended it into concrete subclasses like `Customer`, `Courier`, `Restaurant`, and `Manager`. This decision ensures :
— Shared attributes (e.g., username, address) are centralized in the base class.
— Specialized behaviors (e.g., delivery for couriers, menu management for restaurants) are defined in specific subclasses.
— The system can process users uniformly using their shared interface where appropriate.

3.4 Why Interfaces and Strategy Patterns Were Chosen

Many business rules in the Foodora system (e.g., delivery selection, profit targeting, fidelity rewards) are expected to change or expand over time. We abstracted such behaviors behind interfaces and designed them to follow the Strategy pattern. This provides :
— **Pluggability** : Administrators can switch policies without modifying core logic.
— **Testability** : Each strategy implementation can be tested in isolation.

— **Maintainability** : Open/Closed Principle compliance—new strategies can be added without changing existing code.

3.5 Why Core Component Relationships Were Explicitly Modeled

System elements like orders, menus, and fidelity cards represent domain objects that must be carefully linked. Explicit modeling through associations, compositions, and aggregations was chosen to :

— Reflect the logical and physical dependencies (e.g., a menu cannot exist without a restaurant).
— Ensure clear ownership and lifecycle control of objects (e.g., deleting a restaurant also deletes its menu).
— Support data integrity and validation rules (e.g., each customer must have exactly one fidelity card).

3.6 Why a Singleton System Controller Was Implemented

To ensure a consistent and globally accessible application state, a singleton `MyFoodoraSystem` controller was introduced. This decision :

— Prevents duplication of system state across components.
— Serves as the centralized hub for business logic, user actions, and data flow.
— Simplifies integration with user interfaces like the command-line UI (CLUI).

3.7 Why Static Initialization Was Used for CEO Setup

The system requires that a default CEO account always exists upon startup for testing, setup, and configuration. To avoid runtime errors and enforce this requirement deterministically, a static initialization block was placed in the CLUI class. This guarantees :

— The CEO is initialized exactly once before any commands are processed.
— The system always starts in a valid administrative state.
— Credential-based testing (e.g., login scenarios) is reproducible.

Conclusion

These design decisions form the backbone of the Foodora architecture. They enable a clear separation of responsibilities, robust process modeling, and long-term flexibility. The following section explains how specific design patterns and class relationships were implemented to realize these goals in practice.

4 Design Patterns

4.1 Overview

The Foodora system applies several software design patterns to ensure robust architecture, consistent behavior, and long-term maintainability. This section outlines two key aspects of the system's design :

1. The **initialization and control** of a single administrative user (CEO) using the **Singleton** and **Static Initialization Block** patterns.
2. The use of **object-oriented class relationships** to structure responsibilities and interactions among system components.

These patterns and relationships collectively contribute to a stable platform.

4.2 Ensuring a Single, Consistent CEO Account

Problem Context

— The CEO account must always exist at system startup.
— Only one such account should ever be created.
— The CEO credentials must remain stable across system restarts for consistent testing and administrative setup.

Patterns Used

Pattern	Class	Purpose
Singleton	`MyFoodoraSystem`	Ensures only one instance of the system exists and provides a global access point.
Static Initialization Block	`CLUI`	Guarantees the CEO account is created exactly once when the class is loaded.

TABLE 1 – Patterns used for system initialization and CEO management

Outcome

These patterns ensure :
— The system starts with a consistent administrative user.
— No duplicate CEO accounts are created, even under concurrency.
— Known credentials are used for setup and testing (e.g., from test input files).

4.3 Object-Oriented Class Relationships

The Foodora system uses core object-oriented relationships to reflect real-world dependencies between users, orders, policies, and the system. These relationships are foundational to its clean and maintainable architecture.

UML Relationship Types and Their Usage

Type	Description
Generalization (Inheritance)	"Is-a" relationship
Realization (Interface Implementation)	A class fulfills a contract from an interface
Composition	Strong ownership ; part cannot exist without whole
Aggregation	Weak ownership ; part can exist independently
Association	Related objects that can exist independently
Dependency	One class temporarily uses another

TABLE 2 – UML relationship types used in the Foodora system

1. Hierarchical Class Structure

User (Abstract) - Generalization

- `Manager` : Administrative capabilities
- `Restaurant` : Menu management, Order processing
- `Customer` : Order creation, FidelityCard association
- `Courier` : Delivery management, Duty status

This represents an "is-a" relationship where each specific user type inherits from the base `User` class.

FidelityCard (interface) - Realization

- `BasicFidelityCard`
- `PointFidelityCard`
- `LotteryFidelityCard`

Each concrete class implements the `FidelityCard` interface and represents a different loyalty program strategy, and it is associated with the `Customer` class through composition.

DeliveryPolicy (interface) - Realization

- `FastestDeliveryPolicy` : Selects courier based on distance
- `FairOccupationPolicy` : Distributes orders evenly among couriers

Implements a strategy pattern for courier selection.

OrderSortingPolicy (interface) - Realization

- `MostOrderedItemAlaCartePolicy`
- `LeastOrderedItemAlaCartePolicy`
- `MostOrderedHalfMealPolicy`
- `LeastOrderedHalfMealPolicy`

Implements a strategy pattern for analyzing order patterns, providing different ways to sort and interpret order data.

TargetProfitPolicy (interface) - Realization
- `TargetProfitByServiceFee`
- `TargetProfitByMarkup`
- `TargetProfitByDeliveryCost`

Each implementation provides a different strategy to adjust fee parameters and optimize system profit.

2. Core System Components and Their Relationships

Menu System
- **Composition** : `Restaurant` strongly owns `Menu`
- **Aggregation** : `Menu` contains `MenuItems` and `Meals`
- **Enumerations** : `MenuItem` categories and types define classification

Order System
- **Associations** : Links `Customer`, `Restaurant`, and `Courier`
- **Composition** : `Order` contains `OrderDetails`
- **Aggregation** : Includes `MenuItems` or `Meals`
- **States** : Managed through `OrderStatus` enumeration

Fidelity System
- **Association** : `Customer` is linked to a `FidelityCard`
- **Implementation** : Supports `Basic`, `Points`, and `Lottery` variants
- **Strategy** : Follows the Strategy pattern to allow interchangeable reward calculations

Dependency
- `CLUI` temporarily uses `MyFoodoraSystem`
- Policy interfaces like `TargetProfitPolicy` or `DeliveryPolicy` dynamically affect system behavior

3. Multiplicities and Constraints

Key multiplicities in the system :
- Restaurant 1 ↔ * Orders
- Customer 1 ↔ * Orders
- Courier 1 ↔ * Orders
- Restaurant 1 ↔ 1 Menu
- Menu 1 ↔ * MenuItems
- Order 1 ↔ * MenuItems
- Customer 1 ↔ 1 FidelityCard

4. Dynamic Relationships

Order Lifecycle

- **Creation** :
  - Customer initiates order with Restaurant (OrderStatus.CREATED)
  - System validates customer and restaurant existence
  - Initial order created with empty items list
- **Processing** :
  - Order moves to PREPARING state
  - Items/meals added to order
  - Price calculation including fidelity discounts
- **Assignment** :
  - System uses DeliveryPolicy to select optimal Courier
  - Order status changes to READY_FOR_DELIVERY
  - Courier assignment triggers status change to IN_DELIVERY
- **Completion** :
  - Courier completes delivery (DELIVERED status)
  - Order moved to completedOrders list
  - Courier's delivery count updated
  - Customer's order history updated
- **Notifications** :
  - Status updates sent to Customer
  - Restaurant notified of new orders
  - Courier receives delivery assignments
  - Special offers broadcast to subscribed customers

Conclusion

The Foodora system demonstrates thoughtful application of both design patterns and object-oriented principles. The use of the **Singleton** and **Static Initialization Block** patterns ensures a reliable and unique administrative user is always present. The comprehensive relationship structure, from inheritance to runtime associations, provides a robust foundation for the system's operations. The clear separation of concerns through interfaces and policy patterns, combined with well-defined object lifecycles and security controls, results in a highly maintainable and extensible architecture.

5 Advantages and Limitations of the Design

In this section, we evaluate the key strengths and limitations of the overall system design. Our goal was to create a scalable, maintainable, and extensible architecture that supports key business logic while remaining realistic and object-oriented.

5.1 Advantages

— **Centralized Control and Data Integrity :** The use of the Singleton pattern for `MyFoodoraSystem` ensures that a single point of control manages all global state, which helps maintain consistency and prevents duplication.
— **High Modularity and Extensibility :** Policies (e.g., `DeliveryPolicy`, `ProfitPolicy`) are abstracted using interfaces and strategy pattern, allowing easy integration of new algorithms without modifying existing logic.
— **Clear Separation of Concerns :** System responsibilities are well distributed across classes such as `CLUI`, `MyFoodoraSystem`, user classes, and policy handlers, making the system easier to understand and maintain.
— **Realistic Object-Oriented Modeling :** The use of association, composition, and dependency mirrors real-world interactions, enhancing code intuitiveness and future adaptability.
— **Automatic Initialization for Admin Access :** The CEO account is reliably created at startup through a static block, ensuring that administrative control is always available for system operations.
— **Encapsulation of Business Rules :** By isolating behaviors like fidelity discounting and delivery strategy, the design supports dynamic runtime behavior without affecting core logic.

5.2 Disadvantages

— **Reduced Flexibility in Initialization :** Hardcoding CEO creation in a static block limits configurability and may make unit testing or system reconfiguration more complex.
— **Global State Risks from Singleton :** The Singleton pattern, while ensuring centralized control, introduces hidden dependencies and can lead to tight coupling, making testing and debugging harder.
— **Potential Overhead from Strategy Abstractions :** While flexible, the strategy-based design for various policies introduces additional classes and boilerplate code, which could be unnecessary for smaller-scale deployments.
— **Scalability Constraints :** The system is built as a centralized monolith, which may need refactoring to support distributed architectures or concurrent multi-user operations in a real-world deployment.
— **Rigid User Role Management :** Predefining user roles and their capabilities may limit dynamic customization and user-specific behaviors unless

further role abstraction is introduced.

6 Test Scenario Description

Purpose This test scenario suite verifies the core functionalities of the myFoodora platform via CLI simulation using JUnit. It focuses on validating commands processed through the `CLUI.executeCommand(String, String[])` interface, simulating realistic user interactions such as login, menu creation, order placement, and policy reporting.

Scenario Summary All test methods simulate usage from a command-line interface. A `ByteArrayOutputStream` is used to capture printed output for assertion. `@BeforeEach` and `@AfterEach` ensure test isolation. Each test case emulates a high-level business operation.

Scenario Breakdown (Function-Oriented)

testSetup()
— **Commands :** `login`, `setup`
— Logs in as manager and sets up the system with default parameters.
— Verifies output contains initialization confirmation.

testLoginLogout()
— **Commands :** `login`, `logout`
— Tests valid login/logout flow.
— Tests invalid credentials raise exception with message `"Invalid credentials"`.

testRegisterRestaurant()
— **Command :** `registerrestaurant`
— Registers a new restaurant, then attempts duplicate registration.
— Expects exception `"Restaurant already exists"`.

testRegisterCustomer()
— **Command :** `registercustomer`
— Registers a customer and re-tests with same credentials.
— Expects exception `"Customer already exists"`.

testRestaurantMenu()
— **Commands :** `adddishrestaurantmenu`, `createmeal`, `adddish2meal`, `savemeal`, `setspecialoffer`
— Adds a dish, composes a meal, and marks it as a special offer.
— Tests end-to-end menu lifecycle for a restaurant.

testOrderProcess()
— **Commands :** `createorder`, `additem2order`, `endorder`
— Customer places and completes an order.
— Tests courier assignment, price finalization, and fidelity discount logic.

testCourierOperations()
— **Commands :** `registercourier`, `onduty`, `offduty`
— Registers and logs in as courier.
— Tests toggling of duty status.

testManagerOperations()
— **Commands :** `setdeliverypolicy`, `setprofitpolicy`, `showcourierdeliveries`, `showrestauranttop`, `showcustomers`, `showtotalprofit`
— Applies delivery and profit policies.
— Retrieves analytics and profit summaries from system.

7 Workload Distribution

- myFoodora Core functionalities
  - User, Menu, Order, myFoodora - Chaeyeon Lee
  - Policy - Uijin Lee
- myFoodora CLUI functionalities - Chaeyeon Lee
- JUnit tests - Chaeyeon Lee
- UML Diagram - Uijin Lee
- Final report
  - Introduction, Design Decisions, Design Patterns, Advantages and Limitations - Chaeyeon Lee
  - Main Characteristics, Test Scenario Description - Uijin Lee

CentraleSupélec