Restaurant Ordering System
Final Project Report

Course: Software Design Patterns

Instructor: Mukhidenov Diyar

Team:

- Aset Syrgabaev
- Kuanysh Asaubaev
- Nuradilet Mustafa

1. Introduction

This report presents the final project for the *Software Design Patterns* course.

Our group implemented a **Restaurant Ordering System** that demonstrates the practical use of multiple design patterns from the course.

The main objectives of the project are:

- To implement **at least six design patterns** in a single cohesive system.
- To cover **all three categories** of patterns: creational, structural and behavioral.
- To design clean, extensible, and maintainable code in **Java 17**.
- To provide a console-based application that models a realistic restaurant workflow.

The system allows a waiter to create orders for tables, add meals with different sizes and toppings, apply various discount strategies, and track order status with notifications for both the kitchen and customers.

2. System Overview

The Restaurant Ordering System simulates the behaviour of a small restaurant:

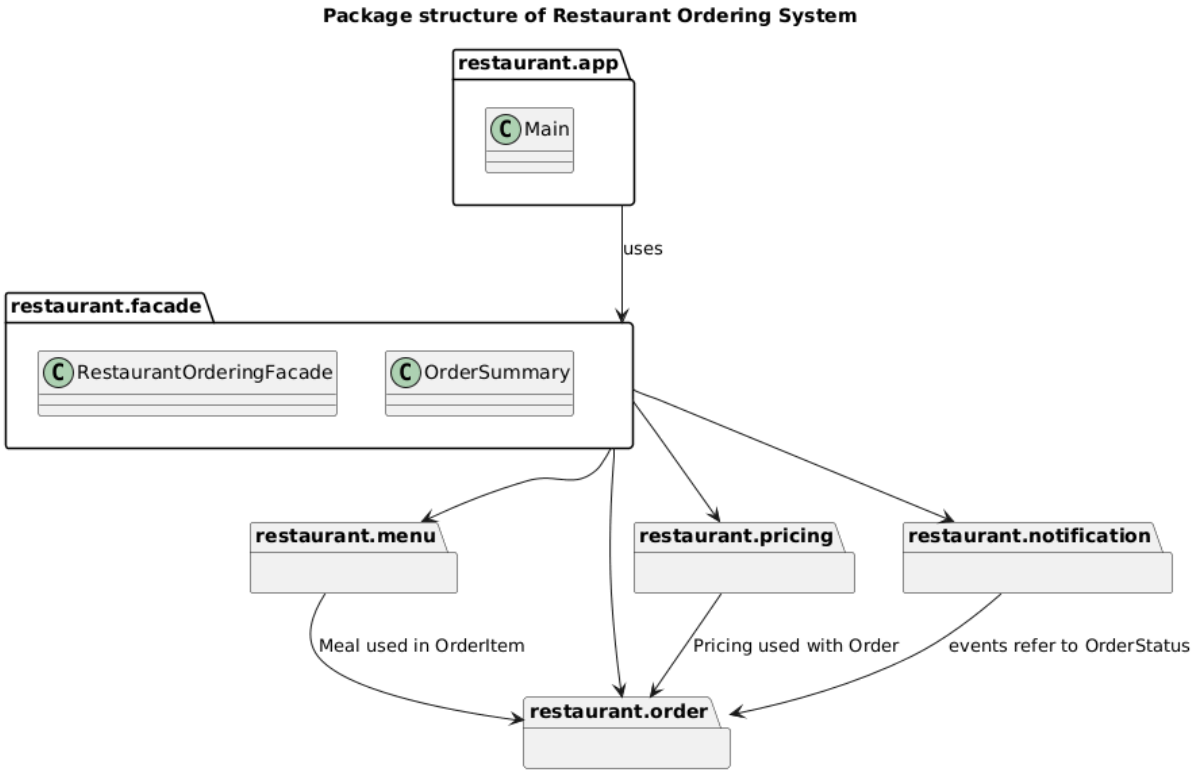
- The waiter creates a new order for a specific table.

- The waiter selects meals from the menu, chooses size and optional toppings.
- A pricing strategy is selected (no discount, student discount, weekend discount).
- The order status changes during its lifecycle (NEW, PLACED, IN_PREPARATION, READY, COMPLETED, CANCELLED).
- The kitchen and customer receive notifications when the order is placed or when its status changes.

The user interacts with the system through a **console menu** provided by the `Main` class.

All application logic is accessed through the `RestaurantOrderingFacade`, which hides the internal complexity of the subsystems.

Insert main package overview diagram:
(`package-diagram.png`)



3. Architecture and Package Structure

The project is organised under the root package `restaurant` with the following subpackages:

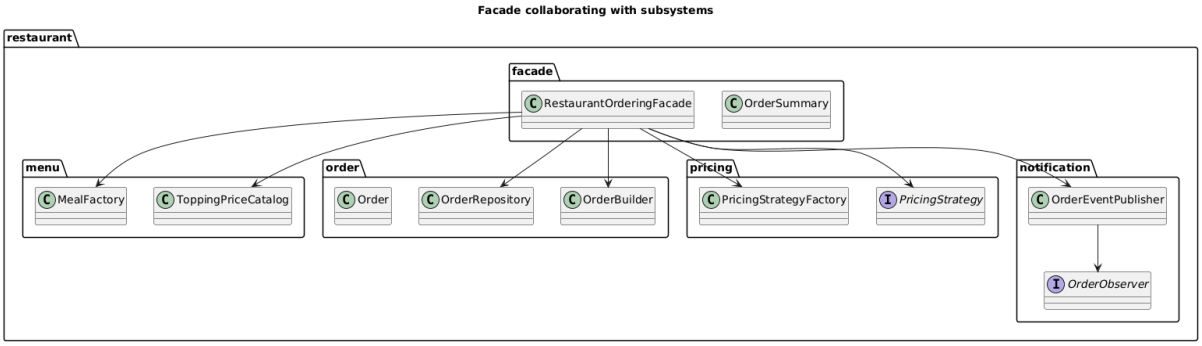
- `restaurant.app` – console user interface.

- `restaurant.facade` – facade layer exposing the main operations of the system.
- `restaurant.menu` – menu model, factory, and topping decorators.
- `restaurant.order` – order domain model, builder and repository.
- `restaurant.pricing` – pricing strategies and strategy factory.
- `restaurant.notification` – event and observer model for notifications.

This structure separates responsibilities clearly and keeps the domain logic independent from the user interface.

3.1 Package-Level Architecture

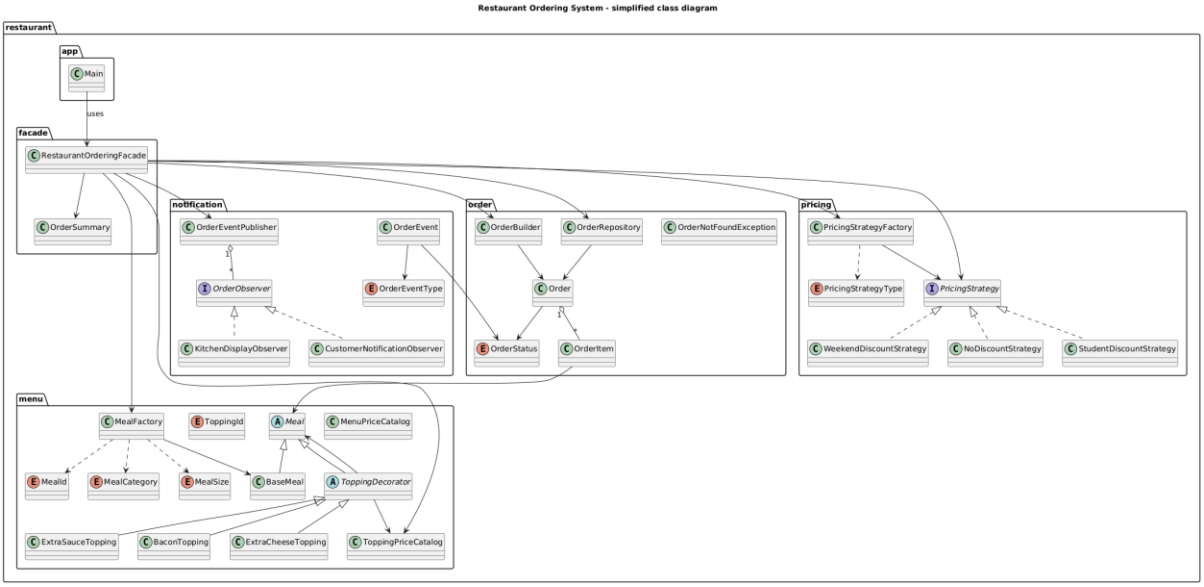
The package diagram below shows how the `app` layer uses the `facade`, and how the facade coordinates other packages.
(`facade-diagram.png`)



3.2 Core Class Model

The core classes and their relationships (meals, orders, pricing, notifications and facade) are illustrated in the following class diagram:

(simple-class-diagram.png)



4. Implemented Design Patterns

The project implements six classic design patterns:

- Factory Method (creational)
- Builder (creational)
- Decorator (structural)
- Facade (structural)
- Strategy (behavioral)
- Observer (behavioral)

  Each pattern is used in a realistic context, and patterns collaborate through the `RestaurantOrderingFacade`.

4.1 Factory Method – Meal Creation

Intent:

Provide an interface for creating objects while allowing subclasses or a factory method to decide which concrete class to instantiate.

Implementation:

- `restaurant.menu.Meal` – abstract base class for all meals.
- `restaurant.menu.BaseMeal` – concrete meal implementation.

- `restaurant.menu.MealId` – enum listing the available meals (e.g., `MARGHERITA_PIZZA`, `CHEESEBURGER`, `COLA`).
- `restaurant.menu.MenuPriceCatalog` – centralised price catalog for meals.
- `restaurant.menu.MealFactory` – factory that creates `Meal` instances based on `MealId` and `MealSize`.

`MealFactory` maps a `MealId` to a category (`MealCategory`), a user-friendly name, and retrieves the base price from `MenuPriceCatalog`. It returns an appropriately configured `BaseMeal` object.

Benefits:

- Client code does not depend on concrete constructors.
- Adding a new meal type requires changes only in `MealId`, `MenuPriceCatalog`, and `MealFactory`.

4.2 Builder – Order Construction

Intent:

Separate the construction of a complex object from its representation, allowing step-by-step creation.

Implementation:

- `restaurant.order.Order` – represents a restaurant order (id, table number, status, items, pricing strategy code).
- `restaurant.order.OrderItem` – record containing `Meal` and `quantity`.
- `restaurant.order.OrderStatus` – enum describing the lifecycle stages of an order.
- `restaurant.order.OrderBuilder` – Builder used to construct `Order` instances.
- `restaurant.order.OrderRepository` – in-memory storage for orders.

The facade uses the builder as follows:

```
OrderBuilder builder = new OrderBuilder()
    .withId(id)
    .forTable(tableNumber)
    .withStatus(OrderStatus.NEW)
    .withPricingStrategyCode(strategyType.name());
Order order = builder.build();
```

Benefits:

- The `Order` constructor is simple and not overloaded with parameters.
- The risk of creating invalid orders is reduced.
- Adding new properties to `Order` is easier and does not break existing code that uses the builder.

4.3 Decorator – Topping Add-Ons

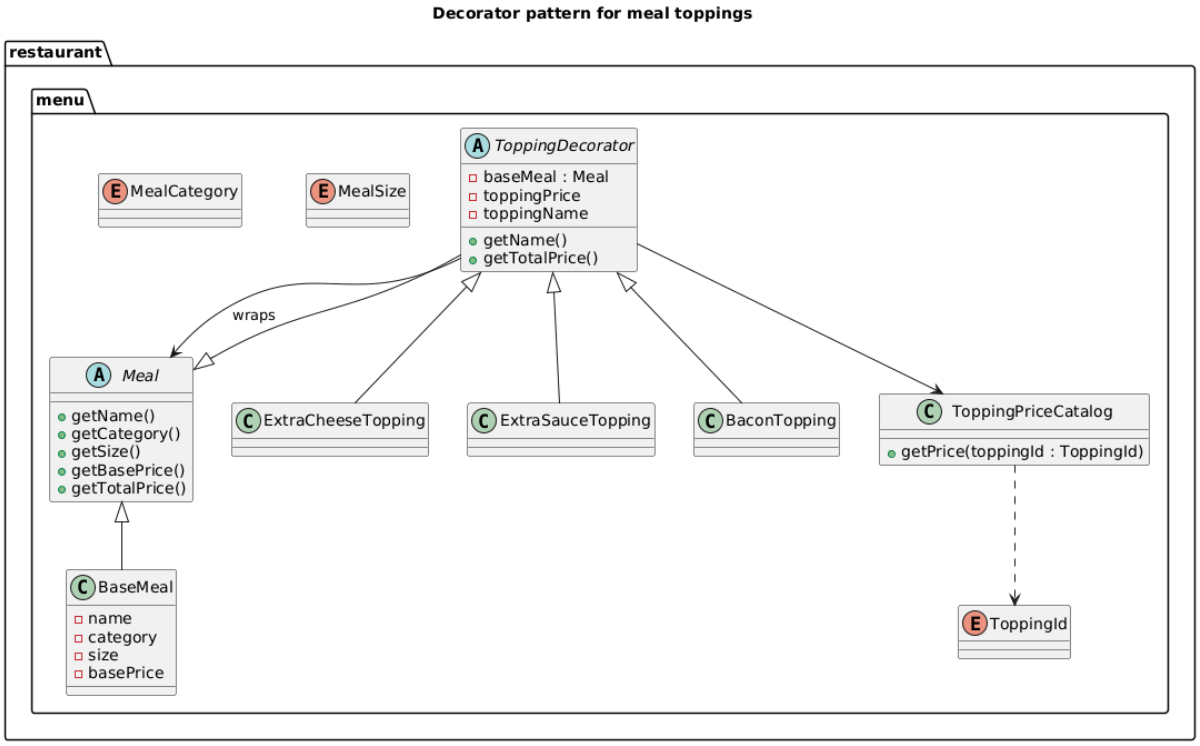
Intent:

Attach additional responsibilities to an object dynamically. Decorators offer a flexible alternative to subclassing.

Implementation:

- `restaurant.menu.Meal` – component base class.
- `restaurant.menu.BaseMeal` – concrete component.
- `restaurant.menu.ToppingDecorator` – abstract decorator extending `Meal` and wrapping a `Meal` instance.
- Concrete decorators:
  - `ExtraCheeseTopping`
  - `ExtraSauceTopping`
  - `BaconTopping`
- `restaurant.menu.ToppingId` – enum listing topping types.
- `restaurant.menu.ToppingPriceCatalog` – catalog of topping prices.

The decorator structure is illustrated in the following diagram:
(`decorator-diagram.png`)



A decorated meal is formed by wrapping a base meal in one or more decorators, e.g.:

- `BaseMeal` → `ExtraCheeseTopping` → `BaconTopping`

Each decorator adds its own topping price to the total and extends the name (e.g., “Margherita Pizza + extra cheese + bacon”).

Benefits:

- New toppings can be added by introducing new decorator classes without modifying existing code.
- Multiple toppings can be combined at runtime in any order.

4.4 Strategy – Pricing and Discounts

Intent:

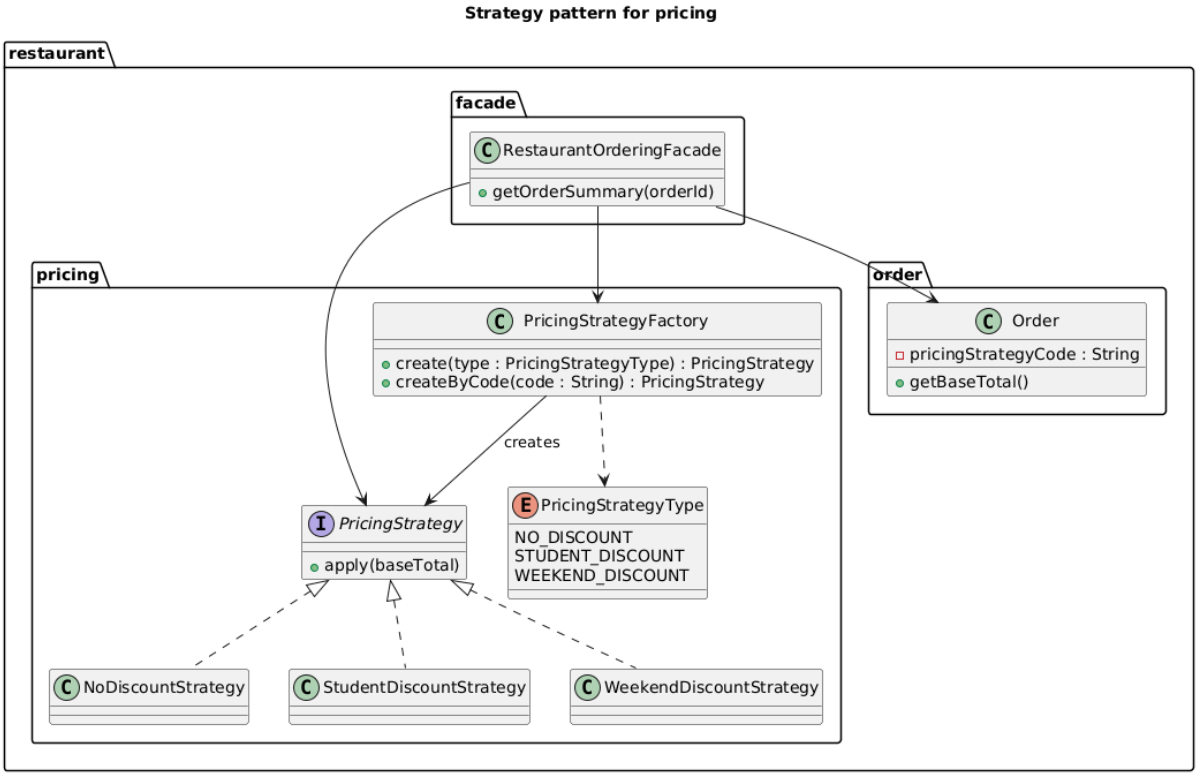
Define a family of algorithms, encapsulate each one, and make them interchangeable.

Implementation:

- `restaurant.pricing.PricingStrategy` – interface with method `apply(BigDecimal baseTotal)`.
- `restaurant.pricing.PricingStrategyType` – enum: `NO_DISCOUNT`, `STUDENT_DISCOUNT`, `WEEKEND_DISCOUNT`.
- Concrete strategies:
  - `NoDiscountStrategy`
  - `StudentDiscountStrategy` (10% discount)
  - `WeekendDiscountStrategy` (15% discount)
- `restaurant.pricing.PricingStrategyFactory` – returns a strategy based on the chosen type or stored code.

The strategy design is shown below:

(strategy-diagram.png)



The facade retrieves the base total from the `Order`, obtains the correct strategy from `PricingStrategyFactory`, and computes the final total using `strategy.apply(baseTotal)`.

Benefits:

- New discount policies can be introduced without changing order or facade classes.
- Pricing logic is encapsulated and easy to test independently.

4.5 Observer – Order Notifications

Intent:

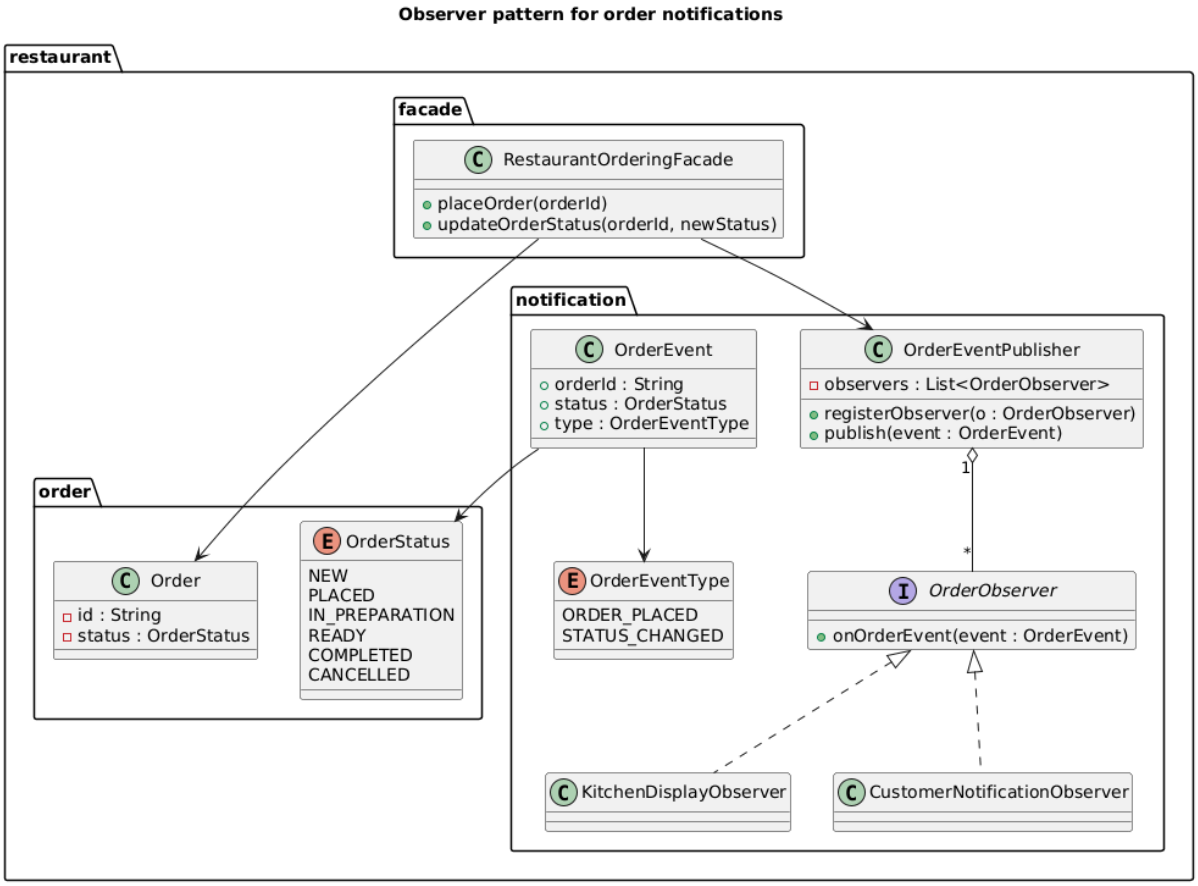
Define a one-to-many dependency so that when one object changes state, all dependents are notified and updated automatically.

Implementation:

- `restaurant.notification.OrderEventType` – enum: `ORDER_PLACED`, `STATUS_CHANGED`.
- `restaurant.notification.OrderEvent` – record holding `orderId`, `OrderStatus`, and `OrderEventType`.
- `restaurant.notification.OrderObserver` – observer interface.
- `restaurant.notification.OrderEventPublisher` – subject managing a list of observers.
- Concrete observers:
    - `KitchenDisplayObserver` – prints kitchen-related notifications.
    - `CustomerNotificationObserver` – prints customer-facing notifications.

The observer structure is presented in this diagram:

(observer-diagram.png)



In `RestaurantOrderingFacade`:

- On `placeOrder`, the order status becomes `PLACED` and an `ORDER_PLACED` event is published.
- On `updateOrderStatus`, a `STATUS_CHANGED` event is published.
- The publisher notifies both observers, which log messages such as:
  - `[KITCHEN] Order <id> event: ORDER_PLACED status: PLACED`
  - `[CUSTOMER] Order <id> is now READY`

Benefits:

- Notification logic is decoupled from order processing.
- New observers (e.g., logging, analytics, external integrations) can be added without modifying existing code.

4.6 Facade – Unified Interface for Subsystems

Intent:

Provide a simple interface to a complex subsystem.

Implementation:

- `restaurant.facade.RestaurantOrderingFacade` – the central entry point used by the UI.
- `restaurant.facade.OrderSummary` – record used to present aggregated order information to the user.

The facade coordinates:

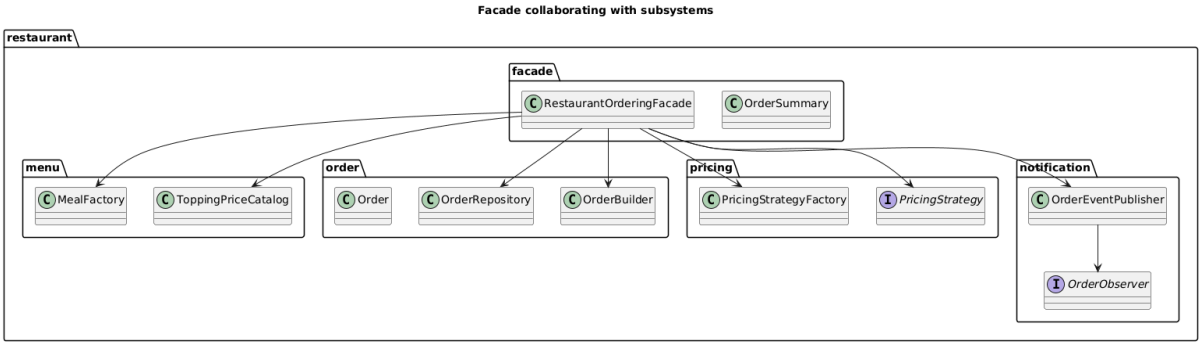
- `MealFactory` and decorators for building meals with toppings.
- `OrderBuilder` and `OrderRepository` for managing orders.
- `PricingStrategyFactory` and strategies for discount calculations.
- `OrderEventPublisher` and observers for notifications.

The facade has the following main operations:

- `createOrder(tableNumber, strategyType)`
- `addMealToOrder(orderId, mealId, size, toppings, quantity)`
- `changePricingStrategy(orderId, type)`
- `placeOrder(orderId)`
- `updateOrderStatus(orderId, newStatus)`
- `getOrderSummary(orderId)`
- `getAllOrders()`

The collaboration between the facade and subsystems is summarised here:

(`facade-diagram.png`)



Benefits:

- The `Main` class is very simple and interacts only with the facade.
- Subsystems remain loosely coupled and can evolve without affecting the UI.

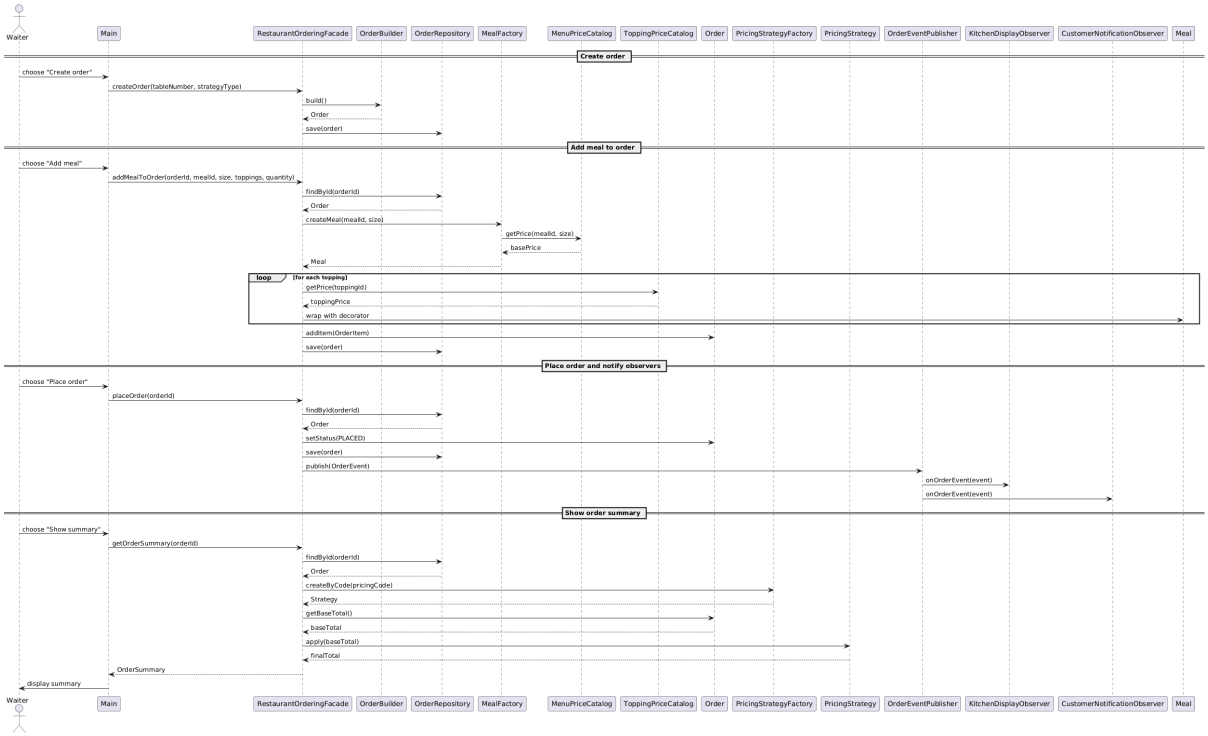
5. Main Scenario – Sequence of Interactions

The main use case covers these steps:

1. The waiter creates a new order for a table and selects a pricing strategy.
2. The waiter adds a meal with a specific size and optional toppings.
3. The order is placed, and observers are notified.
4. The waiter requests an order summary, viewing base total and final total after discount.

The following sequence diagram shows the interaction between the waiter (actor), `Main`, the facade, and the main subsystems:

(`sequence-diagram.png`)



6. Implementation Details and Screenshots

6.1 Technologies and Practices

- Programming language: **Java 17**.
- Project structure compatible with IntelliJ IDEA.
- Clean code practices:
  - clear package separation,
  - use of records for immutable data (`OrderItem`, `OrderEvent`, `OrderSummary`),
  - centralised price catalogs instead of scattered magic numbers,
  - use of enums for identifiers and statuses.

6.2 Console Screenshots (to be added)

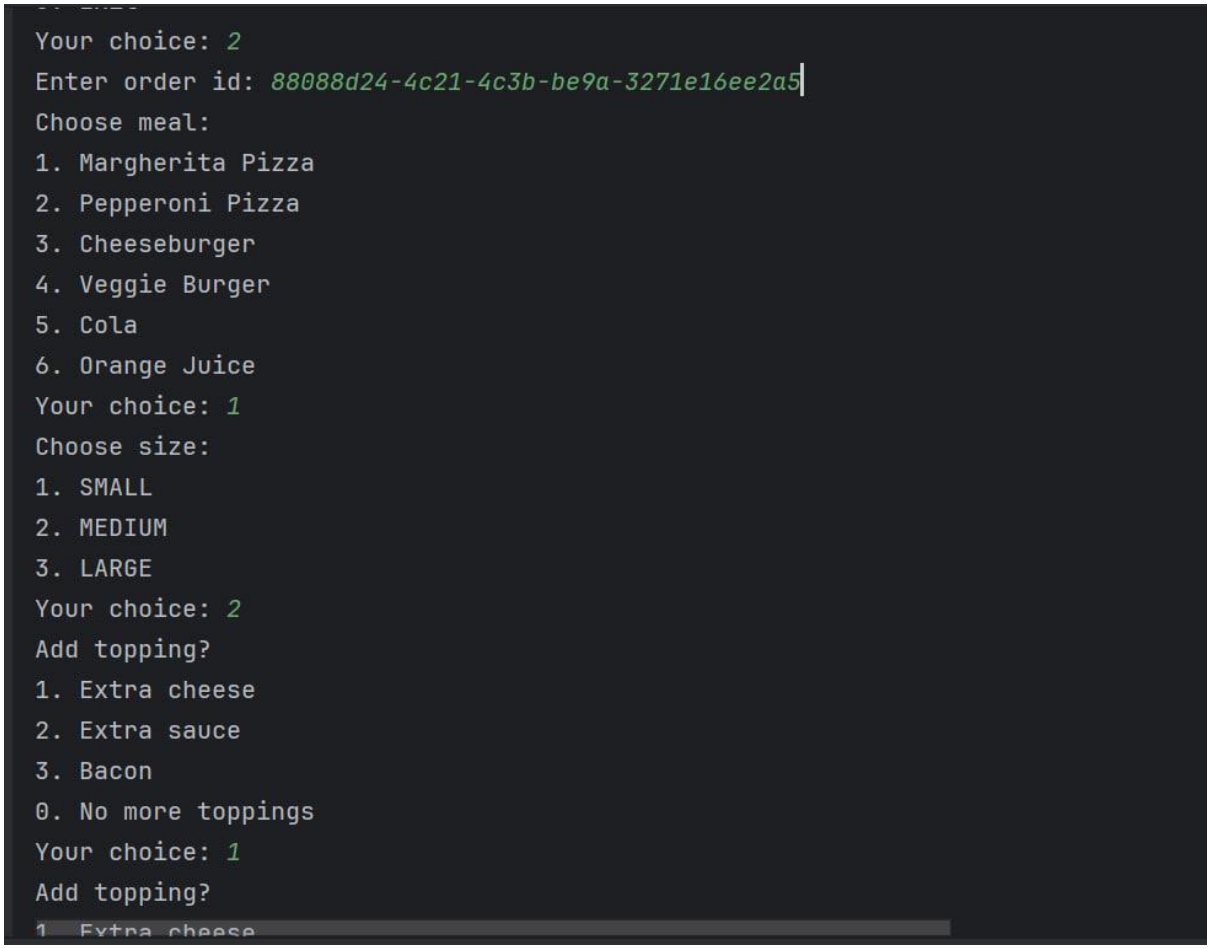
In this section you can insert screenshots of the running console application, for example:

- Main menu view.
- Creating a new order.
- Adding a meal with toppings.
- Placing an order and showing observer messages.
- Displaying order summary with base total and final total.

```
Restaurant Ordering System started

Choose action:
1. Create new order
2. Add meal to order
3. Change pricing strategy
4. Place order
5. Update order status
6. Show order summary
7. List all orders
0. Exit
Your choice:
```

```
Your choice: 1
Enter table number: 12
Choose pricing strategy:
1. No discount
2. Student discount
3. Weekend discount
Your choice: 2
Order created with id: 88088d24-4c21-4c3b-be9a-3271e16ee2a5
```



7. Conclusion

In this project we designed and implemented a **Restaurant Ordering System** that integrates multiple design patterns within a cohesive Java application.

Key outcomes:

- Implemented six classic design patterns: **Factory Method, Builder, Decorator, Strategy, Observer, Facade**.
- Covered all three pattern categories (creational, structural, behavioral).
- Created a clear package structure with separated responsibilities.
- Demonstrated how design patterns lead to flexible and maintainable code.
- Provided a working console application that clearly showcases the behaviour of each pattern.

The project satisfies the course requirements and serves as a solid example of applying design patterns in a realistic domain.

8. Further Work

The current implementation focuses on the core logic and pattern usage. Several improvements and extensions are possible:

1. **Persistent Storage**

Replace the in-memory `OrderRepository` with a database (e.g., JDBC, JPA) and persist menu data.

2. **Graphical or Web User Interface**

Implement a GUI (JavaFX) or a web frontend (REST API with a web client), reusing the existing facade as an application service layer.

3. **Authentication and Roles**

Add users such as waiters, cashiers, and managers, with appropriate permissions.

4. **Advanced Pricing**

Introduce additional strategies, such as happy-hour discounts, loyalty points, or dynamic pricing depending on time and load.

5. **Improved Notifications**

Extend the Observer-based notification system to send emails, push notifications, or integrate with external queueing systems.

These extensions would reuse and validate the flexibility of the current architecture and demonstrate the power of design patterns in real-world evolution of software systems.