Restaurant Ordering System – Project Report SE-2437

Alym Baisal

Nygman Ibrahim

Madyshev Dias

1. Introduction

The Restaurant Ordering System is a Java-based application designed to simulate a real-world restaurant environment. The project demonstrates the practical implementation of multiple design patterns in a single cohesive system. This system allows creating customizable meals, adding toppings, managing order status notifications, and calculating prices dynamically using various strategies.

Objectives:

- Implement at least six design patterns, covering structural, behavioral, and creational types.
- Demonstrate individual contributions of students while integrating patterns into a unified system.
- Produce a working Java application with clean code and clear architecture.

2. Body

2.1 Project Structure

The project is organized using Java packages as follows:

```
com.restaurant
 ├– app // Main class for demonstration
 ├– meal // Factory pattern: meal creation
 ├– decorator // Decorator pattern: toppings
 ├– observer // Observer pattern: order notifications
 ├– strategy // Strategy pattern: pricing and discounts
 ├– facade // Facade pattern: simplified interface
└ bridge // Bridge pattern: delivery abstraction
```

2.2 Design Patterns Implemented

1. Factory Pattern

- Purpose: To create different types of meals (Pizza, Burger, Pasta) without exposing the creation logic to clients.
- Implementation: MealFactory class provides a createMeal(String type) method that returns the requested meal instance.

2. Decorator Pattern

- Purpose: To allow dynamic addition of toppings to meals (e.g., Cheese, Meat, Extra Sauce) without modifying the original meal classes.
- Implementation: MealDecorator abstract class, with concrete decorators like CheeseDecorator, MeatDecorator, ExtraSauceDecorator.

3. Observer Pattern

- Purpose: To notify different stakeholders (kitchen and customer) about order status updates in real time.
- Implementation: Order class acts as the subject; KitchenObserver and CustomerObserver are subscribers implementing the Observer interface.

4. Strategy Pattern

- Purpose: To allow flexible calculation of meal prices and discounts. Multiple pricing algorithms can be applied at runtime without modifying the meal classes.
- Implementation: Separate PricingStrategy interface with implementations like StandardPricing, DiscountPricing, VIPPricing.

5. Facade Pattern

- Purpose: To provide a simplified interface for placing an order, adding toppings, and processing payments.
- Implementation: OrderFacade class encapsulates multiple subsystems and provides a single method placeOrder() for clients.

6. Bridge Pattern

- Purpose: To decouple the delivery abstraction from its implementation, allowing different delivery methods (Courier, Table, Takeaway, Self-Pickup) without changing the core order logic.
- Implementation: DeliveryType abstract class (CourierDelivery, TableDelivery, TakeawayDelivery, SelfPickupDelivery) works with DeliveryImplementor interface (Courier, Table, Takeaway, SelfPickup) to execute the actual delivery.

2.3 Main Program Demonstration

The Main.java class demonstrates the integration of Factory, Decorator, Observer, and Bridge patterns:

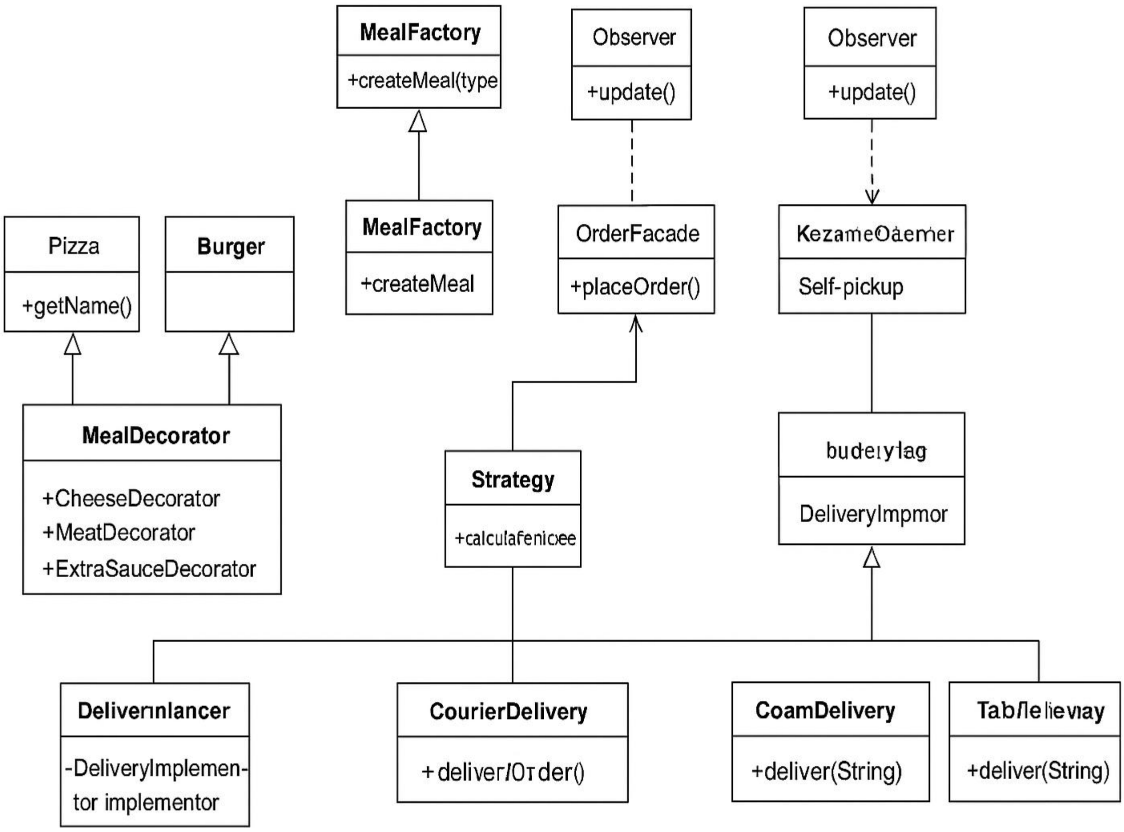
```
MealFactory factory = new MealFactory();
Meal meal = factory.createMeal("pizza");

meal = new CheeseDecorator(meal);
meal = new MeatDecorator(meal);
meal = new ExtraSauceDecorator(meal);

System.out.println("Your order: " + meal.getName());
System.out.println("Total price: $" + String.format("%.2f", meal.getPrice()));

Order order = new Order();
order.addObserver(new KitchenObserver());
order.addObserver(new CustomerObserver());

order.setStatus(OrderStatus.COOKING);
order.setStatus(OrderStatus.READY);
order.setStatus(OrderStatus.COMPLETED);

Delivery courierDelivery = new CourierDelivery(new Courier());
courierDelivery.deliverOrder(meal.getName());

Delivery tableDelivery = new TableDelivery(new Table());
tableDelivery.deliverOrder(meal.getName());
```

Expected Output:

```
Your order: Pizza + Cheese + Meat + Extra Sauce

Total price: $12.79

[Kitchen] New status: COOKING

[Customer] Your order is now: COOKING

[Kitchen] New status: READY

[Customer] Your order is now: READY

[Kitchen] New status: COMPLETED

[Customer] Your order is now: COMPLETED

Courier delivers: Pizza + Cheese + Meat + Extra Sauce

Serve at table: Pizza + Cheese + Meat + Extra Sauce
```

2.4 UML Diagram

UML diagram illustrates relationships between classes and patterns (Factory, Decorator, Observer, Strategy, Facade, Bridge for delivery).



3.Conclusion

The Restaurant Ordering System effectively demonstrates the use of multiple design patterns in a real-world scenario. Each pattern serves a specific purpose: Factory and Decorator provide modular and extendable meal creation; Observer ensures stakeholders receive real-time updates; Strategy allows dynamic pricing flexibility; Facade simplifies system usage for clients; Bridge enables flexible delivery methods decoupled from core logic. The system is structured, maintainable, and easily extendable.

4. Further Work

Future improvements could include:

- Adding a GUI interface for interactive meal selection.
- Implementing database integration to persist orders and menu items.
- Extending Strategy to include complex promotions and loyalty points.
- Introducing Command pattern for undo/redo operations.
- Adding unit tests for automated validation of design pattern implementations.

Our work:https://github.com/redd1eg/RestaurantOrderingSystem.git