Design Decisions Document

1. Overview

This project implements a restaurant ordering system using multiple design patterns to achieve clean architecture, scalability, and maintainability. The main patterns include: - **Factory Pattern**: For menu and item creation. - **Decorator Pattern**: For customizing items with extras. - **Strategy Pattern**: For applying dynamic discounts to menu items. - **Observer Pattern**: For notifying kitchen and waiter components when an order is placed. - **Facade Pattern**: To simplify the ordering workflow for clients.

Each pattern was chosen carefully to separate responsibilities, reduce coupling, and allow future expansion with minimal code changes.

---

2. Strategy Pattern Usage

Why Strategy Pattern?

Different items in the menu may require different discount behaviors. Instead of hard-coding the discount logic inside items, the Strategy Pattern allows attaching different discount algorithms dynamically.

Design Choice

A dedicated **Discount Context** class was created: - Holds a reference to the strategy. - Applies the discount through `apply(price)`. - Allows changing strategy at runtime using `setStrategy()`.

Each menu item (e.g., `CezarSalad`) contains a `Discount` context and assigns a default discount strategy. This isolates discount behavior and allows custom discount logic per item.

Benefits

- Clean separation of item data and discount logic.
- Ability to swap discount behavior dynamically.
- Easy to add new discounts without modifying existing classes.

---

3. Decorator Pattern Usage

Why Decorator Pattern?

Customers can add extras (cheese, sauces, toppings). Each extra affects: - Price - Description

Instead of duplicating dozens of item variations, the Decorator Pattern enables dynamically layering extras around a base item.

Design Choice

Customizations are implemented as decorators: - `ExtraCheeseDecorator` - `SaucesDecorator` - `ToppingsDecorator`

Each decorator wraps an `IMenuItem` and modifies price + description.

Benefits

- Infinite combinational extras.
- Avoids subclass explosion.
- Extensible without affecting base item classes.

---

4. Factory Pattern Usage

Why Factory Pattern?

Each menu type (Vegetarian, Non-Vegetarian, Children) contains different items. The creation logic must not be placed inside the Facade.

Design Choice

- A `MenuFactory` interface.
- Specific implementations such as `VegetarianMenuFactory`.
- Each factory returns an `IMenu`, and the menu can create individual items.

Benefits

- Changing or adding menus requires no modification to the Facade.
- Each menu encapsulates its own item creation logic.

---

5. Facade Pattern Usage

Purpose

The Facade simplifies the complex internal logic: - Selecting menu - Choosing items - Adding decorators - Choosing payment method - Submitting order

The Facade hides complex interactions and exposes a single simple API (`makeOrder()`).

---

6. Observer Pattern Usage

Why Observer?

When a customer confirms an order: - Kitchen must be notified to prepare - Waiter must be notified to serve

Design Choice

`Manager` acts as the subject: - Holds observers such as `Kitchen` and `Waiter`. - Calls `update()` on all observers when a new order is placed.

Benefits

- Extensible: add delivery service observer later.
- Loose coupling between order processing and staff notification.

---

7. Responsibility Separation Summary

Component	Responsibility
MenuItem	Defines item data only (name, price, description).
Discount	Applies discount via strategy.
Decorator	Adds dynamic customizations.
MenuFactory	Creates menu items.
IMenu	Lists items + creates them by name.
Facade	Manages full user workflow.
Manager (Subject)	Notifies observers.
Observers	React to new orders.

This modular architecture ensures that each part of the system evolves independently.

8. Future Extension Possibilities

The chosen patterns make the system easy to extend: - Add new extras (new decorator). - Add new discounts (new strategy). - Add new menus (new factory + menu class). - Add delivery notification (new observer).

The system is designed for scalability and flexibility.

9. Class Diagram