Advanced Software Engineering - Assignment 1

Restaurant Ordering & Billing System

Design Documentation

Course: Advanced Software Engineering 2025

Student ID 1 (Eyad): 20230074

Student ID 2 (Moaid): 20210625

TA Name: Marwa Ahmed

Github Repo: https://github.com/MoaidHashem3/ASE-Assignment-1-Resturant-System

1. Introduction

This document details the design decisions and software architecture for the Restaurant Ordering & Billing System.

The system is designed to handle the complete workflow of a restaurant, from menu display and order customizations(add ons) to payment (Diffrent methods) and billing. This document is divided by the core design patterns implemented, explaining the problem (the requirement), the chosen solution (the pattern), and the justification for that decision.

2. Design Pattern Implementation

2.1. The Abstract Factory Pattern

Requirement: The system must support "Multiple Menu Types" such as Vegetarian, Non-Vegetarian, and Kids menus, and "encapsulate the creation of related menu families without specifying their concrete classes."

Design Decision: The **Abstract Factory** pattern was chosen as the ideal solution. This pattern provides an interface for creating *families of related objects* without specifying their concrete classes. This allows us to define a MenuFactory that can create a set of related items (e.g., a pizza *and* a burger) for a specific menu type.

Implementation:

- **MenuFactory (Interface):** The abstract factory that defines the "family" of products to be created (e.g., createPizza(), createBurger()).

- **MenuItem (Interface):** The abstract product interface that all menu items, like VeggiePizza and ClassicBeefBurger, implement.
- **Concrete Factories (e.g., VegetarianMenuFactory, NonVegetarianMenuFactory):** These classes implement the MenuFactory interface. For example, VegetarianMenuFactory's createPizza() method returns a VeggiePizza object.

SOLID Principles:

- **Open-Closed Principle (O):** The system is open for extension. We can add a new menu (e.g., GlutenFreeMenuFactory) without modifying any existing factory or client code.
- **Dependency Inversion Principle (D):** The client (which will be the Facade) depends on the MenuFactory interface, not the concrete implementations, allowing for loose coupling.

2.2. The Decorator Pattern

Requirement: The system must allow customers to "customize their meals with add-ons like extra cheese, sauces, or toppings."

Design Decision: The **Decorator** pattern was chosen to handle add-ons. This pattern allows for adding new responsibilities (like toppings and their costs) to an object dynamically at runtime. This is vastly superior to subclassing, which would lead to a "class explosion" (e.g., a PizzaWithExtraCheese class, a PizzaWithBacon class, a PizzaWithExtraCheeseAndBacon class, etc.).

Implementation:

- **MenuItem (Interface):** This is the "Component" that both our base items and our decorators will share.
- **Base Products (e.g., VeggiePizza):** The "Concrete Component" that we will wrap.
- **AddOnDecorator (Abstract Class):** This abstract class implements MenuItem and holds a reference to the MenuItem it wraps.
- **Concrete Decorators (e.g., ExtraCheese, Bacon, SpicySauce):** These classes extend AddOnDecorator. They add their own price to the wrapped item's price and append their name to the description.

SOLID Principles:

- **Open-Closed Principle (O):** The system is open to new toppings. We can add a Mushroom decorator at any time without modifying the VeggiePizza or ExtraCheese classes.
- **Single Responsibility Principle (S):** Each class has one job. VeggiePizza is responsible for being a veggie pizza. ExtraCheese is only responsible for adding the logic and price for extra cheese.

2.3. The Observer Pattern

Requirement: When an order is placed, the "kitchen and waiter systems should be acknowledged with the new order."

Design Decision: The **Observer** pattern was selected to create a clean, one-to-many dependency. This pattern allows a "Subject" (the Order) to maintain a list of "Observers" (KitchenDisplay, WaiterScreen) and notify them automatically of any state changes (like placeOrder()). This keeps the Order class loosely coupled from the notification systems.

Implementation:

- **Order (Subject):** This class maintains a List<OrderObserver>. It has addObserver() and removeObserver() methods. When its placeOrder() method is called, it triggers its private notifyObservers() method.
- **OrderObserver (Interface):** The interface all observers must implement, containing an update(Order order) method.
- **Concrete Observers (e.S.g., KitchenDisplay, WaiterScreen):** These classes implement OrderObserver and define what to do when the update notification is received.

SOLID Principles:

- **Open-Closed Principle (O):** The Order class is closed for modification. If we want to add a new notification system (e.g., a ManagerDashboard or an EmailNotifier), we simply create a new class implementing OrderObserver and add it to the order. The Order class's code does not change.
- **Dependency Inversion Principle (D):** The Order (Subject) depends on the OrderObserver *interface*, not on the concrete KitchenDisplay or WaiterScreen classes.

2.4. The Composite Pattern

Requirement: The system must display the restaurant menu in a **hierarchical structure** such as:

- Main Menu
    - Vegetarian Menu
        - v1: Veggie Pizza
        - v2: Veggie Burger
    - Kids Menu
        - k1: Kids Pizza
    - Non-Vegetarian Menu
        - n1: Beef Burger

The design must support categories containing sub-categories and items, while allowing the Facade and UI to treat everything uniformly.

Design Decision: The Composite Pattern was used because it allows treating individual items and groups of items through the same interface (`MenuComponent`).

This pattern is best for tree like structures such as a menu with nested categories.

Implementation:

MenuComponent (Abstract Class)
Defines the common interface for both `MenuLeaf` and `MenuComposite`.
Contains shared methods.

MenuLeaf (Leaf)
Represents individual menu items displayed in the UI.

MenuComposite (Composite)
Represents a menu category like("Vegetarian Menu").
It contains a list of child components (`List<MenuComponent>`), which may be leaves or other composites.

Dynamic Menu Generation (using Factories)
The Facade later automatically builds the menu using the factories defined earlier.

SOLID Principles:

Open Closed Principle :
New menus or items can be added without modifying the Composite or Facade logic.

Single Responsibility Principle:
`MenuComposite` manages only grouping of components.
`MenuLeaf` represents only a leaf item.
Neither handles ordering or pricing.

2.5.The Strategy Pattern

Requirement: The system must support different pricing behaviors:

- Multiple discount types (Combo, Pizza Discount, No Discount)
- Multiple payment types (Cash, Card, Mobile Wallet)
- Allow **easy addition of new discount or payment algorithms** without rewriting the Facade or Order classes.

Design Decision: The Strategy Pattern was chosen because it enables defining a family of algorithms for discounts and payments, and making them interchangeable at runtime.

Implementation:

1- **Discount Strategy**
 DiscountStrategy (Interface)

Concrete Strategies:

- `PizzaDiscount`
- `ComboDiscount`
- `NoDiscount`
  Each returns a `DiscountResult` object indicating the name and amount of discount applied.

2- **Payment Strategy**
 PaymentStrategy (Interface)

Concrete Strategies:

- `CashPayment`
- `CardPayment`
- `MobileWalletPayment`

  Each returns a `PaymentResult` object indicating the success status, method, and a massage.

SOLID Principles:

Open Closed Principle:
Add new discount or payment types by creating a new strategy class.

Dependency Inversion Principle :
The Facade depends on interfaces (`DiscountStrategy`, `PaymentStrategy`) instead of concrete classes.

2.6. The Facade Pattern

Requirement: The user interface must interact with the system through one single, simple API.
It must hide all sub systems complexities like:

- Menu composite structure
- Payment strategy
- Discount strategy
- Order lifecycle
- Observer notifications
- Bill generation

Design Decision: The Facade Pattern was used to provide a clean, high level interface (`RestaurantFacade`) that coordinates the entire workflow of the system. The Facade ensures the UI (Main class) does not directly depend on:

- Factories
- Composite menu tree
- Discount or payment strategies
- Bill generation

- Order observer notifications
- Data storage

Implementation: RestaurantFacade exports a small set of easy to use methods:

- `createOrder()`
- `addItemToOrder()`
- `addAddonToLastItem()`
- `checkout()`
- `showMenu()`
- `listAllOrders()`
- `registerObserver()`

but inside the class, the Facade handles:

- Dynamic menu construction using Composite
- Applying discount strategies
- Applying payment strategies
- Generating the bill using `BillGenerator`
- Publishing notifications to observers
- Saving orders in `DataStorage`
- Returning an invoice after ordering

SOLID Principles:

Single Responsibility Principle
The Facade centralizes workflow but does not calculate prices, handle discounts, or store data.

Open–Closed Principle:
Additional payment systems or discounts require no changes to the facade's code.

Dependency Inversion Principle:
The Facade depends on interfaces for payments and discounts, not implementations.