Restaurant Ordering & Billing System — Design Decisions and Rationale

1. Overview

This document explains the architecture, main components, applied design patterns, and SOLID principles for the Restaurant Ordering & Billing System. The codebase is modular, follows all 5 SOLID principles, and implements 7 design patterns for maximum extensibility and maintainability.

2. High-level Architecture

- **Facade (RestaurantFacade)**: Single entry point for the complete ordering and billing workflow (display menu → create order → notify observers → billing → payment).
- **Domain Entities**: Order, MenuItem and concrete base items (BasePizza, BaseBurger, BaseDessert, BaseBeverage).
- **Customization Layer**: AddOnDecorator and 5 concrete decorators (ExtraCheese, Mushrooms, Olives, SpicySauce, BBQSauce).
- **Menu Creation**: Abstract Factory (MenuFactory) for menu families, Factory Method (PizzaFactory, BurgerFactory) for specific items.
- **Notification System**: OrderNotifier (subject) and observers (KitchenDisplay, WaiterDisplay).
- **Billing Subsystem**: BillingService, TaxCalculator, Bill.
- **Payment Strategies**: PaymentStrategy interface with 3 implementations (CashPayment, CardPayment, MobileWalletPayment).
- **Discount System**: DiscountStrategy interface with 4 implementations (PizzaDiscount, ChickenDiscount, MeatDiscount, NoDiscount) + DiscountFactory for intelligent selection.

---

3. Design Patterns — Mapping and Purpose

3.1 Facade Pattern

Classes: `RestaurantFacade`

Why used:

- Simplifies a complex workflow (menu → order → notify kitchen → billing → payment).
- Shields the client from the complexity of multiple subsystems.
- Reduces coupling between UI/driver code and business logic.

- Ideal for a simulated console or future GUI.

3.2 Observer Pattern

Classes:

- **Subject**: `OrderNotifier`
- **Observer**: `OrderObserver`
- **Concrete Observers**: `KitchenDisplay`, `WaiterDisplay`

Why used:

- Kitchen and waiters need real-time updates when orders are placed.
- Loose coupling between Order and notification displays.
- Easy to add new observers (e.g., DeliveryDriver, Manager) without modifying Order.
- Follows Single Responsibility Principle — Order doesn't know about displays.

3.3 Decorator Pattern

Classes:

- `AddOnDecorator`
- **Concrete decorators**: `ExtraCheese`, `Mushrooms`, `Olives`, etc.

Why used:

- Allows dynamic and flexible customization of menu items (add-ons).
- Supports unlimited combinations of toppings without exploding the number of subclasses.
- Preserves Open/Closed Principle — add-ons are added via new classes, not by editing existing menu items.
- Perfect fit for "add topping" or "add ingredient" behavior.

3.4 Abstract Factory Pattern

Classes:

- `MenuFactory`
- **Concrete factories** for menu families (e.g., `ItalianMenuFactory`, `AmericanMenuFactory`)

Why used:

- Restaurants often have different families of menus (Italian, Kids Menu, Breakfast Menu).
- Ensures consistency between related items (e.g., an Italian factory creates Italian pizzas, Italian desserts…).

- Helps create "product families" without coupling client code to concrete classes.

3.5 Factory Method Pattern

Classes:

- `PizzaFactory`
- `BurgerFactory`

Why used:

- Allows subclasses to decide which specific pizza or burger object to create.
- Avoids having large switch statements or if-else chains.
- Makes it easy to add new pizza/burger types while keeping creator logic isolated.

3.6 Strategy Pattern

Used twice (2 separate Strategy implementations):

A. DiscountStrategy

Classes:

- `DiscountStrategy` (interface)
- `PizzaDiscount`, `ChickenDiscount`, `MeatDiscount`, `NoDiscount` (concrete strategies)
- `DiscountFactory` (factory for automatic selection)

Why used:

- Different discount rules apply to different food categories.
- Allows changing the discount algorithm without changing billing logic.
- DiscountFactory encapsulates selection logic (follows Open/Closed Principle).
- Priority-based: Pizza > Chicken > Meat > None
- Demonstrates proper separation of concerns and DIP.

B. PaymentStrategy

Classes:

- `PaymentStrategy` (interface)
- `CashPayment`, `CardPayment`, `MobileWalletPayment` (concrete strategies)

Why used:

- Payment behavior varies widely depending on method.

- Keeps payment logic interchangeable and loosely coupled.
- Extensible for future payments (PayPal, crypto, reward points, etc.).
- Easy to add new payment methods without modifying existing code.

---

4. SOLID Principles & Justification

Single Responsibility Principle (SRP)

Each class has a single, well-defined responsibility:

- `TaxCalculator` only computes tax
- `BillingService` handles billing workflow
- `DiscountFactory` only selects discount strategies
- `Order` only manages order data
- `RestaurantFacade` only coordinates subsystems

Open/Closed Principle (OCP)

The system is open for extension, closed for modification:

- **New menu families**: implement `MenuFactory` interface
- **New discount rules**: implement `DiscountStrategy` interface (no Facade changes needed)
- **New payment methods**: implement `PaymentStrategy` interface
- **New add-ons**: create new decorator classes
- `DiscountFactory` encapsulates selection logic (critical for OCP compliance)

Liskov Substitution Principle (LSP)

Subtypes can replace their base types:

- All `MenuItem` implementations (`BasePizza`, `BaseBurger`, etc.) are fully interchangeable
- All decorators can wrap any `MenuItem`
- All `MenuFactory` implementations produce compatible `MenuItem` objects
- All `DiscountStrategy` implementations work with any `Order`

Interface Segregation Principle (ISP)

Clients are not forced to depend on unused methods:

- `MenuItem` interface is minimal (`getDescription`, `getPrice`, `getCategory`)
- `OrderObserver` has single `update()` method
- `PaymentStrategy` and `DiscountStrategy` are focused and small

Dependency Inversion Principle (DIP)

High-level modules depend on abstractions:

- `RestaurantFacade` depends on `MenuFactory` (not concrete factories)
- `OrderNotifier` depends on `OrderObserver` (not concrete observers)
- `BillingService` depends on `PaymentStrategy` and `DiscountStrategy` (not concrete implementations)
- `DiscountFactory` returns `DiscountStrategy` abstraction (not hardcoded in Facade)

---

5. Extension Points (How to add new features)

- **New menu family**: Create a class implementing `MenuFactory` interface.
- **New pizza/burger type**: Extend `PizzaFactory` or `BurgerFactory`.
- **New add-on**: Create a class extending `AddOnDecorator`.
- **New discount rule**: Implement `DiscountStrategy`, optionally update `DiscountFactory`.
- **New payment method**: Implement `PaymentStrategy` interface.

---

7. Notes and Caveats

- `Order.quantities` uses `Map` with object identity; consider implementing equals/hashCode on base items for logical equality if needed.
- Console-based simulation; in production, separate UI from business logic.
- Payment simulation: payments are mocked (no real payment gateway integration).
- Tax rate (14%) and discount values are configurable via constructor parameters.

---

8. Testing & Verification

All 7 design patterns have been tested and verified:

- ✅ **Facade**: Single entry point working correctly
- ✅ **Abstract Factory**: 3 menu families (Vegetarian, Non-Vegetarian, Kids) tested
- ✅ **Factory Method**: Italian/Eastern pizzas and burgers tested
- ✅ **Decorator**: Multiple add-ons successfully stacked
- ✅ **Observer**: Kitchen and Waiter notifications working
- ✅ **Strategy (Payment)**: Cash, Card, Mobile Wallet tested
- ✅ **Strategy (Discount)**: Pizza (10%), Meat ($7), No Discount tested and working correctly

---