Final Design Responsibilities

Flow overview: CheckoutService (orchestrates)

Coordinates the entire checkout flow and orchestrates other components by passing in dependencies as parameters.

ProductFactory (builds)

Constructs Product objects and chains decorators based on the recipe input.

PricingService (calculates)

Computes price breakdowns using injected DiscountPolicy and TaxPolicy.

DiscountPolicy & TaxPolicy (business rules)

Encapsulate discount and tax logic; selected and injected as needed.

ReceiptPrinter (formats)

Formats detailed, readable receipts from product and pricing info.

PaymentStrategy (handles payment I/O)

Manages payment actions (cash, card, wallet) via injected strategy.

All dependencies are injected (constructor injection); no globals. Each component can be unit tested separately.

Smells Removed

God class:

Broke up OrderManagerGod by separating business logic into dedicated services/classes (PricingService, ProductFactory, etc.).

Feature envy/multiple responsibilities:

Business logic and formatting isolated (no mixing of printing and calculations).

Global dependencies:

Replaced static fields/globals with full dependency injection; each object only uses what it needs.

Hidden side effects:

Payment I/O isolated via PaymentStrategy.

Refactorings Applied

Extract Class:

Moved pricing, discount, and tax calculation to PricingService from the god class.

Extract Method:

Broke large methods into composable, focused ones (e.g., product breakdown, receipt formatting).

Replace Conditional with Polymorphism:

Used DiscountPolicy and PaymentStrategy interfaces, with concrete types (CouponDiscount, LoyaltyDiscount, etc.).

Dependency Injection:

All collaborators are passed via constructor, reducing coupling and improving testability.

Move Responsibility:

Moved recipe parsing to ProductFactory; moved receipt formatting away from business logic.

SOLID Principles Satisfied

S (Single Responsibility):

Each class handles one logical concern (pricing, printing, payment, product construction).

O (Open-Closed):

Discount and payment logic can be extended without modifying existing classes—just add a new concrete policy/strategy.

L (Liskov Substitution):

All DiscountPolicy implementations can be swapped freely for each other.

I (Interface Segregation):

Payment and discount logic use narrow interfaces with only methods needed for their concern.

D (Dependency Inversion):

High-level flow (CheckoutService or main) depends on abstractions (interfaces), not concrete implementations.

How to Add a New Discount Type (No Editing Required)

Create a new class implementing the DiscountPolicy interface (e.g., BirthdayDiscount implements DiscountPolicy).

Register it (if using a factory or enum, add mapping; if using dependency injection, inject directly).

No changes to existing CheckoutService, PricingService, or other domain classes; simply provide the new policy at runtime.

Responsibility Diagram

CheckoutService
        |
        v
ProductFactory --> Product [+Decorators]
        |
        v
PricingService (uses DiscountPolicy + TaxPolicy)
        |
        v
ReceiptPrinter
        |
        v
PaymentStrategy