Code maintainability Issues – order.java

- **Tight Coupling:** The Order class directly interacts with multiple components (PaymentProcessor, NotificationService, and DiscountCalculator), creating strong dependencies. This makes it harder to modify, extend, or replace these components without changing the Order class.
- **Lack of Extensibility in Payment Processing:** The processPayment() method calls the PaymentProcessor class, which uses if-else statements to handle different payment methods. Adding new payment methods requires modifying the PaymentProcessor class, violating the Open/Closed Principle.
- **Hardcoded Notification Logic:** The sendNotifications() method directly invokes specific notification methods (sendEmailNotification(), sendSMSNotification()), making it difficult to introduce new notification methods without modifying the Order class.
- **No Separation of Discount Calculation Logic:** While the calculateTotal() method offloads discount logic to the DiscountCalculator, it lacks flexibility since the discount conditions (e.g., isHoliday, isMember) must be handled within DiscountCalculator. This setup results in static and limited discount application.

Code maintainability Issues – product.java

- **Scattered Instantiation:** Direct use of constructors (e.g., new Book(), new Electronics()) is scattered throughout the codebase. This creates redundancy and makes adding new product types more difficult.

- **Missing Centralized Creation:** The code lacks a Factory Method or Factory Class that centralizes the creation of product objects. This leads to duplicated logic for creating instances of different products.

Code maintainability Issues – paymentprocessor.java

- **Conditional Logic:** The process() method uses if-else statements to handle different payment methods (CreditCard, PayPal, Bitcoin). This logic is not scalable; every new payment method would require modifying the existing PaymentProcessor class, which violates the Open/Closed Principle.

- **No Clear Abstraction:** Payment processing strategies are not abstracted, resulting in a rigid structure that cannot easily adapt to changes in payment processing requirements.

Code maintainability Issues – notificationservice.java

- **Hardcoded Notification Methods:** The notification logic (e.g., sendEmailNotification(), sendSMSNotification(), sendPushNotification()) is implemented directly in this class. The Order class tightly couples with NotificationService, calling these methods directly.

- **Lack of Flexibility:** There is no way to dynamically add or remove notification types without modifying the NotificationService class. This rigid structure hampers extensibility.

Code maintainability Issues – DiscountCalculator.java

- **Hardcoded Discount Logic and Lack of Flexibility for Discount Types:** The DiscountCalculator class has hardcoded logic for applying discounts. This makes it difficult to change or extend the discount rules (e.g., different holiday discounts, percentage-based discounts, or special offers). If you want to add a new type of discount (e.g., a "Seasonal" discount), you would need to modify the applyDiscounts method directly, violating the Open/Closed Principle.

- **Tight Coupling of Discount Logic:** The discount calculation is entirely encapsulated within the DiscountCalculator class, making it impossible to modify or extend the discount logic without altering this class. This coupling prevents you from reusing the discount logic in other parts of the code or applying different discount strategies to different product types or categories.

Code maintainability Issues – Violations

- **Violation of SOLID Principles:** The current code violates key SOLID principles:
    - **Single Responsibility Principle (SRP):** Classes like Order have multiple responsibilities, such as managing products, handling payments, calculating totals, and sending notifications.
    - **Open/Closed Principle (OCP):** Several classes (PaymentProcessor, DiscountCalculator, NotificationService) are not open for extension without modification. This results in a brittle codebase that is difficult to modify or extend without introducing bugs.
    - **Tight Coupling:** The direct interaction between classes (Order with PaymentProcessor, NotificationService, and DiscountCalculator) creates a tightly coupled codebase. This makes changes or extensions to any part of the system more challenging.

- **Lack of Design Patterns:** The code misses opportunities to use key design patterns that could make it more modular, flexible, and maintainable:
    - **Strategy Pattern** for payment processing.
    - **Observer Pattern** for notifications.
    - **Factory Method Pattern** for product creation.
    - **Decorator Pattern** for discount calculations.