Q1: One-by-One — How SOLID Principles are Achieved

1. Single Responsibility Principle (SRP)

- **Old Code Problem**: OrderProcessor was doing *everything* — fetching orders, deciding payment method, processing payment, and logging.
- **New Code Fix**:
    - OrderRepository → only handles **data fetching**.
    - PaymentStrategyFactory → only **decides which payment strategy** to use.
    - IPaymentStrategy implementations (CreditCardPayment, PayPalPayment) → only **process payments**.
    - ConsoleLogger → only **logs messages**.
    - OrderProcessor → only **coordinates** the order processing.

---

2. Open/Closed Principle (OCP)

- **Old Code Problem**: Adding a new payment type meant **modifying** OrderProcessor (if-else chain).
- **New Code Fix**:
    - Now, you just **add a new class** implementing IPaymentStrategy (e.g., ApplePayPayment) and extend the PaymentStrategyFactory without touching OrderProcessor.

---

3. Liskov Substitution Principle (LSP)

- **New Code**:
    - Anywhere you use IPaymentStrategy, you can replace it with any payment type (CreditCardPayment, PayPalPayment) and the program will still work.
    - Same for ILoggerService and IOrderRepository — any class implementing them can be substituted.

---

4. Interface Segregation Principle (ISP)

- **New Code**:
    - Instead of one giant interface for everything, small, focused interfaces are made:
        - IPaymentStrategy (only for payment)
        - IOrderRepository (only for order fetching)

- ILoggerService (only for logging)

---

5. Dependency Inversion Principle (DIP)

- **Old Code Problem**: OrderProcessor directly depended on concrete classes (Database, hardcoded payment logic).
- **New Code Fix**:
  - OrderProcessor depends on abstractions (IOrderRepository, IPaymentStrategy, ILoggerService) instead of concrete classes.
  - This makes it easy to change implementations without changing OrderProcessor.

Q2: Patterns Implemented & Why

1. Strategy Pattern

- **Where**: IPaymentStrategy and its implementations (CreditCardPayment, PayPalPayment).
- **Why**: To choose payment processing behavior at runtime without if-else chains.

---

2. Factory Pattern

- **Where**: PaymentStrategyFactory.
- **Why**: To centralize object creation for payment strategies instead of creating them inside OrderProcessor.

---

3. Repository Pattern

- **Where**: IOrderRepository and OrderRepository.
- **Why**: To abstract database/data source access so that OrderProcessor doesn't know where data comes from.

---

4. Dependency Injection (DI) Principle *(not exactly a GoF pattern, but important)*

- **Where**: OrderProcessor receives dependencies (IOrderRepository, PaymentStrategyFactory, ILoggerService) via constructor.
- **Why**: Makes the code testable and loosely coupled.

✅ In short:

- **Old Code** = tightly coupled, hard to extend, breaks SRP & OCP.
- **New Code** = modular, extendable, uses **Strategy + Factory + Repository**, and follows **all SOLID principles**.