**. Single Responsibility Principle**

You have a UserService class that:

* Validates user input
* Saves data to the database
* Sends a welcome email

What is the issue?

A. Too many methods  
B. Violates SRP  
C. Needs inheritance  
D. Should be static

**✅ Answer: B**  
**Explanation:** One class is handling 3 different responsibilities. SRP says one class = one reason to change.

**2. Refactoring SRP**

To follow SRP, what’s the **best solution** for the above?

A. Create UserValidator, UserRepository, EmailService  
B. Add more methods to UserService  
C. Use static methods  
D. Use global variables

**✅ Answer: A**  
**Explanation:** Splitting by responsibility is the correct way.

**3. Open/Closed Principle**

There is a PaymentProcessor class with a switch(paymentType) to handle CreditCard, UPI, PayPal. Adding a new payment type requires editing the class.

What principle is violated?

A. LSP  
B. SRP  
C. OCP  
D. ISP

**✅ Answer: C**  
**Explanation:** Class should be open for extension, closed for modification. Switch-case forces modification.

**4. How to fix the above (OCP)**

Best approach?

A. Add more switch cases  
B. Use inheritance & polymorphism (Payment interface)  
C. Make method static  
D. Remove new payment type

**✅ Answer: B**  
**Explanation:** Polymorphism allows extension without modifying existing code.

**5. Liskov Substitution Principle**

class Bird { fly(); }  
class Penguin extends Bird { fly() throws Exception; }  
What’s wrong?

A. No constructor  
B. Violates LSP  
C. Performance issue  
D. Works fine

**✅ Answer: B**  
**Explanation:** Subclass changes expected behavior. Penguin cannot fly, so it shouldn’t extend Bird in that way.

**6. ISP (Interface Segregation Principle)**

Interface IMachine has print(), scan(), fax(). BasicPrinter only needs print().

What principle is violated?

A. DIP  
B. ISP  
C. SRP  
D. OCP

**✅ Answer: B**  
**Explanation:** Classes shouldn’t be forced to implement methods they don’t use.

**7. Fix for ISP**

Best solution?

A. Split interface into IPrinter, IScanner, IFax  
B. Add default methods  
C. Use abstract class  
D. Ignore unused methods

**✅ Answer: A**  
**Explanation:** Small, role-based interfaces are ideal.

**8. Dependency Inversion Principle**

High-level module directly creates new MySQLDatabase() instance inside service class.

Issue?

A. Good practice  
B. Violates DIP  
C. Violates ISP  
D. Faster performance

**✅ Answer: B**  
**Explanation:** High-level modules should depend on abstraction, not concrete classes.

**9. DIP Application**

How to fix the above?

A. Use Factory pattern or dependency injection  
B. Add more methods  
C. Make class static  
D. Use global variable

**✅ Answer: A**  
**Explanation:** Either inject via constructor or use factory. High-level depends on interface.

**10. SRP + OCP Mixed Scenario**

A ReportGenerator class generates reports in PDF. Now you need Excel format. What’s best?

A. Modify same class  
B. Create new class that extends base ReportGenerator  
C. Add big if-else  
D. Merge both formats in same method

**✅ Answer: B**  
**Explanation:** OCP = extend without modifying current code. SRP = each class handles one job.

**LSP Violation in Shapes**

Square extends Rectangle. Setting width auto-updates height. Some code using Rectangle breaks.  
Why?

A. Rectangle is abstract  
B. LSP is violated  
C. Needs static methods  
D. Too many setters

**✅ Answer: B**  
**Explanation:** Subclass should not break expectations of parent behavior.

**12. Better design for Square/Rectangle?**

A. Square should not extend Rectangle  
B. Make everything final  
C. Use global variables  
D. Remove Rectangle class

**✅ Answer: A**  
**Explanation:** Use a common interface instead of inheritance if behavior differs.

**13. ISP Violation**

An interface Vehicle has drive(), fly(), sail(). Car implements all but only uses drive.

Problem?

A. Too many classes  
B. Violates SRP  
C. Violates ISP  
D. Breaks OCP

**✅ Answer: C**  
**Explanation:** Car shouldn't implement unnecessary methods.

**14. Fix Vehicle design (ISP)**

A. Split into Driveable, Flyable, Sailable  
B. Use abstract class  
C. Add default empty methods  
D. Remove Vehicle interface

**✅ Answer: A**  
**Explanation:** Segregate interfaces based on capability.

**15. DIP Violation**

High-level service class creates new EmailNotification() directly.  
Why is this bad?

A. Memory leak  
B. Tight coupling to low-level module  
C. Too many constructors  
D. Violates SRP

**✅ Answer: B**  
**Explanation:** High-level should depend on abstraction, not concrete class.

**16. Applying DIP properly**

A. Use Notification interface and inject implementation  
B. Use static method  
C. Use Singleton  
D. Use enum

**✅ Answer: A**  
**Explanation:** Depending on interface reduces coupling.

**17. OCP in tax calculation**

Every time tax rules change, developers modify old TaxCalculator code. What’s wrong?

A. Duplicate code  
B. Violates OCP  
C. Violates ISP  
D. Too many classes

**✅ Answer: B**  
**Explanation:** You shouldn’t modify existing class for every change.

**18. Best design for tax rules (OCP)**

A. Hardcode logic  
B. Add if-else  
C. Create separate classes implementing TaxStrategy  
D. Combine everything in one method

**✅ Answer: C**  
**Explanation:** Strategy pattern + OCP = extend behavior without changes.

**19. SRP in Logging**

A class handles:

* Business logic
* Logging to file
* Formatting logs

Best SRP approach?

A. Keep it in one place  
B. Separate logging into dedicated utility/service  
C. Inline log formatting  
D. Use static logger in same class

**✅ Answer: B**  
**Explanation:** Logging is a separate responsibility.

**20. Mixing SOLID**

A UserManager does:

* Create user
* Validate input
* Send welcome email
* Save to database  
  What principles are violated? (Choose best answer)

A. Only SRP  
B. SRP + OCP  
C. SRP + DIP  
D. SRP + OCP + DIP

**✅ Answer: D**  
**Explanation:**

* SRP: Too many responsibilities.
* OCP: Hard to extend behavior without modifying.
* DIP: Directly creating dependencies like email/db.

### ****21. Singleton Scenario****

You need only one instance of Logger across the application. Which pattern?

A. Factory  
B. Singleton  
C. Builder  
D. Prototype

**✅ Answer: B**  
**Explanation:** Singleton ensures a single global instance.

### ****22. Singleton Problem****

Why is a **lazy-loaded Singleton without synchronization** dangerous in Java?

A. Slow performance  
B. Memory leak  
C. Thread-safety issues (multiple instances)  
D. Cannot be extended

**✅ Answer: C**  
**Explanation:** Multiple threads may create multiple instances.

### ****23. Factory Pattern Use Case****

You need to create different types of Notification objects (Email, SMS, Push) based on user input. Which pattern?

A. Adapter  
B. Factory  
C. Observer  
D. Mediator

**✅ Answer: B**  
**Explanation:** Factory centralizes object creation.

### ****24. Factory Advantage****

Why use Factory instead of new keyword everywhere?

A. Slower code  
B. Better encapsulation & easier extension  
C. Reduces memory  
D. Removes inheritance

**✅ Answer: B**  
**Explanation:** Factory hides creation logic and supports OCP.

### ****25. Builder Pattern Scenario****

You have a complex Pizza object with optional toppings. Best pattern?

A. Singleton  
B. Builder  
C. Prototype  
D. Strategy

**✅ Answer: B**  
**Explanation:** Builder constructs complex objects step-by-step.

### ****26. Prototype Pattern****

When is Prototype pattern useful?

A. When objects are cheap to create  
B. When objects are expensive to create and you want to clone  
C. To enforce one instance  
D. To build step-by-step

**✅ Answer: B**  
**Explanation:** Prototype clones existing objects for performance.

## 🧱 STRUCTURAL PATTERNS

### ****27. Adapter Pattern****

You have a new payment gateway API that doesn't match existing interface. What to do?

A. Modify all client code  
B. Use Adapter pattern  
C. Use Singleton  
D. Rewrite API

**✅ Answer: B**  
**Explanation:** Adapter converts one interface to another.

### ****28. Decorator Pattern Scenario****

You have to add features to a Coffee object (e.g., Milk, Sugar, Whip) dynamically at runtime.

Best pattern?

A. Strategy  
B. Decorator  
C. Observer  
D. Factory

**✅ Answer: B**  
**Explanation:** Decorator adds behavior without changing original class.

### ****29. Problem if you DON'T use Decorator****

If we use inheritance for every coffee combination (CoffeeWithMilkAndSugarAndWhip…), issue?

A. Follows SOLID  
B. Too many subclasses (class explosion)  
C. Faster performance  
D. Better abstraction

**✅ Answer: B**  
**Explanation:** Inheritance leads to exponential class growth.

### ****30. Facade Pattern Scenario****

Your system has a very complex library with many steps to process an order. You create a simple OrderService that internally calls all steps.

This is:

A. Proxy  
B. Adapter  
C. Facade  
D. Visitor

**✅ Answer: C**  
**Explanation:** Facade provides a simplified interface to complex subsystems.

## BEHAVIORAL PATTERNS

### ****31. Strategy Pattern Scenario****

You have different discount algorithms (Festival, Loyalty, Bulk). You want to choose at runtime.

Best pattern?

A. Factory  
B. Strategy  
C. Observer  
D. Decorator

**✅ Answer: B**  
**Explanation:** Strategy lets you swap algorithms at runtime.

### ****32. Without Strategy Pattern****

if-else or switch used everywhere for discount logic. Problem?

A. Violates LSP  
B. Violates OCP  
C. Violates ISP  
D. Violates DIP

**✅ Answer: B**  
**Explanation:** To add a new discount, you must modify existing code (OCP broken).

### ****33. Observer Pattern Scenario****

You have a Stock class. When price changes, all registered users should auto get notifications.

Which pattern?

A. Observer  
B. Mediator  
C. Chain of Responsibility  
D. Visitor

**✅ Answer: A**  
**Explanation:** Observer notifies dependents on state change.

### ****34. Command Pattern Use Case****

You need to implement UNDO/REDO for text editor actions.

Best pattern?

A. Strategy  
B. Command  
C. Adapter  
D. Prototype

**✅ Answer: B**  
**Explanation:** Command encapsulates actions, supports undo/redo.

### ****35. Chain of Responsibility Scenario****

Form validation: first check empty, then format, then database. Each should pass to next.

Which pattern?

A. Chain of Responsibility  
B. Observer  
C. Builder  
D. Proxy

**✅ Answer: A**  
**Explanation:** Each handler passes request to the next.

### ****36. Template Method Pattern****

Three types of reports: only formatting differs. Steps: collect → process → format.

Best pattern?

A. Strategy  
B. Template Method  
C. Factory  
D. Facade

**✅ Answer: B**  
**Explanation:** Template defines skeleton, subclasses override one step.

### ****37. Mediator Pattern Scenario****

In a chat room, each user should not talk to each user directly. A central chat server manages all communication.

Which pattern?

A. Observer  
B. Mediator  
C. Strategy  
D. Adapter

**✅ Answer: B**  
**Explanation:** Mediator centralizes complex communication.

### ****38. SOLID + Patterns Combo****

You replace switch-case with separate classes for each behavior + use interface.

Which principle + pattern?

A. SRP + Builder  
B. OCP + Strategy  
C. DIP + Singleton  
D. ISP + Adapter

**✅ Answer: B**  
**Explanation:** New classes = extend without modify (OCP) and Strategy picks behavior.

### ****39. Why use Dependency Injection with Strategy?****

A. Reduces performance  
B. Tightly couples client to implementation  
C. Allows injecting different strategies easily  
D. Violates SOLID

**✅ Answer: C**  
**Explanation:** DI + Strategy = high flexibility, low coupling (DIP followed).

### ****40. Master Scenario (Real-World)****

You build a payment module:

* Must support new payment types (extension)
* Each payment has its own algorithm
* Client should not know concrete classes
* High-level code should depend on abstractions

Which combo of SOLID principles + pattern solves this BEST?

A. SRP + Adapter  
B. OCP + Strategy + DIP  
C. LSP + Singleton  
D. ISP + Prototype

**✅ Answer: B**  
**Explanation:**

* OCP → Add new payment without modifying old code
* Strategy → Each payment = separate algorithm
* DIP → Client depends on interface, not class