**1. S – Single Responsibility Principle (SRP)**

**Definition:** A class should have only one reason to change, meaning it should have only one job or responsibility.

// Responsible only for user data

public class User {

private String name;

private String email;

// Getters & Setters

}

// Responsible only for user database operations

public class UserRepository {

public void save(User user) {

// Save user to database

}

}

public class User {

private String name;

private String email;

public void save() {

// Saving logic here violates SRP

}

}

A chef should cook. A waiter should serve. Don’t ask the chef to take orders and serve food.

**2. O – Open/Closed Principle (OCP)**

**Definition:** Software entities (classes, modules, functions) should be **open for extension but closed for modification**.

Example:

public interface Notification {

void send(String message);

}

public class EmailNotification implements Notification {

public void send(String message) {

// send email

}

}

public class SMSNotification implements Notification {

public void send(String message) {

// send SMS

}

}

public class NotificationService {

public void notifyUser(Notification notification, String message) {

notification.send(message);

}

}

Against O

public class NotificationService {

public void send(String type, String message) {

if (type.equals("email")) {

// send email

} else if (type.equals("sms")) {

// send SMS

}

// Needs modification for every new type

}

}

**3. L – Liskov Substitution Principle (LSP)**

**Definition:** Subtypes must be substitutable for their base types without altering the correctness of the program.

class Bird {

public void fly() {

// General flying logic

}

}

class Sparrow extends Bird {

public void fly() {

// Sparrow flies

}

}

Against L

class Bird {

public void fly() {}

}

class Ostrich extends Bird {

public void fly() {

throw new UnsupportedOperationException("Ostriches can't fly");

}

}

**4. I – Interface Segregation Principle (ISP)**

**Definition:** Clients should not be forced to depend on interfaces they do not use.

public interface Printer {

void print();

}

public interface Scanner {

void scan();

}

public class MultiFunctionPrinter implements Printer, Scanner {

public void print() {}

public void scan() {}

}

public interface Machine {

void print();

void scan();

}

Against I

public class OldPrinter implements Machine {

public void print() {}

public void scan() {

throw new UnsupportedOperationException();

}

}

**5. D – Dependency Inversion Principle (DIP)**

**Definition:** High-level modules should not depend on low-level modules. Both should depend on abstractions (interfaces).

public interface Keyboard {

void input();

}

public class WiredKeyboard implements Keyboard {

public void input() {

// Wired input

}

}

public class Computer {

private Keyboard keyboard;

public Computer(Keyboard keyboard) {

this.keyboard = keyboard;

}

public void type() {

keyboard.input();

}

}

You can easily switch from WiredKeyboard to WirelessKeyboard by injecting the dependency.

Against D

// Low-level module

public class WiredKeyboard {

public void input() {

System.out.println("Input from wired keyboard");

}

}

// High-level module

public class Computer {

private WiredKeyboard keyboard;

public Computer() {

this.keyboard = new WiredKeyboard(); // tightly coupled

}

public void type() {

keyboard.input();

}

}

Computer class (high-level) directly depends on WiredKeyboard (low-level).

If you want to change to WirelessKeyboard, you must **modify** the Computer class → violates **Open/Closed Principle** too.

No **abstraction/interface** layer in between.