Single Responsibility Principle (SRP)

1. Class should have only one reason to change.
2. Without SRP: A class that mixes multiple concerns becomes large and harder to comprehend.
3. With SRP: Smaller, focused classes are easier to read, understand, and explain, especially for new developers joining the project.
4. Instead of writing saveToFile in Employee class make it different utility class.
5. Reduces Coupling : By separating responsibilities, you reduce the dependencies, making the system more modular and flexible.
6. Supports Agile Development: change in one responsibility affects only the corresponding class

```java
class Employee {
    private String name;
    private String department;

    public Employee(String name, String department) {
        this.name = name;
        this.department = department;
    }

    public String getName() {
        return name;
    }

    public String getDepartment() {
        return department;
    }
}

// A separate class for saving employee data to a file
class EmployeeFileManager {
    public void saveToFile(Employee employee, String filename) {
        try (FileWriter writer = new FileWriter(filename)) {
            writer.write("Name: " + employee.getName() + ", Department: " +
employee.getDepartment());
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Open/Closed Principle (OCP)

1. Open For Extension But Closed For Modification.

2. Encourages Reusability: Extending functionality without altering existing code increases modularity and reusability.
3. To Implement OCP:
   - Use abstraction (interfaces and abstract classes)
   - Use Polymorphism (instead of conditional statement use method overriding)
4.

```
// Step 1: Define an interface for Discount Policy
interface DiscountPolicy {
    double calculateDiscount(double amount);
}

// Step 2: Implement specific policies
class RegularCustomerDiscount implements DiscountPolicy {
    @Override
    public double calculateDiscount(double amount) {
        return amount * 0.1; // 10% discount for regular customers
    }
}

class PremiumCustomerDiscount implements DiscountPolicy {
    @Override
    public double calculateDiscount(double amount) {
        return amount * 0.2; // 20% discount for premium customers
    }
}

class GoldCustomerDiscount implements DiscountPolicy {
    @Override
    public double calculateDiscount(double amount) {
        return amount * 0.3; // 30% discount for gold customers
    }
}

// Step 3: Use a calculator class that delegates the calculation
class DiscountCalculator {
    public double calculateDiscount(DiscountPolicy discountPolicy, double amount)
{
        return discountPolicy.calculateDiscount(amount);
    }
}

// Step 4: Usage
public class Main {
    public static void main(String[] args) {
        DiscountCalculator calculator = new DiscountCalculator();

        // Calculate discounts for different customer types
        double regularDiscount = calculator.calculateDiscount(new
RegularCustomerDiscount(), 1000);
        double premiumDiscount = calculator.calculateDiscount(new
PremiumCustomerDiscount(), 1000);
```

```
        double goldDiscount = calculator.calculateDiscount(new
GoldCustomerDiscount(), 1000);

        System.out.println("Regular Customer Discount: " + regularDiscount);
        System.out.println("Premium Customer Discount: " + premiumDiscount);
        System.out.println("Gold Customer Discount: " + goldDiscount);
    }
}
```

Liskov Substitution Principle (LSP)

- **Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.**
- The subclass must ensure that it does not introduce constraints or behavior that violate the expectations set by the interface or base class. This is central to the Liskov Substitution Principle (LSP).

```
class PushNotificationSender implements NotificationSender {
    @Override
    public void sendNotification(String message, String recipient) {
        // Additional constraint: Check if the recipient has an active internet
connection
        if (!isInternetAvailable()) {
            throw new IllegalStateException("Internet connection is required for
push notifications.");
        }
        System.out.println("Sending push notification to " + recipient + ": " +
message);
    }

    private boolean isInternetAvailable() {
        // Simulate internet check
        return false; // Internet is not available
    }
}
```

- above code is breaking contract of implemented interface instead seperate constraint do not add it to overrided method

```
    // Validation or Pre-check in the client code
    public class NotificationService {
        public void sendNotification(NotificationSender sender, String message,
String recipient) {
            if (sender instanceof PushNotificationSender &&
!isInternetAvailable()) {
                throw new IllegalStateException("Internet connection is required
for push notifications.");
            }
            sender.sendNotification(message, recipient);
        }

        private boolean isInternetAvailable() {
            return true; // Simulate internet availability
        }
    }
```