

SOLID Principles with Examples

Can you explain SOLID Principles with examples in Java? The example should have a piece of code violating each principle and a solution that refactors the code to comply with the principle. Also, for each principle, I would like to have a list of possible question and answers to prepare for interview.

Sure! Let's go over the **SOLID principles** one by one, explaining them with examples of code that violates each principle, followed by a solution that refactors the code to comply with the principle. At the end of each principle, I'll provide potential interview questions and answers to help you prepare.

1. Single Responsibility Principle (SRP)

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

Violating Example:

```
java

class Employee {
    private String name;
    private String address;

    public void saveToDatabase() {
        // logic to save employee to database
    }

    public void generateReport() {
        // logic to generate report
    }
}
```

Issue: The `Employee` class has two responsibilities—storing employee data and generating reports. If the requirements change for either, the class needs to be modified, violating SRP.

Refactored Example:

```
java

class Employee {
    private String name;
    private String address;

    // Employee data management logic
}

class EmployeeReport {
    public void generateReport(Employee employee) {
        // logic to generate report
    }
}

class EmployeeDatabase {
    public void saveToDatabase(Employee employee) {
        // logic to save employee to database
    }
}
```

Explanation: Now, `Employee` only manages employee data, `EmployeeReport` is responsible for generating reports, and `EmployeeDatabase` is responsible for saving data. Each class has a single responsibility.

Interview Questions:

- What is the Single Responsibility Principle?
- How can a class violate the Single Responsibility Principle?
- How would you refactor a class to comply with SRP?

Sample Answers:

- **SRP states that a class should only have one reason to change.** This means that if the business logic or storage format changes, only one part of the system should be affected.
- A class violates SRP when it has more than one job, such as combining data storage and report generation into a single class.

- To refactor, you would split the class into smaller classes, each handling one responsibility, like separating data management, reporting, and database operations.
-

2. Open/Closed Principle (OCP)

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

Violating Example:

java

```
class Shape {
    public double area() {
        // logic for area calculation
        return 0;
    }
}

class AreaCalculator {
    public double calculateArea(Shape shape) {
        if (shape instanceof Circle) {
            // calculate area for Circle
        } else if (shape instanceof Rectangle) {
            // calculate area for Rectangle
        }
        return 0;
    }
}

class Circle extends Shape {
    private double radius;

    public double area() {
        return Math.PI * radius * radius;
    }
}

class Rectangle extends Shape {
    private double width, height;
```

```
    public double area() {  
        return width * height;  
    }  
}
```

Issue: The `AreaCalculator` class is tightly coupled to the `Circle` and `Rectangle` classes. Every time a new shape type is added, the `AreaCalculator` needs to be modified.

Refactored Example:

java

```
interface Shape {  
    double area();  
}  
  
class Circle implements Shape {  
    private double radius;  
  
    public double area() {  
        return Math.PI * radius * radius;  
    }  
}  
  
class Rectangle implements Shape {  
    private double width, height;  
  
    public double area() {  
        return width * height;  
    }  
}  
  
class AreaCalculator {  
    public double calculateArea(Shape shape) {  
        return shape.area();  
    }  
}
```

Explanation: The `AreaCalculator` is now closed for modification. New shapes can be added by simply creating new classes that implement the `Shape` interface, which follows the Open/Closed Principle.

Interview Questions:

- What is the Open/Closed Principle?
- How would you refactor code that violates the Open/Closed Principle?
- Can you explain how polymorphism supports the Open/Closed Principle?

Sample Answers:

- **OCP means that you should be able to add new functionality without changing existing code.** The goal is to extend the behavior without altering the existing codebase.
 - To refactor code violating OCP, you would introduce interfaces or abstract classes that allow the new functionality to be added without changing the existing code. In this example, the `Shape` interface allows adding new shapes easily.
 - **Polymorphism supports OCP** by allowing objects of different types to be treated uniformly. This way, the `AreaCalculator` doesn't need to know about specific shapes and can just work with any `Shape`.
-

3. Liskov Substitution Principle (LSP)

Definition: Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

Violating Example:

```
java

class Bird {
    public void fly() {
        System.out.println("Flying");
    }
}

class Ostrich extends Bird {
    public void fly() {
        throw new UnsupportedOperationException("Ostriches cannot fly");
    }
}
```

Issue: The `Ostrich` class is a subclass of `Bird`, but it cannot fly. Replacing a `Bird` object with an `Ostrich` object breaks the behavior, violating LSP.

Refactored Example:

```
java

abstract class Bird {
    public abstract void move();
}

class Sparrow extends Bird {
    public void move() {
        System.out.println("Flying");
    }
}

class Ostrich extends Bird {
    public void move() {
        System.out.println("Running");
    }
}
```

Explanation: Now, both `Sparrow` and `Ostrich` override the `move` method, but they have different implementations. This follows LSP because both subclasses can be used interchangeably without breaking expected behavior.

Interview Questions:

- What is the Liskov Substitution Principle?
- How would you refactor a class that violates LSP?
- Can you give an example of how violating LSP could lead to problems?

Sample Answers:

- LSP states that objects of a superclass should be replaceable by objects of a subclass without altering the correctness of the program. Subclasses should extend the base class without changing its expected behavior.
- To refactor, you should ensure that subclasses adhere to the expectations set by the superclass or refactor the superclass to accommodate the new behavior.

- Violating LSP might lead to runtime errors, like calling methods that are unsupported or result in unexpected behavior.

4. Interface Segregation Principle (ISP)

Definition: A client should not be forced to implement interfaces it does not use.

Violating Example:

```
java

interface Worker {
    void work();
    void eat();
}

class Manager implements Worker {
    public void work() {
        System.out.println("Managing");
    }

    public void eat() {
        System.out.println("Eating");
    }
}

class Robot implements Worker {
    public void work() {
        System.out.println("Working");
    }

    public void eat() {
        throw new UnsupportedOperationException("Robots don't eat");
    }
}
```

Issue: The `Robot` class is forced to implement the `eat` method, which it doesn't need. This violates ISP.

Refactored Example:

java

```
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

class Manager implements Workable, Eatable {
    public void work() {
        System.out.println("Managing");
    }

    public void eat() {
        System.out.println("Eating");
    }
}

class Robot implements Workable {
    public void work() {
        System.out.println("Working");
    }
}
```

Explanation: Now, `Manager` implements both `Workable` and `Eatable` interfaces, while `Robot` only implements `Workable`, adhering to ISP.

Interview Questions:

- What is the Interface Segregation Principle?
- Can you provide an example where violating ISP causes problems?
- How would you refactor code that violates ISP?

Sample Answers:

- **ISP suggests that classes should not be forced to implement methods they don't use.** Each interface should have a small, specific set of methods.
- Violating ISP may lead to unnecessary implementation of methods, which can cause exceptions or unused code, like the `eat` method in the `Robot` class.

- To refactor, you would create more granular interfaces so that clients only implement what is necessary.

5. Dependency Inversion Principle (DIP)

Definition: High-level modules should not depend on low-level modules. Both should depend on abstractions. Furthermore, abstractions should not depend on details. Details should depend on abstractions.

Violating Example:

```
java

class LightBulb {
    public void turnOn() {
        System.out.println("Light is ON");
    }

    public void turnOff() {
        System.out.println("Light is OFF");
    }
}

class Switch {
    private LightBulb bulb;

    public Switch(LightBulb bulb) {
        this.bulb = bulb;
    }

    public void operate() {
        bulb.turnOn();
    }
}
```

Issue: The `Switch` class directly depends on the `LightBulb` class, which violates DIP. This makes it hard to switch to a different type of light source (like a fan).

Refactored Example:

java

```
interface Switchable {
    void turnOn();
    void turnOff();
}

class LightBulb implements Switchable {
    public void turnOn() {
        System.out.println("Light is ON");
    }

    public void turnOff() {
        System.out.println("Light is OFF");
    }
}

class Fan implements Switchable {
    public void turnOn() {
        System.out.println("Fan is ON");
    }

    public void turnOff() {
        System.out.println("Fan is OFF");
    }
}

class Switch {
    private Switchable device;

    public Switch(Switchable device) {
        this.device = device;
    }

    public void operate() {
        device.turnOn();
    }
}
```

Explanation: The `Switch` class now depends on the `Switchable` interface, which allows it to operate different types of devices without being tightly coupled to a specific implementation like `LightBulb`.

Interview Questions:

- What is the Dependency Inversion Principle?
- How does DIP promote flexibility in code?
- How would you refactor code that violates DIP?

Sample Answers:

- **DIP suggests that high-level classes should depend on abstractions** (e.g., interfaces), not concrete classes. Low-level classes should also depend on abstractions, which allows flexibility and easier modification.
 - DIP promotes flexibility by allowing you to swap low-level details without changing high-level logic. By depending on abstractions, you can easily introduce new classes or services.
 - To refactor, replace direct dependencies with interfaces or abstract classes that high-level modules depend on, as shown with the `Switchable` interface.
-

These principles together help make the code more maintainable, flexible, and testable. I hope this helps you with your interview prep!

Is this conversation helpful so far?