SOLID Principles Guide with Java Examples

Introduction

The SOLID principles are a set of five design principles in object-oriented programming that help developers create more maintainable, understandable, and flexible software. This guide provides an intuitive explanation of each principle, along with analogies and Java examples.

1. Single Responsibility Principle (SRP)

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

Analogy: Think of a library. The librarian is responsible for managing books, not repairing the building. If the librarian had to manage both, it would be too much for one person and could lead to mistakes.

Java Example:

```java
// Violation of SRP: This class has two responsibilities: managing
    employee data and calculating salary.
public class Employee {
    private String name;
    private double salary;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }

    public double calculateSalary() {
        // Salary calculation logic
        return salary * 1.2;
    }
}

// Following SRP: Separate responsibilities into different classes.
public class Employee {
    private String name;
    private double salary;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getSalary() {
        return salary;
    }

    public void setSalary(double salary) {
        this.salary = salary;
    }
}

public class SalaryCalculator {
    public double calculateSalary(Employee employee) {
        // Salary calculation logic
        return employee.getSalary() * 1.2;
    }
}
```

2. Open/Closed Principle (OCP)

Principle: Software entities should be open for extension but closed for modification. This means you should be able to add new functionality without changing existing code.

Analogy: Think of a power strip. You can plug in new devices without modifying the power strip itself.

Java Example:

```java
// Violation of OCP: Modifying existing class to add new functionality.
public class GraphicEditor {
    public void drawShape(Shape shape) {
        if (shape instanceof Circle) {
            drawCircle((Circle) shape);
        } else if (shape instanceof Square) {
            drawSquare((Square) shape);
        }
    }

    private void drawCircle(Circle circle) {
        // Draw circle
    }

    private void drawSquare(Square square) {
        // Draw square
    }
}

public class Shape {}
public class Circle extends Shape {}
public class Square extends Shape {}

// Following OCP: Extend functionality by adding new classes.
public abstract class Shape {
    public abstract void draw();
}

public class Circle extends Shape {
    @Override
    public void draw() {
        // Draw circle
    }
}

public class Square extends Shape {
    @Override
    public void draw() {
        // Draw square
    }
}

public class GraphicEditor {
    public void drawShape(Shape shape) {
        shape.draw();
    }
}
```

3. Liskov Substitution Principle (LSP)

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

Analogy: Think of a USB charger. Any device that follows the USB standard should work with any USB charger.

Java Example:

```java
// Violation of LSP: Subclass alters the expected behavior.
public class Rectangle {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}

public class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        this.width = width;
        this.height = width;
    }

    @Override
    public void setHeight(int height) {
        this.height = height;
        this.width = height;
    }
}

// Following LSP: Avoid altering behavior by using composition instead
    of inheritance.
public interface Shape {
    int getArea();
}

public class Rectangle implements Shape {
    protected int width;
    protected int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    @Override
    public int getArea() {
        return width * height;
    }
}

public class Square implements Shape {
    private int side;

    public void setSide(int side) {
        this.side = side;
    }

    @Override
    public int getArea() {
        return side * side;
    }
}
```

4. Interface Segregation Principle (ISP)

Principle: Clients should not be forced to depend on interfaces they do not use. This means creating specific interfaces for specific needs rather than one general-purpose interface.

Analogy: Think of a multi-tool. If you only need a knife, you shouldn't be forced to carry a tool with a screwdriver, pliers, and a bottle opener.

Java Example:

```java
// Violation of ISP: One general-purpose interface.
public interface Worker {
    void work();
    void eat();
}

public class HumanWorker implements Worker {
    @Override
    public void work() {
        // Working
    }

    @Override
    public void eat() {
        // Eating
    }
}

public class RobotWorker implements Worker {
    @Override
    public void work() {
        // Working
    }

    @Override
    public void eat() {
        // Robots don't eat
        throw new UnsupportedOperationException();
    }
}

// Following ISP: Specific interfaces for specific needs.
public interface Workable {
    void work();
}

public interface Eatable {
    void eat();
}

public class HumanWorker implements Workable, Eatable {
    @Override
    public void work() {
        // Working
    }

    @Override
    public void eat() {
        // Eating
    }
}

public class RobotWorker implements Workable {
    @Override
    public void work() {
        // Working
    }
}
```

5. Dependency Inversion Principle (DIP)

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

Analogy: Think of a car. A driver doesn't need to know how the engine works; they just use the interface provided (e.g., the gas pedal, brake pedal).

Java Example:

```java
// Violation of DIP: High-level module depends on low-level module.
public class Light {
    public void turnOn() {
        // Turn on the light
    }

    public void turnOff() {
        // Turn off the light
    }
}

public class Switch {
    private Light light;

    public Switch(Light light) {
        this.light = light;
    }

    public void operate(String command) {
        if (command.equals("ON")) {
            light.turnOn();
        } else if (command.equals("OFF")) {
            light.turnOff();
        }
    }
}

// Following DIP: High-level module depends on abstraction.
public interface Switchable {
    void turnOn();
    void turnOff();
}

```

```java
public class Light implements Switchable {
    @Override
    public void turnOn() {
        // Turn on the light
    }

    @Override
    public void turnOff() {
        // Turn off the light
    }
}

public class Switch {
    private Switchable device;

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

    public void operate(String command) {
        if (command.equals("ON")) {
            device.turnOn();
        } else if (command.equals("OFF")) {
            device.turnOff();
        }
    }
}
```