SOLID Principles

The SOLID principles are a set of five design principles intended to make software designs more understandable, flexible, and maintainable. These principles were introduced by Robert C. Martin (also known as Uncle Bob) and are a subset of many principles promoted by the agile software development methodology. The acronym SOLID stands for:

1. **S**ingle Responsibility Principle (SRP)
2. **O**pen/Closed Principle (OCP)
3. **L**iskov Substitution Principle (LSP)
4. **I**nterface Segregation Principle (ISP)
5. **D**ependency Inversion Principle (DIP)

1. Single Responsibility Principle (SRP)

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

Bad Example:

```
class User {
    private String name;
    private String email;

    // Getters and Setters for name and email

    // Save user to database
    public void saveUser() {
        // Code to save user to the database
    }

    // Send welcome email
    public void sendWelcomeEmail() {
        // Code to send welcome email to the user
    }
}
```

Explanation: In this example, the `User` class is handling multiple responsibilities: managing user data, saving the user to the database, and sending a welcome email. This violates the SRP.

Good Example:

```
class User {
    private String name;
    private String email;

    // Getters and Setters for name and email
}

class UserRepository {
    public void saveUser(User user) {
        // Code to save user to the database
    }
}

class UserEmailService {
    public void sendWelcomeEmail(User user) {
        // Code to send welcome email to the user
    }
}
```

Explanation: Here, each class has a single responsibility: `User` manages user data, `UserRepository` handles database operations, and `UserEmailService` manages email operations.

2. Open/Closed Principle (OCP)

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

Bad Example:

```
class Rectangle {
    private double length;
    private double width;

    // Getters and Setters

    public double area() {
        return length * width;
    }
}

class Circle {
    private double radius;

    // Getter and Setter

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

class AreaCalculator {
    public double calculate(Object shape) {
        if (shape instanceof Rectangle) {
            Rectangle rectangle = (Rectangle) shape;
            return rectangle.area();
        } else if (shape instanceof Circle) {
            Circle circle = (Circle) shape;
            return circle.area();
        }
        return 0;
    }
}
```

Explanation: The `AreaCalculator` class needs to be modified every time a new shape is added. This violates the OCP.

Good Example:

```
interface Shape {
    double area();
}

class Rectangle implements Shape {
    private double length;
    private double width;

    // Getters and Setters

    @Override
    public double area() {
        return length * width;
    }
}

class Circle implements Shape {
    private double radius;

    // Getter and Setter

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

class AreaCalculator {
    public double calculate(Shape shape) {
        return shape.area();
    }
}
```

Explanation: By using an interface (`Shape`), the `AreaCalculator` class can handle any shape without modification, adhering to the OCP.

3. Liskov Substitution Principle (LSP)

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

Bad Example:

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

class Ostrich extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Ostriches can't fly");
    }
}
```

Explanation: Substituting an `Ostrich` for a `Bird` will break the program, violating the LSP.

Good Example:

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

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

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

Explanation: Both `FlyingBird` and `Ostrich` are subclasses of `Bird` and can be substituted without affecting the correctness of the program.

4. Interface Segregation Principle (ISP)

Definition: No client should be forced to depend on methods it does not use.

Bad Example:

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

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

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

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

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

Explanation: `RobotWorker` is forced to implement the `eat` method, which it does not use, violating the ISP.

Good Example:

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

interface Eatable {
    void eat();
}

class HumanWorker implements Workable, Eatable {
    @Override
    public void work() {
        System.out.println("Working...");
    }

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

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

Explanation: By splitting the interfaces, `RobotWorker` only implements what it needs, adhering to the ISP.

5. Dependency Inversion Principle (DIP)

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

Bad Example:

```java
class LightBulb {
    public void turnOn() {
        System.out.println("LightBulb turned on");
    }

    public void turnOff() {
        System.out.println("LightBulb turned off");
    }
}

class Switch {
    private LightBulb lightBulb;

    public Switch() {
        this.lightBulb = new LightBulb();
    }

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

Explanation: The `Switch` class directly depends on the `LightBulb` class, violating the DIP.

Good Example:

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

class LightBulb implements Switchable {
    @Override
    public void turnOn() {
        System.out.println("LightBulb turned on");
    }

    @Override
    public void turnOff() {
        System.out.println("LightBulb turned off");
    }
}

class Switch {
    private Switchable device;

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

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

Explanation: The `Switch` class depends on the `Switchable` interface rather than a specific implementation, adhering to the DIP.

Interview Tips

1. **Understand Each Principle:**
   - Be able to define each principle and explain why it's important.
   - Use simple language to ensure clarity.
2. **Use Examples:**
   - Be prepared to give examples, both good and bad, to illustrate your understanding.
   - Practice explaining these examples succinctly.
3. **Common Interview Questions:**
   - **What are the SOLID principles? Can you explain each one?**
     - Provide definitions and a brief explanation for each principle.
   - **Can you give an example of a violation of the Single Responsibility Principle?**
     - Describe a scenario where a class has multiple responsibilities and how to refactor it.
   - **How would you refactor a class to adhere to the Open/Closed Principle?**
     - Explain how to use abstraction and interfaces to achieve this.
   - **What is the Liskov Substitution Principle and why is it important?**
     - Provide an example where substituting a subclass would break the program.
   - **How do you ensure your interfaces follow the Interface Segregation Principle?**
     - Describe how to break down interfaces so that implementing classes only need to implement what they use.
   - **Can you explain the Dependency Inversion Principle with an example?**
     - Describe how high-level modules should depend on abstractions, not concrete implementations.
4. **Approach to Answering:**
   - **Be Clear and Concise:** Avoid overly technical jargon. Make sure your explanations are easy to follow.
   - **Use Diagrams:** If possible, draw diagrams to illustrate your points during in-person interviews or on a whiteboard.
   - **Relate to Experience:** If you have used SOLID principles in past projects, mention these experiences to demonstrate practical knowledge.
   - **Practice:** Practice explaining these principles out loud to get comfortable with your explanations.