SOLID principles in Java

SOLID principles are object-oriented design concepts relevant to software development. SOLID is an acronym for five other class-design principles: Single Responsibility Principle, Open-Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle. SOLID is a structured design approach that ensures your software is modular and easy to maintain, understand, debug, and refactor. Following SOLID also helps save time and effort in both development and maintenance. SOLID prevents your code from becoming rigid and fragile, which helps you build long-lasting software.

Single responsibility principle: The Single Responsibility Principle (SRP) is a cornerstone principle in software design, particularly object-oriented programming. It emphasizes that a class, module, or any software component should **have only one reason to change**.

In simpler terms, each component should focus on a **single, well-defined functionality**. This promotes several benefits, including:

- **Improved maintainability:** When a change is needed, it's easier to locate and modify the responsible component without affecting unrelated parts of the code.
- **Enhanced code clarity:** Clearer responsibilities lead to easier understanding of the codebase, improving collaboration and reducing development time.
- **Reduced coupling:** Classes with specific responsibilities have less dependence on each other, making the code more modular and easier to test and reuse.

Java Examples Demonstrating Single Responsibility Principle (SRP):

1. Violating SRP:

```
public class Order {

    public void addProduct(Product product) {// Add product logic}

    public void calculateTotalPrice() { // Calculate total price logic}

    public void generateInvoice() { // Generate invoice logic }

    public void sendEmailNotification(String email) {// Send email
notification logic}

}
```

This Order class violates SRP by having several responsibilities:

- **Order management:** Adding products.
- **Calculation:** Calculating total price.
- **Document generation:** Generating invoice.
- **Communication:** Sending email notification.

```
public class OrderService {
    public void addProduct(Order order, Product product) {
// Add product logic
}
public double calculateTotalPrice(Order order) {
 // Calculate total price logic
  }
}
public class InvoiceService {
    public void generateInvoice(Order order) { // Generate invoice logic }
}
public class EmailNotificationService {
    public void sendEmailNotification(Order order, String email) {
        // Send email notification logic
    }
 }
}
```

Open-Closed Principle (OCP):

The Open-Closed Principle (OCP) states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification. This means you should be able to add new functionality without changing the existing

Code

Example Scenario

Let's imagine you're building a shape calculation application.

Bad Example (Violates OCP):

```
class ShapeCalculator {
    public double calculateArea(String shapeType) {
        if (shapeType.equals("circle")) {
            // Code to calculate circle area
            return area;
        } else if (shapeType.equals("rectangle")) {
            // Code to calculate rectangle area
            return area;
        }
        // ... (Imagine more 'else if' blocks for other shapes)
        return 0.0;
    }
}
```

- **The Problem:** Every time you need to support a new shape (triangle, square, etc.), you modify the `calculateArea` method directly. This makes it prone to errors, and harder to test as it grows.

Good Example (Adheres to OCP):

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

class Circle implements Shape {
    // ... circle-specific data
    @Override
    public double calculateArea() {
         // Code to calculate circle area
    }
}

class Rectangle implements Shape {
   // ... rectangle-specific data
    @Override
```

```
    public double calculateArea() {
        // Code to calculate rectangle area
    }
}

class ShapeCalculator {
    public double calculateArea(Shape shape) {
        return shape.calculateArea();
    }
}
```

How It Works (OCP in Action):

1. We create an interface `Shape` representing the contract of being able to calculate an area.
2. Specific shapes (`Circle`, `Rectangle`) implement the `Shape` interface, providing their own area calculations.
3. The `ShapeCalculator` is now incredibly flexible. To add a new shape (ex: Triangle), you simply create a `Triangle` class implementing `Shape`, and the `ShapeCalculator` can handle it without any internal changes.

Benefits of the Open-Closed Principle

- **Extensibility:** New features are added by writing new code rather than modifying old, working code.
- **Maintainability:** The risk of breaking existing functionality when making changes is reduced.
- **Testability:** Individual components (`Circle`, `Rectangle`, etc.) can be easily tested in isolation.

The Liskov Substitution Principle

The Liskov Substitution Principle states that subclasses should be substitutable for their base classes.

If class B is a subtype of class A, then instances of class B should be able to replace instances of class A without affecting the correctness of the program. This is because class B should inherit all the behavior of class A and possibly add some new behavior, but it should never remove or modify the behavior inherited from class A in a way that breaks the assumptions made by the code using class A.

Illustrative Example

Bad Example (Violates LSP):

```
class Rectangle {
    private int width, height;

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

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

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

class Square extends Rectangle {
    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width); // Force square dimensions
    }

    @Override
    public void setHeight(int height) {
```

```
        super.setWidth(height); // Force square dimensions
        super.setHeight(height);
    }
}
```

Problem: Let's say you have code that works with `Rectangle`:

```
Rectangle rect = new Rectangle();
rect.setWidth(5);
rect.setHeight(10);
System.out.println(rect.getArea()); // Output: 50 - as expected
```

Now, due to LSP, you should be able to substitute a `Square` since it's a subclass:

```
Rectangle rect = new Square(); // Seems fine due to inheritance
rect.setWidth(5);
rect.setHeight(10);
System.out.println(rect.getArea()); // Output: 100 - unexpected!
```

Good Example (Adheres to LSP):

```
interface Shape {
    int getArea();
}

class Rectangle implements Shape {
    // ... implementation with width & height ...
}

class Square implements Shape {
    // ... implementation with a single 'side' attribute ...
}
```

In this approach, you don't run into the substitution problem since the expectations of the `Shape` contract aren't being unexpectedly altered in any subclass.

Interface Segregation Principle (ISP)

The Interface Segregation Principle (ISP) states that clients should not be forced to depend upon interface members they do not use. This means **large, monolithic interfaces** should be broken down into **smaller, more specific interfaces.** Clients can then choose to implement only the interfaces relevant to their needs.

1. Violating ISP:

Consider a device interface for a multifunction printer:

```
public interface MultifunctionDevice {
    void print(String document);
    void scan(String document);
    void fax(String document, String recipient);
    // ... other functionalities like photocopy, etc.
}


public class LaserPrinter implements MultifunctionDevice {
    @Override
    public void print(String document) {
        // Print implementation
    }
    @Override
    public void scan(String document) {
        throw new UnsupportedOperationException("Scanning not supported");
    }
    @Override
    public void fax(String document, String recipient) {
```

```
        throw new UnsupportedOperationException("Faxing not supported");
    }

}
```

This code violates ISP because:

- The `MultifunctionDevice` interface contains methods for various functionalities (print, scan, fax, etc.).
- A class like `LaserPrinter` only supports printing but is forced to implement all methods, even if they are not relevant and throw exceptions.

2. Applying ISP:

We can address this by creating separate interfaces for specific functionalities:

```
public interface Printable {
    void print(String document);
}
public interface Scannable {
    void scan(String document);
}
public interface Faxable {
    void fax(String document, String recipient);
}
```

```
public class LaserPrinter implements Printable {

    @Override

    public void print(String document) {

        // Print implementation

    }

}

public class MultifunctionMachine implements Printable, Scannable, Faxable
{

    // Implementations for all functionalities

}
```

Here, we have:

- Separate interfaces for `Printable`, `Scannable`, and `Faxable`.
- `LaserPrinter` implements only the `Printable` interface as it only supports printing.
- `MultifunctionMachine` implements all three interfaces as it supports all functionalities.

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle (DIP) states that we should depend on abstractions (interfaces and abstract classes) instead of concrete implementations (classes). The abstractions should not depend on details; instead, the details should depend on abstractions.

Absolutely! Here's the code formatted for clarity and understanding, along with some explanations:

```
public class Car {
    private Engine engine; // Car has an Engine (Dependency)
    public Car(Engine e) {
        engine = e;  // Constructor: Assign Engine to the Car
    }
    public void start() {
        engine.start(); // Delegate starting to the Engine
    }
}
public class Engine {
    public void start() {
            System.out.println("Engine starting...");
    }
}
```

The code will work, for now, but what if we wanted to add another engine type, let's say a diesel engine? This will require refactoring the Car class. However, we can solve this by introducing a layer of abstraction. Instead of Car depending directly on Engine, let's add an interface

```
public interface Engine {
    public void start();
}
public class Car {
    private Engine engine;
```

```
    public Car(Engine e) {
        engine = e;
    }
    public void start() {
        engine.start();
    }
}


public class PetrolEngine implements Engine {
    @Override
    public void start() {
          System.out.println("Petrol engine starting...");
    }
}
public class DieselEngine implements Engine {
    @Override
    public void start() {
        // Implementation for starting a diesel engine
        System.out.println("Diesel engine starting...");
    }
}
```