SOLID Principles in Java

A Comprehensive Reference Guide

Table of Contents

1. Introduction
2. Single Responsibility Principle (SRP)
3. Open/Closed Principle (OCP)
4. Liskov Substitution Principle (LSP)
5. Interface Segregation Principle (ISP)
6. Dependency Inversion Principle (DIP)
7. Benefits of SOLID Principles
8. Quick Reference

Introduction

The SOLID principles are five fundamental design principles in object-oriented programming that help create more maintainable, flexible, and robust code. These principles were introduced by Robert C. Martin (Uncle Bob) and form the foundation of clean code architecture.

SOLID stands for:

- **S** - Single Responsibility Principle
- **O** - Open/Closed Principle
- **L** - Liskov Substitution Principle
- **I** - Interface Segregation Principle
- **D** - Dependency Inversion Principle

Single Responsibility Principle (SRP)

Definition

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

Problem

When a class has multiple responsibilities, changes to one responsibility can affect other responsibilities, making the code fragile and harder to maintain.

Example - Violation of SRP

java

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

    // User data management
    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }

    // Email functionality - violates SRP
    public void sendEmail(String message) {
        // Email sending logic
        System.out.println("Sending email to " + email + ": " + message);
    }

    // Database operations - violates SRP
    public void saveToDatabase() {
        // Database save logic
        System.out.println("Saving user " + name + " to database");
    }

    // Validation logic - violates SRP
    public boolean validateEmail() {
        return email.contains("@");
    }
}
```

Example - Following SRP

java

```java
// User class - only handles user data
class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

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

    public String getName() {
        return name;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getEmail() {
        return email;
    }
}

// EmailService class - handles email operations
class EmailService {
    public void sendEmail(User user, String message) {
        System.out.println("Sending email to " + user.getEmail() + ": " + message);
    }
}

// UserRepository class - handles database operations
class UserRepository {
    public void save(User user) {
        System.out.println("Saving user " + user.getName() + " to database");
    }

    public User findByEmail(String email) {
        // Database query logic
        return null; // Placeholder
```

```
    }
}

// UserValidator class - handles validation
class UserValidator {
    public boolean validateEmail(String email) {
        return email != null && email.contains("@") && email.contains(".");
    }

    public boolean validateName(String name) {
        return name != null && !name.trim().isEmpty();
    }
}
```

Benefits

- Easier to understand and maintain
- Changes to one responsibility don't affect others
- Better testability
- Improved code reusability

Open/Closed Principle (OCP)

Definition

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

Problem

Modifying existing code to add new functionality can introduce bugs and break existing functionality.

Example - Following OCP

java

```java
// Base abstraction
abstract class Shape {
    abstract double calculateArea();
    abstract double calculatePerimeter();
}

// Existing implementations
class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }

    @Override
    double calculateArea() {
        return width * height;
    }

    @Override
    double calculatePerimeter() {
        return 2 * (width + height);
    }
}

class Circle extends Shape {
    private double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

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

    @Override
    double calculatePerimeter() {
        return 2 * Math.PI * radius;
    }
```

```java
}

// New shape can be added without modifying existing code
class Triangle extends Shape {
    private double side1, side2, side3;

    public Triangle(double side1, double side2, double side3) {
        this.side1 = side1;
        this.side2 = side2;
        this.side3 = side3;
    }

    @Override
    double calculateArea() {
        // Using Heron's formula
        double s = (side1 + side2 + side3) / 2;
        return Math.sqrt(s * (s - side1) * (s - side2) * (s - side3));
    }

    @Override
    double calculatePerimeter() {
        return side1 + side2 + side3;
    }
}

// Shape calculator that works with any shape
class ShapeCalculator {
    public double getTotalArea(Shape[] shapes) {
        double totalArea = 0;
        for (Shape shape : shapes) {
            totalArea += shape.calculateArea();
        }
        return totalArea;
    }
}
```

Strategy Pattern Example

```java
```

```java
// Payment processing following OCP
interface PaymentProcessor {
    void processPayment(double amount);
}

class CreditCardProcessor implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing $" + amount + " via Credit Card");
    }
}

class PayPalProcessor implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing $" + amount + " via PayPal");
    }
}

// New payment method can be added without modifying existing code
class BitcoinProcessor implements PaymentProcessor {
    @Override
    public void processPayment(double amount) {
        System.out.println("Processing $" + amount + " via Bitcoin");
    }
}

class PaymentService {
    private PaymentProcessor processor;

    public PaymentService(PaymentProcessor processor) {
        this.processor = processor;
    }

    public void processPayment(double amount) {
        processor.processPayment(amount);
    }
}
```

Liskov Substitution Principle (LSP)

Definition

Objects of a superclass should be replaceable with objects of a subclass without breaking the application.

Problem

When subclasses don't properly substitute their parent classes, it can lead to unexpected behavior and violations of the expected contract.

Example - Violation of LSP

java

```java
// Bad example - violates LSP
class Bird {
    public void fly() {
        System.out.println("Bird is flying");
    }
}

class Penguin extends Bird {
    @Override
    public void fly() {
        // Penguins can't fly!
        throw new UnsupportedOperationException("Penguins can't fly");
    }
}

// This would break when penguin is used
class BirdWatcher {
    public void watchBird(Bird bird) {
        bird.fly(); // This will throw exception for Penguin
    }
}
```

Example - Following LSP

java

```java
// Good example - follows LSP
abstract class Bird {
    public void eat() {
        System.out.println("Bird is eating");
    }

    public void sleep() {
        System.out.println("Bird is sleeping");
    }
}

// Separate interface for flying behavior
interface Flyable {
    void fly();
}

// Separate interface for swimming behavior
interface Swimmable {
    void swim();
}

class Sparrow extends Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Sparrow is flying");
    }
}

class Eagle extends Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Eagle is soaring high");
    }
}

class Penguin extends Bird implements Swimmable {
    @Override
    public void swim() {
        System.out.println("Penguin is swimming");
    }
}

class Duck extends Bird implements Flyable, Swimmable {
```

```
    @Override
    public void fly() {
        System.out.println("Duck is flying");
    }

    @Override
    public void swim() {
        System.out.println("Duck is swimming");
    }
}

// Usage examples
class BirdWatcher {
    public void watchBird(Bird bird) {
        bird.eat(); // Works for all birds
        bird.sleep(); // Works for all birds
    }

    public void watchFlyingBird(Flyable flyingBird) {
        flyingBird.fly(); // Only works for birds that can fly
    }

    public void watchSwimmingBird(Swimmable swimmingBird) {
        swimmingBird.swim(); // Only works for birds that can swim
    }
}
```

Rectangle-Square Example

java

```
// Classic LSP violation example
class Rectangle {
    protected double width;
    protected double height;

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

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

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

// This violates LSP
class Square extends Rectangle {
    @Override
    public void setWidth(double width) {
        this.width = width;
        this.height = width; // Square maintains equal sides
    }

    @Override
    public void setHeight(double height) {
        this.width = height;
        this.height = height; // Square maintains equal sides
    }
}

// Better approach following LSP
abstract class Shape {
    abstract double getArea();
}

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

    public Rectangle(double width, double height) {
```

```
        this.width = width;
        this.height = height;
    }

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

class Square extends Shape {
    private double side;

    public Square(double side) {
        this.side = side;
    }

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

Interface Segregation Principle (ISP)

Definition

A client should not be forced to implement interfaces it doesn't use. Many client-specific interfaces are better than one general-purpose interface.

Problem

Large interfaces force classes to implement methods they don't need, leading to unnecessary dependencies and potential violations.

Example - Violation of ISP

java

```
// Bad example - violates ISP
interface Worker {
    void work();
    void eat();
    void sleep();
    void takeBreak();
}

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

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

    @Override
    public void sleep() {
        System.out.println("Human sleeping");
    }

    @Override
    public void takeBreak() {
        System.out.println("Human taking break");
    }
}

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

    @Override
    public void eat() {
        // Robots don't eat - forced to implement
        throw new UnsupportedOperationException("Robots don't eat");
    }

    @Override
```

```
    public void sleep() {
        // Robots don't sleep - forced to implement
        throw new UnsupportedOperationException("Robots don't sleep");
    }

    @Override
    public void takeBreak() {
        // Robots don't take breaks - forced to implement
        throw new UnsupportedOperationException("Robots don't take breaks");
    }
}
```

Example - Following ISP

java

```java
// Good example - follows ISP
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

interface Sleepable {
    void sleep();
}

interface Breakable {
    void takeBreak();
}

class Human implements Workable, Eatable, Sleepable, Breakable {
    @Override
    public void work() {
        System.out.println("Human working");
    }

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

    @Override
    public void sleep() {
        System.out.println("Human sleeping");
    }

    @Override
    public void takeBreak() {
        System.out.println("Human taking break");
    }
}

class Robot implements Workable {
    @Override
    public void work() {
        System.out.println("Robot working");
```

```
    }

    // Robot can also implement Rechargeable interface
}

interface Rechargeable {
    void recharge();
}

class AdvancedRobot implements Workable, Rechargeable {
    @Override
    public void work() {
        System.out.println("Advanced robot working");
    }

    @Override
    public void recharge() {
        System.out.println("Advanced robot recharging");
    }
}
```

Document Management Example

java

```java
// Document management following ISP
interface Readable {
    String read();
}

interface Writable {
    void write(String content);
}

interface Printable {
    void print();
}

interface Scannable {
    void scan();
}

class Document implements Readable, Writable {
    private String content;

    @Override
    public String read() {
        return content;
    }

    @Override
    public void write(String content) {
        this.content = content;
    }
}

class Printer implements Printable {
    @Override
    public void print() {
        System.out.println("Printing document");
    }
}

class Scanner implements Scannable {
    @Override
    public void scan() {
        System.out.println("Scanning document");
    }
```

```
}

class MultiFunctionDevice implements Printable, Scannable {
    @Override
    public void print() {
        System.out.println("MFD printing");
    }

    @Override
    public void scan() {
        System.out.println("MFD scanning");
    }
}
```

Dependency Inversion Principle (DIP)

Definition

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Problem

When high-level modules depend directly on low-level modules, the system becomes tightly coupled and difficult to modify or test.

Example - Violation of DIP

java

```java
// Bad example - violates DIP
class MySQLDatabase {
    public void save(String data) {
        System.out.println("Saving to MySQL: " + data);
    }

    public String retrieve(String id) {
        return "Data from MySQL for ID: " + id;
    }
}

class UserService {
    private MySQLDatabase database; // Direct dependency on concrete class

    public UserService() {
        this.database = new MySQLDatabase(); // Tight coupling
    }

    public void saveUser(String userData) {
        database.save(userData);
    }

    public String getUser(String userId) {
        return database.retrieve(userId);
    }
}
```

Example - Following DIP

java

```java
// Good example - follows DIP
interface Database {
    void save(String data);
    String retrieve(String id);
}

class MySQLDatabase implements Database {
    @Override
    public void save(String data) {
        System.out.println("Saving to MySQL: " + data);
    }

    @Override
    public String retrieve(String id) {
        return "Data from MySQL for ID: " + id;
    }
}

class PostgreSQLDatabase implements Database {
    @Override
    public void save(String data) {
        System.out.println("Saving to PostgreSQL: " + data);
    }

    @Override
    public String retrieve(String id) {
        return "Data from PostgreSQL for ID: " + id;
    }
}

class MongoDatabase implements Database {
    @Override
    public void save(String data) {
        System.out.println("Saving to MongoDB: " + data);
    }

    @Override
    public String retrieve(String id) {
        return "Data from MongoDB for ID: " + id;
    }
}

class UserService {
```

```
    private Database database; // Depends on abstraction

    public UserService(Database database) {
        this.database = database; // Dependency injection
    }

    public void saveUser(String userData) {
        database.save(userData);
    }

    public String getUser(String userId) {
        return database.retrieve(userId);
    }
}

// Usage
class Application {
    public static void main(String[] args) {
        // Can easily switch between different databases
        Database mysqlDb = new MySQLDatabase();
        Database postgresDb = new PostgreSQLDatabase();
        Database mongoDb = new MongoDatabase();

        UserService userService1 = new UserService(mysqlDb);
        UserService userService2 = new UserService(postgresDb);
        UserService userService3 = new UserService(mongoDb);

        userService1.saveUser("User data 1");
        userService2.saveUser("User data 2");
        userService3.saveUser("User data 3");
    }
}
```

Notification System Example

java

```java
// Notification system following DIP
interface NotificationService {
    void sendNotification(String message, String recipient);
}

class EmailService implements NotificationService {
    @Override
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending email to " + recipient + ": " + message);
    }
}

class SMSService implements NotificationService {
    @Override
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending SMS to " + recipient + ": " + message);
    }
}

class PushNotificationService implements NotificationService {
    @Override
    public void sendNotification(String message, String recipient) {
        System.out.println("Sending push notification to " + recipient + ": " + message);
    }
}

class OrderService {
    private NotificationService notificationService;

    public OrderService(NotificationService notificationService) {
        this.notificationService = notificationService;
    }

    public void processOrder(String orderId, String customerContact) {
        // Process order logic
        System.out.println("Processing order: " + orderId);

        // Send notification
        notificationService.sendNotification(
            "Your order " + orderId + " has been processed",
            customerContact
        );
```

```
    }
}
```

Benefits of SOLID Principles

Code Maintainability

- **Single Responsibility**: Each class has a single, well-defined purpose
- **Open/Closed**: New features can be added without modifying existing code
- **Liskov Substitution**: Subclasses can be used interchangeably with their parent classes
- **Interface Segregation**: Classes only implement methods they actually need
- **Dependency Inversion**: High-level modules are not affected by low-level changes

Testability

- **Easier Unit Testing**: Each class has a single responsibility, making it easier to test
- **Mocking**: Dependencies can be easily mocked due to loose coupling
- **Isolation**: Components can be tested in isolation

Flexibility and Extensibility

- **Easy to Extend**: New functionality can be added without breaking existing code
- **Plug-and-Play**: Components can be easily swapped due to dependency injection
- **Modular Design**: System is composed of loosely coupled, highly cohesive modules

Code Reusability

- **Modular Components**: Single-purpose classes can be reused in different contexts
- **Interface-based Design**: Implementations can be reused wherever the interface is needed

Reduced Coupling

- **Loose Coupling**: Components depend on abstractions rather than concrete implementations
- **High Cohesion**: Related functionality is grouped together in single classes

---

Quick Reference

Single Responsibility Principle (SRP)

- **Remember**: One class, one job

- **Question to ask**: "Does this class have more than one reason to change?"
- **Solution**: Split classes with multiple responsibilities into separate classes

Open/Closed Principle (OCP)

- **Remember**: Open for extension, closed for modification
- **Question to ask**: "Can I add new functionality without modifying existing code?"
- **Solution**: Use abstractions, interfaces, and inheritance

Liskov Substitution Principle (LSP)

- **Remember**: Subtypes must be substitutable for their base types
- **Question to ask**: "Can I replace the parent class with this subclass without breaking anything?"
- **Solution**: Ensure subclasses honor the contract of their parent classes

Interface Segregation Principle (ISP)

- **Remember**: Many small interfaces are better than one large interface
- **Question to ask**: "Am I forcing clients to implement methods they don't need?"
- **Solution**: Split large interfaces into smaller, more specific ones

Dependency Inversion Principle (DIP)

- **Remember**: Depend on abstractions, not concretions
- **Question to ask**: "Is my high-level module depending on low-level modules?"
- **Solution**: Use dependency injection and depend on interfaces

---

Conclusion

The SOLID principles are not just theoretical concepts but practical guidelines that lead to better software design. By following these principles, you create code that is:

- **Maintainable**: Easy to understand and modify
- **Testable**: Components can be tested in isolation
- **Flexible**: Easy to extend and adapt to changing requirements
- **Robust**: Less prone to bugs and breaking changes
- **Reusable**: Components can be used in different contexts

Remember that these principles work best when applied together. They complement each other and, when combined, create a robust foundation for object-oriented design.

The key is to understand the intent behind each principle and apply them judiciously. Not every situation requires strict adherence to all principles, but understanding them helps you make better design decisions and write cleaner, more maintainable code.

Document prepared: July 2025 Subject: SOLID Principles in Java Version: 1.0