In the world of software development, creating clear, efficient, and maintainable code is paramount. Among the many guidelines that have emerged over time, the SOLID principles stand out as a foundational set of concepts for object-oriented design. The SOLID acronym represents five design principles that, when followed, can lead to better software architecture and easier code maintenance. Let's dive into each principle with a focus on Java.

S - Single Responsibility Principle (SRP)

Every class should have a single responsibility. In other words, a class should only have one reason to change. This makes the code easier to understand and modify.

Example:

```
class Report {
    public void generateReport() {
        // Code for generating report
    }
}

class ReportPrinter {
    public void printReport(Report report) {
        // Code for printing the report
    }
}
```

In the above example, we've separated the responsibilities of generating and printing a report into two classes. This makes it clearer what each class does and reduces the risk of changes affecting multiple functionalities.

O - Open/Closed Principle (OCP)

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 existing code.

Example:

```
abstract class Shape {
    abstract void draw();
}
```

```
class Circle extends Shape {
    void draw() {
        // Drawing a circle
    }
}

class Rectangle extends Shape {
    void draw() {
        // Drawing a rectangle
    }
}
```

In this case, we can add new shapes by creating new classes (like `Triangle`), without touching the existing `Shape` class or its implementations.

L - Liskov Substitution Principle (LSP)

Objects of a superclass should be replaceable with objects of a subclass without affecting the functionality of the program. In simpler terms, if class B is a subclass of class A, you should be able to use B in place of A without breaking the program.

Example:

```
class Bird {
    void fly() {
        // Logic for flying
    }
}

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

Here, using `Ostrich` in a context where a bird is expected would violate the LSP, since `Ostrich` does not conform to the behavior expected of a `Bird`. A better design would be to separate birds that can fly from those that can't.

I - Interface Segregation Principle (ISP)

Clients should not be forced to depend on interfaces they do not use. This encourages the creation of more focused, smaller interfaces rather than a large, general-purpose one.

Example:

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

interface Eatable {
    void eat();
}

class Human implements Workable, Eatable {
    public void work() {
        // Logic for working
    }

    public void eat() {
        // Logic for eating
    }
}

class Robot implements Workable {
    public void work() {
        // Logic for working
    }
}
```

By having separate interfaces (`Workable` and `Eatable`), we allow different implementations, such as the `Robot` , to only implement the functionality it needs.

D - Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules; both should depend on abstractions. Additionally, abstractions should not depend on details; details should depend on abstractions. This principle encourages a decoupled architecture.

Example:

```
interface Database {
    void connect();
}

class MySQLDatabase implements Database {
    public void connect() {
        // MySQL connection logic
    }
}

class UserService {
    private Database database;

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

    public void saveUser() {
        database.connect();
        // Logic for saving a user
    }
}
```

In this example, `UserService` depends on the `Database` abstraction instead of a concrete implementation, allowing us to switch between different database systems easily.

By understanding and applying the SOLID principles in Java, you can greatly improve the design and maintainability of your software projects. The principles might seem abstract at first, but through consistent practice, they can become second nature to any developer striving for excellence in their code.

What is the Single Responsibility Principle (SRP)?

The Single Responsibility Principle is one of the five SOLID principles of object-oriented design. Coined by Robert C. Martin, it states that a class should have only one reason to change; that is, it should have one responsibility. This means that a class should encapsulate only one part of the functionality provided by the software and not take on too many responsibilities.

Importance of SRP in Software Development

Adhering to the Single Responsibility Principle can yield several benefits:

1. **Improved Maintainability**: When a class focuses on a single responsibility, it becomes easier to understand and maintain. Changes in that particular functionality only require modifications within that specific class.
2. **Enhanced Readability**: A class that follows SRP has a clear purpose, making it more readable and understandable for new developers entering the codebase.
3. **Reduced Complexity**: By breaking down classes into smaller and more focused units, the complexity of a software application is significantly reduced.
4. **Easier Testing**: When classes have a single responsibility, unit testing becomes more manageable. Each class can be tested in isolation, leading to a more robust testing framework overall.

Example of SRP in Java

Let's illustrate the Single Responsibility Principle with a simple Java example. We start with a class that violates SRP:

```
public class User {
    private String username;
    private String email;

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

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

    public String generateReport() {
        // Code to generate user report
        return "User Report";
    }
}
```

In this example, the `User` class is responsible for both user management (saving the user) and report generation. This violates the SRP because it has more than one reason to change.

Now, let's refactor this example to adhere to the principle:

```
public class User {
    private String username;
    private String email;

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

    // Getters and other user-related methods
}

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

public class UserReport {
    public String generateReport(User user) {
        // Code to generate user report
        return "User Report for: " + user.getUsername();
    }
}
```

In this refactored code, we have split the responsibilities into three classes:

- The `User` class maintains user data.
- The `UserRepository` handles database interactions.
- The `UserReport` is responsible for generating reports.

This makes each class adhere to the Single Responsibility Principle.

Interview Questions

Question 1: What is the Single Responsibility Principle?

Answer: The Single Responsibility Principle (SRP) is a design principle that states that a class should have only one reason to change, meaning it should only have one responsibility. This helps in reducing complexity and improving maintainability in software design.

Question 2: Can you provide an example of SRP violation?

Answer: An SRP violation occurs when a class takes on multiple responsibilities. For instance, in a Java class that handles user authentication and also generates user reports, if any change is required for report generation, it could impact authentication functionality, leading to code that is harder to maintain.

Question 3: Why is SRP important for software maintainability?

Answer: SRP is vital for maintainability because it localizes changes within a specific class. With a single responsibility, developers can modify or enhance functionality without affecting unrelated parts of the codebase, leading to fewer bugs and easier debugging.

Question 4: How can adhering to SRP influence unit testing?

Answer: Adhering to SRP allows for more effective unit testing since each class can be tested independently. Since each class has only one responsibility, testing becomes straightforward, making it easier to identify what part of the code is failing.

Question 5: What are common challenges when implementing SRP?

Answer: One common challenge is identifying the correct level of granularity for a class's responsibility. It can be tempting to group too many functionalities into one class, leading to potential SRP violations. Finding a balance between too many small classes and too few larger classes can be difficult, especially in larger applications.

Question 6: Can you give an example of how SRP could be misapplied?

Answer: An instance of misapplying SRP could be creating multiple classes for very minor responsibilities, leading to an over-engineered codebase with excessive class fragmentation. This can result in unnecessary complexity rather than simplifying the code. It's essential to ensure that while adhering to SRP, the benefits of clarity and maintainability do not come at the cost of increased complexity.

Question 7: How do you identify the responsibilities of a class?

Answer: Responsibilities can be identified by analyzing the functionalities that a class should provide. Consider interactions with other classes, functionalities required by users, and the overall design of the system. Techniques like use-case analysis and discussions with stakeholders can help clarify what the class's responsibilities should be.

What is the Open/Closed Principle?

The Open/Closed Principle (OCP) is all about maximizing code maintainability. Essentially, it advises developers to design software modules in such a way that they can be extended with new functionality without altering existing code. By following this principle, developers can minimize the risk of introducing new bugs and ensure that existing functionality remains stable.

In simple terms, if you want to add new features to a codebase, you should be able to do so without changing the existing code structure. This is crucial for large applications, where changes can have widespread effects.

Example of the Open/Closed Principle

Let's create a simple example to illustrate the Open/Closed Principle. Suppose we have a system that calculates the area of different shapes. Initially, we have just a rectangle.

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

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

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

Here's how you would normally calculate the area using the above class. The problem arises when you want to add new shapes like circles or squares. If you directly modify the current class to accommodate new shapes, you go against the Open/Closed Principle.

Applying OCP via Interfaces

To adhere to the Open/Closed Principle, we can define a Shape interface:

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

Now, let's implement this interface for our Rectangle and add a Circle:

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

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

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

class Circle implements Shape {
    private double radius;

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

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

With this structure, you can add new shapes by simply creating new classes that implement the `Shape` interface. You can extend the functionality without touching the existing code. This is the essence of the Open/Closed Principle.

Interview Questions on the Open/Closed Principle

Question 1: Can you explain what the Open/Closed Principle is in your own words?

Answer: The Open/Closed Principle states that software entities like classes, modules, and functions should be open for extension but closed for modification. This means you should be able to extend the behavior of a system without modifying the existing code, which helps reduce bugs and maintain the stability of the codebase.

Question 2: Why is the Open/Closed Principle important in software development?

Answer: The Open/Closed Principle is important because it promotes code reusability and maintainability. By designing systems that are open for extension, developers can add new features without affecting the existing code or introducing new bugs. This is particularly crucial in large applications where changes can have unforeseen consequences.

Question 3: Give an example of a scenario where not following the Open/Closed Principle can lead to issues.

Answer: Consider a scenario in an e-commerce application where we have a class called `Order` that processes different types of orders. If we keep adding methods to this class to handle new order types, it becomes bloated and hard to maintain. Not following the OCP can lead to a situation where modifying the `Order` class to add a new order type introduces bugs in existing order types, making the code fragile and difficult to manage.

Question 4: How can you implement the Open/Closed Principle in Java applications?

Answer: You can implement the Open/Closed Principle in Java using interfaces or abstract classes. By defining a contract through an interface, you allow new functionalities to be added as new classes implementing that interface. This avoids modification of existing classes, thereby adhering to OCP. Using design patterns like Strategy or Factory can also help in applying this principle effectively.

Question 5: Can you provide a real-world example of the Open/Closed Principle in action?

Answer: A real-world example of the Open/Closed Principle can be seen in payment processing systems. Initially, the system might support credit card payments. As time goes on, support for PayPal, Bitcoin, and other payment methods may be required. By using an interface (`PaymentMethod`) and creating separate classes for each payment method, we can easily extend the functionality without altering the existing payment processing code. This encapsulation makes the system more maintainable and flexible for future enhancements.

The Liskov Substitution Principle (LSP) is one of the five SOLID principles of object-oriented programming introduced by Barbara Liskov in 1987. The principle states that if S is a subtype of T, then objects of type T should be replaceable with objects of type S without altering any of the desirable properties of the program. In simpler terms, this means that a subclass should be substitutable for its superclass.

Why is LSP Important?

LSP promotes the concept of "behavioral compatibility" and encourages developers to create robust and maintainable code. When subclasses can be substituted for their parent classes without any issues,

it ensures that the system as a whole remains stable and predictable. This is crucial when working on larger projects or teams, as it allows for easier code reuse and modification.

Practical Example in Java

Let's consider a simple example of LSP in Java involving shapes:

```
// Base class
abstract class Shape {
    abstract double area();
}

// Subclass, inherits from Shape
class Rectangle extends Shape {
    private double width;
    private double height;

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

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

// Subclass, inherits from Shape
class Square extends Shape {
    private double side;

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

    @Override
    double area() {
        return side * side;
    }
}

// Using the classes
```

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

In this example, both `Rectangle` and `Square` are subclasses of `Shape`. The `calculateArea` method can take either a `Rectangle` or a `Square` as a parameter without any problem, as both properly implement the `area` method. This is a demonstration of LSP, where the subclasses can be substituted for the superclass without causing issues.

Interview Questions and Answers

1. What is the Liskov Substitution Principle?

Answer: The Liskov Substitution Principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. This principle emphasizes that subclasses should fully comply with the expectations set by their superclasses.

---

2. Can you give an example of a violation of the Liskov Substitution Principle?

Answer: A classic violation occurs when a subclass overrides a method from the superclass in a way that changes expected behavior. For instance, if a subclass of a `Bird` class represents a `Penguin`, which cannot fly, it violates the LSP if the superclass method `fly()` is called, leading to unexpected behavior or errors.

Example:

```
class Bird {
    public void fly() {
        System.out.println("I can fly!");
    }
}

class Penguin extends Bird {
    @Override
    public void fly() {
        throw new UnsupportedOperationException("Penguins cannot fly!");
    }
}
```

Here, substituting `Penguin` for `Bird` breaks the expected behavior.

---

3. Why is it important to adhere to the Liskov Substitution Principle?

Answer: Adhering to the LSP is crucial for maintaining the integrity and functionality of a system. It allows for the proper use of polymorphism, ensuring that subclasses can be used interchangeably with their superclass. This leads to more reliable, extensible, and understandable code, which is essential for effectively managing larger codebases and teams.

---

4. How can you ensure compliance with the Liskov Substitution Principle in your code?

Answer: To ensure compliance with LSP, developers should:

- Design subclasses to behave in a way that conforms to the contracts established by their superclasses.
- Avoid violating the expectations of superclass methods, especially regarding side effects and preconditions.
- Use abstract classes and interfaces wisely to enforce a consistent behavior across all subclasses.

---

5. How does the Liskov Substitution Principle relate to other SOLID principles?

Answer: LSP is interconnected with other SOLID principles:

- It complements the Open/Closed Principle by allowing subclasses to extend functionality without modifying existing code.
- It supports the Interface Segregation Principle by ensuring interfaces are designed to allow for proper substitutions.
  Understanding and applying these principles together results in more robust object-oriented designs.

By integrating the Liskov Substitution Principle into your Java programming practices, you'll pave the way for scalable and maintainable applications. It not only enhances code reusability but also fosters a collaborative environment where various components work in harmony.

What is the Interface Segregation Principle?

The Interface Segregation Principle is a design guideline that states that no client should be forced to depend on methods it does not use. In simpler terms, it encourages you to create smaller, more specific interfaces rather than a large one that covers multiple functionalities. The primary goal of ISP is to

reduce the side effects and frequency of changes by splitting the software into multiple, specialized interfaces.

Why is ISP Important?

1. **Overhead Reduction**: ISP helps avoid bloated interfaces, which can lead to unnecessary dependencies that increase the complexity of code.
2. **Improved Flexibility**: Smaller, focused interfaces make it easier to update parts of the code without affecting other segments.
3. **Enhanced Maintainability**: Code that adheres to ISP tends to be easier to understand, test, and maintain due to its specialization.

Example of ISP in Java

Let's take a look at a simple example to illustrate this principle.

Without ISP

Consider a large interface for a machine:

```
public interface Machine {
    void printDocument();
    void scanDocument();
    void faxDocument();
}
```

Now, let's assume we have two classes, a `Printer` and a `Scanner`, which implement the `Machine` interface:

```
public class Printer implements Machine {
    public void printDocument() {
        System.out.println("Printing document...");
    }

    public void scanDocument() {
        throw new UnsupportedOperationException("This printer cannot scar
    }

    public void faxDocument() {
```

```
        throw new UnsupportedOperationException("This printer cannot fax'
    }
}

public class Scanner implements Machine {
    public void printDocument() {
        throw new UnsupportedOperationException("This scanner cannot prir
    }

    public void scanDocument() {
        System.out.println("Scanning document...");
    }

    public void faxDocument() {
        throw new UnsupportedOperationException("This scanner cannot fax'
    }
}
```

In this example, both `Printer` and `Scanner` end up with unnecessary methods that are not applicable to them, violating the ISP.

With ISP

Let's refactor the above example to adhere to the ISP:

```
public interface Printer {
    void printDocument();
}

public interface Scanner {
    void scanDocument();
}

public interface FaxMachine {
    void faxDocument();
}

public class SimplePrinter implements Printer {
    public void printDocument() {
        System.out.println("Printing document...");
    }
}
```

```
public class SimpleScanner implements Scanner {
    public void scanDocument() {
        System.out.println("Scanning document...");
    }
}

public class SimpleFax implements FaxMachine {
    public void faxDocument() {
        System.out.println("Faxing document...");
    }
}
```

Now, each class implements only the interfaces that are relevant to their functionality, adhering to the Interface Segregation Principle.

Interview Questions on ISP

To help you prepare for an interview or deepen your understanding of ISP, here are some potential interview questions along with answers.

Question 1: What is the Interface Segregation Principle?

Answer: The Interface Segregation Principle states that clients should not be forced to depend on interfaces they do not use. This principle advocates splitting large interfaces into smaller, more specific ones to reduce dependencies and increase code maintainability.

Question 2: Can you give an example of a violation of ISP?

Answer: Sure! An example of ISP violation is a large interface like `Machine` that includes methods such as `printDocument()`, `scanDocument()`, and `faxDocument()`. If a class like `Printer` implements this interface, it might need to implement the `scanDocument()` and `faxDocument()` methods, which could lead to `UnsupportedOperationExceptions` being thrown, thus violating ISP.

Question 3: How does ISP contribute to code maintainability?

Answer: ISP helps maintainability by ensuring that interfaces contain only relevant methods. Smaller, specialized interfaces lead to a reduction in the number of methods and dependencies. This makes the code easier to understand, validate, and alter over time.

Question 4: What is a practical approach to implementing ISP in a large codebase?

Answer: A practical approach to implement ISP is to start by examining existing large interfaces. Identify methods that are unrelated and categorize them into smaller interfaces. Then, refactor the code to use the new specialized interfaces. This gradual approach allows for smoother transitions without breaking existing functionality.

Question 5: What are common pitfalls while applying ISP?

Answer: Common pitfalls include creating too many interfaces leading to over-segregation, which can make the code harder to manage. It's important to strike a balance — interfaces should be specific yet not so granular that they become cumbersome to use and implement.

By understanding and applying the Interface Segregation Principle, developers can write cleaner, more maintainable, and less error-prone code in Java.

What is the Dependency Inversion Principle?

The Dependency Inversion Principle (DIP) is a core principle of object-oriented programming that states:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces).
2. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.

This principle encourages developers to design their systems such that they are not tied to specific implementations but rather to interfaces that define the behaviors they require. This enhances flexibility, scalability, and testability of code, while also reducing the impact of changes.

Example of DIP in Java

Let's illustrate DIP with a simple example. Consider a scenario where we have a `LightBulb` class that can be turned on and off by a `Switch`.

Without Dependency Inversion

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

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

class Switch {
    private LightBulb lightBulb;

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

    public void operate(boolean turnOn) {
        if(turnOn) {
            lightBulb.turnOn();
        } else {
            lightBulb.turnOff();
        }
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
        LightBulb bulb = new LightBulb();
        Switch lightSwitch = new Switch(bulb);
        lightSwitch.operate(true); // Light Bulb is ON
        lightSwitch.operate(false); // Light Bulb is OFF
    }
}
```

In this example, the `Switch` class is tightly coupled with the `LightBulb` class. If we want to use a different type of light source, we will need to modify the `Switch` class.

With Dependency Inversion

To apply the Dependency Inversion Principle, we can introduce an `ILightSource` interface:

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

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

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

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

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

class Switch {
    private ILightSource lightSource;

    public Switch(ILightSource lightSource) {
        this.lightSource = lightSource;
    }

    public void operate(boolean turnOn) {
        if(turnOn) {
            lightSource.turnOn();
        } else {
            lightSource.turnOff();
        }
    }
}

// Usage
public class Main {
    public static void main(String[] args) {
```

```
        ILightSource bulb = new LightBulb();
        Switch lightSwitch = new Switch(bulb);
        lightSwitch.operate(true); // Light Bulb is ON
        lightSwitch.operate(false); // Light Bulb is OFF

        ILightSource led = new LED();
        Switch ledSwitch = new Switch(led);
        ledSwitch.operate(true); // LED is ON
        ledSwitch.operate(false); // LED is OFF
    }
}
```

In this revised example, the `Switch` class no longer depends on a concrete implementation of `LightBulb` but rather on the `ILightSource` interface. This means we can easily introduce new types of light sources without modifying the existing `Switch` class.

Interview Questions on Dependency Inversion Principle

Q1: What is the Dependency Inversion Principle?

Answer: The Dependency Inversion Principle (DIP) is a design principle that emphasizes the importance of depending on abstractions rather than concrete implementations. It states that both high-level and low-level modules should depend on abstractions, and not the other way around. This enhances code flexibility and maintainability.

Q2: Why is the Dependency Inversion Principle important?

Answer: DIP is crucial because it helps reduce the coupling between different parts of a software system. By relying on abstractions, changes in low-level modules (concrete implementations) do not necessitate changes in high-level modules (business logic), making the system more modular, testable, and easier to maintain.

Q3: Can you give an example of Dependence Injection, which relates to DIP?

Answer: Certainly! Dependency Injection is a design pattern that implements the Dependency Inversion Principle by providing dependencies externally rather than allowing a class to instantiate them itself. For example, instead of having a `Controller` class create instances of a `Service` class, we can pass a `Service` implementation to the `Controller` via its constructor (or setter), decoupling the two:

```
class Controller {
    private Service service;

    public Controller(Service service) {
        this.service = service;
    }

    public void execute() {
        service.performAction();
    }
}
```

Now, the `Controller` does not depend on a concrete `Service` implementation, thus adhering to DIP.

Q4: How do you test a class that follows the Dependency Inversion Principle?

Answer: Testing a class that adheres to the Dependency Inversion Principle is straightforward. You can create mock or stub implementations of the abstractions (interfaces) it relies on. This way, you can test the high-level module independently from low-level modules, ensuring that its behavior is correct without any side effects from concrete implementations.

Q5: What are some common pitfalls of the Dependency Inversion Principle?

Answer: Some common pitfalls include:

- Over-abstraction: Creating too many interfaces can lead to unnecessary complexity and make the code harder to understand.
- Ignoring the principle: Sometimes, developers may underestimate the importance of dependencies and resort to tightly coupled designs for the sake of simplicity.
- Poor interface design: If interfaces are not designed well, they can lead to confusion and inconsistent implementations.

By understanding and applying the Dependency Inversion Principle effectively, software developers can create systems that are both robust and easy to scale or modify as requirements evolve.

The SOLID principles are a set of five design principles intended to make object-oriented designs more understandable, flexible, and maintainable. They were introduced by Robert C. Martin and are widely

adopted in the software development industry. Let's break down each principle and see how we can implement them in Java code.

Single Responsibility Principle (SRP)

The Single Responsibility Principle states that a class should have only one reason to change, meaning it should only have one job. This makes the code easier to understand and manage.

Example:

```
class Report {
    public void generateReport() {
        // logic for generating report
    }
}

class ReportPrinter {
    public void printReport(Report report) {
        // logic for printing report
    }
}
```

In this example, the `Report` class is responsible only for generating reports, while `ReportPrinter` takes care of printing. This separation of concerns makes the code more manageable and easier to test.

Open/Closed Principle (OCP)

The Open/Closed Principle suggests that classes should be open for extension but closed for modification. This means that we should be able to add new functionality through inheritance or interfaces without changing the existing code.

Example:

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

class Circle implements Shape {
    private double radius;
```

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

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

class Rectangle implements Shape {
    private double width, height;

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

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

With the `Shape` interface, we can now create additional shape classes (e.g., `Triangle`) without modifying existing code, adhering to the OCP.

Liskov Substitution Principle (LSP)

The Liskov Substitution Principle asserts that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. This ensures that a subclass can stand in for its parent class.

Example:

```
class Bird {
    public void fly() {
        // logic for flying
    }
}
```

```
class Sparrow extends Bird {
    @Override
    public void fly() {
        // logic specific to sparrow flying
    }
}

class Ostrich extends Bird {
    @Override
    public void fly() {
        // Ostrich doesn't fly: this violates LSP
        throw new UnsupportedOperationException("Ostrich cannot fly");
    }
}
```

In this case, substituting `Ostrich` where `Bird` is expected breaks the contract of the `Bird` class. To adhere to LSP, we might reconsider our design. One solution could be defining a separate interface for flying birds.

Interface Segregation Principle (ISP)

The Interface Segregation Principle states that a client should not be forced to depend on interfaces it does not use. This encourages the design of smaller, more focused interfaces.

Example:

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

interface Eatable {
    void eat();
}

class Human implements Workable, Eatable {
    public void work() {
        // human work logic
    }

    public void eat() {
```

```
        // human eat logic
    }
}

class Robot implements Workable {
    public void work() {
        // robot work logic
    }
}

// Robot does not implement Eatable
```

In this case, `Workable` and `Eatable` are separate interfaces, allowing each implementing class to only depend on the methods relevant to it.

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle emphasizes the need to depend on abstractions rather than concrete implementations. This helps to reduce the coupling between different parts of the application.

Example:

```
interface PaymentProcessor {
    void processPayment(double amount);
}

class PayPalProcessor implements PaymentProcessor {
    public void processPayment(double amount) {
        // PayPal payment logic
    }
}

class ShoppingCart {
    private PaymentProcessor paymentProcessor;

    public ShoppingCart(PaymentProcessor paymentProcessor) {
        this.paymentProcessor = paymentProcessor;
    }

    public void checkout(double amount) {
        paymentProcessor.processPayment(amount);
```

```
        }
    }
```

In this example, `ShoppingCart` depends on the abstraction `PaymentProcessor`, not on any concrete class like `PayPalProcessor`. This makes it easy to switch payment processors without changing `ShoppingCart`.

By utilizing the SOLID principles in our Java applications, we can create a codebase that is easier to understand, maintain, and extend over time. These principles not only help in building efficient applications but also enhance collaboration among developers, making it a cornerstone of modern software development practices.