Quest Journals
Journal of Software Engineering and Simulation
Volume 10 ~ Issue 3 (March 2024) pp: 54-68
ISSN(Online) :2321-3795 ISSN (Print):2321-3809
www.questjournals.org

Research Paper

The Perennial Importance of SOLID Principles in Software Design

Arun Neelan
Independent Researcher
PA, USA

Abstract— *The SOLID principles—Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, and Dependency Inversion—are fundamental concepts in object-oriented design that promote maintainable, flexible, and scalable code. This paper provides an in-depth examination of each principle, discussing how they can be applied to enhance software design. It also highlights the common challenges developers face when implementing these principles and offers practical tips for overcoming them. Ultimately, the goal of this review is to help developers understand the real-world impact of the SOLID principles and provide actionable advice for incorporating them into their projects.*

Keywords—*SOLID Principles, Object-Oriented Design, Software Design Principles, Single Responsibility Principle (SRP), Open/Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), Dependency Inversion Principle (DIP)*

I. INTRODUCTION

In software development, creating code that is maintainable, scalable, and flexible is essential. Over time, various design principles have been developed to help developers build software that not only meets customer requirements but is also easy to understand, extend, and modify. Among these, the SOLID principles are some of the most widely recognized, offering a solid foundation for high-quality object-oriented design.

Introduced by Robert C. Martin, the SOLID principles consist of five key concepts: the Single Responsibility Principle (SRP), Open/Closed Principle (OCP), Liskov Substitution Principle (LSP), Interface Segregation Principle (ISP), and Dependency Inversion Principle (DIP). [1] These principles provide a structured approach to software development, making code easier to maintain, extend, and test -- qualities that are especially important in today's fast-paced development environment. A deeper understanding of these principles, along with awareness of the challenges and best practices, helps development teams create software that is both maintainable and flexible.

II. SINGLE RESPONSIBILITY PRINCIPLE (SRP)

The Single Responsibility Principle (SRP) states that a class should have only one reason to change, meaning that a class should only have one responsibility or job. [1] If a class has more than one responsibility, those responsibilities become coupled, and changes in one area could lead to unintended consequences in other areas, making the code harder to maintain and modify.

Before delving into its technical application in software development, let's first explore this concept through a simple analogy in the context of a car manufacturing plant. We will first look at a scenario where the SRP is violated and then see how applying SRP improves the system.

Imagine a car manufacturing plant where several workers are assigned specific tasks, such as installing the engine, painting the car body, testing the brakes, and assembling the interior, with each working independently. If one person were responsible for assembling the entire car from start to finish, the process would become chaotic and inefficient. The person would be overwhelmed with different tasks, and if any changes were needed, it would disrupt the entire assembly process.

By focusing on a single task, each worker can perform their job more efficiently, ensuring that the overall car manufacturing process runs smoothly and results in a high-quality product. Similarly, in software design, SRP suggests that each component or module should handle only one responsibility. This makes the system easier to manage, maintain, and improve, just like how the specialized workers in the plant contribute to a more efficient process.

A. Violating the SRP: One class attempting to handle all the tasks.

Now, let's apply this analogy to a software development scenario. Consider a class responsible for both handling car assembly and inspecting the car. This is like having a single worker trying to perform all tasks in the manufacturing plant.

```
class CarFactory {
    private String carModel;

    public CarFactory(String carModel) {
        this.carModel = carModel;
    }

    // Method for assembling all of the car parts.
    public void assembleCar() {
        System.out.println("Assembling car: " + carModel);
        // Code to assemble engine, wheels, body, etc.
    }

    // Method for inspecting the assembled car.
    public void inspectCar() {
        System.out.println("Inspecting car: " + carModel);
        // Code to check brakes, safety features, etc.
    }
}
```

Fig. 1 Violating the SRP

In this example, the CarFactory class is responsible for both assembling the car and inspecting it. Just like the worker handling everything in the manufacturing process, this class is trying to manage multiple responsibilities. If the inspection process changes, or if the assembly process needs an update, this class would have to be modified in multiple places, which leads to complexity and inefficiency.

B. Applying the SRP: Segregate responsibilities across classes.

Update the design so the responsibilities are moved to appropriate classes rather than having all in one class: CarAssembler and CarInspector. The CarAssembler class is solely responsible for assembling the car, while the CarInspector class handles the inspection process.

```
// Class responsible for assembling the car.
class CarAssembler {
    private String carModel;

    public CarAssembler(String carModel) {
        this.carModel = carModel;
    }

    public void assembleCar() {
        System.out.println("Assembling car: " + carModel);
        // Code to assemble engine, wheels, body, etc.
    }
}


// Class responsible for inspecting the assembled car.
class CarInspector {
    private String carModel;

    public CarInspector(String carModel) {
        this.carModel = carModel;
    }

    public void inspectCar() {
        System.out.println("Inspecting car: " + carModel);
        // Code to check brakes, safety features, etc.
    }
}
```

Fig. 2 Applying the SRP

By adhering to the SRP, each class now has a single reason to change. If the assembly process needs to be modified, only the CarAssembler class will need to be updated. Likewise, if the inspection process changes, the CarInspector class is the only one affected. This separation simplifies maintenance, reduces complexity, and enhances flexibility, making the system easier to manage.

C. Challenges and Best Practices

Topic	***Challenge***	***Best Practice***
Determining the Right granularity of classes	Over-splitting responsibilities can create too many small classes, making the system hard to maintain and navigate. On the other hand, under-splitting can lead to large, complex classes that take on multiple responsibilities, which violates the SRP. [9]	Start with broader classes and refactor them as the system grows. Break responsibilities into smaller classes only when their purpose becomes clearer. Focus on grouping related behaviors together, while avoiding unnecessary fragmentation.
Managing Dependencies between classes	While SRP promotes separating concerns, it can lead to more dependencies between small, specialized classes. If these dependencies aren't carefully managed, it can cause tight coupling, making the system harder to maintain and extend.	Dependency injection (DI) can be used to decouple classes by injecting dependencies instead of hardcoding them, making classes more flexible and easier to test. [10] Additionally, design patterns like Facade or Mediator can help manage complex dependencies while preserving modularity.
Performance Overhead	Strict adherence to SRP in performance-critical systems can result in unnecessary method calls, object creations, or communication overhead between small classes. In distributed systems, such as microservices, SRP may introduce network latency due to frequent communication between small services. [9]	Regular performance monitoring is essential to ensure that SRP does not create bottlenecks. If fragmentation causes performance issues, consolidating related responsibilities or optimizing critical paths involving heavy communication between classes or services may be necessary. [4]
Applying to Legacy code	Refactoring a legacy system to adhere to SRP can be challenging, particularly in systems with tight coupling and large monolithic classes. This process is often time-consuming and carries risks, such as introducing regression bugs or breaking existing functionality.	SRP should be applied incrementally in legacy code, starting with small, isolated portions that have clear, defined responsibilities. It's important to ensure comprehensive unit tests are in place to verify the refactored code and maintain backward compatibility throughout the process. [4]
Expertise and Brainstorming	Less experienced developers may misapply SRP by over-splitting classes, leading to over-engineering. This can introduce unnecessary complexity without clear benefits, making the system harder to maintain.	SRP should be applied with a clear understanding of the business domain, focusing on meaningful responsibilities rather than just technical reasons. Test Driven Development (TDD) ensures classes remain focused and testable. Brainstorming and iterative approaches help refine the design, aligning it with both business needs and technical goals.

Table 1. Challenges and Best Practices for SRP

III. OPEN/CLOSED PRINCIPLE (OCP)

The Open/Closed Principle (OCP) states that software entities—such as classes, modules, and functions—should be open for extension but closed for modification. [2]

Let's explore this concept in more depth with an example. We will first look at a scenario where the OCP is violated and then see how applying OCP improves the system.

Imagine a software system where logging is implemented in multiple ways: one part of the system logs to the console, other writes to a file, and a third logs to a database. If these logging methods are tightly coupled to the core logic, introducing a new logging strategy—such as logging to a cloud service—would require significant changes across the system. This tight coupling increases complexity, introduces potential bugs, and complicates future maintenance.

By adhering to the OCP, each logging strategy can be encapsulated in independent components that implement a common interface. This design ensures the system is "open for extension but closed for modification." When a new logging method is needed, it can be added as a new strategy without modifying the existing codebase. This approach enhances flexibility, simplifies maintenance, and allows the system to scale and evolve in a modular, efficient manner.

A. Violating the OCP - One class attempting to handle all the implementation.

The design, where all logging implementations (such as logging to the console, file, database, or cloud service) are handled within a single class, violates the OCP. This leads to modifying the same class whenever additional behaviors need to be added.

```java
public class Logger {
    public void log(String message, String logType) {
        if (logType.equals("console")) {
            System.out.println("Logging to console: " + message);
        } else if (logType.equals("file")) {
            // Code to write to a file.
            System.out.println("Logging to file: " + message);
        } else if (logType.equals("database")) {
            // Code to write to database tables.
            System.out.println("Logging to database: " + message);
        } else {
            throw new IllegalArgumentException("Unknown log type.");
        }
    }
}
```

Fig. 3 Violating the OCP

B. Applying the OCP

Update the design by introducing interfaces and separate implementation classes, each handling a specific responsibility. For example, a Logger interface is used, with implementations like ConsoleLogger, FileLogger, and CloudLogger.

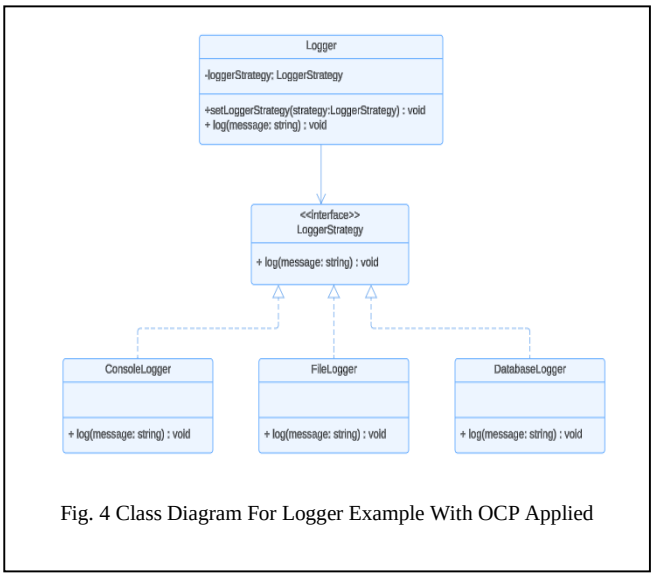

Fig. 4 Class Diagram For Logger Example With OCP Applied

```java
public interface LoggerStrategy {
    void log(String message);
}
```

Fig. 5 LoggerStrategy Interface

```java
public class ConsoleLogger implements LoggerStrategy {
    @Override
    public void log(String message) {
        System.out.println("Logging to console: " + message);
    }
}

public class FileLogger implements LoggerStrategy {
    @Override
    public void log(String message) {
        // Code to write to a file.
        System.out.println("Logging to file: " + message);
    }
}

public class DatabaseLogger implements LoggerStrategy {
    @Override
    public void log(String message) {
        // Code to write to database tables.
        System.out.println("Logging to database: " + message);
    }
}
```

Fig. 6 LoggerStrategy Impl Strategies

```java
public class Logger {
    private LoggerStrategy loggerStrategy;

    public Logger(LoggerStrategy loggerStrategy) {
        this.loggerStrategy = loggerStrategy;
    }

    public void log(String message) {
        loggerStrategy.log(message);
    }
}
```

Fig. 7 Logger class Refactored

This approach allows new functionality to be added without modifying existing code, enhancing encapsulation and maintainability. The system is open for extension but closed for modification, reducing bug risks and increasing reusability and modularity. Additionally, interfaces simplify unit testing and mocking by enabling isolated testing of implementations.

C. Challenges and Best Practices

Topic	***Challenge***	***Best Practice***
Managing Complexity	The OCP can add unnecessary complexity when developers introduce abstractions, like interfaces or abstract classes, to plan for future extensions. Over-engineering—where developers build for potential needs that may never happen—can make the situation worse, leading to a codebase that's bloated and overly complicated. [5]	To avoid unnecessary complexity, keep the design simple and focused on the current requirements. Only introduce abstractions when they offer clear value, and steer clear of over-engineering. Follow the YAGNI (You Aren't Gonna Need It) principle—design for today's needs and refactor as new requirements emerge. [3]
Anticipating Future Changes	Predicting how software will evolve is difficult, and designing for speculative changes can result in solutions that are either too flexible or too rigid. Over-anticipating future requirements can cause the design to deviate from actual needs, leading to wasted effort and poor decisions.	Iterate and refactor as new requirements emerge. Design for the present while focusing on realistic future changes rather than hypothetical ones. Instead of anticipating every possibility, allow the system to evolve organically through future iterations.
Refactoring code to apply OCP	Refactoring code to comply with the OCP can lead to integration issues, particularly when working with legacy systems. Significant changes made to enable extensibility can disrupt existing functionality, resulting in new bugs.	Incremental changes should be made to refactor the code, ensuring small, manageable improvements over time. The code should be modularized with clear interfaces, enabling easier extensions without disrupting existing functionality. Extensive testing

		is essential to identify and resolve any integration issues.
Difficulty in Identifying Extension Points	Identifying where and how to extend a system without modifying existing code can be challenging. If extension points are not clearly defined, adding new features may unintentionally alter core behaviors.	Design clear extension points with well-defined interfaces or abstract classes. Use event-driven or plugin-based architectures, and document extension points to enable easy feature additions without disrupting existing functionality.

Table 2. Challenges and Best Practices for OCP

IV. L - LISKOV SUBSTITUTION PRINCIPLE (LSP)

The Liskov Substitution Principle (LSP) states that objects of a superclass should be replaceable with objects of its subclasses without altering the correctness of the program. [2]

Let's explore this concept in more depth with an example. We will first look at a scenario where the LSP is violated and then see how applying LSP improves the system.

Imagine a software system for managing birds, the core functionality assumes all birds can perform actions like flying, eating, and singing. However, this assumption breaks down when specialized bird types, like Eagles and Penguins, are introduced. Since Penguins can't fly, calling a fly() method on a Penguin would violate the LSP, causing errors or requiring cumbersome checks in the code.

To align with LSP, the design is adjusted by introducing subclasses such as FlyingBird and NonFlyingBird, both inheriting from a common Bird class. This ensures that the fly() method is only called on flying birds, maintaining the correctness of the program and allowing subclasses to be substituted for their superclass without errors or unexpected behavior.

This approach adheres to LSP by ensuring that the behavior of subclasses remains consistent with the expectations set by the superclass, avoiding the need for complex checks or invalid method calls.

A. Violating LSP – Certain classes implementing more than its behavior.

In this case, substituting a NonFlyingBird for a FlyingBird breaks the system, as it unexpectedly throws an exception. This tight coupling between the base class and the subclasses violates the Liskov Substitution Principle, making the code brittle and difficult to maintain.

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


// FlyingBird subclass.
class FlyingBird extends Bird {
    @Override
    public void fly() {
        System.out.println("Flying high in the sky!");
    }
}


// NonFlyingBird subclass.
class NonFlyingBird extends Bird {
    // NonFlyingBird cannot fly - Violates LSP.
    @Override
    public void fly() {
        throw new UnsupportedOperationException("This bird cannot fly!");
    }
}


public class Main {
    public static void main(String[] args) {
        Bird bird = new FlyingBird();
        bird.fly();  // This works fine.

        bird = new NonFlyingBird();
        bird.fly();  // This will throw an exception - Violates LSP.
    }
}
```

Fig. 8 Bird Example with LSP Violation

B. Applying LSP

To address this violation and ensure system flexibility, subclasses should be designed to be interchangeable with their parent class without altering expected behavior. In this case, the behavior of flying birds should be separated from non-flying birds. A solution is to introduce a Flyable interface for birds that can fly, while non-flying birds don't implement it. This design allows Bird objects to be substituted with specific subclasses without violating LSP. Here's how the design and code can be refactored:

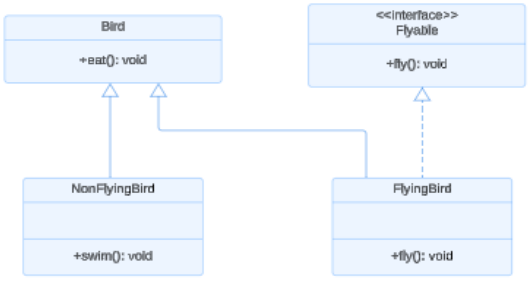

Fig. 9 Class Diagram for Bird Example with LSP Applied

```
// Interface for Birds that can fly.
interface Flyable {
    void fly();
}

// General Bird class.
class Bird {
    public void eat() {
        System.out.println("Eating...");
    }
}

// NonFlyingBird does not implement Flyable.
class NonFlyingBird extends Bird {
    public void swim() {
        System.out.println("Swimming...");
    }
}

// FlyingBird implements Flyable.
class FlyingBird extends Bird implements Flyable {
    @Override
    public void fly() {
        System.out.println("Flying high in the sky!");
    }
}

public class Main {
    public static void main(String[] args) {
        // Creating instances of flying and non-flying birds.
        Bird bird = new FlyingBird();
        bird.eat();
        ((Flyable) bird).fly();  // Only call fly on Flyable birds.

        bird = new NonFlyingBird();
        bird.eat();
        // No fly method on non-flying birds.
    }
}
```

Fig. 10 Bird Example with LSP Applied

By separating bird behavior into distinct categories, the system becomes more flexible, allowing different bird types to be substituted without errors. FlyingBirds and NonFlyingBirds can be used interchangeably without affecting stability. This approach reduces complexity, as non-flying birds do not implement unnecessary methods, making the code cleaner. Scalability improves, with new bird types added by implementing the Flyable interface. This design also simplifies testing and maintenance, as each bird type can be tested independently, leading to more reliable code.

C. Challenges and Best Practices

Topic	***Challenge***	***Best Practice***
Maintaining Consistent Behavior	Ensuring that subclasses preserve the expected behavior of the base class, particularly when overriding methods. This includes not altering inherited methods in a way that violates the established contract.	When overriding methods, subclasses must preserve the behavior of the parent class without changing expected outputs. The same preconditions and postconditions defined by the parent class must also be adhered to.
Maintaining Contract	Subclasses may alter or violate the invariants, preconditions, or postconditions defined by the base class, breaking the expected contract and leading to errors.	Subclasses should avoid introducing more restrictive preconditions, should maintain or strengthen postconditions, and must adhere to the invariants established by the parent class.
Excessive Inheritance & Hierarchy Complexity	Excessive or inappropriate use of inheritance, along with complex inheritance hierarchies, can make it difficult to maintain LSP and ensure proper substitution between classes.	Composition should be preferred over inheritance when possible. Inheritance hierarchies should be simplified, and relationships between base and derived classes should be clearly defined. [7]

Maintaining Consistent Interfaces	Subclasses may expose different interfaces or behaviors compared to the parent class, causing incompatibility.	Subclasses must adhere to the same interface as the parent class or implement interfaces that extend the contract defined by the base class.

Table 3. Challenges and Best Practices for LSP

V. I - INTERFACE SEGREGATION PRINCIPLE (ISP)

Interface Segregation Principle (ISP) states that no client should be forced to depend on methods it does not use. It promotes the design of small, client-specific interfaces, ensuring that classes are not burdened with unnecessary functionality. This leads to systems that are more flexible, maintainable, and easier to understand by keeping interfaces focused on the needs of the implementing class.

Let's explore this concept in more depth with an example. We will first look at a scenario where the ISP is violated and then see how applying ISP improves the system.

Imagine a printing company with different types of printers. Some printers only print, while others can print, scan, and fax. If all printers shared the same interface that included methods for scanning and faxing, even the basic printer would be forced to implement those extra methods, making it unnecessarily complex.

Now, imagine splitting those tasks into separate, more specific interfaces. The basic printer only needs the Printer interface for printing, while the multifunction printer can implement additional interfaces for scanning and faxing. This way, each printer is only responsible for the functionality it needs, adhering to ISP by avoiding unnecessary complexity.

A. Violating ISP – Larger interfaces complicating implementation classes to implementing unsupported behaviors

```
// Large interface with all printer features.
interface Printer {
    void print(String document);
    void scan(String document);
    void sendFax(String document);
}


// Basic Printer: Implements all methods, even though it can't scan or fax.
class BasicPrinter implements Printer {
    @Override
    public void print(String document) {
        System.out.println("Printing document: " + document);
    }

    @Override
    public void scan(String document) {
        throw new UnsupportedOperationException("Basic printer cannot scan.");
    }

    @Override
    public void sendFax(String document) {
        throw new UnsupportedOperationException("Basic printer cannot send fax.");
    }
}


// Advanced Printer: Implements all features (print, scan, fax).
class AdvancedPrinter implements Printer {
    @Override
    public void print(String document) {
        System.out.println("Printing document: " + document);
    }

    @Override
    public void scan(String document) {
        System.out.println("Scanning document: " + document);
    }

    @Override
    public void sendFax(String document) {
        System.out.println("Sending fax of document: " + document);
    }
}
```

Fig. 11 Printer Example with ISP Violated

```
public class Main {
    public static void main(String[] args) {
        // Creating a BasicPrinter (only prints)
        Printer basicPrinter = new BasicPrinter();
        basicPrinter.print("Printing from Basic Printer.");

        try {
            basicPrinter.scan("Scanning from Basic Printer.");  // This will throw an exception.
        } catch (UnsupportedOperationException e) {
            System.out.println(e.getMessage());
        }

        try {
            basicPrinter.sendFax("Faxing from Basic Printer.");  // This will throw an exception.
        } catch (UnsupportedOperationException e) {
            System.out.println(e.getMessage());
        }

        // Creating an AdvancedPrinter (supports printing, scanning, and faxing)
        Printer advancedPrinter = new AdvancedPrinter();
        advancedPrinter.print("Printing from Advanced Printer.");
        advancedPrinter.scan("Scanning from Advanced Printer.");
        advancedPrinter.sendFax("Faxing from Advanced Printer.");
    }
}
```

Fig. 12 Printer Example with ISP Violated (Cont..)

The Printer interface includes methods for all functions (print(), scan(), sendFax()), but the BasicPrinter class, which only supports printing, is forced to implement unnecessary methods. This can lead to exceptions or unwanted behavior. While the AdvancedPrinter class works fine, bundling all methods into one interface violates the Interface Segregation Principle.

B. Applying ISP

A solution is to introduce separate interfaces such as Print, Scan, and Fax, allowing each printer to implement only the methods it requires. This design adheres to the Interface Segregation Principle, promoting simplicity and focus to the system.

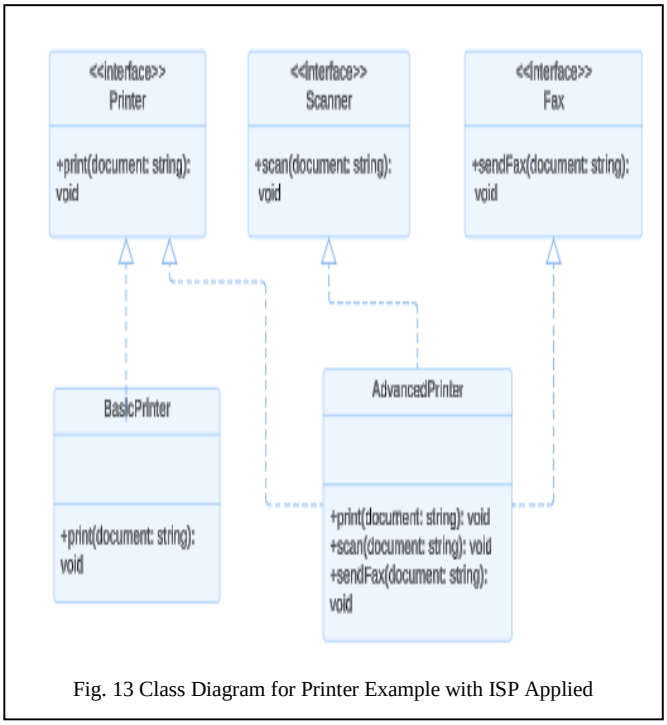

Fig. 13 Class Diagram for Printer Example with ISP Applied

```java
// Interface for printing.
interface Printer {
    void print(String document);
}

// Interface for scanning.
interface Scanner {
    void scan(String document);
}

// Interface for faxing.
interface Fax {
    void sendFax(String document);
}
```

Fig. 14 Printer Example with ISP Applied (Separate Interfaces for Each Feature)

```java
// Basic Printer: Only implements printing functionality.
class BasicPrinter implements Printer {
    @Override
    public void print(String document) {
        System.out.println("Printing document: " + document);
    }
}


//Advanced Printer: Implements all features (print, scan, fax).
class AdvancedPrinter implements Printer, Scanner, Fax {
@Override
public void print(String document) {
    System.out.println("Printing document: " + document);
}

@Override
public void scan(String document) {
    System.out.println("Scanning document: " + document);
}

@Override
public void sendFax(String document) {
    System.out.println("Sending fax of document: " + document);
}
}
```

Fig. 15 Printer Example with ISP Applied (Separate Impl Based on Its Behavior)

```java
//Main Class after ISP.
public class Main {
    public static void main(String[] args) {
        // Creating a BasicPrinter (only prints)
        Printer basicPrinter = new BasicPrinter();
        basicPrinter.print("Printing from Basic Printer.");

        // Creating an AdvancedPrinter (supports printing, scanning, and faxing)
        Printer advancedPrinter = new AdvancedPrinter();
        advancedPrinter.print("Printing from Basic Printer.");
        ((Scanner) advancedPrinter).scan("Scanning from Advanced Printer.");
        ((Fax) advancedPrinter).sendFax("Faxing from Advanced Printer.");
    }
}
```

Fig. 16 Printer Example with ISP Applied (Usage)

After implementing ISP, the BasicPrinter implements only the Printer interface, as it supports only printing, while the AdvancedPrinter implements the Printer, Scanner, and Fax interfaces to support all three functionalities. This design adheres to the Interface Segregation Principle (ISP) by ensuring that each class only implements the methods it actually needs, keeping the system simple and focused.

C. Challenges and Best Practices

Topic	***Challenge***	***Best Practice***
Excessive Interfaces & Implementation Complexity	Over-segregating interfaces can result in an excessive number of small interfaces, which increases complexity. Implementing numerous small interfaces can also complicate implementations, necessitating explicit casting or interface checks. [7]	Striking a balance when creating interfaces is essential. Interfaces should only be split when there is a clear distinction in functionality. Composition or delegation can help maintain clean implementations and avoid excessive inheritance or interface implementation.
Code Duplication	Multiple classes sharing common functionality may lead to code duplication if each class implements separate interfaces.	Composition can be used to delegate shared functionality to reusable components, reducing duplication. Abstract classes or default methods can also help minimize repeated code.
Managing With Changing Requirements	Narrowly defined interfaces may not be flexible enough to adapt to new or changing requirements, requiring frequent refactoring. [6]	Interfaces should be designed to be extensible using inheritance or abstract classes, allowing easy extensions without constant refactoring.
Managing Identifying Boundaries	It can be challenging to identify clear boundaries between different interfaces, leading to confusion. [2]	Collaboration with stakeholders and domain experts can help align interfaces with real-world use cases, ensuring that each interface focuses on a cohesive set of actions.
System Complexity	Applying ISP to simple systems can lead to unnecessary complexity and over-engineering. [7]	ISP should only be applied when necessary. In simpler systems, keeping the design straightforward and avoiding premature complexity is advisable.

Table 4. Challenges and Best Practices for ISP

VI. D - DEPENDENCY INVERSION PRINCIPLE (DIP)

The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules. Both should depend on abstractions, such as interfaces or abstract classes. Additionally, abstractions should not depend on implementation details; rather, details should depend on abstractions. [2]

Let's explore this concept in more depth with an example. We will first look at a scenario where the DIP is violated and then see how applying DIP improves the system.

Consider a trading platform where the order processing system is tightly coupled to specific order types, such as CashOrder or QuantityOrder. As a result, when a new order type like LimitOrder is introduced, the system must be modified to accommodate the new type. This leads to code changes across multiple areas, making the system harder to maintain, extend, and scale.

By applying DIP, we introduce an abstraction—an Order interface. Instead of the order processing system depending directly on specific order types, it now depends on this interface. As a result, we can add new order types without changing the core order processing logic. This makes the system more flexible, scalable, and easier to maintain.

A brief note on the types of orders to better understand the examples:

CashOrder: In a CashOrder, the system determines the number of shares to purchase based on the cash amount provided, ensuring that the total cost does not exceed the specified amount.

QuantityOrder: In a QuantityOrder, the system buys a specified number of shares at the current market price, without considering the total cost.

LimitOrder: A LimitOrder specifies the price at which shares should be bought or sold. The order is only executed if the market price is equal to or better than the specified limit price.

A. Violating DIP – Implementation Dependent

```
// Low-level module: CashOrder.
class CashOrder {
    private double cashAmount;

    public CashOrder(double cashAmount) {
        this.cashAmount = cashAmount;
    }

    public void placeOrder() {
        double pricePerUnit = 100.0;
        int unitsToBuy = (int) (cashAmount / pricePerUnit);
        System.out.println("Placing cash order for " + unitsToBuy +
                " units with " + cashAmount + " amount.");
    }
}


// Another Low-level module: QuantityOrder.
class QuantityOrder {
    private int quantity;

    public QuantityOrder(int quantity) {
        this.quantity = quantity;
    }

    public void placeOrder() {
        double pricePerUnit = 100.0;
        double totalAmount = quantity * pricePerUnit;
        System.out.println("Placing quantity order for " + quantity +
                " units costing " + totalAmount);
    }
}


// High-level module: OrderService directly depends on CashOrder.
class OrderService {
    private CashOrder cashOrder;
    private QuantityOrder quantityOrder;

    public OrderService(double cashAmount, int quantityAmount) {
        // Direct dependency on CashOrder.
        this.cashOrder = new CashOrder(cashAmount);

        //Direct dependency on QuantityOrder.
        this.quantityOrder = new QuantityOrder(quantityAmount);
    }

    public void processOrder(String type) {
        if ("cash".equalsIgnoreCase(type)) {
            cashOrder.placeOrder();
        } else if ("quantity".equalsIgnoreCase(type)) {
            quantityOrder.placeOrder();
        } else {
            throw new UnsupportedOperationException("Invalid type:" + type);
        }
    }
}
```

Fig. 17 Printer Example with DIP Violated

In a system, the order processing module directly depends on specific order types, such as CashOrder and QuantityOrder. This tight coupling results in the need to update the order processing logic in multiple places whenever a new order type, like LimitOrder, is introduced. This violates the DIP, as the high-level order processing module depends on low-level order types. The absence of abstraction makes the system harder to maintain and extend.

B. Applying DIP

Introduce abstraction to the design, Order, and let the CashOrder and QuantityOrder implement it, so OrderService relies on the abstraction rather than implementation and become flexible to support future ones as well.

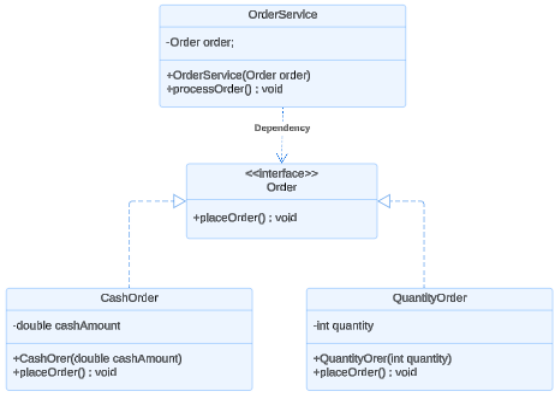

Fig. 18 Class Diagram for Order System With DIP Applied

```java
// High-level module: Order interface (abstraction).
interface Order {
    void placeOrder();
}


// Low-level module: CashOrder implements the Order interface.
class CashOrder implements Order {
    private double cashAmount;

    public CashOrder(double cashAmount) {
        this.cashAmount = cashAmount;
    }

    @Override
    public void placeOrder() {
        double pricePerUnit = 100.0;
        int unitsToBuy = (int)(cashAmount / pricePerUnit);
        System.out.println("Placing cash order for " + unitsToBuy +
                " units with " + cashAmount + " amount.");
    }
}


// Another low-level module: QuantityOrder implements the Order interface.
class QuantityOrder implements Order {
    private int quantity;

    public QuantityOrder(int quantity) {
        this.quantity = quantity;
    }

    @Override
    public void placeOrder() {
        double pricePerUnit = 100.0;
        double totalAmount = quantity * pricePerUnit;
        System.out.println("Placing quantity order for " +
        quantity + " units costing " + totalAmount);
    }
}
```

Fig. 19 Order System With DIP Applied

```
// High-level module: OrderService depends on abstraction (Order interface).
class OrderService {
    private Order order;

    // Dependency Injection via constructor.
    public OrderService(Order order) {
        this.order = order;  // Depend on abstraction, not concrete class.
    }

    public void processOrder() {
        order.placeOrder();
    }
}


public class Main {
    public static void main(String[] args) {
        // Buy orders worth $5000.
        Order cashOrder = new CashOrder(5000.0);
        OrderService orderService = new OrderService(cashOrder);
        orderService.processOrder();

      // Buy 50 units of shares.
        Order quantityOrder = new QuantityOrder(50);
        orderService = new OrderService(quantityOrder);
        orderService.processOrder();
    }
}
```

Fig. 20 Order System With DIP Applied (Cont)

By applying the DIP, the OrderService now relies on the Order interface rather than specific order types like CashOrder and QuantityOrder. This makes it easier to add new order types, such as LimitOrder or MarketOrder, without changing the OrderService. With Dependency Injection, different order types can be easily injected, boosting flexibility and making the system simpler to maintain and scale.

C. Challenges and Best Practices:

Topic	***Challenge***	***Best Practice***
Increased Complexity & Over-Abstraction	Adhering to the DIP can introduce additional abstractions, such as interfaces and abstract classes, increasing system complexity. While it improves decoupling, there is a risk of unnecessary abstractions that may make the system harder to understand. [8]	Abstractions should be introduced gradually, offering clear value such as flexibility, testability, or reusability. Over-abstraction should be avoided; abstractions should be applied only when they enhance flexibility or maintainability. The design should remain lean, creating abstractions only when necessary.
Difficulty in Deciding Abstractions	Choosing the right level of abstraction can be challenging, as determining which concrete classes should depend on which abstractions is highly context-dependent.	Focus on abstracting key components that are prone to change or require flexibility. Dependency Injection can be used to manage dependencies and simplify the design. Dependencies should be carefully assessed to determine which ones truly require abstraction.
Performance Overhead	Dependency Injection and other implementations of DIP can introduce slight performance overhead due to additional layers of indirection or dynamic resolution of dependencies.	In most applications, the performance cost of DIP is negligible. For performance-critical applications, lazy loading or selective injection strategies can help mitigate overhead while preserving the benefits of DIP.
Testing Difficulties	Systems heavily relying on Dependency Injection can require complex setup in testing environments, potentially making unit testing more difficult.	Lightweight Dependency Injection libraries that are easy to mock during tests should be used. Constructor injection should be preferred over setter injection to ensure dependencies are provided upfront, making testing easier.

Table 5. Challenges and Best Practices for DIP

VII. CONCLUSION

In conclusion, this paper has explored the five SOLID principles—Single Responsibility (SRP), Open-Closed (OCP), Liskov Substitution (LSP), Interface Segregation (ISP), and Dependency Inversion (DIP)—which provide valuable guidelines for building modular, maintainable, and flexible software. When applied correctly, these principles help ensure that software remains scalable, adaptable, and easy to maintain, leading to more efficient and robust designs. It's important to keep the challenges and best practices discussed for each principle in mind during development to avoid common pitfalls.

While each principle stands strong on its own, applying them in isolation, especially in complex projects, can lead to unnecessary complications. For the best results, these principles should work together as part of a

cohesive strategy. Ongoing learning, collaboration, and expert insights are essential for refining their application, and regular reviews help ensure they continue to meet the evolving needs of software development.

REFERENCES

[1] "ArticleS.UncleBob.PrinciplesOfOOD." Available: http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[2] "Bob Martin's Design Principles page." Available: https://condor.depaul.edu/dmumaugh/OOT/Design-Principles/

[3] M. Fowler, "bliki: Yagni," martinfowler.com. Available: https://martinfowler.com/bliki/Yagni.html

[4] F. Moretti, "Single Responsibility Principle (SRP)," *Francisco Moretti*, May 31, 2023. Available: https://www.franciscomoretti.com/blog/single-responsibility-principle-srp

[5] F. Moretti, "Open-Closed Principle (OCP)," *Francisco Moretti*, Jun. 01, 2023. Available: https://www.franciscomoretti.com/blog/open-closed-principle-ocp

[6] F. Moretti, "Liskov Substitution Principle (LSP)," *Francisco Moretti*, Jun. 01, 2023. Available: https://www.franciscomoretti.com/blog/liskov-substitution-principle-lsp

[7] F. Moretti, "Interface Segregation Principle (ISP)," *Francisco Moretti*, Jun. 28, 2023. Available: https://www.franciscomoretti.com/blog/interface-segregation-principle-isp

[8] F. Moretti, "Dependency Inversion Principle (DIP)," *Francisco Moretti*, May 30, 2023. Available: https://www.franciscomoretti.com/blog/dependency-inversion-principle-dip

[9] G. Gorantala, "SOLID: Learn about the single responsibility principle with examples," *HackerNoon*, Aug. 18, 2023. Available: https://hackernoon.com/solid-learn-about-the-single-responsibility-principle-with-examples

[10] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern," *martinfowler.com*. Available: https://martinfowler.com/articles/injection.html