SOLID PRINICIPLES

The SOLID principles are a set of design guidelines that make software design more understandable, flexible, and maintainable. By applying SOLID principles in your Java code, you can ensure that your software is easier to manage, extend, and refactor.

**Single Responsibility Principle (SRP)**

A class should have only one reason to change, meaning it should be focused on one task or responsibility.

A screenshot of a computer program

Description automatically generated

In the refactored version, OrderService handles only order processing, and PaymentService handles payment processing. This ensures that future changes related to either responsibility do not interfere with one another.

**Open/Closed Principle (OCP)**

A screen shot of a computer code

Description automatically generatedClasses should be open for extension but closed for modification.

In this scenario, adding a new payment type would require modifying the existing PaymentProcessor class, violating the OCP. Instead, we can use abstraction and polymorphism to achieve extensibility:

A screen shot of a computer program

Description automatically generated

**Liskov Substitution Principle (LSP)**

This principle emphasizes that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

A screen shot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Interface Segregation Principle (ISP):**

Clients should not be forced to depend on interfaces they do not use. Instead of one large interface, break it down into smaller, more specific ones.

Or

A class should not be forced to implement methods it doesn't use. Interfaces should be specific to what the class needs.

Imagine a restaurant ordering system where chefs and servers have different tasks.

// Violation of ISP: One large interface forces implementation of unnecessary methods

A screen shot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

**Explanation:**  
By splitting the RestaurantWorker interface into OrderTaker and Cook interfaces, we ensure that Chef and Server classes only implement the methods they actually need, adhering to ISP.

**Dependency Inversion Principle (DIP)**

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

A computer screen shot of a program

Description automatically generated

A screen shot of a computer program

Description automatically generated

By depending on the MessageService interface instead of the concrete EmailService class, the NotificationService can now send messages through any medium (e.g., SMS or Email), allowing more flexibility and better adherence to the DIP.