SOLID

Object Oriented Programing

Ariel University

What Are SOLID Principles?

- SOLID principles are a set of design guidelines in object-oriented programming.
- Aim to make software designs more understandable, flexible, and maintainable.
- **Acronym:**
  - 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:**
  Each class should focus on a single responsibility or functionality.
- **Benefits:**
  - Improves code readability.
  - Simplifies debugging and testing.
- **Example:**
  - **Violation:** A *ReportManager* class handles report generation and database operations.
  - **Solution:** Separate responsibilities into *ReportGenerator* and *DatabaseManager* classes.

Open/Closed Principle (OCP)

- **Definition:**
  - Classes should be open for extension but closed for modification.
  - You can add new functionality without altering existing code.
- **Benefits:**
  - Enhances code flexibility and prevents breaking existing functionality.
- **Example:**
  - Use abstract classes or interfaces to allow new behavior via inheritance instead of modifying existing classes.

Liskov Substitution Principle (LSP)

- **Definition:**
  - If *S* is a subtype of *T*, then objects of type *T* in a program may be replaced with objects of type *S* without altering any of the desirable properties of that program
- **Benefits:**
  - Ensures polymorphism works correctly.
- **Example:**
  - **Violation:** A subclass overrides a base class method in a way that breaks functionality.
  - **Solution:** Design subclasses to honor the behavior expected by the base class.

Interface Segregation Principle (ISP)

- **Definition:**
  - No client should be forced to depend on methods it does not use.
  - Prefer smaller, specific interfaces over a large, general-purpose interface.
- **Benefits:**
  - Reduces code complexity and avoids implementing unnecessary methods.
- **Example:**
  - **Violation:** A *Printer* interface with methods for *Print() ,Scan()* , dna*Fax()* forces all implementers to define unused methods.
  - **Solution:** Split into *Printable* ,*Scannable* dna ,*Faxable* interfaces.

Dependency Inversion Principle (DIP)

- **Definition:**
  - High-level modules should not depend on low-level modules; both should depend on abstractions.
  - Abstractions should not depend on details; details should depend on abstractions.
- **Benefits:**
  - Increases code flexibility and reduces coupling.
- **Example:**
  - Use dependency injection to pass dependencies (e.g., services) into a class instead of hardcoding them.

Dependency injection

- **Violation:** A *UserService* class directly creates an instance of *EmailService* eht selpuoc ylthgit sihT . .sessalc

- **Solution:** Introduce an abstraction, such as an *IMessageService* interface, which both *EmailService* and other message services implement. The *UserService* class then depends on *IMessageService* and receives it via dependency injection.

```
 //Concrete Implementation
public class EmailService implements
IMessageService}
@    Override
    public void sendMessage(String message)}
        System.out.println( + " :tnes liamE"
;(egassem
{
{

 //High-level Module
public class UserService}
    private final IMessageService messageService;

    public UserService(IMessageService
messageService)}
        this;ecivreSegassem = ecivreSegassem.
{

    public void notifyUser(String message)}
        messageService.sendMessage(message);
{
{
```

Benefits of Applying SOLID Principles

- Enhances code maintainability and scalability.
- Simplifies debugging and testing.
- Encourages better collaboration within development teams.
- Reduces the risk of introducing bugs when adding new features.
- Adopt these principles gradually to improve your software development process.