3. Solid Principles

Thursday, 5 February 2026 12:01 PM

SOLID Principles

The **SOLID** principles are a set of five design principles in object-oriented programming and software design. They aim to make software more understandable, flexible, and maintainable by following a structured approach

SOLID Principles

S – Single Responsibility Principle (SRP)

O – Open/Closed Principle (OCP)

O – Open/Close Principle

L – Liskov Substitution Principle (LSP)

I – Interface Segregation Principle (ISP)

D – Dependency Inversion Principle (DIP)

S-Single Responsibility Principle

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

Example: A `User` class should only handle user-related logic, while database-related operations should be handled by a separate `UserRepository` class.

O – Open/Close Principle

- Software entities (classes, modules, functions) should be **open for extension** but **closed for modification**.
- Example: Adding new functionality to a system using **inheritance** or **composition** without modifying existing code.

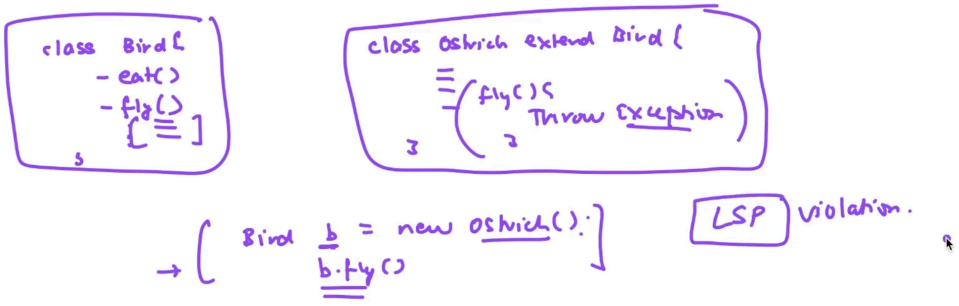
L – Liskov Substitution Principle

- The **Liskov Substitution Principle (LSP)** states that objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program. It ensures that a subclass can stand in for its parent class and function correctly in any context that expects the parent class.

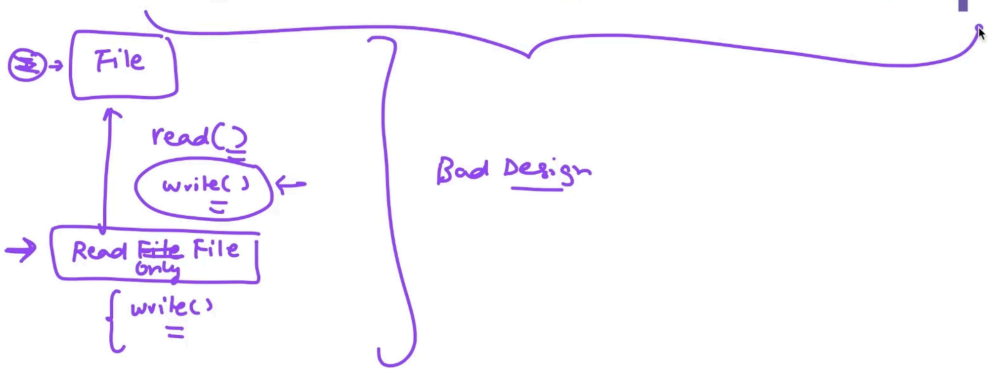
L – Liskov Substitution Principle

SOLID

- The **Liskov Substitution Principle (LSP)** states that objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program. It ensures that a subclass can stand in for its parent class and function correctly in any context that expects the parent class.



L – Liskov Substitution Principle



L – Liskov Substitution Principle

- The **Liskov Substitution Principle (LSP)** states that objects of a superclass should be replaceable with objects of a subclass without altering the correctness of the program. It ensures that a subclass can stand in for its parent class and function correctly in any context that expects the parent class.

- No client should be forced to depend on methods it doesn't use. Split large interfaces into smaller, more specific ones.

I – Interface Segregation Principle

- The **Interface Segregation Principle** ensures that classes are not burdened with methods they don’t need. It promotes better design by breaking large, general-purpose interfaces into smaller, more specific ones.

  It improves **maintainability**, **flexibility**, and **testability** by ensuring that classes only have the dependencies they actually require.

D – Dependency Inversion Principle

- High-level modules should not depend on low-level modules; both should depend on **abstractions**.