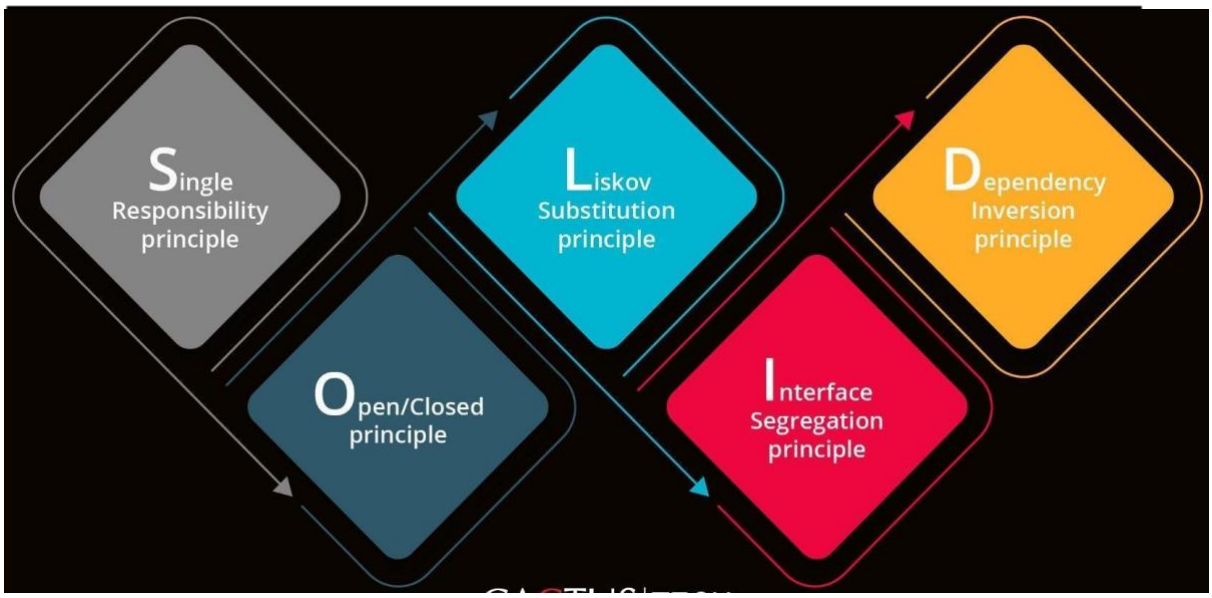
The Understanding Of The SOLID Principles With Functional Programming



Introduction

SOLID principles come into picture because these are the design principles that help us in encouraging creating more maintainable, understandable, and flexible software.

As our applications grow in size, we can reduce their complexity by using **SOLID** principles.

SOLID Principles:-

The following five concepts make up our SOLID principles:

1. **S**ingle Responsibility
2. **O**pen/Closed
3. **L**iskov Substitution
4. Interface Segregation
5. **D**ependency Inversion

These five software development principles are guidelines to follow when building software so that it is easier to scale and maintain. They were made popular by a software engineer, Robert C. Martin.

BENEFITS OF SOLID PRINCIPLES-

Some of the benefits SOLID Principle holds are as follows:-

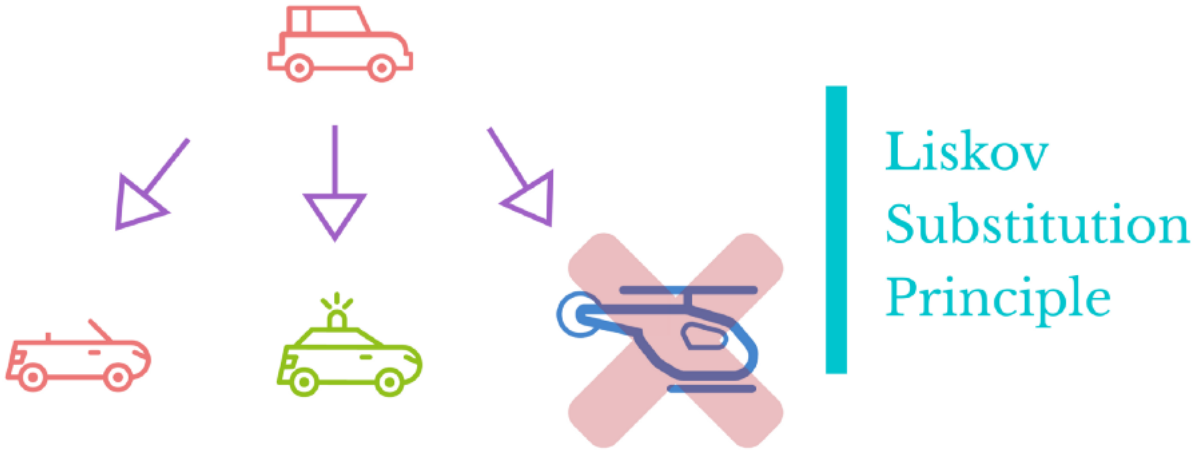
❖Loose Coupling

❖Code Maintainability

❖Dependency Management

Liskov Substitution:

Define:- 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.



- When a child Class cannot perform the same actions as its parent Class, this can cause bugs.

- If you have a Class and create another Class from it, it becomes a parent and the new Class becomes a child. The child Class should be able to do everything the parent Class can do. This process is called Inheritance.

- The child Class should be able to process the same requests and deliver the same result as the parent Class or it could deliver a result that is of the same type.

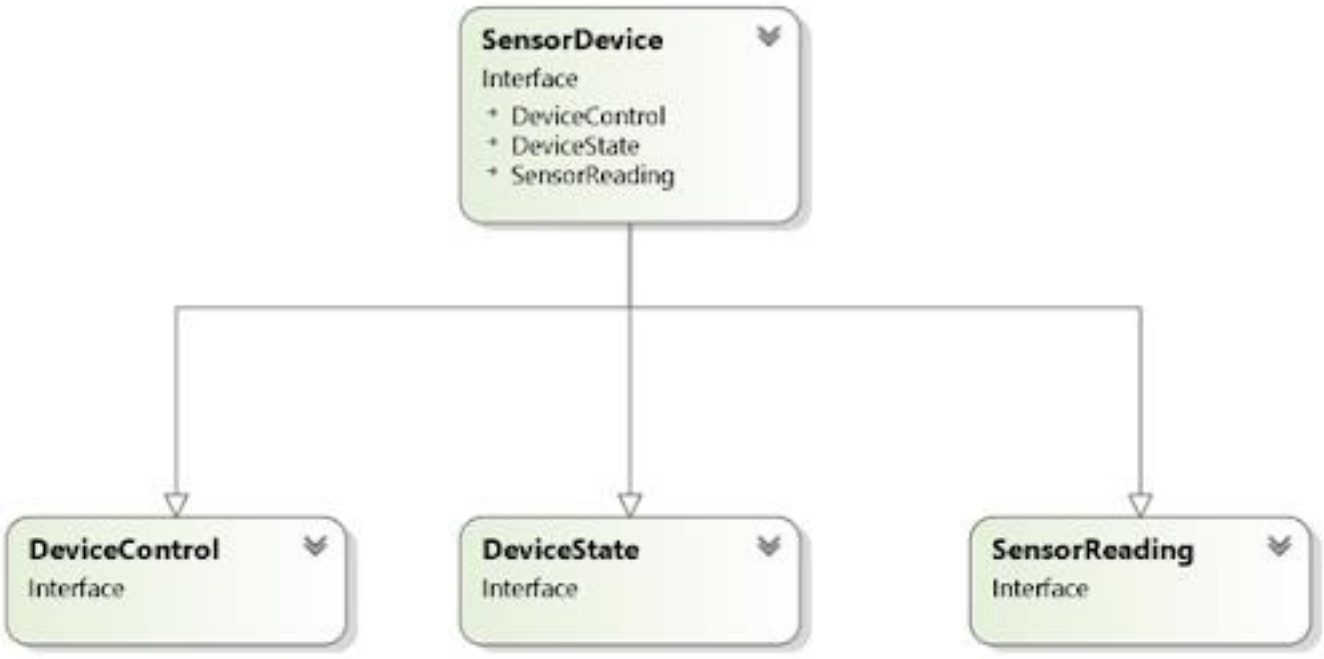
● The picture shows that the parent Class delivers Coffee(it could be any type of coffee). It is acceptable for the child Class to deliver Cappuccino because it is a specific type of Coffee, but it is NOT acceptable to deliver Water.

● If the child Class doesn’t meet these requirements, it means the child Class is changed completely and violates this principle.

● LSP also applies in case we use generic or parametric programming where we create functions that work on a variety of types, they all hold a common truth that makes them interchangeable.

● This pattern is super common in functional programming, where you create functions that embrace polymorphic types (aka generics) to ensure that one set of inputs can seamlessly be substituted for another without any changes to the underlying code.

Interface Segregation:



This principle was first defined by Robert C. Martin as: “**Clients should not be forced to depend upon interfaces that they do not use**“.

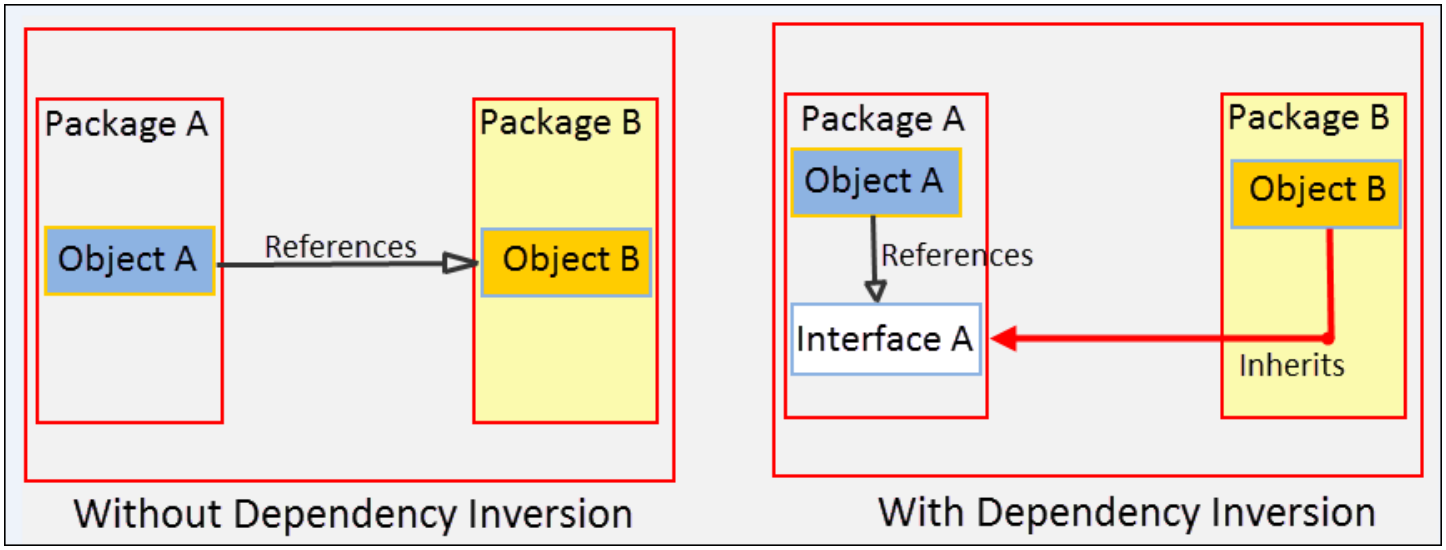
The goal of this principle is to **reduce the side effects of using larger interfaces by breaking application interfaces into smaller ones**. It's

similar to the Single Responsibility Principle, where each class or interface serves a single purpose.

Precise application design and correct abstraction is the key behind the Interface Segregation Principle. **Though it'll take more time and effort in the design phase of an application and might increase the code complexity, in the end, we get a flexible code.**

The Interface Segregation Principle is an important concept while designing and developing applications. Adhering to this principle helps to avoid bloated interfaces with multiple responsibilities. This eventually helps us to follow the Single Responsibility Principle as well.

Dependency Inversion:



Based on this idea, Robert C. Martin's definition of the Dependency Inversion Principle consists of two parts:

1. High-level modules should not depend on low-level modules. Both should depend on the abstraction.
2. Abstractions should not depend on details. Details should depend on abstractions.

The general idea of this principle is as simple as it is important: High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.

An important detail of this definition is, that high-level **and** low-level modules depend on the abstraction. The design principle does not just change the direction of the dependency, as you might have expected when you read its name for the first time. It splits the dependency between the high-level and low-level modules by introducing an abstraction between them. So in the end, you get two dependencies:

1. the high-level module depends on the abstraction, and
2. the low-level depends on the same abstraction.