Software Design Principles

Software design principles are a set of guidelines that helps developers to make a good system design. The most important principle is SOLID principle.

Single Responsibility Principle (SRP)

This principle states that there should never be more than one reason for a class to change. This means that you should design your classes in such a way that each class should have a single purpose.

Example - An Account class is responsible for managing Current and Saving Account but a CurrentAccount and a SavingAccount classes would be responsible for managing current and saving accounts respectively. Hence both are responsible for single purpose only. Hence we are moving towards specialization.

Open/Closed Principle (OCP)

This principle states that software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. The "closed" part of the rule states that once a module has been developed and tested, the code should only be changed to correct bugs. The "open" part says that you should be able to extend existing code in order to introduce new functionality.

Example – A PaymentGateway base class contains all basic payment related properties and methods. This class can be extended by different PaymentGateway classes for different payment gateway vendors to achieve theirs functionalities. Hence it is open for extension but closed for modification

Liscov Substitution Principle (LSP)

This principle states that functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

Example - Assume that you have an inheritance hierarchy with Person and Student. Wherever you can use Person, you should also be able to use a Student, because Student is a subclass of Person.

Interface Segregation Principle (ISP)

This principle states that Clients should not be forced to depend upon interfaces that they don’t use. This means the number of members in the interface that is visible to the dependent class should be minimized.

Example - The service interface that is exposed to the client should contains only client related methods not all.

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle states that:

1. High level modules should not depend upon low level modules. Both should depend upon abstractions.
2. Abstractions should not depend upon details. Details should depend upon abstractions. It helps us to develop loosely couple code by ensuring that high-level modules depend on abstractions rather than concrete implementations of lower-level modules. The Dependency Injection pattern is an implementation of this principle.

Example - The Dependency Injection pattern is an implementation of this principle

Dependency Injection Pattern in C#

What is Dependency Injection?

Dependency Injection (DI) is a software design pattern that allows us to develop loosely coupled code. DI is a great way to reduce tight coupling between software components. DI also enables us to better manage future changes and other complexity in our software. The purpose of DI is to make code maintainable.

What are the advantages of Dependency Injection?

The advantages of using Dependency Injection pattern and Inversion of Control are the following:

- Reduces class coupling
- Increases code reusing
- Improves code maintainability
- Improves application testing