**SOLID Introduction**

1. SOLID principles are the design principles that enable us manage most of the software design problems

2. The term SOLID is an acronym for five design principles intended to make software designs more understandable, flexible and maintainable

3. The principles are a subset of many principles promoted by Robert C. Martin

4. The SOLID acronym was first introduced by Michael Feathers

SOLID Acronym

S : Single Responsibility Principle (SRP)

O : Open closed Principle (OSP)

L : Liskov substitution Principle (LSP)

I : Interface Segregation Principle (ISP)

D : Dependency Inversion Principle (DIP)

Single Responsibility Principle

1. Robert C. Martin expresses the principle as, "A class should have only one reason to change”

2.Every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class

Liskov Substitution Principle

1. Introduced by Barbara Liskov state that “objects in a program should be replaceable with instances of their sub-types without altering the correctness of that program”

2. If a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module

3. We can also state that Derived types must be substitutable for their base types

Open/Closed Principle

1. “Software entities should be open for extension, but closed for modification”

2. The design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code

3. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged

Interface Segregation Principle

1. “Many client-specific interfaces are better than one general-purpose interface”

2.We should not enforce clients to implement interfaces that they don't use. Instead of creating one big interface we can break down it to smaller interfaces

Dependency Inversion Principle

1. One should “depend upon abstractions, [not] concretions"

2. Abstractions should not depend on the details whereas the details should depend on abstractions

3. High-level modules should not depend on low level modules

If we don’t follow SOLID Principles we

1. End up with tight or strong coupling of the code with many other modules/applications

2. Tight coupling causes time to implement any new requirement, features or any bug fixes and some times it creates unknown issues

3. End up with a code which is not testable

4. End up with duplication of code

5. End up creating new bugs by fixing another bug

6. End up with many unknown issues in the application development cycle

Following SOLID Principles helps us to

1. Achieve reduction in complexity of code

2. Increase readability, extensibility and maintenance

3. Reduce error and implement Reusability

4. Achieve Better testability

5. Reduce tight coupling

Solution to develop a successful application depends on

Architecture : choosing an architecture is the first step in designing application based on the requirements. Example : MVC, WEBAPI, MVVM..etc

Design Principles : Application development process need to follow the design principles

Design Patterns : We need to choose correct design patterns to build the software.

**What is Single Responsibility**

2. Single Responsibility Example

In our previous video we discussed S in the SOLID is acronym for Single Responsibility Principle (SRP)

As per the single responsibility principle

1. A class should have only one reason to change

2. Which means, every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.

Encapsulation is one of the fundamentals of OOP. At this moment, understanding more about encapsulation is out of scope of this session. However, we strongly recommend you to refer to the C# tutorial playlist for more details on Object oriented principles. Now you might be wondering what do we achieve with the Single Responsibility Principle or rather with the SOLID Design Principles.

Let's first understand the motivation behind the usage of SOLID Principles

In any enterprise software application development when we design and develop software systems, we need to account the below factors during the development cycle.

Maintainability : Maintainable systems are very important to the organisations.

Testability : Test driven development (TDD) is required when we design and develop large scale systems

Flexibility and Extensibility : Flexibility and extensibility is a very much desirable factor of enterprise applications.Hence we should design the application to make it flexible so that it can be adapt to work in different ways and extensible so that we can add new features easily.

Parallel Development : It is one of the key features in the application development as it is not practical to have the entire development team working simultaneously on the same feature or component.

Loose Coupling : We can address many of the requirements listed above by ensuring that our design results in an application that loosely couples many parts that makes up the application.

SOLID Principles and Design Patterns plays a key role in achieving all of the above points.

In Single Responsibility Principle

1. Each class and module should focus on a single task at a time

2. Everything in the class should be related to that single purpose

3. There can be many members in the class as long as they related to the single responsibility

4. With SRP, classes become smaller and cleaner

5. Code is less fragile

**Interface Segregation Principle**

1. The interface-segregation principle (ISP) states that "no client should be forced to depend on methods it does not use".

2. Which means, Instead of one fat interface many small interfaces are preferred based on groups of methods with each one serving one submodule.

3. The ISP was first used and formulated by Robert C. Martin while consulting for Xerox.

Let us now understand how the ISP was evolved with a case study.

CASE Study

Problem

• As we all know Xerox Corporation manufactures printer systems. In their development process of new systems Xerox had created a new printer system that could perform a variety of tasks such as stapling and faxing along with the regular printing task.

• The software for this system was created from the ground up.

• As the software grew for Xerox, making modifications became more and more difficult so that even the smallest change would take a redeployment cycle of an hour, which made development nearly impossible.

• The design problem was that a single Job class was used by almost all of the tasks. Whenever a print job or a stapling job needed to be performed, a call was made to the Job class.

• This resulted in a 'fat' class with multitudes of methods specific to a variety of different clients.

Because of this design, a staple job would know about all the methods of the print job, even though there was no use for them.

Solution

• To overcome this problem Robert C Martin suggested a solution which is called the Interface Segregation Principle.

• Which means, Instead of one fat interface many small interfaces are preferred based on groups of methods with each one serving one submodule.