SOLID PRINCIPLES

Object Oriented Programming

What is SOLID principles

- Single Responsibility principle
- Open/Closed principle
- Liskov Substitution principle
- Interface Segregation principle
- Dependency Inversion principle

Why should we use SOLID principles

The main goal of the SOLID principles is to reduce dependencies so that developers can change one area of software without impacting other area.

These principles are intended to make designs or source code easier to understand, maintain, and extend.

So, this will lead to better code for readability, maintainability, design patterns, and testability.

It avoids the bad design of the software.

Single Responsibility principle

- It states that “**One class should have one and only one responsibility**” Which specifically means - we should write, change, and maintain a class only for one purpose.
- Change in the class should be done only when there is need to change state of one particular object or instance.
- Example : POJOS follow SRP. Suppose we have Employee and Address Class, If we want to change the state of Employee then we do not need to modify the class Address and vice-versa.
- If we have merged both as single POJO , then modification in one field for address (like state) needs to modify and Whole POJO including Employee
- Worst Design – if we don’t follow SRP Hitting Database in POJO of Employee Class. That’s why we have service layer, DAP layer and Entities separated.

Why is Single Responsibility principle important

- In real world, requirement changes and so does your code implementation to cater the changing requirement , The more responsibilities your class has, the more often you need to change it.
- To prevent frequent changes to same class, testing is easier - with a single responsibility, the class will have fewer test cases and easier to Understand Less functionality also which means fewer dependencies to other classes.
- So best practice is to use layers in your application and break God classes into smaller classes or modules

Open/Closed principle

- It states that **"Software components should be open for extension, but closed for modification" .**
- The term "Open for extension" means that we can extend and include extra functionalities in our code without altering or affecting our existing implementation.
- The term "Closed for modification" means that after we add the extra functionality, we should not modify the existing implementation.
- In real world, we must have noticed that you change something to cater a new requirement and some other functionality breaks because of our change. To prevent that we have this principle in hand. It is one of the most important concept in in solid principles

How to implement Open/closed principle

- The application classes should be designed in such a way that whenever developers want to change the flow of control in specific conditions in application, all they need to extend the class and override some functions and that's it.
- Example - created a pojo employee with id , name. now new functionality comes which says add Training location. your constructor will fail for employees who didnot do training. better extend employee class, name it Trained employee then add constructor.

Liskov Substitution Principle

- It states that **"Software should not alter the desirable results when we replace a parent type with any of the subtypes" or "Derived types must be completely substitutable for their base types".**
- LSP means that the classes, which developers created by extending another class, should be able to fit in application without failure.
- For example, printing employee details from child or parent reference.
- To implement this principle, we require the objects of the subclasses to behave in the same way as the objects of the superclass.

Why is Liskov Substitution Principle important

- This principle avoids misusing inheritance.
- It helps us to confirm the "IS - A" relationship.
- We can also say that subclasses must fulfill a contract defined by the base class.

Interface segregation principle

- **This principle states that "Clients should not be forced to implement unnecessary methods which they will not use".**
- ISP is applicable to interfaces as same a single responsibility principle holds to classes.
- ISP states that we should split our interfaces into smaller and more specific ones.
- It is used to prevent client from unnecessarily getting stuck in implementing unwanted principles.

Dependency Inversion principle

- **The principle states that we must use abstraction (abstract classes and interfaces) instead of concrete implementations.**
- **High level module should not directly depend upon low level module.**
- We should design our software in such a way that various modules can be separated from each other using an abstract layer to bind them together.
- It allows a programmer to remove hardcoded dependencies so that the application becomes loosely couples and extendable.

Example for Dependency Inversion principle

```
Public class student{
    Private Address address;
Public student(){
    address = new Address();
}}
```

In this example, Student class requires an Address object and it is responsible for initializing and using the Address object.

If Address class is changed in future then we have to make changes in Student class also.

This makes the tight coupling between Student and Address objects. We can resolve this problem using the dependency inversion design pattern.

Address object will be implemented independently and will be provided to Student when Student is instantiated by using constructor based or setter based dependency inversion.

Thank you