**SOLID Principles**

Robert C. Martin gave five object oriented design principles, and the acronym S.O.L.I.D is used for it. Each letter of this acronym talks about principles in Java. When you use all the principles of S.O.L.I.D in a combined manner, it becomes easier for you to develop software that can be managed easily. The other features of using S.O.L.I.D are:

* It avoids code smells
* Quickly refractor code
* Can do adaptive or agile software development

When you use the principle of S.O.L.I.D in your coding, you start writing the code that is both efficient and effective.

## **What is the meaning of S.O.L.I.D?**

As stated above, S.O.L.I.D represents five principles of Java which are:

1. S: Single responsibility principle
2. O: Open-closed principle
3. L: Liskov substitution principle
4. I: Interface segregation principle
5. D: Dependency inversion principle

The article discusses each of this principle in-depth. We will start by considering the very first principle, which is the single responsibility principle.

## **Single Responsibility Principle (SRP)**

According to the **single responsibility principle**, there should be only one reason due to which a class has to be changed. It means that a class should have one task to do. This principle is often termed as subjective.

The principle can be well understood with an example. Imagine there is a class that performs the following operations.

* Connected to a database
* Read some data from database tables
* Finally, write it to a file.

Have you imagined the scenario? Here the class has multiple reasons to change, and few of them are the modification of file output, new database adoption. When we are talking about single principle responsibility, we would say, there are too many reasons for the class to change; hence, it doesn’t fit properly in the single responsibility principle.

## **Open Closed Principle**

According to **open closed principle**, entities or objects should remain open for extension, but they should stay closed for modification. To be precise, according to this principle, a class should be written in such a manner that it performs its job flawlessly without the assumption that people in the future will simply come and change it. Hence, the class should remain closed for modification, but it should have the option to get extended. Ways of extending the class include:

* Inheriting from the class
* Overwriting the required behaviors from the class
* Extending certain behaviors of the class

An excellent example of an open-closed principle can be understood with the help of browsers. Do you remember installing extensions in your chrome browser?

The primary function of chrome browser is to surf different sites. Do you want to check grammar when you are writing an email using chrome browser? If yes, you can simply use Grammarly extension, it provides you grammar check on the content.

This mechanism where you are adding things for increasing the functionality of the browser is an extension. Hence, the browser is a perfect example of functionality that is open for extension but is closed for modification. In simple words, you can enhance the functionality by adding/installing plugins on your browser, but cannot build anything new.

Let’s take another example.

You are using any Spring function functionality. You can obviously not change the core logic of it, but you can extend Spring framework classes and create your own one.

## **Liskov Substitution Principle**

Liskov substitution principle assumes q(x) to be a property, provable about entities of x which belong to type T. Now, according to this principle, the q (y) should be now provable for objects y that belong to type S, and the S is actually a subtype of T. Are you now confused and don’t know what Liskov substitution principle actually means? The definition of it might be a bit complex, but in fact, it is quite easy. The only thing is that every subclass or derived class should be substitutable for their parent or base class.

You can say that it is a unique object-oriented principle. The principle can further be simplified by understanding this principle; a child type of a particular parent type without making any complication or blowing things up should have the ability to stand in for that parent.This principle is closely related to the Liskov Substitution principle.

## **Interface Segregation Principle**

According to **interface segregation principle**, a client, no matter what, should never be forced to implement an interface that it does not use or the client should never be obliged to depend on any method, which is not used by them.

So basically, the interface segregation principles as you prefer the interfaces, which are small but client specific instead of the monolithic and bigger interface.

In short, it would be bad for you to force the client to depend on a certain thing, which they don’t need.

## **Dependency Inversion Principle**

According to this, dependency inversion principle, entities should depend only on abstractions but not on concretions. According to it, the high-level module must never rely on any low-level module but should depend on abstractions. Let us again understand it through another practical example.

You go to a local store to buy something, and you decide to pay for it by using your debit card. So, when you give your card to the clerk for making the payment, the clerk doesn’t bother to check what kind of card you have given. Even if you have given a Visa card, he will not put out a Visa machine for swiping your card. The type of credit card or debit card that you have for paying doesn't even matter; they will simply swipe it. So, in this example, you can see that both you and the clerk are dependent on the credit card abstraction, and you are not worried about the specifics of the card. This is what a dependency inversion principle is.

## **Wrap Up**

I hope that now you know the basic definition of all the five components of S.O.L.I.D, which are single responsibility principle, open, closed principle, Liskov substitution principle, interface segregation principle, and dependency inversion. So, basically, whenever you write code, you have to keep these core principles in your mind, and actually, to be honest, you have to take these principles as your base map while writing those codes to make it efficient and effective.

The management of the code written by using these principles is also an easy task to do. Using this principle initially in your coding might look odd, but you need to practice writing your codes using these principles, and gradually it will become a part of you. And then, the code that you have written can be easily modified, extended, refactored, or tested without facing any problem according to the requirement that you meet. So practice this principle whenever possible to make your coding better.