**What is solid principle ?**

SOLID principles are the design principles that enable us to manage most of the software design problems. Robert C. Martin compiled these principles in the 1990s. These principles provide us with ways to move from tightly coupled code and little encapsulation to the desired results of loosely coupled and encapsulated real needs of a business properly. SOLID is an acronym of the following.

* S: Single Responsibility Principle (SRP)
* O: Open closed Principle (OCP)
* L: Liskov substitution Principle (LSP)
* I: Interface Segregation Principle (ISP)
* D: Dependency Inversion Principle (DIP)

Or it can be defined as :

**SOLID** principles are object-oriented design concepts relevant to software development. SOLID is an acronym for five other class-design principles: **S**ingle Responsibility Principle, **O**pen-Closed Principle, **L**iskov Substitution Principle, **I**nterface Segregation Principle, and **D**ependency Inversion Principle.

### **1. Single responsibility principle**

The Single Responsibility Principle states that a class should have one and only one reason for change, i.e., a subsystem, module, class or a function shouldn't have more than one reason for change. If there are two reasons for a class to change, the functionality should be split into two classes with each class handling one responsibility. When a class has more than one responsibility, such responsibilities are coupled and this coupling leads to designs that are fragile and can break over time.

The benefits of SRP include:

1. Reduction in complexity of code
2. Increased readability, extensibility and maintenance
3. Reusability and reduced errors
4. Better testability
5. Reduced coupling

**2. What is open closed principle**

The *Open/Closed (OC) Principle* states that a class object should remain open for extension but closed for modification. The class should be able to be extended (to add new functionality, for example), but the original code should be open to change only to correct bugs.

Or it can be defined as :

The general idea of this principle is great. It tells you to write your code so that you will be able to add new functionality without changing the existing code. That prevents situations in which a change to one of your classes also requires you to adapt all depending classes

**3. What is liskov substitution principle?**

The Liskov Substitution Principle is a Substitutability principle in object-oriented programming Language. This principle states that, if S is a subtype of T, then objects of type T should be replaced with the objects of type S.

In simple words we can say that, when we have a base class and child class relationships i.e. inheritance relationships, then, if we can successfully replace the object/instance of a parent class with an object/instance of the child class, without affecting the behavior of the base class instance, then it is said to be in Liskov Substitution Principle. If you are not getting this point properly, don’t worry, we will see some real-time examples to understand this concept.

For example, a father is a teacher whereas his son is a doctor. So here, in this case, the son can’t simply replace his father even though both belong to the same family.

##### Example: Without using the Liskov Substitution Principle in C#:

Let us first understand one example without using the Liskov Substitution Principle in C#. In the following example, first, we create the Apple class with the method GetColor. Then we create the Orange class which inherits the Apple class as well as overrides the GetColor method of the Apple class. The point is that an Orange cannot be replaced by an Apple, which results in printing the color of the apple as Orange as shown in the below example.

namespace *SOLID\_PRINCIPLES.LSP*

{

class Program

{

static void Main(string[] args)

{

Apple apple = new Orange();

Console.WriteLine(apple.GetColor());

}

}

public class Apple

{

public virtual string GetColor()

{

return "Red";

}

}

public class Orange : Apple

{

public override string GetColor()

{

return "Orange";

}

}

}

As you can see in the above example, Apple is the base class and Orange is the child class i.e. there is a Parent-Child relationship. So, we can store the child class object in the Parent Reference variable i.e. Apple apple = new Orange(); and when we call the GetColor i.e. apple.GetColor(), then we are getting the color of the Orange not the color of Apple. That means once the child object is replaced i.e. Apple storing the Orange object, the behavior is also changed. This is against the LSP Principle. The Liskov Substitution Principle in C# states that even the child object is replaced with the parent, the behavior should not be changed. So, in this case, if we are getting the color of Apple instead of Orange, then it follows the Liskov Substitution Principle. That means there is some issue with our software design. Let us see how to overcome the design issue and makes the application follow Liskov Substitution Principle.

##### Example Using the Liskov Substitution Principle in C#

Let’s modify the previous example to follow the Liskov Substitution Principle. Here, first, we need a generic base class such as Fruit which is going to be the base class for both Apple and Orange. Now you can replace the Fruit class object with its subtypes either Apple and Orage and it will behave correctly. Now, you can see in the below code, we created the super Fruit class as an abstract class with the GetColor abstract method and then the Apple and Orange class inherited from the Fruit class and implement the GetColor method.

namespace *SOLID\_PRINCIPLES.LSP*

{

class Program

{

static void Main(string[] args)

{

Fruit fruit = new Orange();

Console.WriteLine(fruit.GetColor());

fruit = new Apple();

Console.WriteLine(fruit.GetColor());

}

}

public abstract class Fruit

{

public abstract string GetColor();

}

public class Apple : Fruit

{

public override string GetColor()

{

return "Red";

}

}

public class Orange : Fruit

{

public override string GetColor()

{

return "Orange";

}

}

4. interface segregation principle;

The Interface Segregation Principle states that **“Clients should not be forced to implement any methods they don’t use. Rather than one fat interface, numerous little interfaces are preferred based on groups of methods with each interface serving one submodule“.**

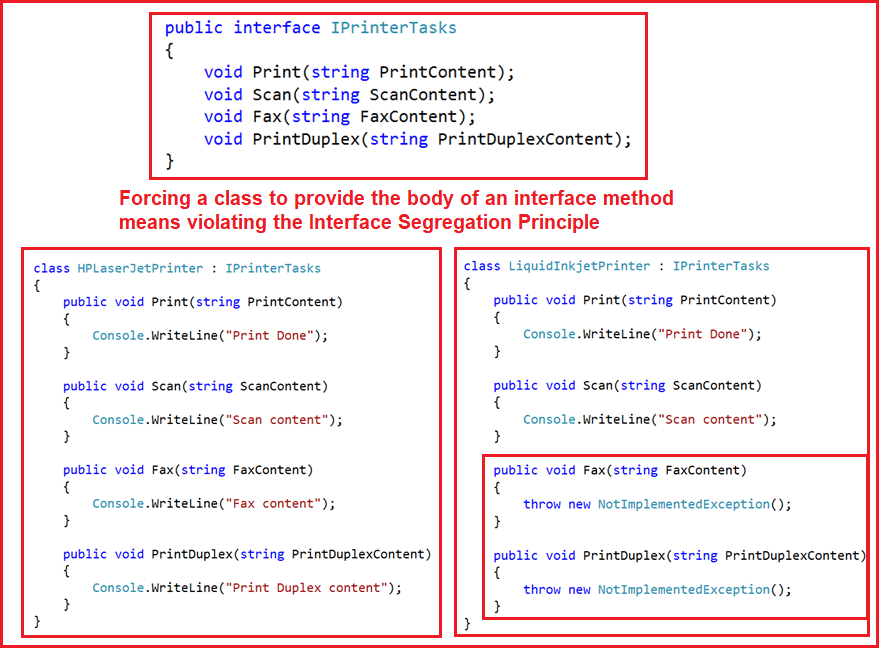
Let us break down the above definition into two parts.

1. First, no class should be forced to implement any method(s) of an interface they don’t use.
2. Secondly, instead of creating large or you can say fat interfaces, create multiple smaller interfaces with the aim that the clients should only think about the methods that are of interest to them.

As per the [**Single Responsibility Principle**](https://dotnettutorials.net/lesson/single-responsibility-principle/) of SOLID, like classes, interfaces also should have a single responsibility. That means we shouldn’t force any class to implement any method(s) which they don’t require.

##### **Let us understand the Interface Segregation Principle in C# with an example.**

Please have a look at the following diagram.



As you can see in the above diagram, we have an interface i.e. IPrinterTasks declared with four methods. Now if any class wants to implement this interface then that class should have to provide the implementation to all the four methods of the IPrinterTasks interface. As you can see in the above diagram, we have two classes HPLaserJetPrinter and LiquidInkjetPrinter who want the printer service.

But the requirement is the HPLaserJetPrinter wants all the services provided by the IPrinterTasks while the LiquidInkjetPrinter wants only the Print and Scan service of the printer. As we have declared all the methods within the IPrinterTasks interface, then it is mandatory for the LiquidInkjetPrinter class to provide implementation to Scan and Print methods along with the Fax and PrinctDulex method which are not required by the class.

.