# **SOLID Design Principles Explained - C#**

In Object Oriented Programming (OOP), SOLID is an acronym, introduced by Michael Feathers, for five design principles used to make software design more understandable, flexible, and maintainable.

There are five SOLID principles:

1. Single Responsibility Principle (SRP)
2. Open Closed Principle (OCP)
3. Liskov Substitution Principle (LSP)
4. Interface Segregation Principle (ISP)
5. Dependency Inversion Principle (DIP)

## Single Responsibility Principle (SRP)

**Definition:** A class should have only one reason to change.

In layman terminology, this means that a class should not be loaded with multiple responsibilities and a single responsibility should not be spread across multiple classes or mixed with other responsibilities. The reason is that more changes requested in the future, the more changes the class need to apply.

### **Understanding**

In simple terms, a module or class should have a very small piece of responsibility in the entire application. Or as it states, a class/module should have not more than one reason to change.

If a class has only a single responsibility, it is likely to be very robust. It’s easy to verify its working as per logic defined. And it’s easy to change in class as it has single responsibility.

The Single Responsibility Principle provides another benefit. Classes, software components and modules that have only one responsibility are much easier to explain, implement and understand than ones that give a solution for everything.

This also reduces number of bugs and improves development speed and most importantly makes developer’s life lot easier.

## Open Closed Principle (OCP)

**Definition:** Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

### **Understanding**

This principle suggests that the class should be easily extended but there is no need to change its core implementations.

The application or software should be flexible to change. How change management is implemented in a system has a significant impact on the success of that application/ software. The OCP states that the behaviors of the system can be extended without having to modify its existing implementation.

i.e. New features should be implemented using the new code, but not by changing existing code. The main benefit of adhering to OCP is that it potentially streamlines code maintenance and reduces the risk of breaking the existing implementation.

### **Implementation**

Let’s take an example of bank accounts like regular savings, salary saving, corporate etc. for different customers. As for each customer type, there are different rules and different interest rates. The code below violates OCP principle if the bank introduces a new Account type. Said code modifies this method for adding a new account type.

We can apply OCP by using interface, abstract class, abstract methods and virtual methods when you want to extend functionality. Here I have used interface for example only but you can go as per your requirement.

interface IAccount

{

// members and function declaration, properties

decimal Balance { get; set; }

decimal CalcInterest();

}

//regular savings account

public class RegularSavingAccount : IAccount

{

public decimal Balance { get; set; } = 0;

public decimal CalcInterest()

{

decimal Interest = (Balance \* 4) / 100;

if (Balance < 1000) Interest -= (Balance \* 2) / 100;

if (Balance < 50000) Interest += (Balance \* 4) / 100;

return Interest;

}

}

//Salary savings account

public class SalarySavingAccount : IAccount

{

public decimal Balance { get; set; } = 0;

public decimal CalcInterest()

{

decimal Interest = (Balance \* 5) / 100;

return Interest;

}

}

//Corporate Account

public class CorporateAccount : IAccount

{

public decimal Balance { get; set; } = 0;

public decimal CalcInterest()

{

decimal Interest = (Balance \* 3) / 100;

return Interest;

}

}

In the above code three new classes are created; regular saving account, SalarySavingAccount, and CorporateAccount, by extending them from IAccount.

This solves the problem of modification of class and by extending interface, we can extend functionality.

Above code is implementing both OCP and SRP principle, as each class has single is doing a single task and we are not modifying class and only doing an extension.

## Liskov Substitution Principle (LSP)

**Definition by Robert C. Martin:** Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

The **Liskov substitution principle (LSP)** is a definition of a subtyping relation, called **(strong) behavioral subtyping,** that was initially introduced by Barbara Liskov in a 1987 conference keynote address titled **Data abstraction and hierarchy.**

### **Understanding**

LSP states that the child class should be perfectly substitutable for their parent class. If class C is derived from P then C should be substitutable for P.

We can check using LSP that inheritance is applied correctly or not in our code.

LSP is a fundamental principle of SOLID Principles and states that if program or module is using base class then derived class should be able to extend their base class without changing their original implementation.

### **Implementation**

Let’s consider the code below where LSP is violated. We cannot simply substitute a Triangle, which results in printing shape of a triangle, with Circle.

namespace Demo

{

public class Program

{

static void Main(string[] args)

{

Triangle triangle = new Circle();

Console.WriteLine(triangle.GetColor());

}

}

public class Triangle

{

public virtual string GetShape()

{

return " Triangle ";

}

}

public class Circle : Triangle

{

public override string GetShape()

{

return "Circle";

}

}

}

To correct above implementation, we need to refactor this code by introducing interface with method called GetShape.

namespace Demo

{

class Program

{

static void Main(string[] args)

{

Shape shape = new Circle();

Console.WriteLine(shape.GetShape());

shape = new Triangle ();

Console.WriteLine(shape.GetShape());

}

}

public abstract class Shape

{

public abstract string GetShape();

}

public class Triangle: Fruit

{

public override string GetShape()

{

return "Triangle";

}

}

public class Circle : Triangle

{

public override string GetShape()

{

return "Circle";

}

}

}

## Interface Segregation Principle (ISP)

**Definition:** No client should be forced to implement methods which it does not use, and the contracts should be broken down to thin ones.

### **Understanding**

Interface segregation principle is required to solve the design problem of the application. When all the tasks are done by a single class or in other words, one class is used in almost all the application classes then it has become a fat class with overburden. Inheriting such class will results in having sharing methods which are not relevant to derived classes but its there in the base class so that will inherit in the derived class.

**Using ISP, we can create separate interfaces for each operation or requirement rather than having a single class to do the same work.**

### **Implementation**

In below code, ISP is broken as process method is not required by OfflineOrder class but is forced to implement.

public interface IOrder

{

void AddToCart();

void CCProcess();

}

public class OnlineOrder : IOrder

{

public void AddToCart()

{

//Do Add to Cart

}

public void CCProcess()

{

//process through credit card

}

}

public class OfflineOrder : IOrder

{

public void AddToCart()

{

//Do Add to Cart

}

public void CCProcess()

{

//Not required for Cash/ offline Order

throw new NotImplementedException();

}

}

We can resolve this violation by dividing IOrder Interface.

public interface IOrder

{

void AddToCart();

}

public interface IOnlineOrder

{

void CCProcess();

}

public class OnlineOrder : IOrder, IOnlineOrder

{

public void AddToCart()

{

//Do Add to Cart

}

public void CCProcess()

{

//process through credit card

}

}

public class OfflineOrder : IOrder

{

public void AddToCart()

{

//Do Add to Cart

}

}