

## Design Principles or SOLID Principles

SOLID principles sets some guide lines or golden rules for writing proper code logic for a software program or project. It helps in maintaining the better scalability, less time effort for a developer to fix any bug and easier reusability. As per the word SOLID refers

S – Single responsibility principle.

O – Open closed principle.

L – Liskov substitution principle.

I – Interface segregation principle.

D – Dependency inversion principle.

### Single Responsibility Principle:

- A class or interface should have only one responsibility. It may have single reason to change and encapsulate.
- If a class or interface have more than one responsibility then divide the class or interface into multiple class or interface.
- It keeps the code small and simple.

In the below example, if we observe the interface **IProcessData** has three methods. Where the InsertProducts and DeleteProducts has responsibility on the class to perform products insert and products delete. Whereas, the CalculatePriceByQuantity would need to perform a different responsibility. Hence, single responsibility principle does not satisfy.

```
1 namespace SRPDP
2 {
3     6 references
4     public class Product
5     {
6         0 references
7         public int ProductId { get; set; }
8         0 references
9         public string? ProductName { get; set; }
10        0 references
11        public decimal ProductPrice { get; set; }
12        0 references
13        public int Quantity { get; set; }
14    }
15    1 reference
16    public interface IProcessData
17    {
18        1 reference
19        void InsertProducts(List<Product> products);
20        1 reference
21        void DeleteProducts(List<Product> products);
22        1 reference
23        void CalculatePriceByQuantity(Product product);
24    }
25 }
```

We can now divide an interface to multiple interfaces **IProducts** and **ICalucatePrice**, where the responsibility will be single for the classes where implemented. Hence, satisfies the single responsibility principle.

```

1 namespace SRPDP
2 {
3     6 references
4     public class Product
5     {
6         0 references
7         public int ProductId { get; set; }
8         0 references
9         public string? ProductName { get; set; }
10        0 references
11        public decimal ProductPrice { get; set; }
12        0 references
13        public int Quantity { get; set; }
14    }
15
16    1 reference
17    public interface IProducts
18    {
19        1 reference
20        void InsertProducts(List<Product> products);
21        1 reference
22        void DeleteProducts(List<Product> products);
23    }
24
25    1 reference
26    public interface ICalucatePrice
27    {
28        1 reference
29        void CalculatePriceByQuantity(Product product);
30    }
31
32    0 references
33    internal class Program : IProducts, ICalucatePrice
34    {
35        0 references
36        static void Main(string[] args)
37        {
38            Console.WriteLine("Hello, World!");
39        }
40    }
41 }

```

### Open Closed Principle:

- A class or software entity should be open for extension and closed for modification.
- We can achieve this by abstract class and interface implementation in child classes.

In the below example the class **PriceCalculation** does only the RevisedProductPrice.

But, if we required to calculate price as per the location then we need to perform and if codition check which does not satisfy the Open Closed Principle.

```

3 public class Product
4 {
5     0 references
6     public int ProductId { get; set; }
7     0 references
8     public string? ProductName { get; set; }
9     2 references
10    public decimal ProductPrice { get; set; }
11    0 references
12    public int Quantity { get; set; }
13    0 references
14    public string? Location { get; set; }
15 }
16
17 1 reference
18 public class PriceCalculation
19 {
20     private readonly List<Product> _products;
21     0 references
22     public PriceCalculation(List<Product> products)
23     {
24         _products = products;
25     }
26
27     0 references
28     public List<Product> RevisedProductPrice()
29     {
30         List<Product> products = new List<Product>();
31
32         foreach (Product product in _products)
33         {
34             decimal price = product.ProductPrice;
35             product.ProductPrice = price / 10;
36             products.Add(product);
37         }
38         return products;
39     }
40 }

```

We can create and abstract class '**DiscountCalculator**' as below and importantly we can add a abstract method, where the subsequent classes can implement those methods and can change or apply the respective implementations. Hence, the class will be open for extension but closed for modification.

```

namespace OCPDP
{
    19 references
    public class Product
    {
        0 references
        public int ProductId { get; set; }
        0 references
        public string? ProductName { get; set; }
        0 references
        public decimal ProductPrice { get; set; }
        0 references
        public int Quantity { get; set; }
        0 references
        public string? Location { get; set; }
    }

    5 references
    public abstract class DiscountCalculator
    {
        3 references
        protected List<Product> ProductDetails { get; private set; }
        2 references
        public DiscountCalculator(List<Product> productDetails)
        {
            ProductDetails = productDetails;
        }

        2 references
        public abstract List<Product> RevisedProductPrice();
    }

    1 reference
    public class LocationWithinIndia : DiscountCalculator
    {
        0 references
        public LocationWithinIndia(List<Product> indiaLocationProductsDetails): base(indiaLocationProductsDetails)
        {
        }

        1 reference
        public override List<Product> RevisedProductPrice()
        {
            List<Product> products = new List<Product>();
            foreach (Product productsDetail in ProductDetails)
            {
                decimal price = productsDetail.ProductPrice;
                productsDetail.ProductPrice = price / 10;
                products.Add(productsDetail);
            }
            return products;
        }
    }

    1 reference
    public class LocationOutsideIndia : DiscountCalculator
    {
        0 references
        public LocationOutsideIndia(List<Product> outsideLocationProductsDetails): base(outsideLocationProductsDetails)
        {
        }

        1 reference
        public override List<Product> RevisedProductPrice()
        {
            List<Product> products = new List<Product>();
            foreach (Product productsDetail in ProductDetails)
            {
                decimal price = productsDetail.ProductPrice;
                productsDetail.ProductPrice = price / 5;
                products.Add(productsDetail);
            }
            return products;
        }
    }
}

```

### Liskov Substitution Principle :

Object in a program should be replaced with sub class instances. There should not be any change in the functionality.

In the below example we are performing the TotalProductsSold and TotalProductsSoldWithinIndia, but here there to perform different calculations we need to call and create different class instances, which does not satisfy Liskov substitution Principle.

```

11
12 5 references
13 public class TotalProductsSold
14 {
15     private readonly List<Product> _ProductsTotal;
16     2 references
17     public TotalProductsSold(List<Product> products)
18     {
19         _ProductsTotal = products;
20     }
21     1 reference
22     public int CalculateTotal()
23     {
24         return _ProductsTotal.Count;
25     }
26 }
27
28 3 references
29 public class TotalProductsSoldWithinIndia : TotalProductsSold
30 {
31     private readonly List<Product> _ProductsTotal;
32     1 reference
33     public TotalProductsSoldWithinIndia(List<Product> products) : base(products)
34     {
35         _ProductsTotal = products;
36     }
37     1 reference
38     public new int CalculateTotal()
39     {
40         return _ProductsTotal.Count;
41     }
42 }
43
44 0 references
45 internal class Program
46 {
47     0 references
48     static void Main(string[] args)
49     {
50         List<Product> products = new List<Product>()
51         {
52             new Product(ProductId = 1, ProductName = "BlueTooth Mike", Location= "India", ProductPrice = 12000, Quantity =1 ),
53             new Product(ProductId = 1, ProductName = "Mouse", Location= "India", ProductPrice = 12000, Quantity =1 ),
54             new Product(ProductId = 1, ProductName = "PC Monitor", Location= "USA", ProductPrice = 12000, Quantity =1 ),
55             new Product(ProductId = 1, ProductName = "Keyboard", Location= "UK", ProductPrice = 12000, Quantity =1 ),
56         };
57         TotalProductsSold totalProductsSold = new TotalProductsSold(products);
58         totalProductsSold.CalculateTotal();
59         TotalProductsSoldWithinIndia totalProductsSoldWithinIndia = new TotalProductsSoldWithinIndia(products);
60         totalProductsSoldWithinIndia.CalculateTotal();
61     }
62 }

```

So, what we can do is we can create an abstract class Calculator and inherit that into child sub classes and override the abstract Methods of the base class 'CalculateTotal'. In this way we do not break the Liskov Substitution principle.

```

2 7 references
3 public abstract class CalculateTotal
4 {
5     private readonly List<Product> _ProductsTotal;
6
7     2 references
8     public CalculateTotal(List<Product> products)
9     {
10         _ProductsTotal = products;
11     }
12     4 references
13     public abstract int CalculateProducts();
14 }
15
16 2 references
17 public class TotalProductsSold : CalculateTotal
18 {
19     private readonly List<Product> _ProductsTotal;
20     1 reference
21     public TotalProductsSold(List<Product> products) : base(products)
22     {
23         _ProductsTotal = products;
24     }
25
26     3 references
27     public override int CalculateProducts()
28     {
29         return _ProductsTotal.Count;
30     }
31 }
32
33 2 references
34 public class TotalProductsSoldWithinIndia : CalculateTotal
35 {
36     private readonly List<Product> _ProductsTotal;
37     1 reference
38     public TotalProductsSoldWithinIndia(List<Product> products) : base(products)
39     {
40         _ProductsTotal = products;
41     }
42
43     3 references
44     public override int CalculateProducts()
45     {
46         return _ProductsTotal.Select(p => p.Location == "India").Count();
47     }
48 }
49
50 0 references
51 internal class Program
52 {
53     0 references
54     static void Main(string[] args)
55     {
56         List<Product> products = new List<Product>()
57         {
58             new Product{ProductId = 1, ProductName = "BlueTooth Mike", Location= "India", ProductPrice = 12000, Quantity =1 },
59             new Product{ProductId = 1, ProductName = "Mouse", Location= "India", ProductPrice = 12000, Quantity =1 },
60             new Product{ProductId = 1, ProductName = "PC Monitor", Location= "USA", ProductPrice = 12000, Quantity =1 },
61             new Product{ProductId = 1, ProductName = "Keyboard", Location= "UK", ProductPrice = 12000, Quantity =1 },
62         };
63
64         CalculateTotal calculateTotal = new TotalProductsSold(products);
65         calculateTotal.CalculateProducts();
66
67         CalculateTotal totalProductsSoldWithinIndia = new TotalProductsSoldWithinIndia(products);
68         totalProductsSoldWithinIndia.CalculateProducts();
69     }
70 }
71

```

## Interface Segregation Principle:

Every class unnecessarily require all the methods of an interface. We can break those interfaces and move the methods as per requirement. Means breaking a big interfaces into small multiple interfaces

The below example has three methods in the interface, where when implemented by two classes which must have unnecessarily require the all the methods to be implemented( when few are not required).

```

SQL Server Object Explorer
3  public interface IProductPriceCalculator
4  {
5      2 references
6      decimal ServiceTax();
7      2 references
8      decimal CalculteVAT();
9      2 references
10     decimal CalculteGST();
11 }
12
13 0 references
14 public class CountryIndiaCalculation : IProductPriceCalculator
15 {
16     1 reference
17     public decimal CalculteGST()
18     {
19         return 200 % 10;
20     }
21
22     1 reference
23     public decimal CalculteVAT()
24     {
25         return -1;
26     }
27
28     1 reference
29     public decimal ServiceTax()
30     {
31         throw new NotImplementedException();
32     }
33 }
34
35 0 references
36 public class CountryOutsideIndiaCalculation : IProductPriceCalculator
37 {
38     1 reference
39     public decimal CalculteGST()
40     {
41         return -1;
42     }
43
44     1 reference
45     public decimal CalculteVAT()
46     {
47         return 200 % 4;
48     }
49
50     1 reference
51     public decimal ServiceTax()
52     {
53         return 2000 % 4;
54     }
55 }

```

Interface Segregation Principle what it does is we need to create multiple interfaces for different method operations according to the requirement. So, that unnecessary we not need to implement the method which are not required and the class where requirement we can implement by inheriting multiple interfaces.

```

4 2 references
5 public interface IProductPriceCommonCalculator
6 {
7     2 references
8     decimal ServiceTax();
9 }
10 1 reference
11 public interface IProductPriceCommonCalculatorWithinIndia
12 {
13     1 reference
14     decimal CalculateGST();
15 }
16 1 reference
17 public interface IProductPriceCalculatorOutsideIndia
18 {
19     1 reference
20     decimal CalculateVAT();
21 }
22 0 references
23 public class CountryIndiaCalculation : IProductPriceCommonCalculator, IProductPriceCommonCalculatorWithinIndia
24 {
25     1 reference
26     public decimal CalculateGST()
27     {
28         return 200 % 10;
29     }
30     1 reference
31     public decimal ServiceTax()
32     {
33         throw new NotImplementedException();
34     }
35 }
36 0 references
37 public class CountryOutsideIndiaCalculation : IProductPriceCommonCalculator, IProductPriceCalculatorOutsideIndia
38 {
39     1 reference
40     public decimal CalculateVAT()
41     {
42         return 200 % 4;
43     }
44     1 reference
45     public decimal ServiceTax()
46     {
47         return 2000 % 4;
48     }
49 }

```

### Dependency inversion principle:

Dependency inversion principle states that “high level module” should not depend on low level module and both should depend on abstraction.

Here, we created instance of the DataAccessLayer and where BusinessLogicLayer “High level module” is dependent on the DataAccessLayer “Low level module”. Hence, dependency inversion principle fails.



```

3 public class BusinessLogicLayer
4 {
5     private readonly DataAccessLayer _dataLayer;
6     0 references
7     public BusinessLogicLayer(DataAccessLayer dataLayer)
8     {
9         _dataLayer = dataLayer;
10    }
11
12    0 references
13    public void SaveData(int Id)
14    {
15        _dataLayer.SaveData(Id);
16    }
17
18    2 references
19    public class DataAccessLayer
20    {
21        1 reference
22        public void SaveData(int Id)
23        {
24            //Save Data
25        }
26    }

```

So, what we can do is create an interface 'IRepositoryLayer' with a Save method. And now create an instance of the interface in the Business Layer and here we could call the interface Save method which would call the DataAccessLayer and we are not directly calling the 'DataAccessLayer' method 'SaveData' directly. This satisfies the Dependency inversion principle.

```

3 public interface IRepositoryLayer
4 {
5     2 references
6     void SaveData(int Id);
7 }
8
9 1 reference
10 public class BusinessLogicLayer
11 {
12     private readonly IRepositoryLayer _iRepositoryLayer;
13     0 references
14     public BusinessLogicLayer(IRepositoryLayer iRepositoryLayer)
15     {
16         _iRepositoryLayer = iRepositoryLayer;
17     }
18
19     0 references
20     public void SaveData(int Id)
21     {
22         _iRepositoryLayer.SaveData(Id);
23     }
24 }
25
26 0 references
27 public class DataAccessLayer: IRepositoryLayer
28 {
29     2 references
30     public void SaveData(int Id)
31     {
32         //Save Data
33     }
34 }

```