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 CalculatePriceByQantity would need to perform a different responsibilty. Hence, single responsibility principle does not satisfy.

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

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

```
namespace SRPDP
{
    6 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; }
    }


    1 reference
    public interface IProducts
    {
        1 reference
        void InsertProducts(List<Product> products);
        1 reference
        void DeleteProducts(List<Product> products);
    }

    1 reference
    public interface ICalucatePrice
    {
        1 reference
        void CalculatePriceByQuantity(Product product);
    }

    0 references
    internal class Program : IProducts, ICalucatePrice
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Hello, World!");
```

No issues found

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.

```
public class Product
{
    public int ProductId { get; set; }
    public string? ProductName { get; set; }
    public decimal ProductPrice { get; set; }
    public int Quantity { get; set; }
    public string? Location { get; set; }
}

public class PriceCalculation
{
    private readonly List<Product> _products;
    public PriceCalculation(List<Product> products)
    {
        _products = products;
    }

    public List<Product> RevisedProductPrice()
    {
        List<Product> products = new List<Product>();

        foreach (Product product in _products)
        {
            decimal price = product.ProductPrice;

            product.ProductPrice = price / 10;

            products.Add(product);
        }
        return products;
    }
```

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.

Program.cs — OCPDP.LocationWithinIndia — RevisedProductPrice()

```
namespace OCPDP
{
    19 references
    public class Product
    {
        0 references
        public int ProductId { get; set; }
        0 references
        public string? ProductName { get; set; }
        6 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();
    }
```

OCPDP.LocationWithinIndia — 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;
        }
    }
    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.

```
5 references
public class TotalProdctsSold
{
    private readonly List<Product> _ProductsTotal;
    2 references
    public TotalProdctsSold(List<Product> products)
    {
        _ProductsTotal = products;
    }

    1 reference
    public int CalculateTotal()
    {
        return _ProductsTotal.Count;
    }
}

3 references
public class TotalProdctsSoldWithinIndia : TotalProdctsSold
{
    private readonly List<Product> _ProductsTotal;
    1 reference
    public TotalProdctsSoldWithinIndia(List<Product> products) : base(products)
    {
        _ProductsTotal = products;
    }

    1 reference
    public new int  CalculateTotal()
    {
        return _ProductsTotal.Count;
    }
}
```

```
0 references
internal class Program
{
    0 references
    static void Main(string[] args)
    {
        List<Product> products = new List<Product>()
        {
            new Product{ProductId = 1, ProductName = "BlueTooth Mike", Location= "India", ProductPrice = 12000, Quantity =1 },
            new Product{ProductId = 1, ProductName = "Mouse", Location= "India", ProductPrice = 12000, Quantity =1 },
            new Product{ProductId = 1, ProductName = "PC Monitor", Location= "USA", ProductPrice = 12000, Quantity =1 },
            new Product{ProductId = 1, ProductName = "Keyboard", Location= "UK", ProductPrice = 12000, Quantity =1 },

        };

        TotalProdctsSold totalProdctsSold = new TotalProdctsSold(products);
        totalProdctsSold.CalculateTotal();

        TotalProdctsSoldWithinIndia totalProdctsSoldWithinIndia = new TotalProdctsSoldWithinIndia(products);
        totalProdctsSoldWithinIndia.CalculateTotal();

    }
}
```

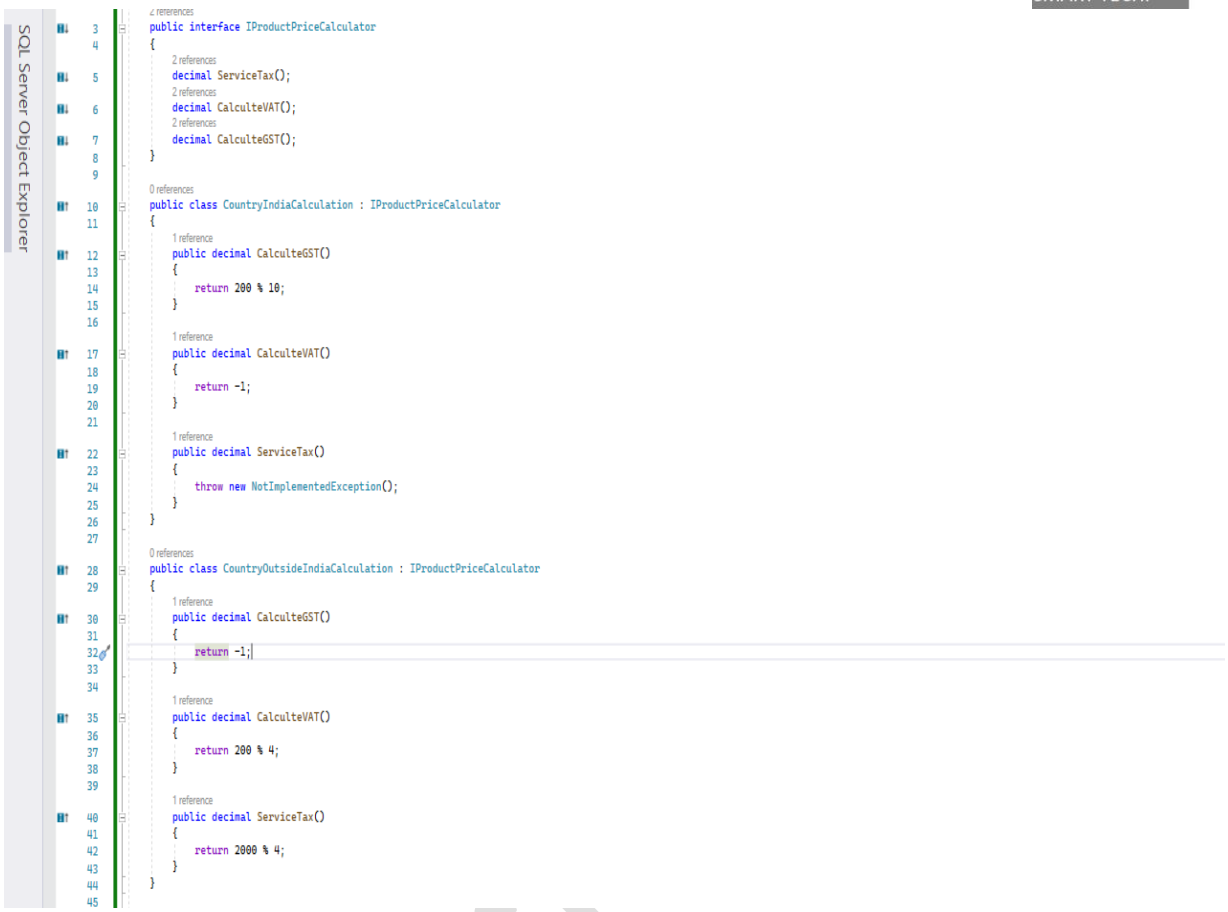
So, what we can do is we can create and abstract class Calculator and inherit that into child sub classes and override the abstract Methods of the base class ‘CalculateTotal’. In this way we do not break the Liskov Substitution principle.

```
7 references
public abstract class CalculateTotal
{
    private readonly List<Product> _ProductsTotal;

    2 references
    public CalculateTotal(List<Product> products)
    {
        _ProductsTotal = products;
    }
    4 references
    public abstract int CalculateProducts();
}

2 references
public class TotalProdctsSold : CalculateTotal
{
    private readonly List<Product> _ProductsTotal;
    1 reference
    public TotalProdctsSold(List<Product> products) : base(products)
    {
        _ProductsTotal = products;
    }

    3 references
    public override int CalculateProducts()
    {
        return _ProductsTotal.Count;
    }
}

2 references
public class TotalProdctsSoldWithinIndia : CalculateTotal
{
    private readonly List<Product> _ProductsTotal;
    1 reference
    public TotalProdctsSoldWithinIndia(List<Product> products) : base(products)
    {
        _ProductsTotal = products;
    }

    3 references
    public override int CalculateProducts()
    {
        return _ProductsTotal.Select(p=> p.Location == "India").Count();
    }
}

0 references
internal class Program
{
    0 references
    static void Main(string[] args)
    {
        List<Product> products = new List<Product>()
        {
            new Product{ProductId = 1, ProductName = "BlueTooth Mike", Location= "India", ProductPrice = 12000, Quantity =1 },
            new Product{ProductId = 1, ProductName = "Mouse", Location= "India", ProductPrice = 12000, Quantity =1 },
            new Product{ProductId = 1, ProductName = "PC Monitor", Location= "USA", ProductPrice = 12000, Quantity =1 },
            new Product{ProductId = 1, ProductName = "Keyboard", Location= "UK", ProductPrice = 12000, Quantity =1 },

        };

        CalculateTotal calclateTotal = new TotalProdctsSold(products);
        calclateTotal.CalculateProducts();

        CalculateTotal totalProdctsSoldWithinIndia = new TotalProdctsSoldWithinIndia(products);
        totalProdctsSoldWithinIndia.CalculateProducts();

    }
}
```

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).



```
4 references
public interface IProductPriceCalculator
{
    2 references
    decimal ServiceTax();
    2 references
    decimal CalculteVAT();
    2 references
    decimal CalculteGST();
}

0 references
public class CountryIndiaCalculation : IProductPriceCalculator
{
    1 reference
    public decimal CalculteGST()
    {
        return 200 % 10;
    }

    1 reference
    public decimal CalculteVAT()
    {
        return -1;
    }

    1 reference
    public decimal ServiceTax()
    {
        throw new NotImplementedException();
    }
}

0 references
public class CountryOutsideIndiaCalculation : IProductPriceCalculator
{
    1 reference
    public decimal CalculteGST()
    {
        return -1;
    }

    1 reference
    public decimal CalculteVAT()
    {
        return 200 % 4;
    }

    1 reference
    public decimal ServiceTax()
    {
        return 2000 % 4;
    }
}
```

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.

```
2 references
public interface IProductPriceCommonCalculator
{
    2 references
    decimal ServiceTax();
}

1 reference
public interface IProductPriceCommonCalculatorWithinIndia
{
    1 reference
    decimal CalculteGST();
}

1 reference
public interface IProductPriceCalculatorOutsideIndia
{
    1 reference
    decimal CalculteVAT();
}
0 references
public class CountryIndiaCalculation : IProductPriceCommonCalculator, IProductPriceCommonCalculatorWithinIndia
{
    1 reference
    public decimal CalculteGST()
    {
        return 200 % 10;
    }

    1 reference
    public decimal ServiceTax()
    {
        throw new NotImplementedException();
    }
}

0 references
public class CountryOutsideIndiaCalculation : IProductPriceCommonCalculator, IProductPriceCalculatorOutsideIndia
{
    1 reference
    public decimal CalculteVAT()
    {
        return 200 % 4;
    }

    1 reference
    public decimal ServiceTax()
    {
        return 2000 % 4;
    }
}
```

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.

```
public class BusinessLogicLayer
{
    private readonly DataAccessLayer _dataLayer;
    0 references
    public BusinessLogicLayer(DataAccessLayer dataLayer)
    {
        _dataLayer = dataLayer;
    }

    0 references
    public void SaveData(int Id)
    {
        _dataLayer.SaveData(Id);
    }

}

2 references
public class DataAccessLayer
{
    1 reference
    public void SaveData(int Id)
    {
        //Save Data
    }
}
```

So, what we can do is create an interface 'IRepsositoryLayer' 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.

```
public interface IRepositoryLayer
{
    2 references
    void SaveData(int Id);
}

1 reference
public class BusinessLogicLayer
{
    private readonly IRepositoryLayer _iRepositoryLayer;
    0 references
    public BusinessLogicLayer(IRepositoryLayer iRepositoryLayer)
    {
        _iRepositoryLayer = iRepositoryLayer;
    }

    0 references
    public void SaveData(int Id)
    {
        _iRepositoryLayer.SaveData(Id);
    }

}

0 references
public class DataAccessLayer: IRepositoryLayer
{
    2 references
    public void SaveData(int Id)
    {
        //Save Data
    }
}
```