Solid Principles (Seminar)

Reference

- https://www.codeproject.com/Articles/703634/SOLID-Architecture-Principles-Using-Simple-Csharp
- https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

What is SOLID?

SOLID are five basic principles which help to create good software architecture. SOLID is an acronym where:-

S stands for SRP (Single responsibility principle

O stands for OCP (Open closed principle)

L stands for LSP (Liskov substitution principle)

I stands for ISP (Interface segregation principle)

D stands for DIP (Dependency inversion principle)

“S”- SRP (Single responsibility principle)

```
class Customer
    {
        public void Add()
        {
            try
            {
                // Database code goes here
            }
            catch (Exception ex)
            {
                System.IO.File.WriteAllText(@"c:\Error.txt", ex.ToString());
            }
        }
    }
```

“S”- SRP (Single responsibility principle)

It is not the load but the OVERLOAD that kills :- Spanish Proverb

Too many responsibilities on a single thing can cause problems.

“S”- SRP (Single responsibility principle)

KISS: Keep It Simple Stupid

“S”- SRP (Single responsibility principle)

```
class FileLogger
    {
        public void Handle(string error)
        {
            System.IO.File.WriteAllText(@"c:\Error.txt", error);
        }
    }
```

“S”- SRP (Single responsibility principle)

```
class Customer
    {
        private FileLogger obj = new FileLogger();
        publicvirtual void Add()
        {
            try
            {
                // Database code goes here
            }
            catch (Exception ex)
            {
                obj.Handle(ex.ToString());
            }
        }
    }
```

"O" - Open closed principle

DID YOU KNOW THESE IT GUYS COPIED MY IDEA OF EXTENSIBILITY!
EXTENDED

"O" - Open closed principle

```
class Customer
{
    private int _CustType;

    public int CustType
    {
      get { return _CustType; }
      set { _CustType = value; }
    }

    public double getDiscount(double TotalSales)
    {
        if (_CustType == 1)
        {
          return TotalSales - 100;
        }
        else
        {
          return TotalSales - 50;
        }
    }
}
```

“O” - Open closed principle

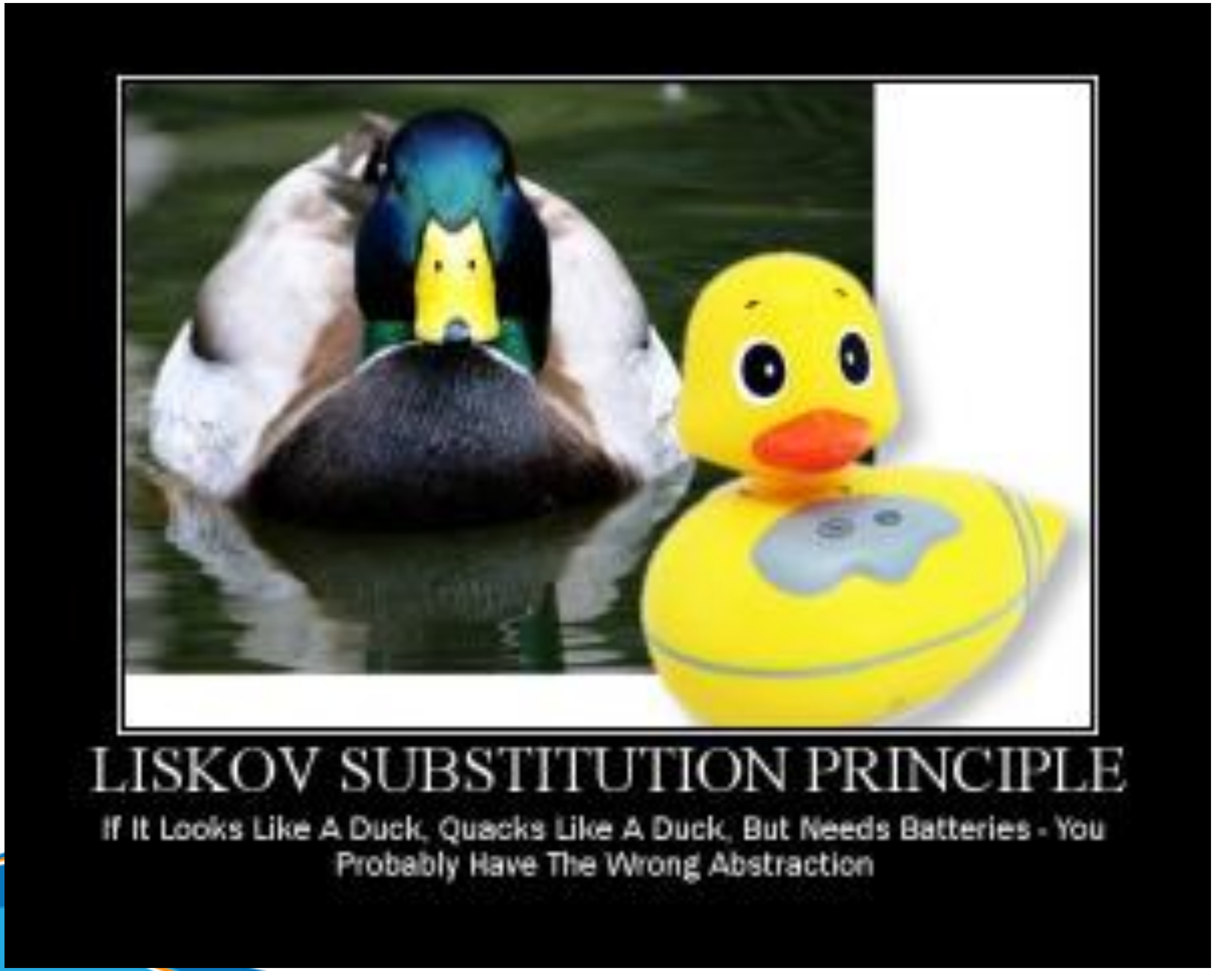
```
class Customer
{
        public virtual double getDiscount(double
TotalSales)
        {
            return TotalSales;
        }
}
```

“O” - Open closed principle

```
class SilverCustomer : Customer
  {
    public override double getDiscount(double TotalSales)
    {
      return base.getDiscount(TotalSales) - 50;
    }
  }

 class goldCustomer : SilverCustomer
  {
    public override double getDiscount(double TotalSales)
    {
      return base.getDiscount(TotalSales) - 100;
    }
  }
```

“L”- LSP (Liskov substitution principle)


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

“L”- LSP (Liskov substitution principle)

```
class Enquiry : Customer
    {
        public override double getDiscount(double TotalSales)
        {
            return base.getDiscount(TotalSales) - 5;
        }

        public override void Add()
        {
            throw new Exception("Not allowed");
        }
    }
```

“L”- LSP (Liskov substitution principle)

```
interface IDiscount
{
	double getDiscount(double TotalSales);
}


interface IDatabase
{
	void Add();
}
```

“L”- LSP (Liskov substitution principle)

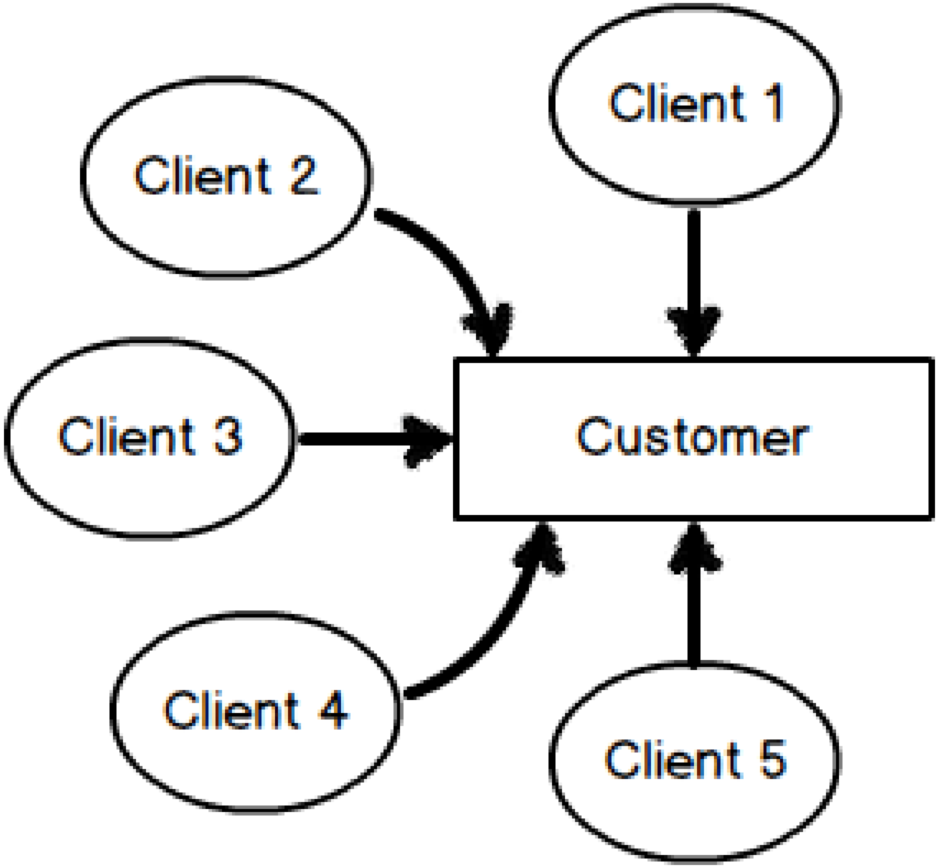
```
class Enquiry : IDiscount
    {
        public  double getDiscount(double TotalSales)
        {
            return TotalSales - 5;
        }
    }
```

“L”- LSP (Liskov substitution principle)

```
class Customer : IDiscount, IDatabase
  {
    private MyException obj = new MyException();
    public virtual void Add()
    {
      try
      {
        // Database code goes here
      }
      catch (Exception ex)
      {
        obj.Handle(ex.Message.ToString());
      }
    }

    public virtual double getDiscount(double TotalSales)
    {
      return TotalSales;
    }
  }
```

“I” - ISP (Interface Segregation principle)



“I” - ISP (Interface Segregation principle)

```
interface IDatabase
{
        void Add(); // old client are happy with these.
        void Read(); // Added for new clients.
}
```

“I” - ISP (Interface Segregation principle)

```
interface IDatabaseV1 : IDatabase // Gets the Add method
{
        void Read();
}
class CustomerwithRead : IDatabase, IDatabaseV1
{
        public void Add()
        {
                Customer obj = new Customer();
                obj.Add();
        }

        public void Read()
        {
                // Implements  logic for read
        }
}
```

"I" - ISP (Interface Segregation principle)

IDatabase i = new Customer(); // 1000 happy old clients not touched

i.Add();

IDatabaseV1 iv1 = new CustomerWithread(); // new clients

iv1.Read();

“D”- Dependency inversion principle

```
class Customer
  {
    private FileLogger obj = new FileLogger();
    public virtual void Add()
    {
      try
      {
        // Database code goes here
      }
      catch (Exception ex)
      {
        obj.Handle(ex.ToString());
      }
    }
  }
```

“D”- Dependency inversion principle

```
interface ILogger
{
    void Handle(string error);
}

class FileLogger : ILogger
  {
    public void Handle(string error)
    {
      System.IO.File.WriteAllText(@"c:\Error.txt", error);
    }
  }
```

“D”- Dependency inversion principle

```
class EverViewerLogger : ILogger
  {
    public void Handle(string error)
    {
      // log errors to event viewer
    }
  }
class EmailLogger : ILogger
 {
   public void Handle(string error)
   {
     // send errors in email
   }
 }
```

“D”- Dependency inversion principle

```
class Customer : IDiscount, IDatabase
 {
        private Ilogger obj;
        public Customer(ILogger i)
        {
            obj = i;
        }
}

IDatabase i = new Customer(new EmailLogger());
```