SOLID architecture principles using simple C# examples

Contents

S stands for SRP (Single responsibility principle):- A class should take care of only one responsibility.
O stands for OCP (Open closed principle):- Extension should be preferred over modification.
L stands for LSP (Liskov substitution principle):- A parent class object should be able to refer child objects seamlessly during runtime polymorphism.
I stands for ISP (Interface segregation principle):- Client should not be forced to use a interface if it does not need it.
D stands for DIP (Dependency inversion principle) :- High level modules should not depend on low level modules but should depend on abstraction.

Understanding “S”- SRP (Single responsibility principle)

The best way to understand SOLID is by understanding what problem it tries to solve. Have a look at the code below, can you guess what the problem is ?(You are not going to get BEER to guess it J , because it’s too simple).

OK, let me give a HINT look at the catch block code.

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

The above customer class is doing things **WHICH HE IS NOT SUPPOSED TO DO**. Customer class should do customer datavalidations, call the customer data access layer etc , but if you see the catch block closely it also doing LOGGING activity. In simple words its over loaded with lot of responsibility.

So tomorrow if add a new logger like event viewer I need to go and change the “Customer”class, that’s very ODD.

It’s like if “JOHN” has a problem why do I need to check “BOB”.

This also reminds me of the famous swiss knife. If one of them needs to be changed the whole set needs to be disturbed. No offense I am great fan of swiss knifes.

But if we can have each of those items separated its simple, easy to maintain and one change does not affect the other. The same principle also applies to classes and objects in software architecture.

So SRP says that a class should have only one responsibility and not multiple.So if we apply SRP we can move that logging activity to some other class who will only look after logging activities.

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

Now customer class can happily delegate the logging activity to the “FileLogger” class and he can concentrate on customer related activities.

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

Now architecture thought process is an evolution. For some people who are seniors looking at above SRP example can contradict that even the try catch should not be handled by the customer class because that is not his work.

Yes, we can create a global error handler must be in theGlobal.asax file , assuming you are using ASP.NET and handle the errors in those section and make the customer class completely free.

So I will leave how far you can go and make this solution better but for now I want to keep this simple and let your thoughts have the freedom to take it to a great level.

Below is a great comment which talks about how we can take this SRP example to the next level.
http://www.codeproject.com/Articles/703634/SOLID-architecture-principles-using-simple-Csharp?msg=4729987#xx4729987xx

Understanding “O” - Open closed principle

Let’s continue with our same customer class example. I have added a simple customer type property to the class. This property decided if this is a “Gold” ora “Silver” customer.

Depending on the same it calculates discount. Have a look at the “getDiscount” function which returns discount accordingly. 1 for Gold customer and 2 for Silver customer.

Guess, what’s the problem with the below code. Hahaha, looks like this article will make you a GUESS champion .

Ok, also let me add a HINT, look at the “IF” condition in the “getDiscount” function.

```
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;
                }
        }
}
```

The problem is if we add a new customer type we need to go and add one more “IF” condition in the “getDiscount” function, in other words we need to change the customer class.

If we are changing the customer class again and again, we need to ensure that the previous conditions with new one’s are tested again , existing client’s which are referencing this class are working properly as before.

In other words we are “MODIFYING” the current customer code for every change and every time we modify we need to ensure that all the previous functionalities and connected client are working as before.

How about rather than “MODIFYING” we go for “EXTENSION”. In other words every time a new customer type needs to be added we create a new class as shown in the below. So whatever is the current code they are untouched and we just need to test and check the new classes.

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

```
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;
        }
    }
```

Putting in simple words the "Customer" class is now closed for any new modification but it's open for extensions when new customer types are added to the project.

Understanding “L”- LSP (Liskov substitution principle)

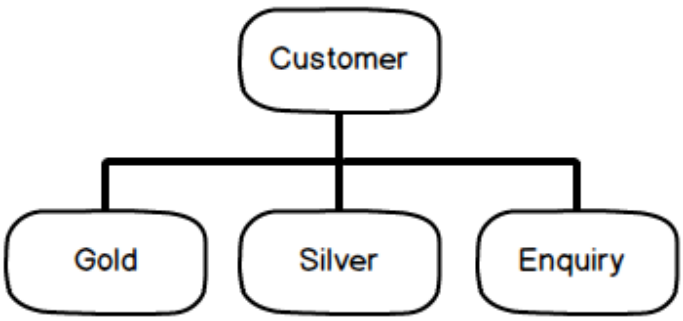
Let’s continue with the same customer. Let’s say our system wants to calculate discounts for Enquiries. Now Enquiries are not actual customer’s they are just leads. Because they are just leads we do not want to save them to database for now.

So we create a new class called as Enquiry which inherits from the “Customer” class. We provide some discounts to the enquiry so that they can be converted to actual customers and we override the “Add’ method with an exception so that no one can add an Enquiry to the database.

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

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

If you visualize the current customer inheritance hierarchy it looks something as shown below. In other word “Customer” is the parent class with “Gold” , “Silver” and “Enquiry” as child classes.



So as per polymorphism rule my parent “Customer” class object can point to any of it child class objects i.e. “Gold”, “Silver” or “Enquiry” during runtime without any issues.

So for instance in the below code you can see I have created a list collection of “Customer” and thanks to polymorphism I can add “Silver” , “Gold” and “Enquiry” customer to the “Customer” collection without any issues.

Thanks to polymorphism I can also browse the “Customer” list using the parent customer object and invoke the “Add” method as shown in the below code.

Now again let me tickle your brains, there is a slight problem here, THINK, THINK THINK.

HINT: -Watch when the Enquiry object is browsed and invoked in the “FOR EACH” loop.

```
List<Customer> Customers = new List<Customer>();
Customers.Add(new SilverCustomer());
Customers.Add(new goldCustomer());
Customers.Add(new Enquiry());

 foreach (Customer o in Customers)
 {
                o.Add();
 }
}
```

As per the inheritance hierarchy the "Customer" object can point to any one of its child objects and we do not expect any unusual behavior.

But when "Add" method of the "Enquiry" object is invoked it leads to below error because our "Equiry" object does save enquiries to database as they are not actual customers.

```
}

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

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

Exception was unhandled

Not allowed

Troubleshooting tips:

Get general help for this exception.

Search for more Help Online...

Exception settings:

Break when this exception type is

Now read the below paragraph properly to understand the problem. If you do not understand the below paragraph read it twice☺..

In other words the "Enquiry" has discount calculation , it looks like a "Customer" but **IT IS NOT A CUSTOMER.** So the parent cannot replace the child object seamlessly. In other words "Customer" is not the actual parent for the "Enquiry"class. "Enquiry" is a different entity altogether.

So LISKOV principle says the parent should easily replace the child object. So to implement LISKOV we need to create two interfaces one is for discount and other for database as shown below.

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


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

Now the "Enquiry" class will only implement "IDiscount" as he not interested in the "Add" method.

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

While the "Customer" class will implement both "IDiscount" as well as "IDatabase" as it also wants to persist the customer to the database.

```
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;
        }
    }
```

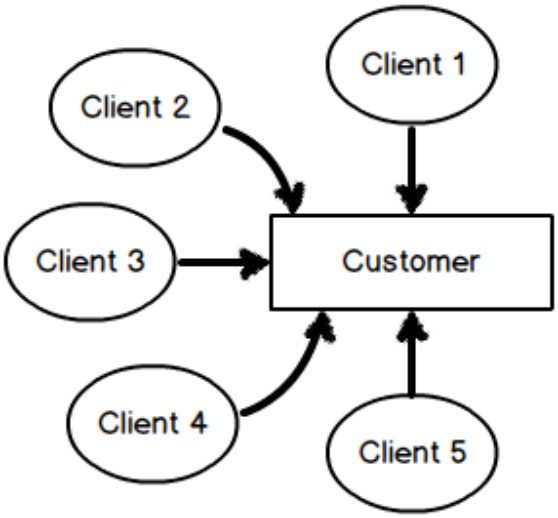
Now there is no confusion, we can create a list of "Idatabase" interface and add the relevant classes to it. In case we make a mistake of adding "Enquiry" class to the list compiler would complain as shown in the below code snippet.

```
List<IDatabase> Customers = new List<IDatabase>();
Customers.Add(new SilverCustomer());
Customers.Add(new goldCustomer());
Customers.Add(new Enquiry());

foreach (IDatabase o in Customers)
{
    o.Add();
}
```

Understanding “I” - ISP (Interface Segregation principle)

Now assume that our customer class has become a SUPER HIT component and it’s consumed across 1000 clients and they are very happy using the customer class.



Now let’s say some new clients come up with a demand saying that we also want a method which will help us to “Read” customer data. So developers who are highly enthusiastic would like to change the “IDatabase” interfaceas shown below.

But by doing so we have done something terrible, can you guess ?

HINT: - Think about the effect of this change on the above image.

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

If you visualize the new requirement which has come up, you have two kinds of client’s: -

- Who want’s just use “Add” method.
- The other who wants to use “Add” + “Read”.

Now by changing the current interface you are doing an awful thing, disturbing the 1000 satisfied current client’s , even when they are not interested in the “Read” method. You are forcing them to use the “Read” method.

So a better approach would be to keep existing clients in their own sweet world and the serve the new client’s separately.

So the better solution would be to create a new interface rather than updating the current interface. So we can keep the current interface "IDatabase" as it is and add a new interface "IDatabaseV1" with the "Read" method the "V1" stands for version 1.

```
interface IDatabaseV1 : IDatabase // Gets the Add method
{
Void Read();
}
```

You can now create fresh classes which implement "Read" method and satisfy demands of your new clients and your old clients stay untouched and happy with the old interface which does not have "Read" method.

```
class CustomerwithRead : IDatabase, IDatabaseV1
    {

public void Add()
{
         Customer obj = new Customer();
Obj.Add();
}
         Public void Read()
         {
         // Implements  logic for read
}
    }
```

So the old clients will continue using the "IDatabase" interface while new client can use "IDatabaseV1" interface.

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

IDatabaseV1 iv1 = new CustomerWithread(); // new clients
Iv1.Read();
```

Understanding “D”- Dependency inversion principle

In our customer class if you remember we had created a logger class to satisfy SRP. Down the line let’s say new Logger flavor classes are created.

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

Just to control things we create a common interface and using this common interface new logger flavors will be created.

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

Below are three logger flavors and more can be added down the line.

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

Now depending on configuration settings different logger classes will used at given moment. So to achieve the same we have kept a simple IF condition which decides which logger class to be used, see the below code.

QUIZ time, what is the problem here.

HINT: - Watch the CATCH block code.

```
class Customer : IDiscount, IDatabase
    {
        private IException obj;

public virtual void Add(int Exhandle)
        {
            try
            {
                // Database code goes here
            }
            catch (Exception ex)
            {
                if (Exhandle == 1)
                {
                    obj = new MyException();
                }
                else
                {
                    obj = new EmailException();
                }
                obj.Handle(ex.Message.ToString());
            }
        }
```

The above code is again violating SRP but this time the aspect is different ,its about deciding which objects should be created. Now it's not the work of "Customer" object to decide which instances to be created , he should be concentrating only on Customer class related functionalities.

If you watch closely the biggest problem is the "NEW" keyword. He is taking extra responsibilities of which object needs to be created.

So if we INVERT / DELEGATE this responsibility to someone else rather the customer class doing it that would really solve the problem to a certain extent.

So here's the modified code with INVERSION implemented. We have opened the constructor mouth and we expect someone else to pass the object rather than the customer class doing it. So now it's the responsibility of the client who is consuming the customer object to decide which Logger class to inject.

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

So now the client will inject the Logger object and the customer object is now free from those IF condition which decide which logger class to inject. This is the Last principle in SOLID Dependency Inversion principle.

Customer class has delegated the dependent object creation to client consuming it thus making the customer class concentrate on his work.

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