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)

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. 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.

class FileLogger

{

public void Handle(string error)

{

System.IO.File.WriteAllText(@"c:\Error.txt", error);

}

}

class Customer

{

private FileLogger obj = new FileLogger();

publicvirtual void Add()

{

try

{

*// Database code goes here*

}

catch (Exception ex)

{

obj.Handle(ex.ToString());

}

}

}

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.

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;

}

}

}

class Customer

{

public virtual double getDiscount(double TotalSales)

{

return TotalSales;

}

}

class SilverCustomer : Customer

{

public override double getDiscount(double TotalSales)

{

return base.getDiscount(TotalSales) - 50;

}

}

Hide   Copy Code

class goldCustomer : SilverCustomer

{

public override double getDiscount(double TotalSales)

{

return base.getDiscount(TotalSales) - 100;

}

}

## 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");

}

}

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();

}

}

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.

interface IDiscount

{

double getDiscount(double TotalSales);

}

interface IDatabase

{

void Add();

}

class Enquiry : IDiscount

{

public double getDiscount(double TotalSales)

{

return TotalSales - 5;

}

}

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;

}

}

## Understanding “I” - ISP (Interface Segregation principle)

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.

interface IDatabase

{

void Add(); *// old client are happy with these.*

void Read(); *// Added for new clients.*

}

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.

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*

}

}

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

class Customer

{

private FileLogger obj = new FileLogger();

public virtual void Add()

{

try

{

*// Database code goes here*

}

catch (Exception ex)

{

obj.Handle(ex.ToString());

}

}

}

interface ILogger

{

void Handle(string error);

}

class FileLogger : ILogger

{

public void Handle(string error)

{

System.IO.File.WriteAllText(@"c:\Error.txt", error);

}

}

Hide   Copy Code

class EverViewerLogger : ILogger

{

public void Handle(string error)

{

*// log errors to event viewer*

}

}

Hide   Copy Code

class EmailLogger : ILogger

{

public void Handle(string error)

{

*// send errors in email*

}

}

Hide   Copy 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.

Hide   Copy Code

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.

Hide   Copy Code

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

**Unit of Work design** pattern does two important things: first it maintains in-memory updates and second it sends these in-memory updates as one transaction to the database.

So to achieve the above goals it goes through two steps:

* It maintains lists of business objects in-memory which have been changed (inserted, updated, or deleted) during a transaction.
* Once the transaction is completed, all these updates are sent as one **big unit of work** to be persisted physically in a database in one **go**.

1 customer CRUD = 1 unit of work

### **Step 1: Create a generalized interface (IEntity) for business objects**

At the end of the day a unit of work is nothing but a collection which maintains and track changes on the business objects.

So the first step is to create a generalized interface called IEntity which represents a business object in our project.

This IEntity interface will have an ID property and methods (insert, update, delete, and load) which will help us to do the CRUD operation on the business object. The ID property is a unique number which helps us uniquely identify the record in a database.

Hide   Copy Code

public interface IEntity

{

int Id { set; get; }

void Insert();

void Update();

List<IEntity> Load();

}

### **Step 2: Implement the IEntity interface**

The next step is to implement the IEntity in all our business objects

public class Customer : IEntity

{

private int \_CustomerCode = 0;

public int Id

{

get { return \_CustomerCode; }

set { \_CustomerCode = value; }

}

private string \_CustomerName = "";

public string CustomerName

{

get { return \_CustomerName; }

set { \_CustomerName = value; }

}

public void Insert()

{

DataAccess obj = new DataAccess();

obj.InsertCustomer(\_CustomerCode, CustomerName);

}

public List<IEntity> Load()

{

DataAccess obj = new DataAccess();

Customer o = new Customer();

SqlDataReader ds = obj.GetCustomer(Id);

while (ds.Read())

{

o.CustomerName = ds["CustomerName"].ToString();

}

List<IEntity> Li = (new List<Customer>()).ToList<IEntity>();

Li.Add((IEntity) o);

return Li;

}

public void Update()

{

DataAccess obj = new DataAccess();

obj.UpdateCustomer(\_CustomerCode, CustomerName);

}

}

### **Step 3: Create the unit of work collection**

The next step is to create the unit of work collection class.

public class SimpleExampleUOW

{

private List<IEntity> Changed = new List<IEntity>();

private List<IEntity> New = new List<IEntity>();

public void Add(IEntity obj)

{

New.Add(obj);

}

public void Committ()

{

using (TransactionScope scope = new TransactionScope())

{

foreach (IEntity o in Changed)

{

o.Update();

}

foreach (IEntity o in New)

{

o.Insert();

}

scope.Complete();

}

}

public void Load(IEntity o)

{

Changed = o.Load() as List<IEntity>;

}

}

### **Step 4: See it working**

On the client side we can create the Customer object, add business objects in memory, and finally all these changes are sent in an atomic manner to the physical database by calling the Commit method.

Hide   Copy Code

Customer Customerobj = new Customer();*// record 1 Customer*

Customerobj.Id = 1000;

Customerobj.CustomerName = "shiv";

Supplier SupplierObj = new Supplier(); *// Record 2 Supplier*

Supplierobj.Id = 2000;

Supplierobj.SupplierName = "xxxx";

SimpleExampleUOW UowObj = new SimpleExampleUOW();

UowObj.Add(Customerobj); *// record 1 added to inmemory*

UowObj.Add(Supplierobj); *// record 1 added to inmemory*

UowObj.Committ(); *// The full inmemory collection is sent for final committ*

Observer Pattern

Consider an online electronics store which has a huge inventory and they keep on updating it. **The store wants to update all its users/customers whenever any product arrives in the store**. **The online electronic store is going to be the subject. Whenever the subject would have any addition in its inventory, the observers (customers/users) who have subscribed to store notifications would be notified through email.**

interface ISubject

{

void Subscribe(Observer observer);

void Unsubscribe(Observer observer);

void Notify();

}

interface IObserver

{

void Update();

}

public class Subject:ISubject

{

private List<Observer> observers = new List<Observer>();

private int \_int;

public int Inventory

{

get

{

return \_int;

}

set

{

*// Just to make sure that if there is an increase in inventory then only we are notifying*

the observers.

if (value > \_int)

Notify();

\_int = value;

}

}

public void Subscribe(Observer observer)

{

observers.Add(observer);

}

public void Unsubscribe(Observer observer)

{

observers.Remove(observer);

}

public void Notify()

{

observers.ForEach(x => x.Update());

}

}

public class Observer:IObserver

{

public string ObserverName { get;private set; }

public Observer(string name)

{

this.ObserverName = name;

}

public void Update()

{

Console.WriteLine("{0}: A new product has arrived at the

store",this.ObserverName);

}

}

Dispose Pattern

When we want to clear resource after uses of any object, we can implement this pattern. In finalize dispose pattern we have to implement one interface called IDisposable in class where we want to implement.

public class Garbage : IDisposable  
    {  
        public String name = String.Empty;  
        public SqlConnection con = null;  
        public Garbage()  
        {  
            name = "This is managable resource";  
            con = new SqlConnection();  
        }  
        ~Garbage()  
        {  
            Dispose(false);

}  
        protected virtual void Dispose(bool disposing)  
        {  
            if (disposing)  
            {  
                //Free managed resource here  
                name = null;  
            }  
            //Free unmanaged resource here  
            con = null;  
            Console.WriteLine("Object has disposed");  
        }  
   
        public void Dispose()  
        {  
            Dispose(true);  
            GC.SuppressFinalize(this);  
        }  
    }  
      
    class Program  
    {  
        static void Main(string[] args)  
        {  
            Garbage g = new Garbage();  
            Console.WriteLine("String created" + g.name);  
            g.Dispose();  //Dispose the g object  
              
            Console.ReadLine();  
        }  
    }