Learn C# Design patterns step by step with a project – Part 1

Shivprasad koirala, 14 Jul 2015

In this article we will learn C# Design pattern and Architecture pattern Step by Step with a project.

Download CustomerProject-noexe.zip - 6.7 MB

Download CustomerProject.zip - 6.7 MB

Contents

Which Patterns and concepts we will learn in Part 1?

RIP :- Replace IF with Polymorphism.

Simple factory :- Move "NEW" keyword in to central class.

Lazy Loading :- Load objects on demand.

Prototype pattern :- Create a clone of object.

Strategy pattern :- Add algorithms dynamically.

IOC control concept :- Unnecessary work should be moved somewhere else.

DI :- Dependency injection implement IOC.

SOLID principles :- SRP and OCP is covered from SOLID principles.

Introduction

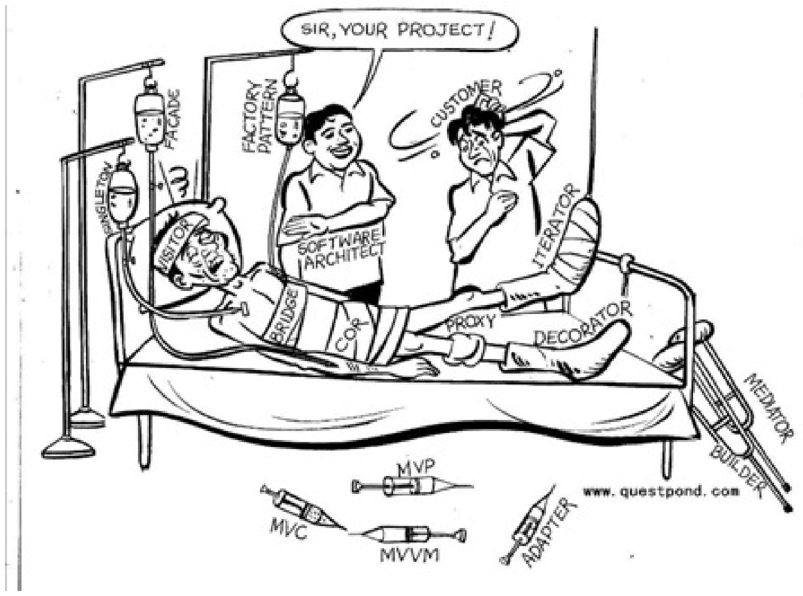
In this article we will learn how to implement design pattern and architecture pattern using C# language. We are not going to go pattern by pattern but rather we take a sample project and try to implement these things on the same.

So why is this article taking a project based approach and why not anexample based approach?

Design pattern and architecture patterns are thought processes. Thought process cannot be explained by PPT, UML Diagrams etc. You need to see the code, you need to feel it ,map it with a real project scenario.

If you see most of the design pattern articles on the internet / books are either explained with just UML diagrams (Not all developers understand UML) or examples like Car, Tree , Human's etc which does not make you feel those examples are real.

So let us take a sample project requirement and let us start coding and designing an application and let design patterns fall NATURALLYalong the way.



Myth 1:- For a good architecture all design patterns have to be implemented in a project.

Fact: - Patterns come out naturally and is completely on-demand.

A small joke just before you start with something heavy

Interviewer: - So tell me where you have used design pattern in real life.

Candidate: - Only during interviews.

Design patterns VS Architecture patternVS Architecture Style

Myth 2:- Design patterns and architecture patterns are same.

Fact: - Design patternare at leastpseudo code level while architecture patterns are at component level.

Meaning of the word "Pseudo ":- Approximately it looks like that.

We have seen lot of people using these vocabularies interchangeably. But there is a significant difference in the way they work.

So first let us try to understand the word "patterns" and then we will dive deeper.

If you see the plain English meaning of patterns: - They are recurring and predictable events.

For example climate change follows a pattern. Generally (at least in India) you have summer followed by rains and then cold. Humans identify these patterns to organize themselves in a better manner.

In the same way in software world the problems which occur mostly have a specific pattern and many developers have solved these problems and have come out with a solution. Later some of these solutions have proven their worth over a period of time and have become standard solution for that problem pattern.

For example if you want to SORT then you have time tested algorithms like bubble sort, inserted sort etc.

Design Pattern are pseudo code level solutions while architecture pattern are 30,000 feet level solutions defined at component level. In simple words if someone says "X" is a Design Pattern, Expect code, if someone say "Y" is an Architecture Pattern expect some kind of component level block diagram.

Architecture style is a thought process, a principle which just comes in one liners. For example REST is a architecture style where we give importance to HTTP.

Below are some examples for each one of them.

Design Pattern	Factory , Iterator , Singleton
Architecture Pattern	MVC, MVP, MVVM
Architecture Style	REST, SOA , IOC

Design pattern definition

Before we move ahead with the project let us try to put a definition around design pattern and later we will define architecture pattern down the line.

If you see the official definition for design pattern, it goes as follows:-

"Time tested Solution for recurring architecture problems".

But frankly this definition will not hold true as you start implementing and reading design pattern. At one moment of time you will see design pattern is all about good OOP (Object oriented programming) principles.

So let me put my definition as per my understanding and experience. I can guarantee that when we start running through all design patterns this definition would be more clear.

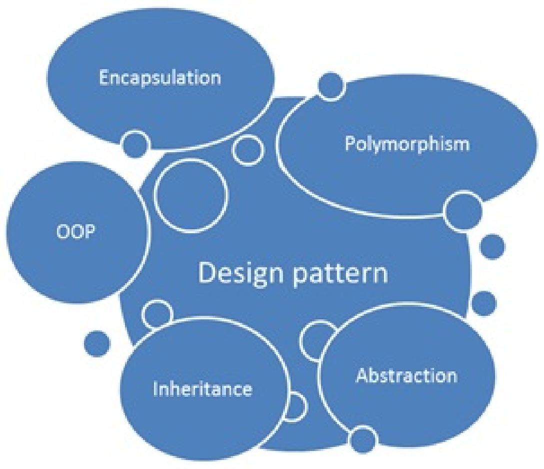
"Time tested Solution for recurring OOP problems".

This definition is iterated by the GOF team themself as well in this interview.

Myth 3 :- Design pattern makes you an complete architect.

Reality :- Design pattern is one of the things for an architect. It make you a better OOP.

Actually design pattern just makes you become better in OOP which is one of the aspects to become an architect. But it's not the only thing to become architect. We can also say Design pattern is a way of understanding OOP with scenarios.



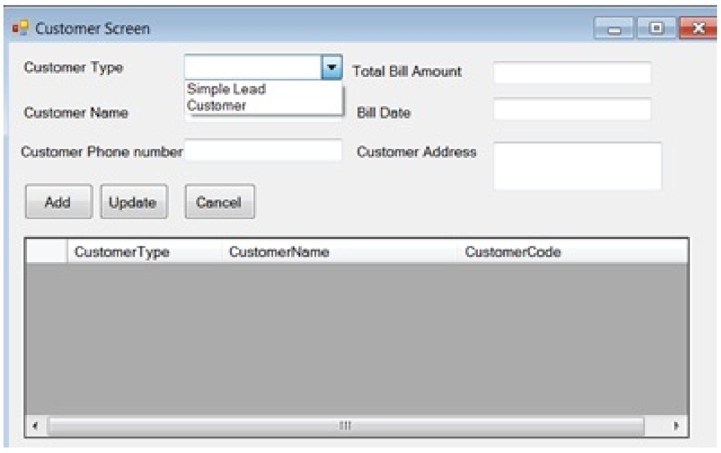
So let us not waste more time and start with a typical software requirement

The Cool Shop project: - Phase 1

"CoolShop" is a large retail shopping mall which has chain of mall's in Mumbai and Pune city. The company management wants a simple customer management system for their retail shops with the following features. Company has decided to launch the project in phases.

So in the first phase they just want to capture the customer information. Below is the requirement in more details:-

1. Application would be capturing 5 fields for now Customer Name, Phone number , Bill Amount , Bill date and Customer Address.
2. In phase 1 two types of customer data are collected. One is the lead and the other is a customer. A lead is a person who comes to cool shop but does not buy anything. He just enquires and goes away. A customer is a person who comes and buys things from the shop. A customer actually does a financial transaction.
3. When it's a lead only Customer name and phone number is compulsory but for a Customer all fields are compulsory. System should have a provision to add new validation rules seamlessly and these validation rules should be flexible and reusable to apply to the system.
4. System should have the ability to display, add, update and delete customer data.
5. For now the system will use SQL Server and ADO.NET as the data layer technology. But in the coming months we would be migrating this data layer to Entity framework. Migration should be seamless without many changes across the system.
6. System should have the ability of cancelling any modification done on the screen. So if the customer is editing a record and he has changed some values, he should have the opportunity to revert back to the old values.



Software Architecture is an evolution

Myth 4 :- Architecture should be perfect and right at the first time.

Reality :- Architecture is evolution. Start small and then keep improving along the way.

This is one of the biggest guru mantra for new architects. Software architecture evolves and patterns NATURALLY and GRADUALLY fall along the way to make the final thing.

Lot of people try to learn design and architecture pattern by going through code after code and pattern after pattern. That way of learning pattern is very harmful because you just see some academic code but it never becomes a natural part of you.

The best way to learn design pattern is to see a full evolution and by doing a project and let patterns naturally gradually fall along the way.

So rather than learning pattern after pattern , we will just try to architect the above project and patterns will fall naturally along the way and we will pointing to that pattern as they come along.

So let's start first with simple OOP concepts, classes, objects and let pattern come on demand basis.

What is an ENTITY?

The first thing we need to do is identify entities. So let's understand what the exact meaning of entities is in English.

Entities are things which you see in your real world. These can be identified uniquely. For example a person, place, things etc are some examples of entities.

In OOP world the technical name to entities is objects. So we would be using these words interchangeably in this article.

Step 1:- Identify your entities / objects

Thought process 1:- Software code should replicate real world, real people. So if you a accountant in real world in your software code you should have an entity called as accountant.

Software applications are created to automate real world and real people. So if your softwarecode replicates the real world objectsyou can manage your software in a better and controlled way.

That's what object oriented programming is all about. OOP says that your code should reflect your domain behavior and domain entities.

Domain means the business unit , its rules and how it works.

So the first step is to identify entity / objects from the above requirement. So from the top requirement below are the identified nouns:-

- Mumbai
- Pune
- Customer
- Lead
- Cool Shop

Thought process 2:- Nouns become entities and verbs become actions for entities. Pronoun becomes the properties and behavior for those entities.

One of the practices many architects follow is to identify nouns , pronoun and verbs. These nouns are then analyzed and identified as objects / entities. But be careful with this approach because you can end up with unwanted nouns and verbs. So keep only those nouns and verbs which are connected with the final software system.

If you review the above identified entities for now "Mumbai" and "Pune" are city names and have no direct connection with the software as such. "CoolShop" which is the name of the mall also does not have a direct connection.

So the only useful entity at this moment in phase 1 is "Lead" and "Customer". We will be adding , updating , deleting data around customer.

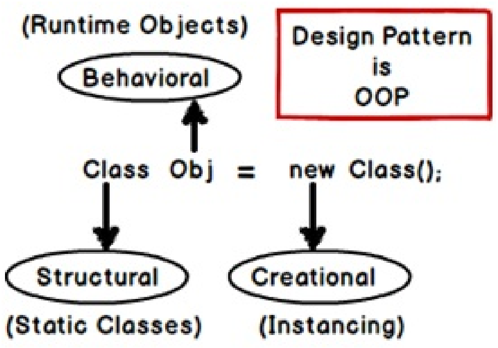
Now in order thatthese entities come live in your computer,you need code and logic. So we need some kind of a template where we can write this code, that's what we term it as "CLASS". This class will thenbe instantiated to create objects and entities inside your computer.

OOP is a three phase process:-

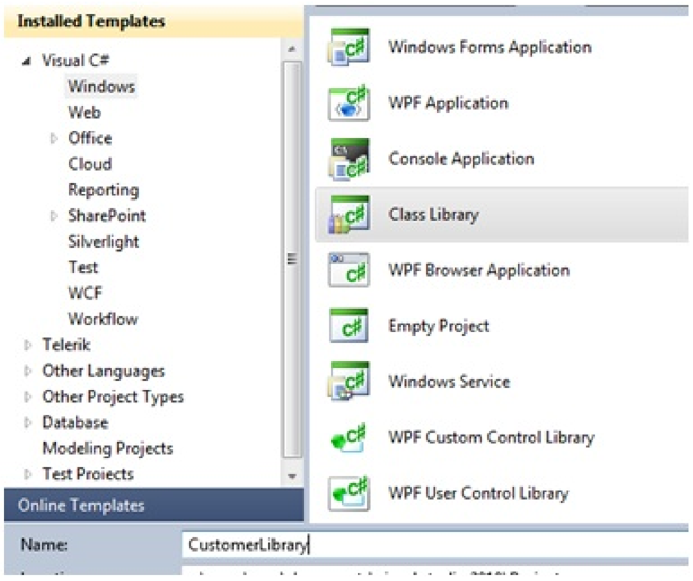
- **Template creation: -** Create classes and write logic in those classes.
- **Instantiate: -** Create entities / objects of these classes and bring them live in RAM / Computer.
- **Run: -** Interact with these objects to achieve the software functionality.

Now as a developer you would encounter common / recurring designproblems in all these three phases. Design pattern has solutions for OOP problems in all these three phases. Design patterns are divided in to three categories and they cover these phases as follows:-

OOP Phase	Design pattern category
Template / Class creation problem	Structural design pattern.
Instantiation problems	Creational design pattern.
Runtime problems	Behavioral design pattern.



So let's go ahead and create two classes one is the lead and the other is the customer.



Name: CustomerLibrary

Below is the code forthose two classes.

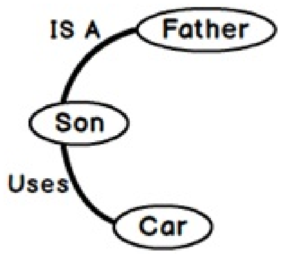
Note :- I can see from here people raising concerns of duplicity in my below code. I will be addressing them shortly , just keep reading.

```
namespace CustomerLibrary
{
public class Lead
    {
        public string LeadName { get; set; }
        public string PhoneNumber { get; set; }
        public string PhoneNumber { get; set; }
        public decimal BillAmount { get; set; }
        public DateTime BillDate { get; set; }
        public string Address { get; set; }
    }
    public class Customer
    {
        public string CustomerName { get; set; }
        public string PhoneNumber { get; set; }
        public decimal BillAmount { get; set; }
        public DateTime BillDate { get; set; }
        public string Address { get; set; }
    }
}
```

Step 2:- Identify relationships between entities

Real world entities interact with each other, they have relationships. So our next step is to identify relationships between entities. If you visualize the relationships that exist in real world, they are of primarily of two types "IS A" and "HAS A".

For example son "IS A" child of his father and son "HAS A" a car gifted by his father. "IS A" is more of a parent child relationship (hierarchical) while "HAS A" is more of a using relationship(Aggregated , Composed and Associated).



If you read requirement number two, it clearly means:-

"Lead IS A type of Customer with less validations".

So now our class code becomes something as shown below. "Lead" is a child class which inherits from the "customer" class.

```
public class Customer
{
        public string CustomerName { get; set; }
        public string PhoneNumber { get; set; }
        public decimal BillAmount { get; set; }
        public DateTime BillDate { get; set; }
        public string Address { get; set; }
}
public class Lead : Customer
{

}
```

For the "Customer" entity Customer Name, Phone number , Bill amount and Bill Date is compulsory. In the below code we have created a simple "Validate" method which checks all the above properties.

The most important point is that the "Validate" method is made virtual. So that new classes can override the validate logic.

```
public class Customer
    {
// All properties are deleted for simplification
        public virtual void Validate()
        {
            if (CustomerName.Length == 0)
            {
                throw new Exception("Customer Name is required");
            }
            if (PhoneNumber.Length == 0)
            {
                throw new Exception("Phone number is required");
            }
            if (BillAmount > 0)
            {
                throw new Exception("Bill is required");
            }
            if (BillDate >= DateTime.Now)
            {
                throw new Exception("Bill date  is not proper");
            }
        }
    }
```

The next step is to create a "Lead" class which inherits from the "Customer" class and overrides the validate method. As discussed a "Lead" class will be less restrictive as it just needs the name and phone number. Below is the code which overrides the customer class less validations.

```
public class Lead : Customer
    {
        public override void Validate()
        {
            if (CustomerName.Length == 0)
            {
                throw new Exception("Customer Name is required");
            }
            if (PhoneNumber.Length == 0)
            {
                throw new Exception("Phone number is required");
            }
        }}
```

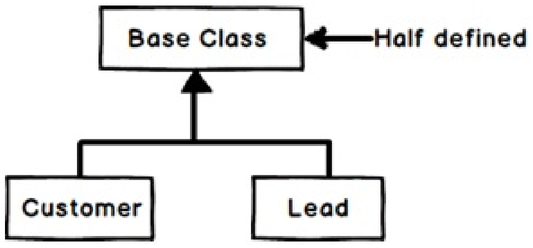
Thought process 3 :- "IS A" relationship is a parent child relationship while "HAS A" is a using relationship.

Step 3 :- Deriving from a common class

If you read Requirement number 4 it says that system should have ability to add new customer types tomorrow.

As per this requirement our class design does not look logical. A logical approach would be that there should be a “HALF DEFINED" class with all properties and the "Validate" method should be empty (half defined). This empty method can then be overridden by the child classes depending on their behavior.

So in other words we should have some kind of base class from which we can derive customer and lead class.



Below is the customer base class with all properties and the “Validate” method is kept empty to be defined by its child classes.

```
public class CustomerBase
{
        public string CustomerName { get; set; }
        public string PhoneNumber { get; set; }
        public decimal BillAmount { get; set; }
        public DateTime BillDate { get; set; }
        public string Address { get; set; }
        public virtual void Validate()
        {
         // Let this be define by the child classes
        }
}
```

Note :- Some of the senior folks around who are reading this are already screaming , make that class “ABSTRACT”. Yes ,thats coming soon.I want to delay it further so that we can understand the real time use of Abstract class.

So now if we want to create a customer class we can just inherit from the base customer class and put validations.

```
public class Customer : CustomerBase
    {
        public override void Validate()
        {
            if (CustomerName.Length == 0)
            {
                throw new Exception("Customer Name is required");
            }
            if (PhoneNumber.Length == 0)
            {
                throw new Exception("Phone number is required");
            }
            if (BillAmount > 0)
            {
                throw new Exception("Bill is required");
            }
            if (BillDate >= DateTime.Now)
            {
                throw new Exception("Bill date  is not proper");
            }
        }
    }
```

If we want to create a Lead class which only validates Name and Phone number we can again inherit from the “CustomerBase” class and write the validation accordingly.

```
public class Lead : CustomerBase
    {
        public override void Validate()
        {
            if (CustomerName.Length == 0)
            {
                throw new Exception("Customer Name is required");
            }
            if (PhoneNumber.Length == 0)
            {
                throw new Exception("Phone number is required");
            }
        }
    }
```

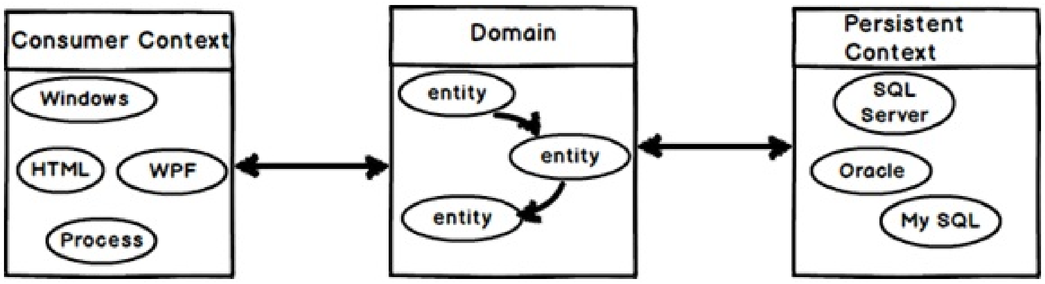
The technical context

The above defined three classes are stored in hard disk in the form of ".CS" extensions. Now two things need to be done:-

1. Some UI should invoke these classes , bring these entities live in to RAM.
2. Second once the end user finishes his operation we need to store these entities in to the hard disk.

In other words our entities need IT infrastructure to run. It needs a UI infrastructure (WPF, Flash,HTML) for invoking and persistent infrastructure (Sql Server , Oracle) for saving to hard disk.

If you put a visual picture of thesesections with the technical context it looks something like this. So we can have consumer context which can be in a form of HTML, WPF, Windows etc. We can have persistent context which can be a file or RDBMS like SQL Server, Oracle etc.



With the above thoughts we end up with three kinds of sections:-

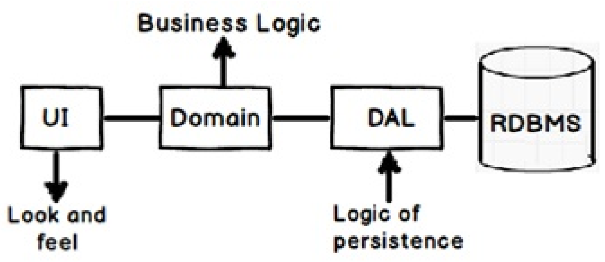
- Consumer section which mostly has UI.
- Domain section which has your classes and business logic.
- Persistent section which is nothing but your RDBMS, file etc.

Thought process 4:- When you visualize a class always them in these two technical context and business entity context.

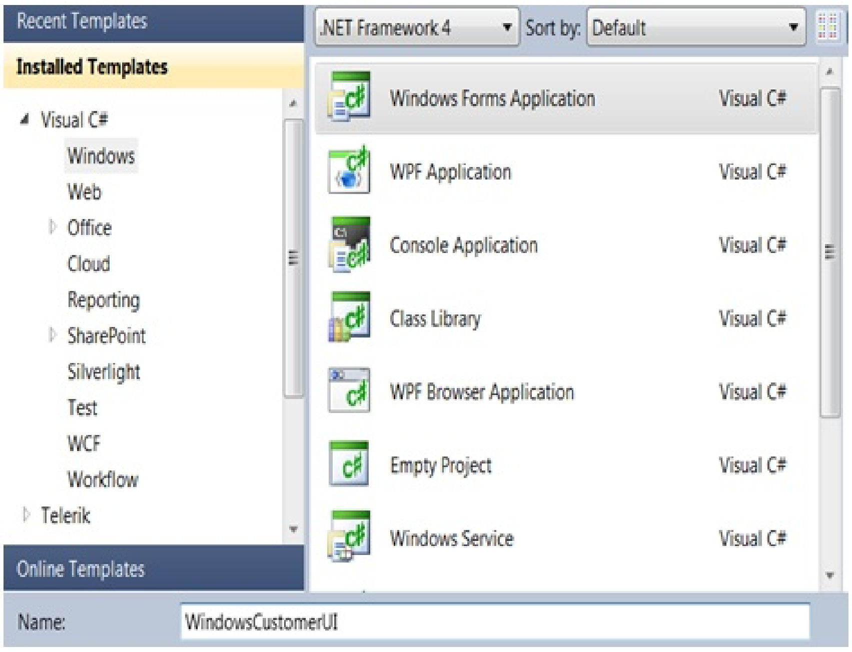
Three Layer architecture – Managing changes

Litmus test of software architecture happens during changes. When you change in one place if you are changing in lot of places that means it's a sign of a bad architecture.

So to avoid changes all over places we need to make proper layers and compartments and put similar nature of responsibility in those layers. So as discussed in technical context we currently have three sections atleast UI ,Domain / Business layer and the Data layer.

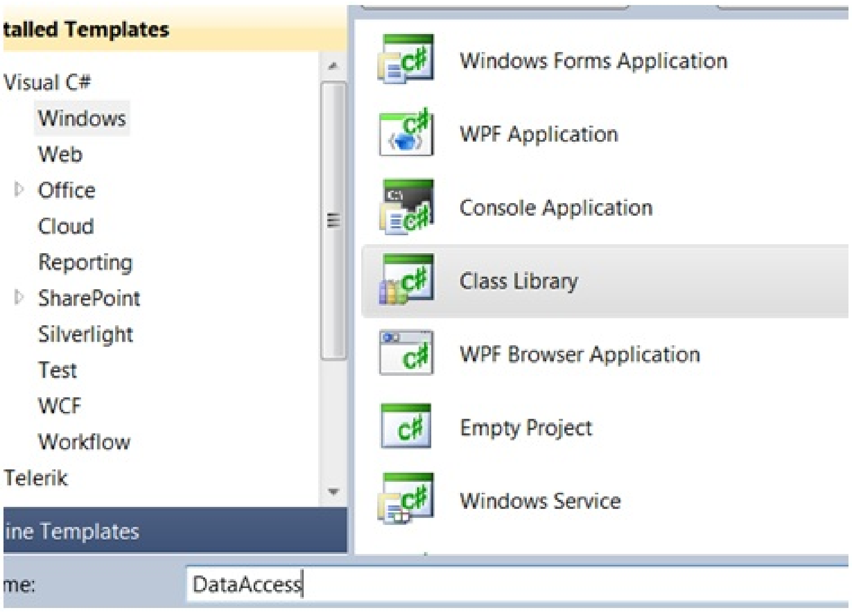


So in the same project let's add a simple Windows UI for the UI section.



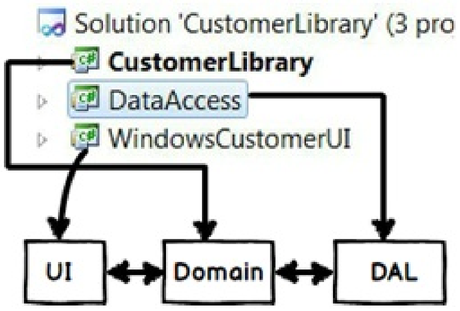
Name: WindowsCustomerUI

We need to class library one for “Data Access” and other for customer entites.



me: DataAccess

So now your solution looks something as shown in the below picture. Three different project layers for three different things.



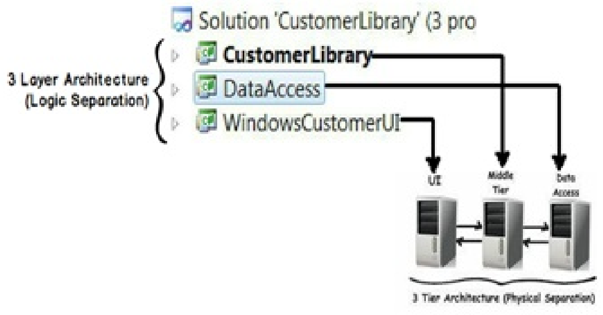
So now how does layering help us to manage change better?Because we have divided our project in to logical layers it’s very easy to know what changes have to go in which layer. For example if we want to upgrade data access layer we just need to change the third layer , if we want to move from one UI technology to other we need to change the UI layer and so on.

The above architecture is termed as “Three Layer Architecture”.

Thought process 5:- Always divide your project in to logical layers and every layer should have some unique responsibility.

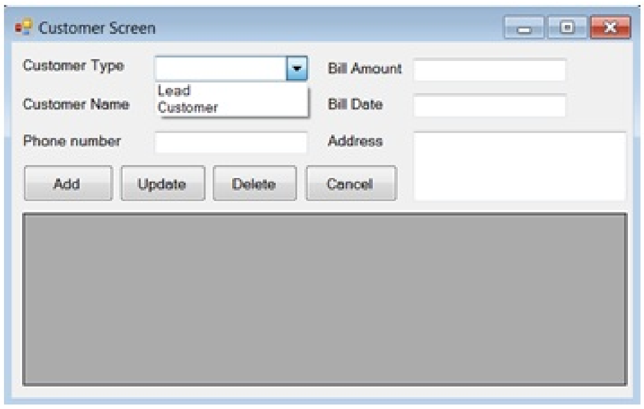
Three layer VS Three tier

One of the confusing terms in the architecture world is “Layer” VS “Tier” architecture. In three layer architecture we just have logical separation. In three tierall these three layers are deployed in to separate physical machines.


Solution 'CustomerLibrary' (3 pro
CustomerLibrary
DataAccess
WindowsCustomerUI
3 Layer Architecture (Logic Separation)
UI
Middle Tier
Data Access
3 Tier Architecture (Physical Separation)

Step 4:- Creating user Interface

In the UI layer we have put the necessary controls for the UI.



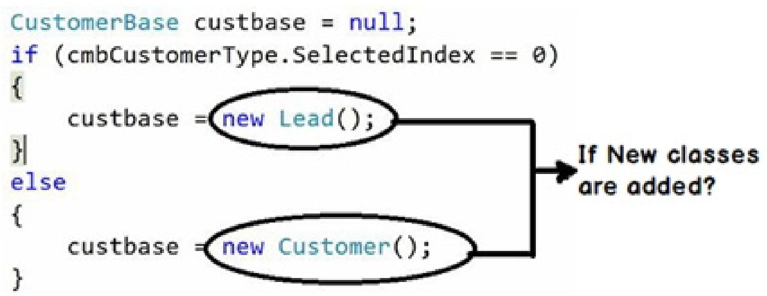
In this UI layer we have added reference of the “Customer” library and in the drop down change we are either creating a “Lead” object or “Customer” object.

```
private void btnAdd_Click(object sender, EventArgs e)
{
            CustomerBase custbase = null;
            if (cmbCustomerType.SelectedIndex == 0)
            {
                custbase = new Lead();
            }
            else
            {
                custbase = new Customer();
            }
            custbase.CustomerName = txtCustomerName.Text;
            custbase.Address = txtAddress.Text;
            custbase.PhoneNumber = txtPhoneNumber.Text;
            custbase.BillDate = Convert.ToDateTime(txtBillingDate.Text);
            custbase.BillAmount = Convert.ToDecimal(txtBillingAmount.Text);
}
```

So can you guess what is the problem with the above code ?.Think ??.

Ok , we have circled the problem below to help your understand the issue. The issue is “Change”. As defined in requirement 4 new customer types can be added in future. So when new customer types get added we need to change the UI code or let me extrapolate many UI screens like this.

Remember what we said a good architecture is that architecture where when we change at one placeand we do not need to change all over the places.



```
CustomerBase custbase = null;
if (cmbCustomerType.SelectedIndex == 0)
{
    custbase = new Lead();
}
else
{
    custbase = new Customer();
}
```

The S of SOLID (Single responsibility principle)

Its time to talk about SOLID principles. SOLID are principles which if we follow will make our OOP better.

- Single Responsibility principle (SRP).
- Open close principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface segregation principle (ISP)
- Dependency inversion principle.

For now let us not talk about all the four SOLID principles, in case you want to know about it right now , you can read this C# Solid article which talks only about SOLID principles.

For now let us just concentrate on "S" of SOLID i.e. Single Responsibility principle (SRP).

SRP says that a class should do only one work at a time and not unrelated things.If you see the UI it is supposed to do layouting, visuals , take inputs and so on.But for now he is creating a "Customer" object which is not his duty.

In other words UI is handling multiple responsibilities which makes the class more complex and unmaintainable in the further times.

Thought process 6:- Keep SOLID principles running at your back ground when you are doing architecting.

The SRP Synonym Separation of concerns(SOC)

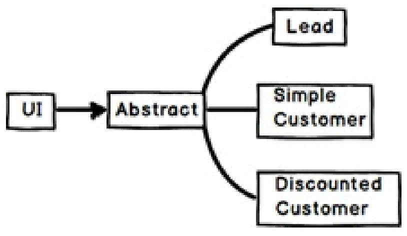
One of the synonyms of SRP is SOC – Separation of concern. Separation of concern rule says that one class should do only his concerns and any unwanted concerns should be moved to some other class. For example in this case the UI should not be creating "Customer" objects directly.

So next time when you see SOC you can think it's a synonym for SRP or vice-versa.

Note: - I am not aware of the history which came first SOC or SRP. But they definitely have the same goals.

Step 5 :- Decoupling needs abstract thinking – Creating interfaces

In order to achieve decoupling between the UI and the Customer types, the UI has to see the Customer types in an abstract way rather than dealing with concrete classes.



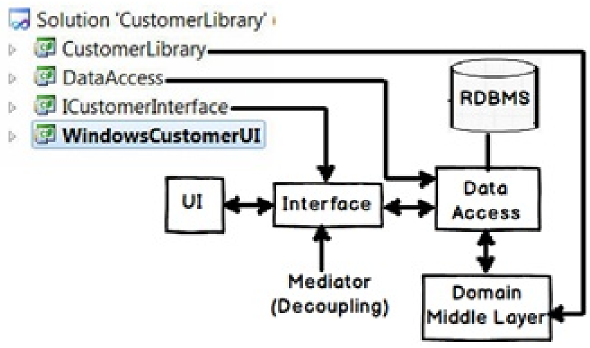
Abstraction: - *It's an OOP principle where we show only necessary things to the consumer.*

The UI should be only talking with pure definitions rather than implemented concrete classes. That's where interfaces come in to picture. Interfaces help you create pure definitions. Your UI will then point to these pure definitions and not worry about the implemented classes at the back end.

So let us create a separate class library with a name "ICustomerInterface" in the same solution. This interface we will reference in the UI. Below is how the interface "ICustomer" looks like , just empty definitions and empty methods.

```
public interface ICustomer
{
         string CustomerName { get; set; }
         string PhoneNumber { get; set; }
         decimal BillAmount { get; set; }
         DateTime BillDate { get; set; }
         string Address { get; set; }
         void Validate();

}
```

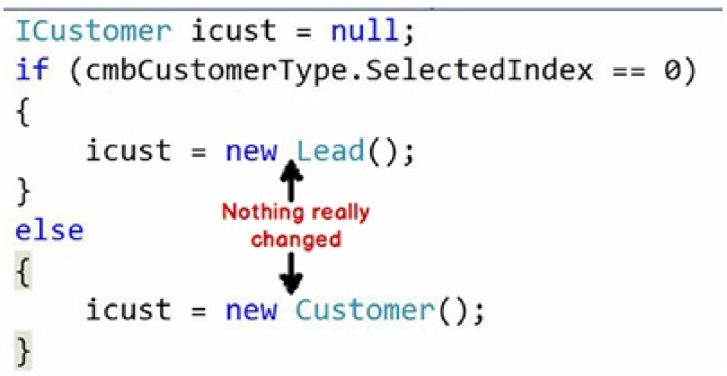
So now the overall solution from the architecture perspective looks something as shown below. The interface now stands as a mediator between the UI and the customer class library.



So if you now see our client code it becomes something as shown below.

```
ICustomer icust = null;
if (cmbCustomerType.SelectedIndex == 0)
{
icust = new Lead();
}
else
{
icust = new Customer();
}
icust.CustomerName = txtCustomerName.Text;
icust.Address = txtAddress.Text;
icust.PhoneNumber = txtPhoneNumber.Text;
icust.BillDate = Convert.ToDateTime(txtBillingDate.Text);
icust.BillAmount = Convert.ToDecimal(txtBillingAmount.Text);
```

But in reality things has not changed. If we add a new class we still need to create the object of concrete implemented classes.



```
ICustomer icust = null;
if (cmbCustomerType.SelectedIndex == 0)
{
    icust = new Lead();
}
else
{
    icust = new Customer();
}
```

So we need to something more , watch the next step for the solution.

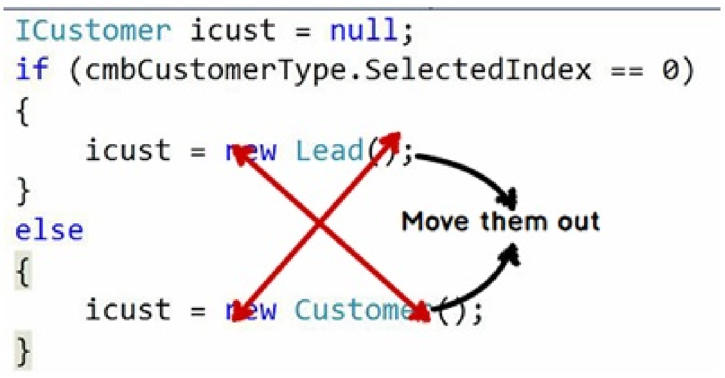
Thought process 7:- The primary work of interfaces is to decouple classes from each other.

Step 6 :- PIC Pattern for decoupling (Simple Factory Pattern)

Acronym of PIC is "Polymorphism + Interfaces + Centralizing object creation"

If you watch the code very closely you will get the core reason why still decoupling is not achieved. Its all because of the "NEW" keyword. "NEW" keyword one of the prime reason why two systems are tightly coupled.

So the first step is to get rid of the "NEW" keyword from the consumer end. So let's start moving the "NEW" keyword to a central factory class.

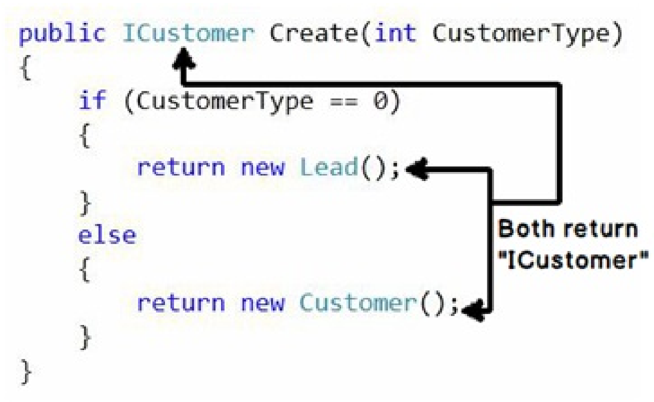


```
ICustomer icust = null;
if (cmbCustomerType.SelectedIndex == 0)
{
    icust = new Lead();
}
else
{
    icust = new Customer();
}
```

So we have added a new class library project called as “FactoryCustomer” with the below code.

```
public class Factory
{
        public ICustomer Create(int CustomerType)
        {
            if (CustomerType == 0)
            {
                return new Lead();
            }
            else
            {
                return new Customer();
            }
        }
}
```

It has a simple “Factory” class with a create method. This “create” method takes a numeric value and depending on the numeric value it either creates a “Lead” object or a “Customer” object. But the specialty of this “Create” function is that it returns “ICustomer” interface type.



```
public ICustomer Create(int CustomerType)
{
    if (CustomerType == 0)
    {
        return new Lead();
    }
    else
    {
        return new Customer();
    }
}
```

So now the console application UI uses the “Factory” to create objects and because the “Create” function returns “ICustomer” type the UI does not have to worry about the concrete customer classes at the back end.

```
ICustomer icust = null;
Factory obj = new Factory();
icust = obj.Create(cmbCustomerType.SelectedIndex);
```

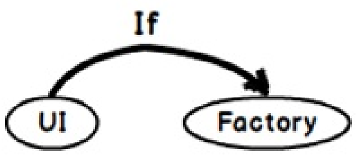
You can see in the above code there is no reference of concrete class like “Lead” or “Customer” type which shows that the UI is not decoupled from the core customer library classes.

In case you are wondering about why we named the class as “factory”. A factory in real world means an entity which creates (manufactures) things and our “factory” class is exactly doing that creating objects. So the name goes hand to hand with the same.

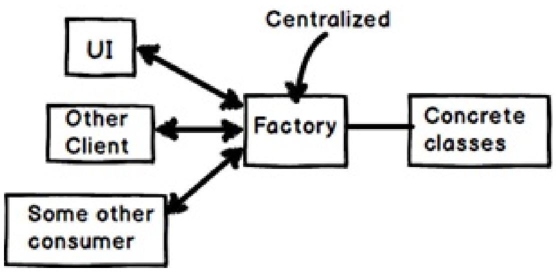
Thought process 8:- “NEW” keyword is the main culprit for tight coupling .

Step 7 :- RIP Pattern (Replace IF with Polymorphism)

If you watch step 6 code we just passed the bucks. The “IF” condition which was in the UI part is now the part of factory which is better but in reality the “IF” condition still exists. It’s just that it has been moved from the UI to the factory.



The advantage of centralization of object creation is that if we are consuming concrete classes in lot of places and clients we do not need to change in other places.



But now let's start thinking how we can remove the "IF" condition.You must have heard the below best practice statement:-

"If there is polymorphism and if you see lots of IF conditions that means polymorphism benefit is not exploited".

To remove "IF" condition is a three step process :-

Step 1:- Create a list of collection of "ICustomer".

```
private List<icustomer> customers = new List<icustomer>();</icustomer></icustomer>
```

Step 2:- In the constructor load the types of customer classes like lead and customer.

```
public Factory()
        {
            customers.Add(new Lead());
            customers.Add(new Customer());
        }
```

Step 3:- The create method just look's up the list by index and returns type of customer. Because of polymorphism the concrete customer classes gets automatically type casted to a generic interface.

```
public ICustomer Create(int CustomerType)
        {
            return customers[CustomerType];
        }
```

Below is the full code of the changed factory class.

```
public class Factory
{
        private List<icustomer> customers = new List<icustomer>();
        public Factory()
        {
            customers.Add(new Lead());
            customers.Add(new Customer());
        }
        public ICustomer Create(int CustomerType)
        {
            return customers[CustomerType];
        }
}
</icustomer></icustomer>
```

Note: - The biggest constraint of RIP pattern is that the concrete classes should come in inheritance hierarchy and have same signatures. Polymorphism and Inheritance is a compulsory feature for RIP pattern to exist.

Thought process 9:- In Polymorphism IF can be replaced with a dynamic polymorphic collection.

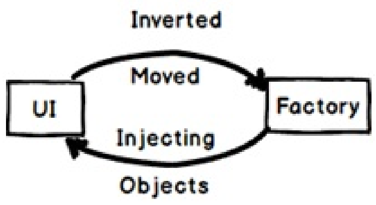
Pattern 1 RIP Pattern: -If you have the advantage of polymorphism and you see lots of IF condition, there is good chance that you can replace the IF condition with a simple collection look up. This pattern falls in to Behavioral category.

IOC a thought , DI a implementation

IOC is a thought or rather I can say principle where unconcerned work of an entity should be moved out somewhere. Read that full form again: - Inversion of control or we can be very specific by saying Inversion of UNWANTEDcontrol to some other entity.

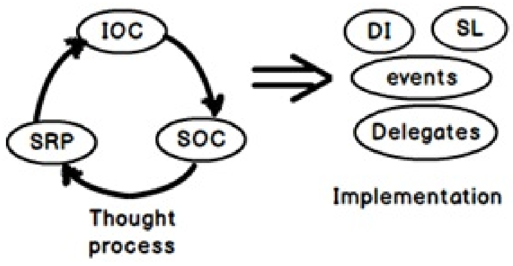
If you see in the above scenario we moved or I will say we inverted the unwanted responsibility of "Customer" object creation from the UI to the "Factory" class.

Now IOC principle can be achieved by multiple ways, Dependency Injection , Delegates and so on.



Now in order to achieve this thought process we used the “Factory” class. If you see from the factory class perspective we are actually injecting an object in to the UI. So the UI needs objects which are injected by the factory class. DI is a process of injecting the dependent objects from a separate entity to achieve decoupling.

All in the Family: - SOC,SRP, IOC and DI



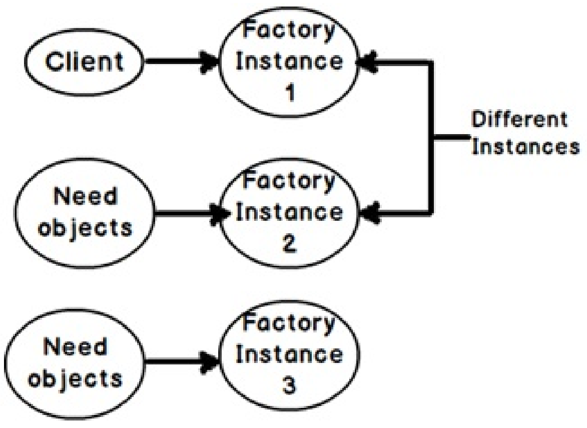
If you see closely SOC , SRP and IOC are almost synonyms and these principles can be implemented by using DI , Events , Delegates , Constructor injection , service locator etc. So for me SOC , SRP and IOC all look synonyms.

I would encourage you to watch this video which explains IOC and DI practically.

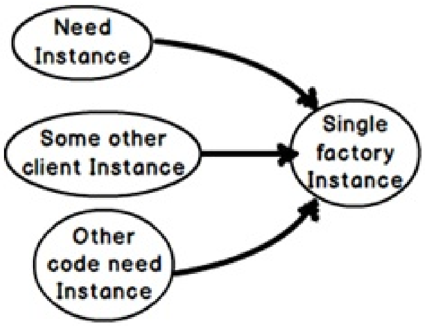
Step 8:- Improving performance of Factory class

In the above architecture the factory class will perform very badly if we have lot of concrete objects

tomorrow and if we are creating factory instances again and again it will lead to lot of memory consumption.



So if we can just have ONE INSTANCE of the factory class with all concrete objects loaded once that would really boost up the performance.This one instance can then be used to serve all the client who needs instances.



In order to have single instance copy we need to do the following:-

- Declare the class as static.
- Declare the list in which the types will be stored as static.
- Finally the Create function should also be defined static so that it can access static variables.

```
public static class Factory
    {
        private static List<icustomer> customers = new List<icustomer>();
        static Factory()
        {
            customers.Add(new Lead());
            customers.Add(new Customer());
        }
        public static ICustomer Create(int CustomerType)
        {
            return customers[CustomerType];
        }
    }
</icustomer></icustomer>
```

On the client side the factory calling code becomes much simpler now.

```
icust = Factory.Create(cmbCustomerType.SelectedIndex);
```

Pattern 2 Simple Factory Pattern: - *By centralizing object creation and returning a generic interface reference helps to minimize changes when changes are applied to the application. This falls in creation category.*

This pattern should not be confused with Factory pattern of gang of four. The base of Factory pattern is simple factory pattern.

Step 9:- Lazy Loading the Factory

Also other great thing we can do with the factory here is loading the objects on demand. If you see for now the objects are loaded irrespective you want them or not. How about we just load Just-in-time, in other words when we want the objects we load them.

So converting the above factory in to lazy loading is a two-step process:-

Step 1 :- Make the object collection types null. Do not load them.

```
private static List<icustomer> customers = null;</icustomer>
```

Step 2 :- The create function will now first check if the object is NULL then load it or else just lookup in the collection.

```
public static ICustomer Create(int CustomerType)
        {
            if (customers == null)
            {
                LoadCustomers();
            }
            return customers[CustomerType];
        }
```

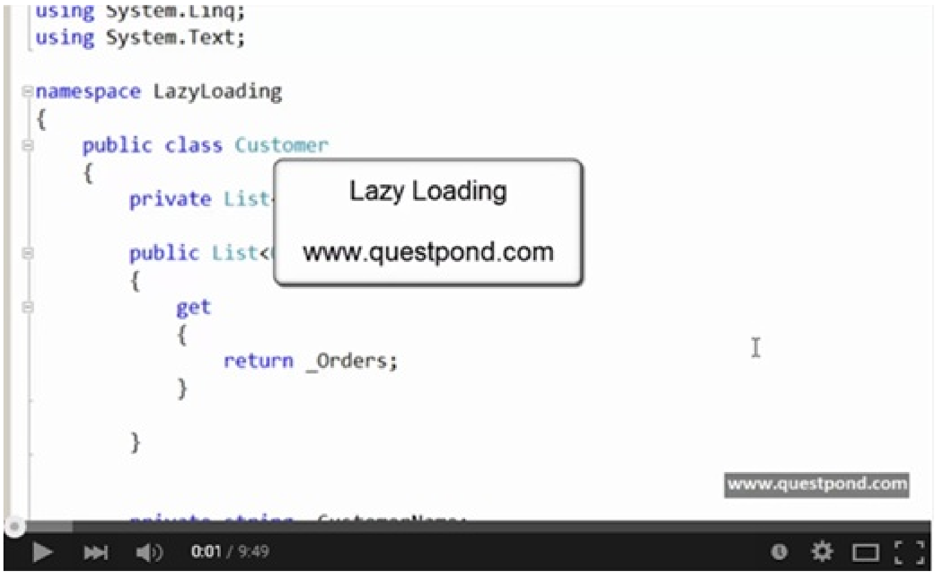
Below goes the full code with Lazy loading.

```
public static class Factory
{
        private static List<icustomer> customers = null;

        private static void LoadCustomers()
        {
            customers = new List<icustomer>();
            customers.Add(new Lead());
            customers.Add(new Customer());
        }
        public static ICustomer Create(int CustomerType)
        {
            if (customers == null)
            {
                LoadCustomers();
            }
            return customers[CustomerType];
        }
}
</icustomer></icustomer>
```

Pattern 3 Lazy Loading :- *This is a creational design pattern where we load objects only when we need it. The opposite of Lazy loading is eager loading.*

Home work: - Automating Lazy loading using Lazy Keyword

Lazy design pattern can be automated and made simple by using C# Lazy keyword. This I will leave as a home work for your guys. See the below youtube video to understand C# Lazy loading concept and then try to replace your custom code with the C# Lazy keyword.



```
using System.Linq;
using System.Text;

namespace LazyLoading
{
    public class Customer
    {
        private List
        
        public List<
        {
            get
            {
                return _Orders;
            }
            
        }
```

Below is the code using the C# Lazy keyword.

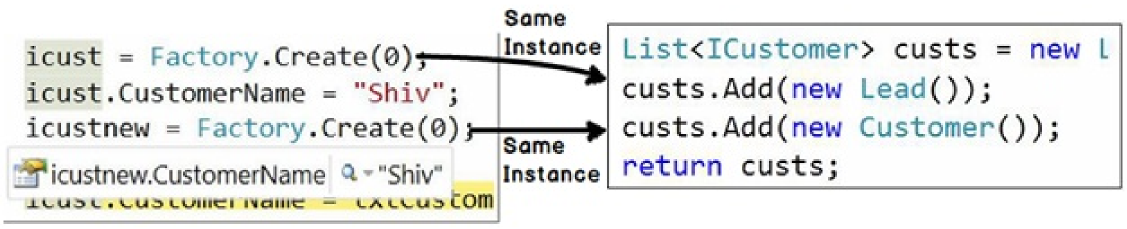
```
public static class Factory
    {
        private static Lazy<list<icustomer>> customers = null;
        public Factory()
        {
            customers = new Lazy<list<icustomer>>(() => LoadCustomers());
        }
        private  List<icustomer> LoadCustomers()
        {
            List<icustomer> custs = new List<icustomer>();
            custs.Add(new Lead());
            custs.Add(new Customer());
            return custs;
        }
        public static ICustomer Create(int CustomerType)
        {
            return customers.Value[CustomerType];
        }
    }
</icustomer></icustomer></icustomer></list<icustomer></list<icustomer>
```

Step 10:- Implementing cloning (Prototype pattern)

Now the above factory pattern class has a defect, can you guess what it is ?.

```
icust = Factory.Create(0);
```

Now why does it return the same instance because the factory pattern is pointing to the same instance of the collection. Now that's disastrous because the whole point of factory is to create new instance and not return the same instance.



```
icust = Factory.Create(0);
icust.CustomerName = "Shiv";
icustnew = Factory.Create(0);
icustnew.CustomerName  "Shiv"
icust.CustomerName = txtCustom
```

```
List<ICustomer> custs = new L
custs.Add(new Lead());
custs.Add(new Customer());
return custs;
```

So we need some kind of mechanism where in rather than returning the same object a CLONE of the object should be returned, something like a BY VAL copy. That's where prototype pattern comes to picture.

So the first step is to define a "Clone" method in the "ICustomer" interface.

```
public interface ICustomer
{
         string CustomerName { get; set; }
         string PhoneNumber { get; set; }
         decimal BillAmount { get; set; }
```

```
        DateTime BillDate { get; set; }
        string Address { get; set; }
        void Validate();
        ICustomer Clone(); // Added an extra method clone
}
```

In order to create a “Clone” of a .NET object we have ready made “MemberwiseClone” function. In the base class of the customer we have implemented the same. With this approach any other type of customer class who is inheriting will also have the ability to clone objects.

```
public class CustomerBase : ICustomer
{
// Other codes removed for readability
public ICustomer Clone()
{
            return (ICustomer) this.MemberwiseClone();
}
}
```

Now the “Create” function of the factory will call the clone method after the lookup from the collection. So the same reference of the object will not be sent it will be a fresh new copy of object.

```
public static class Factory
{
// Other codes are removed for readability purpose
public static ICustomer Create(int CustomerType)
{
return customers.Value[CustomerType].Clone();
}
}
```

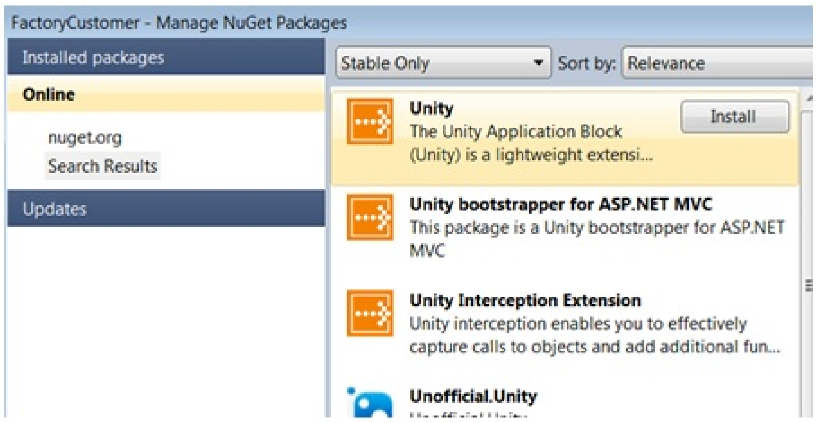
Pattern 4 Prototype pattern: - *This is a creational design pattern where we create a fresh new clone / instance of an object.*

Step 11:- Automating Simple Factory using Unity

A good developer always hunts how a design pattern can be automated by readymade frameworks. For example the above simple factory class is great but now think for a moment what if we need to support other object types like order , logger and so on. So writing factory for each one of them , creating polymorphic collection , look ups and then on top of it testing all these things is a huge task by itself.

The complete simple factory and the polymorphic collection look up can be automated / replaced by using some DI frameworks like unity , ninject , MEF etc.

I can understand some people shouting out their “What is DI ?”. Hold your breath we will discuss that very soon. For now lets concentrate on how the simple factory can be automated by using the DI framework. For now we will choose unity application block.



Unity
The Unity Application Block (Unity) is a lightweight extensi...

Unity bootstrapper for ASP.NET MVC
This package is a Unity bootstrapper for ASP.NET MVC

Unity Interception Extension
Unity interception enables you to effectively capture calls to objects and add additional fun...

Unofficial.Unity

So the first step is to get hold of the unity application block in your factory class using NUGET. In case you are new to NUGET you can see this video which explains NUGET fundamentals.

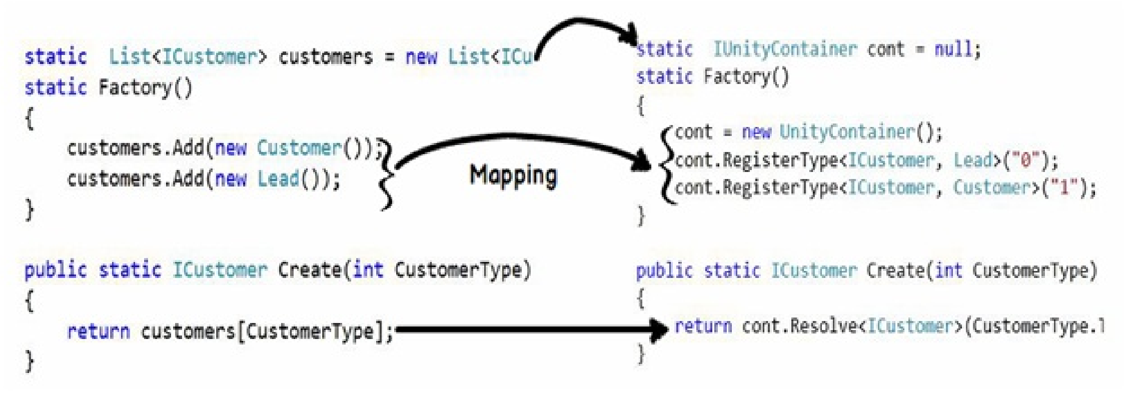
So the first step is to get hold of the namespace of unity application block.

```
using Microsoft.Practices.Unity;
```

In unity or any DI framework we have concept of the containers. These containers are nothing but collections. “RegisterType” and “ResolveType” methods helps to add and get objects from the container collection respectively.

```
static  IUnityContainer cont = null;
        static Factory()
        {
            cont = new UnityContainer();
            cont.RegisterType<icustomer, lead="">("0");
            cont.RegisterType<icustomer, customer="">("1");
        }
        public static ICustomer Create(int CustomerType)
        {
            return cont.Resolve<icustomer>(CustomerType.ToString());
        }
</icustomer></icustomer,></icustomer,>
```

Below is the image which shows how the manual Factory pattern code maps to automated Unity container code.



```
static  List<ICustomer> customers = new List<ICu
static Factory()
{
    customers.Add(new Customer());
    customers.Add(new Lead());
}

public static ICustomer Create(int CustomerType)
{
    return customers[CustomerType];
}
```

```
static  IUnityContainer cont = null;
static Factory()
{
    cont = new UnityContainer();
    cont.RegisterType<ICustomer, Lead>("0");
    cont.RegisterType<ICustomer, Customer>("1");
}

public static ICustomer Create(int CustomerType)
{
    return cont.Resolve<ICustomer>(CustomerType.1
}
```

Step 12 :- Abstract classes – The Half defined thing

If you remember the “CustomerBase” class it’sa half defined class. It defines all the properties but the validate method is later defined by the child concrete classes. Now think for a moment if someone creates the object of this half defined class, what can be the consequences?.

What will happen if he invokes the below empty “Validate”method ?.

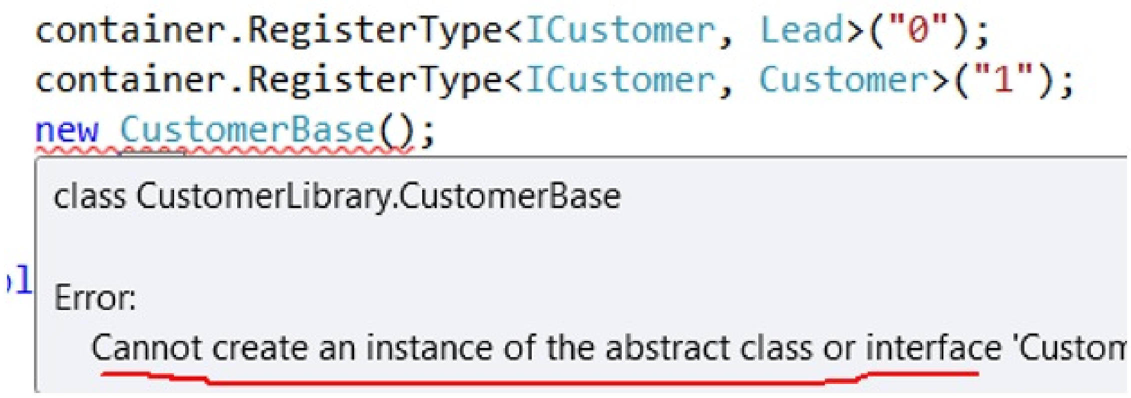
Yes , you guessed right CONFUSION.

```
public class CustomerBase : ICustomer
{
        public string CustomerName { get; set; }
        public string PhoneNumber { get; set; }
        public decimal BillAmount { get; set; }
        public DateTime BillDate { get; set; }
        public string Address { get; set; }
        public void Validate()
      {
        // To be defined by the child classes
  }
}
```

So the solution is to avoid the confusion by not allowing the client to create the object of the half defined class i.e. by creatingan “ABSTRACT CLASS”.

```
public abstract class CustomerBase : ICustomer
{
        public string CustomerName { get; set; }
        public string PhoneNumber { get; set; }
        public decimal BillAmount { get; set; }
        public DateTime BillDate { get; set; }
        public string Address { get; set; }
        public abstract void Validate();
}
```

Now if the client tries to create the object of abstract class he will be thrown with the following error and this helps us to prevent confusion of using half defined classes.



```
container.RegisterType<ICustomer, Lead>("0");
container.RegisterType<ICustomer, Customer>("1");
new CustomerBase();
```

class CustomerLibrary.CustomerBase

Error:
Cannot create an instance of the abstract class or interface 'Custon

Step 13 :- Generic Factory

If you see the factory class its binded with the “Customer” type at this moment. In other words if we want to punch out “Supplier” type we would need one more “Create” method as shown in the below code. So if we have lot of business objects like this we would end up with lot of “Create” method.

```
public static class Factory
{
public static ICustomer Create(int CustomerType)
{
return cont.Resolve<icustomer>(CustomerType.ToString());
}
public static Supplier Create(int Supplier)
{
return cont.Resolve<isupplier>(Supplier.ToString());
}
}
</isupplier></icustomer>
```

So rather than binding the “Factory” with a single type how about making it a “GENERIC” class.

In case you are new to “Generics” I would suggest you to go through this youtube C# Generic video which explains the concept of “Generics” in a detail manner. And also if you are from the school who think “Generic” and “Generic” collections are same you should definitely see the above video to remove that misconception.

Generic helps you to decouple the logic from the data type. So the logic here is “Object Creation” but attaching this logic only with “Customer” would make the architecture rigid. So how about making it a generic type “AnyType” as shown in the below code.

```
public static class Factory<anytype>
{
static IUnityContainer container = null;
public static AnyType Create(string Type)
{
if (container == null)
            {
                container = new UnityContainer();
                container.RegisterType<icustomer, lead="">("Lead");
                container.RegisterType<icustomer, customer="">("Customer");
            }
return container.Resolve<anytype>(Type.ToString());
}
}
</anytype></icustomer,></icustomer,></anytype>
```

Note :- I have changed the key from numeric “0” and “1” to “Lead” and “Customer” so that its more readable.

So now when a client wants to create “Customer” object he need to make the call as shown below.

```
ICustomer Icust =  Factory<icustomer>.Create("Customer");</icustomer>
```

Step 14 :- Strategy Pattern for validation

If you read Requirement number 3 it says that validations are different for “Customer” and different for a “Lead”. For “Customer” all fields are compulsory while for “Lead” only name and phone number is more than enough.

So in order to achieve the same we have created a virtual “Validate” method and this method is overridden with new rules in the new classes separately.

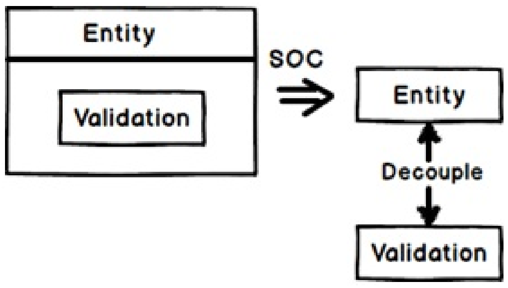
```
public class Customer : CustomerBase
    {
public override void Validate()
        {
if (CustomerName.Length == 0)
            {
throw new Exception("Customer Name is required");
            }
if (PhoneNumber.Length == 0)
            {
throw new Exception("Phone number is required");
            }
if (BillAmount == 0)
            {
throw new Exception("Bill Amount is required");
            }
if (BillDate >= DateTime.Now)
            {
throw new Exception("Bill date  is not proper");
            }
        }
    }

public class Lead : CustomerBase
    {
public override void Validate()
        {
if (CustomerName.Length == 0)
            {
throw new Exception("Customer Name is required");
            }
if (PhoneNumber.Length == 0)
            {
throw new Exception("Phone number is required");
            }
        }
    }
```

But if you read the requirement again it says that further that there is a possibility of adding new validations and we expect the system to be flexible or I will rather say DYNAMIC to achieve the same.

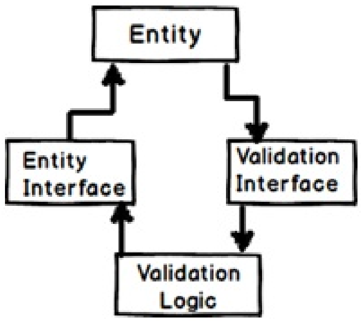
But when we use inheritance its more static than dynamic. Second the “Customer” class and “Lead” class are tightly coupled with validation algorithm / strategy. So if we want to achievedynamic flexibility we need to remove this validation logic from the entity and move it somewhere else.

At this moment the entity classes are tied up with the validation algorithm. In short:-

- We are not following SRP.
- We are not doing SOC and so
- We need to implement IOC that means we need move the logic of algorithm from the entity classes to some other class.



Now in order to achieve decoupling between “Entity” and “Validation Logic” we need to ensure that both these parties talk via a general interfaces rather than talking directly to the concrete classes.



We already have a general interface for “Customer” in the same way let us create a general interface for validation algorithm we will term this interface as “IValidationStratergy”.Also note we have made the interface generic so that we can use this interface tomorrow for other types like “Supplier” , “Accounts” etc.

```
public interface IValidationStratergy<anytype>
{
void Validate(AnyType obj);
}
</anytype>
```

We can now further implement the above interface and create different validationlogics. For example below is a the validation which checks

```
public class CustomerAllValidation  : IValidationStratergy<icustomer>
    {

public void Validate(ICustomer obj)
        {
if (obj.CustomerName.Length == 0)
            {
throw new Exception("Customer Name is required");
            }
if (obj.PhoneNumber.Length == 0)
            {
throw new Exception("Phone number is required");
            }
if (obj.BillAmount == 0)
            {
throw new Exception("Bill Amount is required");
            }
if (obj.BillDate >= DateTime.Now)
            {
throw new Exception("Bill date  is not proper");
            }
        }
    }
</icustomer>
```

```
public class LeadValidation : IValidationStratergy<icustomer>
    {
public void Validate(ICustomer obj)
        {
if (obj.CustomerName.Length == 0)
            {
throw new Exception("Customer Name is required");
            }
if (obj.PhoneNumber.Length == 0)
            {
throw new Exception("Phone number is required");
            }
        }
    }
</icustomer>
```

Now the base class will internally point to the general validation interface. Now the “Customer” and “Lead” class has no idea what kind of validation strategy will he be executing.

```
public abstract class CustomerBase : BoBase, ICustomer
{
// Code removed for simplification
private IValidationStratergy<icustomer> _ValidationType = null;
public CustomerBase(IValidationStratergy<icustomer> _Validate)
{
            _ValidationType = _Validate;
}
public IValidationStratergy<icustomer> ValidationType
{
get
            {
return _ValidationType;
            }
set
            {
                _ValidationType = value;
            }
        }
}
}
</icustomer></icustomer></icustomer>
```

Now in the factory class we can create any entity and inject any validation in to it. You can see we have created a customer class and injectedthe all validation class object in to it. In the same way we have created a lead class and injected lead validation object in to.

```
container.RegisterType<icustomer, customer="">("Customer", new InjectionConstructor(newCustomerAllValidation()));

container.RegisterType<icustomer, lead="">("Lead", new InjectionConstructor(new LeadValidation()));
</icustomer,></icustomer,>
```

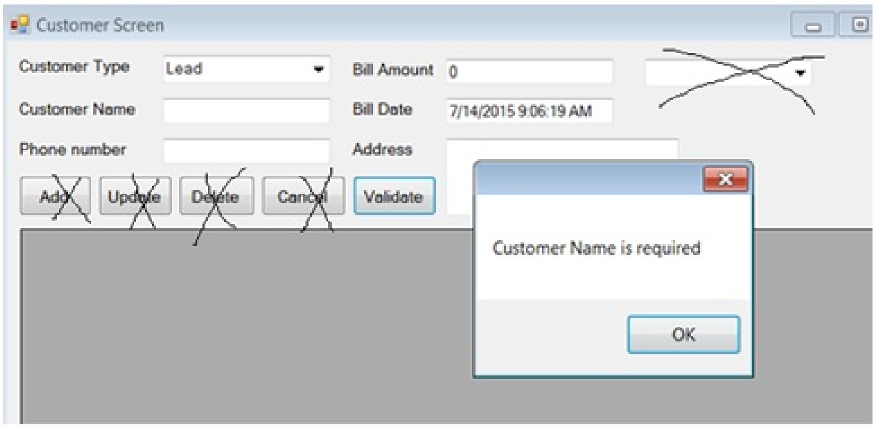
Below goes the full code for Factory.

```
public static class Factory<anytype>
{
static IUnityContainer container = null;

public static AnyType Create(string Type)
        {
if (container == null)
            {
                container = new UnityContainer();
                container.RegisterType<icustomer, customer="">("Customer",
                                new InjectionConstructor(newCustomerAllValidation()));
                container.RegisterType<icustomer, lead="">("Lead",
                                    new InjectionConstructor(new LeadValidation()));

            }
return container.Resolve<anytype>(Type.ToString());
        }

    }
</anytype></icustomer,></icustomer,></anytype>
```

Pattern 5 Strategy pattern: - This is a behavioral design pattern which helps to select algorithms on runtime.

Consuming and making it live

If you remember we already had a UI so I have just consumed the updates in my UI.



Customer Name is required

Below is the code for button validate event.

```
// Create customer or lead type depending on the value of combo box
icust = Factory<icustomer>.Create(cmbCustomerType.Text);

// Set all values
icust.CustomerName = txtCustomerName.Text;
icust.Address = txtAddress.Text;
icust.PhoneNumber = txtPhoneNumber.Text;
icust.BillDate = Convert.ToDateTime(txtBillingDate.Text);
icust.BillAmount = Convert.ToDecimal(txtBillingAmount.Text);


// Call validate method
icust.Validate();
</icustomer>
```

What's in the next part?

In the next part we would be covering the following five patterns:-

- Creating DAL layer using ADO.NET and decoupling them using Repository pattern , UOW and adapter pattern.
- Using template pattern in ADO.NET code to reuse command and connection objects.
- Use Façade pattern to simplify UI code.

Below is a nice Design Pattern youtube video which explains step by step how to use Design pattern in C# projects.

License

Share

About the Author

Shivprasad koirala

Architect https://www.questpond.com
India

Do not forget to watch my Learn step by step video series.

Learn MVC 5 step by step in 16 hours
Learn MVC Core step by step
Learn Angular 2/4 for beginners (step by step)
Learn AngularJS 1.x Step by Step
Learn Xamarin Mobile Programming Step by Step
Learn Design Pattern in 8 hours
Learn C# in 100 hours series
Learn SQL Server in 16 hours series
Learn Javascript in 2 hours series
Learn MSBI in 32 hours
Learn SharePoint Step by Step in 8 hours
Learn TypeScript in 45 minutes

You may also be interested in...

Designing For DevOps

Building Reactive Apps

Learn MVC (Model view controller) Step by Step in 7 days – Day 2

SAPrefs - Netscape-like Preferences Dialog

Learn Microsoft Business intelligence step by step – Day 1

Generate and add keyword variations using AdWords API

Comments and Discussions

52 messages have been posted for this article Visit **https://www.codeproject.com/Articles/1009532/Learn-Csharp-Design-patterns-step-by-step-with-a-p** to post and view comments on this article, or click **here** to get a print view with messages.