EXPLORING DESIGN PATTERN FOR DUMMIES

By Sourav Kayal

Exploring Design Pattern for Dummies

This free book is provided by courtesy of C# Corner and Mindcracker Network and its authors.
Feel free to share this book with your friends and co-workers.
Please do not reproduce, republish, edit or copy this book.

Sourav Kayal
.NET developer

Sam Hobbs
Editor, C# Corner

This book is a basic introduction to "**Exploring Design Pattern for Dummies**" basically for beginners who want to learn about Design Pattern. After completing this book, you will understand:

Basic introduction to all types of Design Patterns

Table of Contents

Welcome to the Design Pattern for Beginners. Here, I will be discussing various design patterns (in other words a few of the most popular) basically targeting beginners. Most of the patterns I will discuss in my own style and my own words. And before starting any pattern, we will try to find the basic need for it. So, let's start our journey with a very nice quote.

"It's easy to walk on water and develop software from specifications, when both are frozen."

Hey, that's not mine!! And I have forgotten from where I read it. My poor memory says I read it somewhere in someone's blog. If anyone has any information please put a few keystrokes in the comments section.

Anyway, let's return to the subject of this article. Really, it's very easy to develop software from specifications and when the requirements are constant. But by nature people are not happy with constant and fixed needs. (Yes that's why language designers created variables! Ha... Ha...)

And here design patterns come in to play. If we implement a proper design pattern then we need not worry much when new requirements are added to the old ones. And here lies the importance of design patterns.

As I said earlier, this article series is targeted to young developers and if you fall in this category then this introduction is enough to make explain "Why a design pattern is very essential".

I am grateful to "Shivaprasad koirala Sir"; yes, from your video tutorial I have heard the word "Design pattern" for the first time and many-many thanks for providing the first gear of my design pattern journey.

Ok, that's a long introduction. Let's start with a very common and easy design pattern called "Singleton Design Pattern".

Singleton Design Pattern

Let's learn why the Singleton Design Pattern is implemented. And then we will see how to implement it in C#.

Basic need: Think, there is a hit counter in any web application. (Yes I know you have already seen many examples like this). Now, by nature when a visitor hits a web application it will increase by one. This is one scenario where we can implement the Singleton Design Pattern.

Or think about another situation where you want to share a single instance of a class across various threads.

Basically we can implement the singleton pattern in one of two ways.

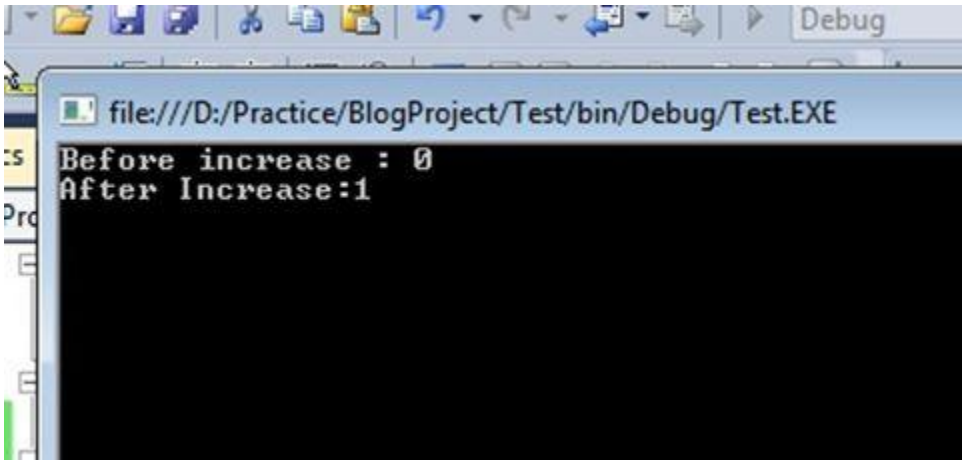
1. Make the constructor private so that no one can create an object of the singleton class

Have a look at the following code to understand the basic concept.

```
using System;
using System.Collections;
using System.Data.SqlClient;
namespace Test1
{
   class singleTon
   {
```

```
        public static int counter = 0;
        private singleTon()
        {
            //Private constructor will not allow create instance
        }
        public static int Returncount()
        {
            return counter;
        }
        public static void IncreaseCount()
        {
            counter++;
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            //singletone design pattern
            Console.WriteLine("Before increase : " + singleTon.counter);
            singleTon.IncreaseCount();
            Console.WriteLine("After Increase:" + singleTon.counter);

            Console.ReadLine();
        }
    }
}
```

Here is some sample output:



Ok, you may thinking, Sourav!! We are not creating any object at all in the above example but in the introduction why did you say that we will create a single object and share it across threads. Yes we will get to that in the next example but believe me it's also a design style of a singleton pattern. Ultimately we are sharing a single class across various threads (though here we did not implement any multi-threading concept). Oh! Are you worried about deadlock problems? Ok let's see the next example.

2. Create single instance and share across threads

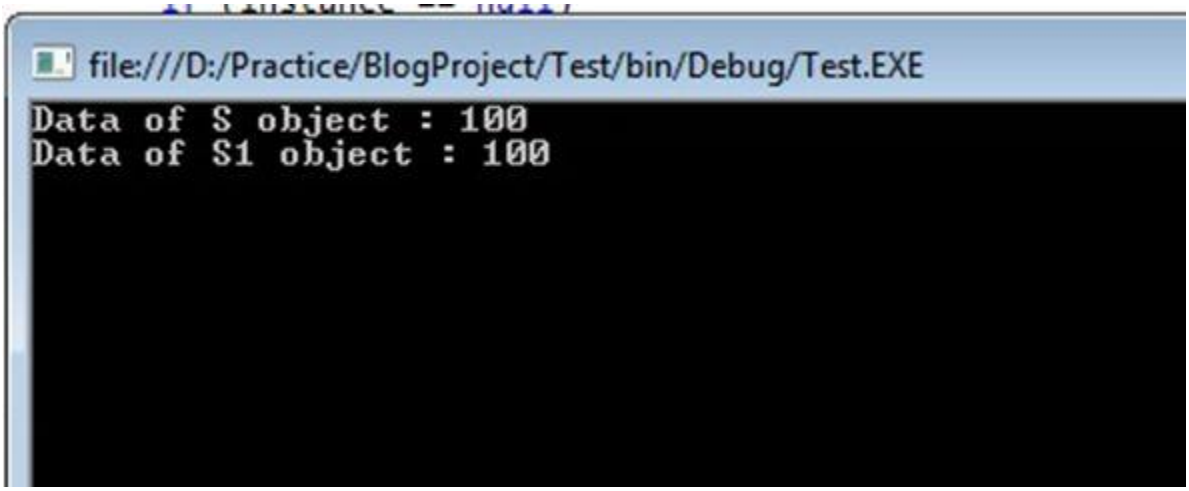
In this example we will implement a singleton class in such a way that only one instance of that class is possible and if the second request comes then it will return the previously created object only. Have a look at the following code.

```
using System;
using System.Collections;
using System.Data.SqlClient;

namespace Test1
{
   public sealed class singleTon
   {
      public Int32 Data = 0;
      private static singleTon instance;
      private static object syncRoot = new Object(); //For locking mechanism

      private singleTon() { } //Private constructor

      public static singleTon Instance //Property
      {
         get
         {
            if (instance == null)
            {
               lock (syncRoot)
               {
                  if (instance == null)
                     instance = new singleTon();
               }
            }

            return instance;
         }
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         //singletone design pattern
         singleTon s = singleTon.Instance;
         s.Data = 100;
         Console.WriteLine("Data of S object : " + s.Data);

         singleTon s1 = singleTon.Instance;
         Console.WriteLine("Data of S1 object : " + s.Data);
         Console.ReadLine();
```

```
    }
  }
}
```

Here we have defined a member called instance within the singleton class. And within the class property (yes it's also Instance, but don't be confused) we are checking whether instance is null using a condition in an if-statement. If it is null then no instance is created of this class and if it is not null then an instance has already been created and it's time to return the previously created instance.

Here is sample output of this example.



Now, we will analyze the Main() function here. Have a look at the following code:

```
singleTon s = singleTon.Instance;  //First instance
s.Data = 100; //Data of first instance
Console.WriteLine("Data of S object : " + s.Data);
singleTon s1 = singleTon.Instance;  //Again request and return old instance
Console.WriteLine("Data of S1 object : " + s.Data);  //Show data of old instance
Console.ReadLine();
```

In the first two lines we are creating the first instance of the singleton class and set the data value. Now in the fourth line we are again requesting a new object of the singleton class but there is no chance to get a new one (siince this is our main target) and from the property the old one is returned. Yes that's why in the fist object we have set Data=100 and it has been reflected in the second instance, hence prooving that no new object has been created.

Factory Design Pattern

Before writing this I searched using Google for the keywords "factory design pattern" and as expected I many good articles with nice examples. Then I started to think "Why write one more?" OK, then I thought; let's check each of them one by one. I proceeded to visit each of the top 10 (from the first page). They really are great but stuffy enough and all the articles maintain a common style and are the same type of article. Somehow there is an absence of reality.

And that led me to write one more on the same topic. Here we will try to understand that topic with realistic examples

and a little fun. And we will enjoy the learning.

So, let's start with the Factory Design Pattern. As promised "We will try to understand the basic need at first before starting any design pattern.".

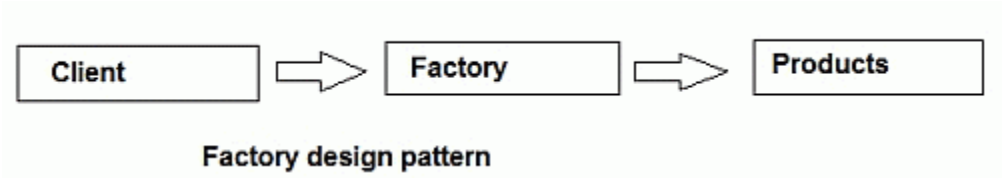
Let me start with a small practical event that I experienced just a few days before (yes just a few days). My present office is within one Tech Park and I see many ITians (yes, you are right, they work in various IT fields in many company's within the same tech park) going towards the food court at lunch time. Of course I am one of them. There is a long staircase in the middle of the food court (since the food court is underground).

One fine lunch time, I was going for lunch as usual and I was hearing a beautiful romantic song of love from someone's voice. I observed that a handicapped fellow was singing and going just before me using his crutch (you know, walking stick). I was thinking being a physical challenger how happy he is? (Dear reader, I am not making fun of his physical disability. I Just mean to say, see how happy he is? However sometimes we may not be a complete person.)

OK, now the staircase came and it's time to climb down it. I noticed that the guy (singing a song of love) just folds up his walking stick and by holding the stair railing began to climb down. I was thinking how the nice stick is? Depending on demand it's behaving.

My story ends here and the walking stick is nothing but one example of a factory class. A factory class is a class that serves the client's demands and depending on the requirements it supplies the proper form.

OK, we have talked too much; let's learn the simple structure of the Factory Design Pattern. We will initially see the logical stricture of the Factory Design Pattern.



Factory design pattern

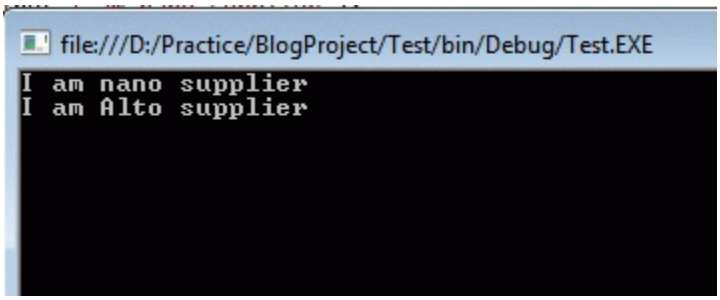
As the name implies, a factory is a place where manufacturing happens. For example, a car factory where many types of car manufacturing happens. Now you want a certain model and it's always possible to produce it from this car factory. Again your friend's taste is different from yours. He wants a different model; you can request a car factory again to bring a smile to your friend's face.
OK, one thing is clear from our discussion; the factory class is nothing but a supplier class that makes the client happy by supplying their demand.

How to will implement it in C# code? OK let's see the following example.

```
using System;
using System.Collections;
using System.Data.SqlClient;
using System.Threading;

namespace Test1
{
   public interface ISupplier
   {
```

```
    void CarSupplier();
}
class namo : ISupplier
{
    public void CarSupplier()
    {
        Console.WriteLine("I am nano supplier");
    }
}
class alto : ISupplier
{
    public void CarSupplier()
    {
        Console.WriteLine("I am Alto supplier");
    }
}
class CarFactory
{
    public static ISupplier GiveMyCar(int Key)
    {
        if (Key == 0)
            return new namo();
        else if (Key == 1)
            return new alto();
        else
            return null;
    }
}
class Program
{
    static void Main(string[] args)
    {
        ISupplier obj = CarFactory.GiveMyCar(0);
        obj.CarSupplier();
        obj = CarFactory.GiveMyCar(1);
        obj.CarSupplier();
        Console.ReadLine();
    }
}
}
```

And here is the output.



Though the code is pretty simple to understand, we will discuss it a little more. At first we have created one interface and implemented that interface within two classes. The classes are nano and alto.

In addition to them there is one more class called CarFactory and it is the hero of the story. If we observe closely, inside the CarFactory class we will find the mechanism to create a car. Though, here we are producing two low-end cars (nano and alto), in the near future we can add many more models. Now here is the original beauty of the factory class.

Try to understand the following few lines carefully. As we indicated previously, we can add many more models in the future. If we add 7 or even 10 (or your preferred number) more models, then the client code will also not be effected by a single line, because we will add code in the factory class, not in the client and will inform the client that, from now on, those models are also available; just send the proper code (such as 0 for nano, 1 for alto) to get them.

Now return to my story of a walking stick. The walking stick was changed in behavior depending on needs. And just now we have seen our factory class also supplying a different form of object depending on needs. Somehow both are sharing a common behavior, right?

OK, now you may think what a useless example this is? Car class? We don't know when our company will get a project from a car merchant and we will implement the CarFactory class there.

Ok, then let me show you a very realistic example that you can implement in your current project, yes tomorrow morning.

Here we will learn how to implement a vendor-independent data access mechanism using a factory class. No, we will not create our own factory class to do that; we will just use a Dbfactory class from the .NET library.

Let's clarify our purpose one more time: "We will implement such a data access mechanism that is able to talk with various database vendors.".

At first we will learn why we need to learn. Today you have developed one software product by targeting one of your clients who uses SQLServer as their database. You have designed and implemented necessary coding for SQLServer. (Yes ADO.NET code, sqlconnection, sqlcommand bla bla..)

Now tomorrow the client may say that we are not happy with SQLServer and we have decided that from now we will use Oracle as our backend database.

The drama starts here. Let me explain the first scene.

By getting this proposal in the mail (from the client) the project manager will call the team lead of this product's team. After an hour of discussion they will make the decision "They need to change the data manipulation policy". It will take 30 days more to fix by 5 resources and the budget for that is a $$ amount.

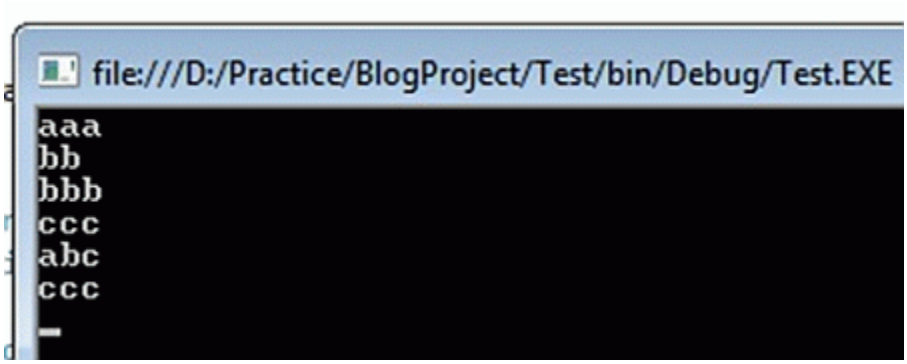
Now the client has received mail from the Software Company and replied: "Why do you people disclose the matter at the very first and we are unable to give a single $ and day to do so. And we want to get it done within this time limit".

OOHhh, it's getting very complex, we will not go farther. Anyway what "If we develop such a data accessing mechanism that will be compatible with all database vendors"? Have a look at the following code.

```
using System;
using System.Collections;
using System.Data.SqlClient;
using System.Threading;
using System.Data.Common;
using System.Data;

namespace Test1
{
  class Program
   {
      static void Main(string[] args)
      {
         DbProviderFactory provider = null;
         DbConnection con = null;
         DbCommand cmd = null;
         DbDataReader rdr = null;
         DataTable dt = new DataTable();
         provider =DbProviderFactories.GetFactory("System.Data.SqlClient");
         con = provider.CreateConnection();   //Create Connection according to Connection Class
         con.ConnectionString = "Data Source=SOURAV-PC\\SQL_INSTANCE;Initial Catalog=test;Integrated
Security=True";
         cmd = provider.CreateCommand();   //Create command according to Provider
         try
         {
            cmd.CommandText = "select * from name";
            cmd.CommandType = CommandType.Text;
            if (con.State == ConnectionState.Closed || con.State == ConnectionState.Broken)
            {
               con.Open();
               cmd.Connection = con;
               using (con)
               {
                  rdr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
                  while (rdr.Read())
                  {
                     Console.WriteLine(rdr["nametest"].ToString());
                     Console.WriteLine(rdr["surname"].ToString());
                  }
               }
            }
         }
         catch (Exception ex)
```

```
        {
            throw;
        }
        finally
        {
            //trn.Rollback();
            con.Dispose();
            cmd.Dispose();
        }
        Console.ReadLine();
    }
  }
}
```

Here is sample output.



You can see that nowhere in the example we have written any database specific ADO.NET code. We have used various methods of the Dbfactory class to create an object depending on Supplier. Have a look at the following code:

```
con = provider.CreateConnection();   //Create Connection according to database provide    cmd =
provider.CreateCommand();   //Create command according to database provider
```

Here the provider is nothing but an object of the DbProviderFactory class and we are using a function like Createconnection() and CreateCommand() to initialize a connection and command object.

Now if you want to change the database then just change the provider name or database supplier name like:

```
provider =DbProviderFactories.GetFactory("System.Data.SqlClient");
```

Here we have provided our database provider as SQLServer and tomorrow, if you want to, you can use MySQL; just modify the code as in the following:

```
provider =DbProviderFactories.GetFactory("MySql.Data.SqlClient");
```

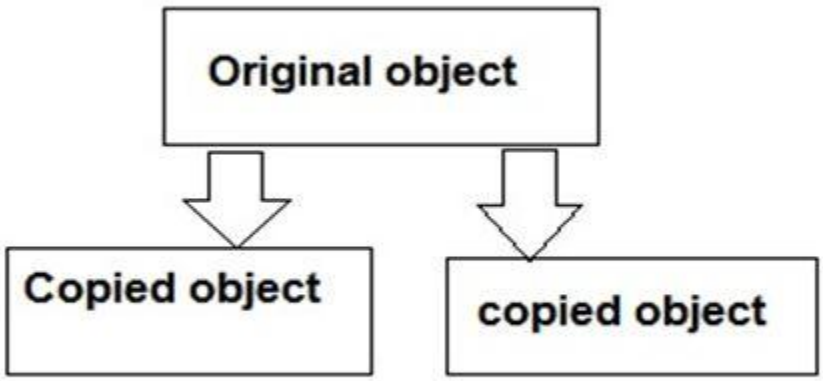
We just need to change the provider name and the rest of the code will work fine. And see again the DbProviderFactory class supplies an object depending on the user's demands and this is the beauty of a factory class.

Prototype Design Pattern

Now we will learn one more very common design pattern called "Prototype Design Pattern".

As the name suggests, prototype means making a copy of something that exists. In this context let me give one nice example. A few days ago one of my friends went to a Tattoo shop to get a Tattoo on his body. (Yes my friend is cool and rocking in nature.) After returning from the shop (with a tattoo in hand) he told me, Sourav, they have some pre-prepared styles that are very less costly but if we want our own design, then they said "it will take time and the cost will be higher". I said Ok. Now consider this for the situation in software development. If the client demands an object that we already have then we can deliver it very quickly and at less cost. In software development there are many scenarios where the same copy of an existing object is frequently needed.

And this is the basic necessity of the Prototype Design Pattern. So, let's clarify the basic requirement "when we need to create the same object again and again then we will implement a Prototype Design Pattern". Here is the simple logical diagram of the Prototype Design Pattern:



Now, one big misconception occurs when we (basically junior programmers, including me) creates the same object in C# or tries to copy one object into another. Let's learn the problem at first. Then we will see how to solve it.

Have a look at the following code; here we will try to create a copy of an existing object.

```
using System;
using System.Collections;
using System.Globalization;

namespace Test1
{
   class Test
   {
      public string Name;
   }
   class Program
```

```
    {
        static void Main(string[] args)
        {
            Test obj1 = new Test();
            obj1.Name = "sourav";

            Test obj2 = obj1; // Trying to make copy in obj2
            obj2.Name = "c-sharpcorner.com";
            Console.WriteLine(obj1.Name);
            Console.ReadLine();
        }
    }
}
```

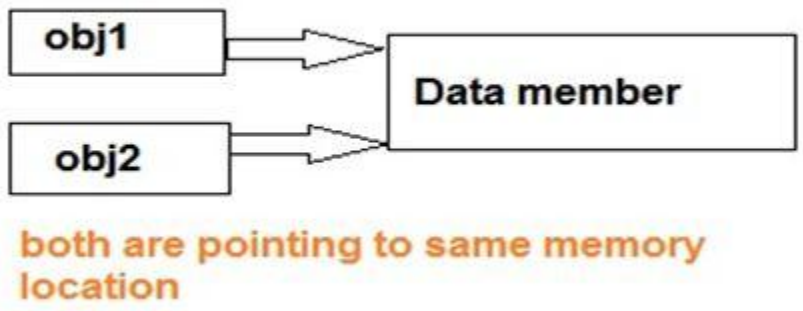
And here is the output:

In this code we are trying to make a copy of one object into another by the following statement:

Test obj2 = obj1;

And we are thinking that one copy of obj1 is getting stored in obj2. But due to the primary concept of the object being a reference type, in other words if we initialize one object to another object then they both point to the same memory location where the actual value is present.

Now the next line is the most interesting. We are initializing a property (data member) of obj2. Here we will see the concept behind the screen. When we initialize obj1 to obj2, then both are:



They both point to the same memory location. (Generally in heap memory and obj1 and obj2 are created on the top of

the stack.) In the next line, when we are setting the property of obj2, it is automatically reflected to obj1. And that is the reason why, when we are trying to print the property of obj1 (initialized to "Sourav") we are getting the value of obj2 (c-sharpcorner.com).

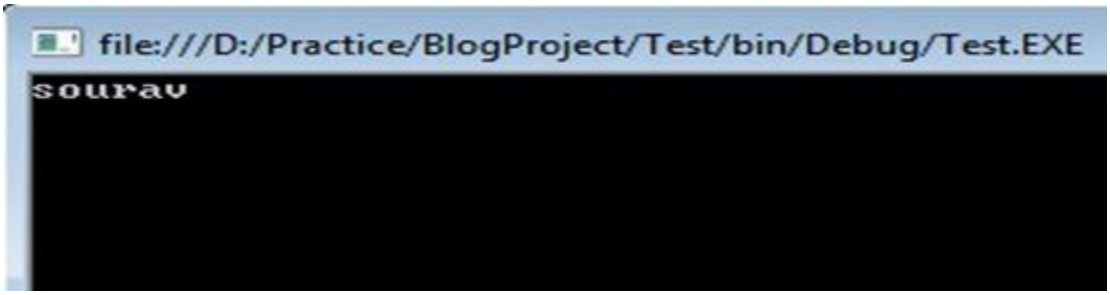
Now this is the problem and we are sure that in this example not at all objects are created from the same copy. Then what is the solution?

Let's use MemberwiseClone() function

Yes, this is the solution. In the following example we will see how to implement the MemberwiseClone() function to copy the same object to another object.

```
using System;
using System.Collections;
using System.Globalization;

namespace Test1
{
   class Test
   {
      public string Name;
      public Test CloneMe(Test t)
      {
         return (Test)this.MemberwiseClone();
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
         Test obj1 = new Test();
         obj1.Name = "sourav";

         Test obj2 = obj1.CloneMe(obj1); // Trying to make copy in obj2
         obj2.Name = "c-sharpcorner.com";
         Console.WriteLine(obj1.Name);
         Console.ReadLine();
      }
   }
}
```

Here is the output.



Oh! At last we have found the right solution to make a copy of the object. And this an example of the Prototype Design Pattern.

Hmm...You are thinking, okay "I understood Prototype pattern but the example is meaningless". Dear reader, I did not forget my promise that I made in my first article of the Design Pattern Series. (If you want to hear it please visit here.) Let's talk about a scenario where the Prototype Design Pattern is relevant.

Consider that you want to create one copy of an object multiple times. For your birthday you want to send an invitation letter to your friends, now the content and sender name will remain the same whereas the recipient name will only change. In this situation we can use a prototype of the invitation card for multiple friends.

Take another example, where one object will be created after heavy database operations (as we know, database operations are very costly) and we want that object 100 times. What if we make a clone of the same object by the Prototype Design Pattern rather than hitting the database 100 times? Good solution right?

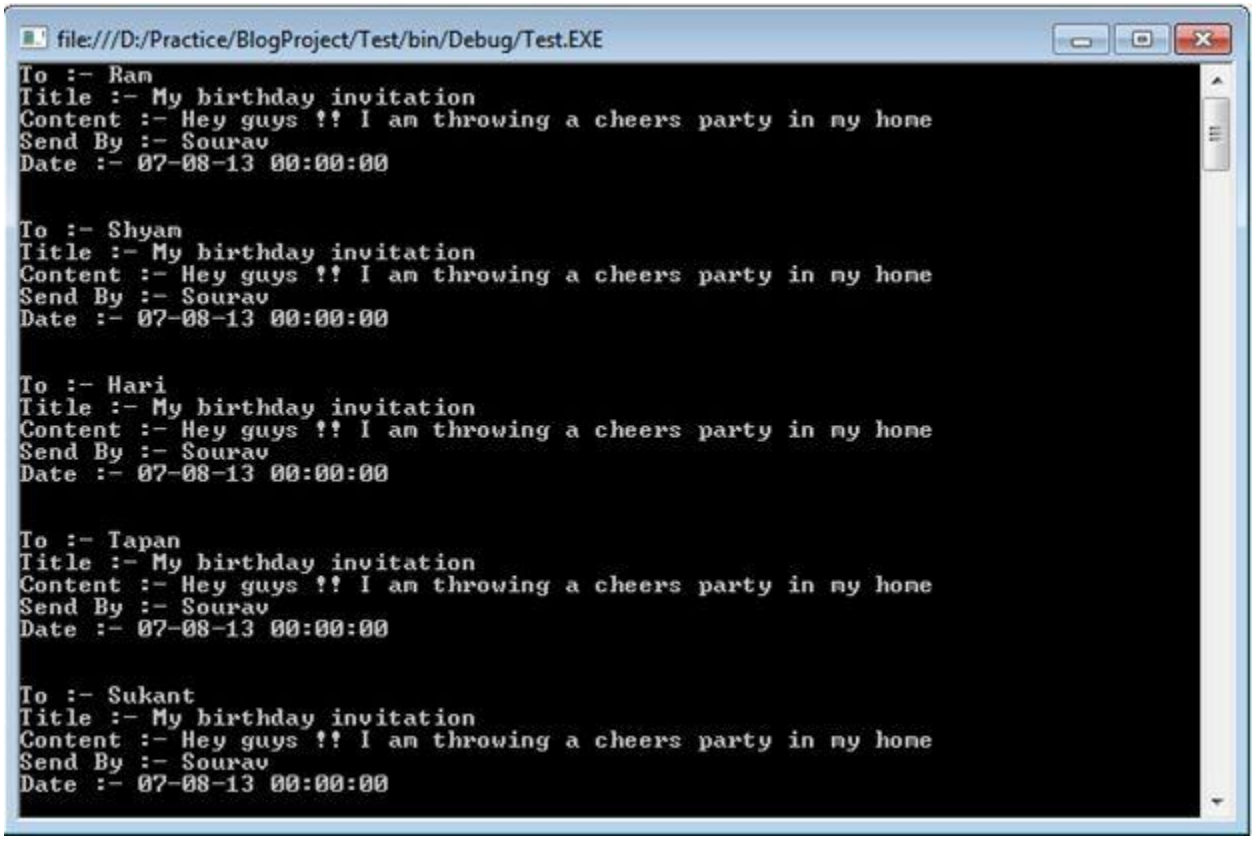
Ok, I am talking too much. Let's implement our first scenario in C# code. Here is the sample code.

```
using System;
using System.Collections;
using System.Globalization;

namespace Test1
{
   class InvitationCard
   {
      public String To;
      public String Title;
      public String Content;
      public String SendBy;
      public DateTime Date;
      public String p_To
      {
         get { return To; }
         set { To = value; }
      }

      public String p_Title
      {
         get { return Title; }
```

```
        set { Title = value; }
    }
    public String p_content
    {
        get { return Content; }
        set { Content = value; }
    }
    public String p_SendBy
    {
        get { return SendBy; }
        set { SendBy = value; }
    }
    public DateTime p_Date
    {
        get { return Date; }
        set { Date = value; }
    }
    public InvitationCard CloneMe(InvitationCard obj)
    {
        return (InvitationCard)this.MemberwiseClone();
    }

}
class Program
{
    static void Main(string[] args)
    {
        InvitationCard obj1 = new InvitationCard();
        obj1.p_To = "Ram";
        obj1.p_Title = "My birthday invitation";
        obj1.p_content = "Hey guys !! I am throwing a cheers party in my home";
        obj1.SendBy = "Sourav";
        obj1.p_Date = Convert.ToDateTime(DateTime.Now.ToShortDateString());
        //Here our first object has created

        InvitationCard[] objList = new InvitationCard[5];
        String[] nameList = { "Ram", "Shyam", "Hari", "Tapan", "Sukant" };
        int i = 0;
        foreach (String name in nameList)
        {
            //objList[i] = new InvitationCard();
            objList[i] = obj1.CloneMe(obj1);
            objList[i].p_To = nameList[i];
            i++;
        }

        // Print all Invitation Card here
```

```
        foreach (InvitationCard obj in objList)
        {
            Console.WriteLine("To :- " + obj.p_To);
            Console.WriteLine("Title :- " + obj.p_Title);
            Console.WriteLine("Content :- " + obj.p_content);
            Console.WriteLine("Send By :- " + obj.p_SendBy);
            Console.WriteLine("Date :- " + obj.Date);
            Console.WriteLine("\n");
        }
        Console.ReadLine();
    }
  }
}
```

And here is the output of what we expected:



Decorator Design Pattern

One more very popular structural design pattern called the Decorator Design Pattern. The name itself implies that this is something related to decoration. Yes, in the Decorator Design Pattern we will be implementing one object in through various steps and in each and every step we will add a little feature to it.

Why decorator pattern?

Before beginning the technical discussion and examples we will learn the basics of the Decorator Design Pattern and the primary need for it.

If there is a need to produce one object with many features but not all the features are needed all the time then the Decorator Design Pattern is useful. For example think about a car beautification shop. Where a new car goes to and according to the owner's demand the shop owner beautifies their car.

One owner may like to fit AC into his car where another owner may not like AC, he might instead want a sound system (Hmm, he likes music..) in his new car. Now, the shop owner has all the facilities, and according to car owner's demand he decorates the car. This is where the Decorator Design Pattern can be a solution.

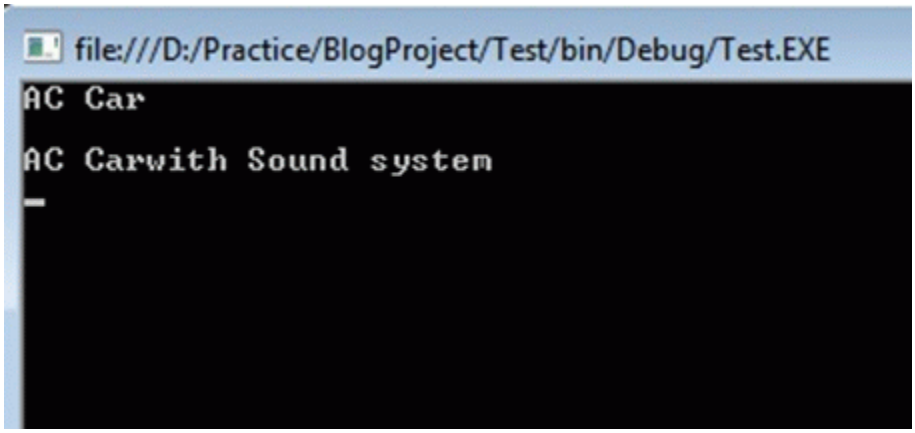
Let's implement our car problem using the Decorator Design Pattern. Have a look at the following code:

```
using System;
using System.Collections;
using System.Globalization;
using System.Data.SqlClient;
using System.Data;
namespace Test1
{
   public class Car
   {
      public virtual void CarType()
      {
         Console.WriteLine("Simple Car");
      }
   }
   public class WithAC : Car
   {
      public override void CarType()
      {
         //base.CarType();
         Console.Write("AC Car");
      }
   }
   public class WithSoundSystemAndAC : WithAC
   {
      public override void CarType()
      {
         base.CarType();
         Console.WriteLine("with Sound system");
      }
   }
   class Program
   {
      static void Main(string[] args)
```

```
        {
            Car objCarAC = new WithAC();
            objCarAC.CarType();
            Console.WriteLine("\n");
            Car objCarAll = new WithSoundSystemAndAC();
            objCarAll.CarType();
            Console.ReadLine();
        }
    }
}
```

Here, Car is the base class and at first we have derived a class called withAC from the Car class and in the next class we have derived from WithAC classes called WithSoundSystemAndAC.

Now, if anyone needs only AC for their Car then he will be happy by creating an object from the WithAC class. And if you want to fit both an AC and Sound system into your car then just create one object from the WithSoundSystemAndAC class. With this implementation the shop owner can make everyone happy. Here is the sample output.



Now, you may ask the question, what if someone demands only a Sound System in his car but not AC. The solution is derive one class from the Car class where we will implement only a Sound System, no other facility (in other words AC). Here is the simple implementation, the same as the WithAC class.

```
public class WithSoundSystem:Car
{
    public override void CarType()
    {
        //base.CarType();
        Console.Write("Car with only sound system");
    }
}
```

Composite Design Pattern

The Composite Design Pattern always forms a tree structure with the objects, where each root element contains sub nodes called child nodes and child nodes contain leaf nodes (last label node). Leaf nodes do not contain any elements.

Now, in this object hierarchy we can point any node and from this node we can traverse all other nodes.

For example, think about an employee stricture of an organization. We can easily represent it by a tree structure and if we start from the CEO (the root of the organization) then we can reach leaf employees by traversing through the middle-level managers.

Take another example of a Composite Design Pattern. Think about the folder structure of your computer. Each drive contains many folders and each folder contains many subfolders and each subfolder contains many files. Here the files are nothing but the leaf nodes. A node from any drive partition you may traverse down to any file by a couple of mouse clicks.

Why composite pattern?

The Composite Design Pattern is useful when individual objects as well as a group of those kinds of objects are treated uniformly.

Hmm.., bookish definition. Let's think of a scenario where we want to draw a few shapes in the same canvas. And we will draw all the components in the same way (by calling a respective function). We will try to implement it with a small working example. Have a look at the following code:

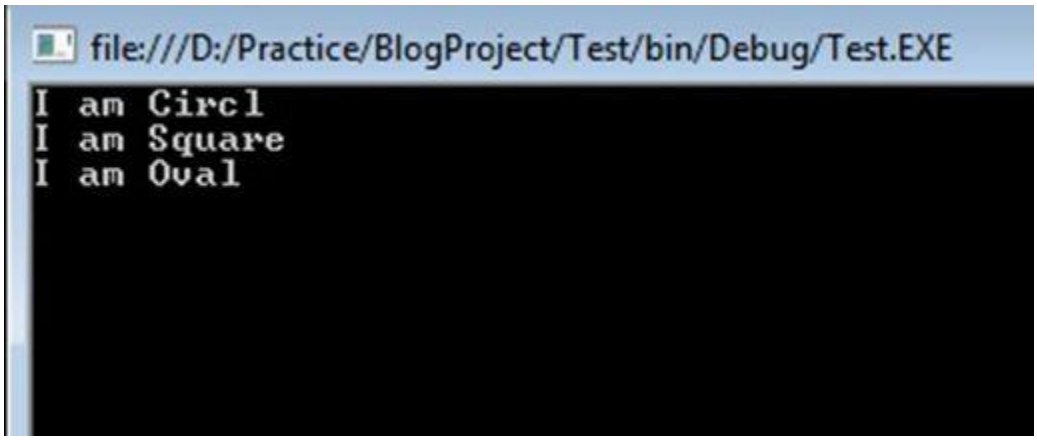
```
using System;
using System.Collections;
using System.Globalization;
using System.Data.SqlClient;
using System.Data;

namespace Test1
{
   interface IDraw
   {
      void Draw();
   }
   class Circle : IDraw
   {
      public void Draw()
      {
         Console.WriteLine("I am Circl");
      }
   }

   class Square: IDraw
   {
      public void Draw()
      {
         Console.WriteLine("I am Square");
      }
   }
   class Oval : IDraw
```

```
    {
        public void Draw()
        {
            Console.WriteLine("I am Oval");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ArrayList objList = new ArrayList();
            IDraw objcircl = new Circle();
            IDraw objSquare = new Square();
            IDraw objOval = new Oval();

            objList.Add(objcircl);
            objList.Add(objSquare);
            objList.Add(objOval);

            foreach (IDraw obj in objList)
            {
                obj.Draw();
            }
            Console.ReadLine();
        }
    }
}
```

Here, we have created a draw interface and three classes are deriving from them. Now, try to match this class hierarchy with the basic structure of the Composite Design Pattern. Here the IDraw interface is the root node and since three classes are derived from it (here implemented) and all of them are child nodes.

Each and every child class contains the same function definition and that's why we are able to call the Draw() function from each class within the for loop.

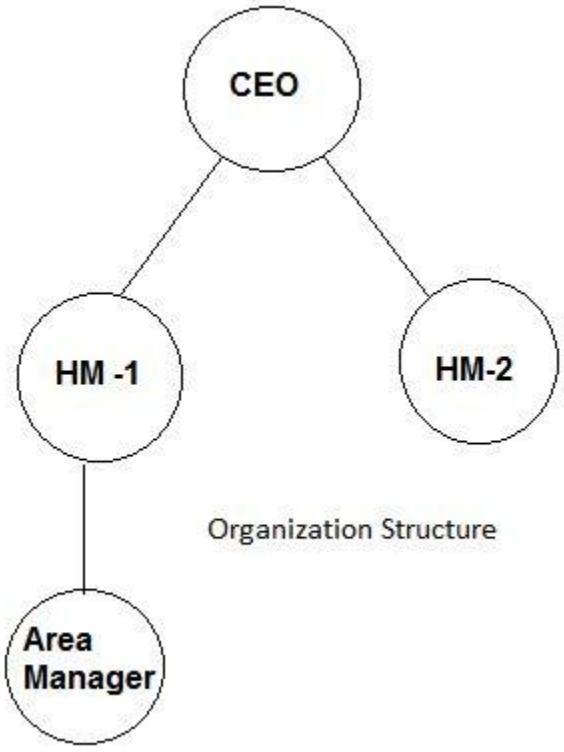
Here is sample output of this example:



In the next example we will implement one organization designation scenario in the Composite Design Pattern. We will assume that it is a small organization where employees are categories only in three levels.

CEO
Head Manager
Area Manager

Now, in this example we have implemented the following tree structure.



Organization Structure

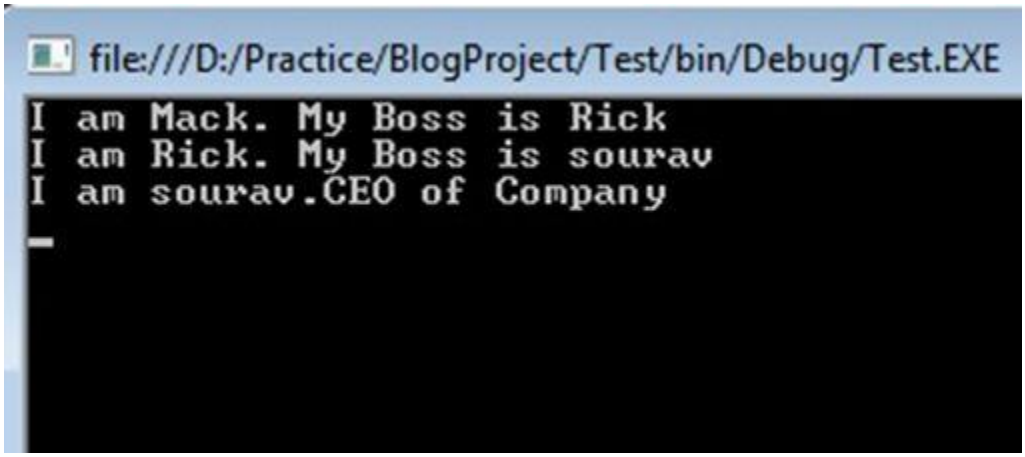
Since this is the tree structure we can reach any leaf node from any non-leaf node. For example if we start from the root (CEO) node then by traversing the HM (Head Manager) nodes we can easily reach an Area Manager node.

We will consider each node as a class in a sample example. Each class will also contain a common function called Designation(). In the following example we will start from a leaf node and will reach to the root node. Have a look at the following implementation.

```
using System;
using System.Collections;
using System.Globalization;
using System.Data.SqlClient;
using System.Data;
using System.Collections.Generic;
namespace Test1
{
   interface IEmployee
   {
      void Designation();
   }
   class CEO :IEmployee
   {
      public virtual void Designation()
      {
         Console.WriteLine("I am sourav.CEO of Company");
      }
   }
   class HeadManager_1:CEO,IEmployee
   {
      public override void Designation()
      {
         Console.WriteLine("I am Rick. My Boss is sourav");
      }
   }
   class HeadManager_2 : CEO,IEmployee
   {
      public override void Designation()
      {
         Console.WriteLine("I am Martin. My Boss is sourav");
      }
   }

   class AreaManager:HeadManager_1,IEmployee
   {
      public new void Designation()
      {
         Console.WriteLine("I am Mack. My Boss is Rick");
      }
   }

   class Program
   {
```

```
        static void Main(string[] args)
        {
            IEmployee AreaManager = new AreaManager();
            IEmployee Hm_1 = new HeadManager_1();
            IEmployee CEO = new CEO();
            List<IEmployee> objEmployee = new List<IEmployee>();
            objEmployee.Add(AreaManager);
            objEmployee.Add(Hm_1);
            objEmployee.Add(CEO);

            foreach (IEmployee O in objEmployee)
            {
                O.Designation();
            }
            Console.ReadLine();
        }
    }
}
```

Here is a sample example.



Adaptor Design Pattern

Now we will discuss a very important design pattern called "Adaptor Design Pattern". As the name suggests, it performs operations very similar to adaptors (Hum; laptop adaptor). Before starting the explanation and example code we will try to understand the real scenario where the Adaptor Design Pattern is relevant.

Why adaptor pattern?

A few years ago I bought a mobile phone (at that time it was a little costly and a decent one with very few features) and had been using it for a couple of years. One day I discovered that the charger for the phone was not working, I went to the mobile shop with both charger and phone. The shop checked both and told me "The charger is gone". Alas! Again I need to buy a new one. The next statement of shopkeeper made me more sorrowful, "This type of charger is out of the market. Since the model is very old, the company has stopped manufacturing it". But he continued, the solution is, you need to buy a multi-pin charger, then you can charge your mobile with that.

I know, while you were reading the paragraph above, you were thinking. Hmm, Sourav is telling a story. Yes my dear reader, this is a story and using it we are attempting to understand the basic need for a Adaptor Design Pattern.

Basically the Adaptor Design Pattern is relevant when two different classes talk with each other. Sometimes a situation may occur like that, you cannot change anything in an existing class and at the same time another class wants to talk with the existing class. In that situation we can implement a middle-level class called an adaptor class and by using the adaptor class both classes are able to talk with each other.

How to implement?

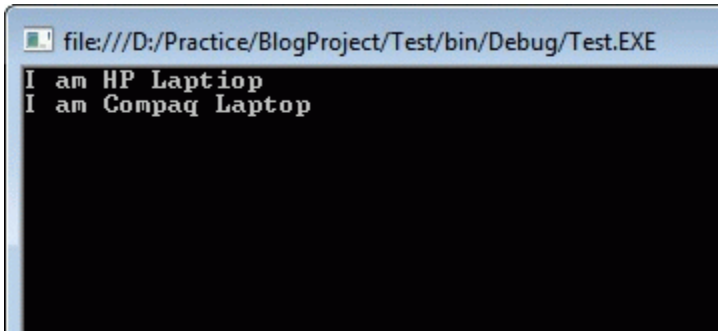
When we use the term adaptor, at first laptop adaptors comes to mind. Right? OK, let's implement an adaptor pattern concept with a few laptop classes.

In the following example we have implemented an ILaptop interface in a few laptop classes. All classes have one common function called ShowModel(). If we call the ShowModel() function then it will show the laptop brand.

```
using System;
using System.Collections;
namespace AdpatorDesign
{
   interface ILaptop
   {
      void ShowModel();
   }
   class HP_Laptop:ILaptop
   {
      public void ShowModel()
      {
         Console.WriteLine("I am HP Laptiop");
      }
   }
   class Sony_Laptop : ILaptop
   {
      public void ShowModel()
      {
         Console.WriteLine("I am Sony Laptop");
      }
   }
   class Compaq_Laptop : ILaptop
   {
      public void ShowModel()
      {
         Console.WriteLine("I am Compaq Laptop");
      }
   }
   class LaptopAdaptor :ILaptop
   {
      public void ShowModel(){}
      public static void ShowModel(ILaptop obj)
      {
         obj.ShowModel();
      }
   }
   class Person
   {
      public void SwitchOn(ILaptop obj)
      {
         LaptopAdaptor.ShowModel(obj);
      }
   }
   class Program
   {
      static void Main(string[] args)
      {
```

```
            Person p = new Person();
            p.SwitchOn(new HP_Laptop()); //On HP Laptop
            p.SwitchOn(new Compaq_Laptop()); //On Compaq laptop
            Console.ReadLine();
        }
    }
}
```

The Person class works with various types of laptops. And the SwitchOn() function will switch on the proper type of laptop through the LaptopAdaptor class. Here is sample output.



Bridge Pattern

Why Bridge Pattern?

Before starting with a technical explanation and example we will try to understand why the Bridge Design Pattern is essential and in which scenario it will be implemented.

As the name suggests, it makes a bridge between two components. Here the component may be two classes or any other entity. So the Bridge Design Pattern basically makes a channel between two components. And in this way it helps to create a de-couple architecture. We can communicate with two classes through the bridge component without changing existing class definitions.

So if we summarize the basic need for the Bridge Design Pattern then we will see that it help us to design a de-couple architecture in the software project. It makes abstraction over implementation.

Let's implement the first example

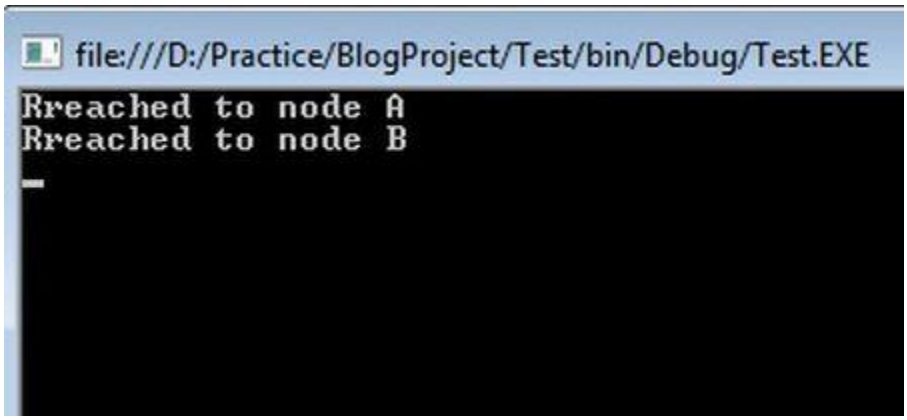
In the following we implement one small example of the Bridge Design Pattern. When we are talking of bridge let's implement a simple bridge between two cities. Here instead of city we will call it node.

In the example we have implemented an Inode interface in both the Node_A and Node_B class. Then in the following, the two node classes, we have defined another class called Bridge, that will communicate between two cities. The ReachTo() function of the bridge class will take us to a specific city (node). Now we are clearly seeing the Program class (Main() function) is talking with any city class (node) through the Bridge class.

```
using System;
using System.Collections;

namespace BridgeDesign
{
   public interface Inode
   {
      void Reach();
   }
   class Node_A : Inode
   {
      public void Reach()
      {
         Console.WriteLine("Rreached to node A");
      }
   }
   class Node_B : Inode
   {
      public void Reach()
      {
```

```
            Console.WriteLine("Rreached to node B");
        }
    }

    class Bridge
    {
        public void ReachTo(Inode obj)
        {
            obj.Reach();
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Bridge br = new Bridge();
            Node_A a = new Node_A();
            Node_B b = new Node_B();
            br.ReachTo(a); //Reach to Node_A
            br.ReachTo(b); //Reach to Node_B

            Console.ReadLine();
        }
    }
}
```

Here is sample output.



Hmm.. What an unrealistic example. Yes this example is only for you to understand the basic implementation of the Bridge Design Pattern. OK, let's work with one useful example (at least related to realistic project development).

The Mail sending operation was implemented by a Bridge Pattern.

Mail sending and notification is a very common task for a software developer. We can now send mail in various ways, for example in a new version of an old project you have developed a mail sending function in C# but in the same project

there is another function to send mail that is written in VB6 (maybe it was written by a senior of your seniors. Ha ..Ha..) and is still working fine.

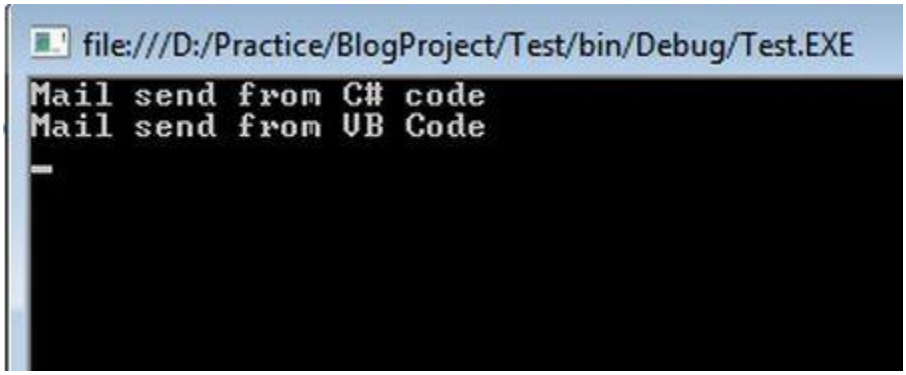
Anyway, your manager said that they want to use both C# and VB versions associated with sending mail from a database (so, in total three different ways). Now, let's implement this scenario using the Bridge Design Pattern. Have a look at the following example.

```
using System;
using System.Collections;

namespace BridgeDesign
{
   public interface IMessage
   {
      void Send();
   }
   class CSharp_Mail : IMessage
   {
      public void Send()
      {
         Console.WriteLine("Mail send from C# code");
      }
   }
   class VB_Mail : IMessage
   {
      public void Send()
      {
         Console.WriteLine("Mail send from VB Code");
      }
   }

   class Databas_Mail : IMessage
   {
      public void Send()
      {
         Console.WriteLine("Mail send from Database");
      }
   }

   class MailSendBridge
   {
      public void SendFrom(IMessage obj)
      {
         obj.Send();
      }
   }

   class Program
```

```
    {
        static void Main(string[] args)
        {
            MailSendBridge mb = new MailSendBridge();
            CSharp_Mail objCS = new CSharp_Mail();
            VB_Mail objVB = new VB_Mail();
            mb.SendFrom(objCS);    //Mail send from C# cod
            mb.SendFrom(objVB);  //Mail Send from VB Code

            Console.ReadLine();
        }
    }
}
```

If you observe closely the body of the Main() function, you will find we are sending just an appropriate object to use the proper mail sending function. Here the MailSendBridge class is creating one abstraction over the implementation of the actual mail sending mechanism that was defined by a different mail sending class.

Here is sample output.



Memento Design Pattern

The Memento Design Pattern is useful when we want to save data in a temporary location and depending on the user's needs us can retrieve the old data.

So, let's think about the scenario with a form (yes, simple Windows Forms form with a few controls) and in the form load event data will be loaded. Now the user may update the loaded data and allowed to save it. And after saving the data the user can restore it (if she wishes).

Now, the problem is, once she updates the form, how will we restore the old data? Don't worry; the Memento Design Pattern will help us.

Let's look behind the scenes

Ok here we will solve our big question. How will we restore the old data after the update is done? Let's apply our common sense, when we are say we will restore the old data that implies that somewhere we are keeping the old data and when necessary we will get it back.

Yes, in the Memento Pattern we will keep a replica of the original object and all modifications will be performed in the original object. Now, when we need the old data to be restored we can get it back from the replica object.

And our problem is solved.

How to implement?

OK, so far we have said what the Memento Design Pattern is and when it is useful. Now for the technical and implementation parts. We have already said that we will keep a copy of the original object. So let's start with a small class implementation.

At first we will design our original class and then we will implement the mechanism to keep a copy of the original object.

The original person class is:

```
public class Person

{

    public String Name { get; set; }

    public String Surname { get; set; }

    MomentoPerson objMPerson = null;


    public Person()

    {
```

```
        Name = "Sourav";
        Surname = "Kayal";
        objMPerson = new MomentoPerson(Name,Surname);
    }
    public void Update(String name, string Surname)
    {
        this.Name = name;
        this.Surname = Surname;
    }
    public void Revert()
    {
        Name = objMPerson.Name;
        Surname = objMPerson.Surname;
    }
}
```

Let's discuss the code snippet. Person() is the constructor and when the object is created it will initialize the Name and surname properties. And within this constructor the most wonderful thing is happening. Here we are maintaining a copy of the original object in another class (yes, in the MomentoPerson class).

In the following constructor, we defined the Update() method to update the original object and beneath Update() we have defined the Revert() method for restoring the modified object.

And here is our MomentoPerson class, which is nothing but a mirror of the original class:

```
public class MomentoPerson
{
    public String Name { get; set; }
    public string Surname { get; set; }
    public MomentoPerson(String Name, String Surname)
    {
```

```
        this.Name = Name;
        this.Surname = Surname;
    }
}
```

We are also seeing that the contents are the same properties as the Person class. And the constructor will be called from the constructor of the original object.

Let's now build the user interface. It's nothing but a simple form containing two TextBoxes.

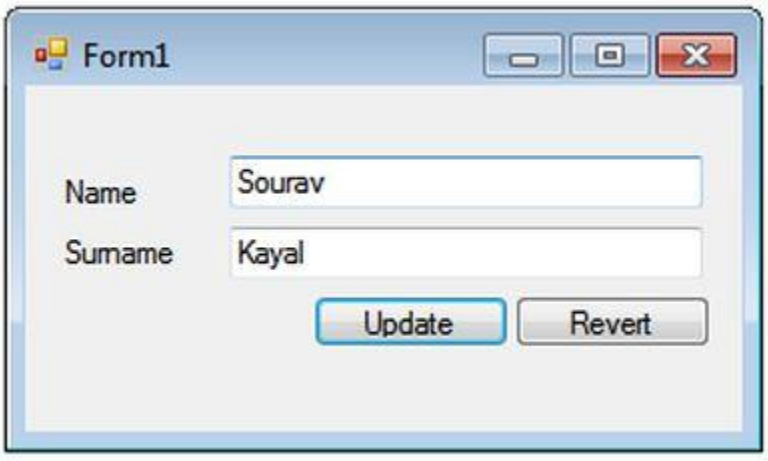
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using DesignPattern;

namespace DesignPattern
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
        Person objP = new Person();

        private void Update_Click(object sender, EventArgs e)
        {
            objP.Update(this.txtName.Text, this.txtSurname.Text);
        }

        public void DisplayCustomer()
        {
            this.txtName.Text = objP.Name;
            this.txtSurname.Text = objP.Surname;
        }
```

```
        private void Cancel_Click(object sender, EventArgs e)
        {
            objP.Revert();
            DisplayCustomer();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            DisplayCustomer();
        }

    }
}
```

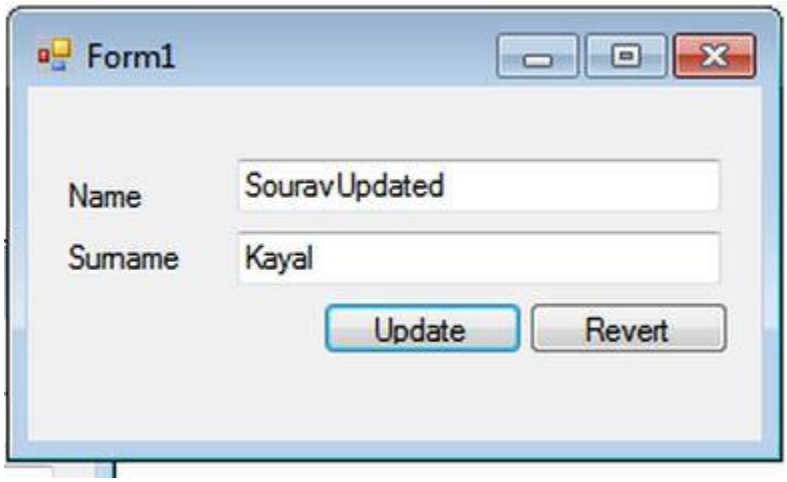
In the form load it will call DisplayCustomer(). And this function will show the values that will be assigned within the constructor.

The Update() function will pass an updated value from/to text boxes and the cancel function will call the revert method that will restore the value of the original object.

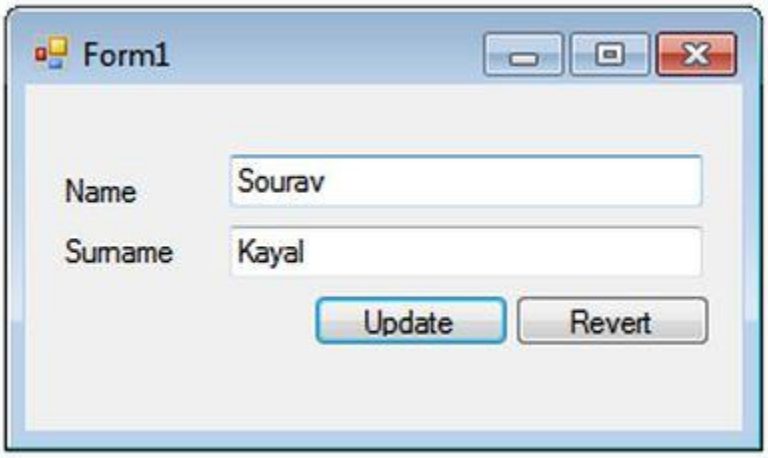
Here is the user interface.



Now we will update those TextBox values.



Again we will restore our old data.



Strategy Design Pattern

Why Strategies Design Pattern?

In this article we will discuss the Strategies Design Pattern. First of all we will try to understand what the Strategies Design Pattern is and in which scenario it is useful. OK, let's think about the situation where we need to make a decision during run time. For example, we want to display a name in an application. In one form we want to display the name with the combination of name+ surname and in another form we want to display surname+ name, in this fashion.

So, how to display the name? Yes, we will make the decision at run time, as needed. Let's think of another scenario. In an automated mailing system we want to send mail to the proper person. When an error occurs, we will study the error and according to classification we would like to send one mail to the relevant department.

If it is a software related error then the mail will go to the software division, if it is a hardware related issue then the mail will go to the hardware division. Now, we don't know which type of error will occur each time. We need to make the decision at run time after analysis of the error.

Let's implement strategy classes

Let's implement our first scenario, where we want to show the combination of name and surname depending on demand (runtime). We will create a Windows application to do that. So, let's create a Windows application and provide a nice name. Then add a .cs file and place the following code into it.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Statergy
{
   public abstract class clsStatergy
   {
      public abstract
         String Display(String Name, String Surname);
   }
   public class clsNameSurname : clsStatergy
   {
      public override String Display(String Name, String Surname)
      {
         return Name + Surname;
      }
   }
   public class clssurnameName : clsStatergy
   {
      public override String Display(String Name, String Surname)
      {
         return Surname + Name;
      }
   }
   public class clsDisplayName
   {
      public String Name = "";
      public String Surname= "";
      private clsStatergy objStatergy;
      public void setStatergy(clsStatergy obj)
      {
         objStatergy = obj;
      }
      public String Show()
      {
         return objStatergy.Display(Name, Surname);
      }
```

```
    }
}
```

Let's explain the code above .At first we created one abstract strategy class, it's nothing but to bring uniformity to the application. Then we have derived two types of display classes and have implemented (better to say overrode) the Display() function in each one of them.

Then another class called clsDisplayName will make the decision which displays the strategy needed to follow.

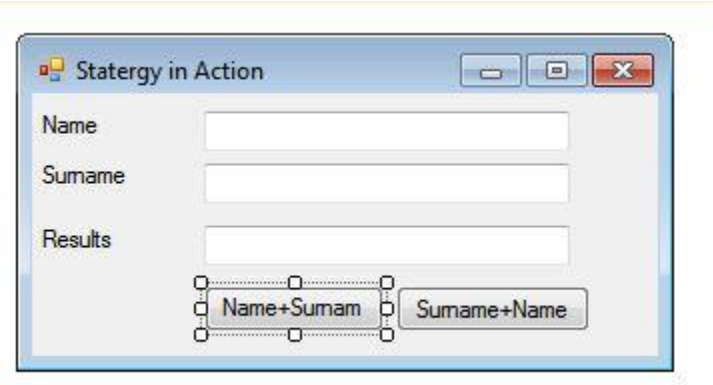
Implement client

Now it's time to implement the client section. Use one Windows Forms form and paste the following code into it.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using Statergy;
namespace WindowsStatergy
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void btnNameSurname_Click(object sender, EventArgs e)
        {
            clsDisplayName obj = new clsDisplayName();
            obj.Name = txtNumber1.Text;
            obj.Surname = txtNumber2.Text;
            obj.setStatergy(new clsNameSurname());
            txtResults.Text = obj.Show().ToString();
        }

        private void btnSurnameName_Click(object sender, EventArgs e)
        {
            clsDisplayName obj = new clsDisplayName();
            obj.Name = txtNumber1.Text;
            obj.Surname = txtNumber2.Text;
            obj.setStatergy(new clssurnameName());
            txtResults.Text = obj.Show().ToString();
        }
    }
```

}

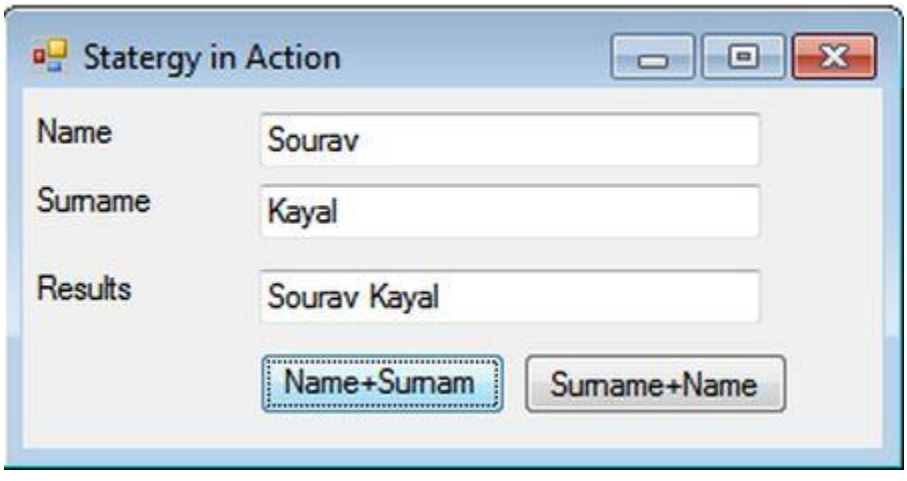
Here is the design mode.



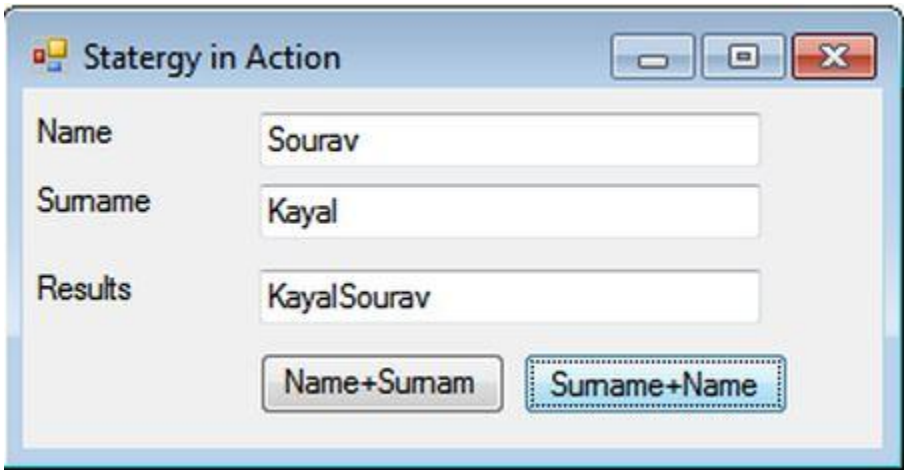
When the user clicks the Name+Surname button after giving the name and surname, it will show the proper combination. If the user needs the surname first then they will use the Surname+Name button.

If we observe the button's click event of each button, we will find that we are creating an object during run time and the object is being set to the strategy class.

Here is the working output of when we press the Name+Surname button.



If we press Surname+Name then the output will show as in the following.



Observer Design Pattern

Let's discuss the importance of the Observer Design Pattern and when it needs to be implemented.

Why observer pattern?

As the name suggests, it's something related to observation. The question is, who is the observer? The observers are nothing but various systems.

The concept is, one or more systems will be the observer simultaneously and if necessary they can start their action. It's like a bodyguard. Right?

Let's talk about a notification system where the user can send notifications in various ways. They may use SMS notification or Mail Notification or Event Log.

Now, all the notification systems will be alive continuosly, and if needed we can use any one of them, or more than one simultaneously. So , if we draw the conclusion, observer pattern is fit that situation where we choose and use systems at run time. Whereas all systems will alive continuosly. Let's try to implement that in code.

Create various notification classes

We are interested in implementing a uniform naming convention. For that we will implement all notification classes from the INotifyObserver Interface. Each notification class will be implementing a Notify() method.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ObserverPattern
{
```

```
    interface INotifyObserver
    {
        void Notify();
    }
    class MailNotify : INotifyObserver
    {
        public void Notify()
        {
            Console.WriteLine("Notify through Mail");
        }
    }
    class EventNotify : INotifyObserver
    {
        public void Notify()
        {
            Console.WriteLine("Notify through Event");
        }
    }
    class SMSNotify : INotifyObserver
    {
        public void Notify()
        {
            Console.WriteLine("Notify through SMS");
        }
    }

}
```

Create notifier class

This is a very interesting and important part of the Observer Design Pattern. We can say clsNotifier (see the following code) is our control room. From here we will control which kind of notification will execute.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Collections;

namespace ObserverPattern
{
    class clsNotifier
    {
        public ArrayList ALNotify = new ArrayList();
        /// <summary>
        /// Add object of notification System
        /// </summary>
        /// <param name="obj">Object is notification class</param>
        public void AddService(INotifyObserver obj)
```

```
        {
            ALNotify.Add(obj);
        }
        /// <summary>
        /// Remove object of notification System
        /// </summary>
        /// <param name="obj">Object of notification Calss</param>
        public void RemoveService(INotifyObserver obj)
        {
            ALNotify.Remove(obj);
        }
        public void ExecuteNotifier()
        {
            foreach (INotifyObserver O in ALNotify)
            {
                //Call all notification System
                O.Notify();
            }
        }
    }
}
```

AddService() and RemoveService() are two functions by which we can add an object of various notification classes and ExecuteNotifier() will call all the Notify() functions from each notification class.

Design client code

This is the last part of the example. We will create client code to set and make decisions about which notification will be fired.

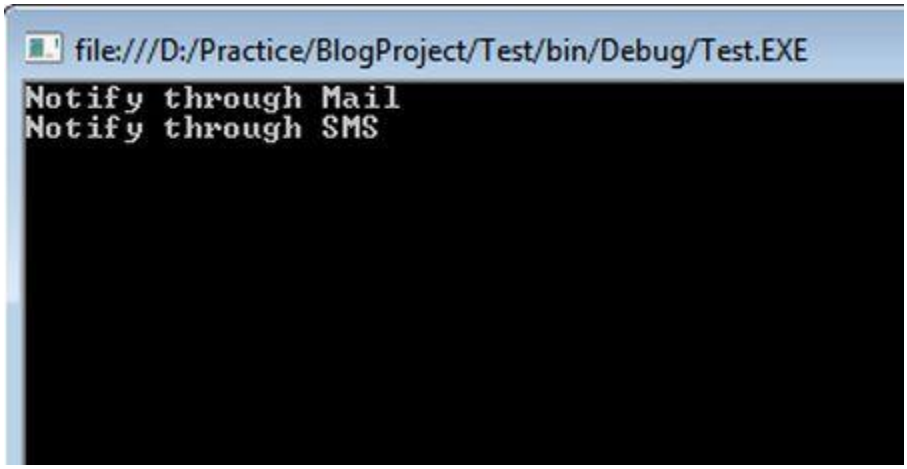
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ObserverPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            //Generate exception to notify all client
            try
            {
                throw new ApplicationException("This is Exception");
            }
            catch (Exception ex)
            {
                INotifyObserver obj1 = new MailNotify();
```

```
            INotifyObserver obj2 = new SMSNotify();
            clsNotifier O = new clsNotifier();
            O.AddService(obj1);
            O.AddService(obj2);
            O.ExecuteNotifier();
        }
        Console.ReadLine();
    }
  }
}
```

Here, we are generating ApplicationException() and within the Catch block we are creating an object to the MailNotification and SMSnotification class.

The executeNotifier() will call all Notify() functions from each notification class.

Output



Implement Decouple Classes in Application

Now we will learn how to implement the Implement Decouple Architecture in applications. Decoupling is very important in application development. It helps up to attach and detach components of any class without affecting another class.

The main goal of decoupling is to reduce dependency. Let's use an example to understand the concept. Think about a Desktop computer. If the monitor does not work then we just unplug it and replace it with a new one, even a different brand would work.

If the hard disk does not work then we just unplug it (hope you know how to unplug it! Ha ..Ha) and install a new one. Now, think of how to decouple each and every component? When we are changing one, it does not at all affect other components and the computer is running happily.

Now, our purpose is to implement the same kind of concept in software development. OK, one thing has been forgotten

for explanation, tight coupling. Yes, it indicates that two entities are very much dependent on each other. If we change one then it will definitely effect another one.

OK, no more discussion. Let's implement the concept of decoupling.

How to implement?

Hmmâ€¦, one big question is, how to implement it? The solution is an interface. As the name suggests, it will be in the middle of two classes.

How it will come?

Let's see with an example. Here we will communicate with two classes using an interface.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace InterfaceService
{
   public interface IService
   {
      void Serve();
   }
   public class Server : IService
   {
      public void Serve()
      {
         Console.WriteLine("Server executes");
      }
   }
   public class Consumer
   {
      IService obj;
      public Consumer(IService temp)
      {
         this.obj = temp;
      }
      public void Execute()
      {
         obj.Serve();
      }
   }
}
```

Let's discuss the code above. At first we have created an Interface called IService. Then we implemented the IService interface within the Server class. Now, let's discuss the Consumer class. Within the Consumer constructor we are assigning an object of the Iservice interface and this object is being set to a local objet of the IService interface.

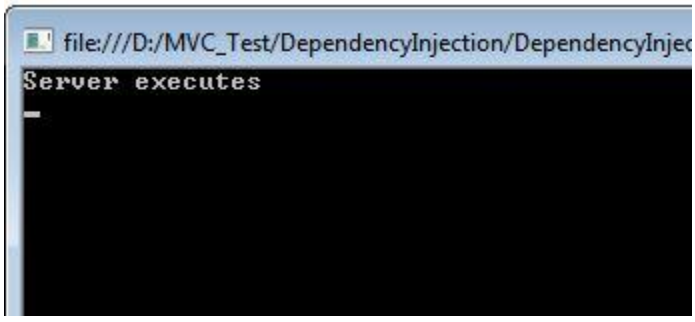
Now you can see that nowhere in the Consumer class have we specified a class name server and hence we are achieving decoupling.

Let's call the consumer class from the client code.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace DependencyInjection
{
   class Program
   {
      static void Main(string[] args)
      {
         Consumer C = new Consumer(new Server());
         C.Execute();
         Console.ReadLine();
      }
   }
}
```

Within the Main function we are creating an object of the Consumer class by passing an object of the server class. And we are calling the Execute function to execute the service.

Here is sample output.



Now the question is, how will it help us? What is the facility for that? Let's change the class name from Server to Server123.

```
public class Server123 : IService
{
   public void Serve()
   {
      Console.WriteLine("Server executes");
   }
}
```

```
public class Consumer
{
   IService obj;
   public Consumer(IService temp)
   {
      this.obj = temp;
   }
   public void Execute()
   {
      obj.Serve();
   }
}
```

See, we have changed the server class name but does not change anything in the Consumer class. Now, just create an object of the new server class (Server123) from the client, as in the following.

```
static void Main(string[] args)
{
   Consumer C = new Consumer(new Server123());
   C.Execute();
   Console.ReadLine();
}
```

And you will get the same output. Now again change the Class name from Consumer to Consumer123.

Have a look at the following code.

```
public class Server : IService
{
   public void Serve()
   {
      Console.WriteLine("Server executes");
   }
}

public class Consumer123
{
   IService obj;
   public Consumer123(IService temp)
   {
      this.obj = temp;
   }
   public void Execute()
   {
      obj.Serve();
   }
}
```

Here is the client code:

```
static void Main(string[] args)
{
   Consumer123 C = new Consumer123(new Server());
   C.Execute();
   Console.ReadLine();
}
```

Now, we have tuned up the Consumer class and within the Main() function (Client code) but it does not even touch the server class. And we will get the same output.

This is the advantage of decouple architecture. Hope you understood the concept.