# **DP**

BEST – Shailendra Chauhan- <https://www.dotnettricks.com/learn/designpatterns>

[**https://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns.html**](https://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns.html)

What are Design Patterns  
Design patterns are reusable solutions to the problems that we encounter in the day-to-day programming.  Design patterns acts as templates which can be applied to the real-world programming problems.

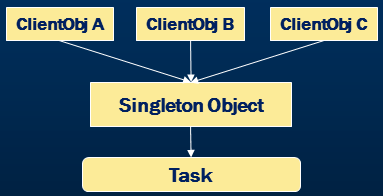
For more details, please refer to the following wikipedia article  
<https://en.wikipedia.org/wiki/Design_Patterns>  
  
Types of Design Patterns  
Gang of Four have categorised the design patterns in to 3 types based on different problems encountered in the real-world applications. They are Creational, Structural and Behavioural.

# Creational design patterns:

These patterns deal with object **creation and initialization**. Creational pattern gives the program more flexibility in deciding which objects need to be created for a given case.  
   
**Examples of Creational design patterns category:**Singleton, Factory and Abstract Factory etc.

* [**Abstract Factory**](https://sourcemaking.com/design_patterns/abstract_factory)  
  Creates an instance of several families of classes
* [**Builder**](https://sourcemaking.com/design_patterns/builder)  
  Separates object construction from its representation
* [**Factory Method**](https://sourcemaking.com/design_patterns/factory_method)  
  Creates an instance of several derived classes
* [**Object Pool**](https://sourcemaking.com/design_patterns/object_pool)  
  Avoid expensive acquisition and release of resources by recycling objects that are no longer in use
* [**Prototype**](https://sourcemaking.com/design_patterns/prototype)  
  A fully initialized instance to be copied or cloned

## Singleton Design Pattern

**Suggested Videos**  
[Part 1 - Introduction to Design Patters](https://www.youtube.com/watch?v=rI4kdGLaUiQ) - [Text](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns.html) - [Slides](http://csharp-video-tutorials.blogspot.com/2017/05/introduction-to-design-patterns_4.html)  
  
**In this tutorial we will discuss**   
1. What is Singleton Design Pattern  
2. Singleton as Creational Pattern  
3. Implementation Guidelines  
4. How do we implement a Singleton class  
  
**Singleton Pattern** belongs to **Creational type pattern**. As discussed in our previous video, Gang of four have defined five design patterns that belongs to creational design type category. Singleton is one among them and the rest are Factory, Abstract Factory, Builder and Prototype patterns. As the name implies, creational design type deals with object creation mechanisms. Basically, to simplify this, creational pattern explains us the creation of objects in a manner suitable to a given situation.  
  
  
Singleton design pattern is used when we need to ensure that only one object of a particular class is Instantiated. That single instance created is responsible to coordinate actions across the application.  
  
  
  
If you look at the illustrated diagram above you will see different objects trying to invoke an object instantiated as singleton. This single instance of the object is responsible to invoke underneath methods or events.  
  
**Advantages and Guidelines for Singleton implementation.**

Concurrent (Parallel) access to the resource is well managed by singleton design pattern.  
  
As part of the Implementation guidelines we need to ensure that only one instance of the class exists by declaring all constructors of the class to be private.  Also, to control the singleton access we need to provide a static property that returns a single instance of the object.  
  
**Singleton Class Implementation Example**

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First version of Singleton demo

/// </summary>

namespace SingletonDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            /\*

             \* Assuming Singleton is created from employee class

             \* we refer to the GetInstance property from the Singleton class

             \*/

            Singleton fromEmployee = Singleton.GetInstance;

            fromEmployee.PrintDetails("From Employee");

            /\*

             \* Assuming Singleton is created from student class

             \* we refer to the GetInstance property from the Singleton class

             \*/

            Singleton fromStudent = Singleton.GetInstance;

            fromStudent.PrintDetails("From Student");

            Console.ReadLine();

        }

    }

}

**Singleton.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First Singleton version

/// </summary>

namespace SingletonDemo

{

    /\*

     \*  Sealed ensures the class being inherited and

     \*  object instantiation is restricted in the derived class

     \*/

    public sealed class Singleton

    {

        private static int counter = 0;

        /\*

         \* Private property initialized with null

         \* Ensures that only one instance of the object is created

         \* Based on the null condition

         \*/

        private static Singleton instance = null;

        /\*

         \* public property is used to return only one instance of the class

         \* leveraging on the private property

         \*/

        public static Singleton GetInstance

        {

            get

            {

                if (instance == null)

                    instance = new Singleton();

                return instance;

            }

        }

        /\*

         \* Private constructor ensures that object is not

         \* Instantiated other than with in the class itself

         \*/

        private Singleton()

        {

            counter++;

            Console.WriteLine("Counter Value " + counter.ToString());

        }

        /\*

         \* Public method which can be invoked through the singleton instance

         \*/

        public void PrintDetails(string message)

        {

            Console.WriteLine(message);

        }

    }

}

### Why is singleton class sealed?

**In our previous video we discussed**

* We have proved that adding Private constructor will prevent us instantiating a new class
* We have further sealed down this class to avoid any inheritance  
    
  You might be wondering why we need to seal the class when a private constructor is present.   
    
  Let’s first remove the sealed keyword and check that. Let’s create another class called DerivedSingleton and Inherit the singleton class. Let's compile the code and look at that it has thrown an error saying Singleton is inaccessible due to its protection level. This error is because of private constructor.  
    
  Now you might be thinking that when a private constructor is restricting the inheritance then why we need to apply sealed keyword to the class.   
    
  Let’s just move this new class inside the Singleton class. By moving this class inside the Singleton class, it has now become nested or child class of the main singleton class.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First Singleton version

/// </summary>

namespace SingletonDemo

{

    /\*

     \*  Sealed restricts the inheritance

     \*/

    public class Singleton

    {

        private static int counter = 0;

        private static object obj = new object();

        /\*

        \* Private constructor ensures that object is not

        \* instantiated other than with in the class itself

        \*/

        private Singleton()

        {

            counter++;

            Console.WriteLine("Counter Value " + counter.ToString());

        }

        private static Singleton instance = null;

        /\*

         \* public property is used to return only one instance of the class

         \* leveraging on the private property

         \*/

        public static Singleton GetInstance

        {

            get

            {

                if (instance == null)

                    instance = new Singleton();

                return instance;

            }

        }

        /\*

         \* Public method which can be invoked through the singleton instance

         \*/

        public void PrintDetails(string message)

        {

            Console.WriteLine(message);

        }

        /\*

         \* By removing sealed keyword, we can inherit the singleton and instantiate multiple objects

         \* This violates singleton design principles.

         \*/

        public class DerivedSingleton : Singleton

        {

        }

    }

}  
  
**What is a nested class?**   
A class with in another class is called a nested class.   
  
Now that we have moved the derived class to nested class lets compile the program and check. Look at that we are able to compile this successfully.   
  
Now, let’s switch to main program and access the nested class.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First version of Singleton demo

/// </summary>

namespace SingletonDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            /\*

            \* Assuming Singleton is created from student class

            \* we refer to the GetInstance property from the Singleton class

            \*/

            Singleton fromStudent = Singleton.GetInstance;

            fromStudent.PrintDetails("From Student");

            /\*

            \* Assuming Singleton is created from employee class

            \* we refer to the GetInstance property from the Singleton class

            \*/

            Singleton fromEmployee = Singleton.GetInstance;

            fromEmployee.PrintDetails("From Employee");

            Console.WriteLine("-------------------------------------");

            /\*

             \* Instantiating singleton from a derived class. This violates singleton pattern principles.

             \*/

            Singleton.DerivedSingleton derivedObj = new Singleton.DerivedSingleton();

            derivedObj.PrintDetails("From Derived");

            Console.ReadLine();

        }

    }

}  
  
Let’s run the program. Look at that the counter value has incremented to 2 proving that we are able to create multiple instances of the singleton using the nested derived class.  
  
This violates the principle of singleton.  Let’s go back to the Singleton and make the class as sealed. Let’s compile the program  
  
Look at that we have got an error when we compile the program saying we cannot derive a sealed class. With this we have proved that private constructor helps in preventing any external instantiations of objects and sealed will prevent the class inheritances.  
  
In the next session we will discuss how to handle thread safety in singleton as the current version can create multiple instances in multi-threaded environments.

### Thread Safety in Singleton

**In this tutorial we will discuss**

* Lazy Initialization in Singleton
* How to use Multithreads in Singleton
* How to implement a Thread Safe singleton class

**Lazy Initialization in Singleton :** GetInstance Property is responsible for the Singleton Instance creation. Singleton object is not instantiated until and unless **GetInstance** is invoked. Hence, there is a delay in instance creation till the GetInstance is accessed. This **Delay** in Instance creation is called **Lazy Initialization**.  
  
**How to use Multithreads in Singleton:**The lazy initialization works perfectly well when we invoke the GetInstance in a Single threaded approach. However, there is a chance that we may end up creating multiple instances when multiple threads invoke the GetInstance at the same time.  
  
This Thread racing situation causes thread safety issues in Singleton Initialization and further the current code ends up in creating multiple instances of Singleton objects in memory.  
  
To achieve and replicate multiple threads accessing GetInstance, we have modified the main program by using Parallel.Invoke method of .NET Framework 4.0.  Please refer to Main program code below for more details.  
  
**How to implement a Thread Safe singleton class:** Locks are the best way to control thread race condition and they help us to overcome the present situation. Please refer to the Singleton.cs code for lock checks and double check locking.  
  
For more details on double check locking please refer to the below article  
<https://en.wikipedia.org/wiki/Double-checked_locking>

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First version of Singleton demo

/// </summary>

namespace SingletonDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            Parallel.Invoke(

                () => PrintStudentdetails(),

                () => PrintEmployeeDetails()

                );

            Console.ReadLine();

        }

        private static void PrintEmployeeDetails()

        {

            /\*

             \* Assuming Singleton is created from employee class

             \* we refer to the GetInstance property from the Singleton class

             \*/

            Singleton fromEmployee = Singleton.GetInstance;

            fromEmployee.PrintDetails("From Employee");

        }

        private static void PrintStudentdetails()

        {

            /\*

                         \* Assuming Singleton is created from student class

                         \* we refer to the GetInstance property from the Singleton class

                         \*/

            Singleton fromStudent = Singleton.GetInstance;

            fromStudent.PrintDetails("From Student");

        }

    }

}

**Singleton.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

/// <summary>

/// First Singleton version

/// </summary>

namespace SingletonDemo

{

    /\*

     \*  Sealed restricts the inheritance

     \*/

    public sealed class Singleton

    {

        private static int counter = 0;

        private static readonly object obj = new object();

        /\*

        \* Private constructor ensures that object is not

        \* instantiated other than with in the class itself

        \*/

        private Singleton()

        {

            counter++;

            Console.WriteLine("Counter Value " + counter.ToString());

        }

        private static Singleton instance = null;

        /\*

         \* public property is used to return only one instance of the class

         \* leveraging on the private property

         \*/

        public static Singleton GetInstance

        {

            get

            {

                if (instance == null)

                {

                    lock (obj)

                    {

                        if (instance == null)

                            instance = new Singleton();

                    }

                }

                return instance;

            }

        }

        /\*

         \* Public method which can be invoked through the singleton instance

         \*/

        public void PrintDetails(string message)

        {

            Console.WriteLine(message);

        }

    }

}

### Lazy vs Eager loading in Singleton

In this tutorial we will discuss the **difference between Lazy Initialization and Eager Initialization**  
  
**Lazy Initialization :**The lazy initialization of an object improves the performance and avoids unnecessary computation till the point the object is accessed. Further, it reduces the memory footprint during the startup of the program. Reducing the memory print will help faster loading of the application.  
  
  
  
**Non-Lazy or Eager Loading :** Eager loading is nothing but to initialize the required object before it’s being accessed.  Which means, we instantiate the object and keep it ready and use it when we need it. This type of initialization is used in lower memory footprints. Also, in eager loading, the common language runtime takes care of the variable initialization and its thread safety. Hence, we don’t need to write any explicit coding for thread safety.  
  
  
  
**Singleton with Lazy keyword (.NET 4.0) :** Lazy keyword provides support for lazy initialization. In order to make a property as lazy, we need to pass the type of object to the lazy keyword which is being lazily initialized.  
  
By default, Lazy<T> objects are thread-safe.  In multi-threaded scenarios, the first thread which tries to access the Value property of the lazy object will take care of thread safety when multiple threads are trying to access the Get Instance at the same time.  
  
Therefore, it does not matter which thread initializes the object or if there are any thread race conditions that are trying to access this property.

**Program.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SingletonDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            Parallel.Invoke(

                () => PrintStudentDetails(),

                () => PrintEmployeeDetails()

            );

            Console.ReadLine();

        }

        private static void PrintEmployeeDetails()

        {

            Singleton fromEmployee = Singleton.GetInstance;

            fromEmployee.PrintDetails("From Employee");

        }

        private static void PrintStudentDetails()

        {

            Singleton fromStudent = Singleton.GetInstance;

            fromStudent.PrintDetails("From Student");

        }

    }

}

**Singleton.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace SingletonDemo

{

    public sealed class Singleton

    {

        private static int counter = 0;

        private Singleton()

        {

            counter++;

            Console.WriteLine("Counter Value " + counter.ToString());

        }

        private static readonly Lazy<Singleton> instance =

new Lazy<Singleton>(()=>new Singleton());

        public static Singleton GetInstance

        {

            get

            {

                return instance.Value;

            }

        }

        public void PrintDetails(string message)

        {

            Console.WriteLine(message);

        }

    }

}

### Why singleton when there is static class

A singleton allows access to a single created instance - that instance (or rather, a reference to that instance) can be passed as a parameter to other methods, and treated as a normal object.

A static class allows only static methods.

**Static Class:-**

1. You cannot create the instance of static class.
2. Loaded automatically by the .NET Framework common language runtime (CLR) when the program or namespace containing the class is loaded.
3. We cannot pass the static class to method.
4. We cannot inherit Static class to another Static class in C#.
5. A class having all static methods.
6. Better performance (static methods are bonded on compile time)

**Singleton:-**

1. You can create one instance of the object and reuse it.
2. Singleton instance is created for the first time when the user requested.
3. You can create the object of singleton class and pass it to method.
4. Singleton class does not say any restriction of Inheritance.
5. We can dispose the objects of a singleton class but not of static class.
6. Methods can be overridden.
7. Can be lazy loaded when need (static classes are always loaded).
8. We can implement interface (static class cannot implement interface).

<https://www.c-sharpcorner.com/UploadFile/akkiraju/singleton-vs-static-classes/>

Singleton can implement interfaces, inherit from other classes and it aligns with the OOPS concepts

**Static class example - Temperature Converter**

**Real world usage of Singleton:** Listed are few real-world scenarios for singleton usage

1. Exception/Information logging
2. Connection pool management
3. File management
4. Device management such as printer spooling
5. Application Configuration management
6. Cache management
7. And Session based shopping cart

### 7. [Exception Logging using Singleton Design Pattern](https://www.youtube.com/watch?v=NjhUK68rzCs) | [Text](http://csharp-video-tutorials.blogspot.com/2017/06/exception-logging-using-singleton.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2017/06/exception-logging-using-singleton_26.html)

## 8. [Factory Design Pattern Introduction](https://www.youtube.com/watch?v=bGqw8crGZ7Y) | [Text](http://csharp-video-tutorials.blogspot.com/2017/07/factory-design-pattern-introduction.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2017/07/factory-design-pattern-introduction_23.html)

<https://www.c-sharpcorner.com/UploadFile/dacca2/design-pattern-for-beginner-part-2-factory-design-pattern/>

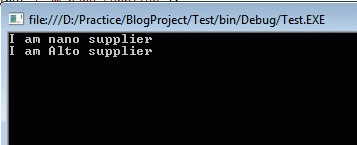
SOURAV KAYAL

We will initially see the logical stricture of the 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 happen. 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.

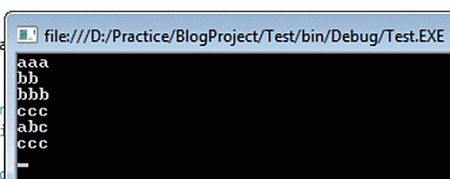
1. **using** System;
2. **using** System.Collections;
3. **using** System.Data.SqlClient;
4. **using** System.Threading;
6. **namespace** Test1
7. {
8. **public** **interface** ISupplier
9. {
10. **void** CarSupplier();
11. }
12. **class** namo : ISupplier
13. {
14. **public** **void** CarSupplier()
15. {
16. Console.WriteLine("I am nano supplier");
17. }
18. }
19. **class** alto : ISupplier
20. {
21. **public** **void** CarSupplier()
22. {
23. Console.WriteLine("I am Alto supplier");
24. }
25. }
26. **class** CarFactory
27. {
28. **public** **static** ISupplier GiveMyCar(**int** Key)
29. {
30. **if** (Key == 0)
31. **return** **new** namo();
32. **else** **if** (Key == 1)
33. **return** **new** alto();
34. **else**
35. **return** **null**;
36. }
37. }
38. **class** Program
39. {
40. **static** **void** Main(**string**[] args)
41. {
42. ISupplier obj = CarFactory.GiveMyCar(0);
43. obj.CarSupplier();
44. obj = CarFactory.GiveMyCar(1);
45. obj.CarSupplier();
46. Console.ReadLine();
47. }
48. }
49. }

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 affected 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.  
  
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.  
  
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.

1. **using** System;
2. **using** System.Collections;
3. **using** System.Data.SqlClient;
4. **using** System.Threading;
5. **using** System.Data.Common;
6. **using** System.Data;
8. **namespace** Test1
9. {
10. **class** Program
11. {
12. **static** **void** Main(**string**[] args)
13. {
14. DbProviderFactory provider = **null**;
15. DbConnection con = **null**;
16. DbCommand cmd = **null**;
17. DbDataReader rdr = **null**;
18. DataTable dt = **new** DataTable();
19. provider =DbProviderFactories.GetFactory("System.Data.SqlClient");
20. con = provider.CreateConnection();   //Create Connection according to Connection Class
21. con.ConnectionString = "Data Source=SOURAV-PC\\SQL\_INSTANCE;Initial Catalog=test;Integrated Security=True";
22. cmd = provider.CreateCommand();   //Create command according to Provider
23. **try**
24. {
25. cmd.CommandText = "select \* from name";
26. cmd.CommandType = CommandType.Text;
27. **if** (con.State == ConnectionState.Closed || con.State == ConnectionState.Broken)
28. {
29. con.Open();
30. cmd.Connection = con;
31. **using** (con)
32. {
33. rdr = cmd.ExecuteReader(CommandBehavior.CloseConnection);
34. **while** (rdr.Read())
35. {
36. Console.WriteLine(rdr["nametest"].ToString());
37. Console.WriteLine(rdr["surname"].ToString());
38. }
39. }
40. }
41. }
42. **catch** (Exception ex)
43. {
44. **throw**;
45. }
46. **finally**
47. {
48. //trn.Rollback();
49. con.Dispose();
50. cmd.Dispose();
51. }
52. Console.ReadLine();
53. }
54. }
55. }

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:

1. con = provider.CreateConnection();   //Create Connection according to database provide
2. 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’s name or database supplier name like:

1. 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:

1. 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.

## [Factory Method Design Pattern](https://www.youtube.com/watch?v=2NhRZn7agHc) | [Text](http://csharp-video-tutorials.blogspot.com/2017/08/factory-method-design-pattern.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2017/08/factory-method-design-pattern_10.html)

## 10. [Abstract Factory Design Pattern](https://www.youtube.com/watch?v=z47aJGe7jR4) | [Text](http://csharp-video-tutorials.blogspot.com/2017/08/abstract-factory-design-pattern.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2017/08/abstract-factory-design-pattern_21.html)

**Gang Of Four Definition :** "The Abstract factory pattern provides a way to encapsulate a group of individual factories that have a common theme without specifying their concrete classes"  
  
The Abstract Factory Pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes  
  
Abstract Factory pattern belongs to creational patterns and is one of the most used design patterns in real world applications

Abstract factory is a super factory that creates other factories  
  
**Implementation Guidelines**  
  
**We need to Choose Abstract Factory Pattern when**

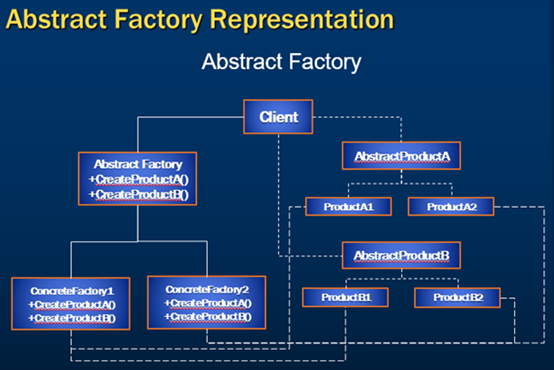
1. The application needs to create multiple families of objects or products
2. We need to use only one of the subsets of families of objects at a given point of time
3. We want to hide the implementations of the families of products by decoupling the implementation of each of these operations

**Business Requirement:** Handout computers to Contract and Permanent employees based on the designation and employee type with below specifications  
  
**Permanent Employee**

* Managerial Position is eligible for Apple MAC Book Laptop
* Non-Managerial Position is eligible for Apple IMac desktop

**Contract Employee**

* Managerial Position is eligible for Dell Laptop
* Non-Managerial Position is eligible for Dell desktop

**Abstract Factory Representation**  
  
  
**If you look at the illustrated diagram of Abstract Factory Representation** 

1. Client is a class which use AbstractFactory and AbstractProduct interfaces to create a family of related objects.
2. AbstractFactory is an interface which is used to create abstract product.
3. ConcreteFactory is a class which implements the AbstractFactory interface to create concrete products.
4. AbstractProduct is an interface which declares a type of product.
5. ConcreteProduct is a class which implements the AbstractProduct interface to create product.

**Steps to solve the above business requirement**  
  
**Step 1 :** Add ComputerDetails to the existing Employee table

CREATE TABLE [dbo].[Employee]

(

    [Id]               INT IDENTITY (1, 1) NOT NULL,

    [Name]             VARCHAR (50) NOT NULL,

    [JobDescription]   VARCHAR (50) NOT NULL,

    [Number]           VARCHAR (50) NOT NULL,

    [Department]       VARCHAR (50) NOT NULL,

    [HourlyPay]        DECIMAL (18) NOT NULL,

    [Bonus]            DECIMAL (18) NOT NULL,

    [EmployeeTypeID]   INT          NOT NULL,

    [HouseAllowance]   DECIMAL (18) NULL,

    [MedicalAllowance] DECIMAL (18) NULL,

    [ComputerDetails]  VARCHAR (250) NULL,

    PRIMARY KEY CLUSTERED ([Id] ASC),

    CONSTRAINT [FK\_Employee\_EmployeeType] FOREIGN KEY ([EmployeeTypeID])

     REFERENCES [dbo].[Employee\_Type] ([Id])

)

#### Step 2 :**Open EmployeePortal.edmx under the Models folder of the solution and update the model from the database (Right click on the model designer and choose update from database option)**

#### Step 3 :**Create AbstractFactory folder under existing Factory folder and add the below classes.**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Web.Factory.AbstractFactory

{

    public interface IBrand

    {

        string GetBrand();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Web.Factory.AbstractFactory

{

    public interface IProcessor

    {

        string GetProcessor();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Web.Factory.AbstractFactory

{

    public interface ISystemType

    {

        string GetSystemType();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using static Web.Factory.AbstractFactory.Enumerations;

namespace Web.Factory.AbstractFactory

{

    public class Laptop : ISystemType

    {

        public string GetSystemType()

        {

            return ComputerTypes.Laptop.ToString();

        }

    }

    public class Desktop : ISystemType

    {

        public string GetSystemType()

        {

            return ComputerTypes.Desktop.ToString();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace Web.Factory.AbstractFactory

{

    public class DELL : IBrand

    {

        public string GetBrand()

        {

            return Enumerations.Brands.DELL.ToString();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using static Web.Factory.AbstractFactory.Enumerations;

namespace Web.Factory.AbstractFactory

{

    public class MAC : IBrand

    {

        public string GetBrand()

        {

            return Brands.APPLE.ToString();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using static Web.Factory.AbstractFactory.Enumerations;

namespace Web.Factory.AbstractFactory

{

    public class I7 : IProcessor

    {

        public string GetProcessor()

        {

            return Processors.I7.ToString();

        }

    }

    public class I5 : IProcessor

    {

        public string GetProcessor()

        {

            return Processors.I5.ToString();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace Web.Factory.AbstractFactory

{

    public class Enumerations

    {

        public enum ComputerTypes

        {

            Laptop,

            Desktop

        }

        public enum Brands

        {

            APPLE,

            DELL

        }

        public enum Processors

        {

            I3,

            I5,

            I7

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace Web.Factory.AbstractFactory

{

    public interface IComputerFactory

    {

        IProcessor Processor();

        IBrand Brand();

        ISystemType SystemType();

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace Web.Factory.AbstractFactory

{

    public class DellFactory : IComputerFactory

    {

        public IBrand Brand()

        {

           return new DELL();

        }

        public IProcessor Processor()

        {

           return new I5();

        }

        public virtual ISystemType SystemType()

        {

            return new Desktop();

        }

    }

    public class DellLaptopFactory : DellFactory

    {

        public override ISystemType SystemType()

        {

            return new Laptop();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

using Web.Models;

namespace Web.Factory.AbstractFactory

{

    public class EmployeeSystemFactory

    {

        public IComputerFactory Create(Employee e)

        {

            IComputerFactory returnValue = null;

            if(e.EmployeeTypeID ==1)

            {

                if(e.JobDescription == "Manager")

                {

                    returnValue = new MACLaptopFactory();

                }

                else

                {

                    returnValue = new MACFactory();

                }

            }

            else if (e.EmployeeTypeID == 2)

            {

                if (e.JobDescription == "Manager")

                {

                    returnValue = new DellLaptopFactory();

                }

                else

                    returnValue = new DellFactory();

            }

            return returnValue;

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace Web.Factory.AbstractFactory

{

    public class MACFactory : IComputerFactory

    {

        public IBrand Brand()

        {

            return new MAC();

        }

        public IProcessor Processor()

        {

            return new I7();

        }

        public virtual ISystemType SystemType()

        {

            return new Desktop();

        }

    }

    public class MACLaptopFactory : MACFactory

    {

        public override ISystemType SystemType()

        {

            return new Laptop();

        }

    }

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Web;

namespace Web.Factory.AbstractFactory

{

    public class EmployeeSystemManager

    {

        IComputerFactory \_IComputerFactory;

        public EmployeeSystemManager(IComputerFactory iComputerFactory)

        {

            \_IComputerFactory = iComputerFactory;

        }

        public string GetSystemDetails()

        {

            IBrand brand = \_IComputerFactory.Brand();

            IProcessor processor = \_IComputerFactory.Processor();

            ISystemType systemType = \_IComputerFactory.SystemType();

            string returnValue = string.Format("{0} {1} {2}", brand.GetBrand(),

                systemType.GetSystemType(), processor.GetProcessor());

            return returnValue;

        }

    }

}

**Step 4 :** Now, integrate the AbstractFactory and Client in the EmployeesController’s Create method and replace the existing logic as stated below.

[HttpPost]

[ValidateAntiForgeryToken]

public ActionResult Create([Bind(Include = "Id,Name,JobDescription,Number,Department,HourlyPay,Bonus,EmployeeTypeID")] Employee employee)

{

    if (ModelState.IsValid)

    {

        BaseEmployeeFactory empFactory = new EmployeeManagerFactory().CreateFactory(employee);

        empFactory.ApplySalary();

        IComputerFactory factory = new EmployeeSystemFactory().Create(employee);

        EmployeeSystemManager manager = new EmployeeSystemManager(factory);

        employee.ComputerDetails = manager.GetSystemDetails();

        db.Employees.Add(employee);

        db.SaveChanges();

        return RedirectToAction("Index");

    }

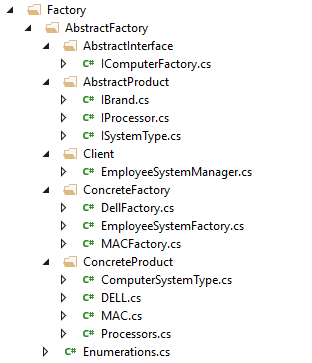
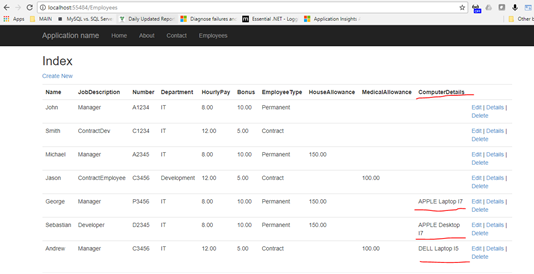
    ViewBag.EmployeeTypeID = new

        SelectList(db.Employee\_Type, "Id", "EmployeeType",

        employee.EmployeeTypeID);

    return View(employee);

}

**Step 5 :** Arrange the classes in the AbstractFactory folder with the below structure.  
  
  
**Step 6 :** Enhance the current Employee index view to add ComputerDetails columns.  
  
**Step 7 :** Run the application and notice that we have achieved the business requirement by using Abstract Factory pattern.  
  
  
**Step 8 :** Abstract Factory and Factory Method

* Abstract factory pattern adds a layer of abstraction to the factory method pattern
* Abstract factory pattern implementation can have multiple factory methods
* Similar products of a factory implementation are grouped in Abstract factory
* Abstract Factory uses object composition to decouple applications form specific implementations
* Factory Method uses inheritance to decouple applications form specific implementations

## 11. [Builder Design Pattern Introduction](https://www.youtube.com/watch?v=xhK7q-oUcHY) | [Text](http://csharp-video-tutorials.blogspot.com/2017/09/builder-design-pattern-introduction.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2017/09/builder-design-pattern-introduction_18.html)

## 12. [Builder Design Pattern Implementation](https://www.youtube.com/watch?v=ogrDaSXhnHQ) | [Text](http://csharp-video-tutorials.blogspot.com/2017/09/builder-design-pattern-implementation.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2017/09/builder-design-pattern-implementation_25.html)

## 13. [Fluent Builder Design Pattern](https://www.youtube.com/watch?v=PBIM67J4RJQ) | [Text](http://csharp-video-tutorials.blogspot.com/2017/10/fluent-builder-design-pattern.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2017/10/fluent-builder-design-pattern-slides.html)

# Structural design patterns:

This pattern deals with class and object composition. In simple words, this pattern focuses on **decoupling interface, implementation of classes and its objects**.   
  
**Examples of Structural design patterns category:**Adapter, Facade and Bridge etc.

* [**Adapter**](https://sourcemaking.com/design_patterns/adapter)  
  Match interfaces of different classes
* [**Bridge**](https://sourcemaking.com/design_patterns/bridge)  
  Separates an object’s interface from its implementation
* [**Composite**](https://sourcemaking.com/design_patterns/composite)  
  A tree structure of simple and composite objects
* [**Decorator**](https://sourcemaking.com/design_patterns/decorator)  
  Add responsibilities to objects dynamically
* [**Facade**](https://sourcemaking.com/design_patterns/facade)  
  A single class that represents an entire subsystem
* [**Flyweight**](https://sourcemaking.com/design_patterns/flyweight)  
  A fine-grained instance used for efficient sharing
* [**Private Class Data**](https://sourcemaking.com/design_patterns/private_class_data)  
  Restricts accessor/mutator access
* [**Proxy**](https://sourcemaking.com/design_patterns/proxy)  
  An object representing another object

## 14. [ProtoType Design Pattern Introduction](https://www.youtube.com/watch?v=f1BG1tkqZQU) | [Text](http://csharp-video-tutorials.blogspot.com/2017/11/prototype-design-pattern-introduction.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2017/11/prototype-design-pattern-introduction_1.html)

## 15. [Prototype Design Pattern Implementation](https://www.youtube.com/watch?v=rsDTqn1ALjw) | [Text](http://csharp-video-tutorials.blogspot.com/2017/11/prototype-design-pattern-implementation.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2017/11/prototype-design-pattern-implementation_13.html)

## 16. [Structural Design Patterns Introduction](https://www.youtube.com/watch?v=GqSRnbxFnQ0) | [Text](http://csharp-video-tutorials.blogspot.com/2017/11/structural-design-patterns-introduction.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2017/11/structural-design-patterns-introduction_21.html)

<https://www.dotnettricks.com/learn/designpatterns/adapter-design-pattern-dotnet>

### **Adapter Design Pattern - C#**

Adapter pattern falls under Structural Pattern of [Gang of Four (GOF) Design Patterns in .Net](http://www.dotnettricks.com/learn/designpatterns/gang-of-four-gof-design-patterns-in-net). The Adapter Design pattern allows a system to use classes of another system that is incompatible with it. It is especially used for toolkits and libraries. In this article, I would like to share what is adapter pattern and how is it work?

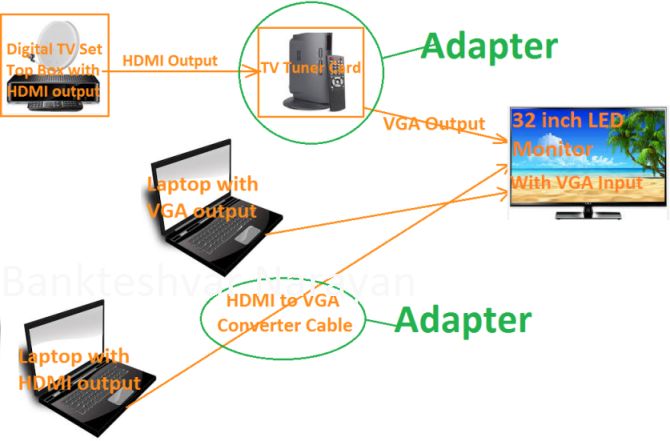
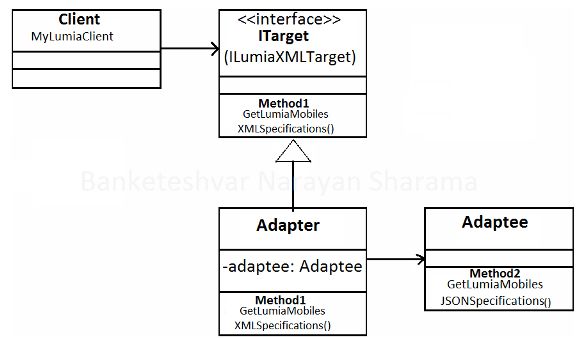
#### **What is Adapter Pattern**

Adapter pattern acts as a bridge between two incompatible interfaces. This pattern involves a single class called adapter which is responsible for communication between two independent or incompatible interfaces.

**For Example:** A card reader acts as an adapter between a memory card and a laptop. You plugins the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

To handle the incompatibility, we use different approaches and based on that we can classify Adapter Pattern in 2 parts.

1. Object Adapter Pattern
2. Class Adapter Pattern

**Object Adapter Design Pattern**  
  
In Object Adapter Pattern Incompatibility is handled by creating the object.  
  
**Class Adapter Design Pattern**  
In Class Adapter Design Pattern Incompatibility is handled by inheritance.  
  
**Some Real-world Example**  
  
The following image is my personal use of LED Monitor. I use my monitor in different ways which you can see in the following image.  
  
  
  
  
Let me explain the UML Diagram of Adapter pattern using a real example of programming. I have an interface which return list of Lumia Mobiles in JSON, but Client is expecting the list of Lumia Mobile features in XML format. Let ‘s solve it using Adapter patter.  
  
  
  
Explaining the same thing with the code:  
  
**LumiaJSONAdaptee.CS**

1. **public** **class** LumiaJSONAdaptee
2. {
3. **public** **string** GetLumiaMobilesJSONSpecifications()
4. {
5. List < LumiaMobile > LumiaMobiles = **new** List < LumiaMobile > ();
6. LumiaMobiles.Add(**new** LumiaMobile
7. {
8. ModelId = "lumia550",
9. Height = "136.1 mm",
10. Width = "67.8 mm",
11. Thickness = "9.9 mm",
12. Weight = "141.9 g"
13. });
14. LumiaMobiles.Add(**new** LumiaMobile
15. {
16. ModelId = "lumia950",
17. Height = "145 mm",
18. Width = "73.2 mm",
19. Thickness = "8.2 mm",
20. Weight = "150 g"
21. });
22. LumiaMobiles.Add(**new** LumiaMobile
23. {
24. ModelId = "Text",
25. Height = "",
26. Width = "",
27. Thickness = "8.2 mm",
28. Weight = "150 g"
29. });
30. dynamic collectionLumiaMobiles = **new**
31. {
32. lumiaMobiles = LumiaMobiles
33. };
34. **return** JsonConvert.SerializeObject(collectionLumiaMobiles);
35. }
36. }

**LumiaMobile.cs**

1. **public** **class** LumiaMobile
2. {
3. **public** **string** ModelId
4. {
5. **get**;
6. **set**;
7. }
8. **public** **string** Height
9. {
10. **get**;
11. **set**;
12. }
13. **public** **string** Width
14. {
15. **get**;
16. **set**;
17. }
18. **public** **string** Thickness
19. {
20. **get**;
21. **set**;
22. }
23. **public** **string** Weight
24. {
25. **get**;
26. **set**;
27. }
28. }

**ILumiaXMLTarget.cs**

1. **interface** ILumiaXMLTarget
2. {
3. XmlDocument GetLumiaMobilesXMLSpecifications();
4. }

**LumiaXMLAdapter.cs**

1. **class** LumiaXMLAdapter : ILumiaXMLTarget
2. {
3. **public** XmlDocument GetLumiaMobilesXMLSpecifications()
4. {
5. LumiaJSONAdaptee lumiaJsonAdaptee = **new** LumiaJSONAdaptee();
6. **string** jsonLumia = lumiaJsonAdaptee.GetLumiaMobilesJSONSpecifications();
7. XmlDocument doc = JsonConvert.DeserializeXmlNode(jsonLumia, "MicrosoftLumiaMoblies",**true**);
8. **return** doc;
9. }
10. }

**MyLumiaClient.cs**

1. **class** MyLumiaClient
2. {
3. **private** ILumiaXMLTarget \_lumiaXmlTarget;
4. **public** MyLumiaClient(ILumiaXMLTarget lumiaXmlTarget)
5. {
6. \_lumiaXmlTarget = lumiaXmlTarget;
7. }
8. **public** XmlDocument GetLumiaData()
9. {
10. **return** \_lumiaXmlTarget.GetLumiaMobilesXMLSpecifications();
11. }
12. }

#### **When to use it?**

1. Allow a system to use classes of another system that is incompatible with it.
2. Allow communication between a new and already existing system which are independent of each other
3. Ado.Net SqlAdapter, OracleAdapter, MySqlAdapter are the best example of Adapter Pattern.

###### **Note**

1. Internally, Adapter uses [Factory design pattern](https://www.dotnettricks.com/learn/designpatterns/factory-method-design-pattern-dotnet) for creating objects. But it can also use [Builder design pattern](https://www.dotnettricks.com/learn/designpatterns/builder-design-pattern-dotnet) and [prototype design pattern](https://www.dotnettricks.com/learn/designpatterns/prototype-design-pattern-dotnet)for creating a product. It completely depends upon your implementation for creating products.
2. Adapter can be used as an alternative to Facade to hide platform-specific classes.
3. When Adapter, Builder, and Prototype define a factory for creating the products, we should consider the following points :
   1. Adapter uses the factory for creating objects of several classes.
   2. Builder uses the factory for creating a complex product by using simple objects and a step by step approach.
   3. Prototype use the factory for building a product by copying an existing product.

### 18. [Bridge Design Pattern](https://www.youtube.com/watch?v=AvszFRYvvt0) | [Text](http://csharp-video-tutorials.blogspot.com/2018/01/bridge-design-pattern.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2018/01/bridge-design-pattern-slides.html)

<https://www.dotnettricks.com/learn/designpatterns/bridge-design-pattern-dotnet>

Bridge design pattern falls under Structural Pattern of [Gang of Four (GOF) Design Patterns in .Net](http://www.dotnettricks.com/learn/designpatterns/gang-of-four-gof-design-patterns-in-net). All we know, Inheritance is a way to specify different implementations of an abstraction. But in this way, implementations are tightly bound to the abstraction and cannot be modified independently. The Bridge pattern provides an alternative to inheritance when there is more than one version of abstraction. In this article, I would like to share what is bridge pattern and how is it work?

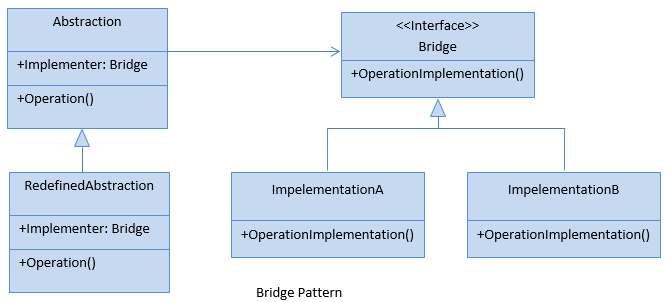
#### **What is Bridge Pattern**

The bridge pattern is used to separate abstraction from its implementation so that both can be modified independently.

This pattern involves an interface which acts as a bridge between the abstraction class and implementer classes and also makes the functionality of implementer class independent from the abstraction class. Both types of classes can be modified without affecting to each other.

#### **Bridge Pattern - UML Diagram & Implementation**

The UML class diagram for the implementation of the bridge design pattern is given below:



The classes, interfaces, and objects in the above UML class diagram are as follows:

##### **Abstraction**

This is an abstract class and containing members that define an abstract business object and its functionality. It contains a reference to an object of type Bridge. It can also act as the base class for other abstractions.

[](https://www.dotnettricks.com/training/masters-program/net-design-patterns-training?s=sb)

##### **Redefined Abstraction**

This is a class which inherits from the Abstraction class. It extends the interface defined by Abstraction class.

##### **Bridge**

This is an interface which acts as a bridge between the abstraction class and implementer classes and also makes the functionality of implementer class independent from the abstraction class.

##### **ImplementationA & ImplementationB**

These are classes which implement the Bridge interface and also provide the implementation details for the associated Abstraction class.

##### **C# - Implementation Code**

public abstract class Abstraction

{

public Bridge Implementer { get; set; }

public virtual void Operation()

{

Console.WriteLine("ImplementationBase:Operation()");

Implementer.OperationImplementation();

}

}

public class RefinedAbstraction : Abstraction

{

public override void Operation()

{

Console.WriteLine("RefinedAbstraction:Operation()");

Implementer.OperationImplementation();

}

}

public interface Bridge

{

void OperationImplementation();

}

public class ImplementationA : Bridge

{

public void OperationImplementation()

{

Console.WriteLine("ImplementationA:OperationImplementation()");

}

}

public class ImplementationB : Bridge

{

public void OperationImplementation()

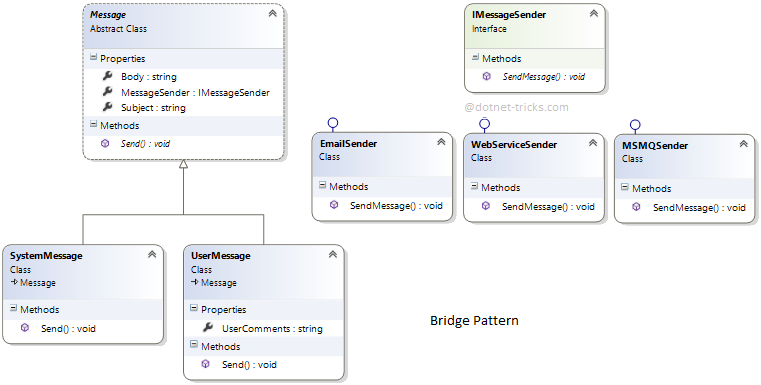
{

Console.WriteLine("ImplementationB:OperationImplementation()");

}

}

#### **Bridge Pattern - Example**



##### **Who is what?**

The classes, interfaces, and objects in the above class diagram can be identified as follows:

1. **Message** - Abstraction Class.
2. **SystemMessage & UserMessage**- Redefined Abstraction Classes.
3. **IMessageSender**- Bridge Interface.
4. **EmailSender, WebServiceSender & MSMQ Sender**- ConcreteImplementation class which implements the IMessageSender interface.

##### **C# - Sample Code**

/// <summary>

/// The 'Abstraction' class

/// </summary>

public abstract class Message

{

public IMessageSender MessageSender { get; set; }

public string Subject { get; set; }

public string Body { get; set; }

public abstract void Send();

}

/// <summary>

/// The 'RefinedAbstraction' class

/// </summary>

public class SystemMessage : Message

{

public override void Send()

{

MessageSender.SendMessage(Subject, Body);

}

}

/// <summary>

/// The 'RefinedAbstraction' class

/// </summary>

public class UserMessage : Message

{

public string UserComments { get; set; }

public override void Send()

{

string fullBody = string.Format("{0}\nUser Comments: {1}", Body, UserComments);

MessageSender.SendMessage(Subject, fullBody);

}

}

/// <summary>

/// The 'Bridge/Implementor' interface

/// </summary>

public interface IMessageSender

{

void SendMessage(string subject, string body);

}

/// <summary>

/// The 'ConcreteImplementor' class

/// </summary>

public class EmailSender : IMessageSender

{

public void SendMessage(string subject, string body)

{

Console.WriteLine("Email\n{0}\n{1}\n", subject, body);

}

}

/// <summary>

/// The 'ConcreteImplementor' class

/// </summary>

public class MSMQSender : IMessageSender

{

public void SendMessage(string subject, string body)

{

Console.WriteLine("MSMQ\n{0}\n{1}\n", subject, body);

}

}

/// <summary>

/// The 'ConcreteImplementor' class

/// </summary>

public class WebServiceSender : IMessageSender

{

public void SendMessage(string subject, string body)

{

Console.WriteLine("Web Service\n{0}\n{1}\n", subject, body);

}

}

/// <summary>

/// Bridge Design Pattern Demo

/// </summary>

class Program

{

static void Main(string[] args)

{

IMessageSender email = new EmailSender();

IMessageSender queue = new MSMQSender();

IMessageSender web = new WebServiceSender();

Message message = new SystemMessage();

message.Subject = "Test Message";

message.Body = "Hi, This is a Test Message";

message.MessageSender = email;

message.Send();

message.MessageSender = queue;

message.Send();

message.MessageSender = web;

message.Send();

UserMessage usermsg = new UserMessage();

usermsg.Subject = "Test Message";

usermsg.Body = "Hi, This is a Test Message";

usermsg.UserComments = "I hope you are well";

usermsg.MessageSender = email;

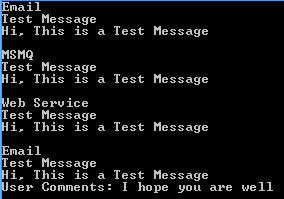
usermsg.Send();

Console.ReadKey();

}

}

##### **Bridge Pattern Demo - Output**



#### **When to use it?**

1. Abstractions and implementations should be modified independently.
2. Changes in the implementation of an abstraction should have no impact on clients.
3. The Bridge pattern is used when a new version of a software or system is brought out, but the older version of the software still running for its existing client. There is no need to change the client code, but the client needs to choose which version he wants to use.

###### **Note**

Bridge pattern has nearly the same structure as the Adapter Pattern. But it is used when designing new systems instead of the Adapter pattern which is used with already existing systems.

19. [Composite Design Pattern](https://www.youtube.com/watch?v=EXuaKKAlfqE) | [Text](http://csharp-video-tutorials.blogspot.com/2018/01/composite-design-pattern.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2018/01/composite-design-pattern-slides.html)

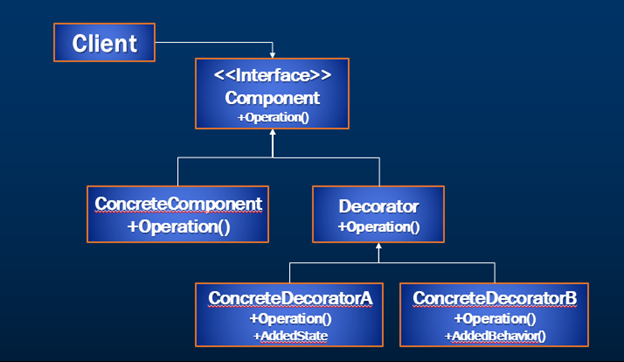
### 20. [Decorator Design Pattern](https://www.youtube.com/watch?v=YObNH6IkIos) | [Text](http://csharp-video-tutorials.blogspot.com/2018/02/decorator-design-pattern.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2018/02/decorator-design-pattern-slides.html)

**In this video we will learn**

* What is Decorator Design Pattern
* Implementation guidelines of decorator design pattern
* Simple Decorator Design Pattern example

**Decorator Design Pattern:** As per the GOF definition, decorator Pattern states that we need to “Attach additional responsibilities to an object dynamically. Decorators provide A flexible alternative to sub classing for extending functionality.”  
  
This pattern Falls under the category of Structural Design Pattern and is also known as Wrapper.  
  
  
**Implementation Guidelines:** We need to use Decorator Design Pattern when 

* We need to add responsibilities to individual objects dynamically and transparently,
* The extension by sub classing is impractical.
* Class definition may be hidden or otherwise unavailable for sub classing.

**Representation Diagram**  
  


* **Client :**Clients use the Component interface to interact with objects
* **Component :** Defines the interface for objects that can have responsibilities added to them dynamically.
* **ConcreteComponent :** Defines an object to which additional responsibilities can be attached.
* **Decorator :** Maintains a reference to a Component object and defines an interface that conforms to Component's interface.
* **ConcreteDecorator**: Adds responsibilities to the component.

Let’s start building our code with Decorator design pattern implementation.  
  
**Step 1 :**Add Interface ICar

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DecoratorDemo.Component

{

    public interface ICar

    {

        string Make { get; }

        double GetPrice();

    }

}

**Step 2 :** Create Concrete component Hyndai to implement ICar

using DecoratorDemo.Component;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DecoratorDemo.ConcreteComponent

{

    public sealed class Hyndai : ICar

    {

        public string Make

        {

            get { return "HatchBack"; }

        }

        public double GetPrice()

        {

            return 800000;

        }

    }

}

**Step 3 :** Create Concrete component Suzuki to implement ICar

using DecoratorDemo.Component;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DecoratorDemo.ConcreteComponent

{

    public sealed class Suzuki : ICar

    {

        public string Make

        {

            get { return "Sedan"; }

        }

        public double GetPrice()

        {

            return 1000000;

        }

    }

}

**Step 4 :** Create Decorator Class CarDecorator to decorate the Icar using CarDecorator

using DecoratorDemo.Component;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DecoratorDemo.Decorator

{

    public abstract class CarDecorator : ICar

    {

        private ICar car;

        public CarDecorator(ICar Car)

        {

            car = Car;

        }

        public string Make { get { return car.Make; } }

        public double GetPrice()

        {

            return car.GetPrice();

        }

        public abstract double GetDiscountedPrice();

    }

}

**Step 5 :**Create Concrete Decorator OfferPrice to decorate the Icar using CarDecorator

using DecoratorDemo.Component;

using DecoratorDemo.Decorator;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DecoratorDemo.ConcreteDecorator

{

    public class OfferPrice : CarDecorator

    {

        public OfferPrice(ICar car) : base(car)

        {

        }

        public override double GetDiscountedPrice()

        {

            return .8 \* base.GetPrice();

        }

    }

}

**Step 6 :** Use the car decorator in the main program to achieve the output

using DecoratorDemo.Component;

using DecoratorDemo.ConcreteComponent;

using DecoratorDemo.ConcreteDecorator;

using DecoratorDemo.Decorator;

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace DecoratorDemo

{

    class Program

    {

        static void Main(string[] args)

        {

            ICar car = new Suzuki();

            CarDecorator decorator = new OfferPrice(car);

            Console.WriteLine(string.Format("Make :{0}  Price:{1} " +

                "DiscountPrice : {2}"

                , decorator.Make,  decorator.GetPrice().ToString(),

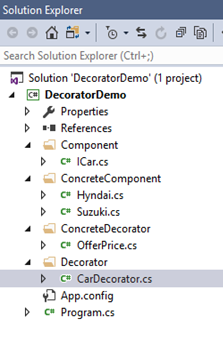
                decorator.GetDiscountedPrice().ToString()));

            Console.ReadLine();

        }

    }

}

**Step 7 :**Run the application to see the expected output  
  
**Make : Sedan Price:1000000 DiscountPrice: 800000**  
  
**Step 8 :**Above mentioned classes are created under the below folder structure. Again it’s not mandatory to follow this folder structure as we are just doing this for representation purpose.  
  


### 21. [Facade Design Pattern](https://www.youtube.com/watch?v=Ah3ig3ysLZA) | [Text](http://csharp-video-tutorials.blogspot.com/2018/02/facade-design-pattern.html) | [Slides](http://csharp-video-tutorials.blogspot.com/2018/02/facade-design-pattern-slides.html)

# Behavioural design patterns:

These patterns deal with communication between Classes and objects.  
  
**Examples of Behavioural design patterns:** Chain of Responsibility, Command and Interpreter etc.

* [**Chain of responsibility**](https://sourcemaking.com/design_patterns/chain_of_responsibility)  
  A way of passing a request between a chain of objects
* [**Command**](https://sourcemaking.com/design_patterns/command)  
  Encapsulate a command request as an object
* [**Interpreter**](https://sourcemaking.com/design_patterns/interpreter)  
  A way to include language elements in a program
* [**Iterator**](https://sourcemaking.com/design_patterns/iterator)  
  Sequentially access the elements of a collection
* [**Mediator**](https://sourcemaking.com/design_patterns/mediator)  
  Defines simplified communication between classes
* [**Memento**](https://sourcemaking.com/design_patterns/memento)  
  Capture and restore an object's internal state
* [**Null Object**](https://sourcemaking.com/design_patterns/null_object)  
  Designed to act as a default value of an object
* [**Observer**](https://sourcemaking.com/design_patterns/observer)  
  A way of notifying change to a number of classes
* [**State**](https://sourcemaking.com/design_patterns/state)  
  Alter an object's behavior when its state changes
* [**Strategy**](https://sourcemaking.com/design_patterns/strategy)  
  Encapsulates an algorithm inside a class
* [**Template method**](https://sourcemaking.com/design_patterns/template_method)  
  Defer the exact steps of an algorithm to a subclass
* [**Visitor**](https://sourcemaking.com/design_patterns/visitor)  
  Defines a new operation to a class without change

## **OBSERVER DESIGN PATTERN**

[Design Pattern For Beginner: Part-10: Observer Design Pattern (c-sharpcorner.com)](https://www.c-sharpcorner.com/UploadFile/dacca2/design-pattern-for-beginner-part-10-observer-design-patter/)

**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.

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **namespace** ObserverPattern
6. {
7. **interface** INotifyObserver
8. {
9. **void** Notify();
10. }
11. **class** MailNotify : INotifyObserver
12. {
13. **public** **void** Notify()
14. {
15. Console.WriteLine("Notify through Mail");
16. }
17. }
18. **class** EventNotify : INotifyObserver
19. {
20. **public** **void** Notify()
21. {
22. Console.WriteLine("Notify through Event");
23. }
24. }
25. **class** SMSNotify : INotifyObserver
26. {
27. **public** **void** Notify()
28. {
29. Console.WriteLine("Notify through SMS");
30. }
31. }
32. }

**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.

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **using** System.Collections;
6. **namespace** ObserverPattern
7. {
8. **class** clsNotifier
9. {
10. **public** ArrayList ALNotify = **new** ArrayList();
11. /// <summary>
12. /// Add object of notification System
13. /// </summary>
14. /// <param name="obj">Object is notification class</param>
15. **public** **void** AddService(INotifyObserver obj)
16. {
17. ALNotify.Add(obj);
18. }
19. /// <summary>
20. /// Remove object of notification System
21. /// </summary>
22. /// <param name="obj">Object of notification Calss</param>
23. **public** **void** RemoveService(INotifyObserver obj)
24. {
25. ALNotify.Remove(obj);
26. }
27. **public** **void** ExecuteNotifier()
28. {
29. **foreach** (INotifyObserver O **in** ALNotify)
30. {
31. //Call all notification System
32. O.Notify();
33. }
34. }
35. }
36. }

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**

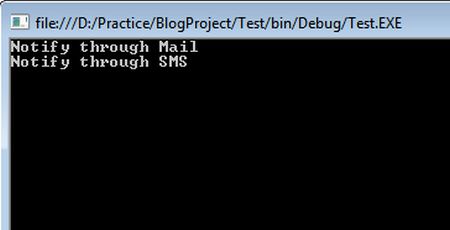
This is the last part of the example. We will create client code to set and make decisions about which notification will be fired.

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **namespace** ObserverPattern
6. {
7. **class** Program
8. {
9. **static** **void** Main(**string**[] args)
10. {
11. //Generate exception to notify all client
12. **try**
13. {
14. **throw** **new** ApplicationException("This is Exception");
15. }
16. **catch** (Exception ex)
17. {
18. INotifyObserver obj1 = **new** MailNotify();
19. INotifyObserver obj2 = **new** SMSNotify();
20. clsNotifier O = **new** clsNotifier();
21. O.AddService(obj1);
22. O.AddService(obj2);
23. O.ExecuteNotifier();
24. }
25. Console.ReadLine();
26. }
27. }
28. }

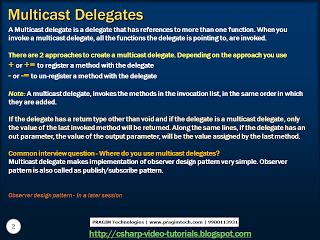
Here, we are generating ApplicationException() and within the Catch block we are creating an object to the MailNotification and SMSnotification class.

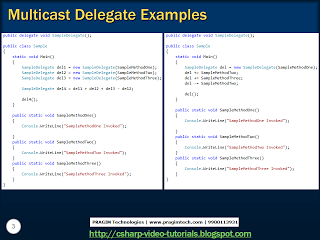
The executeNotifier() will call all Notify() functions from each notification class.

**Output**

****

## **MULTICAST DELEGATE - OBSERVER DESIGN PATTERN**

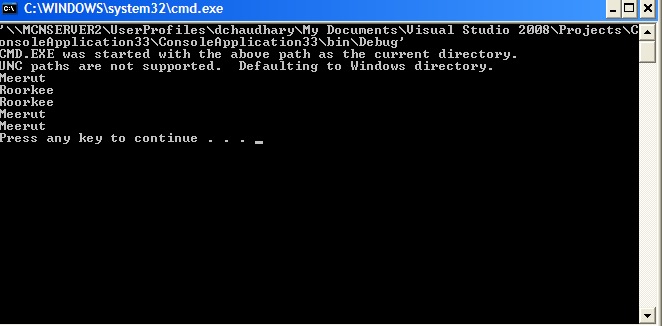
**Where do you use multicast delegates? This is a very common interview question.**  
**Answer:**Multicast delegate makes implementation of observer design pattern very simple. Observer pattern is also called as publish/subscribe pattern.[](http://3.bp.blogspot.com/-F3tqzNgp9CY/UjMxYyYPWJI/AAAAAAAAFE4/qI4UKqnbAwY/s1600/Slide2.PNG)

[](http://4.bp.blogspot.com/-2D2o7ytaSug/UjMxYuDN2sI/AAAAAAAAFE0/yIc9QOKfpeg/s1600/Slide3.PNG)

**Implement Multicast Delegates eExample**

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Text;
5. **delegate** **void** MDelegate();
6. **class** DM
7. {
8. **static** **public** **void** Display()
9. {
10. Console.WriteLine("Meerut");
11. }
12. **static** **public** **void** print()
13. {
14. Console.WriteLine("Roorkee");
15. }
16. }
17. **class** MTest
18. {
19. **public** **static** **void** Main()
20. {
21. MDelegate m1 = **new** MDelegate(DM.Display);
22. MDelegate m2 = **new** MDelegate(DM.print);
23. MDelegate m3 = m1 + m2;
24. MDelegate m4 = m2 + m1;
25. MDelegate m5 = m3 - m2;
26. m3();
27. m4();
28. m5();
29. }
30. }

**Output of the porgram**



# **Factory Design Pattern Vs. Factory Method Design Pattern**

<https://www.c-sharpcorner.com/UploadFile/akkiraju/factory-design-pattern-vs-factory-method-design-pattern/>

# **Which DP have you used?**

# **Unit of Work**

# **MVVM**

<http://www.codeproject.com/Articles/98598/How-I-explained-Design-Patterns-to-my-wife-Part-1>

<http://www.codeproject.com/Articles/127647/Design-Patterns-An-Overview-For-Beginners>

<http://www.go4expert.com/articles/design-pattern-simple-examples-t5127/>

# **SOLID OOPS**

[SOLID Principles In C# (c-sharpcorner.com)](https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/)

The following are the design flaws that cause damage in software, mostly.

1. Putting more stress on classes by assigning more responsibilities to them. (A lot of functionality not related to a class.)
2. Forcing the classes to depend on each other. If classes are dependent on each other (in other words tightly coupled), then a change in one will affect the other.
3. Spreading duplicate code in the system/application.

**Solution**

1. Choosing the correct architecture (in other words MVC, 3-tier, Layered, MVP, MVVP and so on).
2. Following Design Principles.
3. Choosing correct Design Patterns to build the software based on its specifications.

## **Intro to SOLID principles**

SOLID principles are the design principles that enable us to manage most of the software design problems. Robert C. Martin compiled these principles in the 1990s.

These principles provide us with ways to move from tightly coupled code and little encapsulation to the desired results of loosely coupled and encapsulated real needs of a business properly.

SOLID is an acronym of the following.

* S: Single Responsibility Principle (SRP)
* O: Open closed Principle (OCP)
* L: Liskov substitution Principle (LSP)
* I: Interface Segregation Principle (ISP)
* D: Dependency Inversion Principle (DIP)

## **S: Single Responsibility Principle (SRP)**

SRP says "Every software module should have only one reason to change".



This means that every class, or similar structure, in your code should have only one job to do. Everything in that class should be related to a single purpose. Our class should not be like a Swiss knife wherein if one of them needs to be changed then the entire tool needs to be altered. It does not mean that your classes should only contain one method or property. There may be many members as long as they relate to single responsibility.

The Single Responsibility Principle gives us a good way of identifying classes at the design phase of an application and it makes you think of all the ways a class can change. A good separation of responsibilities is done only when we have the full picture of how the application should work. Let us check this with an example.

1. **public** **class** UserService
2. {
3. **public** **void** Register(**string** email, **string** password)
4. {
5. **if** (!ValidateEmail(email))
6. **throw** **new** ValidationException("Email is not an email");
7. var user = **new** User(email, password);

   SendEmail(**new** MailMessage("mysite@nowhere.com", email) { Subject="HEllo foo" });

1. }
2. **public** **virtual** **bool** ValidateEmail(**string** email)
3. {
4. **return** email.Contains("@");
5. }
6. **public** **bool** SendEmail(MailMessage message)
7. {
8. \_smtpClient.Send(message);
9. }
10. }

It looks fine, but it is not following SRP.

The SendEmail and ValidateEmail methods have nothing to do within the UserService class. Let's refract it.

1. **public** **class** UserService
2. {
3. EmailService \_emailService;
4. DbContext \_dbContext;
5. **public** UserService(EmailService aEmailService, DbContext aDbContext)
6. {
7. \_emailService = aEmailService;
8. \_dbContext = aDbContext;
9. }
10. **public** **void** Register(**string** email, **string** password)
11. {
12. **if** (!\_emailService.ValidateEmail(email))
13. **throw** **new** ValidationException("Email is not an email");
14. var user = **new** User(email, password);
15. \_dbContext.Save(user);
16. emailService.SendEmail(**new** MailMessage("myname@mydomain.com", email) {Subject="Hi. How are you!"});
18. }
19. }

1. **public** **class** EmailService
2. {
3. SmtpClient \_smtpClient;
4. **public** EmailService(SmtpClient aSmtpClient)
5. {
6. \_smtpClient = aSmtpClient;
7. }
8. **public** **bool** **virtual** ValidateEmail(**string** email)
9. {
10. **return** email.Contains("@");
11. }
12. **public** **bool** SendEmail(MailMessage message)
13. {
14. \_smtpClient.Send(message);
15. }
16. }

If we put more than one functionality in one class then it introduces coupling between two functionalities. So, if we change one functionality there is a chance we broke coupled functionality, which requires another round of testing to avoid any bug in the production environment.

It reduces bug fixes and testing time once an application goes into the maintenance phase. It follows the DRY principle.

## **O: Open/Closed Principle**

The Open/closed Principle says "A software module/class is open for extension and closed for modification".



Here "Open for extension" means, we need to design our module/class in such a way that the new functionality can be added only when new requirements are generated. "Closed for modification" means we have already developed a class and it has gone through unit testing. We should then not alter it until we find bugs. As it says, a class should be open for extensions, we can use inheritance to do this. Okay, let's dive into an example.

Suppose we have a Rectangle class with the properties Height and Width.

1. **public** **class** Rectangle{
2. **public** **double** Height {**get**;**set**;}
3. **public** **double** Wight {**get**;**set**; }
4. }

Our app needs the ability to calculate the total area of a collection of Rectangles. Since we already learned the Single Responsibility Principle (SRP), we don't need to put the total area calculation code inside the rectangle. So here I created another class for area calculation.

1. **public** **class** AreaCalculator {
2. **public** **double** TotalArea(Rectangle[] arrRectangles)
3. {
4. **double** area;
5. **foreach**(var objRectangle **in** arrRectangles)
6. {
7. area += objRectangle.Height \* objRectangle.Width;
8. }
9. **return** area;
10. }
11. }

Hey, we did it. We made our app without violating SRP. No issues for now. But can we extend our app so that it could calculate the area of not only Rectangles but also the area of Circles as well? Now we have an issue with the area calculation issue because the way to do circle area calculation is different. Hmm. Not a big deal. We can change the TotalArea method a bit so that it can accept an array of objects as an argument. We check the object type in the loop and do area calculation based on the object type.

1. **public** **class** Rectangle{
2. **public** **double** Height {**get**;**set**;}
3. **public** **double** Wight {**get**;**set**; }
4. }
5. **public** **class** Circle{
6. **public** **double** Radius {**get**;**set**;}
7. }
8. **public** **class** AreaCalculator
9. {
10. **public** **double** TotalArea(**object**[] arrObjects)
11. {
12. **double** area = 0;
13. Rectangle objRectangle;
14. Circle objCircle;
15. **foreach**(var obj **in** arrObjects)
16. {
17. **if**(obj **is** Rectangle)
18. {
19. area += obj.Height \* obj.Width;
20. }
21. **else**
22. {
23. objCircle = (Circle)obj;
24. area += objCircle.Radius \* objCircle.Radius \* Math.PI;
25. }
26. }
27. **return** area;
28. }
29. }

Wow. We are done with the change. Here we successfully introduced Circle into our app. We can add a Triangle and calculate it's the area by adding one more "if" block in the TotalArea method of AreaCalculator. But every time we introduce a new shape we need to alter the TotalArea method. So the AreaCalculator class is not closed for modification. How can we make our design to avoid this situation? Generally, we can do this by referring to abstractions for dependencies, such as interfaces or abstract classes, rather than using concrete classes. Such interfaces can be fixed once developed so the classes that depend upon them can rely upon unchanging abstractions. Functionality can be added by creating new classes that implement the interfaces. So let's refract our code using an interface.

1. **public** **abstract** **class** Shape
2. {
3. **public** **abstract** **double** Area();
4. }

Inheriting from Shape, the Rectangle and Circle classes now look like this:

1. **public** **class** Rectangle: Shape
2. {
3. **public** **double** Height {**get**;**set**;}
4. **public** **double** Width {**get**;**set**;}
5. **public** **override** **double** Area()
6. {
7. **return** Height \* Width;
8. }
9. }
10. **public** **class** Circle: Shape
11. {
12. **public** **double** Radius {**get**;**set**;}
13. **public** **override** **double** Area()
14. {
15. **return** Radius \* Radus \* Math.PI;
16. }
17. }

Every shape contains its area with its own way of calculation functionality and our AreaCalculator class will become simpler than before.

1. **public** **class** AreaCalculator
2. {
3. **public** **double** TotalArea(Shape[] arrShapes)
4. {
5. **double** area=0;
6. **foreach**(var objShape **in** arrShapes)
7. {
8. area += objShape.Area();
9. }
10. **return** area;
11. }
12. }

Now our code is following SRP and OCP both. Whenever you introduce a new shape by deriving from the "Shape" abstract class, **you need not change the "AreaCalculator" class**. Awesome. Isn't it?

## **L: Liskov Substitution Principle**

**The “L” in SOLID is for***Liskov Substitution Principle***,** which states that subclases should be substitutable for the classes from which they were derived. For example, if MySubclass is a subclass of MyClass, you should be able to replace MyClass with MySubclass without bunging up the program.

<https://dotnettutorials.net/lesson/liskov-substitution-principle/#:~:text=The%20Liskov%20Substitution%20Principle%20in%20C%23%20states%20that%20even%20the,follows%20the%20Liskov%20Substitution%20Principle>.

The **Liskov Substitution Principle**is a Substitutability principle in object-oriented programming Language. This principle states that, if **S** is a subtype of **T**, then objects of type **T** should be replaced with the objects of type **S**.

##### **Example: Without using the Liskov Substitution Principle in C#:**

First, we create the Apple class with the method GetColor. Then we create the orange class which inherits the Apple class as well as overrides the GetColor method of the Apple class. The point is that an Orange cannot be replaced by an Apple, which results in printing the color of the apple as Orange as shown in the below example.

**namespace** *SOLID\_PRINCIPLES.LSP*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Apple apple = new Orange**()**;

Console.WriteLine**(**apple.GetColor**())**;

**}**

**}**

**public** **class** Apple

**{**

**public** **virtual** string GetColor**()**

**{**

**return** "Red";

**}**

**}**

**public** **class** Orange : Apple

**{**

**public** **override** string GetColor**()**

**{**

**return** "Orange";

**}**

**}**

**}**

##### **Example Using the Liskov Substitution Principle in C#**

Let’s modify the previous example to follow the Liskov Substitution Principle. Here, first, we need a generic base class such as Fruit which is going to be the base class for both Apple and Orange. Now you can replace the Fruit class object with its subtypes either Apple and Orage and it will behave correctly. Now, you can see in the below code, we created the super Fruit class as an abstract class with the GetColor abstract method and then the Apple and Orange class inherited from the Fruit class and implement the GetColor method.

**namespace** *SOLID\_PRINCIPLES.LSP*

**{**

**class** Program

**{**

**static** **void** Main**(**string**[]** args**)**

**{**

Fruit fruit = new Orange**()**;

Console.WriteLine**(**fruit.GetColor**())**;

fruit = new Apple**()**;

Console.WriteLine**(**fruit.GetColor**())**;

**}**

**}**

**public** **abstract** **class** Fruit

**{**

**public** **abstract** string GetColor**()**;

**}**

**public** **class** Apple : Fruit

**{**

**public** **override** string GetColor**()**

**{**

**return** "Red";

**}**

**}**

**public** **class** Orange : Fruit

**{**

**public** **override** string GetColor**()**

**{**

**return** "Orange";

**}**

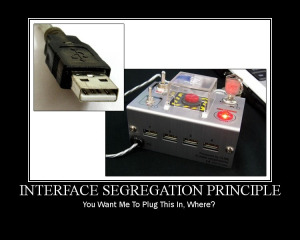
**}**

**}**

Now, run the application and it should give the output as expected. Here we are following the LSP as we are now able to change the object with its subtype.

## **I: Interface Segregation Principle (ISP)**

The Interface Segregation Principle states "that clients should not be forced to implement interfaces they don't use. Instead of one fat interface, many small interfaces are preferred based on groups of methods, each one serving one submodule.".



Based on specifications, we need to create an interface and a TeamLead class to implement it.

1. **public** Interface ILead
2. {
3. **void** CreateSubTask();
4. **void** AssginTask();
5. **void** WorkOnTask();
6. }
7. **public** **class** TeamLead : ILead
8. {
9. **public** **void** AssignTask()
10. {
11. //Code to assign a task.
12. }
13. **public** **void** CreateSubTask()
14. {
15. //Code to create a sub task
16. }
17. **public** **void** WorkOnTask()
18. {
19. //Code to implement perform assigned task.
20. }
21. }

OK. The design looks fine for now. Later another role like Manager, who assigns tasks to TeamLead and will not work on the tasks, is introduced into the system. Can we directly implement an ILead interface in the Manager class, like the following?

1. **public** **class** Manager: ILead
2. {
3. **public** **void** AssignTask()
4. {
5. //Code to assign a task.
6. }
7. **public** **void** CreateSubTask()
8. {
9. //Code to create a sub task.
10. }
11. **public** **void** WorkOnTask()
12. {
13. **throw** **new** Exception("Manager can't work on Task");
14. }
15. }

Since the Manager can't work on a task and at the same time no one can assign tasks to the Manager, this WorkOnTask() should not be in the Manager class. But we are implementing this class from the ILead interface, we need to provide a concrete Method. Here we are forcing the Manager class to implement a WorkOnTask() method without a purpose. This is wrong. The design violates ISP. Let's correct the design.

Since we have three roles, 1. Manager, that can only divide and assign the tasks, 2. TeamLead that can divide and assign the tasks and can work on them as well, 3. The programmer that can only work on tasks, we need to divide the responsibilities by segregating the ILead interface. An interface that provides a contract for WorkOnTask().

1. **public** **interface** IProgrammer
2. {
3. **void** WorkOnTask();
4. }

An interface that provides contracts to manage the tasks:

1. **public** **interface** ILead
2. {
3. **void** AssignTask();
4. **void** CreateSubTask();
5. }

Then the implementation becomes:

1. **public** **class** Programmer: IProgrammer
2. {
3. **public** **void** WorkOnTask()
4. {
5. //code to implement to work on the Task.
6. }
7. }
8. **public** **class** Manager: ILead
9. {
10. **public** **void** AssignTask()
11. {
12. //Code to assign a Task
13. }
14. **public** **void** CreateSubTask()
15. {
16. //Code to create a sub taks from a task.
17. }
18. }

TeamLead can manage tasks and can work on them if needed. Then the TeamLead class should implement both of the IProgrammer and ILead interfaces.

1. **public** **class** TeamLead: IProgrammer, ILead
2. {
3. **public** **void** AssignTask()
4. {
5. //Code to assign a Task
6. }
7. **public** **void** CreateSubTask()
8. {
9. //Code to create a sub task from a task.
10. }
11. **public** **void** WorkOnTask()
12. {
13. //code to implement to work on the Task.
14. }
15. }

Wow. Here we separated responsibilities/purposes and distributed them on multiple interfaces and provided a good level of abstraction too.

## **D: Dependency Inversion Principle**

<https://www.c-sharpcorner.com/article/solid-principles-in-c-sharp-dependency-inversion-principle/>

The Dependency Inversion Principle (DIP) states that a high-level class must not depend upon a lower level class. They must both depend upon abstractions. And, secondly, an abstraction must not depend upon details, but the details must depend upon abstractions. This will ensure the class and ultimately the whole application is very robust and easy to maintain and expand, if required. Let us look at this with an example.

Loosely Coupled.

Let us create a new .NET Core 3.1 console application in Visual Studio 2019 Community Edition as below:

Inside this project, I have created a new class file “DependencyInversionPrinciple”. In this file, I created the following classes.

1. // Not following the Dependency Inversion Principle
3. blic **class** SalaryCalculator
4. {
5. **public** **float** CalculateSalary(**int** hoursWorked, **float** hourlyRate) => hoursWorked \* hourlyRate;
6. }
8. **public** **class** EmployeeDetails
9. {
10. **public** **int** HoursWorked { **get**; **set**; }
11. **public** **int** HourlyRate { **get**; **set**; }
12. **public** **float** GetSalary()
13. {
14. var salaryCalculator = **new** SalaryCalculator();
15. **return** salaryCalculator.CalculateSalary(HoursWorked, HourlyRate);
16. }
17. }

These classes do not follow the “Dependency Inversion Principle” as the higher-level class EmployeeDetails is directly depending upon the lower level SalaryCalculator class.

We can fix this issue as below:

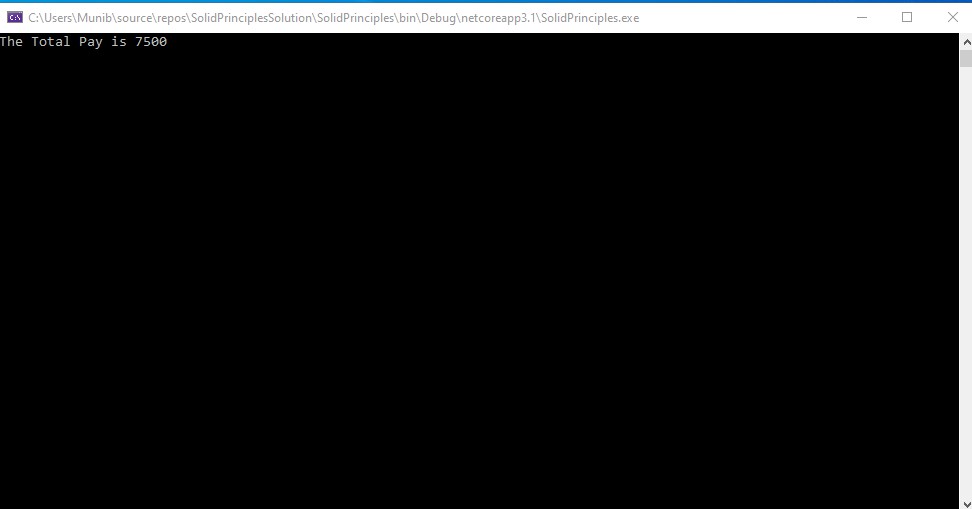
1. // Following the Dependency Inversion Principle
3. **public** **interface** ISalaryCalculator
4. {
5. **float** CalculateSalary(**int** hoursWorked, **float** hourlyRate);
6. }
8. **public** **class** SalaryCalculatorModified : ISalaryCalculator
9. {
10. **public** **float** CalculateSalary(**int** hoursWorked, **float** hourlyRate) => hoursWorked \* hourlyRate;
11. }
13. **public** **class** EmployeeDetailsModified
14. {
15. **private** **readonly** ISalaryCalculator \_salaryCalculator;
16. **public** **int** HoursWorked { **get**; **set**; }
17. **public** **int** HourlyRate { **get**; **set**; }
18. **public** EmployeeDetailsModified(ISalaryCalculator salaryCalculator)
19. {
20. \_salaryCalculator = salaryCalculator;
21. }
22. **public** **float** GetSalary()
23. {
24. **return** \_salaryCalculator.CalculateSalary(HoursWorked, HourlyRate);
25. }
26. }

In the above code, we see that we have created an interface ISalaryCalculator and then we have a class called SalaryCalculatorModified that implements this interface. Finally, in the higher-level class EmployeeDetailsModified, we only depend upon the ISalaryCalculator interface and not the concrete class. Hence, when we create the EmployeeDetailsModified class, we specify the abstraction implementation to use.

In addition to this, the details of the CalculateSalary function are hidden from the EmployeeDetailsModified class and any changes to this function will not affect the **interface** being used. Hence, we can see that in this new design the higher-level class does not depend upon the lower level class but on an abstraction and the abstraction does not depend upon the details.

To use the EmployeeDetailsModified class, we use the below code,

1. var employeeDetailsModified = **new** EmployeeDetailsModified(**new** SalaryCalculatorModified());
2. employeeDetailsModified.HourlyRate = 50;
3. employeeDetailsModified.HoursWorked = 150;
4. Console.WriteLine($"The Total Pay is {employeeDetailsModified.GetSalary()}");



# **Repository Pattern In MVC Application Using Entity Framework**

<https://www.c-sharpcorner.com/UploadFile/rahul4_saxena/repository-pattern-in-mvc-application-using-entity-framework/>

**What is Repository Pattern**  
  
**Repository Pattern** is an abstraction of the Data Access Layer. It hides the details of how exactly the data is saved or retrieved from the underlying data source. The details of how the data is stored and retrieved is in the respective repository. For example, you may have a repository that stores and retrieves data from an in-memory collection. You may have another repository that stores and retrieves data from a database like SQL Server.

public class Employee

{

    public int Id { get; set; }

    public string Name { get; set; }

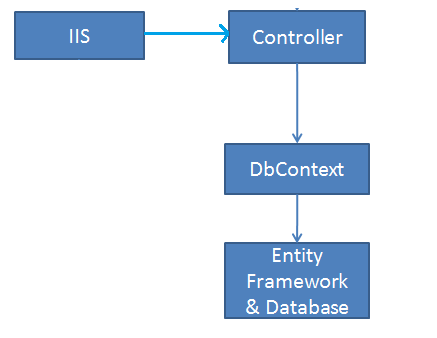
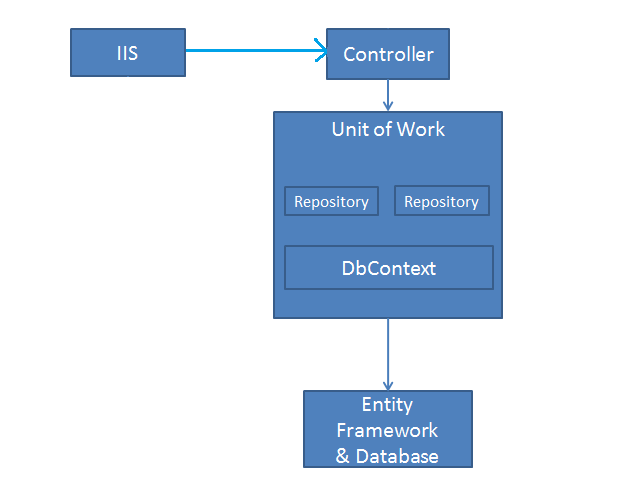
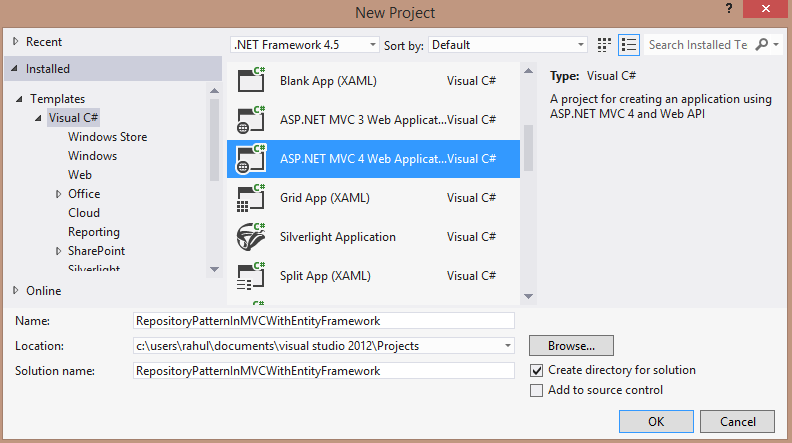
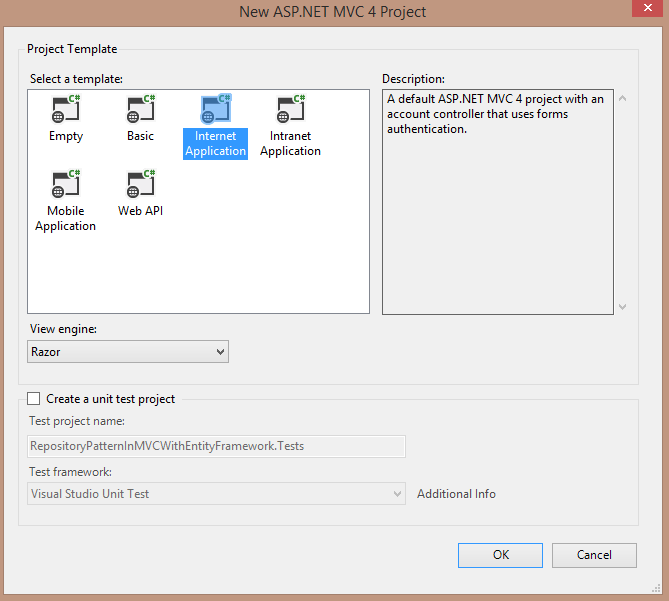
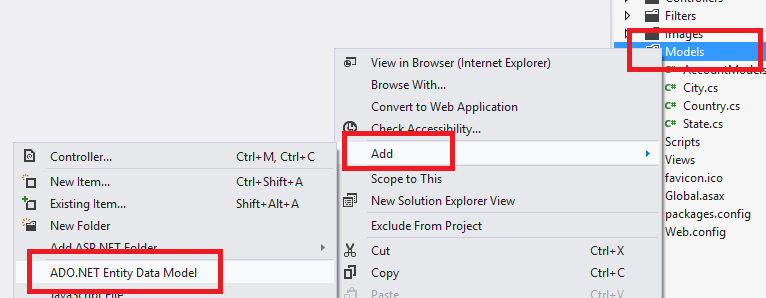
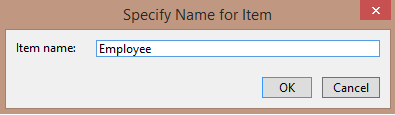
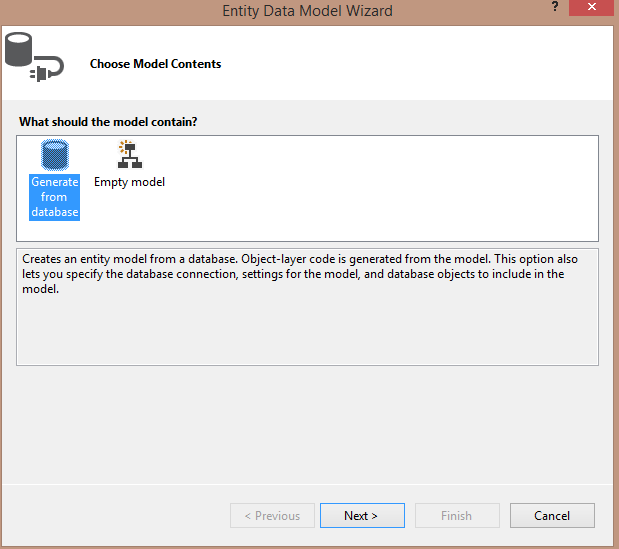
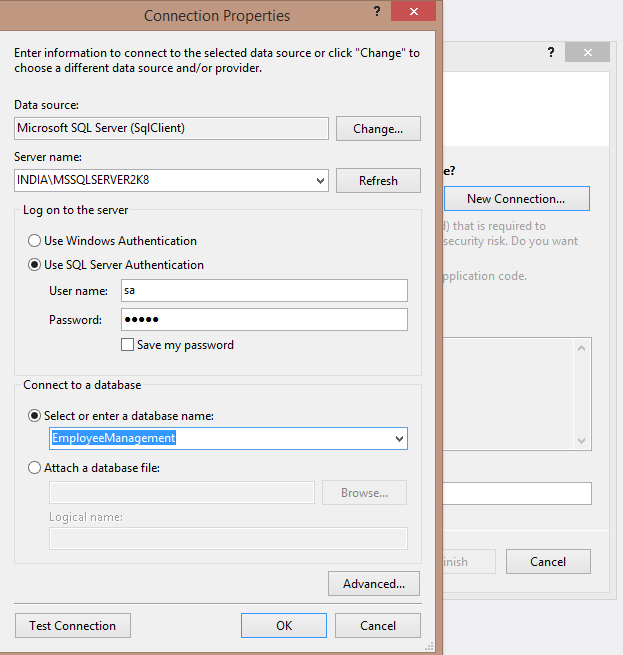
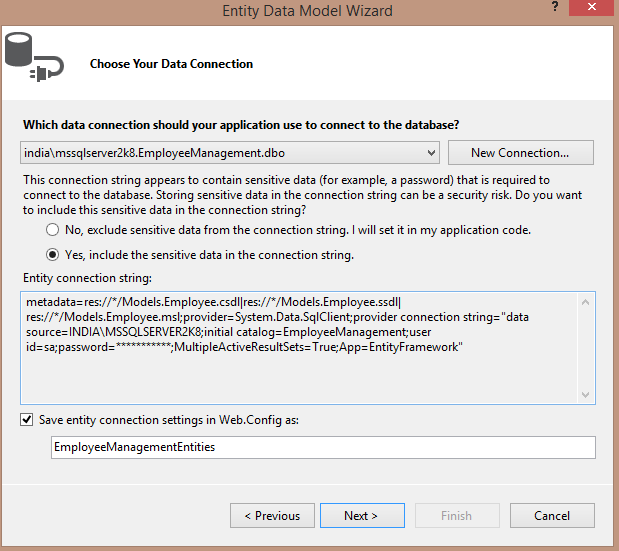
    public string Email { get; set; }

    public Dept? Department { get; set; }

}

**Repository Pattern Interface**  
  
**The interface in the repository pattern specifies**

* What operations (i.e methods) are supported by the repository
* The data required for each of the operations i.e the parameters that need to be passed to the method and the data the method returns
* The repository interface contains what it can do, but not, how it does, what it can do
* The implementation details are in the respective repository class that implements the repository Interface

This article explains the Repository Pattern in MVC applications using the Entity Framework.  
  
First we need to understand what the Repository Pattern is. See the following Image.  
  
**MVC Application without Repository Pattern**  
  
  
  
**MVC Application with Repository pattern**  
  
  
  
The Repository Pattern separates the data access logic and maps it to the entities in the business logic.  
  
Now we will see the Repository Pattern with a sample application.  
  
Open Visual Studio 2012 and go to "File" -> "New" -> "Project...".  
  
  
  
  
  
Now Add a ADO.NET Entity Data Model to your application. Right-click on the project in the Solution Explorer then select Add -> ADO.NET Entity Data Model.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Now add a new folder to your application named DAL.  
  
Right-click on the DAL Folder then select Add Interface -> IEmployeeRepository.cs.  
  
  
  
Now my IEmployeeRepository.cs is as in the following:

1. **using** System;
2. **using** System.Collections.Generic;
3. **using** System.Linq;
4. **using** System.Web;
5. **using** RepositoryPatternInMVCWithEntityFramework.Models;
6. **using** System.Data;
8. **namespace** RepositoryPatternInMVCWithEntityFramework.DAL
9. {
10. **public** **class** EmployeeRepository : IEmployeeRepository, IDisposable
11. {
12. **private** EmployeeManagementEntities context;
14. **public** EmployeeRepository(EmployeeManagementEntities context)
15. {
16. **this**.context = context;
17. }
19. **public** IEnumerable<Employee> GetAllEmployee()
20. {
21. **return** context.Employee.ToList();
22. }
24. **public** Employee GetEmployeeByID(**int** id)
25. {
26. **return** context.Employee.Find(id);
27. }
29. **public** **void** InsertEmployee(Employee emp)
30. {
31. context.Employee.Add(emp);
32. }
34. **public** **void** DeleteEmployee(**int** emp\_ID)
35. {
36. Employee emp = context.Employee.Find(emp\_ID);
37. context.Employee.Remove(emp);
38. }
40. **public** **void** UpdateEmployee(Employee emp)
41. {
42. context.Entry(emp).State = EntityState.Modified;
43. }
45. **public** **void** Save()
46. {
47. context.SaveChanges();
48. }
50. **private** **bool** disposed = **false**;
52. **protected** **virtual** **void** Dispose(**bool** disposing)
53. {
54. **if** (!**this**.disposed)
55. {
56. **if** (disposing)
57. {
58. context.Dispose();
59. }
60. }
61. **this**.disposed = **true**;
62. }
64. **public** **void** Dispose()
65. {
66. Dispose(**true**);
67. GC.SuppressFinalize(**this**);
68. }
69. }
70. }

Now right-click on the Controller folder in your project in the Solution Explorer then select Add New Empty Controller -> EmployeeController and provide the following code.  
  
**EmployeeController**

1. **using** RepositoryPatternInMVCWithEntityFramework.DAL;
2. **using** System;
3. **using** System.Collections.Generic;
4. **using** System.Linq;
5. **using** System.Web;
6. **using** System.Web.Mvc;
7. **using** RepositoryPatternInMVCWithEntityFramework.Models;
8. **using** PagedList;
9. **using** System.Data;
11. **namespace** RepositoryPatternInMVCWithEntityFramework.Controllers
12. {
13. **public** **class** EmployeeController : Controller
14. {
15. **private** IEmployeeRepository employeeRepository;
17. **public** EmployeeController()
18. {
19. **this**.employeeRepository = **new** EmployeeRepository(**new** EmployeeManagementEntities());
20. }
22. **public** EmployeeController(IEmployeeRepository employeeRepository)
23. {
24. **this**.employeeRepository = employeeRepository;
25. }
27. **public** ViewResult Index(**string** sortOrder, **string** currentFilter, **string** searchString, **int**? page)
28. {
29. ViewBag.CurrentSort = sortOrder;
30. ViewBag.NameSortParm = String.IsNullOrEmpty(sortOrder) ? "Emp\_ID" : "";
31. **if** (searchString != **null**)
32. {
33. page = 1;
34. }
35. **else**
36. {
37. searchString = currentFilter;
38. }
39. ViewBag.CurrentFilter = searchString;
41. var employees = from s **in** employeeRepository.GetAllEmployee()
42. select s;
43. **if** (!String.IsNullOrEmpty(searchString))
44. {
45. employees = employees.Where(s => s.Name.ToUpper().Contains(searchString.ToUpper())
46. || s.Name.ToUpper().Contains(searchString.ToUpper()));
47. }
48. **switch** (sortOrder)
49. {
50. **case** "Emp ID":
51. employees = employees.OrderByDescending(s => s.Emp\_ID);
52. **break**;
53. **case** "Name":
54. employees = employees.OrderBy(s => s.Name);
55. **break**;
56. **case** "State":
57. employees = employees.OrderByDescending(s => s.State);
58. **break**;
59. **case** "Country":
60. employees = employees.OrderByDescending(s => s.Country);
61. **break**;
62. **default**:
63. employees = employees.OrderBy(s => s.Emp\_ID);
64. **break**;
65. }
67. **int** pageSize = 5;
68. **int** pageNumber = (page ?? 1);
69. **return** View(employees.ToPagedList(pageNumber, pageSize));
70. }
72. // GET: /Employee/Details/5
74. **public** ViewResult Details(**int** id)
75. {
76. Employee emp = employeeRepository.GetEmployeeByID(id);
77. **return** View(emp);
78. }
80. // GET: /Employee/Create
82. **public** ActionResult Create()
83. {
84. **return** View();
85. }
87. // POST: /Employee/Create
89. [HttpPost]
90. [ValidateAntiForgeryToken]
91. **public** ActionResult Create(
92. [Bind(Include = "Name, Email, Designation,City, State, Country")]
93. Employee emp)
94. {
95. **try**
96. {
97. **if** (ModelState.IsValid)
98. {
99. employeeRepository.InsertEmployee(emp);
100. employeeRepository.Save();
101. **return** RedirectToAction("Index");
102. }
103. }
104. **catch** (Exception ex)
105. {
106. ModelState.AddModelError(**string**.Empty, "Some Error Occured.");
107. }
108. **return** View(emp);
109. }
111. // GET: /Employee/Edit/5
113. **public** ActionResult Edit(**int** id)
114. {
115. Employee emp = employeeRepository.GetEmployeeByID(id);
116. **return** View(emp);
117. }
119. // POST: /Employee/Edit/5
121. [HttpPost]
122. [ValidateAntiForgeryToken]
123. **public** ActionResult Edit(Employee emp)
124. {
125. **try**
126. {
127. **if** (ModelState.IsValid)
128. {
129. employeeRepository.UpdateEmployee(emp);
130. employeeRepository.Save();
131. **return** RedirectToAction("Index");
132. }
133. }
134. **catch** (Exception ex)
135. {
136. ModelState.AddModelError(**string**.Empty, "Some error Occured.");
137. }
138. **return** View(emp);
139. }
141. // GET: /employee/Delete/5
143. **public** ActionResult Delete(**bool**? saveChangesError = **false**, **int** id = 0)
144. {
145. **if** (saveChangesError.GetValueOrDefault())
146. {
147. ViewBag.ErrorMessage = "Some Error Occured.";
148. }
149. Employee emp = employeeRepository.GetEmployeeByID(id);
150. **return** View(emp);
151. }
153. // POST: /Employee/Delete/5
155. [HttpPost]
156. [ValidateAntiForgeryToken]
157. **public** ActionResult Delete(**int** id)
158. {
159. **try**
160. {
161. Employee emp = employeeRepository.GetEmployeeByID(id);
162. employeeRepository.DeleteEmployee(id);
163. employeeRepository.Save();
164. }
165. **catch** (Exception ex)
166. {
167. **return** RedirectToAction("Delete", **new** { id = id, saveChangesError = **true** });
168. }
169. **return** RedirectToAction("Index");
170. }
172. **protected** **override** **void** Dispose(**bool** disposing)
173. {
174. employeeRepository.Dispose();
175. **base**.Dispose(disposing);
176. }
178. }
179. }

Now add a View. Right-click on the Index Action Method and select Add view.

**Index.cshtml**

1. @using PagedList.Mvc;
3. @model PagedList.IPagedList<RepositoryPatternInMVCWithEntityFramework.Employee>
4. <link href="~/Content/PagedList.css" rel="stylesheet" type="text/css" />
6. @{
7. ViewBag.Title = "Employee Management System";
8. }
10. <h2>Employee Management System</h2>

13. @using (Html.BeginForm("Index", "Employee", FormMethod.Get))
14. {
15. <p style="background-color:red; color:white; font-size:16pt; padding:10px;">
16. Search Employee By Name: @Html.TextBox("SearchString", ViewBag.CurrentFilter as string)
17. <input type="submit" value="Search" />
18. @Html.ActionLink("Add New Employee", "Create")
19. </p>
20. }
21. <table style="background-color:white;">
22. <tr>
23. <th></th>
24. <th style="width: 100px;">
25. @Html.ActionLink("Emp ID", "Index", new { sortOrder = ViewBag.NameSortParm, currentFilter = ViewBag.CurrentFilter })
26. </th>
27. <th>
28. @Html.ActionLink("Name", "Index", new { sortOrder = ViewBag.NameSortParm, currentFilter = ViewBag.CurrentFilter })
29. </th>
30. <th>Email
31. </th>
32. <th>Designation
33. </th>
34. <th>City
35. </th>
36. <th>
37. @Html.ActionLink("State", "Index", new { sortOrder = ViewBag.NameSortParm, currentFilter = ViewBag.CurrentFilter })
38. </th>
39. <th>
40. @Html.ActionLink("Country", "Index", new { sortOrder = ViewBag.NameSortParm, currentFilter = ViewBag.CurrentFilter })
41. </th>
42. <th style="width: 150px;"></th>
43. </tr>
45. @foreach (var item in Model)
46. {
47. <tr>
48. <td></td>
49. <td>
50. @Html.DisplayFor(modelItem => item.Emp\_ID)
51. </td>
52. <td style="width:130px;">
53. @Html.DisplayFor(modelItem => item.Name)
54. </td>
55. <td>
56. @Html.DisplayFor(modelItem => item.Email)
57. </td>
58. <td style="width:140px;">
59. @Html.DisplayFor(modelItem => item.Designation)
60. </td>
61. <td style="width:120px;">
62. @Html.DisplayFor(modelItem => item.City)
63. </td>
64. <td style="width:120px;">
65. @Html.DisplayFor(modelItem => item.State)
66. </td>
67. <td>
68. @Html.DisplayFor(modelItem => item.Country)
69. </td>
70. <td style="width:270px;">
71. @Html.ActionLink("Edit", "Edit", new { id = item.Emp\_ID }) |
72. @Html.ActionLink("Details", "Details", new { id = item.Emp\_ID }) |
73. @Html.ActionLink("Delete", "Delete", new { id = item.Emp\_ID })
74. </td>
75. </tr>
76. }
78. </table>
79. <br />
80. <div style="background-color:orange; padding-left:15px; padding-top:10px;">
81. Showing Records @(Model.PageCount < Model.PageNumber ? 0 : Model.PageNumber) of @Model.PageCount
82. @Html.PagedListPager(Model, page => Url.Action("Index", new { page, sortOrder = ViewBag.CurrentSort,currentFilter = ViewBag.CurrentFilter }))
83. </div>

Here I am using PagedList. You can add the reference of PagedList by right-clicking on the project in the Solution Explorer and selecting Manage NuGet Packages.

Now add a Detail View by right-clicking on the Details Action method. (Make it Strongly Typed View with class Employee and Scaffold template Details.)  
  
**Details.cshtml**

1. @model RepositoryPatternInMVCWithEntityFramework.Employee
3. <h2>Employee Details</h2>
5. <table>
6. <tr>
7. <td>@Html.DisplayNameFor(model => model.Emp\_ID)</td>
8. <td>@Html.DisplayFor(model => model.Emp\_ID)</td>
9. </tr>
10. <tr>
11. <td>@Html.DisplayNameFor(model => model.Name)</td>
12. <td>@Html.DisplayFor(model => model.Name)</td>
13. </tr>
14. <tr>
15. <td>@Html.DisplayNameFor(model => model.Email)</td>
16. <td>@Html.DisplayFor(model => model.Email)</td>
17. </tr>
18. <tr>
19. <td>@Html.DisplayNameFor(model => model.Designation)</td>
20. <td>@Html.DisplayFor(model => model.Designation)</td>
21. </tr>
22. <tr>
23. <td>@Html.DisplayNameFor(model => model.City)</td>
24. <td>@Html.DisplayFor(model => model.City)</td>
25. </tr>
26. <tr>
27. <td>@Html.DisplayNameFor(model => model.State)</td>
28. <td>@Html.DisplayFor(model => model.State)</td>
29. </tr>
30. <tr>
31. <td>@Html.DisplayNameFor(model => model.Country)</td>
32. <td>@Html.DisplayFor(model => model.Country)</td>
33. </tr>
34. <tr style="background-color: orange; padding: 25px;">
35. <td></td>
36. <td>@Html.ActionLink("Edit", "Edit", new { id = Model.Emp\_ID }) |
37. @Html.ActionLink("Back to List", "Index")</td>
39. </tr>
40. </table>

Now create a View by right-clicking on the Create Action Method.  
  
**Create.cshtml**

1. @model RepositoryPatternInMVCWithEntityFramework.Employee
3. <script src="~/Scripts/jquery-1.7.1.min.js"></script>
4. <script src="~/Scripts/jquery.validate.min.js"></script>
5. <script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script>
7. @using (Html.BeginForm())
8. {
9. @Html.AntiForgeryToken()
10. @Html.ValidationSummary(true)
12. <fieldset>
13. <legend>Employee</legend>
15. <div class="editor-label">
16. @Html.LabelFor(model => model.Name)
17. </div>
18. <div class="editor-field">
19. @Html.EditorFor(model => model.Name)
20. @Html.ValidationMessageFor(model => model.Name)
21. </div>
23. <div class="editor-label">
24. @Html.LabelFor(model => model.Email)
25. </div>
26. <div class="editor-field">
27. @Html.EditorFor(model => model.Email)
28. @Html.ValidationMessageFor(model => model.Email)
29. </div>
31. <div class="editor-label">
32. @Html.LabelFor(model => model.Designation)
33. </div>
34. <div class="editor-field">
35. @Html.EditorFor(model => model.Designation)
36. @Html.ValidationMessageFor(model => model.Designation)
37. </div>
39. <div class="editor-label">
40. @Html.LabelFor(model => model.City)
41. </div>
42. <div class="editor-field">
43. @Html.EditorFor(model => model.City)
44. @Html.ValidationMessageFor(model => model.City)
45. </div>
47. <div class="editor-label">
48. @Html.LabelFor(model => model.State)
49. </div>
50. <div class="editor-field">
51. @Html.EditorFor(model => model.State)
52. @Html.ValidationMessageFor(model => model.State)
53. </div>
55. <div class="editor-label">
56. @Html.LabelFor(model => model.Country)
57. </div>
58. <div class="editor-field">
59. @Html.EditorFor(model => model.Country)
60. @Html.ValidationMessageFor(model => model.Country)
61. </div>
63. <p>
64. <input type="submit" value="Create" />
65. </p>
66. </fieldset>
67. }
69. <div>
70. @Html.ActionLink("Back to List", "Index")
71. </div>

Now add an Edit View by right-clicking on the Edit Action Method.

**Edit.cshtml**

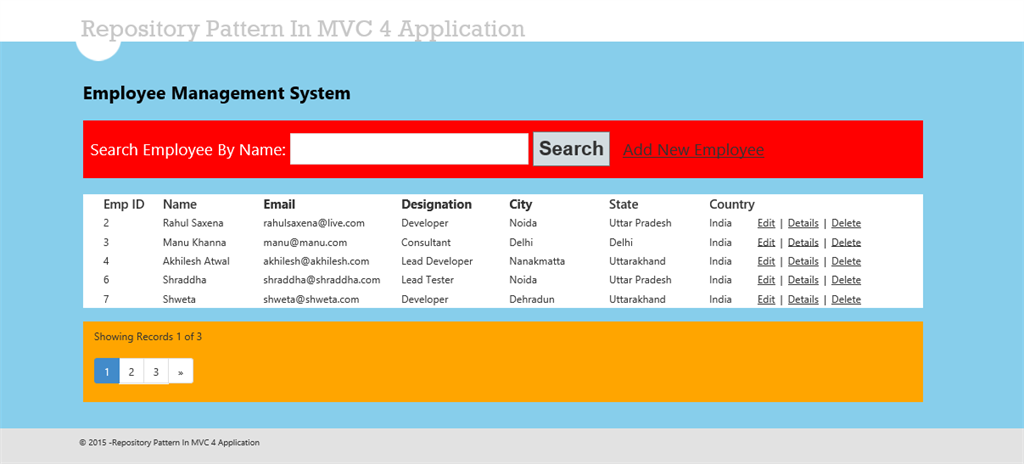
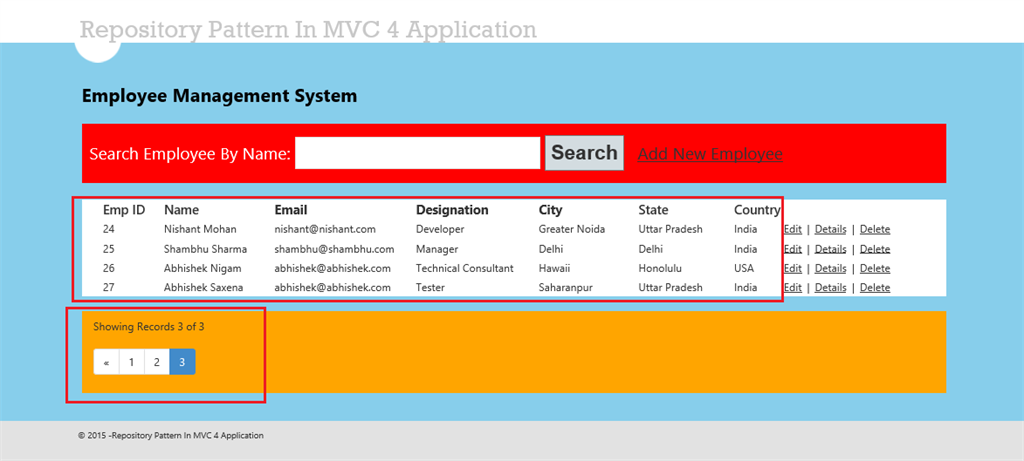
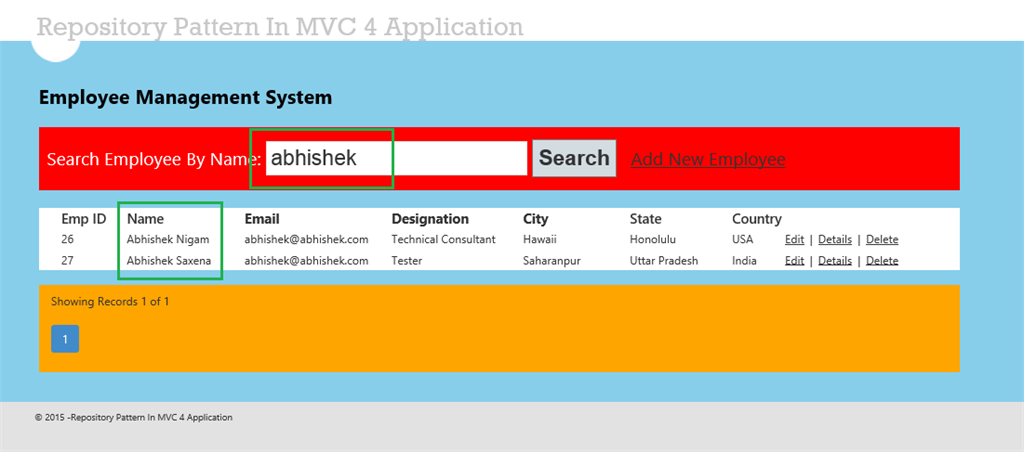
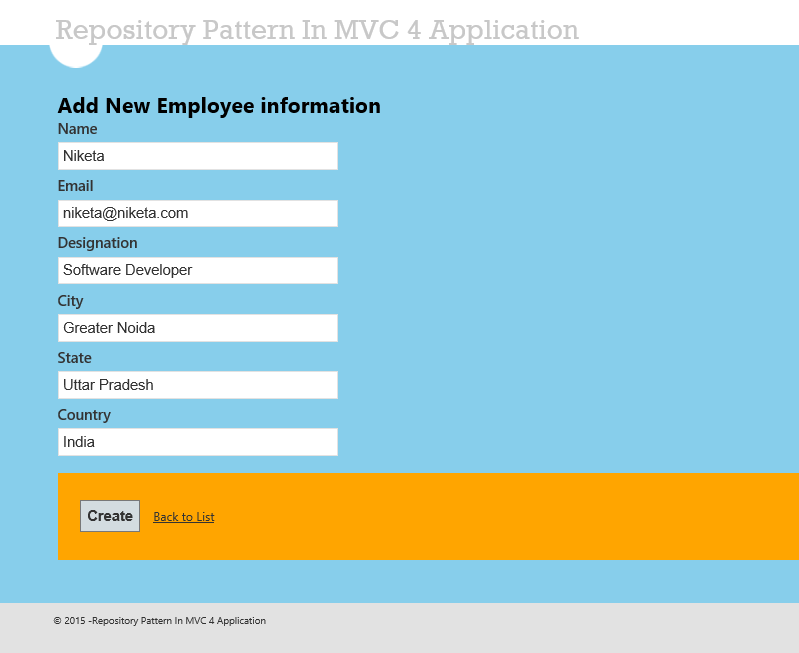
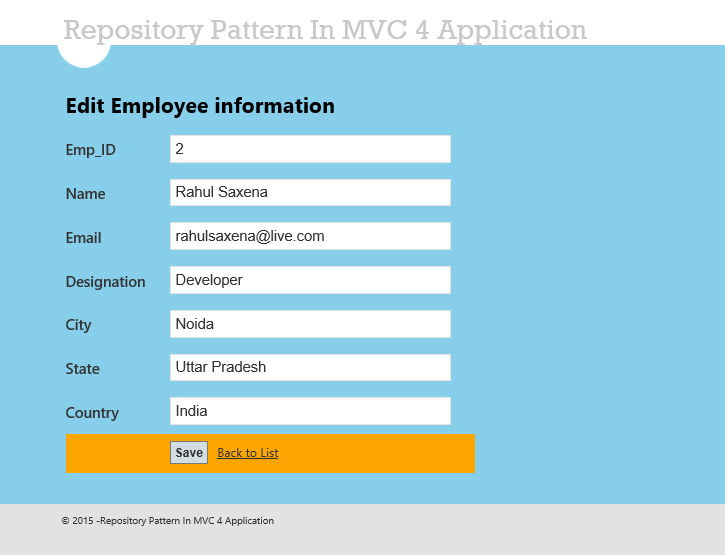
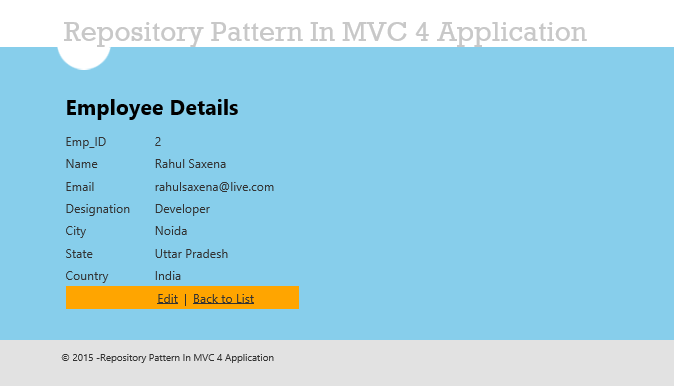
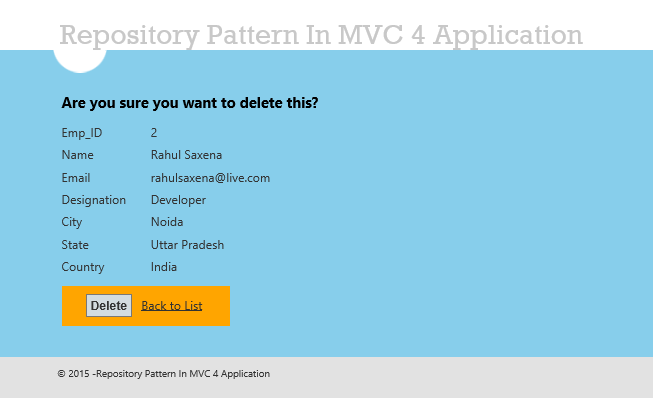
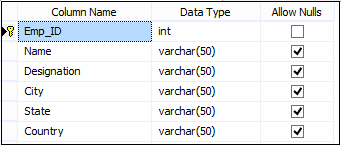
1. @model RepositoryPatternInMVCWithEntityFramework.Employee
3. <script src="~/Scripts/jquery-1.7.1.min.js"></script>
4. <script src="~/Scripts/jquery.validate.min.js"></script>
5. <script src="~/Scripts/jquery.validate.unobtrusive.min.js"></script>
6. <h2>Edit Employee information</h2>
8. @using (Html.BeginForm())
9. {
10. @Html.AntiForgeryToken()
12. @Html.ValidationSummary(true)
14. <table>
15. <tr>
16. <td>@Html.LabelFor(model => model.Emp\_ID)</td>
17. <td>
18. @Html.EditorFor(model => model.Emp\_ID, new { disabled = "disabled", @readonly = "readonly" })
20. @Html.ValidationMessageFor(model => model.Emp\_ID)</td>
21. </tr>
23. <tr>
24. <td>@Html.LabelFor(model => model.Name)
25. </td>
26. <td>@Html.EditorFor(model => model.Name)
27. @Html.ValidationMessageFor(model => model.Name)</td>
28. </tr>
30. <tr>
31. <td>@Html.LabelFor(model => model.Email)</td>
32. <td>
33. @Html.EditorFor(model => model.Email)
34. @Html.ValidationMessageFor(model => model.Email)</td>
35. </tr>
36. <tr>
37. <td>@Html.LabelFor(model => model.Designation)</td>
38. <td>
39. @Html.EditorFor(model => model.Designation)
40. @Html.ValidationMessageFor(model => model.Designation)
41. </td>
42. </tr>
43. <tr>
44. <td>@Html.LabelFor(model => model.City)</td>
45. <td>
46. @Html.EditorFor(model => model.City)
47. @Html.ValidationMessageFor(model => model.City)</td>
48. </tr>
49. <tr>
50. <td>@Html.LabelFor(model => model.State)</td>
51. <td>@Html.EditorFor(model => model.State)
52. @Html.ValidationMessageFor(model => model.State)</td>
53. </tr>
54. <tr>
55. <td>@Html.LabelFor(model => model.Country)</td>
56. <td>
57. @Html.EditorFor(model => model.Country)
58. @Html.ValidationMessageFor(model => model.Country)</td>
59. </tr>
60. <tr style="background-color: orange; padding: 25px;">
61. <td></td>
62. <td>
63. <input type="submit" value="Save" />
64. @Html.ActionLink("Back to List", "Index")
65. </td>
66. </tr>
67. </table>
68. }

Now add a Delete View by right-clicking on the Delete Action Method.

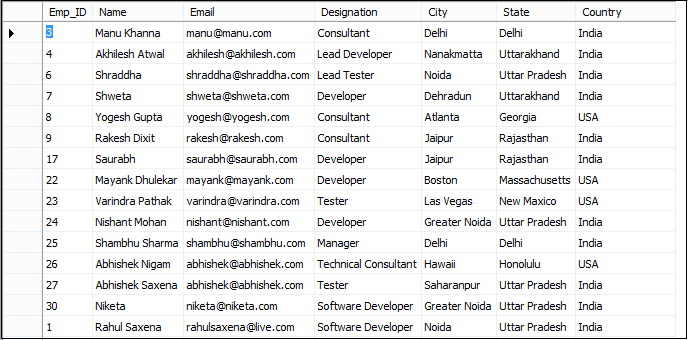
**Delete.cshtml**

1. @model RepositoryPatternInMVCWithEntityFramework.Employee
3. <h3>Are you sure you want to delete this?</h3>
4. <table>
5. <tr>
6. <td>@Html.DisplayNameFor(model => model.Emp\_ID)</td>
7. <td>@Html.DisplayFor(model => model.Emp\_ID)</td>
8. </tr>
9. <tr>
10. <td>@Html.DisplayNameFor(model => model.Name)</td>
11. <td>@Html.DisplayFor(model => model.Name)</td>
12. </tr>
13. <tr>
14. <td>@Html.DisplayNameFor(model => model.Email)</td>
15. <td>@Html.DisplayFor(model => model.Email)</td>
16. </tr>
17. <tr>
18. <td>@Html.DisplayNameFor(model => model.Designation)</td>
19. <td>@Html.DisplayFor(model => model.Designation)</td>
20. </tr>
21. <tr>
22. <td>@Html.DisplayNameFor(model => model.City)</td>
23. <td>@Html.DisplayFor(model => model.City)</td>
24. </tr>
25. <tr>
26. <td>@Html.DisplayNameFor(model => model.State)</td>
27. <td>@Html.DisplayFor(model => model.State)</td>
28. </tr>
29. <tr>
30. <td>@Html.DisplayNameFor(model => model.Country)</td>
31. <td>@Html.DisplayFor(model => model.Country)</td>
32. </tr>
33. </table>
35. @using (Html.BeginForm())
36. {
37. @Html.AntiForgeryToken()
39. <table>
40. <tr style="background-color: orange; padding: 25px;">
41. <td></td>
42. <td>
43. <input type="submit" value="Delete" />
45. @Html.ActionLink("Back to List", "Index")
46. </td>
48. </tr>
49. </table>
50. }

Now run the application.

Showing All Records with Paging and Sorting.  
  
  
Now go to page 3 .  
  
  
Now do a search.  
  
  
Now click on Add New Employer.  
  
  
Now click on any of the records Edit button.  
  
  
Now click on Details.  
  
  
Now click on the Delete option.  
  
  
To perform CRUD operations you will see that we are not calling the DB method directly here. We are using the Repository Pattern.  
  
For this application my Employee Table in design mode is:  
  
  
  
The following is the script of my table:

1. **CREATE** **TABLE** [dbo].[Employee](
2. [Emp\_ID] [**int**] IDENTITY(1,1) NOT NULL,
3. [**Name**] [**varchar**](50) NULL,
4. [Email] [**varchar**](500) NULL,
5. [Designation] [**varchar**](50) NULL,
6. [City] [**varchar**](50) NULL,
7. [State] [**varchar**](50) NULL,
8. [Country] [**varchar**](50) NULL,
9. **CONSTRAINT** [PK\_Employee] **PRIMARY** **KEY** CLUSTERED
10. (
11. [Emp\_ID] **ASC**)**WITH** (PAD\_INDEX = **OFF**, STATISTICS\_NORECOMPUTE = **OFF**, IGNORE\_DUP\_KEY = **OFF**, ALLOW\_ROW\_LOCKS = **ON**, ALLOW\_PAGE\_LOCKS = **ON**)
12. **ON** [**PRIMARY**]) **ON** [**PRIMARY**]
14. GO
16. **SET** ANSI\_PADDING **OFF**
17. GO

The following is the data in my table:  
  
  
  
Enjoy programming.  
  
Next I will write about the Generic Repository Pattern.

# **Generic Repository Pattern in MVC Application Using Entity Framework**

<https://www.c-sharpcorner.com/UploadFile/rahul4_saxena/generic-repository-pattern-in-mvc-application-using-entity-f/>