Factory Pattern in .Net With an Example

Source: http://www.c-sharpcorner.com/UploadFile/97fc7a/factory-pattern-in-net-with-an-example/

While learning about design patterns, I came to understand the most frequently used term, Factory Pattern. I searched the internet and came across numerous learning points. After a lot of search and study, I endeavored to find the definition of Factory Pattern.

In this article I try to explore this pattern in a way that is easy and with the very interesting topic Mobile.

An important aspect of software design is the manner in which objects are created. Thus, it is not only important what an object does or what it models, but also in what manner it was created.

Initially an object is created with the "new" operator. That basic mechanism of object creation could result in design problems or added complexity to the design. On each Object creation we must use the new keyword. The Factory helps you to reduce this practice and use the common interface to create an object.

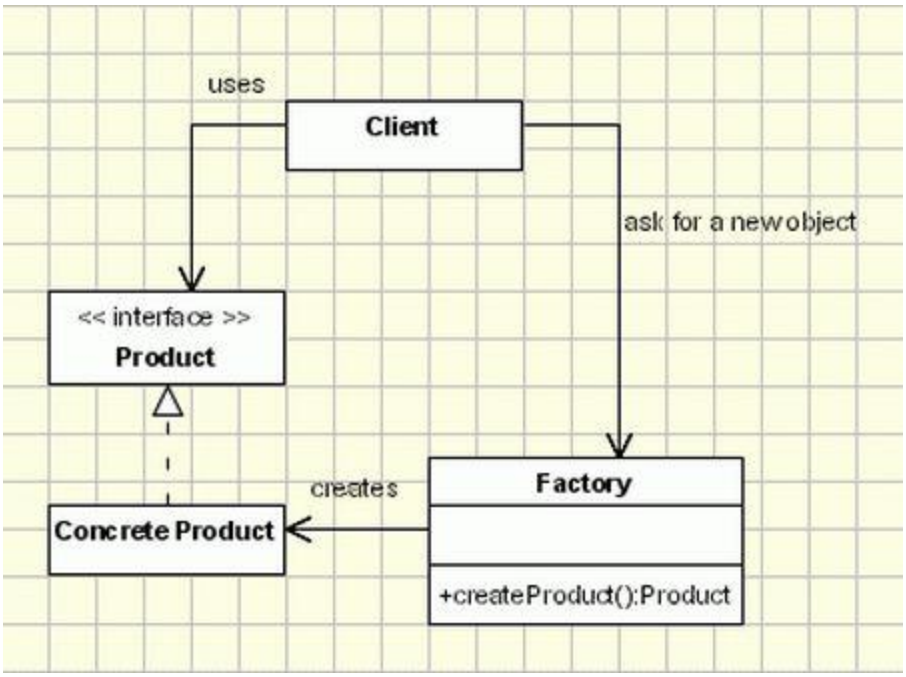
Factory Pattern Definition

GOF says: Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory Method lets a class defer instantiation to subclasses.

The Factory Pattern is a Creational Pattern that simplifies object creation. You need not worry about the object creation; you just need to supply an appropriate parameter and factory to provide you a product as needed.

- Creating objects without exposing the instantiation logic to the client
- Referring to the newly created objects through a common interface

One of the most traditional designs of a Factory Pattern is depicted below:



Let's proceed and use a very general example for this situation. That is about getting the Mobile details, wherein a customer may wish to get the details of various kinds of mobiles and their OS types. An illustrative image is depicted below:

A mobile detail provides three different types of mobiles (Samsung, Apple and Nokia). Based on the client choice, an application provides the details of a specific mobile. The easiest way to get the details is, the client applications will instantiate the desired class by directly using the new operator.

However it shows the tight coupling between the Client and the Mobile class. The Client application must be aware of all the concrete Mobiles (Samsung, Apple and Nokia) and their instantiation logic. This methodology often introduces a great deal of the rigidity in the system. Also the coupling introduced

between classes is extremely difficult to overcome if the requirements change frequently.

The Factory does the same work in a bit of a different manner to create an object. Which is not visible to the client? How the Factory overcomes these fall backs is by delegating the task of object creation to a factory-class (for example FactoryClass.cs) in our sample. The factory-class completely abstracts the creation and initialization of the product from the client-application. It helps to the client application to keep away from Object creation and the concrete details of a mobile.

A typical image of a Factory pattern is depicted below to understand the fact.

The images given above simply specifies the association of the Client app with the Factory class and the creation of the object Via the Factory class.

We do have a Factory class named Factory Class. Factory class is responsible for creating an object and its initiation. The Factory class has one method named CreateMobileObject that creates an object on behalf of the parameter being passed by the user (client) and returns the mobile type.

Code snippet for the Factory class:

```
public static class FactoryClass
{
   public static IMobile CreateMobileObject(MobileType mobileType)
   {
      IMobile objIMobile=null;
      switch (mobileType)
      {
         case MobileType.Samsung:
            objIMobile = new Samsung();
            return objIMobile;

         case MobileType.Apple:
            objIMobile = new Apple();
            return objIMobile;

         case MobileType.Nokia:
            objIMobile = new Nokia();
            return objIMobile;

         default:
            return null;
      }
   }
```

```
}
```

There are a few classes that I've built for the mobile classes that implements the IMobile interface.

```
public class Samsung : IMobile
{
   public string ModelName()
   {
      return "Samsung Galaxy Grand";
   }

    public string OperatingSystem()
   { return "Samsung Uses Android OS For Galaxy Mobile series "; }
}

public class Apple : IMobile
{
   public string ModelName()
   {
      return "Apple IPhone 5";
   }

    public string OperatingSystem()
   { return "Apple Uses ios OS for Apple Mobiles "; }
}

public class Nokia : IMobile
{
   public string ModelName()
   {
      return "Nokia Lumia 960";
   }

   public string OperatingSystem()
   { return "Nokia Uses Symbion OS for Lumia Mobile series "; }
}
```

Now proceeding further and passing a parameter to the static method CreateMobileObject() it accepts a parameter of the desired type.

Code snippet for the Main is as follows:

```
                                                    Passing an Samsung type of parameter to get
                                                    the details using Factory class Method
class Program
{
    static void Main(string[] args)
    {
        IMobile objSamsung = FactoryClass.CreateMobileObject(MobileType.Samsung);

        Console.WriteLine(string.Format("The Mobile has been created of type {0} and {1} ", objSamsung.ModelName(), objSamsung.OperatingSystem()));
        //Console.ReadLine();
        Console.Write(Environment.NewLine);                Passing an Apple type of parameter to get the
        Console.Write(Environment.NewLine);                details using Factory class Method

        //objFac = new FactoryClass();
        IMobile objApple = FactoryClass.CreateMobileObject(MobileType.Apple);

        Console.WriteLine(string.Format("The Mobile has been created of type {0} and {1} ", objApple.ModelName(), objApple.OperatingSystem()));
        Console.ReadLine();

    }
}
```

Press F5 and you will get results as in the following black window:

The image above gives you the details of two mobiles that we passed into the factory class method.

Advantages

- Easy to implement

- Client application code doesn't need to change drastically
- Moreover, the tight coupling between client and mobile classes is overcome and turned into a coupling between the factory and mobile classes. Hence the client needs not understand the instantiation logic of the products.

Disadvantages

- If we add any new product (mobile) then we need a new case statement in the CreateMobileObject method of the Factory class. This violates open/closed design principle.
- We can avoid modifying the Factory class by using sub classing. But sub classing means replacing all the factory class references everywhere throughout the code.
- We have a tight coupling between the Factory class and products.

A sample app is attached as a reference.

Thanks,

Keep Coding and Be Happy.