Xoru.eu
Thierry Brémard
t.bremard@gmail.com

2nd November 2014

Factory v1.0

First, let's recall what is an abstract class.

Msdn defines abstract classes as:
Abstract classes are closely related to interfaces. They are classes that cannot be instantiated, and are frequently either partially implemented, or not at all implemented. One key difference between abstract classes and interfaces is that a class may implement an unlimited number of interfaces, but may inherit from only one abstract (or any other kind of) class. A class that is derived from an abstract class may still implement interfaces. Abstract classes are useful when creating components because they allow you specify an invariant level of functionality in some methods, but leave the implementation of other methods until a specific implementation of that class is needed. They also version well, because if additional functionality is needed in derived classes, it can be added to the base class without breaking code.

```
// C#
abstract class WashingMachine
{
   public WashingMachine()
   {
      // Code to initialize the class goes here.
   }

   abstract public void Wash();
   abstract public void Rinse(int loadSize);
   abstract public long Spin(int speed);
}
```

In the above example, an abstract class is declared with one implemented method and three unimplemented methods. A class inheriting from this class would have to implement the Wash, Rinse, and Spin methods. The following example shows what the implementation of this class might look like:

```
// C#
class MyWashingMachine : WashingMachine
{
   public MyWashingMachine()
   {
      // Initialization code goes here.
   }

   override public void Wash()
   {
      // Wash code goes here.
   }

   override public void Rinse(int loadSize)
   {
      // Rinse code goes here.
   }

   override public long Spin(int speed)
   {
      // Spin code goes here.
   }
}
```

When implementing an abstract class, you must implement each abstract (**MustOverride**) method in that class, and each implemented method must receive the same number and type of arguments, and have the same return value, as the method specified in the abstract class.

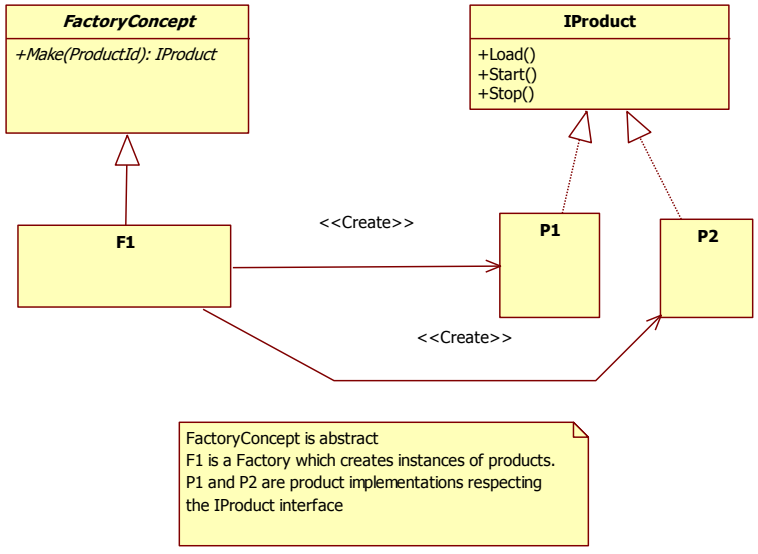
The Factory pattern shows how design pattern can be powerful but also shows that concentration is very important to understand it!

Imagine a Mercedes car, it is built in a Factory. The Mercedes Factory can build SLR car or SLK cars. Imagine another factory from Audi, this other factory will build other kind of cars like A8. The factory pattern give you the tool to program things such as you can exchange Mercedes and Audi, and still be able to produce and test cars! If you understand this, you are on the way to understand the Factory design pattern. I use this example to give the full sense of the Factory name, so you understand the origins of its meaning: Factory is a place where you build things and different factories can make different kind of objects even if this objects have the same specifications.

The factory design pattern consists of:

- An abstract class (or interface) is defined to **create** some products, let's call it FactoryConcept (or IFactory)
- An implementation of FactoryConcept is done to really create the products, let's call the implementation F1.
- Products that are really created by F1 implement a common interface: IProduct
- This same interface IProduct is returned by Make() operation of the Factory.

The power of this architecture is that the function which receives an instance of FactoryConcept can receive either a real F1 or another factory like F2 which could return other products than P1 and P2 if these other product also implement IProduct interface.



UML Class diagram for Factory design pattern

Let's analyse the graph of the Factory Pattern:

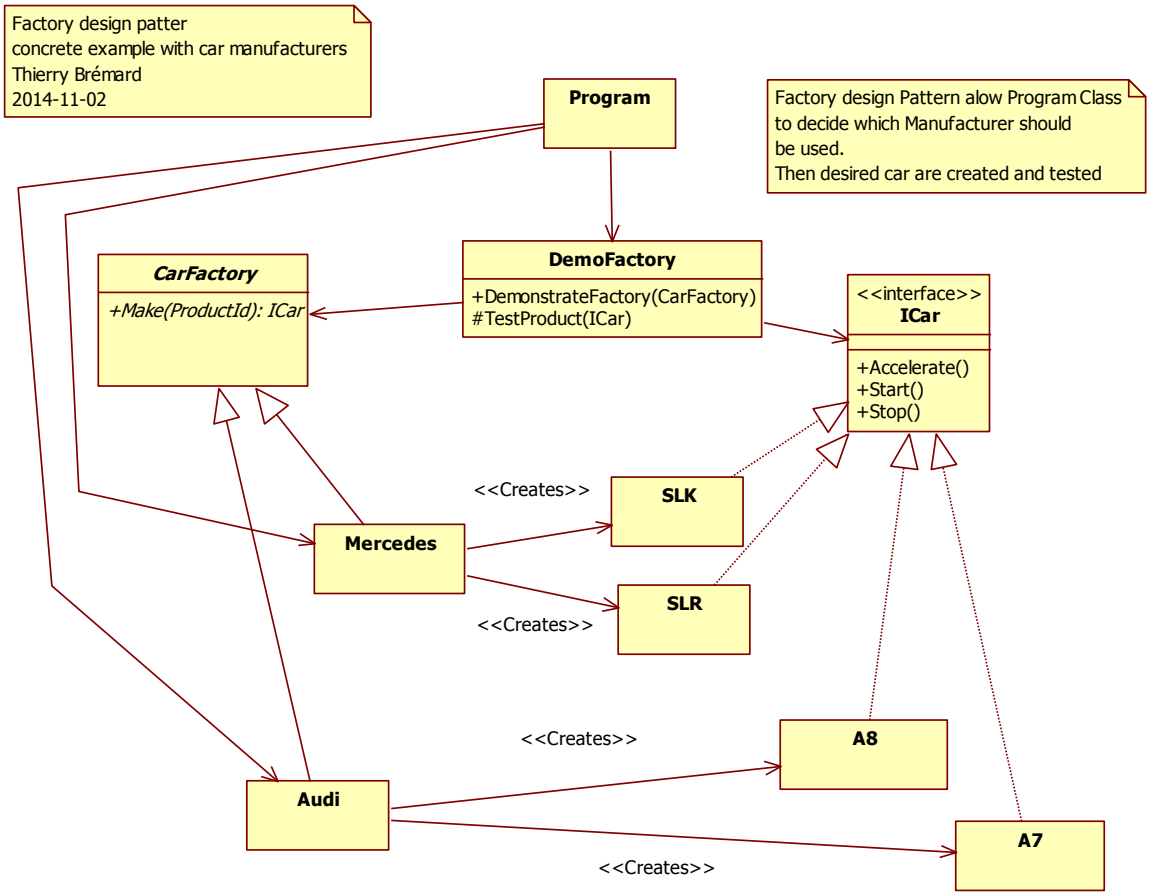
- F1 is linked to FactoryConcept with a arrow, which is finished by a triangle, hence F1 inherits from FactoryConcept.
- The FactoryConcept class has its name in italics, so this class is abstract.
- The Make Operation has its name in italics, so this operation is abstract, it takes in input a ProcuctId and returns a product instance as an Interface.
- The F1 Class has two associations

- The association which links to P1 is an association because the arrow head is a V (not a triangle). The double quote is called a stereotype. Stereotypes define the kind of association, here “Create” means that the F1 Class create a new instance of class P1.
- The association which links F1 to P2 means that the F1 Class create a new instance of P2.
- The arrow which links P1 to IProduct is with a triangle in the head this means that P1 inheritance from IProduct. The fact that the line is dashed means that IProduct is an interface, and not a class.
- Thus P2 also implements the IProduct interface.

If you understand everything you can match the first example with the design pattern example:

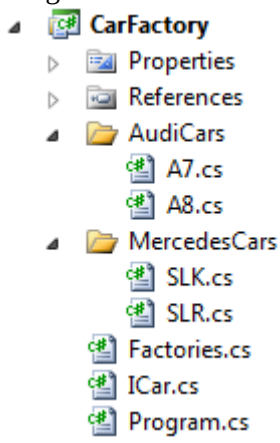
- F1 is Mercedes Factory
- IProduct is the car specification
- P1 is a real car model like the SLR model from Mercedes
- P2 is another real car model like the SLK model from Mercedes.

Now let's work on our concrete example the car factories with class diagram and c# source code



The key of Factory Pattern is that the class DemoFactory has no reference to any concrete class. There is only association to abstract class and to interface. This is really important in code design of object oriented software.
The fact to use interface or abstract class is not important. You may do your own choice.

Code
I organized the code with the different files:



Factories.cs

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

namespace CarFactory
{

    public abstract class CarFactory
    {
        abstract public ICar Make(ProductId id);
    }

    class Mercedes : CarFactory
    {
        public override ICar Make(ProductId p)
        {
            ICar ret = null;
            switch (p)
            {
                case ProductId.ID1:
                    ret = new SLR();
                    break;
                case ProductId.ID2:
                    ret = new SLK();
                    break;
            }
            return ret;
        }
    }


    class Audi : CarFactory
    {
        public override ICar Make(ProductId p)
        {
            ICar ret = null;
            switch (p)
            {
                case ProductId.ID1:
                    ret = new A7();
                    break;
                case ProductId.ID2:
                    ret = new A8();
                    break;
            }
            return ret;
        }
    }


}
```

ICar.cs

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

namespace CarFactory
{
    public enum ProductId { ID1, ID2 };
    public interface ICar
    {
        void Accelerate();
        void Start();
        void Stop();
    }

}
```

SLK.cs

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

namespace CarFactory
{
    public class SLK : ICar
    {
        public void Accelerate()
        {
            Console.WriteLine("SLK is accelerating");
        }
        public void Start()
        {
            Console.WriteLine("SLK is started");
        }
        public void Stop()
        {
            Console.WriteLine("SLK is stopped and made 100 meters");
        }
    }
}
```

SLR.cs

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

namespace CarFactory
{
    public class SLR : ICar
    {
        public void Accelerate()
        {
            Console.WriteLine("SLR is accelerating");
        }
        public void Start()
        {
                Console.WriteLine("SLR is started");
        }
        public void Stop()
        {
            Console.WriteLine("SLR is stopped and made 1000 meters");
        }

    }
}
```

A7.cs

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

namespace CarFactory
{
    class A7:ICar
    {
        public void Accelerate()
        {
            Console.WriteLine("A7 is accelerating");
        }
        public void Start()
        {
            Console.WriteLine("A7 is started");
        }
        public void Stop()
        {
            Console.WriteLine("A7 is stopped and made 300 meters");
        }
    }
}
```

A8.cs

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

namespace CarFactory
{
    class A8 : ICar
    {
        public void Accelerate()
        {
            Console.WriteLine("A8 is accelerating");
        }
        public void Start()
        {
            Console.WriteLine("A8 is started");
        }
        public void Stop()
        {
            Console.WriteLine("A8 is stopped and made 800 meters");
        }
    }
}
```

Program.cs

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

namespace CarFactory
{
    class Program
    {
        static void Main(string[] args)
        {

            DemoFactory demo;
            demo = new DemoFactory();

            Mercedes mercedes;
            mercedes = new Mercedes(); // we could choose another Factory implementation
            demo.DemonstrateFactory(mercedes);// We give an instance of Mercedes to a function which takes as input
an instance of CarFactory

            Audi audi;
            audi = new Audi();
            demo.DemonstrateFactory(audi);
            Console.ReadKey();
        }


    }

    class DemoFactory
    {

        /// <summary>
        /// The caller must have chosen the Factory
        /// implementation and gives us the reference as
        /// an abstract class
        /// </summary>
        /// <param name="f">an implementation of CarFactory
        /// whatever the kind of implementation we are able
        /// to produce Products and test them</param>
        public void DemonstrateFactory(CarFactory f)
        {
            ICar car;
            car = f.Make(ProductId.ID1);
            TestProduct(car);

            car = f.Make(ProductId.ID2);
            TestProduct(car);
        }

        protected void TestProduct(ICar car)
        {
            car.Start();
            car.Accelerate();
            car.Stop();
        }
    }
}
```

Program Output
SLR is started SLR is accelerating SLR is stopped and made 1000 meters SLK is started SLK is accelerating SLK is stopped and made 100 meters A7 is started A7 is accelerating A7 is stopped and made 300 meters A8 is started A8 is accelerating A8 is stopped and made 800 meters