عنوان: **پیاده سازی ServiceHostFactory سفارشی در WCF**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۱۱/۱۲ ۲۱:۵
آدرس: www.dotnettips.info
گروه‌ها: WCF, Self-Hosting, ServiceFactory, IIS-Hosting

در این مثال برای اینکه Instance Provider سفارشی خود را بتوانیم به عنوان یک Behavior به سرویس اضافه نماییم باید به خاصیت Description.Behaviors شی ServiceHost دسترسی داشته باشیم. زمانی که در پروژه‌های WCF از روش Self Hosting برای هاست سرویس‌ها استفاده کنیم به دلیل دسترسی مستقیم به شی ServiceHost هر گونه تنظیمات و عملیات Customization به راحتی امکان پذیر است ؛ اما در IIS Hosting، از آن جا که به صورت پیش فرض از ServiceHostFactory موجود در WCF استفاده می‌شود ما دسترسی به شی ServiceHost نداریم. برای حل این مسئله باید یک CustomServiceHostFactory ایجاد نماییم که به راحتی در WCF این امکان تدارک دیده شده است.
بررسی یک مثال:
ابتدا کلاسی به صورت زیر ایجاد نمایید. در این کلاس می‌توانید کدهای لازم برای سفارشی کردن شی ServiceHost را قرار دهید:

```
public class CustomServiceHost : ServiceHost
{
   public CustomServiceHost( Type t, params Uri baseAddresses ) :
      base( t, baseAddresses ) {}

   public override void OnOpening()
   {
       this.Description.Add( new MyServiceBehavior() );
   }
}
```

اگر از این به بعد به جای استفاده از ServiceHost مستقیما از CustomServiceHost استفاده نماییم، MyServiceBehavior به صورت خودکار به عنوان یک ServiceBehavior برای سرویس مورد نظر در نظر گرفته می‌شود. برای این که هنگام هاست سرویس مورد نظر به صورت خودکار از این شی کلاس استفاده شود می‌توان کلاس Factory ساخت سرویس را تغییر داد به صورت زیر:

```
public class CustomServiceHostFactory : ServiceHostFactory
{
   public override ServiceHost CreateServiceHost( Type t, Uri[] baseAddresses )
   {
      return new CustomServiceHost( t, baseAddresses )
   }
}
```

حال بر روی سرویس مورد نظر کلیک راست کرده و گزینه View MarkUp را انتخاب نمایید، چیزی شبیه به گزینه زیر را مشاهده خواهید کرد:

```
<%@ ServiceHost Language="C#" Debug="true" Service="WcfService1.Service1" CodeBehind="Service1.svc.cs"
%>
```

کافیست کلاس CustomServiceHostFactory را به عنوان Factory این سرویس مشخص نماییم. به صورت زیر:

```
<%@ ServiceHost Language="C#" Debug="true" Factory="CustomServiceHostFactory"
Service="WcfService1.Service1" CodeBehind="Service1.svc.cs" %>
```

از این به بعد عملیات وهله سازی از سرویس بر اساس تنظیمات پیش فرض صورت گرفته در این کلاس‌ها انجام می‌گیرد.

عنوان: **Design Pattern: Factory**
نویسنده: مهسا حسن کاشی
تاریخ: ۱۳۹۲/۱۱/۲۶ ۲۲:۱۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Design patterns, SOLID, Object Oriented Design, ServiceFactory, Patterns, Software architecture

الگوهای طراحی، سندها و راه حلهای از پیش تعریف شده و تست شده‌ای برای مسائل و مشکلات روزمره‌ی برنامه نویسی می‌باشند که هر روزه ما را درگیر خودشان می‌کنند. هر چقدر مقیاس پروژه وسیعتر و تعداد کلاسها و اشیاء بزرگتر باشند، درگیری برنامه نویس و چالش برای مرتب سازی و خوانایی برنامه و همچنین بالا بردن کارآیی و امنیت افزون‌تر می‌شود. از همین رو استفاده از ساختارهایی تست شده برای سناریوهای یکسان، امری واجب تلقی می‌شود.

الگوهای طراحی از لحاظ سناریو، به سه گروه عمده تقسیم می‌شوند:

-1 تکوینی: هر چقدر تعداد کلاسها در یک پروژه زیاد شود، به مراتب تعداد اشیاء ساخته شده از آن نیز افزوده شده و پیچیدگی و درگیری نیز افزایش می‌یابد. راه حل‌هایی از این دست، تمرکز بر روی مرکزیت دادن به کلاسها با استفاده از رابطها و کپسوله نمودن (پنهان سازی) اشیاء دارد.

-2 ساختاری: گاهی در پروژه‌ها پیش می‌آید که می‌خواهیم ارتباط بین دو کلاس را تغییر دهیم. از این رو امکان از هم پاشی اجزایِ دیگر پروژه پیش می‌آید. راه حلهای ساختاری، سعی در حفظ انسجام پروژه در برابر این دست از تغییرات را دارند.

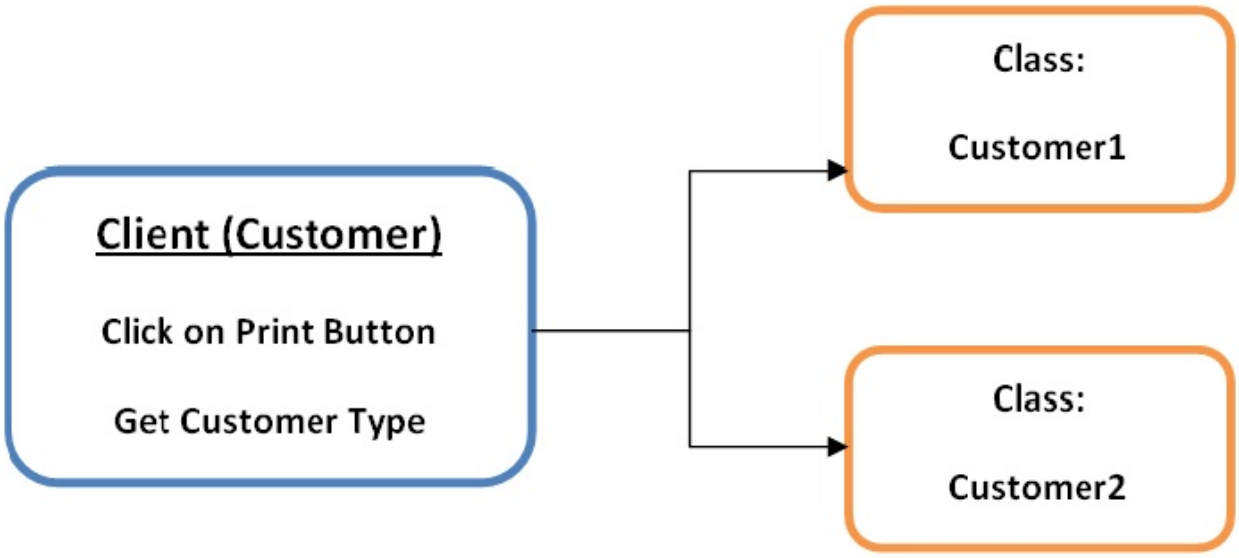
-3 رفتاری: گاهی بنا به مصلحت و نیاز مشتری، رفتار یک کلاس می‌بایستی تغییر نماید. مثلا چنانچه کلاسی برای ارائه صورتحساب داریم و در آن میزان مالیات 30% لحاظ شده است، حال این درصد باید به عددی دیگر تغییر کند و یا پایگاه داده به جای مشاهده‌ی تعدادِ معدودی گره از درخت، حال می‌بایست تمام گره‌ها را ارائه نماید.

الگوی فکتوری:

الگوی فکتوری در دستهء اول قرار می‌گیرد. من در اینجا به نمونه‌ای از مشکلاتی که این الگو حل می‌نماید، اشاره می‌کنم:

فرض کنید یک شرکت بزرگ قصد دارد تا جزییات کامل خرید هر مشتری را با زدن دکمه چاپ ارسال نماید. چنین شرکت بزرگی بر اساس سیاستهای داخلی، بر حسب میزان خرید، مشتریان را به چند گروه مشتری معمولی و مشتری ممتاز تقسیم می‌نماید. در نتیجه نمایش جزییات برای آنها با احتساب میزان تخفیف و به عنوان مثال تعداد فیلدهایی که برای آنها در نظر گرفته شده است، تفاوت دارد. بنابراین برای هر نوع مشتری یک کلاس وجود دارد.

یک راه این است که با کلیک روی دکمه‌ی چاپ، نوع مشتری تشخیص داده شود و به ازای نوع مشتری، یک شیء از کلاس مشخص شده برای همان نوع ساخته شود.



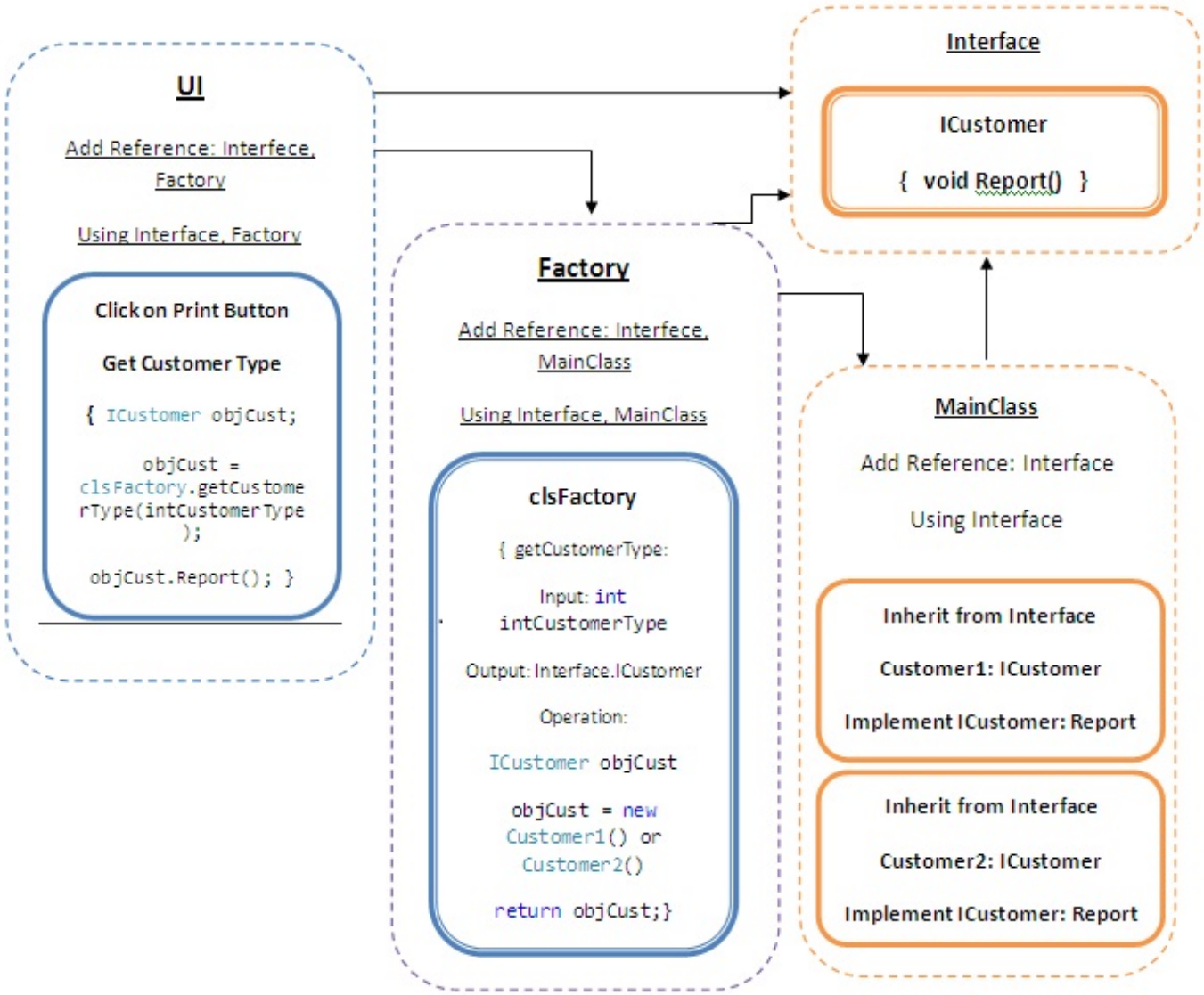

```
// Get Customer Type from Customer click on Print Button
            int customerType = 0;

            // Create Object without instantiation
            object obj;


            //Instantiate obj according to customer Type
            if (customerType == 1)
            {
                obj = new Customer1();
            }
            else if (customerType == 2)
            {
                obj = new Customer2();
            }
            // Problem:
            //          1: Scattered New Keywords
            //          2: Client side is aware of Customer Type
```

همانگونه که مشاهده می‌نمایید در این سبک کدنویسی غیرحرفه‌ای، مشکلاتی مشهود است که قابل اغماض نیستند. در ابتدا سمت کلاینت دسترسی مستقیم به کلاسها دارد و همانگونه که در شکل بالا قابل مشاهده است کلاینت مستقیما به کلاس وصل است. مشکل دوم عدم پنهان سازی کلاس از دید مشتری است.

راه حل: این مشکل با استفاده از الگوی فکتوری قابل حل است. با استناد به الگوی فکتوری، کلاینت تنها به کلاس فکتوری و یک اینترفیس دسترسی دارد و کلاسهای فکتوری و اینترفیس، حق دسترسی به کلاسهای اصلی برنامه را دارند.



گام نخست: در ابتدا یک class library به نام Interface ساخته و در آن یک کلاس با نام ICustomer می سازیم که متد Report() را معرفی می‌نماید.

Interface//

```
namespace Interface
{
    public interface ICustomer
    {
        void Report();
    }
}
```

گام دوم: یک class library به نام MainClass ساخته و با Add Reference کلاس Interface را اضافه نموده، در آن دو کلاس با نام Customer1, Customer2 می‌سازیم و using Interface را Import می‌نماییم. هر دو کلاس از ICustomer ارث می‌برند و سپس متد Report() را در هر دو کلاس Implement می‌نماییم.

```
// Customer1
using System;
```

```
using Interface;

namespace MainClass
{
    public class Customer1 : ICustomer
    {
        public void Report()
        {
            Console.WriteLine("این گزارش مخصوص مشتری نوع اول است");
        }
    }
}

//Customer2
using System;
using Interface;

namespace MainClass
{
   public class Customer2 : ICustomer
    {
        public void Report()
        {
            Console.WriteLine("این گزارش مخصوص مشتری نوع دوم است");
        }
    }
}
```

گام سوم: یک class library به نام FactoryClass ساخته و با Add Reference کلاس Interface, MainClass را اضافه نموده، در آن یک کلاس با نام clsFactory می سازیم و using Interface, using MainClass را Import می‌نماییم. پس از آن یک متد با نام getCustomerType ساخته که ورودی آن نوع مشتری از نوع int است و خروجی آن از نوع Interface-ICustomer و بر اساس کد نوع مشتری object را از کلاس Customer1 و یا Customer2 می‌سازیم و آن را return می نماییم.

```
//Factory
using System;
using Interface;
using MainClass;

namespace FactoryClass
{
    public class clsFactory
    {
        static public ICustomer getCustomerType(int intCustomerType)
        {
            ICustomer objCust;
            if (intCustomerType == 1)
            {
                objCust = new Customer1();
            }
            else if (intCustomerType == 2)
            {
                objCust = new Customer2();
            }
            else
            {
                return null;
            }
            return objCust;
        }
    }
}
```

گام چهارم (آخر): در قسمت UI Client، کد نوع مشتری را از کاربر دریافت کرده و با Add Reference کلاس Interface, FactoryClass را اضافه نموده (دقت نمایید هیچ دسترسی به کلاس‌های اصلی وجود ندارد)، و using Interface, using FactoryClass را Import می‌نماییم. از clsFactory تابع getCustomerType را فراخوانی نموده (به آن کد نوع مشتری را پاس می‌دهیم) و خروجی آن را که از نوع اینترفیس است به یک object از نوع ICustomer نسبت می‌دهیم. سپس از این object متد Report را فراخوانی می‌نماییم. همانطور که از شکل و کدها مشخص است، هیچ رابطه ای بین UI(Client) و کلاسهای اصلی برقرار نیست.

```
//UI (Client)
using System;
using FactoryClass;
using Interface;

namespace DesignPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            int intCustomerType = 0;
            ICustomer objCust;
            Console.WriteLine("نوع مشتری را وارد نمایید");
            intCustomerType = Convert.ToInt16(Console.ReadLine());
            objCust = clsFactory.getCustomerType(intCustomerType);
            objCust.Report();
            Console.ReadLine();
        }
    }
}
```