عنوان: **پیاده سازی INotifyPropertyChanged با استفاده از Unity Container**
نویسنده: برات جوادی
تاریخ: ۱۳۹۴/۰۱/۲۴ ۱۴:۲۵
آدرس: www.dotnettips.info
گروه‌ها: WPF, UnityContainer, AOP

AOP چیست

AOP یکی از فناوری‌های مرتبط با توسعه نرم افزار محسوب می‌شود که توسط آن می‌توان اعمال مشترک و متداول موجود در برنامه را در یک یا چند ماژول مختلف قرار داد (که به آن‌ها Aspects نیز گفته می‌شود) و سپس آن‌ها را به مکان‌های مختلفی در برنامه متصل ساخت. عموما Aspects، قابلیت‌هایی را که قسمت عمده‌ای از برنامه را تحت پوشش قرار می‌دهند، کپسوله می‌کنند. اصطلاحا به این نوع قابلیت‌های مشترک، تکراری و پراکنده مورد نیاز در قسمت‌های مختلف برنامه، Cross cutting concerns نیز گفته می‌شود؛ مانند اعمال ثبت وقایع سیستم، امنیت، مدیریت تراکنش‌ها و امثال آن. با قرار دادن این نیازها در Aspects مجزا، می‌توان برنامه‌ای را تشکیل داد که از کدهای تکراری عاری است.

پیاده سازی INotifyPropertyChanged یکی از این مسائل می‌باشد که می‌توان آن را در یک Aspect محصور و در ماژول‌های مختلف استفاده کرد.

مسئله:

کلاس زیر مفروض است:

```
public class Foo
{
        public virtual int Id { get; set; }

        public virtual string Name { get; set; }
}
```

اکنون می‌خواهیم کلاس Foo را به INotifyPropertyChanged مزین، و یک Subscriber به قسمت set پراپرتی‌های کلاس تزریق کنیم.

راه حل:

ابتدا پکیج‌های Unity را از Nuget دریافت کنید:

```
PM> Install-Package Unity.Interception
```

این پکیج وابستگی‌های خود را که Unity و CommonServiceLocator هستند نیز دریافت می‌کند.

حال یک Interceptor که اینترفیس IInterceptionBehavior را پیاده سازی می‌کند، می‌نویسیم:

```
namespace NotifyPropertyChangedInterceptor.Interceptions
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel;
    using System.Reflection;
    using Microsoft.Practices.Unity.InterceptionExtension;

    class NotifyPropertyChangedBehavior : IInterceptionBehavior
    {
        private event PropertyChangedEventHandler PropertyChanged;

        private readonly MethodInfo _addEventMethodInfo =
            typeof(INotifyPropertyChanged).GetEvent("PropertyChanged").GetAddMethod();

        private readonly MethodInfo _removeEventMethodInfo =
            typeof(INotifyPropertyChanged).GetEvent("PropertyChanged").GetRemoveMethod();


        public IMethodReturn Invoke(IMethodInvocation input, GetNextInterceptionBehaviorDelegate
getNext)
        {
            if (input.MethodBase == _addEventMethodInfo)
            {
                return AddEventSubscription(input);
            }
```

```
            if (input.MethodBase == _removeEventMethodInfo)
            {
                return RemoveEventSubscription(input);
            }

            if (IsPropertySetter(input))
            {
                return InterceptPropertySet(input, getNext);
            }

            return getNext()(input, getNext);
        }

        public bool WillExecute
        {
            get { return true; }
        }

        public IEnumerable<Type> GetRequiredInterfaces()
        {
            yield return typeof(INotifyPropertyChanged);
        }

        private IMethodReturn AddEventSubscription(IMethodInvocation input)
        {
            var subscriber = (PropertyChangedEventHandler)input.Arguments[0];
            PropertyChanged += subscriber;

            return input.CreateMethodReturn(null);
        }

        private IMethodReturn RemoveEventSubscription(IMethodInvocation input)
        {
            var subscriber = (PropertyChangedEventHandler)input.Arguments[0];
            PropertyChanged -= subscriber;

            return input.CreateMethodReturn(null);
        }

        private bool IsPropertySetter(IMethodInvocation input)
        {
            return input.MethodBase.IsSpecialName && input.MethodBase.Name.StartsWith("set_");
        }

        private IMethodReturn InterceptPropertySet(IMethodInvocation input,
GetNextInterceptionBehaviorDelegate getNext)
        {
            var propertyName = input.MethodBase.Name.Substring(4);

            var subscribers = PropertyChanged;
            if (subscribers != null)
            {
                subscribers(input.Target, new PropertyChangedEventArgs(propertyName));
            }

            return getNext()(input, getNext);
        }
    }
}
```

متد Invoke : این متد Behavior مورد نظر را پردازش می‌کند (در اینجا، تزریق یک Subscriber در قسمت set پراپرتی ها). **متد GetRequiredInterfaces** : یک روش است برای یافتن کلاس هایی که با اینترفیس IInterceptionBehavior مزین شده‌اند.
پراپرتی WillExecute : ابن پراپرتی به Unity می‌گوید که این Behavior اعمال شود یا نه. اگر مقدار برگشتی آن false باشد، متد Invoke اجرا نخواهد شد.
همانطور که در متد Invoke مشاهد می‌کنید، شرط هایی برای افزودن و حذف یک Subscriber و چک کردن متد set نوشته شده و در غیر این صورت کنترل به متد بعدی داده می‌شود.

اتصال Interceptor به کلاس ها

در ادامه Unity را برای ساخت یک نمونه از کلاس پیکربندی می‌کنیم:

```
var container = new UnityContainer();

container.RegisterType<Foo, Foo>(
                new AdditionalInterface<INotifyPropertyChanged>(),
                new Interceptor<VirtualMethodInterceptor>(),
                new InterceptionBehavior<NotifyPropertyChangedBehavior>())
                .AddNewExtension<Interception>();
```

توسط متد RegisterType یک Type را با پیکربندی دلخواه به Unity معرفی می‌کنیم. در اینجا به ازای درخواست Foo (اولین پارامتر جنریک)، یک Foo (دومین پارامتر جنریک) برگشت داده می‌شود. این متد تعدادی InjetctionMember (بصورت params) دریافت می‌کند که در این مثال سه InjetctionMember به آن پاس داه شده است:

Interceptor : اطلاعاتی در مورد IInterceptor و نحوه‌ی Intercept یک شیء را نگه داری می‌کند. در اینجا از VirtualMethodInterceptor برای تزریق کد استفاده شده.

InterceptionBehavior : این کلاس Behavior مورد نظر را به کلاس تزریق می‌کند.

AddintionalInterface : کلاس target را مجبور به پیاده سازی اینترفیس دریافتی از پارامتر می‌کند. اگر کلاس behavior، متد GetRequiredInterfaces اینترفیس INotifyPropertyChanged را برمی گرداند، نیازی نیست از AddintionalInterface در پارامتر متد فوق استفاده کنید.

نکته : کلاس VirtualMethodInterceptor فقط اعضای virtual را تحت تاثیر قرار می‌دهد.

اکنون نحوه‌ی ساخت یک نمونه از کلاس Foo به شکل زیر است:

```
var foo = container.Resolve<Foo>();
(foo as INotifyPropertyChanged).PropertyChanged += FooPropertyChanged;
private void FooPropertyChanged (object sender, PropertyChangedEventArgs e)
 {
      // Do some things.......
 }
```

نکته‌ی تکمیلی

طبق مستندات MSDN ، کلاس VirtualMethodInterceptor یک کلاس جدید مشتق شده از کلاس target (در اینجا Foo) می‌سازد. بنابراین اگر کلاس‌های شما دارای Data annotation و یا در کلاس‌های Mapper یک ORM استفاده شده‌اند (مانند کلاس‌های لایه Domain)، بجای VirtualMethodInterceptor از TransparentProxyInterceptor استفاده کنید. سرعت اجرای VirtualMethodInterceptor سریعتر است ؛ اما به یاد داشته که برای استفاده از TransparentProxyInterceptor باید کلاس target از کلاس MarshalByRefObject ارث بری کند. دریافت مثال کامل این مقاله

نظرات خوانندگان

نویسنده: جلال
تاریخ: ۱۳۹۴/۰۱/۲۴ ۲۰:۲۳

این روش به همه‌ی Property Setterهای کلاس بدون در نظر گرفتن نیازهای کاربر/برنامه نویس، فراخوانی PropertyChanged رو اضافه می‌کنه. همینطور ممکنه کاربر بخواد با فراخوانی یه PropertyChanged برای یه Property، بعدش مجددا این رویداد رو برای یه Property دیگه فراخوانی کنه. به نظرم بهتره از روش‌های Attribute Base مثل این استفاده بشه.

نویسنده: برات جوادی
تاریخ: ۱۳۹۴/۰۱/۲۵ ۸:۵۴

- این Interceptor فقط کار تزریق یک Subscriber برای PropertyChanged را به عهده دارد و به سایر نیازها کاری ندارد. ضمن اینکه نیازهای کاربر/برنامه نویس اینجا کمی نامفهوم است!

- هنگام تشخیص متد set در Interceptor میتوان یک شرط دیگر گذاشت و اینکار را انجام داد.
- بسته به سناریو می‌توان از attribute هم استفاده کرد. در اینجا قصدم تزریق برای همه پراپرتی‌ها بوده، درصورتی که تزریق فقط برای برخی از انها باشد، میشه Attribute هم تعریف کرد .