عنوان: **استفاده از AOP Interceptors برای حذف کدهای تکراری INotifyPropertyChanged در WPF**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۱/۲۰ ۱۰:۳
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, AOP, C#, Architecture

هرکسی که با WPF کار کرده باشد با دردی به نام اینترفیس INotifyPropertyChanged و پیاده سازی‌های تکراری مرتبط با آن آشنا است:

```
public class MyClass : INotifyPropertyChanged
{
    private string _myValue;
    public event PropertyChangedEventHandler PropertyChanged;
    public string MyValue
    {
        get
        {
            return _myValue;
        }
        set
        {
            _myValue = value;
            RaisePropertyChanged("MyValue");
        }
    }
    protected void RaisePropertyChanged(string propertyName)
    {
        if (PropertyChanged != null)
            PropertyChanged(this, new PropertyChangedEventArgs(propertyName));
    }
}
```

چندین راه‌حل هم برای ساده سازی و یا بهبود آن وجود دارد از Strongly typed کردن آن تا روش‌های اخیر دات نت 4 و نیم در مورد استفاده از ویژگی‌های متدهای فراخوان. اما ... با استفاده از AOP Interceptors می‌توان در وهله سازی‌ها و فراخوانی‌ها دخالت کرد و کدهای مورد نظر را در مکان‌های مناسبی تزریق نمود. بنابراین در مطلب جاری قصد داریم ارائه متفاوتی را از پیاده سازی خودکار INotifyPropertyChanged ارائه دهیم. به عبارتی چقدر خوب می‌شد فقط می‌نوشتیم :

```
public class MyDreamClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler PropertyChanged;
    public string MyValue { get; set; }
}
```

و ... همه چیز مثل سابق کار می‌کرد. برای رسیدن به این هدف، باید فراخوانی‌های set خواص را تحت نظر قرار داد (یا همان Interception در اینجا). ابتدا باید اجازه دهیم تا set صورت گیرد، پس از آن کدهای معروف RaisePropertyChanged را به صورت خودکار فراخوانی کنیم.

پیشنیازها

ابتدا یک برنامه جدید WPF را آغاز کنید. تنظیمات آن‌را از حالت Client profile به Full تغییر دهید.
سپس همانند قسمت قبل، ارجاعات لازم را به StructureMap و Castle.Core نیز اضافه نمائید:

```
PM> Install-Package structuremap
PM> Install-Package Castle.Core
```

ساختار برنامه

برنامه ما از یک اینترفیس و کلاس سرویس تشکیل شده است:

```
namespace AOP01.Services
{
    public interface ITestService
    {
        int GetCount();
    }
}

namespace AOP01.Services
{
    public class TestService: ITestService
    {
        public int GetCount()
        {
            return 10; //این فقط یک مثال است برای بررسی تزریق وابستگی‌ها
        }
    }
}
```

همچنین دارای یک ViewModel به شکل زیر می‌باشد:

```
using AOP01.Services;
using AOP01.Core;

namespace AOP01.ViewModels
{
    public class TestViewModel  : BaseViewModel
    {
        private readonly ITestService _testService;
        //تزریق وابستگی‌ها در سازنده کلاس
        public TestViewModel(ITestService testService)
        {
            _testService = testService;
        }

        // Note: it's a virtual property.
        public virtual string Text { get; set; }
    }
}
```

سه نکته در این ViewModel حائز اهمیت هستند:

الف) استفاده از کلاس پایه BaseViewModel برای کاهش کدهای تکراری مرتبط با INotifyPropertyChanged که به صورت زیر تعریف شده است:

```
using System.ComponentModel;

namespace AOP01.Core
{
    public abstract class BaseViewModel : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;

        public void RaisePropertyChanged(string propertyName)
        {
            var handler = PropertyChanged;

            if (handler != null)
                handler(this, new PropertyChangedEventArgs(propertyName));
        }
    }
}
```

ب) کلاس سرویس، در حالت تزریق وابستگی‌ها در سازنده کلاس در اینجا مورد استفاده قرار گرفته است. وهله سازی خودکار آن توسط کلاس‌های پروکسی و DI صورت خواهند گرفت.

ج) خاصیتی که در اینجا تعریف شده از نوع virtual است؛ بدون پیاده سازی مفصل قسمت set آن و فراخوانی مستقیم RaisePropertyChanged کلاس پایه به صورت متداول. علت virtual تعریف کردن آن به امکان دخل و تصرف در نواحی get و set

این خاصیت توسط Interceptor ایی که در ادامه تعریف خواهیم کرد بر می‌گردد.

پیاده سازی NotifyPropertyInterceptor

```
using System;
using Castle.DynamicProxy;

namespace AOP01.Core
{
    public class NotifyPropertyInterceptor : IInterceptor
    {
        public void Intercept(IInvocation invocation)
        {
            // متد ست، ابتدا فراخوانی می‌شود و سپس کار اطلاع رسانی را انجام خواهیم داد
            invocation.Proceed();

            if (invocation.Method.Name.StartsWith("set_"))
            {
                var propertyName = invocation.Method.Name.Substring(4);
                raisePropertyChangedEvent(invocation, propertyName, invocation.TargetType);
            }
        }

        void raisePropertyChangedEvent(IInvocation invocation, string propertyName, Type type)
        {
            var methodInfo = type.GetMethod("RaisePropertyChanged");
            if (methodInfo == null)
            {
                if (type.BaseType != null)
                    raisePropertyChangedEvent(invocation, propertyName, type.BaseType);
            }
            else
            {
                methodInfo.Invoke(invocation.InvocationTarget, new object[] { propertyName });
            }
        }
    }
}
```

با اینترفیس IInterceptor در قسمت قبل آشنا شدیم.
در اینجا ابتدا اجازه خواهیم داد تا کار set به صورت معمول انجام شود. دو حالت get و set ممکن است رخ دهند. بنابراین در ادامه بررسی خواهیم کرد که اگر حالت set بود، آنگاه متد RaisePropertyChanged کلاس پایه BaseViewModel را یافته و به صورت پویا با propertyName صحیحی فراخوانی می‌کنیم.
به این ترتیب دیگر نیازی نخواهد بود تا به ازای تمام خواص مورد نیاز، کار فراخوانی دستی RaisePropertyChanged صورت گیرد.

اتصال Interceptor به سیستم

خوب! تا اینجای کار صرفا تعاریف اولیه تدارک دیده شده‌اند. در ادامه نیاز است تا DI و DynamicProxy را از وجود آن‌ها مطلع کنیم.
برای این منظور فایل App.xaml.cs را گشوده و در نقطه آغاز برنامه تنظیمات ذیل را اعمال نمائید:

```
using System.Linq;
using System.Windows;
using AOP01.Core;
using AOP01.Services;
using Castle.DynamicProxy;
using StructureMap;

namespace AOP01
{
    public partial class App
    {
        protected override void OnStartup(StartupEventArgs e)
        {
            base.OnStartup(e);
```

```
            ObjectFactory.Initialize(x =>
            {
                x.For<ITestService>().Use<TestService>();

                var dynamicProxy = new ProxyGenerator();
                x.For<BaseViewModel>().EnrichAllWith(vm =>
                {
                    var constructorArgs = vm.GetType()
                            .GetConstructors()
                            .FirstOrDefault()
                            .GetParameters()
                            .Select(p => ObjectFactory.GetInstance(p.ParameterType))
                            .ToArray();

                    return dynamicProxy.CreateClassProxy(
                                classToProxy: vm.GetType(),
                                constructorArguments: constructorArgs,
                                interceptors: new[] { new NotifyPropertyInterceptor() });
                });
            });
        }
    }
}
```

مطابق این تنظیمات، هرجایی که نیاز به نوعی از ITestService بود، از کلاس TestService استفاده خواهد شد.
همچنین در ادامه به DI مورد استفاده اعلام می‌کنیم که ViewModelهای ما دارای کلاس پایه BaseViewModel هستند. بنابراین هر زمانی که این نوع موارد وهله سازی شدند، آن‌ها را یافته و با پروکسی حاوی NotifyPropertyInterceptor مزین کن.
مثالی که در اینجا انتخاب شده، تقریبا مشکل‌ترین حالت ممکن است؛ چون به همراه تزریق خودکار وابستگی‌ها در سازنده کلاس ViewModel نیز می‌باشد. اگر ViewModelهای شما سازنده‌ای به این شکل ندارند، قسمت تشکیل constructorArgs را حذف کنید.

استفاده از ViewModel مزین شده با پروکسی در یک View

```
<Window x:Class="AOP01.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <TextBox Text="{Binding Text, Mode=TwoWay, UpdateSourceTrigger=PropertyChanged}" />
    </Grid>
</Window>
```

اگر فرض کنیم که پنجره اصلی برنامه مصرف کننده ViewModel فوق است، در code behind آن خواهیم داشت:

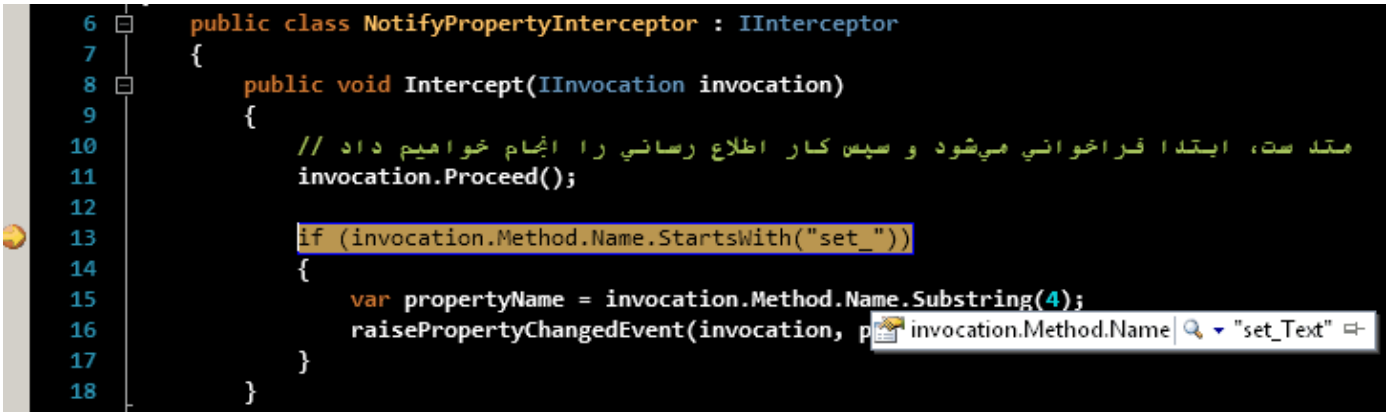
```
using AOP01.ViewModels;
using StructureMap;

namespace AOP01
{
    public partial class MainWindow
    {
        public MainWindow()
        {
            InitializeComponent();

            //علاوه بر تشکیل پروکسی
            //کار وهله سازی و تزریق وابستگی‌ها در سازنده را هم به صورت خودکار انجام می‌دهد
            var vm = ObjectFactory.GetInstance<TestViewModel>();
            this.DataContext = vm;
        }
    }
}
```

به این ترتیب یک ViewModel محصور شده توسط DynamicProxy مزین با NotifyPropertyInterceptor به DataContext ارسال می‌گردد.

اکنون اگر برنامه را اجرا کنیم، مشاهده خواهیم کرد که با وارد کردن مقداری در TextBox برنامه، NotifyPropertyInterceptor مورد استفاده قرار می‌گیرد:



دریافت مثال کامل این قسمت

AOP01.zip