عنوان: AOP با استفاده از Microsoft Unity
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۸/۰۶ ۱۳:۵
آدرس: www.dotnettips.info
برچسب‌ها: UnityContainer, AOP

چند روز پیش فرصتی پیش آمد تا بتوانم مروری بر مطلب منتشر شده درباره AOP داشته باشم. به حق مطلب مورد نظر، بسیار خوب و مناسب شرح داده شده بود و همانند سایر مقالات جناب نصیری چیزی کم نداشت. اما امروز قصد پیاده سازی یک مثال AOP، با استفاده از Microsoft Unity Application Block را به عنوان IOC Container دارم. اگر شما هم، مانند من از UnityContainer به عنوان IOC Container در پروژه‌های خود استفاده می‌کنید نگران نباشید. این کتابخانه به خوبی از مباحث Interception پشتیبانی می‌کند. در ادامه طی یک مقاله این مورد را با هم بررسی می‌کنیم.
برای دوستانی که با AOP آشنایی ندارند پیشنهاد می‌شود ابتدا مطلب مورد نظر را یک بار مطالعه نمایند.
برای شروع یک پروژه در VS.Net بسازید و ارجاع به اسمبلی‌های زیر را در پروژه فراموش نکنید:

«Microsoft.Practices.EnterpriseLibrary.Common
«Microsoft.Practices.Unity
«Microsoft.Practices.Unity.Configuration
«Microsoft.Practices.Unity.Interception
«Microsoft.Practices.Unity.Interception.Configuration

یک اینترفیس به نام IMyOperation بسازید:

```
public interface IMyOperation
    {
        void DoIt();
    }
```

کلاسی می‌سازیم که اینترفیس بالا را پیاده سازی نماید:

```
public void DoIt()
  {
     Console.WriteLine( "this is main block of code" );
  }
```

قصد داریم با استفاده از AOP یک سری کد مورد نظر خود(در این مثال کد لاگ کردن عملیات در یک فایل مد نظر است) را به کدهای متدهای مورد نظر تزریق کنیم. یعنی با فراخوانی این متد کدهای لاگ عملیات در یک فایل ذخیره شود بدون تکرار یا فراخوانی دستی متد لاگ.
ابتدا یک کلاس برای لاگ عملیات می‌سازیم:

```
public class Logger
    {
        const string path = @"D:\Log.txt";

        public static void WriteToFile( string methodName )
        {
            object lockObject = new object();
            if ( !File.Exists( path ) )
            {
                File.Create( path );
            }
            lock ( lockObject )
            {
                using ( TextWriter writer = new StreamWriter( path , true ) )
                {
                    writer.WriteLine( string.Format( "{0} at {1}" , methodName , DateTime.Now ) );
                }
            }
        }
    }
```

حال نیاز به یک Handler برای مدیریت فراخوانی کدهای تزریق شده داریم. برای این کار یک کلاس می‌سازیم که اینترفیس ICallHandler را پیاده سازی نماید.

```
public class LogHandler : ICallHandler
    {
        public IMethodReturn Invoke( IMethodInvocation input , GetNextHandlerDelegate getNext )
        {
            Logger.WriteToFile( input.MethodBase.Name );

            var methodReturn = getNext()( input , getNext );

            return methodReturn;
        }

        public int Order { get; set; }
    }
```

کلاس بالا یک متد به نام Invoke دارد که فراخوانی متدهای مورد نظر برای تزریق کد را در دست خواهد گرفت. در این متد ابتدا عملیات لاگ در فایل مورد نظر ثبت می‌شود(با استفاده از Logger.WriteToFile). سپس با استفاده از getNext که از نوع GetNextHandlerDelegate است، اجرا را به کدهای اصلی برنامه منتقل می‌کنیم.

```
var methodReturn = getNext()( input , getNext );
```

برای مدیریت بهتر عملیات لاگ یک Attribute می‌سازیم که فقط متد هایی که نیاز به لاگ کردن دارند را مشخص کنیم. به صورت زیر:

```
public class LogAttribute : HandlerAttribute
    {
        public override ICallHandler CreateHandler( Microsoft.Practices.Unity.IUnityContainer container
)
        {
            return new LogHandler();
        }
    }
```

فقط دقت داشته باشید که کلاس مورد نظر به جای ارث بری از کلاس Attribute باید از کلاس HandlerAttribute که در فضای نام Microsoft.Practices.Unity.InterceptionExtension تعبیه شده است ارث ببرد(خود این کلاس از کلاس Attribute ارث برده است). کافیست در متد CreateHandler آن که Override شده است یک نمونه از کلاس LogHandler را برگشت دهیم.
برای آماده سازی Ms Unity جهت عملیات Interception باید کدهای زیر در ابتدا برنامه قرار داده شود:

```
var  unityContainer = new UnityContainer();

 unityContainer.AddNewExtension<Interception>();

  unityContainer.Configure<Interception>().SetDefaultInterceptorFor<IMyOperation>( new
InterfaceInterceptor() );

  unityContainer.RegisterType<IMyOperation, MyOperation>();
```

توضیح چند مطلب:
بعد از نمونه سازی از کلاس UnityContainer باید Interception به عنوان یک Extension به این Container اضافه شود. سپس با استفاده از متد Configure برای اینترفیس IMyOperation یک Interceptor پیش فرض تعیین می‌کنیم. در پایان هم به وسیله متد RegisterType کلاس MyOperation به اینترفیس IMyOperation رجیستر می‌شود. از این پس هر گاه درخواستی برای اینترفیس IMyOperation از unityContainer شود یک نمونه از کلاس MyOperation در اختیار خواهیم داشت.
به عنوان نکته آخر متد DoIt در اینترفیس بالا باید دارای LogAttribute باشد تا عملیات مزین سازی با کدهای لاگ به درستی انجام شود.

یک نکته تکمیلی:

در هنگام مزین سازی متد set خاصیت ها، به دلیل اینکه اینترفیسی برای این کار وجود ندارد باید مستقیما عملیات AOP به خود کلاس اعمال شود. برای این کار باید به صورت زیر عمل نمود:

```
var container = new UnityContainer();
container.RegisterType<Book , Book>();

container.AddNewExtension<Interception>();

 var policy = container.Configure<Interception>().SetDefaultInterceptorFor<Book>( new
VirtualMethodInterceptor() ).AddPolicy( "MyPolicy" );

  policy.AddMatchingRule( new PropertyMatchingRule( "*" , PropertyMatchingOption.Set ) );
  policy.AddCallHandler<Handler.NotifyChangedHandler>();
```

همان طور که مشاهده می‌کنید عملیات Interception مستقیما برای کلاس پیکر بندی می‌شود و به جای InterfaceInterceptor از VirtualMethodInterceptor برای تزریق کد به بدنه متدها استفاده شده است. در پایان نیز با تعریف یک Policy می‌توانیم به راحتی(با استفاده از "*") متد Set تمام خواص کلاس را به NotifyChangedHandler مزین نماییم.

سورس کامل مثال بالا