عنوان: **معرفی Aspect oriented programming**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۱/۱۹ ۱۰:۳۵
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, AOP, C#, Architecture

AOP یا Aspect oriented programming 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.

مثالی از کدهای تکراری پراکنده در برنامه

به برنامه ذیل و قسمت‌های مختلف ثبت وقایع آن دقت کنید:

```
using System;

namespace AOP00
{
    class Program
    {
        static void Main(string[] args)
        {
            Log.Debug("Program has started.");
            //.....
            try
            {

            }
            catch (Exception ex)
            {
                Log.Error(ex);
                throw;
            }
            finally
            {
                //.....
                Log.Debug("Program has ended.");
            }
        }
    }
}
```

همانطور که ملاحظه می‌کنید، حجم بالایی از کدهای تکراری ثبت وقایع، تنها در قسمت کوچکی از برنامه تدارک دیده شده‌اند. این مساله نقض اصل DRY یا Don't repeat yourself است. کاری که برای رفع این مشکل قرار است انجام دهیم، استفاده از AOP و کپسوله سازی اعمال تکراری و سپس اتصال آن به قسمت‌های مختلف برنامه است.

معرفی Aspects و مزایای استفاده از آن‌ها

همانطور که عنوان شد اولین گام در AOP، کپسوله سازی کدهای تکراری است که اصطلاحا یک Aspect را تشکیل می‌دهند. بنابراین هر Aspect صرفا یک محصور کننده قابلیتی خاص و تکراری در برنامه است. این Aspect باید اصل SRP یا Single responsibility principle (تک مسئولیتی) را رعایت کند. برای اتصال یک Aspect به قطعه‌های مختلف کدهای برنامه از الگوی طراحی تزئین کننده یا Decorator pattern استفاده می‌شود. به این ترتیب که این Aspect خاص قرار است قسمتی از کدهای برنامه را تزئین کند. همچنین در این حالت، open closed principle نیز بهتر رعایت خواهد گردید. از این جهت که کدهای تکراری برنامه، به Aspects منتقل شده‌اند و دیگر نیازی نیست برای تغییر آن‌ها، کدهای قسمت‌های مختلف را تغییر داد (کدهای برنامه باز خواهند بود برای

توسعه و بسته برای تغییر). بنابراین با استفاده از Aspects، به یک طراحی شیءگرای بهتر نیز دست خواهیم یافت.

مراحل اجرای یک Aspect

هر Aspect برای تزئین یا اتصال به قسمت‌های مختلف برنامه، یک طول عمر کاری مشخص را طی می‌کند:

الف) مرحله OnStart

```
public User GetUserById(int id)
        {
            if (Cache.ExistsFor(id))
            {
                return Cache[id];
            }
            else
            {
                var user = LoadFromDb(id);
                Cache.AddFor("User", id, user);
                return user;
            }
        }
```

مرحله اول اجرای یک Aspect، در آغاز کار قطعه‌ای است که قرار است آن‌را مزین کند. بنابراین بلافاصله قبل از اجرای کدی، برای مثال در یک متد، قادر خواهیم بود تا قطعه کد موجود در Aspect ایی را فراخوانی و اجرا کنیم.
برای مثال در متد GetUserById، پیش از اینکه کار به مراجعه به بانک اطلاعاتی برسد، ابتدا وضعیت کش سیستم بررسی می‌شود. بنابراین در این مثال می‌توان قسمت بررسی کش را به یک Aspect مجزا منتقل ساخته و در صورتیکه اطلاعاتی موجود بود، بازگشت داده شود؛ در غیر اینصورت مجوز اجرای ادامه کدها صادر گردد.

ب) مرحله OnSuccess

مرحله OnSuccess زمانی اجرا می‌شود که اجرای یک متد بدون بروز استثنایی خاتمه یافته است.

ج) مرحله OnExit

مرحله OnExit همانند مرحله OnSuccess است؛ با این تفاوت که مرحله OnSuccess در صورت بروز استثنایی در کدها اجرا نخواهد شد اما مرحله OnExit همواره در پایان کار یک متد فراخوانی می‌گردد.

د) مرحله OnError

مرحله OnError در طول عمر یک Aspect، در زمان بروز استثنایی رخ می‌دهد. برای مثال به این ترتیب می‌توان قسمت ثبت وقایع بروز استثناهای سیستم را کلا به یک Aspect مشخص انتقال داده و حجم کدهای تکراری را به این ترتیب به شدت کاهش داد.

انواع مختلف AOP

تا اینجا شاید این سؤال برای شما پیش آمده باشد که خوب! جالب است! اما چطور می‌خواهید در مراحلی که یاد شد، دخالت کرده و قطعه کدی را تزریق کنید؟
در AOP دو روش متداول کلی برای انجام اعمال تزریق کد وجود دارند:

1) استفاده از Interceptors

به کمک Interceptors، فرآیند فراخوانی متدها و خواص یک کلاس، تحت کنترل و نظارت قرار خواهند گرفت. برای انجام این امر، عموما از IOC Containers استفاده می‌شود (Inversion of control). احتمالا تا کنون از این کتابخانه‌ها تنها برای تزریق وابستگی‌های برنامه خود کمک گرفته‌اید و از سایر توانمندی‌های آن‌ها آنچنان استفاده‌ای نکرده‌اید. در این حالت، زمانیکه یک IOC Container کار وهله سازی کلاس خاصی را انجام می‌دهد، در همین حین می‌تواند مراحل یاد شده شروع، پایان و خطای متدها یا فراخوانی‌های خواص را نیز تحت نظر قرار داده و به این ترتیب مصرف کننده امکان تزریق کدهایی را در این مکان‌ها خواهد یافت.
مزیت مهم استفاده از Interceptors، عدم نیاز به کامپایل و یا تغییر ثانویه اسمبلی‌های موجود برای تغییری در کدهای آن‌ها است (برای تزریق نواحی تحت کنترل قرار دادن اعمال) و تمام کارها به صورت خودکار در زمان اجرای برنامه مدیریت می‌گردند.

2) بهره گیری از فناوری IL Code Weaving

در فناوی IL Code Weaving، ابتدا برنامه و ماژول‌های آن به نحو متداولی کامپایل و تبدیل به dll یا exe خواهند شد. سپس این dllها و فایل‌های اجرایی به پردازشگر ثانویه یک فریم ورک AOP برای تغییر و تزریق کدها سپرده خواهند شد. برای مثال در این حالت، کدهای سطح پایین IL مرتبط با مراحل مختلف اجرای یک Aspect، تولید و به اسمبلی‌های نهایی برنامه تزریق می‌شوند. اکنون به dll یا فایل اجرایی جدیدی خواهیم رسید که علاوه بر کدهای اصلی برنامه، حاوی کدهای تزریق شده تمام Aspects تعریف شده نیز هستند.

عنوان: آشنایی با AOP Interceptors
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۱/۱۹ ۱۷:۱
آدرس: www.dotnettips.info
گروه‌ها: C#, Design patterns, AOP, Architecture

در حین استفاده از Interceptors، کار مداخله و تحت نظر قرار دادن قسمت‌های مختلف کدها، توسط کامپوننت‌های خارجی صورت خواهد گرفت. این کامپوننت‌های خارجی، به صورت پویا، تزئین کننده‌هایی را جهت محصور سازی قسمت‌های مختلف کدهای شما تولید می‌کنند. این‌ها، بسته به توانایی‌هایی که دارند، در زمان اجرا و یا حتی در زمان کامپایل نیز قابل تنظیم می‌باشند.

ابزارهایی جهت تولید AOP Interceptors

متداول‌ترین کامپوننت‌های خارجی که جهت تولید AOP Interceptors مورد استفاده قرار می‌گیرند، همان IOC Containers معروف هستند مانند MS Unity، Ninject، StructureMap و غیره.
سایر ابزارهای تولید AOP Interceptors، از روش تولید Dynamic proxies بهره می‌گیرند. به این ترتیب مزین کننده‌هایی پویا، در زمان اجرا، کدهای شما را محصور خواهند کرد. (نمونه‌ای از آن‌را شاید در حین کار با ORMهای مختلف دیده باشید).

نگاهی به فرآیند Interception

زمانیکه از یک IOC Container در کدهای خود استفاده می‌کنید، مراحلی چند رخ خواهند داد:
الف) کد فراخوان، از IOC Container، یک شیء مشخص را درخواست می‌کند. عموما اینکار با درخواست یک اینترفیس صورت می‌گیرد؛ هرچند محدودیتی نیز وجود نداشته و امکان درخواست یک کلاس از نوعی مشخص نیز وجود دارد.
ب) در ادامه IOC Container به لیست اشیاء قابل ارائه توسط خود نگاه کرده و در صورت وجود، وهله سازی شیء درخواست شده را انجام و نهایتا شیء مطلوب را بازگشت خواهد داد.
ج) سپس، کد فراخوان، وهله دریافتی را مورد پردازش قرار داده و شروع به استفاده از متدها و خواص آن خواهد نمود.

اکنون با اضافه کردن Interception به این پروسه، چند مرحله دیگر نیز در این بین به آن اضافه خواهند شد:
الف) در اینجا نیز در ابتدا کد فراخوان، درخواست وهله‌ای را بر اساس اینترفیسی خاص به IOC Container ارائه می‌دهد.
ب) IOC Container نیز سعی در وهله سازی درخواست رسیده بر اساس تنظیمات اولیه خود می‌کند.
ج) اما در این حالت IOC Container تشخیص می‌دهد، نوعی که باید بازگشت دهد، علاوه بر وهله سازی، نیاز به مزین سازی توسط Aspects و پیاده سازی Interceptors را نیز دارد. بنابراین نوع مورد انتظار را در صورت وجود، به یک Dynamic Proxy، بجای بازگشت مستقیم به فراخوان ارائه می‌دهد.
د) در ادامه Dynamic Proxy، نوع مورد انتظار را توسط Interceptors محصور کرده و به فراخوان بازگشت می‌دهد.
ه) اکنون فراخوان، در حین استفاده از امکانات شیء وهله سازی شده، به صورت خودکار مراحل مختلف اجرای یک Aspect را که در قسمت قبل بررسی شدند، سبب خواهد شد.

نحوه ایجاد Interceptors

برای ایجاد یک Interceptor دو مرحله باید انجام شود:
الف) پیاده سازی یک اینترفیس
ب) اتصال آن به کدهای اصلی برنامه

در ادامه قصد داریم از یک IOC Container معروف به نام StructureMap در یک برنامه کنسول استفاده کنیم. برای دریافت آن نیاز است دستور پاورشل ذیل را در کنسول نوگت ویژوال استودیو فراخوانی کنید:

```
PM> Install-Package structuremap
```

پس از آن یک برنامه کنسول جدید را ایجاد کنید. (هدف از استفاده از این نوع پروژه خاص، توضیح جزئیات یک فناوری، بدون

درگیر شدن با لایه UI است)
البته باید دقت داشت که برای استفاده از StructureMap نیاز است به خواص پروژه مراجعه و سپس حالت Client profile را به Full profile تغییر داد تا برنامه قابل کامپایل باشد.

```
using System;
using StructureMap;

namespace AOP00
{
    public interface IMyType
    {
        void DoSomething(string data, int i);
    }

    public class MyType : IMyType
    {
        public void DoSomething(string data, int i)
        {
            Console.WriteLine("DoSomething({0}, {1});", data, i);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            ObjectFactory.Initialize(x =>
            {
                x.For<IMyType>().Use<MyType>();
            });

            var myType = ObjectFactory.GetInstance<IMyType>();
            myType.DoSomething("Test", 1);
        }
    }
}
```

اکنون کدهای این برنامه را به نحو فوق تغییر دهید.
در اینجا یک اینترفیس نمونه و پیاده سازی آن‌را ملاحظه می‌کنید. همچنین نحوه آغاز تنظیمات StructureMap و نحوه دریافت یک وهله متناظر با IMyType نیز بیان شده‌اند.
نکته‌ی مهمی که در اینجا باید به آن دقت داشت، وضعیت شیء myType حین فراخوانی متد myType.DoSomething است. شیء myType در اینجا، دقیقا یک وهله‌ی متداول از کلاس myType است و هیچگونه دخل و تصرفی در نحوه اجرای آن صورت نگرفته است.
خوب! تا اینجای کار را احتمالا پیشتر نیز دیده بودید. در ادامه قصد داریم یک Interceptor را طراحی و مراحل چهارگانه اجرای یک Aspect را در اینجا بررسی کنیم.

در ادامه نیاز خواهیم داشت تا یک Dynamic proxy را نیز مورد استفاده قرار دهیم؛ از این جهت که StructureMap تنها دارای Interceptorهای وهله سازی اطلاعات است و نه Method Interceptor. برای دسترسی به Method Interceptors نیاز به یک Dynamic proxy نیز می‌باشد. در اینجا از Castle.Core استفاده خواهیم کرد:

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

برای دریافت آن تنها کافی است دستور پاور شل فوق را در خط فرمان کنسول پاورشل نوگت در VS.NET اجرا کنید.
سپس کلاس ذیل را به پروژه جاری اضافه کنید:

```
using System;
using Castle.DynamicProxy;

namespace AOP00
{
    public class LoggingInterceptor : IInterceptor
    {
        public void Intercept(IInvocation invocation)
        {
```

```
            try
            {
                Console.WriteLine("Logging On Start.");

                invocation.Proceed(); //فراخوانی متد اصلی در اینجا صورت می‌گیرد

                Console.WriteLine("Logging On Success.");
            }
            catch (Exception ex)
            {
                Console.WriteLine("Logging On Error.");
                throw;
            }
            finally
            {
                Console.WriteLine("Logging On Exit.");
            }
        }
    }
}
```

در کلاس فوق کار Method Interception توسط امکانات Castle.Core انجام شده است. این کلاس باید اینترفیس IInterceptor را پیاده سازی کند. در این متد سطر invocation.Proceed دقیقا معادل فراخوانی متد مورد نظر است. مراحل چهارگانه شروع، پایان، خطا و موفقیت نیز توسط try/catch/finally پیاده سازی شده‌اند.

اکنون برای معرفی این کلاس به برنامه کافی است سطرهای ذیل را اندکی ویرایش کنیم:

```
static void Main(string[] args)
        {
            ObjectFactory.Initialize(x =>
            {
                var dynamicProxy = new ProxyGenerator();
                x.For<IMyType>().Use<MyType>();
                x.For<IMyType>().EnrichAllWith(myTypeInterface =>
dynamicProxy.CreateInterfaceProxyWithTarget(myTypeInterface, new LoggingInterceptor()));
            });

            var myType = ObjectFactory.GetInstance<IMyType>();
            myType.DoSomething("Test", 1);
        }
```

در اینجا تنها سطر EnrichAllWith آن جدید است. ابتدا یک پروکسی پویا تولید شده است. سپس این پروکسی پویا کار دخالت و تحت نظر قرار دادن اجرای متدهای اینترفیس IMyType را عهده دار خواهد شد.
برای مثال اکنون با فراخوانی متد myType.DoSomething، ابتدا کنترل برنامه به پروکسی پویای تشکیل شده توسط Castle.Core منتقل می‌شود. در اینجا هنوز هم متد DoSomething فراخوانی نشده است. ابتدا وارد بدنه متد public void Intercept خواهیم شد. سپس سطر invocation.Proceed، فراخوانی واقعی متد DoSomething اصلی را انجام می‌دهد. در ادامه باز هم فرصت داریم تا مراحل موفقیت، خطا یا خروج را لاگ کنیم.
تنها زمانیکه کار متد public void Intercept به پایان می‌رسد، سطر پس از فراخوانی متد myType.DoSomething اجرا خواهد شد.
در این حالت اگر برنامه را اجرا کنیم، چنین خروجی را نمایش می‌دهد:

```
Logging On Start.
DoSomething(Test, 1);
Logging On Success.
Logging On Exit.
```

بنابراین در اینجا نحوه دخالت و تحت نظر قرار دادن اجرای متدهای یک کلاس عمومی خاص را ملاحظه می‌کنید. برای اینکه کنترل کامل را در دست بگیریم، کلاس پروکسی پویا وارد عمل شده و اینجا است که این کلاس پروکسی تصمیم می‌گیرد چه زمانی باید فراخوانی واقعی متد مورد نظر انجام شود.
برای اینکه فراخوانی قسمت On Error را نیز ملاحظه کنید، یک استثنای عمدی را در متد DoSomething قرار داده و مجددا برنامه را اجرا کنید.

نظرات خوانندگان

نویسنده: علی ملکی
تاریخ: ۱۳۹۲/۱۰/۰۹ ۱۱:۷

با سلام
در صورتی که بخواهیم یک Interceptor فقط برای لایه سرویس داشته باشیم چطور میتونیم هنگام رجیستر کردن یکباره (بدون نوشتن تک تک تایپ ها) این اینترسپتور(EnrichAllWith) رو اضافه کنیم.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۰/۰۹ ۱۴:۳۶

باید از قابلیت scan در StructureMap استفاده کنید:

```
ObjectFactory.Initialize(x =>
            {
                var dynamicProxy = new ProxyGenerator();
                x.Scan(scanner =>
                    {
                        scanner.AssemblyContainingType<IMyType>(); // نحوه یافتن اسمبلی لایه سرویس

                        // Connect `IName` interface to 'Name' class automatically
                        scanner.WithDefaultConventions()
                               .OnAddedPluginTypes(plugin => plugin.EnrichWith(target =>

dynamicProxy.CreateInterfaceProxyWithTargetInterface(target.GetType().GetInterfaces().First(),
                                                                    target.GetType().GetInterfaces(),
target,
                                                                    new LoggingInterceptor())
                                                    ));
                    });
            });
```

- در این حالت AssemblyContainingType مشخص می‌کند که کدام اسمبلی باید اسکن شود.
- WithDefaultConventions یعنی هرجایی IName داشتیم را به صورت خودکار به Name متصل کن. (روال پیش فرض سیم کشی اینترفیس‌ها و کلاس‌ها برای وهله سازی)
- OnAddedPluginTypes یک Callback هست که زمان انجام اولیه تنظیمات به ازای هر type یافت شده فراخوانی می‌شود. در اینجا می‌شود با استفاده از EnrichWith و ProxyGenerator کار اتصال کلاس Interceptor را انجام داد.

نویسنده: رضا شش
تاریخ: ۱۳۹۲/۱۰/۱۱ ۱۶:۲۹

برای اینکه استثای عمدی تولید کنم من از مثال ساده زیر استفاده کردم اما استثنا رخ نمی‌دهد. دلیل آن چیست؟
با تشکر Test_ExceptionAspect.zip

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۰/۱۱ ۱۹:۶

قسمت EnrichAllWith را حذف کنید. بعد برنامه را اجرا کنید. باز هم اجرا می‌شود و استثنایی صادر نمی‌شود. چرا؟ چون اجرای کد آن معادل است با:

```
double d = 0;
Console.WriteLine(1 / d); // compiles, runs, results in: Infinity
```

مقدار infinity برای نوع double تعریف شده اما برای نوع int خیر؛ اینطوری طراحی شده .

نویسنده: رضا شش
تاریخ: ۱۳۹۲/۱۰/۱۵ ۱۰:۳

برای حالتهایی مثل حالتی که قرار است بر روی چند صد هزار رکورد ، محاسباتی صورت گیرد و نتیجه در دیتابیس ذخیره شود اگر بخواهیم یکسری کارها مثل لاگ و استثنا و ... را به درون اینترسپتر بکشانیم و از پروکسی استفاده کنیم آیا کارایی را پایین نمی‌آورد؟
در همین حالت اگر انتیتی‌های متفاوتی داشته باشیم و همزمان از انتیتی‌های مختلف نیاز به وهله سازی باشد چطور؟
با تشکر

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۰/۱۵ ۱۰:۸

وجود یک Interceptor تداخلی در روند کاری جزئیات متد شما ندارد. جائیکه فراخوانی متد invocation.Proceed انجام می‌شود، روند انجام آن مستقل است از وجود Interceptor و فقط پیش و پس از آن یا استثنای حاصل تحت نظر قرار می‌گیرند.

نویسنده: رضا شش
تاریخ: ۱۳۹۲/۱۰/۱۵ ۱۴:۲۳

منظورم این بود که برای حالتی که از امکاناتی مثل Castle.Core استفاده می‌کنیم یک پروکسی از کلاس اصلی ما تولید می‌کند و با توجه به اینترسپتر در زمان اجرا این کلاس تزیین شده را اجرا می‌کند. مگر برای هر وهله از کلاس اصلی ما این اتفاق رخ نمی‌دهد؟ اگر چنین است پس پروکسی‌های زیادی با توجه به کلاس اصلی و اینترسپتورهای مختلفی که تعریف کرده ایم ایجاد می‌شود.
می‌خواستم ببینم این عملیات کارایی را پایین نمی‌آورد؟.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۰/۱۵ ۱۴:۴۱

- سؤال شما این بود که در کلاس اصلی من، در متدی داخل آن، با چند صدهزار رکورد کار انجام می‌شود. پاسخ این است که اصلا این پروکسی ایجاد شده ربطی به داخل متد شما ندارد. کاری به وهله سازی‌های انجام شده داخل آن نیز ندارد.
invocation.Proceed یعنی این متد رو اجرا کن؛ نه اینکه هر وهله‌ای که داخل آن متد قرار می‌گیرد را نیز با پروکسی مزین کن.
- تمام ORMها برای پیاده سازی مباحث Lazy loading یک شیء پروکسی را از شیء اصلی شما ایجاد می‌کنند. نمونه‌اش را شاید با EF Code first با نام‌های خودکاری مانند ClassName_00394CF1F92740F13E3 دیده باشید؛ NHibernate هم یک زمانی از همین Castle.Core برای تدارک پروکسی‌های اشیاء استفاده می‌کرد. سربار آن در حین ایجاد چندین هزار وهله از یک شیء، در حد همان کار با ORMهایی است که هر روزه از آن‌ها استفاده می‌کنید (اگر می‌خواهید یک حسی از این قضیه داشته باشید).

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۱۳ ۱۸:۴۱

به روز رسانی
اگر از StructureMap نگارش 3 استفاده کنید، کلیه متدهای XYZ **Enrich** به XYZ **Decorate** تبدیل شده‌اند.

نویسنده: داستان
تاریخ: ۱۳۹۳/۰۱/۳۰ ۱۲:۳۵

ممنون بابت این دوره زیبا؛
ایا روشی توکار برای بازگشت مقدار، از یک Interceptor به متد اجراشونده هست؟
بعنوان مثال رشته ای که Log می‌شود رو بعنوان مقدار بازگشتی در متد اجرا شونده دریافت کنیم؟
باتشکر

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۱/۳۰ ۱۳:۴

- یکی از اهداف مهم AOP این است که به صورت لایه‌ای نامریی عمل کند و هر زمان که نیاز باشد، بتوان بدون کوچکترین تغییری در کدهای اصلی برنامه، کل منطق آن‌را حذف، یا با نمونه‌ای دیگر جایگزین کرد. بنابراین دریافت یک مقدار از Interceptor داخل متدی در برنامه، نقض کننده‌ی فلسفه‌ی وجودی این عملیات است.
- اما توسط پارامتر IInvocation و مقداری Reflection، دسترسی کاملی به اطلاعات متد فراخوان هست و در اینجا می‌توان در صورت نیاز، پارامتر و مقداری را نیز به آن ارسال کرد.
- در ASP.NET MVC، مفهوم فیلترها دقیقا پیاده سازی کننده‌ی Interceptorهای AOP هستند. در اینجا نیز مستقیما اطلاعاتی به فراخوان، در صورت نیاز بازگشت داده نمی‌شود. اما Context جاری در اختیار Interceptor و فیلتر هست. به این ترتیب Interceptor فرصت خواهد داشت به این Context مشترک، اطلاعاتی را اضافه کند یا تغییر دهد. مثلا به لیست خطاهای آن یک خطای اعتبارسنجی جدید را اضافه کند.

نویسنده: داستان
تاریخ: ۱۳۹۳/۰۲/۱۴ ۱۲:۲۸

بله،ممنون
جناب نصیری میشه یک مثال در ASP.NET MVC بزنید؟
مثلا پیاده سازی LoggerInterceptor برای اکشن هایی ک یک ActionResult برمیگردانند و درصورت بروز استثناء پیغامی مرتبط ب کاربر نمایش داده شود...
پیاپیش ممنون

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۲/۱۴ ۱۲:۳۴

پیشنیازهای آن در سایت مطرح شده‌اند. ابتدا نیاز دارید تا تزریق وابستگی‌ها را در ASP.MVC پیاده سازی کنید . پس از اینکه کنترل وهله سازی یک کنترلر تماما در اختیار IoC Container قرار گرفت، سایر مباحث آن با مطلب جاری تفاوتی نمی‌کند و یکی است.
این یک راه حل است. راه دیگر آن استفاده از امکانات توکار خود ASP.NET MVC است و استفاده از فیلترهای آن که در حقیقت نوعی Interceptor توکار و یکپارچه هستند.

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:

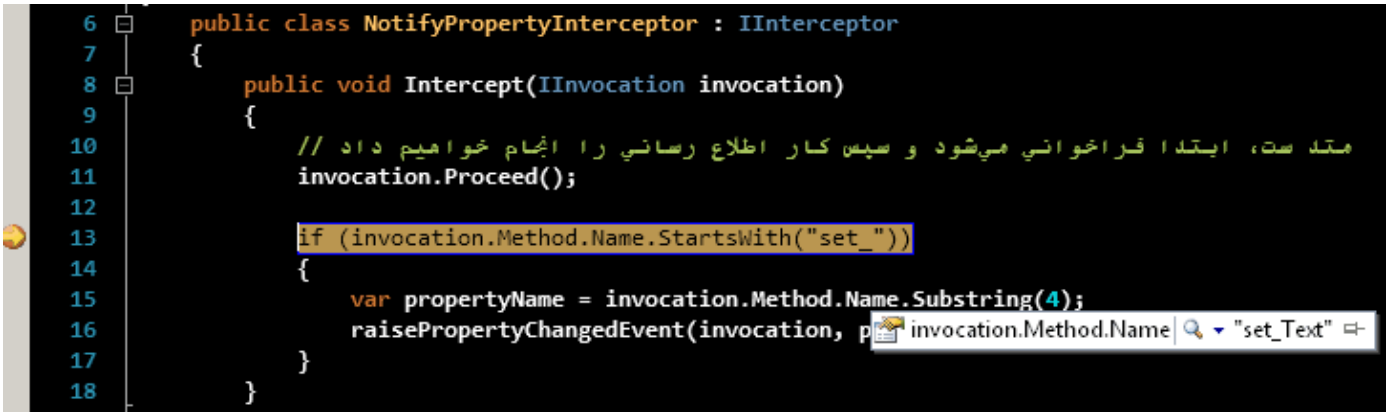
```
using AOP01.ViewModels;
using StructureMap;

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

عنوان: **استفاده از AOP Interceptors برای حذف کدهای تکراری کش کردن اطلاعات در لایه سرویس برنامه**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۱/۲۰ ۱۹:۳۶
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, AOP, C#, Architecture

اکثر برنامه‌های ما دارای قابلیت‌هایی هستند که با موضوعاتی مانند امنیت، کش کردن اطلاعات، مدیریت استثناها، ثبت وقایع و غیره گره خورده‌اند. به هر یک از این موضوعات یک Aspect یا cross-cutting concern نیز گفته می‌شود.
در این قسمت قصد داریم اطلاعات بازگشتی از لایه سرویس برنامه را کش کنیم؛ اما نمی‌خواهیم مدام کدهای مرتبط با کش کردن اطلاعات را در مکان‌های مختلف لایه سرویس پراکنده کنیم. می‌خواهیم یک ویژگی یا Attribute سفارشی را تهیه کرده (مثلا به نام CacheMethod) و به متد یا متدهایی خاص اعمال کنیم. سپس برنامه، در زمان اجرا، بر اساس این ویژگی‌ها، خروجی‌های متدهای تزئین شده با ویژگی CacheMethod را کش کند.
در اینجا نیز از ترکیب StructureMap و DynamicProxy پروژه Castle، برای رسیدن به این مقصود استفاده خواهیم کرد. به کمک StructureMap می‌توان در زمان وهله سازی کلاس‌ها، آن‌ها را به کمک متدی به نام EnrichWith توسط یک محصور کننده دلخواه، مزین یا غنی سازی کرد. این مزین کننده را جهت دخالت در فراخوانی‌های متدها، یک DynamicProxy درنظر می‌گیریم. با پیاده سازی اینترفیس IInterceptor کتابخانه DynamicProxy مورد استفاده و تحت کنترل قرار دادن نحوه و زمان فراخوانی متدهای لایه سرویس، یکی از کارهایی را که می‌توان انجام داد، کش کردن نتایج است که در ادامه به جزئیات آن خواهیم پرداخت.

پیشنیازها

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

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

همچنین ارجاعی را به اسمبلی استاندارد System.Web.dll نیز اضافه نمائید.
از این جهت که از HttpRuntime.Cache قصد داریم استفاده کنیم. HttpRuntime.Cache در برنامه‌های کنسول نیز کار می‌کند. در این حالت از حافظه سیستم استفاده خواهد کرد و در پروژه‌های وب از کش IIS بهره می‌برد.

ویژگی CacheMethod مورد استفاده

```
using System;

namespace AOP02.Core
{
    [AttributeUsage(AttributeTargets.Method)]
    public class CacheMethodAttribute : Attribute
    {
        public CacheMethodAttribute()
        {
            // مقدار پیش فرض
            SecondsToCache = 10;
        }

        public double SecondsToCache { get; set; }
    }
}
```

همانطور که عنوان شد، قصد داریم متدهای مورد نظر را توسط یک ویژگی سفارشی، مزین سازیم تا تنها این موارد توسط AOP Interceptor مورد استفاده پردازش شوند.
در ویژگی CacheMethod، خاصیت SecondsToCache بیانگر مدت زمان کش شدن نتیجه متد خواهد بود.

ساختار لایه سرویس برنامه

```
using System;
using System.Threading;
using AOP02.Core;

namespace AOP02.Services
{
    public interface IMyService
    {
        string GetLongRunningResult(string input);
    }

    public class MyService : IMyService
    {
        [CacheMethod(SecondsToCache = 60)]
        public string GetLongRunningResult(string input)
        {
            Thread.Sleep(5000); // simulate a long running process
            return string.Format("Result of '{0}' returned at {1}", input, DateTime.Now);
        }
    }
}
```

اینترفیس IMyService و پیاده سازی نمونه آن‌را در اینجا مشاهده می‌کنید. از این لایه در برنامه استفاده شده و قصد داریم نتیجه بازگشت داده شده توسط متدی زمانبر را در اینجا توسط AOP Interceptors کش کنیم.

تدارک یک CacheInterceptor

```
using System;
using System.Web;
using Castle.DynamicProxy;

namespace AOP02.Core
{
    public class CacheInterceptor : IInterceptor
    {
        private static object lockObject = new object();

        public void Intercept(IInvocation invocation)
        {
            cacheMethod(invocation);
        }

        private static void cacheMethod(IInvocation invocation)
        {
            var cacheMethodAttribute = getCacheMethodAttribute(invocation);
            if (cacheMethodAttribute == null)
            {
                // متد جاری توسط ویژگی کش شدن مزین نشده است
                // بنابراین آن‌را اجرا کرده و کار را خاتمه می‌دهیم
                invocation.Proceed();
                return;
            }

            // دراینجا مدت زمان کش شدن متد از ویژگی کش دریافت می‌شود
            var cacheDuration = ((CacheMethodAttribute)cacheMethodAttribute).SecondsToCache;

            // برای ذخیره سازی اطلاعات در کش نیاز است یک کلید منحصربفرد را
            //  بر اساس نام متد و پارامترهای ارسالی به آن تهیه کنیم
            var cacheKey = getCacheKey(invocation);

            var cache = HttpRuntime.Cache;
            var cachedResult = cache.Get(cacheKey);


            if (cachedResult != null)
            {
                // اگر نتیجه بر اساس کلید تشکیل شده در کش موجود بود
                // همان را بازگشت می‌دهیم
                invocation.ReturnValue = cachedResult;
            }
            else
            {
```

```
                lock (lockObject)
                {
                    // در غیر اینصورت ابتدا متد را اجرا کرده
                    invocation.Proceed();
                    if (invocation.ReturnValue == null)
                        return;

                    // سپس نتیجه آن‌را کش می‌کنیم
                    cache.Insert(key: cacheKey,
                                 value: invocation.ReturnValue,
                                 dependencies: null,
                                 absoluteExpiration: DateTime.Now.AddSeconds(cacheDuration),
                                 slidingExpiration: TimeSpan.Zero);
                }
            }
        }

        private static Attribute getCacheMethodAttribute(IInvocation invocation)
        {
            var methodInfo = invocation.MethodInvocationTarget;
            if (methodInfo == null)
            {
                methodInfo = invocation.Method;
            }
            return Attribute.GetCustomAttribute(methodInfo, typeof(CacheMethodAttribute), true);
        }

        private static string getCacheKey(IInvocation invocation)
        {
            var cacheKey = invocation.Method.Name;

            foreach (var argument in invocation.Arguments)
            {
                cacheKey += ":" + argument;
            }

            // todo: بهتر است هش این کلید طولانی بازگشت داده شود
            // کار کردن با هش سریعتر خواهد بود
            return cacheKey;
        }
    }
}
```

کدهای CacheInterceptor مورد استفاده را در بالا مشاهده می‌کنید.
توضیحات ریز قسمت‌های مختلف آن به صورت کامنت، جهت درک بهتر عملیات، ذکر شده‌اند.

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.

```
using System;
using AOP02.Core;
using AOP02.Services;
using Castle.DynamicProxy;
using StructureMap;

namespace AOP02
{
    class Program
    {
        static void Main(string[] args)
        {
            ObjectFactory.Initialize(x =>
            {
                var dynamicProxy = new ProxyGenerator();
                x.For<IMyService>()
                 .EnrichAllWith(myTypeInterface =>
                        dynamicProxy.CreateInterfaceProxyWithTarget(myTypeInterface, new
CacheInterceptor()))
                 .Use<MyService>();
            });

            var myService = ObjectFactory.GetInstance<IMyService>();
```

```
            Console.WriteLine(myService.GetLongRunningResult("Test"));
            Console.WriteLine(myService.GetLongRunningResult("Test"));
        }
    }
}
```

در قسمت تنظیمات اولیه DI مورد استفاده، هر زمان که شیءایی از نوع IMyService درخواست شود، کلاس MyService وهله سازی شده و سپس توسط CacheInterceptor محصور می‌گردد. اکنون ادامه برنامه با این شیء محصور شده کار می‌کند.
حال اگر برنامه را اجرا کنید یک چنین خروجی قابل مشاهده خواهد بود:

```
Result of 'Test' returned at 2013/04/09 07:19:43
Result of 'Test' returned at 2013/04/09 07:19:43
```

همانطور که ملاحظه می‌کنید هر دو فراخوانی یک زمان را بازگشت داده‌اند که بیانگر کش شدن اطلاعات اولی و خوانده شدن اطلاعات فراخوانی دوم از کش می‌باشد (با توجه به یکی بودن پارامترهای هر دو فراخوانی).

از این پیاده سازی می‌شود به عنوان کش سطح دوم ORMها نیز استفاده کرد (صرفنظر از نوع ORM در حال استفاده).

دریافت مثال کامل این قسمت

AOP02.zip

نظرات خوانندگان

نویسنده: MehRad
تاریخ: ۱۳۹۲/۰۶/۱۹ ۱۸:۲۹

سلام
فرق این روش از کش کردن با کش سطح دوم که در این قسمت معرفی نمودین در چیست ؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۶/۱۹ ۱۸:۳۷

- خلاصه‌ای از قسمت اول این دوره
«هر Aspect صرفا یک محصور کننده قابلیتی خاص و تکراری در برنامه است. از این جهت که کدهای تکراری برنامه، به Aspects منتقل شده‌اند و دیگر نیازی نیست برای تغییر آن‌ها، کدهای قسمت‌های مختلف را تغییر داد (کدهای برنامه باز خواهند بود برای توسعه و بسته برای تغییر). بنابراین با استفاده از Aspects، به یک طراحی شیءگرای بهتر نیز دست خواهیم یافت.»
بنابراین فرق مهمش با روش کار با Expressions این است که شما در اینجا به یک Attribute جدید رسیدید که منطق پیاده سازی آن جایی در لابلای کدهای شما قرار نگرفته. هر زمان که نیازی به آن نبود، فقط کافی است که قسمت EnrichAllWith تنظیمات IoC Container یاده شده را حذف کرد. این روش یک دید دیگر طراحی شیءگرا است.
- از دیدگاه صرفا کاربردی:
الف) روش AOP یاد شده با هر نوع ORM ایی سازگار است. اصلا مهم نیست که الزاما EF باشد یا NH.
ب) چون درگیر بسیاری از جزئیات ریز تفسیر Expressions نشده، سریعتر است.

عنوان: **آشنایی با AOP IL Weaving**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۱/۲۱ ۹:۲۶
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, AOP, C#, Architecture

IL Weaving در AOP به معنای اتصال Aspects تعریف شده، پس از کامپایل برنامه به فایل‌های باینری نهایی است. اینکار با ویرایش اسمبلی‌ها در سطح IL یا کد میانی صورت می‌گیرد. بنابراین در این حالت دیگر یک محصور کننده و پروکسی، در این بین جهت مزین سازی اشیاء، در زمان اجرای برنامه تشکیل نمی‌شود. بلکه فراخوانی Aspects به معنای فراخوانی واقعی قطعه کدهایی است که به اسمبلی‌های برنامه پس از کامپایل آن‌ها تزریق شده‌اند.
در دنیای دات نت، ابزارهای چندی امکان انجام IL Weaving را فراهم ساخته‌اند که تعدادی از آن‌ها به قرار ذیل هستند:

- PostSharp
- LOOM.NET
- Wicca
- و ...

در بین این‌ها، PostSharp معروفترین فریم ورک AOP بوده و در ادامه از آن استفاده خواهیم کرد.

پیشنیاز ادامه بحث

ابتدا یک پروژه کنسول جدید را آغاز کرده و سپس در خط فرمان پاور شل نوگت در VS.NET دستور ذیل را اجرا کنید:

```
PM> Install-Package PostSharp
```

به این ترتیب ارجاعی به PostSharp به پروژه جاری اضافه خواهد شد. البته حجم آن نسبتا بالا است؛ نزدیک به 20 مگ به همراه ابزارهای تزریق کد همراه با آن. مجوز استفاده از آن نیز تجاری و مدت دار است.

مراحل ایجاد یک Aspect برای پروسه IL Code Weaving

ابتدا یک کلاس پایه مشتق شده از کلاسی ویژه موجود در یکی از فریم ورک‌های AOP باید تعریف شود. مرحله بعد، کار اتصال این Aspect می‌باشد که توسط پردازشگر ثانویه IL Code Weaving انجام می‌شود.
در ادامه قصد داریم همان مثال LoggingInterceptor قسمت دوم این سری را با استفاده از IL Code Weaving پیاده سازی کنیم.

```
using System;

namespace AOP03
{
    public class MyType
    {
        public void DoSomething(string data, int i)
        {
            Console.WriteLine("DoSomething({0}, {1});", data, i);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            new MyType().DoSomething("Test", 1);
        }
    }
}
```

کدهای برنامه همانند قبل است. اما اینبار بجای استفاده از Interceptors، با ارث بری از کلاس OnMethodBoundaryAspect کتابخانه PostSharp شروع خواهیم کرد:

```
using System;
using PostSharp.Aspects;

namespace AOP03
{
    [Serializable]
    public class LoggingAspect : OnMethodBoundaryAspect
    {
        public override void OnEntry(MethodExecutionArgs args)
        {
            Console.WriteLine("On Entry");
        }

        public override void OnExit(MethodExecutionArgs args)
        {
            Console.WriteLine("On Exit");
        }

        public override void OnSuccess(MethodExecutionArgs args)
        {
            Console.WriteLine("On Success");
        }

        public override void OnException(MethodExecutionArgs args)
        {
            Console.WriteLine("On Exception");
        }
    }
}
```

نیاز است این کلاس توسط ویژگی Serializable مزین شود تا توسط PostSharp قابل استفاده گردد. همانطور که ملاحظه می‌کنید، مراحل مختلف اجرای یک Aspcet در اینجا با override متدهای کلاس پایه OnMethodBoundaryAspect پیاده سازی شده‌اند. این مراحل را پیشتر در زمان استفاده از Interceptors توسط try/finally/catch بررسی کرده بودیم.

اکنون اگر برنامه را اجرا کنیم، اتفاق خاصی رخ نداده و همان خروجی معمول متد DoSomething در کنسول نمایش داده خواهد شد. بنابراین در مرحله بعد نیاز است تا این Aspect را به کدهای برنامه متصل کنیم.

کلاس OnMethodBoundaryAspect در کتابخانه PostSharp، از کلاس MulticastAttribute مشتق می‌شود. بنابراین LoggingAspect ایی را که ایجاد کرده‌ایم نیز می‌توان به صورت یک ویژگی به متدهای مورد نظر خود افزود:

```
public class MyType
    {
        [LoggingAspect]
        public void DoSomething(string data, int i)
        {
            Console.WriteLine("DoSomething({0}, {1});", data, i);
        }
    }
```

اکنون اگر برنامه را اجرا کنیم، با خروجی زیر مواجه خواهیم شد:

```
On Entry
DoSomething(Test, 1);
On Success
On Exit
```

برای اینکه بتوان عملیات رخ داده را بهتر توضیح داد می‌تواند از یک دی‌کامپایلر مانند برنامه معروف Reflector استفاده کرد:

```
public void DoSomething(string data, int i)
{
    <>z__Aspects.a0.OnEntry(null);
    try
    {
        Console.WriteLine("DoSomething({0}, {1});", data, i);
        <>z__Aspects.a0.OnSuccess(null);
    }
    catch (Exception)
    {
        <>z__Aspects.a0.OnException(null);
```

```
        throw;
    }
    finally
    {
        <>z__Aspects.a0.OnExit(null);
    }
}
```

این کدی است که به صورت پویا توسط PostSharp به اسمبلی نهایی فایل اجرایی برنامه تزریق شده است.

خوب! این یک روش اتصال Aspects به برنامه است. اما اگر همانند Interceptors بخواهیم Aspect تعریف شده را سراسری اعمال کنیم چکار باید کرد (بدون نیاز به قرار دادن ویژگی بر روی تک تک متدها)؟
برای اینکار ابتدا نیاز است میدان عملکرد Aspect تعریف شده را توسط ویژگی MulticastAttributeUsage محدود کنیم تا برای مثال به خواص اعمال نشوند:

```
[Serializable]
[MulticastAttributeUsage(MulticastTargets.Method, TargetMemberAttributes =
MulticastAttributes.Instance)]
public class LoggingAspect : OnMethodBoundaryAspect
```

سپس فایل AssemblyInfo.cs استاندارد پروژه را گشوده و سطر زیر را به آن اضافه کنید:

```
[assembly: LoggingAspect(AttributeTargetTypes = "AOP03.*")]
```

توسط AttributeTargetTypes می‌توان اعمال این Aspect را به یک فضای نام خاص نیز محدود کرد.

مزیت روش IL Code Weaving نسبت به Interceptors، کارآیی و سرعت بالاتر است. از این جهت که کدهایی که قرار است اجرا شوند، پیشتر در اسمبلی برنامه قرار گرفته‌اند و نیازی نیست تا در زمان اجرا، کدی به برنامه به صورت پویا تزریق گردد.

عنوان: **نگاهی به انواع Aspects موجود در کتابخانه PostSharp**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۱/۲۱ ۲۰:۱۴
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, AOP, C#, Architecture

تعدادی Aspect توکار در کتابخانه PostSharp قرار دارند که نقطه آغازین کار با آن‌را تشکیل می‌دهند. نمونه‌ای از آن‌را در قسمت قبل به نام OnMethodBoundaryAspect بررسی کردیم. اغلب این‌ها کلاس‌هایی هستند Abstract که با تهیه‌ی کلاس‌هایی مشتق شده از آن‌ها و override نمودن متدهای کلاس پایه، می‌توان Aspect جدیدی را ایجاد نمود. تمام این نوع Aspects در حقیقت نوعی مزین کننده به شمار می‌روند. در ادامه قصد داریم نگاهی داشته باشیم به سایر Aspects مهیای در کتابخانه PostSharp.

1) OnExceptionAspect

از OnExceptionAspect برای مدیریت استثناءهای متدها استفاده می‌شود. کار این Aspect، اضافه کردن try/catch به کدهای یک متد است و سپس فراخوانی متد OnException در صورت بروز خطایی در این بین.

```
using System;
using System.Reflection;
using PostSharp.Aspects;

namespace AOP03
{
    public class ApplicationExceptionHandlerAspect : OnExceptionAspect
    {
        public override void OnException(MethodExecutionArgs args)
        {
            Console.WriteLine("Exception Type: {0}, StackTrace: {1}",
                              args.Exception.GetType().Name,
                              args.Exception.StackTrace);
        }

        public override Type GetExceptionType(MethodBase targetMethod)
        {
            return typeof(ApplicationException);
        }
    }
}
```

مثالی را در این زمینه در کدهای فوق ملاحظه می‌کنید. اگر تنها متد OnException تحریف شود، try/catch خودکار اضافه شده به کدها، هر نوع استثنایی را مدیریت خواهد کرد. اما اگر متد GetExceptionType نیز در این بین مقدار دهی گردد، بر اساس نوع استثنای تعریف شده، کار فیلتر استثناها انجام می‌پذیرد و از مابقی صرفنظر خواهد شد.
نحوه استفاده از این Aspect نیز همانند مثال قسمت قبل است و جزئیات آن تفاوتی نمی‌کند.

2) LocationInterceptionAspect

این Aspect برخلاف سایر Aspectهایی که تاکنون بررسی کردیم، تنها در سطح خواص و فیلدهای یک کلاس عمل می‌کند. کار Interception در اینجا به معنای تحت کنترل قرار دادن اعمال set (پیش از فراخوانی set) و get (پیش از بازگشت مقدار) این خواص عمومی و حتی خصوصی تعریف شده است. کلمه Location در این Aspect به معنای متادیتای زمینه کاری است؛ مانند Name و FullName خواصی که مشغول به کار با آن‌ها هستیم.

```
using System;
using PostSharp.Aspects;

namespace AOP03
{
    public class ObjectInitializationAspect : LocationInterceptionAspect
    {
        public override void OnGetValue(LocationInterceptionArgs args)
        {
            if (args.GetCurrentValue() == null)
            {
                Console.WriteLine("Property {0} is null.", args.LocationFullName);
```

```
            }
        }
    }
}
```

یک نمونه از کاربرد آن‌را در مثال فوق مشاهده می‌کنید. در اینجا با تحریف متد OnGetValue، پیش از بازگشت مقداری از یک خاصیت، بررسی می‌شود که آیا مقدار آن null است یا خیر.
برای استفاده از آن نیز کافی است تا ویژگی ObjectInitializationAspect به خاصیتی دلخواه اضافه شود.
در اینجا 4 متد args.GetCurrentValue برای دریافت مقدار جاری خاصیت، args.SetNewValue جهت تنظیم مقداری جدید، args.ProceedGetValue و args.ProceedSetValue سبب اجرای حالت‌های get و set می‌شوند (چیزی شبیه به عملکرد اینترفیس IInterceptor که در قسمت‌های قبلی بررسی کردیم).

3) EventInterceptionAspect

EventInterceptionAspect همانطور که از نام آن نیز پیدا است، در سطح رخدادهای یک کلاس عمل می‌کند. سه متدی که این کلاس پایه برای تحت نظر قرار دادن اعمال رویدادگردان‌های یک کلاس در اختیار ما قرار می‌دهند شامل OnAddHandler، OnRemoveHandler و OnInvokeHandler هستند.

```
using PostSharp.Aspects;
using System;

namespace AOP03
{
    public class LogEventAspect : EventInterceptionAspect
    {
        public override void OnAddHandler(EventInterceptionArgs args)
        {
            Console.WriteLine("Event {0} added", args.Event.Name);
            args.ProceedAddHandler();
        }

        public override void OnRemoveHandler(EventInterceptionArgs args)
        {
            Console.WriteLine("Event {0} removed", args.Event.Name);
            args.ProceedRemoveHandler();
        }

        public override void OnInvokeHandler(EventInterceptionArgs args)
        {
            Console.WriteLine("Event {0} invoked", args.Event.Name);
            args.ProceedInvokeHandler();
        }
    }
}
```

مثالی را از نحوه تعریف یک EventInterceptionAspect مشاهده می‌کنید. در تمام حالاتی که متدهای کلاس پایه تحریف شده‌اند نیاز است از متدهای Proceed متناظر نیز استفاده شود تا برای مثال اضافه شدن، حذف و یا اجرای یک رویداد رخ دهند.

مدیریت اعمال Aspects در زمان کامپایل

یکی از متدهایی که در کلیه Aspects توکار فوق قابل تحریف است، CompileTimeValidate نام دارد.

```
public class LoggingAspect : OnMethodBoundaryAspect
    {
        public override bool CompileTimeValidate(System.Reflection.MethodBase method)
        {
            return !method.IsStatic;
        }
```

برای نمونه اگر آن‌را به OnMethodBoundaryAspect پیاده سازی شده در قسمت قبل، با تعاریف فوق اعمال کنیم، این Aspect

سفارشی دیگر به متدهای استاتیک، اعمال نخواهد شد. به این ترتیب می‌توان بر روی نحوه کامپایل ثانویه کدهایی که قرار است به اسمبلی برنامه اضافه شوند، تاثیر گذار بود.

چند نکته تکمیلی در مورد توزیع برنامه‌های مبتنی بر PostSharp

الف) اگر نیاز است به اسمبلی‌های خود امضای دیجیتال اضافه کنید، در حالت استفاده از PostSharp به علت بازنویسی کدهای IL اسمبلی تولیدی، نیاز است حالت delay signing انتخاب شود. به این معنا که ابتدا اسمبلی به صورت متداول کامپایل می‌شود. سپس PostSharp کار خود را انجام داده و در نهایت با استفاده از ابزارهای اعمال امضای دیجیتال باید کار افزودن آن‌ها در مرحله آخر انجام شود.
ب) در حال حاضر تنها برنامه Dotfuscator است که با PostSharp برای obfuscation سازگاری دارد.

نظرات خوانندگان

نویسنده: رضا1356
تاریخ: ۱۳۹۲/۱۰/۰۳ ۹:۴۵

اگر امکان دارد لینک مستقیمی جهت دریافت postsharp معرفی کنید.

ضمن اینکه با توجه به اینکه که فرمودید postsharp بسته تجاری و مدت دار است آیا درست است که پروژه امان را وابسته به آن کنیم.

تشکر

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۰/۰۳ ۱۱:۵۱

- لینک مستقیمی ندارم. جهت تست از بسته NuGet آن استفاده کنید.
- SQL Server هم تجاری است. Windows هم از بنیان تجاری است. احتمالا از هر دوی این‌ها استفاده می‌کنید. تجاری بودن دلیلی برای سرکوب اشتیاق به یادگیری مطلبی نیست و نخواهد بود.
- ضمن اینکه در قسمت‌های بعدی نمونه‌های سورس باز هم معرفی شده‌اند.

نویسنده: رضا شش
تاریخ: ۱۳۹۲/۱۰/۰۴ ۱۵:۵۳

با سلام
من آخرین نسخه postsharp رو از سایت نیوگت دریافت کردم در یک برنامه ساده HelloWord استفاده کردم وقتی دیباگ می‌کنم وارد قسمت Aspect نمی‌شود با چند dll مختلف امتحان کردم و فقط یک ورژن 2 پیدا کردم که جواب داد آیا تنظیمات خاصی نیاز دارد؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۰/۰۴ ۱۷:۱۰

جزئیات مراحل اتصال Aspects در قسمت قبل بررسی شدند. همچنین این کتابخانه صرفا DLL ایی نیست. یک سری مراحل post build را باید به VS.NET اضافه کند تا پس از کامپایل اولیه برنامه، کار تغییر اسمبلی را انجام دهد.

نویسنده: رضا شش
تاریخ: ۱۳۹۲/۱۰/۰۵ ۹:۳۵

من مثال ذکر شده helloword را در سایت قرار می‌دهم. اسمبلی postsharp استفاده شده فقط ورژنش فرق می‌کند ولی در یکی کار می‌کند و در دیگری خیر.

HelloWorld.rar

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۰/۰۵ ۱۰:۹

روی سیستم من هیچکدام از مثال‌های شما کار نکردند. دلایل:
الف) همانطور که عرض شد، PostSharp فقط یک DLL نیست (IL Weaving به معنای دستکاری کدهای IL و اسمبلی نهایی است و

افزودن کدهایی در این میان). بسته نیوگت آن، یک سری مراحل Post Build را به فایل csproj اضافه می‌کند؛ برای مثال:

```
<Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
  <Import Project="..\packages\PostSharp.2.1.7.30\tools\PostSharp.targets"
    Condition="Exists('..\packages\PostSharp.2.1.7.30\tools\PostSharp.targets')" />
```

ب) حتما باید سیستم licensing آن توسط نیوگت نصب شود تا عملیات IL Weaving را انجام دهد.
ج) زمانیکه از طریق نیوگت نصب می‌شود، پوشه packages\PostSharp.2.1.7.30\tools آن کار اصلی IL Weaving را انجام می‌دهد و این پوشه بالای 10 مگابایت است.

نویسنده: رضا شش
تاریخ: ۱۳۹۲/۱۰/۰۷ ۹:۵۰

با عرض معذرت چند سوال دارم:
-1 اینطور که من متوجه شدم اگر بخواهیم در هر پروژه ای از postsharp استفاده کنیم حتما باید به اینترنت وصل باشیم و بسته چندین مگاباتی نیوگت آن را نصب کنیم. اگر اینطور است در شرکت‌ها و سازمان‌ها همه سیستم‌ها اجازه دسترسی به اینترنت را ندارند.
-2 در پروژه من بعد از کامپایل یک پیغام در قسمت output درج می‌شود که می‌گوید چند روز تا انقضای این بسته فرصت دارید. پس از انقضای مهلت مقرر چکار باید کرد چون بنا دارم از این امکان در پروژه ام استفاده کنم.
-3 در مثالهایی دریافتی از اینترنت یک فایل اجرایی وجود دارد به نام PostSharp.MSBuild.Samples.exe این فایل چه کاربردی دارد. چون در سیستم من اجرا نمی‌شود.
ExceptionHandling.zip

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۱۰/۰۷ ۱۰:۱۱

- بحث در مورد AOP بدون ذکر نامی از PostSharp بی‌معنا بود. به همین جهت چند قسمتی به آن اختصاص داده شد. حداقل از لحاظ بحث مفهومی ارزشمند است.
- در سازمان‌ها امکان تشکیل یک مخزن نیوگت محلی وجود دارد. یعنی فقط کافی است یکی از سیستم‌ها تبدیل به مخزن شود و بقیه از آن استفاده کنند. اطلاعات بیشتر در اینجا
- پیشنهاد من استفاده از پروژه‌های سورس باز مشابهی است مانند Fody. یک نمونه از کاربرد آن‌را در ادامه این دوره بررسی کرده‌ایم: « معرفی پروژه NotifyPropertyWeaver ». امکانات زیادی دارد . یا اینکه اصلا از IL Weaving استفاده نکنید و از dynamic proxy مطرح شده مانند پروژه castle core که در قسمت‌های قبل بررسی شد، استفاده نمائید.
- post sharp زمانیکه از طریق نیوگت نصب می‌شود، خودش را در سیستم build ویژوال استودیو مرتبط با پروژه جاری ثبت می‌کند. پس از اینکه dll یا فایل exe شما توسط VS.NET تولید شد، به صورت خودکار کار post sharp آغاز شده و کدهای IL اضافی پیاده سازی کننده aspects مدنظر را به اسمبلی‌های برنامه اضافه می‌کند.

عنوان: **استفاده از IL Code Weaving برای تولید ویژگی‌های تکراری مورد نیاز در WCF**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۱/۲۲ ۰:۱۲
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, AOP, C#, Architecture

با استفاده از IL Code Weaving علاوه بر مدیریت اعمال تکراری پراکنده در سراسر برنامه مانند ثبت وقایع، مدیریت استثناءها، کش کردن داده‌ها و غیره، می‌توان قابلیتی را به کدهای موجود نیز افزود. برای مثال یک برنامه معمول WCF را درنظر بگیرید.

```
using System.Runtime.Serialization;

namespace AOP03.DataContracts
{
    [DataContract]
    public class User
    {
        [DataMember]
        public int Id { set; get; }

        [DataMember]
        public string Name { set; get; }
    }
}
```

نیاز است کلاس‌ها و خواص آن توسط ویژگی‌های DataContract و DataMember مزین شوند. در این بین نیز اگر یکی فراموش گردد، کار دیباگ برنامه مشکل خواهد شد و در کل حجم بالایی از کدهای تکراری در اینجا باید در مورد تمام کلاس‌های مورد نیاز انجام شود. در ادامه قصد داریم تولید این ویژگی‌ها را توسط PostSharp انجام دهیم. به عبارتی یک پوشه خاص به نام DataContracts را ایجاد کرده و کلاس‌های خود را به نحوی متداول و بدون اعمال ویژگی خاصی تعریف کنیم. در ادامه پس از کامپایل آن، به صورت خودکار با ویرایش کدهای IL توسط PostSharp، ویژگی‌های لازم را به اسمبلی نهایی اضافه نمائیم.

تهیه DataContractAspect جهت اعمال خودکار ویژگی‌های DataContract و DataMember

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Runtime.Serialization;
using PostSharp.Aspects;
using PostSharp.Extensibility;
using PostSharp.Reflection;

namespace AOP03
{
    [Serializable]
    //این ویژگی تنها نیاز است به کلاس‌ها اعمال شود//
    [MulticastAttributeUsage(MulticastTargets.Class)]
    public class DataContractAspect : TypeLevelAspect, IAspectProvider
    {
        public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
        {
            var targetType = (Type)targetElement; //همان نوعی است که ویژگی جاری به آن اعمال خواهد شد

            //این سطر معادل است با درخواست تولید ویژگی دیتاکانترکت
            var introduceDataContractAspect = new CustomAttributeIntroductionAspect(
                new ObjectConstruction(typeof(DataContractAttribute).GetConstructor(Type.EmptyTypes)));

            //این سطر معادل است با درخواست تولید ویژگی دیتاممبر
            var introduceDataMemberAspect = new CustomAttributeIntroductionAspect(
                new ObjectConstruction(typeof(DataMemberAttribute).GetConstructor(Type.EmptyTypes)));

            //در اینجا کار اعمال ویژگی دیتاکانترکت به کلاسی که به عنوان پارامتر متد جاری
            //دریافت شده انجام خواهد شد
            yield return new AspectInstance(targetType, introduceDataContractAspect);

            //مرحله بعد کار اعمال ویژگی دیتاممبر به خواص کلاس است
            foreach (var property in targetType.GetProperties(BindingFlags.Public |
                                                              BindingFlags.DeclaredOnly |
                                                              BindingFlags.Instance))
            {
```

```
                if (property.CanWrite)
                    yield return new AspectInstance(property, introduceDataMemberAspect);
            }
        }
    }
}
```

توضیحات مرتبط با قسمت‌های مختلف این Aspect سفارشی، به صورت کامنت در کدهای فوق ارائه شده‌اند.
برای اعمال آن به سراسر برنامه تنها کافی است به فایل AssemblyInfo.cs پروژه مراجعه و سپس سطر زیر را به آن اضافه کنیم:

```
[assembly: DataContractAspect(AttributeTargetTypes = "AOP03.DataContracts.*")]
```

به این ترتیب در زمان کامپایل پروژه، Aspect تعریف شده به تمام کلاس‌های موجود در فضای نام AOP03.DataContracts اعمال خواهند شد.

در این حالت اگر کلیه ویژگی‌های کلاس User فوق را حذف و برنامه را کامپایل کنیم، با مراجعه به برنامه ILSpy می‌توان صحت اعمال ویژگی‌ها را به کمک PostSharp بررسی کرد:

```
using System;
using System.Diagnostics;
using System.Runtime.CompilerServices;
using System.Runtime.Serialization;
namespace AOP03.DataContracts
{
    [DataContract]
    public class User
    {
        [System.Runtime.CompilerServices.CompilerGenerated, System.Diagnostics.DebuggerNonUserCode]
        internal sealed class <>z__Aspects...
        [DataMember]
        public int Id...
        [DataMember]
        public string Name...
    }
}
```

عنوان: **معرفی پروژه NotifyPropertyWeaver**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۱/۲۲ ۹:۱۲
آدرس: www.dotnettips.info
گروه‌ها: C#, Design patterns, AOP, Architecture

پس از معرفی مباحث IL Code Weaving و همچنین ارائه راه حلی در مورد «استفاده از AOP Interceptors برای حذف کدهای تکراری INotifyPropertyChanged در WPF» راه حل مشابهی به نام NotifyPropertyWeaver ارائه شده است که همان کار AOP Interceptors را انجام می‌دهد؛ اما بدون نیاز به تشکیل پروکسی و سربار اضافی. کار نهایی را توسط ویرایش اسمبلی و افزودن کدهای IL لازم انجام می‌دهد؛ البته بدون استفاده از PostSharp. این پروژه از کتابخانه سورس باز پایه‌ای به نام Fody استفاده می‌کند که جهت IL Code weaving طراحی شده است.
اگر به Wiki آن مراجعه نمائید، لیست افزونه‌های قابل توجهی را در مورد آن خواهید یافت که PropertyChanged تنها یکی از آن‌ها است.

پیشنیازها

الف) صفحه پروژه در GitHub
ب) دریافت از طریق نوگت

روش استفاده

پس از نصب بسته نوگت پروژه PropertyChanged.Fody

```
PM> Install-Package PropertyChanged.Fody
```

کلاسی را که باید پس از کامپایل، پیاده سازی‌های خودکار OnPropertyChanged را شامل شود، با ویژگی ImplementPropertyChanged مزین کنید.

```
using PropertyChanged;

namespace AOP02
{
    [ImplementPropertyChanged]
    public class Person
    {
        public string Id { set; get; }
        public string Name { set; get; }
    }
}
```

و سپس پروژه را کامپایل نمائید. خروجی کنسول Build در VS.NET :

```
------ Build started: Project: AOP02, Configuration: Debug x86 ------
  Fody (version 1.13.6.1) Executing
  Finished Fody 287ms.
  AOP02 -> D:\Prog\AOP02\bin\Debug\AOP02.exe
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

اکنون اگر فایل اسمبلی نهایی پروژه را در برنامه ILSpy باز کنیم، چنین پیاده سازی را می‌توان شاهد بود:

```
using System;
using System.ComponentModel;
using System.Runtime.CompilerServices;
namespace AOP02
{
    public class Person : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        public string Id
```

```
        {
            [System.Runtime.CompilerServices.CompilerGenerated]
            get
            {
                return this.<Id>k__BackingField;
            }
            [System.Runtime.CompilerServices.CompilerGenerated]
            set
            {
              if (string.Equals(this.<Id>k__BackingField, value, System.StringComparison.Ordinal))
              {
                  return;
              }
              this.<Id>k__BackingField = value;
              this.OnPropertyChanged("Id");
            }
        }
        public string Name
        {
            [System.Runtime.CompilerServices.CompilerGenerated]
            get
            {
               return this.<Name>k__BackingField;
            }
            [System.Runtime.CompilerServices.CompilerGenerated]
            set
           {
             if (string.Equals(this.<Name>k__BackingField, value, System.StringComparison.Ordinal))
             {
                return;
             }
             this.<Name>k__BackingField = value;
             this.OnPropertyChanged("Name");
            }
        }
        public virtual void OnPropertyChanged(string propertyName)
        {
            PropertyChangedEventHandler propertyChanged = this.PropertyChanged;
            if (propertyChanged != null)
            {
                propertyChanged(this, new PropertyChangedEventArgs(propertyName));
            }
        }
    }
}
```

نظرات خوانندگان

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۲/۰۱/۲۲ ۱۰:۱۶

تشکر فراوان.
برای پیاده سازی زیر از NotifyPropertyWeaver می‌توان استفاده کرد ؟

```
public Project ActiveProject
{
            get { return _activeProject; }
            set
            {
                _activeProject = value;
                RaisePropertyChanged(() => ActiveProject);
                RaisePropertyChanged(() => HasProject);
            }
}

  public bool HasProject
        {
            get { return ActiveProject != null; }
        }
```

چون تغییر HasProject به ActiveProject وابسته است قصد داریم هر زمان ActiveProject عوض می‌شود RaisePropertyChanged برای HasProject هم فراخوانی شود.
پ ن : آقای نصیری تلفظ صحیحش نوگت هست یا نیوگت؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۱/۲۲ ۱۰:۴۹

- بله. یک سری ویژگی دیگر هم دارد به نام‌های AlsoNotifyFor و DoNotNotify.
- بله؛ نیوگت هست.

نویسنده: ایمان محمدی
تاریخ: ۱۳۹۲/۰۳/۱۵ ۱۳:۲۴

حیفم اومد نظر ندم دستتون درد نکنه فوق العاده عالی بود.
دو تا نکته جالبی که من دیدم:
اگه بدنه set رو تعریف کرده باشید ولی OnPropertyChanged نداشته باشه بازهم به بدنه ست OnPropertyChanged اضافه میکنه.
اگه پروپرتی‌های دیگه ای تو متد get خودشون از این پروپرتی استفاده کرده باشند برای هر کدوم از اونها هم OnPropertyChanged اضافه می‌کنه.

نویسنده: سوین
تاریخ: ۱۳۹۲/۰۹/۰۸ ۱۱:۴۳

سلام
من هیچ Packge ای رو نمی‌تونم با Nuget نصب کنم و این خطا رو می‌ده
Install-Package : The underlying connection was closed: An unexpected error occurred on a send.

اما کانکشنم close نیست ممنون میشم راهنمایی کنید .

نویسنده: وحید نصیری

تاریخ: ۱۳۹۲/۰۹/۰۸ ۱۲:۲۰

- فید NuGet در VS.NET به Https تنظیم شده است. اگر دسترسی به **s** Http برای شما به کندی صورت می‌گیرد فقط کافی است مسیر فید آن‌را در منوی Tools، گزینه‌ی Options، ذیل قسمت Package manager یافته و به http://nuget.org/api/v2 تغییر دهید؛ یعنی به Http خالی، بجای **s** Http ؛ تا سرعت دریافت بسته‌های NuGet مورد نظر افزایش یابند.
- این بسته از طریق آدرس ذیل نیز قابل دریافت است:
https://az320820.vo.msecnd.net/packages/propertychanged.fody.1.42.0.nupkg
همین آدرس را در **IE** وارد کنید. اگر کار نکرد احتمالا تنظیمات IE شما به هم ریخته است؛ چون تنظیمات آن به صورت مستقیم روی تنظیمات اتصالی برنامه‌های دات نت تاثیر دارند.

نویسنده: داستان
تاریخ: ۱۳۹۳/۰۲/۲۱ ۱۰:۲۷

ممنون؛ خب چطوری رویداد propertyChanged رو هندل کنیم؟ (همچنین رویدادی به کلاس اضافه نمی‌شود)

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۲/۲۱ ۱۱:۰

- نیازی به اینکار نیست (چون کدهای آن به صورت خودکار به اسمبلی اضافه می‌شوند).
- اگر هدف این است که در اینجا در مورد یک خاصیت دیگر نیز اطلاع رسانی صورت گیرد، از ویژگی‌های دیگر آن به نام‌های AlsoNotifyFor و DoNotNotify استفاده کنید.

نویسنده: جوادی
تاریخ: ۱۳۹۳/۰۴/۱۴ ۱۳:۵۷

سلام؛
چطور یک Subscriber برای رویداد propertyChanged بنویسیم، درصورتیکه این رویداد وجود ندارد؟ درصورت امکان، یک مثال از نحوه اطلاع رسانی هنگام تغییر مقدار Property بزنید. ممنون

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۴/۱۴ ۱۴:۵

از NotificationInterception آن استفاده کنید یا از روش On_PropertyName_Changed