عنوان: **کاربردهای Static reflection - قسمت اول**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۵/۲۰ ۱۹:۰۸:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: #C

در مورد static reflection مقدمه‌ای پیشتر در این سایت قابل مطالعه است (^) و پیشنیاز بحث جاری است. در ادامه قصد داریم یک سری از کاربردهای متداول آن‌را که این روزها در گوشه و کنار وب یافت می‌شود، به زبان ساده بررسی کنیم.

بهبود کدهای موجود

از static reflection در دو حالت کلی می‌توان استفاده کرد. یا قرار است کتابخانه‌ای را از صفر طراحی کنیم یا اینکه خیر؛ کتابخانه‌ای موجود است و می‌خواهیم کیفیت آن‌را بهبود ببخشیم. هدف اصلی هم «حذف رشته‌ها» و «استفاده از کد بجای رشته‌ها» است.
برای مثال قطعه کد زیر یک مثال متداول مرتبط با WPF و یا Silverlight است. در آن با پیاده سازی اینترفیس INotifyPropertyChanged و استفاده از متد raisePropertyChanged ، به رابط کاربری برنامه اعلام خواهیم کرد که لطفا خودت را بر اساس اطلاعات جدید تنظیم شده در قسمت set خاصیت Name ، به روز کن:

```
using System.ComponentModel;

namespace StaticReflection
{
    public class User : INotifyPropertyChanged
    {
        string _name;
        public string Name
        {
            get { return _name; }
            set
            {
                if (_name == value) return;
                _name = value;
                raisePropertyChanged("Name");
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;
        void raisePropertyChanged(string propertyName)
        {
            var handler = PropertyChanged;
            if (handler == null) return;
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
```

```
    }
}
```

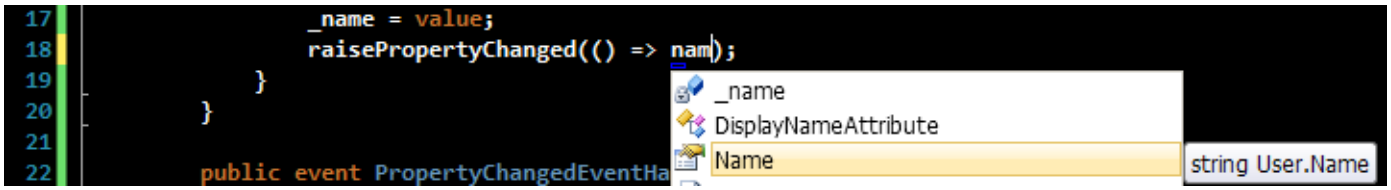
تعاریف قسمت PropertyChangedEventArgs این پیاده سازی، خارج از کنترل ما است و در دات نت فریم ورک تعریف شده است. حتما هم نیاز به رشته دارد؛ آن هم نام خاصیتی که تغییر کرده است. چقدر خوب می‌شد اگر می‌توانستیم این رشته را حذف کنیم تا کامپایلر بتواند صحت بکارگیری اطلاعات وارد شده را دقیقا پیش از اجرای برنامه بررسی کند. الان فقط در زمان اجرا است که متوجه خواهیم شد، مثلا آیا به روز رسانی مورد نظر صورت گرفته‌است یا خیر؛ اگر نه، یعنی احتمالا یک اشتباه تایپی جایی وجود دارد.

برای بهبود این کد همانطور که در قسمت قبل نیز گفته شد، از ترکیب کلاس‌های Expression و Func استفاده خواهیم کرد. در اینجا Func قرار نیست چیزی را اجرا کند، بلکه از آن به عنوان قطعه کدی که اطلاعاتش قرار است استخراج شود (Lambdas as Data) استفاده می‌شود. این استخراج اطلاعات هم توسط کلاس Expression انجام می‌شود. بنابراین قسمت اول بهبود کد به صورت زیر شروع می‌شود:

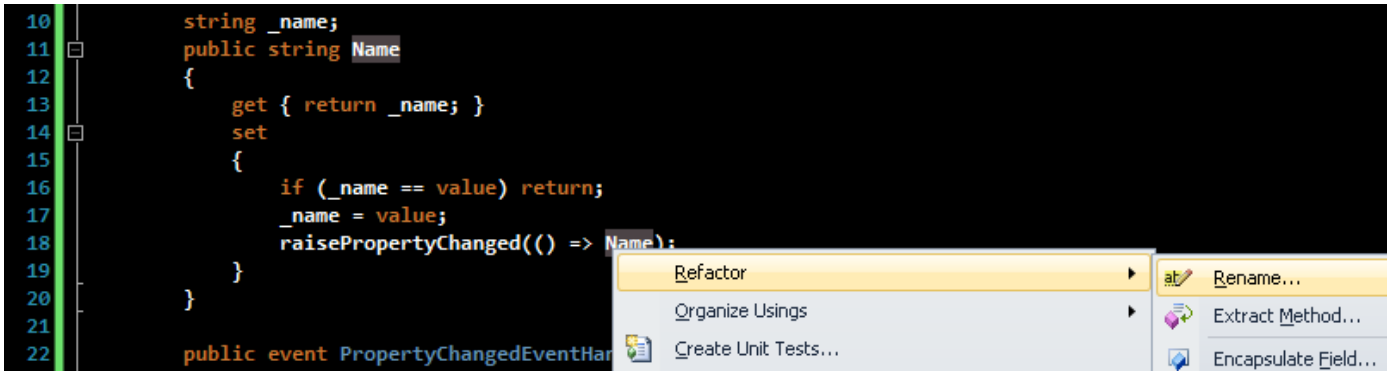
```
void raisePropertyChanged(Expression<Func<object>> expression)
```

الان اگر متد raisePropertyChanged بکارگرفته شده در خاصیت Name را بخواهیم اصلاح کنیم، حداقل با دو واقعه‌ی مطلوب زیر مواجه خواهیم شد:

Intellisense به صورت خودکار کار می‌کند:



حتی بدوی‌ترین ابزارهای Refactoring موجود (منظور همان ابزار توکار VS.NET است!) هم امکان Refactoring را در اینجا فراهم خواهند ساخت:



در پایان کد تکمیل شده فوق به شرح زیر خواهد بود که در آن از کلاس Expression جهت استخراج Member.Name استفاده شده است:

```
using System;
using System.ComponentModel;
using System.Linq.Expressions;

namespace StaticReflection
{
    public class User : INotifyPropertyChanged
    {
        string _name;
        public string Name
        {
            get { return _name; }
            set
            {
                if (_name == value) return;
                _name = value;
                raisePropertyChanged(() => Name);
            }
        }

        public event PropertyChangedEventHandler PropertyChanged;
        void raisePropertyChanged(Expression<Func<object>> expression)
        {
            var memberExpression = expression.Body as MemberExpression;
            if (memberExpression == null)
                throw new InvalidOperationException("Not a member access.");

            var handler = PropertyChanged;
            if (handler == null) return;
            handler(this, new PropertyChangedEventArgs(memberExpression.Member.Name));
        }
    }
}
```

در اینجا باز هم نهایتا به همان PropertyChangedEventArgs استاندارد و موجود، برمی‌گردیم؛ اما آرگومان رشته‌ای آن‌را به کمک ترکیب کلاس‌های Expression و Func تامین خواهیم کرد.

نظرات خوانندگان

نویسنده: Meysam Javadi
تاریخ: ۱۳۹۰/۰۵/۲۳ ۱۰:۱۳:۲۲

این روش مشکل کارایی داره، INPC ها بسیار بیشتر از اون چیزی که ما انتظارش رو داریم اجرا میشن!
به جاش Resharper 6.0 رو نصب بکنید(بایندینگ های XAML رو هم اصلاح میکنه یا میگه تو این DataContext نیست و...)

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۵/۲۳ ۱۱:۱۴:۲۱

- یک بررسی علمی (بدون علامت تعجب احساسی در انتهای جمله) اینجا هست: (+)
در «یک میلیون بار» اجرا، حدودا 10 ثانیه تفاوت اجرا است نسبت به حالت بکارگیری رشته‌ها.
البته شما در عمل، نه در محیط آزمایشگاهی، پیدا کنید برنامه‌ای را که یک میلیون بار بخواهد خواصی را مرتبا به روز کند.
- زمانیکه LINQ هم ارائه شد، اولین مقالاتی که در این مورد ... در مورد نقد آن منتشر شد، تمرکز را گذاشتند روی کارآیی؛ که این کمی کند است! البته الان کمتر کسی است که در پروژه‌هایش حداقل از LINQ to Objects استفاده نکند. به این دلایل:
- هدف استفاده از LINQ اصلا مسابقه‌ی سرعت نیست.
- هدف تولید کدهای Strongly typed که این اهمیت‌ها را دارند: تحت نظر کامپایلر هستند، قابلیت refactoring دارند و intellisense خودکاری را به همراه خواهند داشت. تمام این‌ها نگهداری یک پروژه را (که اصل زمان اختصاص داده شده به توسعه یک نرم افزار هم همین قسمت نگهداری است)، ساده‌تر و قابل تحمل‌تر می‌کند.
- کاهش حجم کدهای نوشته شده. شما می‌تونید حجم بالایی از if-else و for و حلقه‌ها و غیره رو با یک سطر LINQ نمایش بدید. این هم در بالابردن خوانایی و همچنین نگهداری ساده‌تر برنامه مؤثر است.
- تبدیل ساده‌تر اطلاعات خام به اشیاء (LINQ to xyz ها)
و ...

شما خیلی از مزایا رو بدست خواهید آورد اما خوب مسلما این‌ها هزینه هم دارند. اما نه آنچنان که کسی بخواهد از آن‌ها صرفنظر کند.

نویسنده: Meysam Javadi
تاریخ: ۱۳۹۰/۰۵/۲۳ ۱۱:۲۷:۰۸

کامنتم بیشتر لحن آموزشی داشت تا انتقاد. تو یکی از پست های شما در مورد پیاده سازی INPC پیشنهادم رو داده بودم فکر کنم
http://justinangel.net/AutomagicallyImplementingINotifyPropertyChanged

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۵/۲۳ ۱۲:۰۷:۳۵

هدف من از این بحث، بحث در مورد refactoring متدی بود که رشته‌ای را که دقیقا نام یکی از خاصیت‌های یک کلاس است را قبول می‌کند. می‌تونست یک مثال دیگر باشد. می‌تونست اصلا ربطی به این INPC نداشته باشد.