عنوان: **آشنایی با Refactoring - قسمت 1**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۲ ۰۹:۴۶:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

کارهای سورس باز قابل توجهی از برنامه نویس‌های ایرانی یافت نمی‌شوند؛ عموما کارهای ارائه شده در حد یک سری مثال یا کتابخانه‌های کوچک است و در همین حد. یا گاهی هم انگشت شمار پروژه‌هایی کامل. مثل یک وب سایت یا یک برنامه نصفه نیمه دبیرخانه و امثال آن. این‌ها هم خوب است از دیدگاه به اشتراک گذاری اطلاعات، ایده‌ها و هم ... یک مزیت دیگر را هم دارد و آن این است که بتوان کیفیت عمومی کد نویسی را حدس زد.
اگر کیفیت کدها رو بررسی کنید به یک نتیجه‌ی کلی خواهید رسید: "عموم برنامه نویس‌های ایرانی (حداقل این‌هایی که چند عدد کار سورس باز به اشتراک گذاشته‌اند) با مفهومی به نام Refactoring هیچگونه آشنایی ندارند". مثلا یک برنامه‌ی WinForm تهیه کرده‌اند و کل سورس برنامه همان چند عدد فرم برنامه است و هر فرم بالای 3000 سطر کد دارد. دوستان عزیز! به این می‌گویند «فاجعه‌ای به نام کدنویسی!» صاحب اول و آخر این نوع کدها خودتان هستید! شاید به همین جهت باشد که عمده‌ی پروژه‌های سورس باز پس از اینکه برنامه نویس اصلی از توسعه‌ی آن دست می‌کشد، «می‌میرند». چون کسی جرات نمی‌کند به این کدها دست بزند. مشخص نیست الان این قسمت را که تغییر دادم، کجای برنامه به هم ریخت. تستی ندارند. ساختاری را نمی‌توان از آن‌ها دریافت. منطق قسمت‌های مختلف برنامه از هم جدا نشده است. برنامه یک فرم است با چند هزار سطر کد در یک فایل! کار شما شبیه به کد اسمبلی چند هزار سطری حاصل از decompile یک برنامه که نباید باشد!
به همین جهت قصد دارم یک سری «ساده» Refactoring را در این سایت ارائه دهم. روی سادگی هم تاکید کردم، چون اگر عموم برنامه نویس‌ها با همین موارد به ظاهر ساده آشنایی داشتند، کیفیت کد نویسی بهتری را می‌شد در نتایج عمومی شده، شاهد بود.
این مورد در راستای نظر سنجی انجام شده هم هست؛ درخواست مقالات خالص سی شارپ در صدر آمار فعلی قرار دارد.

Refactoring چیست؟

Refactoring به معنای بهبود پیوسته کیفیت کدهای نوشته شده در طی زمان است؛ بدون ایجاد تغییری در عملکرد اصلی برنامه. به این ترتیب به کدهایی دست خواهیم یافت که قابلیت آزمون پذیری بهتری داشته، در مقابل تغییرات مقاوم و شکننده نیستند و همچنین امکان به اشتراک گذاری قسمت‌هایی از آن‌ها در پروژه‌های دیگر نیز میسر می‌شود.

قسمت اول - مجموعه‌ها را کپسوله کنید

برای مثال کلاس‌های ساده زیر را در نظر بگیرید:

```
namespace Refactoring.Day1.EncapsulateCollection
{
    public class OrderItem
    {
        public int Id { set; get; }
        public string Name { set; get; }
        public int Amount { set; get; }
    }
}
```

```
using System.Collections.Generic;

namespace Refactoring.Day1.EncapsulateCollection
{
    public class Orders
    {
        public List<OrderItem> OrderItems { set; get; }
    }
```

```
}
```

نکته اول: هر کلاس باید در داخل یک فایل جدا قرار گیرد. «لطفا» یک فایل درست نکنید با 50 کلاس داخل آن. البته اگر باز هم یک فایل باشد که بتوان 50 کلاس را داخل آن مشاهده کرد که چقدر هم عالی! نه اینکه یک فایل باشد تا بعدا 50 کلاس را با Refactoring از داخل آن بیرون کشید!

قطعه کد فوق، یکی از روش‌های مرسوم کد نویسی است. مجموعه‌ای به صورت یک List عمومی در اختیار مصرف کننده قرار گرفته است. حال اجازه دهید تا با استفاده از برنامه FxCop برنامه فوق را آنالیز کنیم. یکی از خطاهایی را که نمایش خواهد داد عبارت زیر است:

```
Error, Certainty 95, for Do Not Expose Generic Lists
```

بله. لیست‌های جنریک را نباید به همین شکل در اختیار مصرف کننده قرار داد؛ چون به این صورت هر کاری را می‌توانند با آن انجام دهند، مثلا کل آن را تعویض کنند، بدون اینکه کلاس تعریف کننده آن از این تغییرات مطلع شود.
پیشنهاد FxCop این است که بجای List از Collection یا IList و موارد مشابه استفاده شود. اگر اینکار را انجام دهیم اینبار به خطای زیر خواهیم رسید:

```
Warning, Certainty 75, for Collection Properties Should Be ReadOnly
```

FxCop پیشنهاد می‌دهد که مجموعه تعریف شده باید فقط خواندنی باشد.

چکار باید کرد؟
بجای استفاده از List جهت ارائه مجموعه‌ها، از IEnumerable استفاده کنید و اینبار متدهای Add و Remove اشیاء به آن‌را به صورت دستی تعریف نمائید تا بتوان از تغییرات انجام شده بر روی مجموعه ارائه شده، در کلاس اصلی آن مطلع شد و امکان تعویض کلی آن‌را از مصرف کننده گرفت. برای مثال:

```
using System.Linq;
using System.Collections.Generic;

namespace Refactoring.Day1.EncapsulateCollection
{
    public class Orders
    {
        private int _orderTotal;
        private List<OrderItem> _orderItems;

        public IEnumerable<OrderItem> OrderItems
        {
            get { return _orderItems; }
        }

        public void AddOrderItem(OrderItem orderItem)
        {
            _orderTotal += orderItem.Amount;
            _orderItems.Add(orderItem);
        }

        public void RemoveOrderItem(OrderItem orderItem)
        {
            var order = _orderItems.Find(o => o == orderItem);
            if (order == null) return;
```

```
            _orderTotal -= orderItem.Amount;
            _orderItems.Remove(orderItem);
        }
    }
}
```

اکنون اگر برنامه را مجددا با fxCop آنالیز کنیم، دو خطای ذکر شده دیگر وجود نخواهند داشت. اگر این تغییرات صورت نمی‌گرفت، امکان داشتن فیلد _orderTotal غیر معتبری در کلاس Orders به شدت بالا می‌رفت. زیرا مصرف کننده مجموعه OrderItems می‌توانست به سادگی آیتمی را به آن اضافه یا از آن حذف کند، بدون اینکه کلاس Orders از آن مطلع شود یا اینکه بتواند عکس العمل خاصی را بروز دهد.

نظرات خوانندگان

نویسنده: mrdotnet
تاریخ: ۱۳۹۰/۰۷/۱۲ ۱۳:۲۶:۰۸

سلام
با توجه به اینکه نسخه جدید fxCop با Windows SDK ارائه شده که حجم SDK حدود 600 مگ هست ، دوستانی که به هر دلیلی مایل به دانلود کل SDK نیستند میتونن از فایل زیر به در یافت تنها fxCop با حجم 10 مگ اقدام کنند.
http://www.mediafire.com/?hq3k13d7cuoxe7rدر ضمن یک آموزش نحوه استفاده مختصر و مفید از این ابزار رو میتونید در این لینک ببینید http://www.codeproject.com/KB/dotnet/FxCop.aspx

حالا شما آماده هستید تا سری آموزش های آشنایی با Refactoring رو دنبال کنید.

نویسنده: Tohid Azizi
تاریخ: ۱۳۹۰/۰۹/۰۳ ۲۰:۱۱:۴۷

با سلام و تشکر از سری مقالات بسیار مفید ریفکتورینگ.
در مورد خطای «Do Not Expose Generic Lists» و کد ریفکتور شده ی آن، آیا راهی وجود دارد که بتوان از قابلیت های اندکس ICollection برای پروپرتی استفاده کرد اما نتوان با استفاده از Add یا Insert عضوی به آن اضافه کرد؟مثلا ـ طبق مثال شما ـ داشته باشیم:for (int i=0; i<Orders.OrderItems.Count; i++) Console.WriteLine(Orders.OrderItems[i].Price);یا حتا:for (int i=0; i<Orders.OrderItems.Count; i++) Orders.OrderItems[i].Tax = Orders.OrderItems[i].Price * .05;اما نتوان نمونه ی جدیدی به لیست OrderItems اضافه کرد؟Orders.OrderItems.Add(newOrderItem); //raise errorبا تشکر

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۹/۰۳ ۲۱:۱۳:۵۲

می‌تونید چیزی شبیه به ReadOnlyCollection درست کنید ((^)). ReadOnlyCollection در حقیقت ICollection را پیاده سازی کرده با این تفاوت که پیاده سازی متد Add آن را معادل throw NotSupportedException قرار داده ((^)).

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳۹۲/۰۹/۱۳ ۱۴:۱۸

مبحث مهمی رو دارید پیش میبرید، امیدوارم به قسمت‌های پیشرفته و حساس‌تر هم برسیم، همچنین تجربیات احتمالی خودتون رو هم دخیل در توضیحات کنید تا اهمیت مبحث مطروحه دو چندان بشه!
ممنونم.

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳۹۲/۰۹/۱۳ ۱۴:۳۷

سلام، اگه میشه جای دیگه ای آپلود کنید، لینک خرابه! ممنونم/.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۹/۱۳ ۱۴:۳۹

تکمیل شده این مبحث را در برچسب refactoring در طی 14 قسمت می‌توانید پیگیری کنید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۹/۱۳ ۱۴:۵۶

- 2 سال قبل آپلود شده بوده.
- از اینجا دریافت کنید:

FxCop10.7z

عنوان: **آشنایی با Refactoring - قسمت 2**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۳ ۰۰:۰۵:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

قسمت دوم آشنایی با Refactoring به معرفی روش « **استخراج متدها** » اختصاص دارد. این نوع Refactoring بسیار ساده بوده و مزایای بسیاری را به همراه دارد؛ منجمله:

- بالا بردن خوانایی کد؛ از این جهت که منطق طولانی یک متد به متدهای کوچکتری با نام‌های مفهوم شکسته می‌شود.
- به این ترتیب نیاز به مستند سازی کدها نیز بسیار کاهش خواهد یافت. بنابراین در یک متد، هر جایی که نیاز به نوشتن کامنت وجود داشت، یعنی باید همینجا آن قسمت را جدا کرده و در متد دیگری که نام آن، همان خلاصه کامنت مورد نظر است، قرار داد.
- این نوع جدا سازی منطق‌های پیاده سازی قسمت‌های مختلف یک متد، در آینده نگهداری کد نهایی را نیز ساده‌تر کرده و انجام تغییرات بر روی آن را نیز تسهیل می‌بخشد؛ زیرا اینبار بجای هراس از دستکاری یک متد طولانی، با چند متد کوچک و مشخص سروکار داریم.

برای نمونه به مثال زیر دقت کنید:

```
using System.Collections.Generic;

namespace Refactoring.Day2.ExtractMethod.Before
{
    public class Receipt
    {
        private IList<decimal> Discounts { get; set; }
        private IList<decimal> ItemTotals { get; set; }

        public decimal CalculateGrandTotal()
        {
            // Calculate SubTotal
            decimal subTotal = 0m;
            foreach (decimal itemTotal in ItemTotals)
                subTotal += itemTotal;

            // Calculate Discounts
            if (Discounts.Count > 0)
            {
                foreach (decimal discount in Discounts)
                    subTotal -= discount;
            }

            // Calculate Tax
            decimal tax = subTotal * 0.065m;
            subTotal += tax;

            return subTotal;
        }
    }
}
```

همانطور که از کامنت‌های داخل متد CalculateGrandTotal مشخص است، این متد سه کار مختلف را انجام می‌دهد؛ جمع اعداد، اعمال تخفیف، اعمال مالیات و نهایتا یک نتیجه را باز می‌گرداند. بنابراین بهتر است هر عمل را به یک متد جداگانه و مشخص منتقل کرده و کامنت‌های ذکر شده را نیز حذف کنیم. نام یک متد باید به اندازه‌ی کافی مشخص و مفهوم باشد و آنچنان نیازی به مستندات خاصی نداشته باشد:

```
using System.Collections.Generic;

namespace Refactoring.Day2.ExtractMethod.After
{
    public class Receipt
```

```
    {
        private IList<decimal> Discounts { get; set; }
        private IList<decimal> ItemTotals { get; set; }

        public decimal CalculateGrandTotal()
        {
            decimal subTotal = CalculateSubTotal();
            subTotal = CalculateDiscounts(subTotal);
            subTotal = CalculateTax(subTotal);
            return subTotal;
        }

        private decimal CalculateTax(decimal subTotal)
        {
            decimal tax = subTotal * 0.065m;
            subTotal += tax;
            return subTotal;
        }

        private decimal CalculateDiscounts(decimal subTotal)
        {
            if (Discounts.Count > 0)
            {
                foreach (decimal discount in Discounts)
                    subTotal -= discount;
            }
            return subTotal;
        }

        private decimal CalculateSubTotal()
        {
            decimal subTotal = 0m;
            foreach (decimal itemTotal in ItemTotals)
                subTotal += itemTotal;
            return subTotal;
        }
    }
}
```

بهتر شد! عملکرد کد نهایی، تغییری نکرده اما کیفیت کد ما بهبود یافته است (همان مفهوم و معنای Refactoring). خوانایی کد افزایش یافته است. نیاز به کامنت نویسی به شدت کاهش پیدا کرده و از همه مهم‌تر، اعمال مختلف، در متدهای خاص آن‌ها قرار گرفته‌اند.
به همین جهت اگر حین کد نویسی، به یک متد طولانی برخوردید (این مورد بسیار شایع است)، در ابتدا حداقل کاری را که جهت بهبود کیفیت آن می‌توانید انجام دهید، «استخراج متدها» است.

ابزارهای کمکی جهت پیاده سازی روش «استخراج متدها»:

- ابزار Refactoring توکار ویژوال استودیو پس از انتخاب یک قطعه کد و سپس کلیک راست و انتخاب گزینه‌ی Refactor-> Extract method، این عملیات را به خوبی می‌تواند مدیریت کند و در وقت شما صرفه جویی خواهد کرد.
- افزونه‌های ReSharper و همچنین CodeRush نیز چنین قابلیتی را ارائه می‌دهند؛ البته توانمندی‌های آن‌ها از ابزار توکار یاد شده بیشتر است. برای مثال اگر در میانه کد شما جایی return وجود داشته باشد، گزینه‌ی Extract method ویژوال استودیو کار نخواهد کرد. اما سایر ابزارهای یاده شده به خوبی از پس این موارد و سایر موارد پیشرفته‌تر بر می‌آیند.

نتیجه گیری:
نوشتن کامنت، داخل بدنه‌ی یک متد مزموم است؛ حداقل به دو دلیل:
- ابزارهای خودکار مستند سازی از روی کامنت‌های نوشته شده، از این نوع کامنت‌ها صرفنظر خواهند کرد و در کتابخانه‌ی شما مدفون خواهند شد (یک کار بی‌حاصل).
- وجود کامنت در داخل بدنه‌ی یک متد، نمود آشکار ضعف شما در کپسوله سازی منطق مرتبط با آن قسمت است.

و ... «لطفا» این نوع پیاده سازی‌ها را خارج از فایل code behind هر نوع برنامه‌ی winform/wpf/asp.net و غیره قرار دهید. تا حد امکان سعی کنید این مکان‌ها، استفاده کننده‌ی «نهایی» منطق‌های پیاده سازی شده توسط کلاس‌های دیگر باشند؛ نه اینکه خودشان

محل اصلی قرارگیری و ابتدای تعریف منطق‌های مورد نیاز قسمت‌های مختلف همان فرم مورد نظر باشند. «لطفا» یک فرم درست نکنید با 3000 سطر کد که در قسمت code behind آن قرار گرفته‌اند. code behind را محل «نهایی» ارائه کار قرار دهید؛ نه نقطه‌ی آغاز تعریف منطق‌های پیاده سازی کار. این برنامه نویسی چندلایه که از آن صحبت می‌شود، فقط مرتبط با کار با بانک‌های اطلاعاتی نیست. در همین مثال، کدهای فرم برنامه، باید نقطه‌ی نهایی نمایش عملیات محاسبه مالیات باشند؛ نه اینکه همانجا دوستانه یک قسمت مالیات حساب شود، یک قسمت تخفیف، یک قسمت جمع بزند، همانجا هم نمایش بدهد! بعد از یک هفته می‌بینید که code behind فرم در حال انفجار است! شده 3000 سطر! بعد هم سؤال می‌پرسید که چرا اینقدر میل به «بازنویسی» سیستم این اطراف زیاد است! برنامه نویس حاضر است کل کار را از صفر بنویسد، بجای اینکه با این شاهکار بخواهد سرو کله بزند! هر چند یکی از روش‌های برخورد با این نوع کدها جهت کاهش هراس نگهداری آن‌ها، شروع به Refactoring است.

عنوان: **آشنایی با Refactoring - قسمت 3**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۴ ۰۰:۰۵:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

قسمت سوم آشنایی با Refactoring در حقیقت به تکمیل قسمت قبل که در مورد «استخراج متدها» بود اختصاص دارد و به مبحث «استخراج یک یا چند کلاس از متدها» یا Extract Method Object اختصاص دارد.
زمانیکه کار «استخراج متدها» را شروع می‌کنیم، پس از مدتی به علت بالا رفتن تعداد متدهای کلاس جاری، به آنچنان شکل و شمایل خوشایند و زیبایی دست پیدا نخواهیم کرد. همچنین اینبار بجای متدی طولانی، با کلاسی طولانی سروکار خواهیم داشت. در این حالت بهتر است از متدهای استخراج شده مرتبط، یک یا چند کلاس جدید تهیه کنیم. به همین جهت به آن Extract Method Object می‌گویند.
بنابراین مرحله‌ی اول کار با یک قطعه کد با کیفیت پایین، استخراج متدهایی کوچک‌تر و مشخص‌تر، از متدهای طولانی آن است. مرحله بعد، کپسوله کردن این متدها در کلاس‌های مجزا و مرتبط با آن‌ها می‌باشد (logic segregation). بر این اساس که یکی از اصول ابتدایی شیء گرایی این مورد است: هر کلاس باید یک کار را انجام دهد (Single Responsibility Principle).
بنابراین اینبار از نتیجه‌ی حاصل از مرحله‌ی قبل شروع می‌کنیم و عملیات Refactoring را ادامه خواهیم داد:

```
using System.Collections.Generic;

namespace Refactoring.Day2.ExtractMethod.After
{
    public class Receipt
    {
        private IList<decimal> _discounts;
        private IList<decimal> _itemTotals;

        public decimal CalculateGrandTotal()
        {
            _discounts = new List<decimal> { 0.1m };
            _itemTotals = new List<decimal> { 100m, 200m };

            decimal subTotal = CalculateSubTotal();
            subTotal = CalculateDiscounts(subTotal);
            subTotal = CalculateTax(subTotal);
            return subTotal;
        }

        private decimal CalculateTax(decimal subTotal)
        {
            decimal tax = subTotal * 0.065m;
            subTotal += tax;
            return subTotal;
        }

        private decimal CalculateDiscounts(decimal subTotal)
        {
            if (_discounts.Count > 0)
            {
                foreach (decimal discount in _discounts)
                    subTotal -= discount;
            }
            return subTotal;
        }

        private decimal CalculateSubTotal()
        {
            decimal subTotal = 0m;
            foreach (decimal itemTotal in _itemTotals)
                subTotal += itemTotal;
            return subTotal;
        }
    }
}
```

این مثال، همان نمونه‌ی کامل شده‌ی کد نهایی قسمت قبل است. چند اصلاح هم در آن انجام شده است تا قابل استفاده و مفهوم‌تر شود. عموما متغیرهای خصوصی یک کلاس را به صورت فیلد تعریف می‌کنند؛ نه خاصیت‌های set و get دار. همچنین مثال قبل نیاز به مقدار دهی این فیلدها را هم داشت که در اینجا انجام شده.
اکنون می‌خواهیم وضعیت این کلاس را بهبود ببخشیم و آن‌را از این حالت بسته خارج کنیم:

```
using System.Collections.Generic;

namespace Refactoring.Day3.ExtractMethodObject.After
{
    public class Receipt
    {
        public IList<decimal> Discounts { get; set; }
        public decimal Tax { get; set; }
        public IList<decimal> ItemTotals { get; set; }

        public decimal CalculateGrandTotal()
        {
            return new ReceiptCalculator(this).CalculateGrandTotal();
        }
    }
}
```

```
using System.Collections.Generic;

namespace Refactoring.Day3.ExtractMethodObject.After
{
    public class ReceiptCalculator
    {
        Receipt _receipt;

        public ReceiptCalculator(Receipt receipt)
        {
            _receipt = receipt;
        }

        public decimal CalculateGrandTotal()
        {
            decimal subTotal = CalculateSubTotal();
            subTotal = CalculateDiscounts(subTotal);
            subTotal = CalculateTax(subTotal);
            return subTotal;
        }

        private decimal CalculateTax(decimal subTotal)
        {
            decimal tax = subTotal * _receipt.Tax;
            subTotal += tax;
            return subTotal;
        }

        private decimal CalculateDiscounts(decimal subTotal)
        {
            if (_receipt.Discounts.Count > 0)
            {
                foreach (decimal discount in _receipt.Discounts)
                    subTotal -= discount;
            }
            return subTotal;
        }

        private decimal CalculateSubTotal()
        {
            decimal subTotal = 0m;
            foreach (decimal itemTotal in _receipt.ItemTotals)
                subTotal += itemTotal;
            return subTotal;
        }
    }
}
```

بهبودهای حاصل شده نسبت به نگارش قبلی آن:
در این مثال کل عملیات محاسباتی به یک کلاس دیگر منتقل شده است. کلاس ReceiptCalculator شیءایی از نوع Receipt را در سازنده خود دریافت کرده و سپس محاسبات لازم را بر روی آن انجام می‌دهد. همچنین فیلدهای محلی آن تبدیل به خواصی عمومی و قابل تغییر شده‌اند. در نگارش قبلی، تخفیف‌ها و مالیات و نحوه‌ی محاسبات به صورت محلی و در همان کلاس تعریف شده بودند. به عبارت دیگر با کدی سروکار داشتیم که قابلیت استفاده مجدد نداشت. نمی‌توانست نوع‌های مختلفی از Receipt را بپذیرد. نمی‌شد از آن در برنامه‌ای دیگر هم استفاده کرد. تازه شروع کرده بودیم به جدا سازی منطق‌های قسمت‌های مختلف محاسبات یک متد اولیه طولانی. همچنین اکنون کلاس ReceiptCalculator تنها عهده دار انجام یک عملیات مشخص است.
البته اگر به کلاس ReceiptCalculator قسمت سوم و کلاس Receipt قسمت دوم دقت کنیم، شاید آنچنان تفاوتی را نتوان حس کرد. اما واقعیت این است که کلاس Receipt قسمت دوم، تنها یک پیش نمایش مختصری از صدها متد موجود در آن است.

نظرات خوانندگان

نویسنده: shahin kiassat
تاریخ: ۱۳۹۰/۰۷/۱۵ ۱۹:۰۵:۲۵

سلام.
ممنون از آموزش های بسیار مفیدتون.
آقای نصیری اگر در این مثال نیاز به ذخیره ی اطلاعات Reciept در دیتابیس باشد باید این وظیفه به کلاس دیگری در لایه ی دسترسی به داده ها سپرد ؟
اگر ممکن است قدری در این رابطه توضیح دهید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۵ ۲۱:۳۶:۱۳

سلام؛ بله. البته در این حالت Receipt repository (الگوی مخزن)، اطلاعات نهایی حاصل از عملیات این کلاس رو می‌تونه جداگانه در کلاس خاص خودش، دریافت و ثبت کنه. این کلاس به همین صورت که هست باید باقی بمونه و اصل‌های مرتبط با جدا سازی منطق‌ها رو نباید نقض نکنه.

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۵/۲۴ ۲۱:۲۳

سلام ،
این Receipt در Project.Domain قرار می‌گیرد ؟ در واقع همان موجودیت ما هست ؟
من تصور می‌کردم همه‌ی منطق تجاری را باید در Service Layer پیاده سازی کرد ، اما در بعضی سورس‌ها و چارچوب‌ها (مثل sharp-lite) دیدم که متدهای محاسباتی مثل مجموع هزینه‌های مربوط به یک سفارش را در همان موجودیت قرار می‌دهند :

```
public class OrderLineItem : Entity
    {
        /// <summary>
        /// many-to-one from OrderLineItem to Order
        /// </summary>
        public virtual Order Order { get; set; }

        /// <summary>
        /// Money is a component, not a separate entity; i.e., the OrderLineItems table will have
        /// column for the amount
        /// </summary>
        public virtual Money Price { get; set; }

        /// <summary>
        /// many-to-one from OrderLineItem to Product
        /// </summary>
        public virtual Product Product { get; set; }

        public virtual int Quantity { get; set; }

        /// <summary>
        /// Example of adding domain business logic to entity
        /// </summary>
        public virtual Money GetTotal() {
            return new Money(Price.Amount * Quantity);
        }
    }
```

ممنون می‌شم قدری در این باره توضیح بدید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۲۴ ۲۱:۳۹

EF Code first هم برای معرفی فیلدهای محاسباتی ویژگی NotMapped را دارد. این فیلدها در بانک اطلاعاتی معادلی ندارند و صرفا

جهت اعمال به UI، به کلاس اضافه می‌شن.
البته منطق آنچنانی ندارند و در حد calculated field در sql server به آن نگاه کنید. عموما جمع و ضرب روی فیلدها است یا تبدیل تاریخ و در این حد ساده است. بیشتر از این بود باید از کلاس مدل خارج شود و به لایه سرویس سپرده شود.
و یا روش بهتر تعریف آن‌ها انتقال این موارد به ViewModel است. مدل را باید از این نوع خواص خالی کرد. ViewModel بهتر است محل قرارگیری فیلدهای محاسباتی از این دست باشد که صرفا کاربرد سمت UI دارند و در بانک اطلاعاتی معادل متناظری ندارند.

عنوان: **آشنایی با Refactoring - قسمت 4**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۵ ۰۰:۰۳:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

قسمت چهار آشنایی با Refactoring به معرفی روش « انتقال متدها » اختصاص دارد؛ انتقال متدها به مکانی بهتر. برای نمونه به کلاس‌های زیر پیش از انجام عمل Refactoring دقت کنید:

```
namespace Refactoring.Day3.MoveMethod.Before
{
    public class BankAccount
    {
        public int AccountAge { get; private set; }
        public int CreditScore { get; private set; }

        public BankAccount(int accountAge, int creditScore)
        {
            AccountAge = accountAge;
            CreditScore = creditScore;
        }
    }
}
```

```
namespace Refactoring.Day3.MoveMethod.Before
{
    public class AccountInterest
    {
        public BankAccount Account { get; private set; }

        public AccountInterest(BankAccount account)
        {
            Account = account;
        }

        public double InterestRate
        {
            get { return CalculateInterestRate(); }
        }

        public bool IntroductoryRate
        {
            get { return CalculateInterestRate() < 0.05; }
        }

        public double CalculateInterestRate()
        {
            if (Account.CreditScore > 800)
                return 0.02;

            if (Account.AccountAge > 10)
                return 0.03;

            return 0.05;
        }
    }
}
```

قسمت مورد نظر ما در اینجا، متد AccountInterest.CalculateInterest است. کلاس AccountInterest مرتبا نیاز دارد که از

اطلاعات فیلدها و خواص کلاس BankAccount استفاده کند (نکته تشخیص نیاز به این نوع Refactoring). بنابراین بهتر است که این متد را به همان کلاس تعریف کننده‌ی فیلدها و خواص اصلی آن انتقال داد. پس از این نقل و انتقالات خواهیم داشت:

```
namespace Refactoring.Day3.MoveMethod.After
{
    public class BankAccount
    {
        public int AccountAge { get; private set; }
        public int CreditScore { get; private set; }

        public BankAccount(int accountAge, int creditScore)
        {
            AccountAge = accountAge;
            CreditScore = creditScore;
        }

        public double CalculateInterestRate()
        {
            if (CreditScore > 800)
                return 0.02;

            if (AccountAge > 10)
                return 0.03;

            return 0.05;
        }
    }
}
```

```
namespace Refactoring.Day3.MoveMethod.After
{
    public class AccountInterest
    {
        public BankAccount Account { get; private set; }

        public AccountInterest(BankAccount account)
        {
            Account = account;
        }

        public double InterestRate
        {
            get { return Account.CalculateInterestRate(); }
        }

        public bool IntroductoryRate
        {
            get { return Account.CalculateInterestRate() < 0.05; }
        }
    }
}
```

به همین سادگی!

یک مثال دیگر:

در ادامه به دو کلاس خودرو و موتور خودروی زیر دقت کنید:

```
namespace Refactoring.Day4.MoveMethod.Ex2.Before
{
    public class CarEngine
    {
        public float LitersPerCylinder { set; get; }
        public int NumCylinders { set; get; }
```

```
        public CarEngine(int numCylinders, float litersPerCylinder)
        {
            NumCylinders = numCylinders;
            LitersPerCylinder = litersPerCylinder;
        }
    }
}
```

```
namespace Refactoring.Day4.MoveMethod.Ex2.Before
{
    public class Car
    {
        public CarEngine Engine { get; private set; }

        public Car(CarEngine engine)
        {
            Engine = engine;
        }

        public float ComputeEngineVolume()
        {
            return Engine.LitersPerCylinder * Engine.NumCylinders;
        }
    }
}
```

در اینجا هم متد Car.ComputeEngineVolume چندین‌بار به اطلاعات داخلی کلاس CarEngine دسترسی داشته است؛ بنابراین بهتر است این متد را به جایی منتقل کرد که واقعا به آن تعلق دارد:

```
namespace Refactoring.Day4.MoveMethod.Ex2.After
{
    public class CarEngine
    {
        public float LitersPerCylinder { set; get; }
        public int NumCylinders { set; get; }

        public CarEngine(int numCylinders, float litersPerCylinder)
        {
            NumCylinders = numCylinders;
            LitersPerCylinder = litersPerCylinder;
        }

        public float ComputeEngineVolume()
        {
            return LitersPerCylinder * NumCylinders;
        }
    }
}
```

```
namespace Refactoring.Day4.MoveMethod.Ex2.After
{
    public class Car
    {
        public CarEngine Engine { get; private set; }

        public Car(CarEngine engine)
        {
            Engine = engine;
        }
    }
}
```

بهبودهای حاصل شده:

یکی دیگر از اصول برنامه نویسی شیء گرا " Tell, Don't Ask " است؛ که در مثال‌های فوق محقق شده. به این معنا که: در برنامه نویسی رویه‌ای متداول، اطلاعات از قسمت‌های مختلف کد جاری جهت انجام عملی دریافت می‌شود. اما در برنامه نویسی شیء گرا به اشیاء گفته می‌شود تا کاری را انجام دهند؛ نه اینکه از آن‌ها وضعیت یا اطلاعات داخلی‌اشان را جهت اخذ تصمیمی دریافت کنیم.
به وضوح، متد Car.ComputeEngineVolume پیش از Refactoring ، اصل کپسوله سازی اطلاعات کلاس CarEngine را زیر سؤال برده است. بنابراین به اشیاء بگوئید که چکار کنند و اجازه دهید تا خودشان در مورد نحوه‌ی انجام آن، تصمیم گیرنده نهایی باشند.

نظرات خوانندگان

نویسنده: Ebrahim Byagowi
تاریخ: ۱۳۹۰/۰۷/۱۵ ۱۲:۴۷:۰۷

عالی بود :)

نویسنده: A.Karimi
تاریخ: ۱۳۹۰/۰۷/۱۵ ۲۳:۰۰:۳۵

با این اوصاف یعنی الگوی Active Record تنها الگوی شی‌گرا است؟ و اینکه مثلاً ما یک Entity از یک Domain را به یک متد Business جهت اعمال تغییرات و یا انجام‌کارهایی خاص می‌دهیم از نظر شی‌گرایی غلط است؟

در این زمینه هرچه گشتم تنها صحبتی که پیدا می‌کنم این است که هر دو راه صحیح است. برای مثال چه بگویید:
()Person.PaySalary
و یا
(SalaryBusiness.PaySalary(Person
هر دو صحیح است!
یک مثال دیگر Attached Property ها در WPF است.

در این زمینه باید به کجا رجوع کرد؟

نویسنده: A.Karimi
تاریخ: ۱۳۹۰/۰۷/۱۵ ۲۳:۰۲:۲۲

البته منظور از «تنها الگوی شی گرا»، فقط در زمینه کار با ORM ها بود. و البته ممکن است الگوهای شبیه دیگری هم باشد. منظورم دو حالت اساسی است که یکی شبیه Active Record و دیگری آنگونه که توضیح دادم است.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۶ ۰۰:۴۲:۱۴

Active record جزو الگوهای مطلوب به شمار نمی‌رود. در این مورد یک سری مقاله مفصل اینجا هست:
ORM anti-patterns - Part 1: Active Record
باز هم بگردید اکثرا ضد روش Active record هستند.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۶ ۰۱:۰۰:۴۶

Attached Properties در WPF در حقیقت یک نوع تزریق شیء به شیء مورد است. شما خواص شیءایی را به شیء دیگر تزریق می‌کنید. مثلا یک دکمه دارید، سپس Canvas.Top یا Grid.Row به آن متصل می‌کنید، علت هم این است که اگر قرار بود از روز اول برای یک دکمه تمام این خواص را تعریف کنند، باید بی‌نهایت خاصیت تعریف می‌کردند؛ چون WPF قابل توسعه است و می‌شود layout panel سفارشی هم طراحی کرد. البته این تازه یک مورد از کاربردهای این مبحث است.
به صورت خلاصه، Attached Properties ، کپسوله سازی شیء جاری را زیر سؤال نمی‌برد. (موضوع اصلی بحث جاری)
هر چند با استفاده از Attached Properties می‌توان به تمام خواص و کلیه رویدادهای شیء تزریق شده به آن هم دسترسی پیدا کرد. اینجا هم باز هم برخلاف بحث جاری ما نیست؛ چون اساسا این شیء الحاقی یا ضمیمه شده، نهایتا با شیء جاری از دید سیستم یکپارچه به نظر می‌رسد. این تزریق هم به همین دلیل صورت گرفته. بنابراین در اینجا هم دسترسی یا تغییر خواص شیء ضمیمه شده، خلاف مقررات شیءگرایی و کپسوله سازی نیست. چون ما در اساس داریم راجع به مثلا یک دکمه صحبت می‌کنیم. اگر خاصیتی هم به آن تزریق شده باز هم نهایتا جزو خواص همان دکمه در نظر گرفته می‌شود.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۶ ۰۱:۲۹:۳۴

در مورد POCO یا مثالی که زدید:
حین کار با ORMs ، کلاس‌های تعریف شده نمی‌دادند که آیا ذخیره شده‌اند یا اینکه چگونه ذخیره شده‌اند یا حتی چگونه به بانک اطلاعاتی نگاشت شده‌اند یا نشده‌اند. کل عملیات transperant است (persistence ignorance). همچنین این نوع کلاس‌ها فقط اهداف display / reference دارند و نه بیشتر. بحث کلاس‌های مثال فوق متفاوت است. ما در مورد ده‌ها و صدها متد موجود در آن‌ها بحث کردیم. این کلاس‌ها هیچکدام در رده تعریف POCO یا Plain old .Net classes قرار نمی‌گیرند.

نویسنده: A.Karimi
تاریخ: ۱۳۹۰/۰۷/۱۶ ۰۲:۳۷:۳۴

من از کاراکتر LTR کردن استفاده کردم که متاسفانه متن را به هم ریخت. البته در Notepad درست بود!

در هر صورت متنی که بعد از کد SalaryBusiness آمده به شکل زیر است:

...
هر دو صحیح است!یک مثال دیگر Attached Property ها در WPF است.در این زمینه باید به کجا رجوع کرد؟

نویسنده: A.Karimi
تاریخ: ۱۳۹۰/۰۷/۱۶ ۰۲:۵۴:۲۲

در مورد Active Record موافقم و هرگز هم از این الگو استفاده نکرده‌ام. در مورد Attached Property هم با مکانیزم آن آشنا هستم و استفاده‌های متفاوت تری از Layouting همچون اعمال Security توسط AP بر روی یک المان را تجربه کردم که انصافاً طراحی هوشمندانه AP را برایم آشکار کرد اما تصور می‌کنم برخی از موارد مانند hiding و حتی overriding در مورد AP و Dependency Property ها رعایت نمی‌شود (شاید انتظار overriding درست نباشد جون AP و DP ففط مخصوص ذخیره‌سازی هستند). مثلا شما می‌توانید با دستور GetValue مقدار یک خصیصه از جنس DP یک شی را در خارج از آن شی به دست آورید در حالی که اگر بخواهید از خاصیت Binding استفاده کنید استفاده از DP توصیه شده. البته امتحان نکرده‌ام اما فکر می‌کنم با protected کردن متغییرهای static مربوط به DP به این حالت دست یافت اما فکر می‌کنم بازهم از سطح private محروم می‌مانیم.

در نهایت در مورد Active Record و Entity ها صحبتهایتان را با این جمله کامل کردید که: «این نوع کلاس‌ها فقط اهداف display / reference دارند و نه بیشتر ... این کلاس‌ها (کلاسهایی که در پست تعریف شده بود/م) هیچکدام در رده تعریف POCO یا Plain old .Net classes قرار نمی‌گیرند.»

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۶ ۰۸:۲۰:۰۳

ادیتور بلاگر به کاراکترهایی که در XML باید escape شوند حساس است. اگر در متن ارسالی وجود داشته باشد، حذفشان می‌کند.

عنوان: آشنایی با Refactoring - قسمت 5
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۶ ۰۰:۰۸:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

یکی دیگر از تکنیک‌های Refactoring بسیار متداول، «حذف کدهای تکراری» است. کدهای تکراری هم عموما حاصل بی‌حوصلگی یا تنبلی هستند و برنامه نویس نیاز دارد در زمانی کوتاه، حجم قابل توجهی کد تولید کند؛ که نتیجه‌اش مثلا به صورت زیر خواهد شد:

```
using System;

namespace Refactoring.Day4.RemoveDuplication.Before
{
    public class PersonalRecord
    {
        public DateTime DateArchived { get; private set; }
        public bool Archived { get; private set; }

        public void ArchiveRecord()
        {
            Archived = true;
            DateArchived = DateTime.Now;
        }

        public void CloseRecord()
        {
            Archived = true;
            DateArchived = DateTime.Now;
        }
    }
}
```

Refactoring ما هم در اینجا عموما به انتقال کدهای تکراری به یک متد مشترک خلاصه می‌شود:

```
using System;

namespace Refactoring.Day4.RemoveDuplication.After
{
    public class PersonalRecord
    {
        public DateTime DateArchived { get; private set; }
        public bool Archived { get; private set; }

        public void ArchiveRecord()
        {
            switchToArchived();
        }

        public void CloseRecord()
        {
            switchToArchived();
        }

        private void switchToArchived()
        {
            Archived = true;
            DateArchived = DateTime.Now;
        }
    }
}
```

اهمیت حذف کدهای تکراری:

- اگر باگی در این کدهای تکراری یافت شود، همه را در سراسر برنامه باید اصلاح کنید (زیرا هم اکنون همانند یک ویروس به سراسر برنامه سرایت کرده‌است) و احتمال فراموشی یک قسمت هم ممکن است وجود داشته باشد.
- اگر نیاز به بهبود یا تغییری در این قسمت‌های تکراری وجود داشت، باز هم کار برنامه نویس به شدت زیاد خواهد بود.

ابزارهای کمکی:

واقعیت این است که در قطعه کد کوتاه فوق، یافتن قسمت‌های تکراری بسیار ساده بوده و با یک نگاه قابل تشخیص است؛ اما در برنامه‌های بزرگ خیر. به همین منظور تعداد قابل توجهی برنامه‌ی کمکی جهت تشخیص کدهای تکراری پروژه‌ها تابحال تولید شده‌اند؛ مانند Clone detective ، CopyPasteKiller و غیره.

علاوه بر این‌ها نگارش بعدی ویژوال استودیو (نگارش 11) حاوی ابزار Code Clone Detection توکاری است (+) و همچنین یک لیست قابل توجه دیگر را در این زمینه در این پرسش و پاسخ می‌توانید بیابید: (+)

نظرات خوانندگان

نویسنده: سی شارپ 2012
تاریخ: ۱۳۹۲/۰۸/۲۰ ۱۰:۱۱

سلام وخسته نباشید
آیا نرم افزار Clone detective برای نسخه vs 2012 وجود دارد یا خیر؟ من نرم افزار فوق را در vs 2012 نصب کردم ولی تولبار آن نمایش داده نمیشود
لطفا راهنمایی کنید
مطالب Refactoring بسیار مفید بود تشکر

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۸/۲۰ ۱۰:۲۳

- این مسایل رو در پشتیبانی خود آن پروژه مطرح کنید.
- ضمنا (امروز، بعد از 2 سال) نیازی به این افزونه ندارید. خود VS.NET 2012 به صورت توکار حاوی Code Clone Analysis است.

عنوان: **آشنایی با Refactoring - قسمت 6**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۷ ۰۰:۰۸:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

در ادامه بحث «حذف کدهای تکراری»، روش Refactoring دیگری به نام " Extract Superclass " وجود دارد که البته در بین برنامه نویس‌های دات نت به نام Base class بیشتر مشهور است تا Superclass. هدف آن هم انتقال کدهای تکراری بین چند کلاس، به یک کلاس پایه و سپس ارث بری از آن می‌باشد.

یک مثال:
در WPF و Silverlight جهت مطلع سازی رابط کاربری از تغییرات حاصل شده در مقادیر داده‌ها، نیاز است کلاس مورد نظر، اینترفیس INotifyPropertyChanged را پیاده سازی کند:

```
using System.ComponentModel;

namespace Refactoring.Day6.ExtractSuperclass.Before
{
    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));
        }
    }
}
```

و نکته‌ی مهم این است که اگر 100 کلاس هم داشته باشید، باید این کدهای تکراری اجباری مرتبط با raisePropertyChanged را در آن‌ها قرار دهید. به همین جهت مرسوم است برای کاهش حجم کدهای تکرای، قسمت‌های تکراری کد فوق را در یک کلاس پایه قرار می‌دهند:

```
using System.ComponentModel;

namespace Refactoring.Day6.ExtractSuperclass.After
{
    public class ViewModelBase : INotifyPropertyChanged
    {
        public event PropertyChangedEventHandler PropertyChanged;
        protected void RaisePropertyChanged(string propertyName)
        {
            var handler = PropertyChanged;
            if (handler == null) return;
            handler(this, new PropertyChangedEventArgs(propertyName));
        }
```

```
    }
}
```

و سپس از آن ارث بری می‌کنند:

```
namespace Refactoring.Day6.ExtractSuperclass.After
{
    public class User : ViewModelBase
    {
        string _name;
        public string Name
        {
            get { return _name; }
            set
            {
                if (_name == value) return;
                _name = value;
                RaisePropertyChanged("Name");
            }
        }
    }
}
```

به این ترتیب این کلاس پایه در ده‌ها و صدها کلاس قابل استفاده خواهد بود، بدون اینکه مجبور شویم مرتبا یک سری کد تکراری «اجباری» را copy/paste کنیم.

مثالی دیگر:

اگر با ORM های Code first کار کنید، نیاز است تا ابتدا طراحی کار توسط کلاس‌های ساده دات نتی انجام شود؛ که اصطلاحا به آن‌ها POCO یا Plain old CLR objects یا Plain old .NET Classes هم گفته می‌شود. در بین این کلاس‌ها، متداول است که یک سری از خصوصیات، تکراری و مشترک باشد؛ مثلا تمام کلاس‌ها تاریخ ثبت رکورد را هم داشته باشند به همراه نام کاربر و مشخصاتی از این دست. اینجا هم برای حذف کدهای تکراری، یک Base class طراحی می‌شود: (+)

نظرات خوانندگان

نویسنده: A.Karimi
تاریخ: ۱۳۹۰/۰۷/۱۸ ۰۰:۱۶:۲۰

با عرض پوزش از اینکه مطلبی که می‌نویسم به پست بی ربط است. مایل بودم نظر شما را در مورد یک سوال، در صورتی که با RIA آشنایی دارید، بدانم که در stackoverflow مطرح کردم و پاسخی دریافت نکردم! سوال به طور خلاصه این است که «وقتی ما می‌خواهیم یک DTO پیچیده را به سمت سرور انتقال دهیم و در یک Round trip عملیات مورد نظرمان انجام شود (مثلاً یک Bulk insert یا چیزی شبیه آن) آیا در RIA راهی برای اینکار وجود دارد؟ یا اینکه باید از WCF Services سنتی در کنار RIA استفاده کنیم؟"

لینک StackOverflow:
http://stackoverflow.com/questions/7632337/send-custom-complex-objects-to-silverlight-ria-services

ممنون.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۸ ۰۹:۱۹:۰۵

علاوه بر مطالبی که اونطرف نوشتم، فورم اصلی RIA Services اینجا است: [^] . بگردید از این مورد زیاد دارد.

نویسنده: A.Karimi
تاریخ: ۱۳۹۰/۰۷/۱۸ ۱۹:۱۸:۰۹

از پیگیری شما متشکرم. نمی‌دانم شاید بی دقتی کردم اما گشت و گذار در این زمینه داشتم و متاسفانه چیزی پیدا نکردم. با استفاده از [Composition] مشکل متدهای اضافه Insert حل شد. اما هنوز برای Expose کردن شی اصی نیاز به یک متد Get یا Query دارم. آیا راحی وجود دارد که بدون نوشتن یک متد Get یا Query یک کلاس را با استفاده از RIA به سمت کلاینت Expose کرد؟ انتظار من یک Attribute برای DomainService بود ولی فعلاً چیزی پیدا نکرده‌ام.

ممنوم.

نویسنده: A.Karimi
تاریخ: ۱۳۹۰/۰۷/۱۸ ۱۹:۲۴:۱۶

البته منظور من از یک شی یا کلاس، یک کلاس است که به صورت دستی ساخته شده و نه کلاس‌های EF یا از این قبیل. امکان Expose کردن آنها به راحتی با استفاده از خصیصه‌ی LinqToEntitiesDomainServiceDescriptionProvider امکان پذیر است. اما در مورد یک Entity دست ساز چیزی نیافتم!

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۸ ۲۲:۵۲:۲۷

این سایت در مورد RIA Services و DTO مطلب زیاد دارد. به مشکل مورد نظر شما هم اشاره کرده؛ در قسمت RIA and DTO - Part 2 : [^]

نویسنده: A.Karimi
تاریخ: ۱۳۹۰/۰۷/۲۲ ۱۹:۳۰:۰۸

ممنون. خیلی کمک کردید. البته این وبلاگهای مفید که اکثراً ف.ی.ل.ت.ر هستند و من به سختی توانستم مروری بکنم که متاسفانه چیزی در مورد Expose کردن Entityها با آن روشی که گفتم نیافتم البته باید با دقت بیشتری مرور کنم.

در هر صورت باز هم ممنون و عذر خواهی به علت بی ربط بودن کامنتها به پست.

عنوان: **آشنایی با Refactoring - قسمت 7**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۸ ۰۰:۰۷:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

یکی دیگر از روش‌های Refactoring ، معرفی کردن یک کلاس بجای پارامترها است. عموما تعریف متدهایی با بیش از 5 پارامتر مزموم است:

```
using System;
using System.Collections.Generic;

namespace Refactoring.Day7.IntroduceParameterObject.Before
{
    public class Registration
    {
        public void Create(string name, DateTime date, DateTime validUntil,
                           IEnumerable<string> courses, decimal credits)
        {
            // do work
        }
    }
}
```

در این حالت بجای تعریف این تعداد بالای پارامترهای مورد نیاز، تمام آن‌ها را تبدیل به یک کلاس کرده و استفاده می‌کنند:

```
using System;
using System.Collections.Generic;

namespace Refactoring.Day7.IntroduceParameterObject.After
{
    public class RegistrationContext
    {
        public string Name {set;get;}
        public DateTime Date {set;get;}
        public DateTime ValidUntil {set;get;}
        public IEnumerable<string> Courses {set;get;}
        public decimal Credits { set; get; }
    }
}
```

```
namespace Refactoring.Day7.IntroduceParameterObject.After
{
    public class Registration
    {
        public void Create(RegistrationContext registrationContext)
        {
            // do work
        }
    }
}
```

یکی از مزایای این روش، منعطف شدن معرفی متدها است؛ به این صورت که اگر نیاز به افزودن پارامتر دیگری باشد، تنها کافی است یک خاصیت جدید به کلاس RegistrationContext اضافه شود و امضای متد Create، ثابت باقی خواهد ماند.

روش دیگر تشخیص نیاز به این نوع Refactoring ، یافتن پارامترهایی هستند که در یک گروه قرار می‌گیرند. برای مثال:

```
public int GetIndex(int pageSize, int pageNumber, ...) { ...
```

همانطور که ملاحظه می‌کنید تعدادی از پارامترها در اینجا با کلمه page شروع شده‌اند. بهتر است این پارامترهای مرتبط را به یک کلاس مجزا به نام Page انتقال داد.

نظرات خوانندگان

نویسنده: Farhad Yazdan-Panah
تاریخ: ۱۳۹۰/۰۷/۱۹ ۱۶:۵۳:۴۳

البته به نظر من در زمانیکه تابع مورد نظر یک گزارش (یا بخش از آن) باشد بهتره که استثنا قائل شد.فقط یک سوال: در حالاتیکه از کنترل هایی مثل ObjectDataSource استفاده بشه و بخواهیم یکی از این توابع (با ورودی جدید) را فراخوانی کنیم باید پیچیدگی زیادی به برنامه اضافه بشه (Serialize , ...). آیا چاره ای وجود دارد؟
ممنون

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۹ ۱۷:۱۴:۰۳

بحث Refactoring در مورد طراحی کارهای شما معنا پیدا می‌کند؛ وگرنه اگر کتابخانه‌ی بسته دیگری، نیازهای خاص خودش را دیکته می‌کند، بدیهی است دست شما آنچنان باز نخواهد بود.
در مورد مطلبی که گفتید، بله می‌شود. در این حالت باید DataObject TypeName مربوط به ObjectDataSource را مشخص کنید: [^]
اگر می‌خواهید واقعا این اصول شیءگرایی را رعایت کنید، بهتر است به ASP.NET MVC کوچ کنید. Model binder آن، خودش به صورت خودکار این موارد را پوشش می‌دهد. نگارش بعدی ASP.NET Webforms هم کمی تا قسمتی از این Model binder رو به ارث برده ولی نه آنچنان که یک strongly typed view رو بتونید باهاش 100 درصد مثل MVC تعریف کنید.
در کل معماری ASP.NET Webforms مربوط به روزهای اول دات نت است و به نظر هم قرار نیست آنچنان تغییری بکند. به همین جهت MVC رو این وسط معرفی کرده‌اند.

نویسنده: Farhad Yazdan-Panah
تاریخ: ۱۳۹۰/۰۷/۱۹ ۲۲:۲۷:۰۸

ممنون از توضیحات.
در مورد جمله "البته به نظر من در زمانیکه تابع مورد نظر یک گزارش (یا بخش از آن) باشد بهتره که استثنا قائل شد." منظور من بخش ها و توابعی هستند که ما برای گزارشات استفاده می کنیم. (استخراج تعداد دانشجویان بر اساس بازه تولد، جنسیت، کلمه کلیدی از نام و .. و ...).
در این گونه موارد چون این تابع فقط یک بار و یک جا استفاده می شود آیا استفاده از این رویه کمی دست و پاگیر نیست؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۱۹ ۲۲:۵۴:۲۸

خیر. اگر کمی با الگوهای MVC ، MVVM و امثال آن کار کنید، تهیه مدل جهت این موارد برای شما عادی خواهد شد. چون مجبورید که این‌ها را با حداقل یک کلاس مدل کنید.

عنوان: **آشنایی با Refactoring - قسمت 8**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۲۱ ۰۰:۴۹:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

یکی از اشتباهاتی که همه‌ی ما کم و بیش به آن دچار هستیم ایجاد کلاس‌هایی هستند که «زیاد می‌دانند». اصطلاحا به آن‌ها God Classes هم می‌گویند و برای نمونه، پسوند یا پیشوند Util دارند. این نوع کلاس‌ها اصل SRP را زیر سؤال می‌برند (Single responsibility principle). برای مثال یک فایل ایجاد می‌شود و داخل آن از انواع و اقسام متدهای «کمکی» کار با دیتابیس تا رسم نمودار تا تبدیل تاریخ میلادی به شمسی و ... در طی بیش از 10 هزار سطر قرار می‌گیرند. یا برای مثال گروه بندی‌های خاصی را در این یک فایل از طریق کامنت‌های نوشته شده برای قسمت‌های مختلف می‌توان یافت. Refactoring مرتبط با این نوع کلاس‌هایی که «زیاد می‌دانند»، تجزیه آن‌ها به کلاس‌های کوچکتر، با تعداد وظیفه‌ی کمتر است.
به عنوان نمونه کلاس CustomerService زیر، دو گروه کار متفاوت را انجام می‌دهد. ثبت و بازیابی اطلاعات ثبت نام یک مشتری و همچنین محاسبات مرتبط با سفارشات مشتری‌ها:

```
using System;
using System.Collections.Generic;

namespace Refactoring.Day8.RemoveGodClasses.Before
{
    public class CustomerService
    {
        public decimal CalculateOrderDiscount(IEnumerable<string> products, string customer)
        {
            // do work
            throw new NotImplementedException();
        }

        public bool CustomerIsValid(string customer, int order)
        {
            // do work
            throw new NotImplementedException();
        }

        public IEnumerable<string> GatherOrderErrors(IEnumerable<string> products, string customer)
        {
            // do work
            throw new NotImplementedException();
        }

        public void Register(string customer)
        {
            // do work
        }

        public void ForgotPassword(string customer)
        {
            // do work
        }
    }
}
```

بهتر است این دو گروه، به دو کلاس مجزا بر اساس وظایفی که دارند، تجزیه شوند. به این ترتیب نگهداری این نوع کلاس‌های کوچکتر در طول زمان ساده‌تر خواهند شد:

```
using System;
using System.Collections.Generic;

namespace Refactoring.Day8.RemoveGodClasses.After
{
    public class CustomerOrderService
```

```
    {
        public decimal CalculateOrderDiscount(IEnumerable<string> products, string customer)
        {
            // do work
            throw new NotImplementedException();
        }

        public bool CustomerIsValid(string customer, int order)
        {
            // do work
            throw new NotImplementedException();
        }

        public IEnumerable<string> GatherOrderErrors(IEnumerable<string> products, string customer)
        {
            // do work
            throw new NotImplementedException();
        }
    }
}
```

```
namespace Refactoring.Day8.RemoveGodClasses.After
{
    public class CustomerRegistrationService
    {
        public void Register(string customer)
        {
            // do work
        }

        public void ForgotPassword(string customer)
        {
            // do work
        }
    }
}
```

عنوان: **آشنایی با Refactoring - قسمت 9**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۲۵ ۲۲:۴۰:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

این قسمت از آشنایی با Refactoring به کاهش cyclomatic complexity اختصاص دارد و خلاصه آن این است: «استفاده از if های تو در تو بیش از سه سطح، مذموم است» به این علت که پیچیدگی کدهای نوشته شده را بالا برده و نگهداری آن‌ها را مشکل می‌کند. برای مثال به شبه کد زیر دقت کنید:

```
if
   if
     if
       if
         do something
       endif
     endif
   endif
 endif
```

که حاصل آن شبیه به نوک یک پیکان (Arrow head) شده است. یک مثال بهتر:

```
namespace Refactoring.Day9.RemoveArrowhead.Before
{
    public class Role
    {
        public string RoleName { set; get; }
        public string UserName { set; get; }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Refactoring.Day9.RemoveArrowhead.Before
{
    public class RoleRepository
    {
        private IList<Role> _rolesList = new List<Role>();

        public IEnumerable<Role> Roles { get { return _rolesList; } }

        public void AddRole(string username, string roleName)
        {
            if (!string.IsNullOrWhiteSpace(roleName))
            {
                if (!string.IsNullOrWhiteSpace(username))
                {
                    if (!IsInRole(username, roleName))
                    {
                        _rolesList.Add(new Role
                        {
                            UserName=username,
                            RoleName=roleName
                        });
                    }
                    else
                    {
                        throw new InvalidOperationException("User is already in this role.");
```

```
                    }
                }
                else
                {
                    throw new ArgumentNullException("username");
                }
            }
            else
            {
                throw new ArgumentNullException("roleName");
            }
        }

        public bool IsInRole(string username, string roleName)
        {
            return _rolesList.Any(x => x.RoleName == roleName && x.UserName == username);
        }
    }
}
```

متد AddRole فوق، نمونه‌ی بارز پیچیدگی بیش از حد حاصل از اعمال if های تو در تو است و ... بسیار متداول. برای حذف این نوک پیکان حاصل از if های تو در تو، از بالاترین سطح شروع کرده و شرط‌ها را برعکس می‌کنیم؛ با این هدف که هر چه سریعتر متد را ترک کرده و خاتمه دهیم:

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Refactoring.Day9.RemoveArrowhead.After
{
    public class RoleRepository
    {
        private IList<Role> _rolesList = new List<Role>();

        public IEnumerable<Role> Roles { get { return _rolesList; } }

        public void AddRole(string username, string roleName)
        {
            if (string.IsNullOrWhiteSpace(roleName))
                throw new ArgumentNullException("roleName");

            if (string.IsNullOrWhiteSpace(username))
                throw new ArgumentNullException("username");

            if (IsInRole(username, roleName))
                throw new InvalidOperationException("User is already in this role.");

            _rolesList.Add(new Role
            {
                UserName = username,
                RoleName = roleName
            });
        }

        public bool IsInRole(string username, string roleName)
        {
            return _rolesList.Any(x => x.RoleName == roleName && x.UserName == username);
        }
    }
}
```

اکنون پس از اعمال این Refactoring ، متد AddRole بسیار خواناتر شده و هدف اصلی آن که اضافه کردن یک شیء به لیست نقش‌ها است، واضح‌تر به نظر می‌رسد. به علاوه اینبار قسمت‌های مختلف متد AddRole، فقط یک کار را انجام می‌دهند و وابستگی‌های آن‌ها به یکدیگر نیز کاهش یافته است.

نظرات خوانندگان

نویسنده: Farhad Yazdan-Panah
تاریخ: ۱۳۹۰/۰۷/۲۵ ۲۳:۴۳:۳۹

نکته جالب تولید کد میانی کمتر و واضحتر نیز هست (به دلیل عمق کمتر درخت تصمیم).

نویسنده: M.Safdel
تاریخ: ۱۳۹۰/۰۷/۲۶ ۱۰:۴۷:۳۲

سری مطالب Refactoring عالی هستن و از شما ممنونم. امیدوارم که همچنان ادامه داشته باشن. احتمالا همه برنامه نویسها مثل خودم خیلی از این روشها را بصورت تجربی می دونن ولی اینکه این روشها در قالبهای خاص ارائه بشن خیلی جالبه

نویسنده: HIWA NAZARI
تاریخ: ۱۳۹۰/۰۷/۲۶ ۱۴:۵۳:۱۴

سلام می بخشید که سوالم رو اینجا میپرسم ولی چاره ای نیست من می خوام تعدادی کنترل رو رو بصورت runtime در Asp.net به صفحه اضافه کنم سپس مقادیرش رو هم بخونم ولی مشکل اینجا ست زمانی که من میخوام به صفحه ای دیگه برم وبرگردم کنترل ها ازدست میرند بهتر بگم میخوام چیزی شبیه به پروفایل facebook باشه

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۲۶ ۱۶:۵۶:۱۰

برای اینکه احتمالا ASP.NET Webforms page life cycle رو رعایت نکردید و الان ViewState صفحه چیزی از وجود کنترل‌های پویای شما نمی‌دونه. مثلا می‌تونید از DynamicControlsPlaceholder استفاده کنید. اگر جزئیات بیشتری نیاز داشتید این مطالب مفید هستند:

How To Perpetuate Dynamic Controls Between Page Views in ASP.NET

Dynamic Web Controls, Postbacks, and View State

Creating Dynamic Data Entry User Interfaces

(ASP.Net Dynamic Controls (Part 1

(ASP.Net Dynamic Controls (Part 2

(ASP.Net Dynamic Controls (Part 3

(ASP.Net Dynamic Controls (Part 4

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۶/۰۳ ۱۵:۳۳

سلام ،
اگر فرض کنیم RoleRepository مطلب جاری پیاده سازی منطق تجاری قسمت کاربران در یک پروژه‌ی ASP.NET MVC می‌باشد، این استثناءها کجا باید مدیریت شوند ؟ در بدنه‌ی Controller ؟
به عبارتی دیگر بهتر است نوع بازگشتی لایه‌ی سرویس یک شیء باشد که موفقیت / عدم موفقیت عملیات به همراه پیغام خطا را بازگرداند یا اینکه در صورت صحیح نبودن روند مثلا تکراری بودن نام کاربری Exception ارسال شود و استفاده کننده از Service مثل Controller مسئولیت Handle کردن استثناءها را بر عهده بگیرد ؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۶/۰۳ ۱۶:۵۳

من تا حد امکان هیچ نوع استثنایی رو مدیریت نمی‌کنم. استثناء یعنی مشکل و باید کاربر با کرش برنامه متوجه آن بشود. فقط برای لاگ کردن خطاهای برنامه‌های ASP.NET از ELMAH استفاده می‌کنم به علاوه تنظیم نمایش صفحه خطای عمومی. بیشتر از این نیازی

نیست کاری انجام شود. اولین اصل مدیریت خطاها، عدم مدیریت آن‌ها است .

عنوان: **آشنایی با Refactoring - قسمت 10**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۲۸ ۰۰:۲۰:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

یکی دیگر از روش‌هایی که جهت بهبود کیفیت کدها مورد استفاده قرار می‌گیرد، « طراحی با قراردادها » است؛ به این معنا که «بهتر است» متدهای تعریف شده پیش از استفاده از آرگومان‌های خود، آن‌ها را دقیقا بررسی کنند و به این نوع پیش شرط‌ها، قرارداد هم گفته می‌شود.
نمونه‌ای از آن‌را در قسمت 9 مشاهده کردید که در آن اگر آرگومان‌های متد AddRole، خالی یا نال باشند، یک استثناء صادر می‌شود. این نوع پیغام‌های واضح و دقیق در مورد عدم اعتبار ورودی‌های دریافتی، بهتر است از پیغام‌های کلی و نامفهوم null reference exception که بدون بررسی stack trace و سایر ملاحظات، علت بروز آن‌ها مشخص نمی‌شوند.
در دات نت 4، جهت سهولت این نوع بررسی‌ها، مفهوم Code Contracts ارائه شده است. (این نام هم از این جهت بکارگرفته شده که Design by Contract نام تجاری شرکت ثبت شده‌ای در آمریکا است!)

یک مثال:
متد زیر را در نظر بگیرید. اگر divisor مساوی صفر باشد، استثنای کلی DivideByZeroException صادر می‌شود:

```
namespace Refactoring.Day10.DesignByContract.Before
{
    public class MathMehods
    {
        public double Divide(int dividend, int divisor)
        {
            return dividend / divisor;
        }
    }
}
```

روش متداول «طراحی با قراردادها» جهت بهبود کیفیت کد فوق پیش از دات نت 4 به صورت زیر است:

```
using System;

namespace Refactoring.Day10.DesignByContract.After
{
    public class MathMehods
    {
        public double Divide(int dividend, int divisor)
        {
            if (divisor == 0)
                throw new ArgumentException("divisor cannot be zero", "divisor");

            return dividend / divisor;
        }
    }
}
```

در اینجا پس از بررسی آرگومان divisor، قرارداد خود را به آن اعمال خواهیم کرد. همچنین در استثنای تعریف شده، پیغام واضح‌تری به همراه نام آرگومان مورد نظر، ذکر شده است که از هر لحاظ نسبت به استثنای استاندارد و کلی DivideByZeroException مفهوم‌تر است.

در دات نت 4 ، به کمک امکانات مهیای در فضای نام System.Diagnostics.Contracts، این نوع بررسی‌ها نام و امکانات درخور

خود را یافته‌اند:

```
using System.Diagnostics.Contracts;

namespace Refactoring.Day10.DesignByContract.After
{
    public class MathMehods
    {
        public double Divide(int dividend, int divisor)
        {
            Contract.Requires(divisor != 0, "divisor cannot be zero");

            return dividend / divisor;
        }
    }
}
```

البته اگر قطعه کد فوق را به همراه divisor=0 اجرا کنید، هیچ پیغام خاصی را مشاهده نخواهید کرد؛ از این لحاظ که نیاز است تا فایل‌های مرتبط با آن‌را از این آدرس دریافت و نصب کنید. این کتابخانه با VS2008 و VS2010 سازگار است. پس از آن، برگه‌ی Code contracts به عنوان یکی از برگه‌های خواص پروژه در دسترس خواهد بود و به کمک آن می‌توان مشخص کرد که برنامه حین رسیدن به این نوع بررسی‌ها چه عکس العملی را باید بروز دهد.

برای مطالعه بیشتر:

Code Contracts in C#

Code Contracts in .NET 4

Introduction to Code Contracts

نظرات خوانندگان

نویسنده: Nima
تاریخ: ۱۳۹۰/۰۷/۳۰ ۱۰:۰۹:۴۹

سلام آقای نصیری
با تشکر از مطالب بسیار ارزنده شما. من در مورد Code Contract تحقیق کردم و سعی کردم یکم باهاش کار کنم. اول اینکه واقعا دلیل این رو نمیدونم که چرا باید ما یک برنامه خارجی را نصب کنیم تا این کدها واکنش نشان بدن.دوم اینکه نمیشه از این کدها داخل بلاک Try-Catch استفاده کرد. و میخواستم نظر شما را راجع به http://fluentvalidation.codeplex.com/ بدونم. این فریم ورک تقریبا همون کار رو انجام میده ولی استفاده ازش راحتتره. در کل به نظرم در بحث Refactoring به یک نقطه حساس رسیدیم و اون Validation هست. به نظر حقیر مسئله Validation و حلش میتونه سهم به سزایی در خوانایی کد داشته باشه.
ممنون میشم اگر مثل همیشه ما رو از نظرات ارزشمند خودتون مستفیض کنید

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۷/۳۰ ۱۱:۵۵:۲۶

- در مورد طراحی آن اگر نظری دارید لطفا به تیم BCL اطلاع دهید: http://blogs.msdn.com/b/bclteam/
- بحث code contacts در اینجا فراتر است از validation متداول. این نوع اعتبارسنجی‌های متداول عموما و در اکثر موارد جهت بررسی preconditions هستند؛ در حالیکه اینجا post-conditions را هم شامل می‌شوند.
- در مورد کتابخانه‌های Validation هر کسی راه و روش خاص خودش را دارد. یکی ممکن است از DataAnnotations خود دات نت استفاده کند (و http://xval.codeplex.com/)، یکی از http://validationframework.codeplex.com/ یا از http://code.google.com/p/kosher/ و یا http://tnvalidate.codeplex.com/ و یا حتی NHibernate هم کتابخانه اعتبارسنجی خاص خودش را دارد.
در کل هدف این است که این کار بهتر است انجام شود. حالا با هر کدام که راحت هستید. مانند وجود انواع فریم‌ورک‌های Unit test یا انواع مختلف سورس کنترل‌ها. مهم این است که از یکی استفاده کنید.

عنوان: **آشنایی با Refactoring - قسمت 11**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۸/۰۱ ۲۳:۰۸:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

قسمت یازدهم آشنایی با Refactoring به توصیه‌هایی جهت بالا بردن خوانایی تعاریف مرتبط با اعمال شرطی می‌پردازد.

الف) شرطهای ترکیبی را کپسوله کنید

عموما حین تعریف شرطهای ترکیبی، هدف اصلی از تعریف آن‌ها پشت انبوهی از && و || گم می‌شود و برای بیان مقصود، نیاز به نوشتن کامنت خواهند داشت. مانند:

```
using System;

namespace Refactoring.Day11.EncapsulateConditional.Before
{
    public class Element
    {
        private string[] Data { get; set; }
        private string Name { get; set; }
        private int CreatedYear { get; set; }

        public string FindElement()
        {
            if (Data.Length > 1 && Name == "E1" && CreatedYear > DateTime.Now.Year - 1)
                return "Element1";

            if (Data.Length > 2 && Name == "RCA" && CreatedYear > DateTime.Now.Year - 2)
                return "Element2";

            return string.Empty;
        }
    }
}
```

برای بالا بردن خوانایی این نوع کدها که برنامه نویس در همین لحظه‌ی تعریف آن‌ها دقیقا می‌داند که چه چیزی مقصود اوست، بهتر است هر یک از شرط‌ها را تبدیل به یک خاصیت با معنا کرده و جایگزین کنیم. برای مثال مانند:

```
using System;

namespace Refactoring.Day11.EncapsulateConditional.After
{
    public class Element
    {
        private string[] Data { get; set; }
        private string Name { get; set; }
        private int CreatedYear { get; set; }

        public string FindElement()
        {
            if (hasOneYearOldElement)
                return "Element1";

            if (hasTwoYearsOldElement)
                return "Element2";

            return string.Empty;
        }

        private bool hasTwoYearsOldElement
        {
```

```
            get { return Data.Length > 2 && Name == "RCA" && CreatedYear > DateTime.Now.Year - 2; }
        }

        private bool hasOneYearOldElement
        {
            get { return Data.Length > 1 && Name == "E1" && CreatedYear > DateTime.Now.Year - 1; }
        }
    }
}
```

همانطور که ملاحظه می‌کنید پس از این جایگزینی، خوانایی متد FindElement بهبود یافته است و برنامه نویس اگر 6 ماه بعد به این کدها مراجعه کند نخواهد گفت: «من این کدها رو نوشتم؟!»؛ چه برسد به سایرینی که احتمالا قرار است با این کدها کار کرده و یا آن‌ها را نگهداری کنند.

ب) از تعریف خواص Boolean با نام‌های منفی پرهیز کنید

یکی از مواردی که عموما علت اصلی بروز بسیاری از خطاها در برنامه است، استفاده از نام‌های منفی جهت تعریف خواص است. برای مثال در کلاس مشتری زیر ابتدا باید فکر کنیم که مشتری‌های علامتگذاری شده کدام‌ها هستند که حالا علامتگذاری نشده‌ها به این ترتیب تعریف شده‌اند.

```
namespace Refactoring.Day11.RemoveDoubleNegative.Before
{
    public class Customer
    {
        public decimal Balance { get; set; }

        public bool IsNotFlagged
        {
            get { return Balance > 30m; }
        }
    }
}
```

همچنین از تعریف این نوع خواص در فایل‌های کانفیگ برنامه‌ها نیز جدا پرهیز کنید؛ چون عموما کاربران برنامه‌ها با این نوع نامگذاری‌های منفی، مشکل مفهومی دارند.

Refactoring قطعه کد فوق بسیار ساده است و تنها با معکوس کردن شرط و نحوه‌ی نامگذاری خاصیت IsNotFlagged پایان می‌یابد:

```
namespace Refactoring.Day11.RemoveDoubleNegative.After
{
    public class Customer
    {
        public decimal Balance { get; set; }

        public bool IsFlagged
        {
            get { return Balance <= 30m; }
        }
    }
}
```

عنوان: **آشنایی با Refactoring - قسمت 12**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۸/۰۵ ۲۳:۵۶:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

قبلا در مورد تبدیل switch statement به الگوی استراتژی مطلبی را در این سایت مطالعه کرده‌اید (^) و بیشتر مربوط است به حالتی که داخل هر یک از case های یک switch statement چندین و چند سطر کد و یا فراخوانی یک تابع وجود دارد. حالت ساده‌تری هم برای refactoring یک عبارت switch وجود دارد و آن هم زمانی است که هر case، تنها از یک سطر تشکیل می‌شود؛ مانند:

```
namespace Refactoring.Day12.RefactoringSwitchStatement.Before
{
    public class Translator
    {
        public string ToPersian(string englishWord)
        {
            switch (englishWord)
            {
                case "zero":
                    return "صفر";
                case "one":
                    return "یک";
                default:
                    return string.Empty;
            }
        }
    }
}
```

در اینجا می‌توان از امکانات ساختار داده‌های توکار دات نت استفاده کرد و این switch statement را به یک dictionary تبدیل نمود:

```
using System.Collections.Generic;

namespace Refactoring.Day12.RefactoringSwitchStatement.After
{
    public class Translator
    {
        IDictionary<string, string> Words = new Dictionary<string, string>
        {
            { "zero", "صفر" },
            { "one", "یک" }
        };

        public string ToPersian(string englishWord)
        {
            string persianWord;
            if (Words.TryGetValue(englishWord, out persianWord))
            {
                return persianWord;
            }

            return string.Empty;
        }
    }
}
```

همانطور که ملاحظه می‌کنید هر case به یک key و هر return به یک value در Dictionary تعریف شده، تبدیل گشته‌اند. در اینجا هم بهتر است از متد TryGetValue جهت دریافت مقدار کلیدها استفاده شود؛ زیرا در صورت فراخوانی یک Dictionary با کلیدی که در آن موجود نباشد یک استثناء بروز خواهد کرد.
برای حذف این متد TryGetValue، می‌توان یک enum را بجای کلیدهای تعریف شده، معرفی کرد. به صورت زیر:

```
using System.Collections.Generic;

namespace Refactoring.Day12.RefactoringSwitchStatement.After
{
    public enum EnglishWord
    {
        Zero,
        One
    }

    public class Translator2
    {
        IDictionary<EnglishWord, string> Words = new Dictionary<EnglishWord, string>
        {
            { EnglishWord.Zero, "صفر" },
            { EnglishWord.One, "یک" }
        };

        public string ToPersian(EnglishWord englishWord)
        {
            return Words[englishWord];
        }
    }
}
```

به این ترتیب از یک خروجی پر از if و else و switch به یک خروجی ساده و بدون وجود هیچ شرطی رسیده‌ایم.

نظرات خوانندگان

نویسنده: afsharm
تاریخ: ۱۳۹۰/۰۸/۰۶ ۱۰:۴۸:۲۳

این یکی خیلی جالب بود

نویسنده: Farhad Yazdan-Panah
تاریخ: ۱۳۹۰/۰۸/۰۶ ۱۳:۱۰:۱۷

بسیار عالی. در معماری پروسسورهای اینتل دستوری برای کاری مشابه وجود دارد (فکر کنم یه چیزی به نام XLAT یا شبیهش). به این صورت که شما یک Look-up Table می سازید و همانند Dictionary در اینجا عمل می کنه.
تجربه شخصیم (در حد اسمبلی ماشین های X86) این روش سرعت بسیار بالاتری از حالت شرطی (مبتنی بر if) داره. در مورد Switch مطمئن نیستم.
در کل ممنون دارید کلی کیفیت کد نویسی ملتو بالا می برید.
اگه 10 تا وبلاگه دیگه مثل http://www.dotnettips.info بود الان ما وضع خیلی بهتری داشتیم.

عنوان: **آشنایی با Refactoring - قسمت 13**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۸/۱۹ ۱۲:۲۲:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

یکی از مواردی که حین کار کردن با iTextSharp واقعا اعصاب خردکن است، طراحی نامناسب ثوابت این کتابخانه می‌باشد. برای مثال:

```
public class PdfWriter
{
        /** A viewer preference */
        public const int PageLayoutSinglePage = 1;
        /** A viewer preference */
        public const int PageLayoutOneColumn = 2;
        /** A viewer preference */
        public const int PageLayoutTwoColumnLeft = 4;
        /** A viewer preference */
        public const int PageLayoutTwoColumnRight = 8;
        /** A viewer preference */
        public const int PageLayoutTwoPageLeft = 16;
        /** A viewer preference */
        public const int PageLayoutTwoPageRight = 32;

        // page mode (section 13.1.2 of "iText in Action")

        /** A viewer preference */
        public const int PageModeUseNone = 64;
        /** A viewer preference */
        public const int PageModeUseOutlines = 128;
        /** A viewer preference */
        public const int PageModeUseThumbs = 256;
        /** A viewer preference */
        public const int PageModeFullScreen = 512;
        /** A viewer preference */
        public const int PageModeUseOC = 1024;
        /** A viewer preference */
        public const int PageModeUseAttachments = 2048;

        //...
        //...
}
```

6 ثابت اول مربوط به گروه PageLayout هستند و 6 ثابت دوم به گروه PageMode ارتباط دارند و این کلاس پر است از این نوع ثوابت (این کلاس نزدیک به 3200 سطر است!). این نوع طراحی نامناسب است. بجای گروه بندی خواص یا ثوابت با یک پیشوند، مثلا PageLayout یا PageMode، این‌ها را به کلاس‌ها یا در اینجا (حین کار با ثوابت عددی) به enumهای متناظر خود منتقل و Refactor کنید. مثلا:

```
public enum ViewerPageLayout
{
     SinglePage = 1,
     OneColumn = 2,
     TwoColumnLeft = 4,
     TwoColumnRight = 8,
     TwoPageLeft = 16,
     TwoPageRight = 32
}
```

مزیت‌ها:

- طبقه بندی منطقی ثوابت در یک enum و گروه بندی صحیح آن‌ها، بجای گروه بندی توسط یک پیشوند
- استفاده بهینه از intellisense در visual studio
- منسوخ سازی استفاده از اعداد بجای معرفی ثوابت خصوصا عددی (در این کتابخانه شما می‌توانید بنویسید PdfWriter.PageLayoutSinglePage و یا 1 و هر دو صحیح هستند؛ این خوب نیست. ترویج استفاده از اصطلاحا magic numbers هم حین طراحی یک کتابخانه مذموم است.)
- کم شدن حجم کلاس اولیه (مثلا کلاس PdfWriter در اینجا) و در نتیجه نگهداری ساده‌تر آن در طول زمان

عنوان: refactoring کدهای تکراری به کمک Actionها
نویسنده: محمد رضا منشادی
تاریخ: ۱۳۹۲/۰۸/۱۳ ۱:۱۵
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

در برخی از مواقع بر روی اشیاء یک لیست، در یک کلاس، با استفاده از حلقه‌های foreach یا for کارهای متفاوتی انجام می‌شود. به عنوان مثال در یک لیست که از سطرهای فاکتور تشکیل شده است، می‌خواهیم جمع مقادیر کلیه سطرهای فاکتور یا جمع مبلغ یا مالیات یا تخفیف آنها را بدست آوریم . با وجود سادگی حلقه‌های foreach و for، ممکن است که در برخی از مواقع از راه متفاوتی استفاده شود. برای مثال اجازه بدهید مثال ذیل را با هم بررسی کنیم:

در کلاس Invoice دو متد وجود دارد با نام های CalculateTotalTax و CalculateTotal

متد CalculateTotalTax مجموع مالیات و متد CalculateTotal مجموع مقدار این فاکتور را بدست می‌آورد .

```
public float CalculateTotalTax1()
        {
            IList<InvoiceLineItem> invoiceLineItem = new List<InvoiceLineItem>();
            Decimal result = 0M;
            foreach (InvoiceLineItem index in invoiceLineItem)
            {
                result += (Decimal)index.CalculateTax();
            }
            return (float)result;
        }


        public float CalculateTotal()
        {
            IList<InvoiceLineItem> invoiceLineItem = new List<InvoiceLineItem>();
            Decimal result = 0M;
            foreach (InvoiceLineItem index in invoiceLineItem)
            {
                result += (Decimal)index.CalculateSubTotal();
            }
            return (float)result;
        }
```

این دو متد به طور عمومی یک چیز را دارند: هر دو کارهای متفاوتی را بر روی لیست سطرهای فاکتور انجام می‌دهند.

ما می‌توانیم مسئولیت چرخش در لیست سطرهای فاکتور را از این متدها برداریم و آن را از IEnumerable جدا کنیم؛ به وسیله ایجاد یک متد که پارامتر ورودی Action<T> delegate را دریافت می‌کند و این delegate را برای هر سطر در هر چرخش اجرا می‌کند.

```
public void PerformActionOnAllLineItems(Action<InvoiceLineItem> action)
        {
            IList<InvoiceLineItem> invoiceLineItem = new List<InvoiceLineItem>();

            invoiceLineItem.Add(new InvoiceLineItem { Id = 1, amount = 10, Price = 10000 });
            invoiceLineItem.Add(new InvoiceLineItem { Id = 2, amount = 10, Price = 10000 });
            invoiceLineItem.Add(new InvoiceLineItem { Id = 3, amount = 10, Price = 10000 });
            invoiceLineItem.Add(new InvoiceLineItem { Id = 4, amount = 10, Price = 10000 });

            foreach (InvoiceLineItem index in invoiceLineItem)
            {
                action(index);
            }
        }
```

و همچنین می‌توانیم دو متد خود را به شکل ذیل تغییر دهیم

```
float CalculateTotal()
        {
            Decimal result = 0M;
            PerformActionOnAllLineItems(delegate(InvoiceLineItem ili)
            {
```

```
            result += (Decimal)ili.CalculateSubTotal();
        });
        return (float)result;
    }
    float CalculateTotalTax()
    {
        Decimal result = 0M;
        PerformActionOnAllLineItems(delegate(InvoiceLineItem ili)
        {
            result += (Decimal)ili.CalculateTax();
        });
        return (float)result;
    }
```

منبع : کتاب Refactoring with Microsoft Visual Studio 2010

سورس نمونه : Advancedduplicatecoderefactoring.rar

نظرات خوانندگان

نویسنده: محمد رعیت پیشه
تاریخ: ۱۳۹۲/۰۸/۱۳ ۹:۳۵

ممنون از مطلبتون. از کجا می‌تونم کتابی رو که به عنوان منبع معرفی کردید، پیدا کنم؟
اصلا نسخه PDF اش هست؟

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۸/۱۳ ۱۵:۱۰

بله. میشه از گوگل هم کمک گرفت. همون اولین یا حداکثر دومین لینک حاصل کافی است.

عنوان: **تشخیص خودکار کدهای تکراری در یک پروژه Visual Studio**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۷/۰۸/۰۹ ۱۴:۳۲:۰۴
آدرس: www.dotnettips.info
برچسب‌ها: Tools

آیا می‌دانید چند درصد از کدهای یک پروژه شما در قسمت‌های مختلف آن تکراری هستند و تا چه حد نیاز به refactoring کدهای موجود جهت مدیریت و نگهداری ساده‌تر از آن پروژه وجود دارد؟
اخیرا پروژه سورس بازی در سایت CodePlex به نام Clone detective ارائه شده است که این کار را به صورت خودکار با یکپارچه شدن با Visual studio برای شما انجام می‌دهد. این افزودنی از آدرس زیر قابل دریافت است:
http://www.codeplex.com/CloneDetectiveVS

بهترین آموزش نحوه استفاده از آن هم از طریق ویدیوی زیر در دسترس است:
مشاهده

در نگارش فعلی آن تنها پروژه‌های سی شارپ پشتیبانی می‌شوند و در نگارش‌های آتی آن قرار است VB.net و CPP نیز افزوده شوند.

به چه دلیلی به این ابزار نیاز داریم؟
فرض کنید کلاسی را جهت انجام مقصودی خاص توسعه داده‌اید. در کلاسی دیگر برای اتمام آن، 15 سطر از یکی از توابع کلاس اول را کپی کرده و مورد استفاده قرار داده‌اید. در یک پروژه بزرگ از این موارد شاید زیاد رخ دهد (خصوصا در یک کار تیمی که ممکن است قسمتی از کار شما به‌عنوان پایه اولیه کاری دیگر مورد استفاده قرار گیرد). پس از مدتی، تغییراتی را در کلاس اول ایجاد کرده و یک سری از عیوب آن 15 سطر را که جزئی از یک تابع است برطرف خواهید کرد. بسیار هم خوب! آیا این پایان کار است؟ خیر!
آیا این مورد به کل پروژه منتقل شده است؟ آیا نگهداری یک پروژه بزرگ که دارای قسمت‌های تکراری زیادی است کار ساده‌ای است؟

علاوه بر ابزار فوق، برنامه دیگری نیز جهت تشخیص کدهای تکراری در یک پروژه به نام Simian موجود است. Simian را از آدرس زیر می‌توانید دریافت کنید:
http://www.redhillconsulting.com.au/products/simian/overview.html
این ابزار به صورت یک افزودنی VS.net ارائه نشده است اما می‌توان از طریق منوی tools آنرا به مجموعه ابزارهای مورد استفاده اضافه کرد. نحوه انجام اینکار به صورت مصور در وبلاگ زیر بیان شده است:
مشاهده

همچنین از ابزارهای دیگری از این دست می‌توان به برنامه رایگان CCFinder اشاره کرد: (ثبت نام دریافت آن رایگان است)
http://www.ccfinder.net

نظرات خوانندگان

نویسنده: salarblog
تاریخ: ۱۳۸۷/۰۸/۰۹ ۱۹:۵۹:۰۰

بسیار جالب!
ممنون. حتما تست می کنم