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: آشنایی با Refactoring - قسمت 14
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۹/۰۵ ۱۲:۵۶:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

در بسیاری از زبان‌های برنامه نویسی امکان null بودن Reference types وجود دارد. به همین جهت مرسوم است پیش از استفاده از آن‌ها، بررسی شود آیا شیء مورد استفاده نال است یا خیر و سپس برای مثال متد یا خاصیت مرتبط با آن فراخوانی گردد؛ در غیر اینصورت برنامه با یک استثناء خاتمه خواهد یافت.
مشکلی هم که با این نوع بررسی‌ها وجود دارد این است که پس از مدتی کد موجود را تبدیل به مخزنی از انبوهی از if و else ها خواهند کرد که هم درجه‌ی پیچیدگی متدها را افزایش می‌دهند و هم نگهداری آن‌ها را در طول زمان مشکل می‌سازند. برای حل این مساله، الگوی استانداردی وجود دارد به نام null object pattern ؛ به این معنا که بجای بازگشت دادن null و یا سبب بروز یک exception شدن، بهتر است واقعا مطابق شرایط آن متد یا خاصیت، «هیچ‌کاری» را انجام نداد. در ادامه، توضیحاتی در مورد نحوه‌ی پیاده سازی این «هیچ‌کاری» را انجام ندادن، ارائه خواهد شد.

الف) حین معرفی خاصیت‌ها از محافظ استفاده کنید.

برای مثال اگر قرار است خاصیتی به نام Name را تعریف کنید که از نوع رشته است، حالت امن آن رشته بجای null بودن، «خالی» بودن است. به این ترتیب مصرف کننده مدام نگران این نخواهد بود که آیا الان این Name نال است یا خیر. مدام نیاز نخواهد داشت تا if و else بنویسد تا این مساله را چک کند. نحوه پیاده سازی آن هم ساده است و در ادامه بیان شده است:

```
private string name = string.Empty;
public string Name
{
    get { return this.name; }
    set
    {
        if (value == null)
        {
            this.name = "";
            return;
        }
        this.name = value;
    }
}
```

دقیقا در زمان انتساب مقداری به این خاصیت، بررسی می‌شود که آیا مثلا null است یا خیر. اگر بود، همینجا و نه در کل برنامه، مقدار آن «خالی» قرار داده می‌شود.

ب) سعی کنید در متدها تا حد امکان null بازگشت ندهید.

برای نمونه اگر متدی قرار است لیستی را بازگشت دهد:

```
public IList<string> GetCultures()
{
    //...
}
```

و حین تهیه این لیست، عضوی مطابق منطق پیاده سازی آن یافت نشد، null را بازگشت ندهید؛ یک new List خالی را بازگشت

دهید. به این ترتیب مصرف کننده دیگری نیازی به بررسی نال بودن خروجی این متد نخواهد داشت.

ج) از متدهای الحاقی بجای if و else استفاده کنید.

پیاده سازی حالت الف زمانی میسر خواهد بود که طراح اصلی ما باشیم و کدهای برنامه کاملا در اختیار ما باشند. اما در شرایطی که امکان دستکاری کدهای یک کتابخانه پایه را نداریم چه باید کرد؟ مثلا دسترسی به تعاریف کلاس XElement دات نت فریم ورک را نداریم (یا حتی اگر هم داریم، تغییر آن تا زمانیکه در کدهای پایه اعمال نشده باشد، منطقی نیست). در این حالت می‌شود یک یا چند extension method را طراحی کرد:

```
public static class LanguageExtender
{
        public static string GetSafeStringValue(this XElement input)
        {
            return (input == null) ? string.Empty : input.Value;
        }

        public static DateTime GetSafeDateValue(this XElement input)
        {
            return (input == null) ? DateTime.MinValue : DateTime.Parse(input.Value);
        }
}
```

به این ترتیب می‌توان امکانات کلاس پایه‌ای را بدون نیاز به دسترسی به کدهای اصلی آن مطابق نیازهای خود تغییر و توسعه داد.

عنوان: **اهمیت code review**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۱۶ ۱۴:۱۰:۰۰
آدرس: www.dotnettips.info
گروه‌ها: Refactoring

تا جایی که دقت کردم (در بلاگ‌هایی که منتشر می‌شوند) در آنسوی آب‌ها، « code review » یک شغل محسوب می‌شود. سازمان‌ها، شرکت‌ها و امثال آن از مشاورین یا برنامه نویس‌هایی با مطالعه بیشتر دعوت می‌کنند تا از کدهای آن‌ها اشکال‌گیری کنند و بابت اینکار هم هزینه می‌کنند.
اگر علاقمند باشید قسمتی از یک پروژه سورس باز دریافت شده از همین دور و اطراف را با هم مرور کنیم:

```
//It's only for code review purpose!
protected void Button1_Click1(object sender, EventArgs e)
{
        string  strcon;
        string strUserURL;
        string strSQL;
        string strSQL1;
        strSQL = "SELECT UserLevel FROM listuser " + "WHERE Username='" + TextBox2.Text + "' " + "And
Password='" + TextBox3.Text + "';";
        strSQL1 = "SELECT Pnumber FROM listuser " + "WHERE Username='" + TextBox2.Text + "' " + "And
Password='" + TextBox3.Text + "';";
        strcon = @"Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\bimaran.mdf;Integrated
Security=True;User Instance=True";
        SqlConnection myConnection = new SqlConnection(strcon);

        SqlCommand myCommand = new SqlCommand(strSQL, myConnection);
        SqlCommand myCommand1 = new SqlCommand(strSQL1, myConnection);
        myConnection.Open();

        strUserURL = (string)myCommand.ExecuteScalar();
        send = (string)myCommand1.ExecuteScalar();
        myCommand.Dispose();
        myCommand1.Dispose();
        myConnection.Close();


        if (strUserURL != null)
        {
            Label1.Text = "";

            url = "?Pn=" + code(send);
            FormsAuthentication.SetAuthCookie(TextBox2.Text, true);
            Response.Redirect("Page/" + strUserURL + url);
       }
        else
            Label3.Text = "چنین کاربری با این مشخصات ثبت نشده است.";
}
```

مروری بر این کد یا «مشکلات این کد»:
- کانکشن استرینگ داخل کدها تعریف شده. یعنی اگر نیاز به تغییری در آن بود باید کدهای برنامه تغییر کنند. آن هم نه فقط در این تابع بلکه در کل برنامه.
- از پارامتر استفاده نشده. کد 100 درصد به تزریق اس کیوال آسیب پذیر است.
- نحوه‌ی dispose شیء کانکشن غلط است. هیچ ضمانتی وجود ندارد که کدهای فوق سطر به سطر اجرا شود و خیلی زیبا به سطر بستن کانکشن استرینگ برسد. فقط کافی است این میان یک استثنایی صادر شود و تمام. به عبارتی این سایت فقط با کمتر از 30 کاربر همزمان از کار می‌افته. بعد نیاید بگید من یک سرور دارم با 16 گیگ رم ولی باز کم میاره! همش برنامه کند میشه. همش سایت بالا نمیاد!
- همین تعریف کردن متغیرها در ابتدای تابع یعنی این برنامه نویس هنوز حال و هوای ANSI C را دارد!

- مهم نیست لایه بندی کنید. ولی یک لایه در این نوع پروژه‌ها الزامی است و آن هم DAL نام دارد. DAL یعنی کثافت کاری نکنید. یعنی داخل هر تابع گپه گپه بر ندارید open و close بذارید. برید یک تابع یک گوشه‌ای درست کنید که این عملیات را محصور کند.
- همین وجود Button1 و Label1 یعنی تو خود شرح مفصل بخوان از این مجمل!

نظرات خوانندگان

نویسنده: amir hosein jelodari
تاریخ: ۱۳۹۰/۱۰/۱۶ ۱۹:۵۷:۴۸

سلام ...
خیلی خیلی جالب و مفیده این سبک آموزشی!

الان تو این تابع دو تا سلکت متفاوت از دیتابیس انجام نمیشه؟ نه؟ ... یعنی دو تا عملیات متفاوت داریم . خوب حالا بهتر نیست هر کدوم تو یه متود wrap بشه؟!

نویسنده: aliaghdam
تاریخ: ۱۳۹۰/۱۰/۱۶ ۲۲:۴۳:۲۷

اندر همین باب!!!
http://www.aliaghdam.ir/2011/06/blog-post_1694.html

نویسنده: shahin kiassat
تاریخ: ۱۳۹۰/۱۰/۱۶ ۲۲:۵۶:۵۹

سلام.
به نظر من هر دو Select باید در یک تابع باشه.چون شرط هر دو select یکسان هست و برنامه نویس این کد از نتیجه ی هر دو query در محدوده ی همین تابع استفاده کرده.

نویسنده: Meysam Hooshmand
تاریخ: ۱۳۹۰/۱۰/۱۶ ۲۳:۳۲:۵۴

ان الله مع الصابرین
یک نکته، اگر دست به آچار نباشیم و بزار نشناسیم، گاهی اوقات این تر و تمیز کد نویسی ،خودش میشه مایه دردسر، خصوصا برای همونایی که تو حال و هوای قدیما هستند. مثلا، داشتن لایه ای برای wrap کردن، spها، بدون جنریتور،!!!!
کثافت کاری، عجب کلمه بدیه
!

نویسنده: Javad Darvish Amiry
تاریخ: ۱۳۹۰/۱۰/۱۷ ۰۰:۰۴:۲۶

- همین وجود Button1 و Label1 یعنی تو خود شرح مفصل بخوان از این مجمل!

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۱۷ ۰۰:۰۴:۳۳

همون، مگر اینکه این «جنریتور»ها با چهارتا علامت تعجب به داد برسه و گرنه کمی صحبت کردن در مورد زیرساخت این‌ها کمی باعث سرگیجه میشه؛ خصوصا تفکر در مورد آن‌ها.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۱۷ ۰۰:۲۸:۴۱

هر دو کوئری رو میشه تبدیل به یک کوئری کرد : SELECT Pnumber, UserLevel FROM listuser where blabla و اون وقت بهتر خواهد بود که از DataReader استفاده شود. البته نه در این متد.
و کسانی هم که نمی‌خواهند از ORM استفاده کنند، Wrappers خیلی خوبی بعد از دات نت 4 برای ADO.NET اومده. مطلب در موردش

در سایت هست تحت عنوان Micro ORMs

نویسنده: A. Karimi
تاریخ: ۱۳۹۰/۱۰/۱۷ ۰۰:۳۰:۲۴

این بنده خدا سعی کرده نام گذاری مجارستانی رو رعایت کنه ولی ...! دوتا امتیاز منفی دیگه، یکی برای اینکه نامگذاری به این سبک توی سی شارپ منسوخه و دیگری اینکه همون نام گذاری رو هم برای متغییر اول رعایت نکرده.

نویسنده: Parham.D
تاریخ: ۱۳۹۰/۱۰/۱۷ ۰۷:۴۸:۳۴

سلام. چطور میشه روش کد نویسی صحیح را یاد گرفت؟ در اثر تجربه به دست میاد، یا برای این کار مرجع و آموزشی وجود داره؟ کدنویسی بلدم، خوب نویسی نه؟! چطور این مشکل را حل کنم؟

نویسنده: zahra abdolahi
تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۰:۳۸:۱۷

من چند وقتیه که مشترک بلاگ شما شدم. پست هاتون بسیار مفید و کاربردیه. خواستم ازتون تشکر کرده باشم.
در مورد این پست هم موافقم باهاتون. ولی من تو شرکت هایی که کار کردم معمولا یه قوانینی در این زمینه وجود داره که موقع شروع به کار برنامه نویس بهش می دن. هرچند شرکت تا شرکت فرق می کنه و گاهی به طور کامل رعایت نمیشه ولی حداقل موقعی که از اواسط یک پروژه واردش میشیم و قراره در ادامه کدهای دیگران کد بنویسیم مفیده.
البته من خودم رو برنامه نویس با تجربه ای نمی دونم اما به شخصه با وجود لایه DAL بسیار موافقم. نظم خاصی به کد میده و باعث میشه موقع تست یا بازبینی کد دیگه درگیر دیتابیس نباشیم.
یه خواهشی هم دارم میشه لطف کنید یه کم در مورد لزوم تحلیل درست پیش از شروع به کدنویسی هم بنویسید.
پایدار باشید.

نویسنده: m.jamshidi
تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۱:۵۴:۴۳

سلام، خسته نباشید، ما وقتی با c# برنامه می نویسیم اصولا متغیرهایی را که نیاز داریم را در ابتدای تابع تعریف می کنیم پس منظورتان از جمله "تعریف کردن متغیرها در ابتدای تابع یعنی این برنامه نویس هنوز حال و هوای ANSI C را دارد!" چیست؟ یعنی هرجا نیاز داشتیم باز هم متغیر تعریف کنیم. موفق باشید

نویسنده: m.jamshidi
تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۲:۲۶:۵۶

عجب!!!
آخه مگه دو تا select متفاوت از یه جدول توی یک فرمان انجام میشه!
اونم توی دستوری مثل login جالبه .....

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۲:۵۶:۱۶

در ANSI C ، متغیرها فقط ابتدای تابع (scope block) مجاز به تعریف هستند؛ و هر زمان که به دنبال تعریف متغیرها در ابتدای متدی گشتید یعنی هنوز حال و هوای ANSI C برقرار است. اما از زمان سی++ به بعد خیر؛ هر چند کامپایلر با هر دو حالت مشکلی ندارد.
بحث تکمیلی در اینجا : (^)
به صورت خلاصه: از سی++ به بعد متغیرها را در نزدیکترین مکانی که اولین استفاده از آنها صورت می‌گیرد تعریف کنید تا بهتر بدانید که به چه علتی تعریف شده. همچنین خیالتان هم راحت می‌شود که به اشتباه جای دیگری استفاده نشده یا نمی‌شود.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۳:۰۲:۰۴

هیچ ایرادی نداره که حتی 10 کوئری متفاوت هم در یک متد صادر شوند. این business logic که می‌گن همینه. بسته به منطق تجاری که قرار است پیاده سازی شود، شما هر تعداد کوئری که نیاز داشتید را فراخوانی کنید.
فقط بحث اینجا است که فایل code behind ، محل پیاده سازی Data access layer و همچنین محل تعریف هیچ نوع منطق تجاری نیست. این‌ها باید از فایل code behind دور شوند. اینجا فقط محل استفاده نهایی از منطق تهیه شده است.

نویسنده: ناصر طاهری
تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۴:۰۵:۱۶

درود بر شما
به نظر من برای این کد همون کلمه کثافت کاری که استفاده کردید مناسبه. من خواستم پروژه ای رو توسعه بدم که با همچین کدی برخوردم. واقعا خیلی بد بود. خیلی

نویسنده: ناصر طاهری
تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۴:۰۸:۰۶

:)) فکر کردم عکس برای قسمت عکس مخاطب انتخاب میشه. مثل اینکه اشتباه شد.پاکم نمیشه.شرمنده

نویسنده: rahmat rezaei
تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۶:۱۴:۳۳

شک ندارم خیلی از ماها زمانی کدهای بدتر از این هم نوشتیم. بنابراین اینقدر کد بیچاره را مسخره نکنید.
طرف اگر بیاد اینجا و نظرات شما رو بخونه که سرشو میندازه پایین و برنامه نویسی رو میذاره کنار و احتمالا میره سمت تحلیل و طراحی!
دستگیری کنید از ضعیفان نه سرکوفت بزنید!

نویسنده: Mehdi
تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۶:۴۰:۳۵

با سلام و ممنون از مطلب خوب شما اقای نصیری عزیز
یک سوال در مورد نحوه dispose کردن شی کانکشن دارم
- شما گفتید که اگر استثنایی اون وسط اتفاق بیفته دیگه به متد dispose نمیرسه و کانکشن باقی میمونه خب این درسته حالا اگه این کدها رو داخل بلاک using قرار بدیم و اتفاقا استثنایی اتفاق بیفته اون وقت شی کانکشن dispose میشه یا به همون صورت قبلی رفتار میشه؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۶:۵۱:۰۹

using توسط کامپایلر به try/finally ترجمه میشه و قسمت dispose رو در قسمت finally قرار میده. بنابراین اگر این وسط استثنایی رخ بده، حتما قسمت finally اجرا خواهد شد.

نویسنده: Mehdi
تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۷:۱۷:۳۹

بسیار ممنون
پس به نظر شما نوشتن using هم بهتر و هم راحت تر از نوشتن try catch finally هستش درسته؟

نویسنده: وحید نصیری

تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۷:۳۸:۴۱

بله. من خودم همیشه از using استفاده می‌کنم. جمع و جورتر هست و تمیزتر. به علاوه مثلا زمان dispose خودش بررسی می‌کنه که آیا شیء الان نال هست یا نه. در کل یک سرویس رایگان هست از طرف کامپایلر!

نویسنده: A. Karimi
تاریخ: ۱۳۹۰/۱۰/۱۷ ۱۸:۴۲:۵۷

حرف شما درسته فقط در مورد تحلیل و طراحی، کسی که نمیتونه یک قطعه کد با نظم رو ایجاد کنه خیلی خیلی خطرناک تر هست که بره تحلیل و طراحی کنه.

نویسنده: Mohsen
تاریخ: ۱۳۹۰/۱۰/۱۸ ۱۳:۴۸:۵۴

پس هیچ یک از شما برنامه های فاکس رو که طرف یک جدول تعریف کرده با 70 تا فیلد و اسمشون رو از 1 تا 60 گذاشته(در مورد کدهای نوشته شده چیزی نمیگم!)رو تا به حال پشتیبانی نکرده تا درد منو کشیده باشه!):

نویسنده: Mohsen
تاریخ: ۱۳۹۰/۱۰/۱۸ ۱۳:۵۶:۵۰

اما گذشته از بحث مثالی که زدم(اشتباه و بی اهمیت بودن برنامه نویس به چیزی که خلق می کند!) واقعا وجود امکانات هم در نوشتن کد تمیز و اصولی واقعا تاثیر گذاره.یعنی با وجود هزارتا داستان مثل Intellisense و ابزارهای Refactor که با IDE ای با قدرت VS موجود هستند،دیگه بی انصافیه که طوری کد نوشته شود که دیگران از آن چیزی متوجه نشوند...

نویسنده: Mohsen Najafzadeh
تاریخ: ۱۳۹۰/۱۰/۱۸ ۱۴:۰۵:۱۰

سلام مهندس
مواردی که سایت فقط با کمتر از 30 کاربر همزمان از کار می‌افته رو اگه لیست کنید ممنون می شم

نویسنده: ناصر طاهری
تاریخ: ۱۳۹۰/۱۰/۱۸ ۱۵:۳۲:۲۴

درسته اما بیشتر مواقع آدم باید بهش برخوره که بتونه بره دنبال کاری که تو اون زمینه مورد تمسخر(که مسخره کردن نیست بلکه یک انتقاده) قرار گرفته. :)

نویسنده: Mohammad Safdel
تاریخ: ۱۳۹۰/۱۰/۱۸ ۲۳:۵۰:۲۵

ممنون. متاسفانه تو اغلب شرکتهای ایرانی شعارشون اینه که "کدی که کار می کنه نباید تغییر بدن"! بنابراین Code Review یعنی کشک.

نویسنده: مهدی موسوی
تاریخ: ۱۳۹۰/۱۰/۲۰ ۱۷:۳۳:۱۱

سلام.
اگر کدی آزمایش شده، مرور شده و "کار میکنه"، دیگه نیازی به تغییر اون وجود نداره. برنامه نویس ها عموما در دوران حرفه ای خودشون، حداقل یک بار با "وسوسه بازنویسی همه چیز از نو" روبرو میشن، وسوسه ای که در ابتدا، افق های روشنی رو برامون ترسیم میکنه، اما در انتها، منجر به داشتن کدی به مراتب بدتر از اون چیزی که در ابتدا داشتیم، میشه.

وقتی کدی قدیمی (که بدون مشکل کار میکنه) رو دور میندازیم، در حقیقت داریم زمانی رو که صرف رفع ایرادهای موجود در اون

کرده بودیم (که میتونه روزها، هفته ها یا ماه ها باشه) رو هدر میدیم. گذشته از این، چون احتمالا به تمام بخش های کد و عملکرد اون اشراف نداریم، چیزهایی ممکنه در کد ببینیم که به نظرمون احمقانه بیاد و حذف اونها، باز موجب از کار افتادن بخش هایی از سیستم بشه که Debug کردن اون، مستلزم صرف زمانی هستش که تیم قبلی اون زمان رو یکبار صرف این کار کرده بوده. بنابراین، نمی تونیم به عنوان یه اصل کلی عبارت "کدی که کار میکنه رو نباید تغییر داد" رو رد کنیم! این مساله، باید بازای Case های مختلف، بدقت بررسی بشه و بعد در مورد اون Case خاص، نظر داده بشه.

طبیعتا، با دیدن کد آورده شده در این پست میشه به این مساله پی برد که نویسنده اون کد، در وهله اول، با اصول و مفاهیم اولیه نوشتن یک کد تمیز، بیگانه بوده. چنین افرادی، ابتدا باید آموزش ببینن و مرور یا عدم مرور کد اونها، در طولانی مدت، هیچ سودی در پی نخواهد داشت.

موفق باشید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۰ ۲۰:۴۶:۰۹

این کد بالا هم «کار می‌کنه». این فرد یا این شرکت اگر جرات داره اون برنامه رو در طی یک سایت بذاره روی اینترنت! خیلی‌ها علاقمند هستند تا کمی اهمیت مرور کدها رو به اون‌ها به نحو مقتضی یادآوری کنند!

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۰ ۲۰:۵۲:۱۴

«نمی تونیم به عنوان یه اصل کلی عبارت "کدی که کار میکنه رو نباید تغییر داد" رو رد کنیم»
این مساله فقط زمانی رخ می‌ده که هیچ تستی وجود نداشته باشه. هیچ باگی در وهله اول به عنوان یک آزمون واحد جدید تعریف و سپس بررسی نشده باشه. در این صورت چون باگ‌ها به نحو شایسته‌ای مستند نشدن، سیستم دربرابر تغییرات شکننده خواهد بود، همچنین دلایل وجودی قسمت‌های «احمقانه» کد هم مشخص نخواهد بود.

نویسنده: Shima2012
تاریخ: ۱۳۹۰/۱۰/۲۰ ۲۱:۳۱:۰۱

با سلام خدمت شما استاد عزیز
در لینکی که در ادامه قرار دادم سمپلی از روشی که مدتی در ASP.Net WebForm استفاده میکردم رو قرار دادم.
میخواستم خواهش کنم اگه میشه لطف کنید و مشابه این پست، در پست دیگری روش من رو هم در سایت قرار بدید تا شما و دوستان دیگر نظرات و انتقادات خود را ارائه کنند.
باور کنید انتقادات شما اساتید بزرگوار بسیار در پیشرفت من موثر است.
پیشاپیش ممنون از لطف شما.

http://s2.picofile.com/file/7243730642/TestApp.rar.html

نویسنده: مهدی موسوی
تاریخ: ۱۳۹۰/۱۰/۲۰ ۲۲:۳۲:۳۱

سلام.
من به برخی از مشکلات کد شما اشاره می کنم:

1. ترکیب کدهای DAL و کد UI.
2. عدم جداسازی Business Object ها.
3. ایجاد کلاسی به اسم SqlHelper و قرار دادن N تا متود static در این کلاس، نشون میده که برنامه از هیچ یک از معیارهای موجود پیروی نمیکنه. مطلقا دلیلی نداره که متودها static تعریف بشن و ... حتی اگر قرار باشه چنین کلاسی تعریف کنیم، (که من

کاملا باهاش مخالفم)، باید متودهای اونو بر اساس Property ی Singleton ای از کلاس به بیرون Expose کنیم. نه اینکه همه متودهاشو static بذاریم و کلاس رو هم sealed کنیم و ... چنین کلاسی هرگز قابل توسعه نیست.
4. وقتی همه متودهای کلاس static هستش، چرا Constructor ای private برای اون کلاس تعریف کرده اید؟ تازه بهتر بود جای اینکه کلاس رو sealed تعریف می کردید، اونو static تعریف می کردید (اگر مجاب میشدیم که تعریف چنین کلاسی صحیح هستش).
5. catch کردن کلیه SqlException ها در متود CloseCnt.
6. عدم استفاده از Parametric Command ها (و در نتیجه فراهم اومدن امکان SQL Injection).
7. عدم اجرای StyleCop روی کد (و در نتیجه، نوشتن Comment های مربوط به هر تابع بر اساس Style ای من در آوردی، عدم رعایت Spacing، نوشتن if ها در یک خط و ...)
8. و ...

موفق باشید.

نویسنده: Shima2012
تاریخ: ۱۳۹۰/۱۰/۲۰ ۲۲:۵۳:۵۳

واقعاً ممنون
لطف کردید.
شما مثالی دارید که شبیه مثالی که من زدم باشه و استاندارد هایی که گفتید رو رعایت کرده باشه تا من بتونم یاد بگیرم؟

نویسنده: پریسا زاهدی
تاریخ: ۱۳۹۳/۱۰/۱۹ ۱۱:۵۷

سلام
اگر از Try Catch استفاده نکنیم, با استفاده از Using پس چطوری به اطلاعات استثنای رخ داده شده دسترسی پیدا کنیم که بخواهیم لاگش کنیم ؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۱۹ ۱۲:۱۷

تهیه لاگ خودکار را واگذار کنید به ابزارهایی مانند ELMAH

عنوان: **مروری بر کدهای کلاس SqlHelper**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۱ ۰۰:۱۲:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring

قسمتی از یک پروژه به همراه کلاس SqlHelper آن در کامنت‌های مطلب « اهمیت Code review » توسط یکی از خوانندگان بلاگ جهت Code review مطرح شده که بهتر است در یک مطلب جدید و مجزا به آن پرداخته شود. قسمت مهم آن کلاس SqlHelper است و مابقی در اینجا ندید گرفته می‌شوند:

```
//It's only for code review purpose!
using System.Data;
using System.Data.SqlClient;
using System.Web.Configuration;


public sealed class SqlHelper
{
    private SqlHelper() { }


    //  Send Connection String
    //-----------------------------------------------------------------------------------------
    public static string GetCntString()
    {
        return WebConfigurationManager.ConnectionStrings["db_ConnectionString"].ConnectionString;
    }


    //  Connect to Data Base SqlServer
    //-----------------------------------------------------------------------------------------
    public static SqlConnection Connect2Db(ref SqlConnection sqlCnt, string cntString)
    {
        try
        {
            if (sqlCnt == null) sqlCnt = new SqlConnection();
            sqlCnt.ConnectionString = cntString;
            if (sqlCnt.State != ConnectionState.Open) sqlCnt.Open();
            return sqlCnt;
        }
        catch (SqlException)
        {
            return null;
        }
    }


    //  Run ExecuteScalar Command
    //-----------------------------------------------------------------------------------------
    public static string RunExecuteScalarCmd(ref SqlConnection sqlCnt, string strCmd, bool blnClose)
    {
        Connect2Db(ref sqlCnt, GetCntString());
        using (sqlCnt)
        {
            using(SqlCommand sqlCmd = sqlCnt.CreateCommand())
            {
                sqlCmd.CommandText = strCmd;
                object objResult = sqlCmd.ExecuteScalar();
                if (blnClose) CloseCnt(ref sqlCnt, true);
                return (objResult == null) ? string.Empty : objResult.ToString();
            }
        }
    }

    //   Close SqlServer Connection
    //-----------------------------------------------------------------------------------------
    public static bool CloseCnt(ref SqlConnection sqlCnt, bool nullSqlCnt)
    {
        try
        {
            if (sqlCnt == null) return true;
            if (sqlCnt.State == ConnectionState.Open)
```

```
            {
                sqlCnt.Close();
                sqlCnt.Dispose();
            }
            if (nullSqlCnt) sqlCnt = null;
            return true;
        }
        catch (SqlException)
        {
            return false;
        }
    }
}
```

مثالی از نحوه استفاده ارائه شده:

```
protected void BtnTest_Click(object sender, EventArgs e)
        {
            SqlConnection sqlCnt = new SqlConnection();
            string strQuery = "SELECT COUNT(UnitPrice) AS PriceCount FROM [Order Details]";


            // در این مرحله پارامتر سوم یعنی کانکشن باز نگه داشته شود
            string strResult = SqlHelper.RunExecuteScalarCmd(ref sqlCnt, strQuery, false);



            strQuery = "SELECT LastName + N'-' + FirstName AS FullName FROM Employees WHERE (EmployeeID
= 9)";
            // در این مرحله پارامتر سوم یعنی کانکشن بسته شود
            strResult = SqlHelper.RunExecuteScalarCmd(ref sqlCnt, strQuery, true);
        }
```

مروری بر این کد:

1) نحوه کامنت نوشتن

بین سی شارپ و زبان سی++ تفاوت وجود دارد. این نحوه کامنت نویسی بیشتر در سی++ متداول است. اگر از ویژوال استودیو استفاده می‌کنید، مکان نما را به سطر قبل از یک متد منتقل کرده و سه بار پشت سر هم forward slash را تایپ کنید. به صورت خودکار ساختار خالی زیر تشکیل خواهد شد:

```
/// <summary>
    ///
    /// </summary>
    /// <param name="sqlCnt"></param>
    /// <param name="cntString"></param>
    /// <returns></returns>
    public static SqlConnection Connect2Db(ref SqlConnection sqlCnt, string cntString)
```

این روش مرسوم کامنت نویسی کدهای سی شارپ است. خصوصا اینکه ابزارهایی وجود دارند که به صورت خودکار از این نوع کامنت‌ها، فایل CHM درست می‌کنند.

2) وجود سازنده private

احتمالا هدف این بوده که نه شخصی و نه حتی کامپایلر، وهله‌ای از این کلاس را ایجاد نکند. بنابراین بهتر است کلاسی را که تمام متدهای آن static است (که به این هم خواهیم رسید!) ، راسا static معرفی کنید. به این ترتیب نیازی به سازنده private

نخواهد بود.

3) وجود try/catch

یک اصل کلی وجود دارد: اگر در حال طراحی یک کتابخانه پایه‌ای هستید، try/catch را در هیچ متدی از آن لحاظ نکنید. بله؛ درست خوندید! لطفا try/catch ننویسید! کرش کردن برنامه خوب است! لایه‌های بالاتر برنامه که در حال استفاده از کدهای شما هستند متوجه خواهند شد که مشکلی رخ داده و این مشکل توسط کتابخانه مورد استفاده «خفه» نشده. برای مثال اگر هم اکنون SQL Server در دسترس نیست، لایه‌های بالاتر برنامه باید این مشکل را متوجه شوند. Exception اصلا چیز بدی نیست! کرش برنامه اصلا بد نیست!

فرض کنید که دچار بیماری شده‌اید. اگر مثلا تبی رخ ندهد، از کجا باید متوجه شد که نیاز به مراقبت پزشکی وجود دارد؟ اگر هیچ علامتی بروز داده نشود که تا الان نسل بشر منقرض شده بود!

4) وجود ref و out

دوستان گرامی! این ref و out فقط جهت سازگاری با زبان C در سی شارپ وجود دارد. لطفا تا حد ممکن از آن استفاده نکنید! مثلا استفاده از توابع API ویندوز که با C نوشته شده‌اند.

یکی از مهم‌ترین کاربردهای pointers در زبان سی، دریافت بیش از یک خروجی از یک تابع است. برای مثال یک متد API ویندوز را فراخوانی می‌کنید؛ خروجی آن یک ساختار است که به کمک pointers به عنوان یکی از پارامترهای همان متد معرفی شده. این روش به وفور در طراحی ویندوز بکار رفته. ولی خوب در سی شارپ که از این نوع مشکلات وجود ندارد. یک کلاس ساده را طراحی کنید که چندین خاصیت دارد. هر کدام از این خاصیت‌ها می‌توانند نمایانگر یک خروجی باشند. خروجی متد را از نوع این کلاس تعریف کنید. یا برای مثال در دات نت 4، امکان دیگری به نام Tuples معرفی شده برای کسانی که سریع می‌خواهند چند خروجی از یک تابع دریافت کنند و نمی‌خواهند برای اینکار یک کلاس بنویسند.

ضمن اینکه برای مثال در متد Connect2Db، هم کانکشن یکبار به صورت ref معرفی شده و یکبار به صورت خروجی متد. اصلا نیازی به استفاده از ref در اینجا نبوده. حتی نیازی به خروجی کانکشن هم در این متد وجود نداشته. کلیه تغییرات شما در شیء کانکشنی که به عنوان پارامتر ارسال شده، در خارج از آن متد هم منعکس می‌شود (شبیه به همان بحث pointers در زبان سی). بنابراین وجود ref غیرضروری است؛ وجود خروجی متد هم به همین صورت.

5) استفاده از using در متد RunExecuteScalarCmd

استفاده از using خیلی خوب است؛ همیشه اینکار را انجام دهید!

اما اگر اینکار را انجام دادید، بدانید که شیء sqlCnt در پایان بدنه using ، توسط GC نابوده شده است. بنابراین اینجا bool blnClose دیگر چه کاربردی دارد؟! تصمیم شما دیگر اهمیتی نخواهد داشت؛ چون کار تخریبی پیشتر انجام شده.

6) متد CloseCnt

این متد زاید است؛ به دلیلی که در قسمت (5) عنوان شد. using های استفاده شده، کار را تمام کرده‌اند. بنابراین بستن اشیاء dispose شده معنا نخواهد داشت.

7) در مورد نحوه استفاده

اگر SqlHelper را در اینجا مثلا یک DAL ساده فرض کنیم (data access layer)، جای قسمت BLL (business logic layer) در اینجا خالی است. عموما هم چون توضیحات این موارد را خیلی بد ارائه داده‌اند، افراد از شنیدن اسم آن‌ها هم وحشت می‌کنند. BLL یعنی کمی دست به Refactoring بزنید و این پیاده سازی منطق تجاری ارائه شده در متد BtnTest_Click را به یک کلاس مجزا خارج از code behind پروژه منتقل کنید. Code behind فقط محل استفاده نهایی از آن باشد. همین! فعلا با همین مختصر شروع کنید.

مورد دیگری که در اینجا باز هم مشهود است، عدم استفاده از پارامتر در کوئری‌ها است. چون از پارامتر استفاده نکرده‌اید، SQL Server مجبور است برای حالت EmployeeID = 9 یکبار execution plan را محاسبه کند، برای کوئری بعدی مثلا EmployeeID = 19، اینکار را تکرار کند و الی آخر. این یعنی مصرف حافظه بالا و همچنین سرعت پایین انجام کوئری‌ها. بنابراین اینقدر در قید و بند باز نگه داشتن یک کانکشن نباشید؛ مشکل اصلی جای دیگری است!

8) برنامه وب و اطلاعات استاتیک!

این پروژه، یک پروژه ASP.NET است. دیدن تعاریف استاتیک در این نوع پروژه‌ها یک علامت خطر است! در این مورد قبلا مطلب نوشتم:
متغیرهای استاتیک و برنامه‌های ASP.NET

یک درخواست عمومی!

لطف کنید در پروژهای «جدید» خودتون این نوع کلاس‌های SqlHelper رو «دور بریزید». یاد گرفتن کار با یک ORM جدید اصلا سخت نیست. مثلا طراحی Entity framework مایکروسافت به حدی ساده است که هر شخصی با داشتن بهره هوشی در حد یک عنکبوت آبی یا حتی جلبک دریایی هم می‌تونه با اون کار کنه! فقط NHibernate هست که کمی مرد افکن است و گرنه مابقی به عمد ساده طراحی شده‌اند.

مزایای کار کردن با ORM ها این است:

- کوئری‌های حاصل از آن‌ها «پارامتری» است؛ که این دو مزیت عمده را به همراه دارد:

امنیت: مقاومت در برابر SQL Injection

سرعت و همچنین مصرف حافظه کمتر: با کوئری‌های پارامتری در SQL Server همانند رویه‌های ذخیره شده رفتار می‌شود.

- عدم نیاز به نوشتن DAL شخصی پر از باگ. چون ORM یعنی همان DAL که توسط یک سری حرفه‌ای طراحی شده.
- یک دست شدن کدها در یک تیم. چون همه بر اساس یک اینترفیس مشخص کار خواهند کرد.
- امکان استفاده از امکانات جدید زبان‌های دات نتی مانند LINQ و نوشتن کوئری‌های strongly typed تحت کنترل کامپایلر.
- پایین آوردن هزینه‌های آموزشی افراد در یک تیم. مثلا EF را می‌شود به عنوان یک پیشنیاز در نظر گرفت؛ عمومی است و همه گیر. کسی هم از شنیدن نام آن تعجب نخواهد کرد. کتاب(های) آموزشی هم در مورد آن زیاد هست.

و ...

نظرات خوانندگان

نویسنده: M_royasaz
تاریخ: ۰۱:۰۷:۱۹ ۱۳۹۰/۱۰/۲۱

"برنامه وب و اطلاعات استاتیک!"این مسئله شامل متدهای استاتیک در لایه بیزینس هم میشه ؟

نویسنده: hossein moradinia
تاریخ: ۰۱:۰۸:۲۱ ۱۳۹۰/۱۰/۲۱

سلام
گفته شد که از out استفاده نکنیم.پس تکلیف استفاده از دستوری TryParse در سی شارپ چه میشود؟!!
جایگزینی برای آن وجود دارد یا اینکه بایدا استفاده شود؟!!

نویسنده: وحید نصیری
تاریخ: ۰۸:۲۵:۰۸ ۱۳۹۰/۱۰/۲۱

طراحی TryParse مربوط به دات نت یک است که هنوز حال و هوای دوران C برقرار بود. اگر امروز می‌خواستند آن‌را طراحی کنند هیچ وقت از Out در آن استفاده نمی‌شد.

نویسنده: وحید نصیری
تاریخ: ۰۸:۲۸:۰۹ ۱۳۹۰/۱۰/۲۱

اگر فقط متدها استاتیک باشد، خیر. مانند مثال بالا. اما کیفیت این کد طوری است که تمایل به استفاده از اطلاعات استاتیک در آن بالا است. احتمالا شاید چون شیک‌تر به نظر می‌رسه. در اون صورت اگر جایی نوشته شده public static bool IsAdmin یعنی تمام کاربران سایت هم اکنون ادمین هستند یا می‌توانند باشند.

نویسنده: Shima 2012
تاریخ: ۰۸:۳۳:۲۷ ۱۳۹۰/۱۰/۲۱

WOW Wonderful
با سلام و تشکر از شما استاد گرامی بابت وقت گرانبهایی که برای آموزش افرادی مانند من هزینه میکنید.در ابتدا عرض کنم ارزش لطف شما را به خوبی میدانم چرا که برنامه نویس بزرگی مثل شما به جای وقت گذاشتن برای کدهای امثال من میتواند به Business خود پرداخته و در همین زمان صرف شده... (اجرکم عندالله)

خواهش دیگری که دارم این است که میتوانید یک سمپل از یکی از کارهایی که از لحاظ فنی مورد تائید شما میباشد را جهت دانلود معرفی کنید تا من و امثال من بتوانیم از آن جهت یادگیری استفاده نماییم؟ (اگر از کارهای خودتان باشد که دیگر ...)

در هر صورت ممنونم از لطف شما.

نویسنده: وحید نصیری
تاریخ: ۰۸:۴۳:۱۹ ۱۳۹۰/۱۰/۲۱

اول نیاز به پایه تئوری هست برای کار عملی. از اینجا (^) می‌شود شروع کرد.

نویسنده: K Khodaei
تاریخ: ۰۹:۳۶:۵۸ ۱۳۹۰/۱۰/۲۱

مشکل سرعت ORM را چگونه برطرف کنیم؟؟ یکی از مزایای خود datareader در وب سرعت آن هستش. با دور انداختن SqlHelper می توان این درخواست شما را عملی کرد؟؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۱ ۰۹:۵۹:۰۰

این سربار اینقدر نیست که اهمیتی داشته باشد. فقط قرار است یک کوئری LINQ به معادل SQL آن ترجمه شود. خیلی سریع است. همچنین امکان تهیه Compiled linq queries هم وجود دارد (^) .
ضمن اینکه مثلا NHibernate قابلیتی دارد به نام second level cache که اساسا برای پروژه‌های وب طراحی شده. قابلیت کش در سطح کوئری یا اطلاعات پرکاربرد و عمومی سایت را به صورت خودکار دارد. در موردش قبلا مطلب نوشتم : (^) . سطح اول کش آن هم پیاده سازی حرفه‌ای همین باز نگه داشتن کانکشنی است که در کد SqlHelper بالا نویسنده موفق به پیاده سازی آن نشده، به علاوه کاهش رفت و آمدها به سرور: (^)
به علاوه NHibernate یک قابلیت دیگر هم دارد به نام ToFuture که می‌تونه چندین کوئری رو در طی یک رفت و برگشت برای شما انجام بده (^) .
و ... خیلی از best practices دیگر هم در آن لحاظ شده. خلاصه اینکه توانایی‌های بسیار ارزنده‌ای رو با عدم استفاده از ORMs از دست خواهید داد. منجمله همان بحث کوئری‌های پارامتری که عموما از نوشتن آن طفره می‌روند اما اینجا به صورت خودکار برای شما انجام می‌شود.

نویسنده: محمد صاحب
تاریخ: ۱۳۹۰/۱۰/۲۱ ۱۰:۱۱:۱۰

خیلی ممنون عالی بود.
در رابطه با بحث Exception تو کلاس های پایه اگه Exception مجددا Throw بشه (با اضافه شدن کمی اطلاعات) این کار تو این کلاس پایه قابل توجیه؟

یه همچین بحثی اینجا شده.
/http://social.msdn.microsoft.com/Forums/en-US/vbgeneral/thread/5092363e-649d-45a6-8ae1-e9cf9f6db867

یه سوال بی ربط:با چه روشی شما به مطالب تو کامنت ها لینک میدید من کامنتم رو از Word هم میارم لینک به مطالب حذف میشه!

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۱ ۱۰:۳۰:۱۹

- اگر مجددا throw بشه مشکلی نداره. مثلا اگر به کدهای کتابخانه NHibernate استفاده کنید، گاهی از اوقات از این روش استفاده کرده، مثلا برای ارائه پیغام واضح‌تری به کاربر اگر پیغام exception اولیه مفهوم نیست. یا می‌خواهید یک Exception کلی را لاگ کنید اما یک نمونه ساده‌تر را مثلا به دلایل امنیتی به کاربر نمایش دهید. اما حتما این لاگ کردن اولیه باید لحاظ شود چون فوق العاده در طول زمان کیفیت کد را بالا می‌برد و مشکلات را نمایان می‌کند.
در کل یک کلاس پایه تا حد امکان نباید try/catch داشته باشد. سطح‌های بالاتر باید در این مورد تصمیم گیری کنند.
- من تگ a href الی آخر را دستی درست می‌کنم.

نویسنده: Samin_c2700
تاریخ: ۱۳۹۰/۱۰/۲۱ ۱۵:۲۷:۲۲

سلام
از مطالب مفید و پربارتون ممنون .
در مورد استفاده از EF یه مشکلی که برای من وجود داره و نتونستم براش یه راه حل درست و درمون پیدا کنم نحوه‌ی استفاده از EF با سایر بانک‌های اطلاعاتی (مثلا اراکله) که خود شما هم تو نقدی که بر کتاب آقای راد نوشته بودید ذکر کردیدش. من هرچی تو نت گشتم تنها راه حلی رو که برای اینکار گفته بودند استفاده از پروایدر های شرکت های ثالث بوده.آیا برای اینکار یه راه حل استاندارد وجود داره ؟

نویسنده: Vb_asp_net
تاریخ: ۱۳۹۰/۱۰/۲۱ ۱۷:۳۹:۰۲

استاد من در یک وب سایت از متد های استاتیک استفاده کردم ولی زمانی که تعداد کاربرام خیلی زیاد شدند تداخل اطلاعات به وجود اومد ولی زمانی که اونا رو از حالت استاتیک خارج کردم دیگه همچین مشکلی به وجود نیامد.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۱ ۱۷:۴۵:۴۵

باید کدهای شما رو ببینم. اگر این بین اطلاعات اشتراکی استاتیک وجود داشته حتما مشکل‌زا بوده.

نویسنده: Vb_asp_net
تاریخ: ۱۳۹۰/۱۰/۲۱ ۱۷:۵۴:۱۴

کد من با زبان VB.Net هست که زیاد تفاوتی نداره من کد خودم رو اینجا می زارم

```
(Public Shared Function GetPAGES() As List(Of EntityPAGES
(Dim cn As New SqlConnection(SiteHelper.GetConnectionString
{Dim cmd As New SqlCommand("GET_PAGES", cn) With {.CommandType = CommandType.StoredProcedure
("" ,"")cmd.Parameters.AddWithValue'
()(Dim retlist As New List(Of EntityPAGES
Dim reader As SqlDataReader = Nothing
Try
()cn.Open
()reader = cmd.ExecuteReader
If reader.HasRows Then
Dim row As Integer = 1
()Do While reader.Read
()Dim item As New EntityPAGES
(()item.Division.Division_id = Integer.Parse(reader("Division_id").ToString
()item.Division.Name_persian = reader("DIVISION_NAME").ToString
(()item.Page_id = Integer.Parse(reader("Page_id").ToString
(()item.Page_no = Integer.Parse(reader("Page_no").ToString
(()item.Masterpage.Masterpage_id = Integer.Parse(reader("Masterpage_id").ToString
()item.Page_file_name = reader("Page_file_name").ToString
()item.Page_title = reader("Page_title").ToString
()item.Page_link = reader("Page_link").ToString
(()item.Page_delete = Boolean.Parse(reader("Page_delete").ToString
(()item.Active = Boolean.Parse(reader("Active").ToString
()item.Remark = reader("Remark").ToString
(retlist.Add(item
Loop
End If
Catch e1 As SqlException
Throw
Catch e2 As Exception
Throw
Finally
If reader IsNot Nothing Then
```

```
()reader.Close
End If
If cn.State <> ConnectionState.Closed Then
()cn.Close
()cmd.Dispose
End If
End Try
Return retlist
End Function
```

استاد مثلاً این کدی که من نوشتم اگه تعداد زیادی کاربر در حال DataEntry باشند اطلاعات اونها با هم قاطی میشه.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۱ ۱۸:۰۰:۳۸

مایکروسافت پشتیبانی از پروایدر ADO.NET مرتبط به اوراکل را چندسالی است که رسما قطع کرده و خود اوراکل این پروایدر رو داره ارائه می‌ده. در مورد EF هم به همین صورت : (^) و (^) .

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۱ ۱۸:۲۴:۴۴

نه. این متد استاتیک مشکلی نداره و مشکل از اینجا نیست. بهتر است تراکنش‌ها رو لحاظ کنید تا اطلاعات تداخل نکنند. همچنین من اینجا در رویه ذخیره شده GET_PAGES پارامتری یا آرگومانی نمی‌بینم که مقدار دهی شده باشه.
در کل برای بررسی آن نیاز به مشاهده اطلاعات بیشتری هست مثلا GET_PAGES چی هست یا اینکه این متد بالا کجا فراخوانی میشه. آیا بلافاصله بعد از ورود اطلاعات هست؟ اگر اینطور است هم تراکنش نیاز دارد و هم GET_PAGES نیاز به یک آرگومان یا پارامتر ورودی که مشخص کند، چه مواردی را باید فیلتر کند و نمایش دهد.

نویسنده: Shima2012
تاریخ: ۱۳۹۰/۱۰/۲۱ ۱۹:۱۳:۱۳

ممنون استاددر این زمینه کتابهای خوب فارسی هم سراغ دارید؟
همچنین برای یادگیری MVC چه کتاب فارسی رو پیشنهاد میدید؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۱ ۱۹:۲۸:۰۸

یک سری ویدیوی آزموشی به روز و با کیفیت در این زمینه از سایت pluralsight موجود است. در گوگل جستجو کنید قابل دریافت هستند.

نویسنده: Tahery Naser
تاریخ: ۱۳۹۰/۱۰/۲۲ ۰۰:۱۲:۱۰

مثلا طراحی Entity framework مایکروسافت به حدی ساده است که هر شخصی با داشتن بهره هوشی در حد یک عنکبوت آبی یا حتی جلبک دریایی هم می‌تونه با اون کار کنه! این جمله کاملا تایید میشه چون من یادش گرفتم و خیلی به این ORM ها علاقه مند شدم کار انجام میدن در حد معجزه.
آقای نصیری آیا منبعی فارسی برای NHibernate وجود داره؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۲ ۰۰:۱۹:۵۲

بله. اینجا: (^)

نویسنده: Vb_asp_net
تاریخ: ۱۳۹۰/۱۰/۲۲ ۱۱:۴۵:۳۶

استاد براتون مقدور هست من یک Sample کوچک براتون Email کنم که شما کدهای من رو نگاه کنید؟؟؟ چون واقعاً این مشکل واسه تمام پروژه هام خیلی حیاتی شده.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۱۰/۲۲ ۱۳:۲۷:۴۴

سلام؛ بفرست.

نویسنده: Mohsen
تاریخ: ۱۳۹۰/۱۰/۲۵ ۱۱:۰۶:۲۱

نکات واقعا ارزشمندی بود مهندس نصیری.ممنون از شما.

عنوان: **مروری بر کاربردهای Action و Func - قسمت اول**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۲۵ ۱۳:۵
آدرس: www.dotnettips.info
گروه‌ها: C#, Refactoring

delegateها، نوع‌هایی هستند که ارجاعی را به یک متد دارند؛ بسیار شبیه به function pointers در C و CPP هستند، اما برخلاف آن‌ها، delegates شیءگرا بوده، به امضای متد اهمیت داده و همچنین کد مدیریت شده و امن به شمار می‌روند.
سیر تکاملی delegates را در مثال ساده زیر می‌توان ملاحظه کرد:

```
using System;

namespace ActionFuncSamples
{
    public delegate int AddMethodDelegate(int a);
    public class DelegateSample
    {
        public void UseDelegate(AddMethodDelegate addMethod)
        {
            Console.WriteLine(addMethod(5));
        }
    }

    public class Helper
    {
        public int CustomAdd(int a)
        {
            return ++a;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Helper helper = new Helper();

            // .NET 1
            AddMethodDelegate addMethod = new AddMethodDelegate(helper.CustomAdd);
            new DelegateSample().UseDelegate(addMethod);

            // .NET 2, anonymous delegates
            new DelegateSample().UseDelegate(delegate(int a) { return helper.CustomAdd(a); });

            // .NET 3.5
            new DelegateSample().UseDelegate(a => helper.CustomAdd(a));
        }
    }
}
```

معنای کلمه delegate، واگذاری مسئولیت است. به این معنا که ما در متد UseDelegate، نمی‌دانیم addMethod به چه نحوی تعریف خواهد شد. فقط می‌دانیم که امضای آن چیست.
در دات نت یک، یک وهله از شیء AddMethodDelegate ساخته شده و سپس متدی که امضایی متناسب و متناظر با آن را داشت، به عنوان متد انجام دهنده مسئولیت معرفی می‌شد. در دات نت دو، اندکی نحوه تعریف delegates با ارائه delegates بی‌نام، ساده‌تر شد و در دات نت سه و نیم با ارائه lambda expressions ، تعریف و استفاده از delegates باز هم ساده‌تر و زیباتر گردید.
به علاوه در دات نت 3 و نیم، دو Generic delegate به نام‌های Action و Func نیز ارائه گردیده‌اند که به طور کامل جایگزین تعریف طولانی delegates در کدهای پس از دات نت سه و نیم شده‌اند. تفاوت‌های این دو نیز بسیار ساده است:
اگر قرار است واگذاری قسمتی از کد را به متدی محول کنید که مقداری را بازگشت می‌دهد، از Func و اگر این متد خروجی ندارد از Action استفاده نمائید:

```
Action<int> example1 = x => Console.WriteLine("Write {0}", x);
example1(5);

Func<int, string> example2 = x => string.Format("{0:n0}", x);
Console.WriteLine(example2(5000));
```

در دو مثال فوق، نحوه تعریف inline یک Action و یا Func را ملاحظه می‌کنید. Action به متدی اشاره می‌کند که خروجی ندارد و در اینجا تنها یک ورودی int را می‌پذیرد. Func در اینجا به تابعی اشاره می‌کند که یک ورودی int را دریافت کرده و یک خروجی string را باز می‌گرداند.

پس از این مقدمه، در ادامه قصد داریم مثال‌های دنیای واقعی Action و Func را که در سال‌های اخیر بسیار متداول شده‌اند، بررسی کنیم.

مثال یک) ساده سازی تعاریف API ارائه شده به استفاده کنندگان از کتابخانه‌های ما

عنوان شد که کار delegates، واگذاری مسئولیت انجام کاری به کلاس‌های دیگر است. این مورد شما را به یاد کاربردهای interfaceها نمی‌اندازد؟

در interfaceها نیز یک قرارداد کلی تعریف شده و سپس کدهای یک کتابخانه، تنها با امضای متدها و خواص تعریف شده در آن کار می‌کنند و کتابخانه ما نمی‌داند که این متدها قرار است چه پیاده سازی خاصی را داشته باشند.

برای نمونه طراحی API زیر را درنظر بگیرید که در آن یک interface جدید تعریف شده که تنها حاوی یک متد است. سپس کلاس Runner از این interface استفاده می‌کند:

```
using System;

namespace ActionFuncSamples
{
    public interface ISchedule
    {
        void Run();
    }

    public class Runner
    {
        public void Exceute(ISchedule schedule)
        {
            schedule.Run();
        }
    }

    public class HelloSchedule : ISchedule
    {
        public void Run()
        {
            Console.WriteLine("Just Run!");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            new Runner().Exceute(new HelloSchedule());
        }
    }
}
```

در اینجا ابتدا باید این interface را در طی یک کلاس جدید (مثلا HelloSchedule) پیاده سازی کرد و سپس حاصل را در کلاس Runner استفاده نمود.

نظر شما در مورد این طراحی ساده شده چیست؟

```
using System;

namespace ActionFuncSamples
{
    public class Schedule
    {
        public void Exceute(Action run)
        {
            run();
        }
    }

    class Program
```

```
    {
        static void Main(string[] args)
        {
            new Schedule().Exceute(() => Console.WriteLine("Just Run!"));
        }
    }
}
```

با توجه به اینکه هدف از معرفی interface در طراحی اول، واگذاری مسئولیت نحوه تعریف متد Run به کلاسی دیگر است، به همین طراحی با استفاده از یک Action delegate نیز می‌توان رسید. مهم‌ترین مزیت آن، حجم بسیار کمتر کدنویسی استفاده کننده نهایی از API تعریف شده ما است. به علاوه امکان inline coding نیز فراهم گردیده است و در همان محل تعریف Action، بدنه آن‌را نیز می‌توان تعریف کرد.

بدیهی است delegates نمی‌توانند به طور کامل جای interfaceها را پر کنند. اگر نیاز است قرارداد تهیه شده بین ما و استفاده کنندگان از کتابخانه، حاوی بیش از یک متد باشد، استفاده از interfaceها بهتر هستند.

از دیدگاه بسیاری از طراحان API، اشیاء delegate معادل interface ایی با یک متد هستند و یک وهله از delegate معادل وهله‌ای از کلاسی است که یک interface را پیاده سازی کرده‌است.

علت استفاده بیش از حد interfaceها در سایر زبان‌ها برای ابتدایی‌ترین کارها، کمبود امکانات پایه‌ای آن زبان‌ها مانند نداشتن anonymous methods، lambda expressions و anonymous delegates هستند. به همین دلیل مجبورند همیشه و در همه‌جا از interfaceها استفاده کنند.

ادامه دارد ...

نظرات خوانندگان

نویسنده: بهروز راد
تاریخ: ۱۳۹۱/۰۵/۲۵ ۱۷:۳۰

نمیشه همیشه اینطور گفت. بستگی به کاری داره که قرار هست انجام بشه. اینترفیس IComparable که فقط متد CompareTo رو داره، یک مثال نقض هست.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۲۵ ۱۷:۳۷

طراحی IComparable مربوط به زمان دات نت یک است. اگر آن زمان امکانات زبان مثل امروز بود، می‌شد از طراحی ساده‌تری استفاده کرد.
یک نمونه از طراحی‌های اخیر تیم دات نت رو میشه در WebGrid دید. در این طراحی برای نمونه جهت دریافت فرمول فرمت کردن مقدار یک cell، از Func استفاده کردن. می‌شد این رو با اینترفیس هم نوشت (چون قرار است کاری به خارج از کلاس محول شود و هر بار اطلاعاتی به آن ارسال و نتیجه‌ای جدید اخذ گردد؛ پیاده سازی آن با شما، نتیجه را فقط در اختیار WebGrid ما قرار دهید). اما جدا استفاده از آن تبدیل می‌شد به عذاب برای کاربر که به نحو زیبایی با Func و امکانات جدید زبان حل شده.

نویسنده: بهروز راد
تاریخ: ۱۳۹۱/۰۵/۲۵ ۱۸:۵۱

فکر نمی‌کنم به خاطر دات نت 1 باشه. دلیلی فراتر از این وجود داره. با کمی جستجو، این لینک که بر اساس VS 2010 نوشته شده، در پاراگراف آخر دلیل منطقی‌تری رو ارائه میده. در مورد WebGrid که فرمودید، بحثش جداست. من از کامپونت‌های متن باز Telerik در بستر ASP.NET MVC استفاده می‌کنم و از انعطاف پذیری Action و Func در متدهای اون لذت می‌برم. حرف من در مورد تعریف واجب استفاده از Predefined Delegates به جای اینترفیس‌های تک متدی است.

```
One good example of using a single-method interface instead of a delegate is

IComparable

or the generic version,

<IComparable<T

.

IComparable

declares the

CompareTo

method, which returns an integer that specifies a less than, equal to,
or greater than relationship between two objects of the same type.

IComparable

can be used as the basis of a sort algorithm. Although using a delegate
 comparison method as the basis of a sort algorithm would be valid, it
is not ideal. Because the ability to compare belongs to the class and
the comparison algorithm does not change at run time, a single-method
interface is ideal.
```

نویسنده: وحید نصیری

تاریخ: ۱۳۹۱/۰۵/۲۵ ۱۹:۱۱

- در مورد تعریف «واجب» کسی اینجا بحث نکرده. این هم یک دید طراحی است. آیا کسی می‌تونه بگه اولین طراحی مطرح شده در مطلب جاری اشتباه است؟ خیر. اما ضرورتی ندارد تا این اندازه صرفا جهت واگذاری مسئولیت انجام یک متد به کلاسی دیگر، اینقدر طراحی انجام شده زمخت و طولانی باشد.
- در متن MSDN فوق نوشته شده که استفاده از delegate در این حالت خاص نیز معتبر است؛ اما ایده‌آل نیست. دلیلی که آورده از نظر من ساختگی است. ضرورتی ندارد تعریف یک delegate معرفی شده در runtime عوض شود. یا عنوان کرده که IComparable پایه مرتب سازی یک سری از متدها است. خوب ... بله زمانیکه از روز اول اینطور طراحی کردید همه چیز به هم مرتبط خواهند بود.

پ.ن.
قسمت نظرات MSDN یک زمانی باز بود ولی ... بعد از مدتی پشیمان شدند و به نظر این قابلیت منسوخ شده در این سایت!

عنوان: **مروری بر کاربردهای Action و Func - قسمت دوم**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۲۸ ۱۸:۱۰
آدرس: www.dotnettips.info
برچسب‌ها: C#, Refactoring

در قسمت قبل از Func و Actionها برای ساده سازی طراحی‌های مبتنی بر اینترفیس‌هایی با یک متد استفاده کردیم. این مورد خصوصا در حالت‌هایی که قصد داریم به کاربر اجازه‌ی فرمول نویسی بر روی اطلاعات موجود را بدهیم، بسیار مفید است.

مثال دوم) به استفاده کننده از API کتابخانه خود، اجازه فرمول نویسی بدهید

برای نمونه مثال ساده زیر را درنظر بگیرید که در آن قرار است یک سری عدد که از منبع داده‌ای دریافت شده‌اند، بر روی صفحه نمایش داده شوند:

```
public static void PrintNumbers()
{
    var numbers = new[] { 1,2,3,5,7,90 }; // from a data source
    foreach(var item in numbers)
    {
        Console.WriteLine(item);
    }
}
```

قصد داریم به برنامه نویس استفاده کننده از کتابخانه گزارش‌سازی خود، این اجازه را بدهیم که پیش از نمایش نهایی اطلاعات، بتواند توسط فرمولی که مشخص می‌کند، فرمت اعداد نمایش داده شده را تعیین کند.
روال کار اکثر ابزارهای گزارش‌سازی موجود، ارائه یک زبان اسکریپتی جدید برای حل این نوع مسایل است. اما با استفاده از Func و ... روش‌های Code first (بجای روش‌های Wizard first)، خیلی از این رنج و دردها را می‌توان ساده‌تر و بدون نیاز به اختراع و یا آموزش زبان جدیدی حل کرد:

```
public static void PrintNumbers(Func<int,string> formula)
{
    var numbers = new[] { 1,2,3,5,7,90 };  // from a data source
    foreach(var item in numbers)
    {
        var data = formula(item);
        Console.WriteLine(data);
    }
}
```

اینبار با استفاده از Func، امکان فرمول نویسی را به کاربر استفاده کننده از API ساده گزارش ساز فرضی خود داده‌ایم. Func تعریف شده در اینجا یک عدد int را در اختیار استفاده کننده قرار می‌دهد. در این بین، برنامه نویس می‌تواند هر نوع تغییر یا هر نوع فرمولی را که مایل است بر روی این عدد به کمک دستور زبان جاری مورد استفاده، اعمال کند و در آخر تنها باید نتیجه این عملیات را به صورت یک string بازگشت دهد. برای مثال:

```
PrintNumbers(number => string.Format("{0:n0}",number));
```

البته سطر فوق ساده شده فراخوانی زیر است:

```
PrintNumbers((number) =>{ return string.Format("{0:n0}",number); });
```

به این ترتیب اعداد نهایی با جدا کننده سه رقمی نمایش داده خواهند شد.
از این نوع طراحی، در ابزارها و کتابخانه‌های جدید گزارش سازی مخصوص ASP.NET MVC زیاد مشاهده می‌شوند.

مثال سوم) حذف کدهای تکراری برنامه

فرض کنید قصد دارید در برنامه وب خود مباحث caching را پیاده سازی کنید:

```
using System;
using System.Web;
using System.Web.Caching;
using System.Collections.Generic;

namespace WebToolkit
{
    public static class CacheManager
    {
        public static void CacheInsert(this HttpContextBase httpContext, string key, object data, int
durationMinutes)
        {
            if (data == null) return;
            httpContext.Cache.Add(
                key,
                data,
                null,
                DateTime.Now.AddMinutes(durationMinutes),
                TimeSpan.Zero,
                CacheItemPriority.AboveNormal,
                null);
        }
    }
}
```

در هر قسمتی از برنامه که قصد داشته باشیم اطلاعاتی را در کش ذخیره کنیم، الگوی تکراری زیر باید طی شود:

```
var item = httpContext.Cache[key];
if (item == null)
{
    item = ReadDataFromDataSource();
    if (item == null)
          return null;

    CacheInsert(httpContext, key, item, durationMinutes);
}
```

ابتدا باید وضعیت کش جاری بررسی شود؛ اگر اطلاعاتی در آن موجود نبود، ابتدا از منبع داده‌ای مورد نظر خوانده شده و سپس در کش درج شود.
می‌توان در این الگوی تکراری، خواندن اطلاعات را از منبع داده، به یک Func واگذار کرد و به این صورت کدهای ما به نحو زیر بازسازی خواهند شد:

```
using System;
using System.Web;
using System.Web.Caching;
using System.Collections.Generic;

namespace WebToolkit
{
    public static class CacheManager
    {
        public static void CacheInsert(this HttpContextBase httpContext, string key, object data, int
durationMinutes)
        {
            if (data == null) return;
            httpContext.Cache.Add(
                key,
                data,
                null,
                DateTime.Now.AddMinutes(durationMinutes),
                TimeSpan.Zero,
                CacheItemPriority.AboveNormal,
                null);
        }

        public static T CacheRead<T>(this HttpContextBase httpContext, string key, int durationMinutes,
Func<T> ifNullRetrievalMethod)
        {
            var item = httpContext.Cache[key];
            if (item == null)
            {
```

```
                item = ifNullRetrievalMethod();
                if (item == null)
                    return default(T);

                CacheInsert(httpContext, key, item, durationMinutes);
            }
            return (T)item;
        }
    }
}
```

و استفاده از آن نیز به نحو زیر خواهد بود:

```
var user = HttpContext.CacheRead(
                            "Key1",
                            15,
                            () => _usersService.FindUser(userId));
```

پارامتر سوم متد CacheRead به صورت خودکار تنها زمانیکه اطلاعات کش متناظری با کلید Key1 وجود نداشته باشند، اجرا شده و نتیجه در کش ثبت می‌گردد. در اینجا دیگر از if و else و کدهای تکراری بررسی وضعیت کش خبری نیست.

نظرات خوانندگان

نویسنده: علی علیار
تاریخ: ۱۳۹۱/۰۶/۱۵ ۲۰:۲

سلام میشه یه توضیحی درباره کد زیر بدید؟

```
public static T CacheRead<T>(this HttpContextBase httpContext, string key, int durationMinutes, Func<T>
ifNullRetrievalMethod)
        {
            var item = httpContext.Cache[key];
            if (item == null)
            {
                item = ifNullRetrievalMethod();
                if (item == null)
                    return default(T);

                CacheInsert(httpContext, key, item, durationMinutes);
            }
            return (T)item;
        }
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۶/۱۵ ۲۰:۱۳

در متن توضیح دادم «... الگوی تکراری زیر باید طی شود ...».
برای خواندن اطلاعات از کش سیستم، این الگوی تکراری در هرجایی از برنامه باید انجام شود:
الف) ابتدا باید به شیء Cache مراجعه شود. شاید بر اساس یک key مفروض، اطلاعاتی در آن موجود باشد.
ب) اگر نبود (قطعه if تعریف شده)، باید به یک منبع داده مشخص، مراجعه و اطلاعات دریافت شود. سپس این اطلاعات در کش برای دفعات مراجعه بعدی ثبت گردد.
ج) اطلاعات نهایی بازگشت داده شود.

در اینجا قسمت مراجعه به منبع داده، توسط Func به استفاده کننده از متد CacheRead واگذار شده است. به این صورت ما فقط می‌دونیم که یک تابع در اختیار این متد قرار خواهد گرفت که در زمان مناسب می‌شود آن را فراخوانی کرد.
مثلا در مثالی که در انتهای بحث است یک نمونه از کاربرد آن‌را مشاهده می‌کنید.

عنوان: **مروری بر کاربردهای Action و Func - قسمت سوم**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۲۹ ۲۰:۱۹
آدرس: www.dotnettips.info
گروه‌ها: C#, Refactoring

در ادامه مثال سوم قسمت قبل ، در مورد حذف کدهای تکراری توسط Action و Func، در این قسمت به یک مثال نسبتا پرکاربرد دیگر آن جهت ساده سازی try/catch/finally اشاره خواهد شد.
احتمالا هزاران بار در کدهای خود چنین قطعه کدی را تکرار کرده‌اید:

```
try {
      // code
} catch(Exception ex) {
      // do something
}
```

این مورد را نیز می‌توان توسط Actionها کپسوله کرد و پیاده سازی قسمت بدنه try آن‌را به فراخوان واگذار نمود:

```
void Execute(Action action) {
    try {
       action();
    } catch(Exception ex) {
       // log errors
    }
}
```

و برای نمونه جهت استفاده از آن خواهیم داشت:

```
Execute(() => {open a file});
```

یا اگر عمل انجام شده باید خروجی خاصی را بازگرداند (برخلاف یک Action که خروجی از آن انتظار نمی‌رود)، می‌توان طراحی متد Execute را با Func انجام داد:

```
public static class SafeExecutor
{
    public static T Execute<T>(Func<T> operation)
    {
        try
        {
            return operation();
        }
        catch (Exception ex)
        {
            // Log Exception
        }
        return default(T);
    }
}
```

در این حالت فراخوانی متد Execute به نحو زیر خواهد بود:

```
var data = SafeExecutor.Execute<string>(() =>
{
    // do something
    return "result";
});
```

و اگر در این بین استثنایی رخ دهد، علاوه بر ثبت جزئیات خطای رخ داده شده، نال را بازگشت خواهد داد.

از همین دست می‌توان به کپسوله سازی منطق «سعی مجدد» در انجام کاری اشاره کرد:

```
public static class RetryHelper
{
   public static void RetryOperation(Action action, int numRetries, int retryTimeout)
   {
       if( action == null )
           throw new ArgumentNullException("action");

       do
       {
          try {  action(); return;  }
          catch
          {
              if( numRetries <= 0 ) throw;
              else
                 Thread.Sleep( retryTimeout );
           }
       } while( numRetries-- > 0 );
   }
}
```

برای مثال فرض کنید برنامه قرار است اطلاعاتی را از وب دریافت کند. ممکن است در سعی اول آن، خطای اتصال یا در دسترس نبودن لحظه‌ای سایت رخ دهد. در اینجا نیاز خواهد بود تا این عملیات چندین بار تکرار شود؛ که نمونه‌ای از آن‌را در ذیل ملاحظه می‌کنید:

```
RetryHelper.RetryOperation(() => SomeFunction(), 3, 1000);
```

نظرات خوانندگان

نویسنده: مجتبی صحرائی
تاریخ: ۱۳۹۱/۰۵/۲۹ ۲۱:۶

بسیار زیبا

نویسنده: امیر
تاریخ: ۱۳۹۱/۰۵/۲۹ ۲۲:۴۶

واقعا بحث زیبا و پرکاربردی است.

نویسنده: رضا
تاریخ: ۱۳۹۱/۰۵/۳۰ ۲۲:۲۸

واقعاً مبحث فوق العاده ای هست و شما هم عالی توضیح میدید. حذف کدهای تکراری واقعاً کمک کننده هستش.
ممنون.

نویسنده: Hamid NCH
تاریخ: ۱۳۹۲/۰۹/۱۳ ۱۵:۵۱

در مورد این توضیح میدین.خیلی ممنون

```
return default (T);
```

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۹/۱۳ ۱۷:۳۳

گاهی از اوقات حین کار با نوع‌های جنریک نیاز دارید که مثلا null بازگشت بدید. در این حالت کامپایلر شما را با خطای Cannot convert null to type parameter T متوقف می‌کند. به همین جهت مرسوم است در این حالت از default T استفاده شود که مقدار پیش فرض نوع را برمی‌گرداند. اگر reference type باشد (مثل کلاس‌ها) این مقدار پیش فرض null خواهد بود؛ اگر value type باشد مانند int صفر بازگشت داده می‌شود.

عنوان: **اهمیت Controller های ساده در ASP.NET MVC**
نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۵/۳۱ ۱۴:۳۳
آدرس: www.dotnettips.info
گروه‌ها: Refactoring, ASP.Net MVC

Controllerها به نوعی رابط بین View و Model هستند. *ساده ترین* محل برای قرار دادن کدهای تصمیم گیری (decision-making code) ، قرار دادن منطق تجاری و یا فراهم ساختن داده برای View مثل ایجاد یک لیست از Select List برای یک DropDownList می‌باشند. اما انجام این کارها به نرم افزار ما پیچیدگی تحمیل می‌کند. Controllerها باید در طول زمان توسعه‌ی یک نرم افزار کم حجم و سبک باقی بمانند. در این مطلب بحث شد که یکی از اهداف استفاده از ASP.NET MVC نوشتن نرم افزار هایی تست پذیر می‌باشد. نوشتن آزمون واحد و نگهداری Controller هایی که مسئولیت زیادی بر عهده دارند سخت می‌باشد. Controllerها باید به نحوی توسعه پیدا کنند که نگهداری آن‌ها ساده باشد ، تست پذیر باشند و اصل SRP را رعایت کرده باشند.

نگهداری آسان

کدی که درک آن سخت باشد ، تغییر دادن آن نیز سخت است ، هنگامی که درک کدی سخت باشد زمینه برای به وجود آمدن خطاها ، دوباره کاری و سردرد فراهم می‌شود. هنگامی که مسئولیت یک Action به صورت شفاف مشخص نباشد و انواع و اقسام کارها به آن سپرده شده باشد تغییر در آن سخت می‌شود ، ممکن است این تغییر باید چند جای دیگر هم داده شود در نتیجه فاز نگهداری هزینه و زمان اضافی به نرم افزار تحمیل می‌کند.

تست پذیری

بهترین راه برای اطمینان از این که درک و تست پذیری سورس کد ما ساده هست انجام و تمرین توسعه‌ی تست محور (TDD) می‌باشد. هنگامی که از روش TDD استفاده می‌کنیم با سورسی کدی کار می‌کنیم که هنوز وجود ندارد. در این مرحله از به وجود آمدن کلاس هایی که تست آن‌ها دشوار یا غیر ممکن است (به دلیل داشتن مسئولیت‌های اضافی) از جمله Controllerها جلوگیری می‌شود. نوشتن آزمون‌های واحد برای Controllerهای کم حجم ساده می‌باشد.

تمرکز بر روی یک مسئولیت

بهترین راه برای ساده سازی Controllerها گرفتن مسئولیت‌های اضافی از آن می‌باشد به Action زیر توجه کنید :

```
public RedirectToRouteResult Ship(int orderId)
{
   User user = _userSession.GetCurrentUser();
   Order order = _repository.GetById(orderId);
   if (order.IsAuthorized)
   {
      ShippingStatus status = _shippingService.Ship(order);
      if (!string.IsNullOrEmpty(user.EmailAddress))
      {
         Message message = _messageBuilder
            .BuildShippedMessage(order, user);
         _emailSender.Send(message);
      }
      if (status.Successful)
      {
         return RedirectToAction("Shipped", "Order", new {orderId});
      }
   }
   return RedirectToAction("NotShipped", "Order", new {orderId});
}
```

این Action کار زیادی انجام می‌دهد ، تقریبا می‌توان تعداد مسئولیت‌های این Action را با شمارش تعداد Ifها پیدا کرد . مسئولیت تصمیم گیری درباره این که آیا Order مورد نظر آماده برای تحویل است یا خیر به این Action سپرده شده ، همچنین تصمیم گیری درباره اینکه آیا کاربر دارای آدرس ایمیل جهت ارسال ایمیل می‌باشد یا خیر به این Action سپرده شده است. منطق‌های domain logic و business logic نباید در کلاس‌های presentation همانند Controllerها قرار داده شود. تست و نگهداری کدی مثل کد بالا دشوار خواهد بود. Refactoring که باید در این Code اعمال شود Refactor Architecture by Tiers نام دارد. این Refactoring به توسعه دهنده می‌گوید که منطقی که در لایه‌ی نمایش (Presentation) پیاده کرده را به لایه‌ی Business انتقال دهد. پس از انتقال منطق کد بالا به OrderShippingService ، کد Action ما ساده‌تر می‌شود :

```
public RedirectToRouteResult Ship(int orderId)
{
   var status = _orderShippingService.Ship(orderId);
   if (status.Successful)
```

```
    {
        return RedirectToAction("Shipped", "Order", new {orderId});
    }
    return RedirectToAction("NotShipped", "Order", new {orderId});
}
```

پس از انتقال منطق تجاری به محل مناسب خودش تنها مسئولیتی که برای برای Controller باقی مانده این است که کاربر را به کجا Redirect کند. پس از این Refactoring علاوه برا اینکه مسئولیت‌ها در جای مناسب خود قرار گرفتند ، اکنون می‌توان به سادگی منطق کار را بدون تحت تاثیر قرار گرفتن کدهای لایه‌ی نمایش تغییر داد.
در آینده به تکنیک‌های ساده سازی Controllerها خواهیم پرداخت.

نظرات خوانندگان

نویسنده: محمد صاحب
تاریخ: ۱۳۹۱/۰۵/۳۱ ۱۸:۹

ممنون موضوع جالبیه...
فصل 7وم کتاب Programming Microsoft ASP.NET MVC هم به همین موضوع اختصاص داده شده که برای سبک کردن کنترلر دوتا الگوی Responsibility-Driven Design و iPODD رو معرفی میکنه و...

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۵/۳۱ ۱۸:۱۳

ممنون می‌شم اگر درباره الگوی iPODD لینک بدید.

به این موضوع در سری کتاب‌های MVC In Action هم یک فصل کامل اختصاص داده شده .

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۵/۳۱ ۲۰:۳۵

ممنون
واقعا لذت بردیم
امیدوارم این مباحث رو همچنان ادامه بدید ...

نویسنده: رضا بزرگی
تاریخ: ۱۳۹۱/۰۵/۳۱ ۲۰:۳۶

آیا در معماری چندلایه (N-Tier arch) مرزبندی شفافی وجود دارد که از نظر شی‌گرایی روش یا روش‌های مرجح وجود داشته باشند؟
مثلا علت اینکه شما کنترل‌ها در MVC رو لایه نمایش به‌حساب میارین متوجه نشدم و مگر در واقع BL ما در کنترلرها اتفاق نمیوفته؟ یا مثلا ViewModel ما اگر با View در دولایه متفاوت هستند_(که هستند) جزو همان لایه نمایش به‌حساب میان؟
اگر برایتان مقدور است در مورد مرزهای شفاف تفکیک منطقی و فیزیکی (Layer & Tier) در MVC توضیح بدین،ممنون میشم.

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۵/۳۱ ۲۰:۴۹

طبق چیزی که من میدونم Controller محتوای لازم جهت نمایش در View رو آماده میکنه اما با لایه View متفاوت است. هدف ما از این پیاده سازی همین است (آماده سازی محتوای لازم جهت نمایش در View). با این حال Controller این محتوا را از منبع یا منابع مختلف دریافت میکند. به عبارتی اطلاعات خود را از لایه Service یا همان BL دریافت میکند و کدهای لایه Business درون Controller قرار نمیگیرند.

نویسنده: میثم
تاریخ: ۱۳۹۱/۰۵/۳۱ ۲۰:۵۵

مرسی ، جالب بود

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۶/۰۱ ۰:۳۱

Controller فقط مصرف کننده‌ی منطق نهایی است ، بدنه‌ی کنترلرها جای مناسبی برای پیاده سازی منطق تجاری نیست. BL که شما از آن یاد می‌کنید در لایه‌ی دیگری رخ می‌دهد ، یکی اسم آم را Task می‌گذارد ، یکی Service و دیگری BlaBla ..
MVC جایگزینی برای N-Tier نیست ، بلکه روشی برای سازماندهی لایه‌ی نمایش می‌باشد.
ViewModelها (ViewModel === Model Of View) اشیایی هستند که از طریق لایه‌ی سرویس تولید می‌شوند و در واقع داده ای که باید در View به کاربر نمایش داده شود را نگهداری می‌کنند.
نگرانی و مسئولیت Controller فراهم کردن داده (از طریق اجرای Business logic) برای UI می‌باشد.

نویسنده: رضا بزرگی
تاریخ: ۱۳۹۱/۰۶/۰۱ ۱:۱۴

- تصور من اینست که معماری چندلایه یک stack است از لایه‌های مختلف با وظایف متفاوت. که پایین‌ترین سطح(دیتا سورس) تا بالاترین سطح(رابط کاربری) هست. در این بین بقیه اجزا مثل BL و غیره قرار میگیره. سوالم اینه که اگر بخواهیم یه الگوی نسبتا کلی ارائه بدیم این لایه‌ها چطور روی هم چیده میشه.
- تجسم این لایه‌ها شاید از این لحاظ مهم باشه که یکی از مهمترین مفاهیم شی‌گرایی یعنی کپسوله‌سازی و Information hidding باعث تولید ایده‌ی چند لایه‌ای هست. و دانستن و اجرای درست اون خیلی کار توسعه را آسانتر میکنه.
- N-Tier یک معماری برای طراحی هست. ولی MVC یه الگوی طراحی. و این‌ها جایگزین که نه، بلکه به نوعی با هم‌پوشانی کار را اصولی‌تر میکند. همانطور که جناب نصیری در سری MVC اشاره‌ای موکد داشتند که Model در MVC در واقع همان ViewModel است . که این ViewModel از الگوی MVVM آمده.

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۶/۰۱ ۱:۴۸

-معماری n-tier یک مدل برای توسعه ی نرم افزارهای انعطاف پذیر و با قابلیت استفاده‌ی مجدد می‌باشد. MVC یک Architectural_pattern است.
-MVC در یک معماری چند لایه وظیفه‌ی لایه‌ی نمایش (Presentation) را بر عهده دارد. همانند MVP در Windows Forms یا Web Forms. http://allthingscs.blogspot.de/2011/03/mvc-vs-3-tier-pattern.html

-ViewModel الگوی MVVM با ViewModel که ما در MVC داریم تفاوت زیادی دارند و ارتباطی با هم ندارند.

نویسنده: اصغر ترابی
تاریخ: ۱۳۹۱/۰۶/۰۱ ۴:۷

با عذر خواهی
ولی متاسفانه این ایده که منطق تجاری را در کنترلر قرار دهیم کاملا غلط است.واضح‌ترین دلیل هم این است که منطق تجاری ممکن است بین چندین استفاده کننده مشترک باشد، خیلی ساده فرض کنید قرار است نسخه وب و ویندوز یک سیستم ساخته شود.

نویسنده: رضا
تاریخ: ۱۳۹۱/۰۶/۰۱ ۹:۶

با سلام، معماری لایه ای طراحی‌های متفاوتی داره
و هر طراحی تو سناریوی خاصی ممکنه پیگیری بشه
ولی دو نکته همیشه پابرجا هستش
اول این که چون لایه UI پیچیده‌ترین لایه هستش، با یکی از **الگوهای معماری** زیر سر و سامانی پیدا می‌کنه :
Model - View - Controller
Model View - View Model

Model View Presenter
نکته دوم این هستش که سعی می‌کنن معماری لایه ای رو به صورتی پیاده سازی کنند که بر **N-Tier Development** و Tierهای فیزیکی برنامه منطبق بشه
مثلا DA و BL و Service Layer در سمت سرور، و View و View Model در سمت کلاینت در یک برنامه Desktop
امیدوارم مطلبم شفافیت رو افزایش داده باشه

نویسنده: محمد صاحب
تاریخ: ۱۳۹۱/۰۶/۰۱ ۹:۲۴

iPODD سرنام Idiomatic Presentation, Orchestration, Domain and Data هست من لینک خاصی تو نت ازش ندیدم تو همون کتاب که معرفی کردم باهاش آشنا شدم...
کتاب رو اگه پیدا نکردی بگو برات میل کنم.

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۶/۰۱ ۱۱:۵

در مطلب فعلی هم سعی شده به همین موضوع پرداخته شه که منطق تجاری را در Controller قرار ندهید.
گفته شده Controller باید محل استفاده از خروجی منطق تجاری باشد.

نویسنده: امیرحسین جلوداری
تاریخ: ۱۳۹۱/۰۶/۰۲ ۰:۶

الان برای من یه سوالی پیش اومد :
طبق حرفایه شاهین و بقیه‌ی بچه‌ها الان MVC از دید معماری سه لایه داره رو لایه‌ی نمایش مانور میده ... خوب برا چی نیومدن بگن VMVC ؟! (ViewModel-View-Controller) ...چون الان Model در واقع همون ViewModel هست! (اون چیزی که تو View استفاده میشه نه Domain Model!)

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۶/۰۲ ۰:۲۵

در مطالب قبلی هم گفته شده بود که M در ASP.NET MVC همان ViewModel هست یا Model Of View .
M در الگوی MVC (الگوی MVC به صورت عمومی در همه‌ی چارچوب ها) هم تنها وظیفه‌ی تامین داده‌ی View را دارد. در صفحه‌ی Wikipedia الگوی MVC نوشته شده :

```
A view requests from the model the information that it needs to generate an output representation.
```

تعریف Domain Entity چیز دیگری هست و برای استفاده از الگوی MVC لزوما نیازی نیست که برنامه‌ی ما دیتابیس داشته باشد.
این لینک
هم مفید است.

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۸/۱۹ ۲۲:۳۱

کدهای بالا رو در نظر بگیرید؛
ممکن است در حین عملیات‌های بررسی آماده بودن یک سفارش ، بررسی معتبر بودن ایمیل کاربر و ... در صورتی که نتیجه عملیات false باشد بخواهیم پیامی به کاربر نمایش داده شود(نمایش دلیل خطا به کاربر خطا(سفارش آماده نیست - ایمیل وارد نشده و ...)). در این حالت چگونه می‌توان این عملیات را به درستی در کد پیاده کرد؟

نویسنده: شاهین کیاست

تاریخ: ۱۳۹۱/۰۸/۱۹ ۲۳:۲۷

برای نمایش پیام به کاربر در ASP.NET MVC روش‌های زیادی وجود دارد ، مثلا می‌توان در ViewModel خود یک پراپرتی جهت نمایش پیغام به کاربر تعبیه کرد و در صورت نیاز به کمک یک Helper در View پیام مورد نظر را نشان داد یا از شیء TempData استفاده کرد.

نویسنده: محسن درپرستی
تاریخ: ۱۳۹۲/۱۲/۰۶ ۲۰:۳۱

اگر طبق راهنمایی‌های همین پست پیش بریم ، من می‌تونم این روش رو پیشنهاد بدم که مقدار متغیر status اگر برابر با successful نبود . به یک متد دیگه ای ارسال بشه که مسئولیت ساخت و برگرداندن پیام مناسب رو برعهده داره . اون پیام‌ها هم بهتره که به جای اینکه در قالب یک سری if یا switch محاسبه بشن در قالب یک dictionary نگهداری بشن .

```
public RedirectToRouteResult Ship(int orderId)
{
   var status = _orderShippingService.Ship(orderId);
   if (status.Successful)
   {
      return RedirectToAction("Shipped", "Order", new {orderId});
   }
   return RedirectToAction("NotShipped", "Order", new {id = orderId, desc = status});
}
```

ولی محل فراخوانی این تابع دوم کجا باید باشه ؟ نظر من اینه که نباید توی این اکشن باشه چون وظیفه‌ی این اکشن چیز دیگری است . بهتره که مقدار status به اکشن NotShipped ارسال بشه و در اونجا پیام استخراج و به view ارسال بشه . چون در اون اکشن احتمالا دلیل ship نشدن سفارش باید به کاربر نمایش داده بشه.

عنوان: **مروری بر کاربردهای Action و Func - قسمت چهارم**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۶/۰۵ ۲۳:۵۳
آدرس: www.dotnettips.info
برچسب‌ها: C#, Refactoring

طراحی API برنامه توسط Actionها

روش مرسوم طراحی Fluent interfaces ، جهت ارائه روش ساخت اشیاء مسطح به کاربران بسیار مناسب هستند. اما اگر سعی در تهیه API عمومی برای کار با اشیاء چند سطحی مانند معرفی فایل‌های XML توسط کلاس‌های سی شارپ کنیم، اینبار Fluent interfaces آنچنان قابل استفاده نخواهند بود و نمی‌توان این نوع اشیاء را به شکل روانی با کنار هم قرار دادن زنجیر وار متدها تولید کرد. برای حل این مشکل روش طراحی خاصی در نگارش‌های اخیر NHibernate معرفی شده است به نام loquacious interface که این روزها در بسیاری از APIهای جدید شاهد استفاده از آن هستیم و در ادامه با پشت صحنه و طرز تفکری که در حین ساخت این نوع API وجود دارد آشنا خواهیم شد.

در ابتدا کلاس‌های مدل زیر را در نظر بگیرید که قرار است توسط آن‌ها ساختار یک جدول از کاربر دریافت شود:

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

namespace Test
{
    public class Table
    {
        public Header Header { set; get; }
        public IList<Cell> Cells { set; get; }
        public float Width { set; get; }
    }

    public class Header
    {
        public string Title { set; get; }
        public DateTime Date { set; get; }
        public IList<Cell> Cells { set; get; }
    }

    public class Cell
    {
        public string Caption { set; get; }
        public float Width { set; get; }
    }
}
```

در روش طراحی loquacious interface به ازای هر کلاس مدل، یک کلاس سازنده ایجاد خواهد شد. اگر در کلاس جاری، خاصیتی از نوع کلاس یا لیست باشد، برای آن نیز کلاس سازنده خاصی درنظر گرفته می‌شود و این روند ادامه پیدا می‌کند تا به خواصی از انواع ابتدایی مانند int و string برسیم:

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

namespace Test
{
    public class TableApi
    {
        public Table CreateTable(Action<TableCreator> action)
        {
            var creator = new TableCreator();
            action(creator);
            return creator.TheTable;
        }
    }

    public class TableCreator
    {
        readonly Table _theTable = new Table();
        internal Table TheTable
        {
```

```
            get { return _theTable; }
        }

        public void Width(float value)
        {
            _theTable.Width = value;
        }

        public void AddHeader(Action<HeaderCreator> action)
        {
            _theTable.Header = ...
        }

        public void AddCells(Action<CellsCreator> action)
        {
            _theTable.Cells = ...
        }
    }
}
```

نقطه آغازین API ایی که در اختیار استفاده کنندگان قرار می‌گیرد با متد CreateTable ایی شروع می‌شود که ساخت شیء جدول را به ظاهر توسط یک Action به استفاده کننده واگذار کرده است، اما توسط کلاس TableCreator او را مقید و راهنمایی می‌کند که چگونه باید اینکار را انجام دهد.

همچنین در بدنه متد CreateTable، نکته نحوه دریافت خروجی از Action ایی که به ظاهر خروجی خاصی را بر نمی‌گرداند نیز قابل مشاهده است.

همانطور که عنوان شد کلاس‌های xyzCreator تا رسیدن به خواص معمولی و ابتدایی پیش می‌روند. برای مثال در سطح اول چون خاصیت عرض از نوع float است، صرفا با یک متد معمولی دریافت می‌شود. دو خاصیت دیگر نیاز به Creator دارند تا در سطحی دیگر برای آن‌ها سازنده‌های ساده‌تری را طراحی کنیم.

همچنین باید دقت داشت که در این طراحی تمام متدها از نوع void هستند. اگر قرار است خاصیتی را بین خود رد و بدل کنند، این خاصیت به صورت internal تعریف می‌شود تا در خارج از کتابخانه قابل دسترسی نباشد و در intellisense ظاهر نشود.

مرحله بعد، ایجاد دو کلاس HeaderCreator و CellsCreator است تا کلاس TableCreator تکمیل گردد:

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

namespace Test
{
    public class CellsCreator
    {
        readonly IList<Cell> _cells = new List<Cell>();
        internal IList<Cell> Cells
        {
            get { return _cells; }
        }

        public void AddCell(string caption, float width)
        {
            _cells.Add(new Cell { Caption = caption, Width = width });
        }
    }

    public class HeaderCreator
    {
        readonly Header _header = new Header();
        internal Header Header
        {
            get { return _header; }
        }

        public void Title(string title)
        {
            _header.Title = title;
        }

        public void Date(DateTime value)
        {
            _header.Date = value;
        }

        public void AddCells(Action<CellsCreator> action)
```

```
        {
            var creator = new CellsCreator();
            action(creator);
            _header.Cells = creator.Cells;
        }
    }
}
```

نحوه ایجاد کلاس‌های Builder و یا Creator این روش بسیار ساده و مشخص است:
مقدار هر خاصیت معمولی توسط یک متد ساده void دریافت خواهد شد.
هر خاصیتی که اندکی پیچیدگی داشته باشد، نیاز به یک Creator جدید خواهد داشت.
کار هر Creator بازگشت دادن مقدار یک شیء است یا نهایتا ساخت یک لیست از یک شیء. این مقدار از طریق یک خاصیت internal بازگشت داده می‌شود.

البته عموما بجای معرفی مستقیم کلاس‌های Creator از یک اینترفیس معادل آن‌ها استفاده می‌شود. سپس کلاس Creator را internal تعریف می‌کنند تا خارج از کتابخانه قابل دسترسی نباشد و استفاده کننده نهایی فقط با توجه به متدهای void تعریف شده در interface کار تعریف اشیاء را انجام خواهد داد.

در نهایت، مثال تکمیل شده ما به شکل زیر خواهد بود:

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

namespace Test
{
    public class TableCreator
    {
        readonly Table _theTable = new Table();
        internal Table TheTable
        {
            get { return _theTable; }
        }

        public void Width(float value)
        {
            _theTable.Width = value;
        }

        public void AddHeader(Action<HeaderCreator> action)
        {
            var creator = new HeaderCreator();
            action(creator);
            _theTable.Header = creator.Header;
        }

        public void AddCells(Action<CellsCreator> action)
        {
            var creator = new CellsCreator();
            action(creator);
            _theTable.Cells = creator.Cells;
        }
    }

    public class CellsCreator
    {
        readonly IList<Cell> _cells = new List<Cell>();
        internal IList<Cell> Cells
        {
            get { return _cells; }
        }

        public void AddCell(string caption, float width)
        {
            _cells.Add(new Cell { Caption = caption, Width = width });
        }
    }

    public class HeaderCreator
    {
        readonly Header _header = new Header();
```

```
        internal Header Header
        {
            get { return _header; }
        }

        public void Title(string title)
        {
            _header.Title = title;
        }

        public void Date(DateTime value)
        {
            _header.Date = value;
        }

        public void AddCells(Action<CellsCreator> action)
        {
            var creator = new CellsCreator();
            action(creator);
            _header.Cells = creator.Cells;
        }
    }
}
```

نحوه استفاده از این طراحی نیز جالب توجه است:

```
var data = new TableApi().CreateTable(table =>
            {
                table.Width(1);
                table.AddHeader(header=>
                {
                    header.Title("new rpt");
                    header.Date(DateTime.Now);
                    header.AddCells(cells=>
                    {
                        cells.AddCell("cell 1", 1);
                        cells.AddCell("cell 2", 2);
                    });
                });
                table.AddCells(tableCells=>
                {
                    tableCells.AddCell("c 1", 1);
                    tableCells.AddCell("c 2", 2);
                });
            });
```

این نوع طراحی مزیت‌های زیادی را به همراه دارد:

الف) ساده سازی طراحی اشیاء چند سطحی و تو در تو

ب) امکان درنظر گرفتن مقادیر پیش فرض برای خواص

ج) ساده‌تر سازی تعاریف لیست‌ها

د) استفاده کنندگان در حین استفاده نهایی و تعریف اشیاء به سادگی می‌توانند کدنویسی کنند (مثلا سلول‌ها را با یک حلقه اضافه کنند).

ه) امکان بهتر استفاده از امکانات Intellisense . برای مثال فرض کنید یکی از خاصیت‌هایی که قرار است برای آن Creator درست کنید یک interface را می‌پذیرد. همچنین در برنامه خود چندین پیاده سازی کمکی از آن نیز وجود دارد. یک روش این است که مستندات قابل توجهی را تهیه کنید تا این امکانات توکار را گوشزد کند؛ روش دیگر استفاده از طراحی فوق است. در اینجا در کلاس Creator ایجاد شده چون امکان معرفی متد وجود دارد، می‌توان امکانات توکار را توسط این متدها نیز معرفی کرد و به این ترتیب Intellisense تبدیل به راهنمای اصلی کتابخانه شما خواهد شد.

نظرات خوانندگان

نویسنده: بهروز راد
تاریخ: ۱۳۹۱/۰۶/۰۶ ۱۷:۳۸

این تکنیک و مقاله، یکی از مطالب Must Read سال هست. به شخصه از این تکنیک در توسعه‌ی کامپوننت‌های ASP.NET MVC استفاده می‌کنم. کلاً تکنیک Fluent که برادر نصیری فعلاً در دو مقاله به اون پرداختند، انعطاف پذیری بسیاری به برنامه‌ها میده. مثلاً شبیه سازی روال RowDataBound کنترل GridView در Web Forms، در بستر MVC با استفاده از یک Action.
به نظر من کمبودی که ASP.NET MVC در حال حاضر داره، داشتن مجموعه ای غنی از کامپوننت‌های توکار هست که فکر می‌کنم در نسخه‌های آینده، مایکروسافت این نقیصه رو بر طرف می‌کنه، شاید با مشارکت شرکت‌های دیگه مثل Telerik.

عنوان: **مروری بر کاربردهای Action و Func - قسمت پنجم**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۶/۱۰ ۲۱:۴
آدرس: www.dotnettips.info
برچسب‌ها: C#, Refactoring

یکی دیگر از کاربردهای Action و Func، امکان حذف و بازنویسی switch statements بسیار حجیم و طولانی به نحوی شکیل است؛ و در ادامه این نوع Refactoring را بررسی خواهیم کرد.
در ابتدا مثال زیر را که از یک سوئیچ، برای انتخاب نوع حرکت و اعمال آن استفاده می‌کند، در نظر بگیرید:

```
using System;

namespace ActionFuncSamples
{
    public enum Direction
    {
        Up,
        Down,
        Left,
        Right
    }

    public class Sample5
    {
        public void Move(Direction direction)
        {
            switch (direction)
            {
                case Direction.Up:
                    MoveUp();
                    break;

                case Direction.Down:
                    MoveDown();
                    break;

                case Direction.Left:
                    MoveLeft();
                    break;

                case Direction.Right:
                    MoveRight();
                    break;
            }
        }

        private void MoveRight()
        {
            Console.WriteLine("MoveRight");
        }

        private void MoveLeft()
        {
            Console.WriteLine("MoveLeft");
        }

        private void MoveDown()
        {
            Console.WriteLine("MoveDown");
        }

        private void MoveUp()
        {
            Console.WriteLine("MoveUp");
        }
    }
}
```

یک کلاس ساده که بر اساس مقدار enum دریافتی، حرکت به چهار جهت را سبب خواهد شد. اکنون می‌خواهیم با استفاده از Actionها نحوه تعریف متدهای حرکت را به فراخوان واگذار (کلاسی بسته برای تغییر اما باز برای توسعه) و به علاوه switch را نیز کلا حذف کنیم:

```
public interface IMove
```

```
{
    Dictionary<Direction, Action> MoveMap { get; }
}

public class Move : IMove
{
    public Dictionary<Direction, Action> MoveMap
    {
        get
        {
            return new Dictionary<Direction, Action>
           {
                { Direction.Up, MoveUp},
                { Direction.Down, MoveDown},
                { Direction.Left, MoveLeft},
                { Direction.Right, MoveRight}
           };
        }
    }

    private void MoveRight()
    {
        Console.WriteLine("MoveRight");
    }

    private void MoveLeft()
    {
        Console.WriteLine("MoveLeft");
    }

    private void MoveDown()
    {
        Console.WriteLine("MoveDown");
    }

    private void MoveUp()
    {
        Console.WriteLine("MoveUp");
    }
}

public class Sample5
{
    public void NewMove(IMove move, Direction dirction)
    {
        foreach (var item in move.MoveMap)
        {
            if (item.Key == dirction)
            {
                item.Value(); //run action
                return;
            }
        }
    }
```

توضیحات:

استفاده از Dictionary برای حذف سوئیچ بسیار مرسوم است. خصوصا زمانیکه توسط switch صرفا قرار است یک مقدار ساده بازگشت داده شود. در این حالت می‌توان کل سوئیچ را حذف کرد. caseهای آن تبدیل به keyهای یک Dictionary و مقادیری که باید بازگشت دهد، تبدیل به valueهای متناظر خواهند شد.

اما در اینجا مساله اندکی متفاوت است و خروجی خاصی مد نظر نیست؛ بلکه در هر قسمت قرار است کار مفروضی انجام شود. بنابراین اینبار کلیدهای Dictionary بازهم بر اساس مقادیر caseهای تعریف شده (در اینجا enum ایی به نام Direction) و مقادیر آن نیز Action تعریف خواهند شد. همچنین برای اینکه بتوان امکان تعریف قالبی (کلاسی) را برای تعاریف متدهای متناظر با اعضای enum نیز میسر کرد، این Dictionary را در یک interface قرار داده‌ایم تا کلاس‌های پیاده سازی کننده آن، تعریف متدها را نیز دربر داشته باشند.

همانطور که ملاحظه می‌کنید، اینبار تعاریف متدهای Move از کلاس Sample5 خارج شده‌اند، به علاوه برای دسترسی به آن‌ها نیز switch ایی تعریف نشده و از Action استفاده شده است.

نهایتا اینبار متد جدید Move تعریف شده، با اینترفیس IMove کار می‌کند که یک Dictionary حاوی متدهای متناظر با اعضای enum جهت را ارائه می‌دهد.

یک نکته تکمیلی :

متد NewMove را به شکل‌های زیر نیز می‌توان پیاده سازی کرد:

```
// or ...
move.MoveMap[dirction]();

// or ...
Action action;
if(move.MoveMap.TryGetValue(dirction, out action))
   action();
```

نظرات خوانندگان

نویسنده: ایمان عبیدی
تاریخ: ۱۳۹۱/۰۶/۱۱ ۱۵:۳۸

خیلی ممنون ، من که خیلی دوست دارم این سری مقالات رو و دنبال میکنم.

کد این قسمت رو که برای تمرین خودم انجامش دادم (با یکم تغییرات ناچیز) با اجازه از جناب نصیری اتچش کردم
Sample5.rar

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۶/۱۱ ۱۷:۲۸

مرسی
مطب جالبی هست و من هم دنبال میکنم.

عنوان: **عبارت using و نحوه استفاده صحیح از آن**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۶/۱۲ ۱۳:۳۱
آدرس: www.dotnettips.info
برچسب‌ها: C#, Refactoring, iTextSharp

مثال ساده زیر را که در مورد تعریف یک کلاس Disposable و سپس استفاده از آن توسط عبارت using است را به همراه سه استثنایی که در این متدها تعریف شده است، در نظر بگیرید:

```
using System;

namespace TestUsing
{
    public class MyResource : IDisposable
    {
        public void DoWork()
        {
            throw new ArgumentException("A");
        }

        public void Dispose()
        {
            throw new ArgumentException("B");
        }
    }

    public static class TestClass
    {
        public static void Test()
        {
            using (MyResource r = new MyResource())
            {
                throw new ArgumentException("C");
                r.DoWork();
            }
        }
    }
}
```

به نظر شما قطعه کد زیر چه عبارتی را نمایش می‌دهد؟ C یا B یا A؟

```
try
{
     TestClass.Test();
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

پاسخ: برخلاف تصور (که احتمالا C است؛ چون قبل از فراخوانی متد DoWork سبب بروز استثناء شده است)، فقط B را در خروجی مشاهده خواهیم کرد!
و این دقیقا مشکلی است که در حین کار با کتابخانه iTextSharp برای اولین بار با آن مواجه شدم. روش استفاده متداول از iTextSharp به نحو زیر است:

```
using (var pdfDoc = new Document(PageSize.A4))
{
   //todo: ...
}
```

در این بین هر استثنایی رخ دهد، در لاگ‌های خطای سیستم شما تنها خطاهای مرتبط با خود iTextSharp را مشاهده خواهید کرد و نه مشکل اصلی را که در کدهای ما وجود داشته است. البته این یک مشکل عمومی است و اگر « using statement and suppressed exceptions » را در گوگل جستجو کنید به نتایج مشابه زیادی خواهید رسید.
و خلاصه نتایج هم این است:

اگر به ثبت جزئیات خطاهای سیستم اهمیت می‌دهید (یکی از مهم‌ترین مزیت‌های دات نت نسبت به بسیاری از فریم ورک‌های مشابه که حداکثر خطای 0xABC12EF را نمایش می‌دهند)، از using استفاده نکنید! using در پشت صحنه به try/finally ترجمه می‌شود و بهتر است این مورد را دستی نوشت تا اینکه کامپایلر اینکار را به صورت خودکار انجام دهد.
در اینجا باز هم به یک سری کد تکراری try/finally خواهیم رسید و همانطور که در مباحث کاربردهای Action و Func در این سایت ذکر شد، می‌توان آن‌را تبدیل به کدهایی با قابلیت استفاده مجدد کرد. یک نمونه از پیاده سازی آن‌را در این سایت « C# Using Blocks can Swallow Exceptions » می‌توانید مشاهده کنید که خلاصه آن کلاس زیر است:

```
using System;

namespace Guard
{
    public static class SafeUsing
    {
        public static void SafeUsingBlock<TDisposable>(this TDisposable disposable, Action<TDisposable>
action)
            where TDisposable : IDisposable
        {
            disposable.SafeUsingBlock(action, d => d);
        }

        internal static void SafeUsingBlock<TDisposable, T>(this TDisposable disposable, Action<T>
action, Func<TDisposable, T> unwrapper)
            where TDisposable : IDisposable
        {
            try
            {
                action(unwrapper(disposable));
            }
            catch (Exception actionException)
            {
                try
                {
                    disposable.Dispose();
                }
                catch (Exception disposeException)
                {
                    throw new AggregateException(actionException, disposeException);
                }

                throw;
            }

            disposable.Dispose();
        }
    }
}
```

برای استفاده از کلاس فوق مثلا در حالت بکارگیری iTextSharp خواهیم داشت:

```
new Document(PageSize.A4).SafeUsingBlock(pdfDoc =>
{
  //todo: ...
});
```

علاوه بر اینکه SafeUsingBlock یک سری از اعمال تکراری را کپسوله می‌کند، از AggregateException نیز استفاده کرده است (معرفی شده در دات نت 4). به این صورت چندین استثنای رخ داده نیز در سطحی بالاتر قابل دریافت و بررسی خواهند بود و استثنایی در این بین از دست نخواهد رفت.

نظرات خوانندگان

نویسنده: بهمن خلفی
تاریخ: ۱۳۹۱/۰۶/۱۳ ۷:۵۹

با سلام خدمت شما جناب نصیری
با توجه به این مطلب که زیاد هم قدیمی نیست به نظر شما استفاده از using پیشنهاد میشود یا استفاده از try catch ؟
چون بنده در برنامه‌های کاربردی تحت وب بیشتر از using استفاده میکنم و از Elmah هم برای ثبت خطاهای سیستم استفاده میکنم .

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۶/۱۳ ۸:۴۲

مطلب جاری بیشتر به شبیه سازی **finally** /try معادل using که توسط کامپایلر به صورت خودکار تولید می‌شود مرتبط است نه try/catch کلی. بحث dispose خودکار اشیاء disposable و اینکه استفاده از using به دلیلی که عنوان شد مناسب نیست. بنابراین بجای using از SafeUsingBlock استفاده کنید (شبیه سازی بهتر کاری است که کامپایلر در پشت صحنه جهت معادل سازی یا پیاده سازی using انجام می‌دهد؛ اما بدون از دست رفتن استثناهای رخ داده). مابقی را هم ELMAH انجام می‌دهد.
اگر از using استفاده کنید و ELMAH، فقط خطاهای مرتبط با مثلا iTextSharp رو در لاگ‌ها خواهید یافت؛ مثلا شیء document آن dispose شده، اما خطا و مشکل اصلی که به کدهای ما مرتبط بوده و نه iTextSharp، این میان گم خواهد شد. اما با استفاده از SafeUsingBlock ، دلیل اصلی نیز لاگ می‌شود.

نویسنده: مرادی
تاریخ: ۱۳۹۱/۰۶/۱۳ ۹:۹

البته از حق هم نمی‌شه گذشت که طراح‌های iText Sharp در این جا هم از Best Practiceها پیروی نکردند
این قاعده کلی کار هستش که در Dispose و متد Finalize خطایی ایجاد نشه
حتی چند بار فراخوانی Dispose هم نباید ایجاد خطا کنه، حتی خطای Object Disposed Exception
متاسفانه تفاوت‌های Java و .NET. تونسته رو این تیم که iText Sharp رو از Java به .NET. برده، رو به یک سری اشتباهات بندازه، مثل این مورد، و عدم استفاده از Enum و چند تا مورد دیگه
اگه فرض کنیم ما Dispose رو فراخونی نکنیم و این فراخونی توسط CLR و در فاینالایزر کلاس رخ بده، این خطایی موجود در Dispose می‌تونه مسائل بدتری رو هم در پی داشته باشه
به دوستان توصیه می‌کنم حتما با Best Practiceهای مدیریت خطا مخصوص .NET. آشنا بشن، که در اینترنت منابع زیادی برای این مهم موجوده

نویسنده: افشین
تاریخ: ۱۳۹۱/۰۶/۱۳ ۱۳:۱۰

واقا خسته نباشید.
هر روز بابات این سایت شما رو دعا میکنیم.
ببخشید این موضوع رو این جا مطرح میکنم چون راه دیگه ای پیدا نکردم

نویسنده: سید امیر سجادی
تاریخ: ۱۳۹۲/۰۲/۰۵ ۹:۴۹

سلام. من این مشکلی که گفتید رو توی VB.NET تست کردم مشکلی نداشت. آیا .NET این مشکل رو داره و یا فقط C#?

نویسنده: وحید نصیری

تاریخ: ۱۳۹۲/۰۲/۰۵ ۱۰:۱۴

این رفتار در VB.NET هم قابل مشاهده است:

```
Public Class MyResource
    Implements IDisposable
    Public Sub DoWork()
        Throw New ArgumentException("A")
    End Sub

    Public Overloads Sub Dispose() Implements System.IDisposable.Dispose
        Throw New ArgumentException("B")
    End Sub
End Class

Public NotInheritable Class TestClass
    Private Sub New()
    End Sub
    Public Shared Sub Test()
        Using r As New MyResource()
            Throw New ArgumentException("C")
            r.DoWork()
        End Using
    End Sub
End Class
Module Module1

    Sub Main()
        Try
            TestClass.Test()
        Catch ex As Exception
            Console.WriteLine(ex.Message)
        End Try
    End Sub

End Module
```

عبارت نمایش داده شده در اینجا هم B است.

عنوان: **مثالی از الگوی Delegate Dictionary**
نویسنده: محمد صاحب
تاریخ: ۱۳۹۱/۰۸/۱۹ ۱۵:۲۵
آدرس: www.dotnettips.info
برچسب‌ها: C#, Design patterns, Refactoring

این الگو چیز جدیدی نیست و قبلا تو سری مطالب « مروری بر کاربردهای Action و Func » دربارش مطلب نوشته شده و...
البته با توجه به جدید بودن این الگو اسم واحدی براش مشخص نشده ولی تو این مطلب «الگوی Delegate Dictionary» معرفی شده که بنظرم از بقیه بهتره.
به طور خلاصه این الگو میگه اگه قراره براساس شرایط (ورودی) خاصی کار خاصی انجام بشه بجای استفاده از IF و Switch از FuncوDictionary یا Action استفاده کنیم.

برای مثال فرض کنید مدلی به شکل زیر داریم

```
public class Person
{
    public int Id { get; set; }
    public Gender Gender { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

قراره براساس جنسیت(**شرایط**) شخص اعتبارسنجی متفاوتی(**کار خاص**) رو انجام بدیدم.مثلا در اینجا قراره چک کنیم اگه شخص مرد بود اسم زنونه انتخاب نکرده باشه و...
خب روش معمول به این شکل میتونه باشه

```
switch (person.Gender)
{
    case Gender.Male:
        if (IsMale(person.FirstName))
        {
            //Isvalid
        }
        break;
    case Gender.Female:
        if (IsFemale(person.FirstName))
        {
            //Isvalid
        }
        break;
}
```

خب این روش خوب جواب میده ولی باید در حد توان استفاده از IF و Switch رو کم کرد.مثلا تو همین مثال ما اصل Open/Closed رو نقض کردیم فکر کنید قرار باشه اعتبارسنجی دیگه ای از همین دست به این کد(کلاس) اضافه بشه باید تغیرش بدیم پس این کد(کلاس) برای تغییر بسته نیست.در اینجور موارد «الگوی Delegate Dictionary» به کار ما میاد.
ما میایم توابع مورد نظرمون رو داخل یک Dictionary ذخیره میکنیم.

```
var genderFuncs = new Dictionary<Gender, Func<string, bool>>
                {
                    {Gender.Male , (x) => IsMale(x)},
                    {Gender.Female , (x) => IsFemale(x)}
                };
```

فرض کنید پیاده سازی توابع به شکل زیر باشه

```
public static bool IsMale(string name)
{
    //check...
    return true;
}
public static bool IsFemale(string name)
```

```
{
    //check...
    if (name == "Farzad")
    {
        return false;
    }
    return true;
}
```

نحوه استفاده

```
var dummyPerson = new List<Person>
                {
                    new Person
                        {Id = 1, Gender = Gender.Male, FirstName = "Mohammad", LastName = "Saheb"},
                    new Person
                        {Id = 2, Gender = Gender.Female, FirstName = "Farzad", LastName = "Mojidi"}
                };

foreach (var person in dummyPerson)
{
    bool isValid = genderFuncs[person.Gender].Invoke(person.FirstName);
}
```

با همین روش میشه قسمت آخر مقاله ی خوب آقای کیاست رو هم Refactor کرد.

```
var query = context.Students.AsQueryable();
  if (searchByName)
  {
      query= query.FindStudentsByName(name);
  }
  if (orderByAge)
  {
      query = query.OrderByAge();
  }
  if (paging)
  {
     query =  query.SkipAndTake(skip, take);
  }
  return query.ToList();
```

توابع رو داخل یک دیکشنری ذخیره میکنیم

```
var searchTypeFuncs = new Dictionary<SearchType, Func<IQueryable<Student>, string,
IQueryable<Student>>>
                    {
                        {SearchType.FirstName, (x, y) => x.FindStudentsByName(y)},
                        {SearchType.LastName, (x, y) => x.FindStudentsByLastName(y)}
                    };
```

نحوه استفاده

```
public static IList<Student> SearchStudents(IQueryable<Student> students, SearchType type, string
keyword)
{
    var result = searchTypeFuncs[type].Invoke(students, keyword);
    return result.ToList();
}
```

عنوان: **آموزش Code Contracts**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۱/۱۱/۲۳ ۸:۲۵
آدرس: www.dotnettips.info
برچسب‌ها: Refactoring, Code Contracts

احتمالا تا حالا شده که می‌خواستید متدهایی بنویسید که داده‌های ورودی رو چک کنند و از درست بودن مقادیر اطمینان حاصل کنید و احتمالا کدهای شما هم مثل نمونه پایین هستش

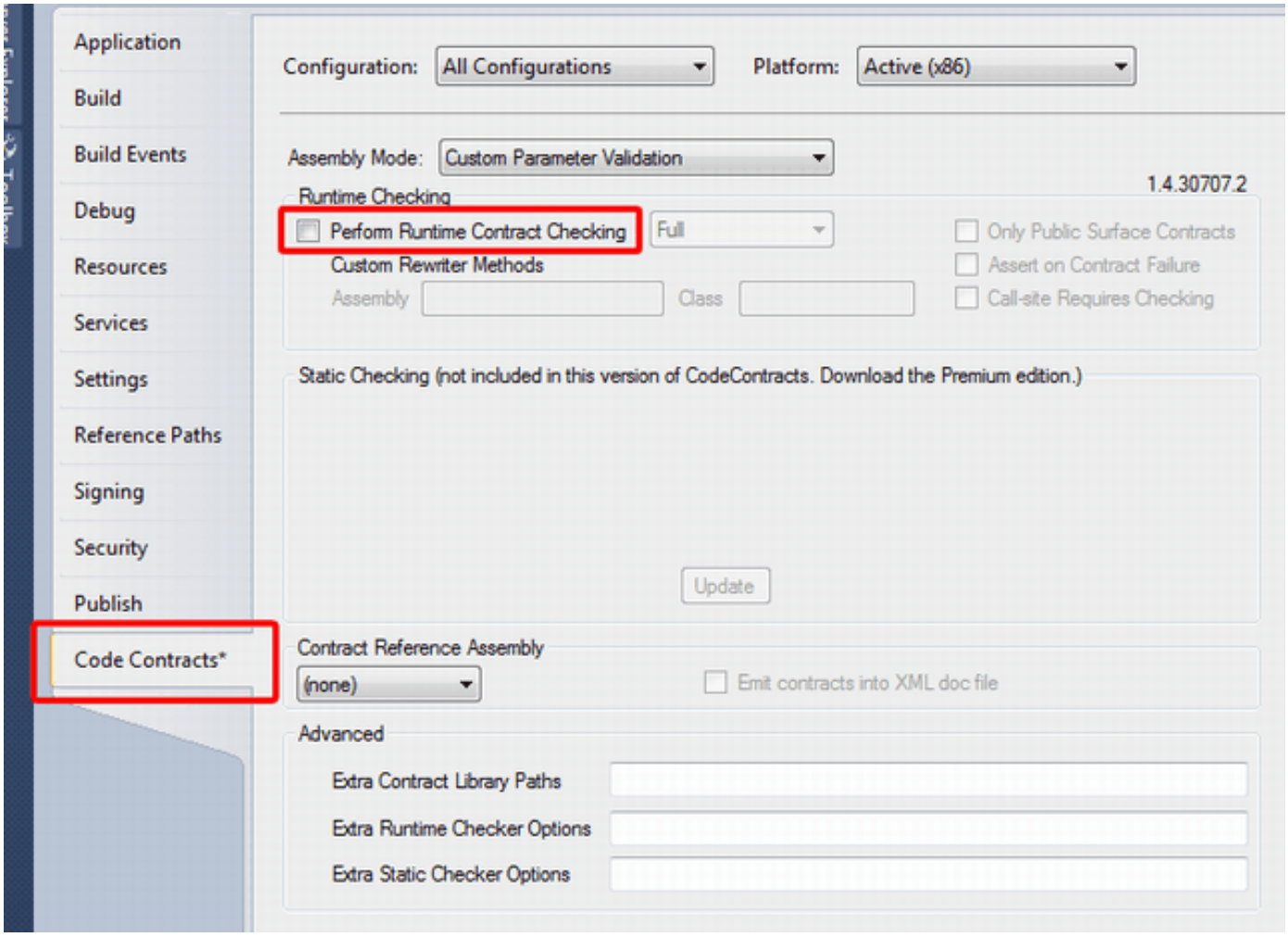
```
public class timeclock
    {
        public void clockin( int32 id, datetime clockdate )
        {
            if ( id < 0 )
            {
                throw new argumentoutofrangeexception( "..." );
            }

            if ( clockdate.date != datetime.now.date )
            {
                throw new argumentexception( "..." );
            }
        }
        public dailyreport getdailyreport( int32 employeeid, datetime fordate )
        {
            var dailyreport = new dailyreport();
            return dailyreport;
        }
    }

    public class dailyreport
    {
        public int32 employeeid { get; set; }
        public int32 hoursworked { get; set; }
    }
```

code contracts در واقع تهیه یک سری قرارداد برای اطمینان از پیاده سازی شروط در برنامه هستش و نکته مهمش اینه که شما را در هنگام کامپایل از این خطاهای احتمالی آگاه میکنه. Microsoft code contract ابزاری برای پیاده سازی این روشه و باید فایلشو دانلود کرده و بر روی visual studio نصب کنید که از این جا می‌تونید دانلودش کنید. بعد از دانلود و نصب، یک قسمت به project properties اضافه می‌شه. اون قسمتی که قرمز رنگ هست برای اجرای قراردادها به صورت runtime هستش و از combo کنار برای تعیین نوع checking استفاده میشه.

نکته: برای استفاده از این روش اگر از net4 به پایین استفاده می‌کنید باید فضای نام microsoft.contracts را به پروژه اضافه کنید ولی برای .net4 به بالا نیازی به این کار نیست. چون کلاس‌های مربوطه در فضای نام system.diagnostics.contracts قرار دارند.

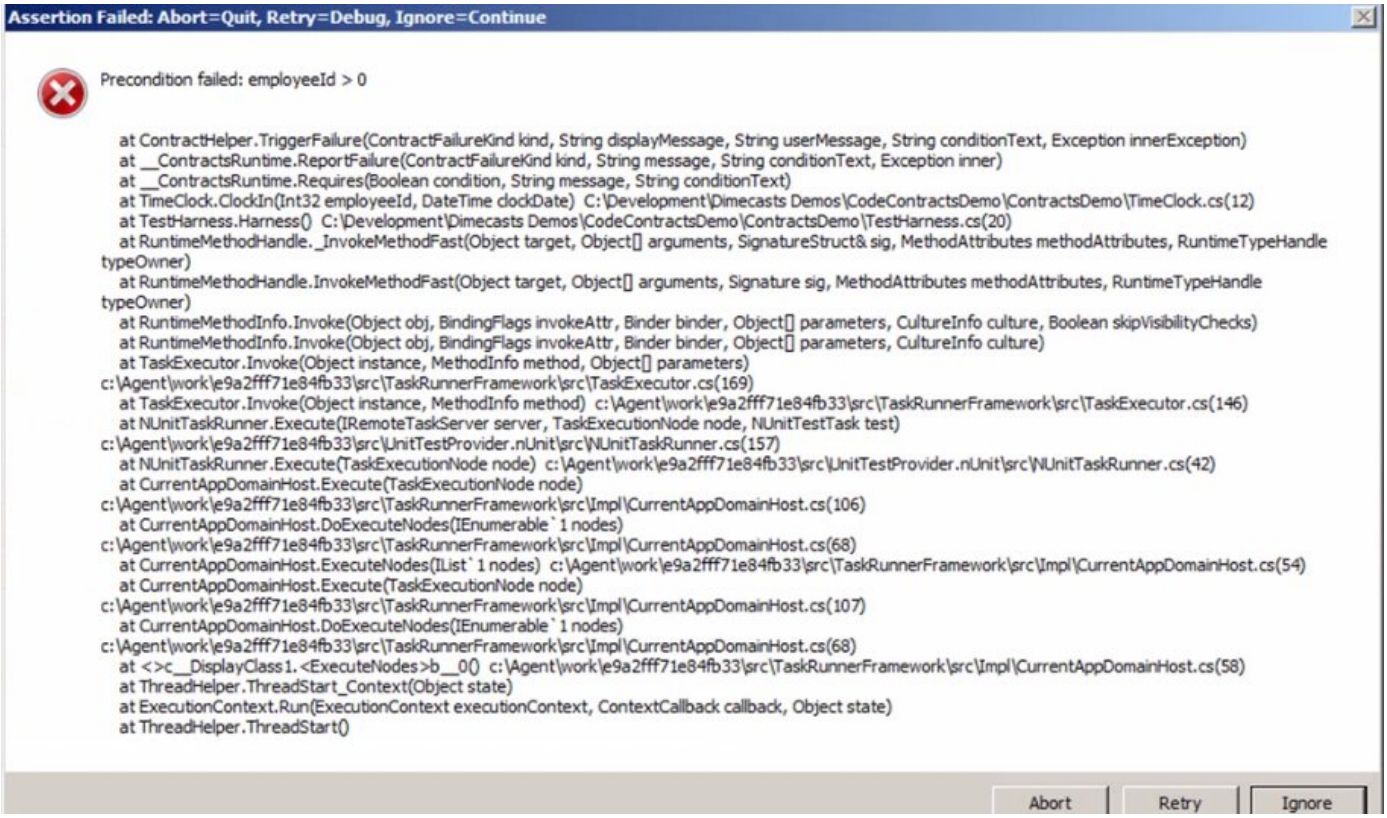


حالا مثال بالا رو به روش زیر پیاده سازی می‌کنیم

```
public void clockin( int32 id, datetime clockdate )
        {
            contract.requires<argumentoutofrangeexception>( id < 0 );
            contract.requires<argumentexception>( clockdate.date != datetime.now.date );
            contract.endcontractblock();
        }
```

فرق این روش با روش قبلی اینه که اگر در برنامه متد clockin رو به روش پایین استفاده کنیم، در هنگام اجرای برنامه با خطای زیر متوقف و رو برو میشیم

```
var timeclock = new timeclock();
  timeclock.clockin( -1, datetime.now );
```



در مطالب بعدی بیشتر به این مورد می‌پردازم

نظرات خوانندگان

نویسنده: داریوش
تاریخ: ۱۳۹۲/۰۸/۱۳ ۱۲:۳۱

مطلب عالی بود. آیا از این راه میشه برای پیاده سازی یک Business Rule Engine استفاده کرد؟

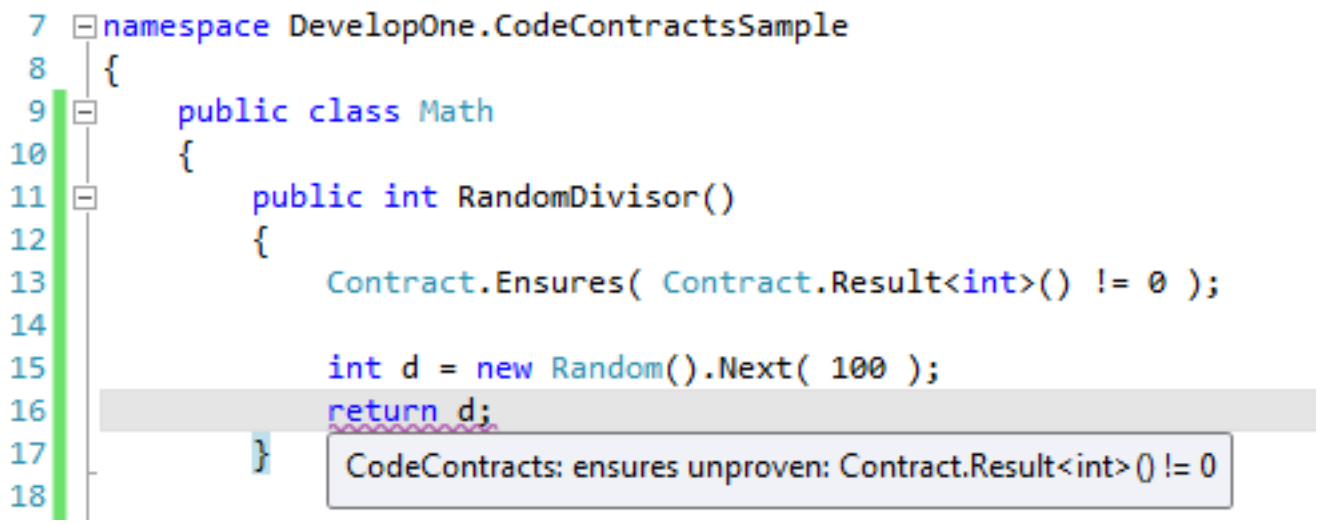
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۸/۱۳ ۱۷:۴

BRE سیستمی است برای تهیه Business Rule توسط شخصی غیر برنامه نویس. در حالی که Code Contract در فاز توسعه نرم افزار مورد استفاده قرار میگرد و فقط به شما در بهتر توسعه دادن سیستم کمک می‌کند. برای مثال:

```
7  namespace DevelopOne.CodeContractsSample
8  {
9      public class Math
10     {
11         public double Divide(int number, int divisor)
12         {
13             Contract.Requires<ArgumentOutOfRangeException>(
14                 divisor != 0,
15                 "Divide by zero is not allowed." );
16
17             return number / divisor;
18         }
19
20         public void Test()
21         {
22             int n = 10;
23             int d = 0;
24
25             var x = Divide( n, d );
26         }
27     }
28 }
29
```

CodeContracts: requires is false: divisor != 0 (Divide by zero is not allowed.)

همان طور که مشاهده می‌کنید با استفاده از تعریف Contract قبل از اجرای برنامه برای ما مشخص خواهد شد مقدار پیش فرض 0 برای متغیر d درست نیست در واقع اصلا این کد کامپایل نمی‌شود.



یا در مثال بالا مشخص شده است که مقدار d ممکن است که برابر صفر باشد و این با Contract تعریف شده مطابقت ندارد. در نتیجه در تهیه یک سیستم BRE کمک خاصی به شما نخواهد کرد.

به این نکته نیز توجه داشته باشید که با تمام مزیت هایی که Code Contracts در اختیار ما قرار می‌دهد، زمان کامپایل پروژه را به شدت افزایش خواهد داد به طوری که در یک Solution نسبتا بزرگ آزار دهنده است.

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.