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.