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.