

طراحی یک معماری خوب و مناسب یکی از عوامل مهم تولید یک برنامه کاربردی موفق می‌باشد. بنابراین انتخاب یک ساختار مناسب به منظور تولید برنامه کاربردی بسیار مهم و تا حدودی نیز سخت است. در اینجا یاد خواهیم گرفت که چگونه یک طراحی مناسب را انتخاب نماییم. همچنین روش‌های مختلف تولید برنامه‌های کاربردی را که مطمئناً شما هم از برخی از این روشها استفاده نمودید را بررسی می‌نماییم و مزایا و معایب آن را نیز به چالش می‌کشیم.

ضد الگو (Antipattern) - رابط کاربری هوشمند (Smart UI)

با استفاده از Visual Studio یا به اختصار VS ، می‌توانید برنامه‌های کاربردی را به راحتی تولید نمایید. طراحی رابط کاربری به آسانی عمل کشیدن و رها کردن (Drag & Drop) کنترل‌ها بر روی رابط کاربری قابل انجام است. همچنین در پشت رابط کاربری (Code Behind) تمامی عملیات مربوط به مدیریت رویدادها، دسترسی به داده‌ها، منطق تجاری و سایر نیازهای برنامه کاربردی، کد نویسی خواهند شد. مشکل این نوع کدنویسی بدین شرح است که تمامی نیازهای برنامه در پشت رابط کاربری قرار می‌گیرند و موجب تولید کدهای تکراری، غیر قابل تست، پیچیدگی کدنویسی و کاهش قابلیت استفاده مجدد از کد می‌گردد.

به این روش کد نویسی Smart UI می‌گویند که موجب تسهیل تولید برنامه‌های کاربردی می‌گردد. اما یکی از مشکلات عمده‌ی این روش، کاهش قابلیت نگهداری و پشتیبانی و عمر کوتاه برنامه‌های کاربردی می‌باشد که در برنامه‌های بزرگ به خوبی این مشکلات را حس خواهید کرد.

از آنجایی که تمامی برنامه نویسان مبتدی و تازه کار، از جمله من و شما در روزهای اول برنامه نویسی، به همین روش کدنویسی می‌کردیم، لزومی به ارائه مثال در رابطه با این نوع کدنویسی نمی‌بینم.

تفکیک و جدا سازی اجزای برنامه کاربردی (Separating Your Concern)

راه حل رفع مشکل Smart UI ، لایه بندی یا تفکیک اجزای برنامه از یکدیگر می‌باشد. لایه بندی برنامه می‌تواند به شکل‌های مختلفی صورت بگیرد. این کار می‌تواند توسط تفکیک کدها از طریق فضای نام (Namespace) ، پوشه بندی فایل‌های حاوی کد و یا جداسازی کدها در پروژه‌های متفاوت انجام شود. در شکل زیر نمونه ای از معماری لایه بندی را برای یک برنامه کاربردی بزرگ می‌بینید.



به منظور پیاده سازی یک برنامه کاربردی لایه بندی شده و تفکیک اجزای برنامه از یکدیگر، مثالی را پیاده سازی خواهیم کرد. ممکن است در این مثال با مسائل جدید و شیوه‌های پیاده سازی جدیدی مواجه شوید که این نوع پیاده سازی برای شما قابل درک نباشد. اگر کمی صبر پیشه نمایید و این مجموعه‌ی آموزشی را پیگیری کنید، تمامی مسائل نامانوس با جزئیات بیان خواهند شد و درک آن برای شما ساده خواهد گشت. قبل از شروع این موضوع را هم به عرض برسانم که علت اصلی این نوع پیاده سازی، انعطاف پذیری بالای برنامه کاربردی، پشتیبانی و نگهداری آسان، قابلیت تست پذیری با استفاده از ابزارهای تست، پیاده سازی پروژه بصورت تیمی و تقسیم بخشهای مختلف برنامه بین اعضای تیم و سایر مزایای فوق العاده آن می‌باشد.

1- Visual Studio را باز کنید و یک Solution خالی با نام SoCPatterns.Layered ایجاد نمایید.

• جهت ایجاد Solution خالی، پس از انتخاب New Project، از سمت چپ گزینه Other Project Types و سپس Visual Studio Solutions را انتخاب نمایید. از سمت راست گزینه Blank Solution را انتخاب کنید.

2- بر روی Solution کلیک راست نموده و از گزینه Add > New Project یک پروژه Class Library با نام SoCPatterns.Layered.Repository ایجاد کنید.

3- با استفاده از روش فوق سه پروژه Class Library دیگر با نامهای زیر را به Solution اضافه کنید:

SoCPatterns.Layered.Model

SoCPatterns.Layered.Service

SoCPatterns.Layered.Presentation

4- با توجه به نیاز خود یک پروژه دیگر را باید به Solution اضافه نمایید. نوع و نام پروژه در زیر لیست شده است که شما باید با توجه به نیاز خود یکی از پروژههای موجود در لیست را به Solution اضافه کنید.

(Windows Forms Application (SoCPatterns.Layered.WinUI

(WPF Application (SoCPatterns.Layered.WpfUI

(ASP.NET Empty Web Application (SoCPatterns.Layered.WebUI

(ASP.NET MVC 4 Web Application (SoCPatterns.Layered.MvcUI

5- بر روی پروژه SoCPatterns.Layered.Repository کلیک راست نمایید و با انتخاب گزینه Add Reference به پروژهی SoCPatterns.Layered.Model ارجاع دهید.

6- بر روی پروژه SoCPatterns.Layered.Service کلیک راست نمایید و با انتخاب گزینه Add Reference به پروژههای SoCPatterns.Layered.Repository و SoCPatterns.Layered.Model ارجاع دهید.

7- بر روی پروژه SoCPatterns.Layered.Presentation کلیک راست نمایید و با انتخاب گزینه Add Reference به پروژههای SoCPatterns.Layered.Service و SoCPatterns.Layered.Model ارجاع دهید.

8- بر روی پروژهی UI خود به عنوان مثال SoCPatterns.Layered.WebUI کلیک راست نمایید و با انتخاب گزینه Add Reference به پروژههای SoCPatterns.Layered.Model ، SoCPatterns.Layered.Repository ، SoCPatterns.Layered.Service و SoCPatterns.Layered.Presentation ارجاع دهید.

9- بر روی پروژهی UI خود به عنوان مثال SoCPatterns.Layered.WebUI کلیک راست نمایید و با انتخاب گزینه Set as StartUp Project پروژهی اجرایی را مشخص کنید.

10- بر روی Solution کلیک راست نمایید و با انتخاب گزینه Add > New Solution Folder پوشه‌های زیر را اضافه نموده و پروژه‌های مرتبط را با عمل Drag & Drop در داخل پوشه‌ی مورد نظر قرار دهید.

UI .1

SoCPatterns.Layered.WebUI §

Presentation Layer .2

SoCPatterns.Layered.Presentation §

Service Layer .3

SoCPatterns.Layered.Service §

Domain Layer .4

SoCPatterns.Layered.Model §

Data Layer .5

SoCPatterns.Layered.Repository §

توجه داشته باشید که پوشه بندی برای مرتب سازی لایه ها و دسترسی راحت تر به آنها می باشد.

پیاده سازی ساختار لایه بندی برنامه به صورت کامل انجام شد. حال به پیاده سازی کدهای مربوط به هر یک از لایه ها و بخش ها می پردازیم و از لایه Domain شروع خواهیم کرد.

نظرات خوانندگان

نویسنده: آرمان فرقانی
تاریخ: ۲۰:۲ ۱۳۹۱/۱۲/۲۸

مباحثی از این دست بسیار مفید و ضروری است و به شدت استقبال می‌کنم از شروع این سری مقالات. البته پیش‌تر هم مطالبی از این دست در سایت ارائه شده است که امیدوارم این سری مقالات بتونه تا حدی پراکندگی مطالب مربوطه را از بین ببرد. فقط لطف بفرمایید در این سری مقالات مرز بندی مشخصی برای برخی مفاهیم در نظر داشته باشید. به عنوان مثال گاهی در یک مقاله مفهوم Repository معادل مفهوم لایه سرویس در مقاله دیگر است. یا Domain Model مرز مشخصی با View Model داشته باشد. همچنین بحث‌های خوبی مهندس نصیری عزیز در مورد عدم نیاز به ایجاد Repository در مفهوم متداول در هنگام استفاده از EF داشتند که در رفرنس‌های معتبر دیگری هم مشاهده می‌شود. لطفاً در این مورد نیز بحث بیشتری با مرز بندی مشخص داشته باشید.

نویسنده: حسن
تاریخ: ۲۲:۵ ۱۳۹۱/۱۲/۲۸

آیا صرفاً تعریف چند ماژول مختلف برنامه را لایه بندی می‌کند و ضمانتی است بر لایه بندی صحیح، یا اینکه استفاده از الگوهای MVC و MVVM می‌توانند ضمانتی باشند بر جدا سازی حداقل لایه نمایشی برنامه از لایه‌های دیگر، حتی اگر تمام اجزای یک برنامه داخل یک اسمبلی اصلی قرار گرفته باشند؟

نویسنده: میثم خوشبخت
تاریخ: ۰:۵۳ ۱۳۹۱/۱۲/۲۹

این سری مقالات جمع بندی کامل معماری لایه بندی نرم افزار است. پس از پایان مقالات یک پروژه کامل رو با معماری منتخب پیاده سازی میکنم تا تمامی شک و شبهات برطرف بشه. در مورد مرز بندی لایه‌ها هم صحبت می‌کنم و مفهوم هر کدام را دقیقاً توضیح میدم.

نویسنده: میثم خوشبخت
تاریخ: ۰:۵۹ ۱۳۹۱/۱۲/۲۹

اگر مقاله فوق رو با دقت بخونید متوجه میشوید که MVC و MVVM در لایه UI پیاده سازی میشن. البته در MVC لایه Model رو به Domain Model و Repository در برخی مواقع لایه Controller رو در لایه Presentation قرار میدن. در MVVM نیز لایه Model در Domain Model و Repository و لایه View Model نیز در لایه Presentation قرار میگیره. همچنین View Model‌ها نیز در لایه Service قرار میگیرن.

در مورد ماژول بندی هم اگر در مقاله خونده باشید میتونید لایه‌ها رو از طریق پوشه‌ها، فضای نام و یا پروژه‌ها از هم جدا کنید

نویسنده: حسن
تاریخ: ۱۰:۱۴ ۱۳۹۱/۱۲/۲۹

شما در مطلبتون با ضدالگو شروع کردید و عنوان کردید که روش code behind یک سری مشکلاتی رو داره. سؤال من هم این بود که آیا صرفاً تعریف چند ماژول جدید می‌تواند ضمانتی باشد بر رفع مشکل code behind یا اینکه با این ماژول‌ها هم نهایتاً همان مشکل قبل پابرجا است یا می‌تواند پابرجا باشد.

ضمن اینکه تعریف شما از لایه دقیقاً چی هست؟ به نظر فقط تعریف یک اسمبلی در اینجا لایه نام گرفته.

نویسنده: آرمان فرقانی
تاریخ: ۱۱:۵۸ ۱۳۹۱/۱۲/۲۹

صحبت شما کاملاً صحیح است و صرفاً با ماژولار کردن به معماری چند لایه نمی‌رسیم. اما نویسنده مقاله نیز چنین نگفته و در پایان مقاله بحث پایان ساختار چند لایه است و نه پایان پروژه. این قسمت اول این سری مقالات است و قطعاً در هنگام پیاده سازی کدهای هر لایه مباحثی مطرح خواهد شد تا تضمین مفهوم مورد نظر شما باشد.

نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۱/۱۲/۲۹ ۱۲:۳۸

با تشکر از دوست عزیزم جناب آقای آرمان فرقانی با توضیحی که دادند. یکی از دلایل این شیوه کد نویسی امکان تست نویسی برای هر یک از لایه‌ها و همچنین استقلال لایه‌ها از هم دیگه هست که هر لایه بتونه بدون وجود لایه‌ی دیگه تست بشه. ماژولار کردنه ممکنه مشکل Smart UI رو حل کنه و ممکنه حل نکنه. بستگی به شیوه کد نویسی داره.

نویسنده: بهروز
تاریخ: ۱۳۹۱/۱۲/۲۹ ۱۳:۱۱

وقتی نظرات زیر مطلب شما رو میخونم میفهمم که نیاز به این سری آموزشی که دارید ارائه میدید چقدر زیاد احساس میشه فقط می‌خواستم بگم بر سر این مبحثی که دارید ارائه می‌دید اختلاف بین علما زیاد است! (حتی در عمل و در شرکت‌های نرم افزاری که تا به حال دیدم چه برسد در سطح آموزش...) امیدوارم این حساسیت رو در نظر بگیرید و همه ما پس از مطالعه این سری آموزشی به فهم مشترک و یکسانی در مورد مفاهیم موجود برسیم فکر می‌کنم وجود یک پروژه برای دست یافتن به این هدف هم ضروری باشد باز هم تشکر

نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۱/۱۲/۲۹ ۱۳:۵۹

من هم وقتی کار بر روی این معماری رو شروع کردم با مشکلات زیادی روبرو بودم و خیلی از مسائل برای من هم نامانوس و غیر قابل هضم بود. ولی بعد از اینکه چند پروژه نرم افزاری رو با این معماری پیاده سازی کردم فهم بیشتری نسبت به اون پیدا کردم و خیلی از مشکلات موجود رو با دقت بالا و با در نظر گرفتن تمامی الگوها رفع کردم. امیدوارم این حس مشترک بوجود بیاد. ولی دلیل اصلی ایجاد تکنولوژی‌ها و معماری‌های جدید اختلاف نظر بین علماست. این اختلاف نظر در اکثر مواقع میتونه مفید باشه. ممنون دوست عزیز

نویسنده: مسعود مشهدی
تاریخ: ۱۳۹۲/۰۱/۰۴ ۱۸:۳۳

با سلام
بابت مطالبتون سپاسگذارم
همون طور که خودتون گفتید نظرات و شیوه‌های متفاوتی در نوع لایه بندی‌ها وجود داره.
در مقام مقایسه لایه بندی زیر چه وجه اشتراک و تفاوتی با لایه بندی شما داره.

Application.Web

Application.Manager

Application.DAL

Application.DTO

Application.Core

با تشکر

نویسنده: محمود راستین
تاریخ: ۱۳۹۴/۰۵/۲۵ ۳:۳۰

با سلام

ممنون بابت به اشتراک گذاری این مطلب.

میخواستم بدونم چرا ما باید Presentation Layer رو ایجاد کنیم ؟ شما در Presentation Layer اومدید MVP Pattern رو پیاده سازی کردید ، مثلا اگر من از MVC استفاده کنم مگه هر دوی اونها (یعنی MVP و MVC) یک presentation pattern نیستند. پس چرا باید از هردو استفاده کنیم ؟

فرمودید برای آزمون واحد راحتتر هستیم وقتی Presentation Layer رو ایجاد کنیم ، مگر MVC این قابلیت رو نداره ؟

نویسنده: محسن خان
تاریخ: ۱۳۹۴/۰۵/۲۵ ۸:۳۶

این قسمت اول بود و یک طرح کلی غیر وابسته به فناوری خاصی. در قسمت سوم آن به مثال وب فرمها می‌رسید.

نویسنده: محمود راستین
تاریخ: ۱۳۹۴/۰۵/۲۵ ۱۴:۴۷

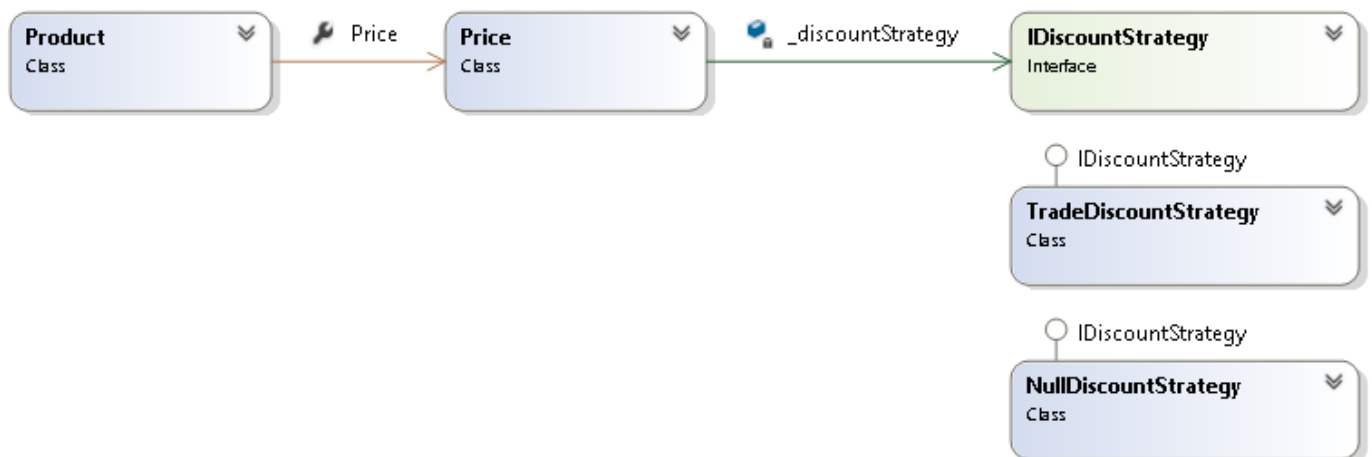
من قسمت سوم رو هم خوندم. اونجا هم به MVC اشاره ای نشده. اگر در Web Form از الگوی MVP استفاده بشه منطقی به نظر میرسه ولی سوال من این بود وقتی از MVC استفاده کنیم باز هم باید از MVP استفاده کنیم ؟ آیا این منطقیه ؟

Business Layer یا Domain Model

پیاده سازی را از منطق تجاری یا Business Logic آغاز می‌کنیم. در روش کد نویسی Smart UI ، منطق تجاری در Code Behind قرار می‌گرفت اما در روش لایه بندی، منطق تجاری و روابط بین داده‌ها در Domain Model طراحی و پیاده سازی می‌شوند. در مطالب بعدی راجع به Domain Model و الگوهای پیاده سازی آن بیشتر صحبت خواهیم کرد اما بصورت خلاصه این لایه یک مدل مفهومی از سیستم می‌باشد که شامل تمامی موجودیت‌ها و روابط بین آنهاست.

الگوی Domain Model جهت سازماندهی پیچیدگی‌های موجود در منطق تجاری و روابط بین موجودیت‌ها طراحی شده است.

شکل زیر مدلی را نشان می‌دهد که می‌خواهیم آن را پیاده سازی نماییم. کلاس Product موجودیتی برای ارائه محصولات یک فروشگاه می‌باشد. کلاس Price جهت تشخیص قیمت محصول، میزان سود و تخفیف محصول و همچنین استراتژی‌های تخفیف با توجه به منطق تجاری سیستم می‌باشد. در این استراتژی همکاران تجاری از مشتریان عادی تفکیک شده اند.



Domain Model را در پروژه SoCPatterns.Layered.Model پیاده سازی می‌کنیم. بنابراین به این پروژه یک Interface به نام IDiscountStrategy را با کد زیر اضافه نمایید:

```

public interface IDiscountStrategy
{
    decimal ApplyExtraDiscountsTo(decimal originalSalePrice);
}
  
```

علت این نوع نامگذاری Interface فوق، انطباق آن با الگوی Strategy Design Pattern می‌باشد که در مطالب بعدی در مورد این الگو بیشتر صحبت خواهیم کرد. استفاده از این الگو نیز به این دلیل بود که این الگو مختص الگوریتم‌هایی است که در زمان اجرا قابل انتخاب و تغییر خواهند بود.

توجه داشته باشید که معمولا نام Design Pattern انتخاب شده برای پیاده سازی کلاس را بصورت پسوند در انتهای نام کلاس ذکر می‌کنند تا با یک نگاه، برنامه نویس بتواند الگوی مورد نظر را تشخیص دهد و مجبور به بررسی کد نباشد. البته به دلیل تشابه برخی از الگوها، امکان تشخیص الگو، در پاره ای از موارد وجود ندارد و یا به سختی امکان پذیر است.

الگوی Strategy یک الگوریتم را قادر می‌سازد تا در داخل یک کلاس کپسوله شود و در زمان اجرا به منظور تغییر رفتار شی، بین رفتارهای مختلف سوئیچ شود.

حال باید دو کلاس به منظور پیاده سازی روال تخفیف ایجاد کنیم. ابتدا کلاسی با نام TradeDiscountStrategy را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```
public class TradeDiscountStrategy : IDiscountStrategy
{
    public decimal ApplyExtraDiscountsTo(decimal originalSalePrice)
    {
        return originalSalePrice * 0.95M;
    }
}
```

سپس با توجه به الگوی Null Object کلاسی با نام NullDiscountStrategy را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```
public class NullDiscountStrategy : IDiscountStrategy
{
    public decimal ApplyExtraDiscountsTo(decimal originalSalePrice)
    {
        return originalSalePrice;
    }
}
```

از الگوی Null Object زمانی استفاده می‌شود که نمی‌خواهید و یا در برخی مواقع نمی‌توانید یک نمونه (Instance) معتبر را برای یک کلاس ایجاد نمایید و همچنین مایل نیستید که مقدار Null را برای یک نمونه از کلاس برگردانید. در مباحث بعدی با جزئیات بیشتری در مورد الگوها صحبت خواهیم کرد.

با توجه به استراتژی‌های تخفیف کلاس Price را ایجاد کنید. کلاسی با نام Price را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```
public class Price
{
    private IDiscountStrategy _discountStrategy = new NullDiscountStrategy();
    private decimal _rrp;
    private decimal _sellingPrice;
    public Price(decimal rrp, decimal sellingPrice)
    {
        _rrp = rrp;
        _sellingPrice = sellingPrice;
    }
    public void SetDiscountStrategyTo(IDiscountStrategy discountStrategy)
    {
        _discountStrategy = discountStrategy;
    }
    public decimal SellingPrice
    {
        get { return _discountStrategy.ApplyExtraDiscountsTo(_sellingPrice); }
    }
}
```

```

    }
    public decimal Rrp
    {
        get { return _rrp; }
    }
    public decimal Discount
    {
        get {
            if (Rrp > SellingPrice)
                return (Rrp - SellingPrice);
            else
                return 0;
        }
    }
    public decimal Savings
    {
        get{
            if (Rrp > SellingPrice)
                return 1 - (SellingPrice / Rrp);
            else
                return 0;
        }
    }
}

```

کلاس Price از نوعی Dependency Injection به نام Setter Injection در متد SetDiscountStrategyTo استفاده نموده است که استراتژی تخفیف را برای کالا مشخص می‌نماید. نوع دیگری از Dependency Injection با نام Constructor Injection وجود دارد که در مباحث بعدی در مورد آن بیشتر صحبت خواهیم کرد.

جهت تکمیل لایه Model ، کلاس Product را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```

public class Product
{
    public int Id {get; set;}
    public string Name { get; set; }
    public Price Price { get; set; }
}

```

موجودیت‌های تجاری ایجاد شدند اما باید روشی اتخاذ نمایید تا لایه Model نسبت به منبع داده ای بصورت مستقل عمل نماید. به سرویسی نیاز دارید که به کلاینت‌ها اجازه بدهد تا با لایه مدل در ارتباط باشند و محصولات مورد نظر خود را با توجه به تخفیف اعمال شده برای رابط کاربری برگردانند. برای اینکه کلاینت‌ها قادر باشند تا نوع تخفیف را مشخص نمایند، باید یک نوع شمارشی ایجاد کنید که به عنوان پارامتر ورودی متد سرویس استفاده شود. بنابراین نوع شمارشی CustomerType را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```

public enum CustomerType
{
    Standard = 0,
    Trade = 1
}

```

برای اینکه تشخیص دهیم کدام یک از استراتژی‌های تخفیف باید بر روی قیمت محصول اعمال گردد، نیاز داریم کلاسی را ایجاد کنیم تا با توجه به CustomerType تخفیف مورد نظر را اعمال نماید. کلاسی با نام DiscountFactory را با کد زیر ایجاد نمایید:

```

public static class DiscountFactory
{

```

```

public static IDiscountStrategy GetDiscountStrategyFor
(CustomerType customerType)
{
    switch (customerType)
    {
        case CustomerType.Trade:
            return new TradeDiscountStrategy();
        default:
            return new NullDiscountStrategy();
    }
}
}

```

در طراحی کلاس فوق از الگوی Factory استفاده شده است. این الگو یک کلاس را قادر می‌سازد تا با توجه به شرایط، یک شی معتبر را از یک کلاس ایجاد نماید. همانند الگوهای قبلی، در مورد این الگو نیز در مباحث بعدی بیشتر صحبت خواهیم کرد.

لایه‌ی سرویس با برقراری ارتباط با منبع داده‌ای، داده‌های مورد نیاز خود را بر می‌گرداند. برای این منظور از الگوی Repository استفاده می‌کنیم. از آنجایی که لایه Model باید مستقل از منبع داده‌ای عمل کند و نیازی به شناسایی نوع منبع داده‌ای ندارد، جهت پیاده‌سازی الگوی Repository از Interface استفاده می‌شود. یک Interface به نام IProductRepository را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```

public interface IProductRepository
{
    IList<Product> FindAll();
}

```

الگوی Repository به عنوان یک مجموعه‌ی در حافظه (In-Memory Collection) یا انباره‌ای از موجودیت‌های تجاری عمل می‌کند که نسبت به زیر بنای ساختاری منبع داده‌ای کاملاً مستقل می‌باشد.

کلاس سرویس باید بتواند استراتژی تخفیف را بر روی مجموعه‌ای از محصولات اعمال نماید. برای این منظور باید یک Collection سفارشی ایجاد نماییم. اما من ترجیح می‌دهم از Extension Methods برای اعمال تخفیف بر روی محصولات استفاده کنم. بنابراین کلاسی به نام ProductListExtensionMethods را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```

public static class ProductListExtensionMethods
{
    public static void Apply(this IList<Product> products,
                            IDiscountStrategy discountStrategy)
    {
        foreach (Product p in products)
        {
            p.Price.SetDiscountStrategyTo(discountStrategy);
        }
    }
}

```

الگوی Separated Interface تضمین می‌کند که کلاینت از پیاده‌سازی واقعی کاملاً نامطلع می‌باشد و می‌تواند برنامه نویسی را به سمت Abstraction و Dependency Inversion به جای پیاده‌سازی واقعی سوق دهد.

حال باید کلاس Service را ایجاد کنیم تا از طریق این کلاس، کلاینت با لایه Model در ارتباط باشد. کلاسی به نام ProductService را با کد زیر به پروژه SoCPatterns.Layered.Model اضافه کنید:

```
public class ProductService
{
    private IProductRepository _productRepository;
    public ProductService(IProductRepository productRepository)
    {
        _productRepository = productRepository;
    }
    public IList<Product> GetAllProductsFor(CustomerType customerType)
    {
        IDiscountStrategy discountStrategy =
            DiscountFactory.GetDiscountStrategyFor(customerType);
        IList<Product> products = _productRepository.FindAll();
        products.Apply(discountStrategy);
        return products;
    }
}
```

در اینجا کدنویسی منطق تجاری در Domain Model به پایان رسیده است. همانطور که گفته شد، لایه‌ی Business یا همان Domain Model به هیچ منبع داده‌ای خاصی وابسته نیست و به جای پیاده‌سازی کدهای منبع داده‌ای، از Interface ها به منظور برقراری ارتباط با پایگاه داده استفاده شده است. پیاده‌سازی کدهای منبع داده‌ای را به لایه‌ی Repository واگذار نمودیم که در بخش‌های بعدی نحوه پیاده‌سازی آن را مشاهده خواهید کرد. این امر موجب می‌شود تا لایه Model درگیر پیچیدگی‌ها و کد نویسی‌های منبع داده‌ای نشود و بتواند به صورت مستقل و فارغ از بخش‌های مختلف برنامه تست شود. لایه بعدی که می‌خواهیم کد نویسی آن را آغاز کنیم، لایه‌ی Service می‌باشد.

در کد نویسی‌های فوق از الگوهای طراحی (Design Patterns) متعددی استفاده شده است که به صورت مختصر در مورد آنها صحبت کردم. اصلاً جای نگرانی نیست، چون در مباحث بعدی به صورت مفصل در مورد آنها صحبت خواهیم کرد. در ضمن، ممکن است روال یادگیری و آموزش بسیار نامفهوم باشد که برای فهم بیشتر موضوع، باید کدها را بصورت کامل تست نموده و مثالهایی را پیاده‌سازی نمایید.

نظرات خوانندگان

نویسنده: سینا کردی
تاریخ: ۱۳۹۱/۱۲/۳۰ ۴:۱۰

سلام
ممنون از شما این بخش هم کامل و زیبا بود
ولی کمی فشرده بود
لطفا اگر ممکن هست در مورد معماری ها و الگوها و بهترین های آنها کمی توضیح دهید یا منبعی معرفی کنید تا این الگوها و معماری
برای ما بیشتر مفهوم بشه
من در این زمینه تازه کارم و از شما میخوام که من رو راهنمایی کنید که چه مقدماتی در این زمینه ها نیاز دارم
باز هم ممنون.

نویسنده: علی
تاریخ: ۱۳۹۱/۱۲/۳۰ ۹:۱۲

در همین سایت مباحث [الگوهای طراحی](#) و [Refactoring](#) مفید هستند.

و یا الگوهای طراحی Agile رو هم [در اینجا](#) می تونید پیگیری کنید.

نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۱/۱۲/۳۰ ۱۱:۳۸

فشرده گی این مباحث بخاطر این بود که میخواستم فعلا یک نمونه پروژه رو آموزش بدم تا یک شمای کلی از کاری که می خواهیم
انجام بدیم رو ببینید. در مباحث بعدی این مباحث رو بازتر می کنم. خود من برای مطالعه و جمع بندی این مباحث منابع زیادی رو
مطالعه کردم. واقعا برای بعضی مباحث همیشه به یک منبع اکتفا کرد.

نویسنده: محسن.د
تاریخ: ۱۳۹۱/۱۲/۳۰ ۱۷:۱

بسیار عالی

آیا فراخوانی مستقیم تابع SetDiscountStrategyTo کلاس Price در تابع الحاقی Apply از نظر کپسوله سازی مورد اشکال نیست
؟ بهتر نیست که برای خود کلاس Product یک تابع پیاده سازی کنیم که در درون خودش تابع Price.SetDiscountStrategyTo را
فراخوانی کند و به این شکل کلاس های بیرونی رو از تغییرات درونی کلاس Product مستقل کنیم ؟

نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۱/۱۲/۳۰ ۱۸:۱

دوست عزیزم. متد Apply یک Extension Method برای `ICollection<Product>` است. اگر این متد تعریف نمی شد شما باید در کلاس
سرویس حلقه foreach رو قرار می دادید. البته با این حال در قسمت هایی از طراحی کلاسها که الگوهای طراحی را زیر سوال
نمی برد و تست پذیری را دچار مشکل نمی کند، طراحی سلیقه ای است. مقاله من هم آیهی نازل شده نیست که دستخوش تغییرات
نشود. شما می توانید با سلیقه و دید فنی خود تغییرات مورد نظر رو اعمال کنید. ولی اگر نظر من را بخواهید این طراحی مناسب تر
است.

نویسنده: رضا عرب
تاریخ: ۱۳۹۲/۰۱/۰۹ ۱۴:۴۵

خسته نباشید، واقعا ممنونم آقای خوشبخت، لطفا به نگارش این دست مطالب مرتبط با طراحی ادامه دهید، زمینه بکریه که کمتر عملی به آن پرداخته شده و این نوع نگارش شما فراتر از یک معرفیه که واقعا جای تشکر داره.

نویسنده: f.tahan36
تاریخ: ۱۷:۱۰ ۱۳۹۲/۰۲/۲۹

با سلام

تفاوت factory با design factory در چیست؟ (با مثال کد)

و virtual کردن یک تابع معمولی با virtual کردن تابع سازنده چه تفاوتی دارد؟

با تشکر

نویسنده: محسن خان
تاریخ: ۰:۴۰ ۱۳۹۲/۰۲/۳۰

از همون رندهایی هستی که تمرین کلاسیت رو آوردی اینجا؟! :

Service Layer

نقش لایه‌ی سرویس این است که به عنوان یک مدخل ورودی به برنامه کاربردی عمل کند. در برخی مواقع این لایه را به عنوان لایه‌ی Facade نیز می‌شناسند. این لایه، داده‌ها را در قالب یک نوع داده‌ای قوی (Strongly Typed) به نام View Model، برای لایه‌ی Presentation فراهم می‌کند. کلاس View Model یک Strongly Typed محسوب می‌شود که نماهای خاصی از داده‌ها را که متفاوت از دید یا نمای تجاری آن است، بصورت بهینه ارائه می‌نماید. در مورد الگوی View Model در مباحث بعدی بیشتر صحبت خواهیم کرد.

الگوی Facade یک Interface ساده را به منظور کنترل دسترسی به مجموعه‌ای از Interface‌ها و زیر سیستم‌های پیچیده ارائه می‌کند. در مباحث بعدی در مورد آن بیشتر صحبت خواهیم کرد.

کلاسی با نام ProductViewModel را با کد زیر به پروژه SoCPatterns.Layered.Service اضافه کنید:

```
public class ProductViewModel
{
    Public int ProductId {get; set;}
    public string Name { get; set; }
    public string Rrp { get; set; }
    public string SellingPrice { get; set; }
    public string Discount { get; set; }
    public string Savings { get; set; }
}
```

برای اینکه کلاینت با لایه‌ی سرویس در تعامل باشد باید از الگوی Request/Response Message استفاده کنیم. بخش Request توسط کلاینت تغذیه می‌شود و پارامترهای مورد نیاز را فراهم می‌کند. کلاسی با نام ProductListRequest را با کد زیر به پروژه SoCPatterns.Layered.Service اضافه کنید:

```
using SoCPatterns.Layered.Model;

namespace SoCPatterns.Layered.Service
{
    public class ProductListRequest
    {
        public CustomerType CustomerType { get; set; }
    }
}
```

در شی Response نیز بررسی می‌کنیم که درخواست به درستی انجام شده باشد، داده‌های مورد نیاز را برای کلاینت فراهم می‌کنیم و همچنین در صورت عدم اجرای صحیح درخواست، پیام مناسب را به کلاینت ارسال می‌نماییم. کلاسی با نام ProductListResponse را با کد زیر به پروژه SoCPatterns.Layered.Service اضافه کنید:

```
public class ProductListResponse
{
    public bool Success { get; set; }
}
```

```
public string Message { get; set; }
public IList<ProductViewModel> Products { get; set; }
}
```

به منظور تبدیل موجودیت Product به ProductViewModel، به دو متد نیاز داریم، یکی برای تبدیل یک Product و دیگری برای تبدیل لیستی از Product. شما می‌توانید این دو متد را به کلاس Product موجود در Domain Model اضافه نمایید، اما این متدها نیاز واقعی منطق تجاری نمی‌باشند. بنابراین بهترین انتخاب، استفاده از Extension Method ها می‌باشد که باید برای کلاس Product و در لایه‌ی سرویس ایجاد نمایید. کلاسی با نام ProductMapperExtensionMethods را با کد زیر به پروژه SoCPatterns.Layered.Service اضافه کنید:

```
public static class ProductMapperExtensionMethods
{
    public static ProductViewModel ConvertToProductViewModel(this Model.Product product)
    {
        ProductViewModel productViewModel = new ProductViewModel();
        productViewModel.ProductId = product.Id;
        productViewModel.Name = product.Name;
        productViewModel.RRP = String.Format("{0:C}", product.Price.RRP);
        productViewModel.SellingPrice = String.Format("{0:C}", product.Price.SellingPrice);
        if (product.Price.Discount > 0)
            productViewModel.Discount = String.Format("{0:C}", product.Price.Discount);
        if (product.Price.Savings < 1 && product.Price.Savings > 0)
            productViewModel.Savings = product.Price.Savings.ToString("#%");
        return productViewModel;
    }
    public static IList<ProductViewModel> ConvertToProductListViewModel(
        this IList<Model.Product> products)
    {
        IList<ProductViewModel> productViewModels = new List<ProductViewModel>();
        foreach (Model.Product p in products)
        {
            productViewModels.Add(p.ConvertToProductViewModel());
        }
        return productViewModels;
    }
}
```

حال کلاس ProductService را جهت تعامل با کلاس سرویس موجود در Domain Model و به منظور برگرداندن لیستی از محصولات و تبدیل آن به لیستی از ProductViewModel، ایجاد می‌نماییم. کلاسی با نام ProductService را با کد زیر به پروژه SoCPatterns.Layered.Service اضافه کنید:

```
public class ProductService
{
    private Model.ProductService _productService;
    public ProductService(Model.ProductService ProductService)
    {
        _productService = ProductService;
    }
    public ProductListResponse GetAllProductsFor(
        ProductListRequest productListRequest)
    {
        ProductListResponse productListResponse = new ProductListResponse();
        try
        {
            IList<Model.Product> productEntities =
                _productService.GetAllProductsFor(productListRequest.CustomerType);
            productListResponse.Products = productEntities.ConvertToProductListViewModel();
            productListResponse.Success = true;
        }
        catch (Exception ex)
        {
            // Log the exception...
            productListResponse.Success = false;
            // Return a friendly error message
        }
    }
}
```



```


        productListResponse.Message = ex.Message;
    }
    return productListResponse;
}

```

کلاس Service تمامی خطاها را دریافت نموده و پس از مدیریت خطا، پیغامی مناسب را به کلاینت ارسال می‌کند. همچنین این لایه محل مناسبی برای Log کردن خطاها می‌باشد. در اینجا کد نویسی لایه سرویس به پایان رسید و در ادامه به کدنویسی Data Layer می‌پردازیم.

Data Layer

برای ذخیره سازی محصولات، یک بانک اطلاعاتی با نام Shop01 ایجاد کنید که شامل جدولی به نام Product با ساختار زیر باشد:

Product			
	Column Name	Data Type	Allow Nulls
	ProductId	int	<input type="checkbox"/>
	ProductName	nvarchar(50)	<input type="checkbox"/>
	Rrp	smallmoney	<input type="checkbox"/>
	SellingPrice	smallmoney	<input type="checkbox"/>
			<input type="checkbox"/>

برای اینکه کدهای بانک اطلاعاتی را سریعتر تولید کنیم از روش Linq to SQL در Data Layer استفاده می‌کنیم. برای این منظور یک Data Context برای Linq to SQL به این لایه اضافه می‌کنیم. بر روی پروژه SoCPatterns.Layered.Repository کلیک راست نمایید و گزینه Add > New Item را انتخاب کنید. در پنجره ظاهر شده و از سمت چپ گزینه Data و سپس از سمت راست گزینه Linq to SQL Classes را انتخاب نموده و نام آن را Shop.dbml تعیین نمایید.

از طریق پنجره Server Explorer به پایگاه داده مورد نظر متصل شوید و با عمل Drag & Drop جدول Product را به بخش Design کشیده و رها نمایید.



اگر به یاد داشته باشید، در لایه Model برای برقراری ارتباط با پایگاه داده از یک Interface به نام `IProductRepository` استفاده نمودیم. حال باید این Interface را پیاده سازی نماییم. کلاسی با نام `ProductRepository` را با کد زیر به پروژه `SoCPatterns.Layered.Repository` اضافه کنید:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using SoCPatterns.Layered.Model;

namespace SoCPatterns.Layered.Repository
{
    public class ProductRepository : IProductRepository
    {
        public IList<Model.Product> FindAll()
        {
            var products = from p in new ShopDataContext().Products
                           select new Model.Product
                           {
                               Id = p.ProductId,
                               Name = p.ProductName,
                               Price = new Model.Price(p.Rrp, p.SellingPrice)
                           };
            return products.ToList();
        }
    }
}
```

در متد `FindAll`، با استفاده از دستورات `Linq to SQL`، لیست تمامی محصولات را برگرداندیم. کدنویسی لایه `Data` هم به پایان رسید و در ادامه به کدنویسی لایه `Presentation` و `UI` می‌پردازیم.

Presentation Layer

به منظور جداسازی منطق نمایش (Presentation) از رابط کاربری (User Interface) ، از الگوی Model View Presenter یا همان MVP استفاده می‌کنیم که در مباحث بعدی با جزئیات بیشتری در مورد آن صحبت خواهیم کرد. یک Interface با نام IProductListView را با کد زیر به پروژه SoCPatterns.Layered.Presentation اضافه کنید:

```
using SoCPatterns.Layered.Service;

public interface IProductListView
{
    void Display(IList<ProductViewModel> Products);
    Model.CustomerType CustomerType { get; }
    string ErrorMessage { set; }
}
```

این Interface توسط Web Form های ASP.NET و یا Win Form ها باید پیاده سازی شوند. کار با Interface ها موجب می‌شود تا تست View ها به راحتی انجام شوند. کلاسی با نام ProductListPresenter را با کد زیر به پروژه SoCPatterns.Layered.Presentation اضافه کنید:

```
using SoCPatterns.Layered.Service;

namespace SoCPatterns.Layered.Presentation
{
    public class ProductListPresenter
    {
        private IProductListView _productListView;
        private Service.ProductService _productService;
        public ProductListPresenter(IProductListView ProductListView,
            Service.ProductService ProductService)
        {
            _productService = ProductService;
            _productListView = ProductListView;
        }
        public void Display()
        {
            ProductListRequest productListRequest = new ProductListRequest();
            productListRequest.CustomerType = _productListView.CustomerType;
            ProductListResponse productResponse =
                _productService.GetAllProductsFor(productListRequest);
            if (productResponse.Success)
            {
                _productListView.Display(productResponse.Products);
            }
            else
            {
                _productListView.ErrorMessage = productResponse.Message;
            }
        }
    }
}
```

کلاس Presenter وظیفه‌ی واکنشی داده‌ها، مدیریت رویدادها و بروزرسانی UI را دارد. در اینجا کدنویسی لایه‌ی Presentation به پایان رسیده است. از مزایای وجود لایه‌ی Presentation این است که تست نویسی مربوط به نمایش داده‌ها و تعامل بین کاربر و سیستم به سهولت انجام می‌شود بدون آنکه نگران دشواری Unit Test نویسی Web Form ها باشید. حال می‌توانید کد نویسی مربوط به UI را انجام دهید که در ادامه به کد نویسی در Win Forms و Web Forms خواهیم پرداخت.

نظرات خوانندگان

نویسنده: محسن
تاریخ: ۱۸:۲۹ ۱۳۹۲/۰۱/۰۲

ممنون از زحمات شما.

چند سؤال و نظر:

- با تعریف الگوی مخزن به چه مزیتی دست پیدا کردید؟ برای مثال آیا هدف این است که کدهای پیاده سازی آن، با توجه به وجود اینترفیس تعریف شده، شاید روزی با مثلاً NHibernate تعویض شود؟ در عمل متاسفانه حتی پیاده سازی LINQ اینها هم متفاوت است و من تابحال در عمل ندیدم که ORM یک پروژه بزرگ رو عوض کنند. یعنی تا آخر و تا روزی که پروژه زنده است با همان انتخاب اول سر می‌کنند. یعنی شاید بهتر باشه قسمت مخزن و همچنین سرویس یکی بشن.
- چرا لایه سرویس تعریف شده از یک یا چند اینترفیس مشتق نمی‌شود؟ اینطوری تهیه تست برای اون ساده‌تر میشه. همچنین پیاده سازی‌ها هم وابسته به یک کلاس خاص نمی‌شن چون از اینترفیس دارن استفاده می‌کنند.
- این اشیاء Request و Response هم در عمل به نظر نوعی ViewModel هستند. درسته؟ اگر اینطوره بهتر یک مفهوم کلی دنبال بشه تا سردرگمی‌ها رو کمتر کنه.

یک سری نکته جانبی هم هست که می‌تونه برای تکمیل بحث جالب باشه:

- مثلاً الگوی Context per request بجای نوشتن new ShopDataContext بهتر استفاده بشه تا برنامه در طی یک درخواست در یک تراکنش و اتصال کار کنه.
- در مورد try/catch و استفاده از اون بحث زیاد هست. خیلی‌ها توصیه می‌کنن که یا اصلاً استفاده نکنید یا استفاده از اون‌ها رو به بالاترین لایه برنامه موکول کنید تا این وسط کرش یک قسمت و بروز استثناء در اون، از ادامه انتشار صدمه به قسمت‌های بعدی جلوگیری کنه.

نویسنده: میثم خوشبخت
تاریخ: ۲۳:۳۵ ۱۳۹۲/۰۱/۰۲

محسن عزیز. از شما ممنونم که به نکته‌های ظریفی اشاره کردید.

در سری مقالات اولیه فقط دارم یک دید کلی به کسایی میدم که تازه دارن با این مفاهیم آشنا میشن. این پروژه اولیه دستخوش تغییرات زیادی میشه. در واقع محصول نهایی این مجموعه مقالات بر پایه همین نوع لایه بندی ولی بادید و طراحی مناسب‌تر خواهد بود.

در مورد ORM هم من با چند Application سروکار داشتم که در روال توسعه بخش‌های جدید رو بنا به دلایلی با ORM یا DB متفاوتی توسعه داده اند. غیر از این موضوع، حتی بخشهایی از مدل، سرویس و یا مخزن رو در پروژه‌های دیگری استفاده کرده اند. همچنین برخی از نکات مربوط به تفکیک لایه‌ها به منظور تست پذیری راحت‌تر رو هم در نظر بگیرید.

در مورد اشیاء Request و Response هم باید خدمتان عرض کنم که برای درخواست و پاسخ به درخواست استفاده می‌شوند که چون پروژه ای که مثال زدم کوچک بوده ممکنه کاملاً درکش نکرده باشید. ما کلاسهای Request و Response متعددی در پروژه داریم که ممکنه خیلی از اونها فقط از یک View Model استفاده کنن ولی پارامترهای ارسالی یا دریافتی آنها متفاوت باشد.

در مورد try...catch هم من با شما کاملاً موافقم. به دلیل هزینه ای که دارد باید در آخرین سطح قرار بگیرد. در این مورد ما میتونیم اونو به Presentation و یا در MVC به Controller منتقل کنیم.

در مورد DbContext هم هنوز الگویی رو معرفی نکردم. در واقع هنوز وارد جزئیات لایه‌ی Data نشدم. در مورد اون اگه اجازه بدی بعداً صحبت میکنم.

نویسنده: ایلیا
تاریخ: ۰۰:۴۳ ۱۳۹۲/۰۱/۰۳

آقای خوشبخت خداقوت.

مرسی از مطالب خوبتون.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۱/۰۳ ۰:۴۸

لطفا برای اینکه نظرات حالت فنی تر و غنای بیشتری پیدا کنند، از ارسال پیام های تشکر خودداری کنید. برای ابراز احساسات و همچنین تشکر، لطفا از گزینه رای دادن به هر مطلب که ذیل آن قرار دارد استفاده کنید. این مطلب تا این لحظه 76 بار دیده شده، اما فقط 4 رای دارد. لطفا برای ابراز تشکر، امتیاز بدهید. ممنون.

نویسنده: محسن
تاریخ: ۱۳۹۲/۰۱/۰۳ ۱:۰۰

- من در عمل تفاوتی بین لایه مخزن و سرویس شما مشاهده نمی کنم. یعنی لایه مخزن داره GetAll می کنه، بعد لایه سرویس هم داره همون رو به یک شکل دیگری بر می گردونه. این تکرار کد نیست؟ این دو یکی نیستند؟

عموما در منابع لایه مخزن رو به صورت روکشی برای دستورات مثلا EF یا LINQ to SQL معرفی می کنند. فرضشون هم این است که این روش ما رو از تماس مستقیم با ORM برحذر می داره (شاید فکر می کنند ایدز می گیرند اگر مستقیم کار کنند!). ولی عرض کردم این روکش در واقعیت فقط شاید با EF یا L2S قابل تعویض باشه نه با ORM های دیگر با روش های مختلف و بیشتر یک تصور واهی هست که جنبه عملی نداره. بیشتر تئوری هست بدون پایه تجربه دنیای واقعی. ضمن اینکه این روکش باعث میشه نتونید از خیلی از امکانات ORM مورد استفاده درست استفاده کنید. مثلا ترکیب کوئری ها یا روش های به تاخیر افتاده و امثال این.

- پس در عمل شما Request ViewModel و Response ViewModel تعریف کردید.

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۲/۰۱/۰۳ ۱۲:۲۷

سپاس از سری مطالبی که منتشر می کنید.

- پیشنهادی که من دارم اینه که لایه Repository حذف شود، همانطور که در مطالب قبلی ذکر شده DbSet در Entity Framework همان پیاده سازی الگوی مخزن هست و ایجاد Repository جدید روی آن یک Abstraction اضافه هست. در نتیجه اگر Repository حذف شود همه ی منطق ها مانند GetBlaBla به Service منتقل می شود.

- یک پیشنهاد دیگر اینکه استفاده از کلمه New در Presentation Layer را به حداقل رساند و همه جا نیاز مندی ها را به صورت وابستگی به کلاس های استفاده کننده تزریق شود تا در زمان نوشتن تست ها همه ی اجزاء قابل تعویض با Mock objects باشند.

نویسنده: افشین
تاریخ: ۱۳۹۲/۰۱/۰۶ ۱۱:۱۵

لطفا دمو یا سورس برنامه رو هم قرار بدید که یادگیری و آموزش سریعتر انجام بشه.

ممنون

نویسنده: صابر فتح الهی
تاریخ: ۱۳۹۲/۰۱/۱۰ ۰:۱۱

با سلام از کار بزرگی که دارین می کنین سپاس
یک سوال؟
جای الگوی Unit Of Work در این پروژه کجا میشه؟

در این [پست](#) جناب آقای مهندس نصیری در لایه سرویس الگوی واحد کار را پیاده کرده اند، با توجه به وجود الگوی Repository

در پروژه شما ممنون میشم شرح بیشتری بدین که جایگاه پیاده سازی الگو واحد کار با توجه به مزایایی که دارد در کدام لایه است؟

نویسنده: رام
تاریخ: ۵:۲۹ ۱۳۹۲/۰۱/۱۶

محسن جان، چیزی که من از این الگو در مورد واکنشی و نمایش داده‌ها برداشت میکنم اینه:

کلاس‌های لایه مخزن با دریافت دستور از لایه سرویس آبجکت مدل مربوطه را پر میکنند و به بالا (لایه سرویس) پاس میدهند.

بعد

در لایه سرویس نمونه‌ی مدل مربوطه به ویومدل متناظر باهاش تبدیل میشه و به لایه بالاتر فرستاده میشه

بنابراین

کار در "لایه مخزن" روی "مدل‌ها" انجام میگیره

و

کار در "لایه سرویس" روی "ویومدل‌ها" انجام میشه

نتیجه: لایه سرویس هدف دیگری را نسبت به لایه مخزن دنبال میکند و این هدف آنقدر بزرگ و مهم هست که برایش یه لایه مجزا در نظر گرفته بشه

نویسنده: رام
تاریخ: ۵:۴۹ ۱۳۹۲/۰۱/۱۶

شاهین جان، من با حذف لایه مخزن مخالف هستم. زیرا:

ما لایه ای به نام "لایه مخزن" را میسازیم تا در نهایت کلیه متدهایی که برای حرف زدن با داده هامون را نیاز داریم داشته باشیم. حالا این اطلاعات ممکنه از پایگاه داده یا جاهای دیگه جمع آوری بشوند (و الزاما توسط EF قابل دسترسی و ارائه نباشند)

همچنین گاهی نیاز هست که بر مبنای چند متد که EF به ما میرسونه (مثلا چند SP) یک متد کلی‌تر را تعریف کنیم (چند فراخوانی را در یک متد مثلا متد X در لایه مخزن انجام دهیم) و در لایه بالاتر آن متد را صدا بزنیم (بجای نوشتن و تکرار پاپی همه کدهای نوشت شده در متد X)

علاوه بر این در لایه مخزن میشه چند ORM را هم کنار هم دید (نه فقط EF) که همونطور که آقای خوشبخت در کامنت‌ها نوشتند گاهی نیاز میشه.

بنابراین:

من وجود لایه مخزن را ضروری میدونم.

(فراموش نکنیم که هدف از این آموزش تعریف یک الگوی معماری مناسب برای پروژه‌های بزرگ هست و الا بدون خیلی از اینها هم میشه برنامه ساخت. همونطور که اکثرا بدون این ساختارها و خیلی ساده‌تر میسازند)

نویسنده: محسن
تاریخ: ۹:۳ ۱۳۹۲/۰۱/۱۶

- بحث آقای شاهین و من در مورد مثال عینی بود که زده شد. در مورد کار با ORM که کدهاش دقیقا ارائه شده. این روش قابل نقد و رد است.

شما الان اومدی یک بحث انتزاعی کلی رو شروع کردید. بله. اگر ORM رو کنار بگذارید مثلاً می‌رسید به ADO.NET (یک نمونه که خیلی‌ها در این سایت حداقل یکبار باهاش کار کردن). این افراد پیش از اینکه این مباحث مطرح باشن برای خودشون لایه DAL داشتند و تمام جزئیات ADO.NET رو کپسوله کرده بودن در اون. حالا با اومدن ORM‌ها این لایه DAL کنار رفته چون خود ORM هست که کپسوله کننده ADO.NET است. همین‌ها هم یک لایه دیگر داشتند به نام BLL که از لایه DAL استفاده می‌کرد برای پیاده سازی منطق تجاری برنامه. این لایه الان اسمش شده لایه سرویس.

یعنی تمام مواردی رو که عنوان کردید در مورد ADO.NET صدق می‌کنه. یکی اسمش رو می‌ذاره شما اسمش رو گذاشتید Repository. ولی این مباحث ربطی به یک ORM تمام عیار که کپسوله کننده ADO.NET است ندارد.

- ترکیب چند SP در لایه مخزن انجام نمیشه. چیزی رو که عنوان کردید یعنی پیاده سازی منطق تجاری و این مورد باید در لایه سرویس باشه. اگر از ADO.NET استفاده میشه، می‌تونیم با استفاده از DAL جزئیات دسترسی به SP رو مخفی و ساده‌تر کنیم با کدی یک دست‌تر در تمام برنامه. اگر از EF استفاده می‌کنیم، باز همین ساده سازی در طی فراخوانی فقط یک متد انجام شده. بنابراین بهتر است وضعیت و سطح لایه‌ای رو که داریم باهاش کار می‌کنیم خوب بررسی و درک کنیم.

- می‌تونید در عمل در بین پروژه‌های سورس باز و معتبر موجود فقط یک نمونه رو به من ارائه بدید که در اون از 2 مورد ORM مختلف همزمان استفاده شده باشه؟ این مورد یعنی سؤ مدیریت. یعنی پراکندگی و انجام کاری بسیار مشکل مثلاً یک نمونه: ORM لایه‌ای دارند به نام سطح اول کش که مثلاً در EF اسمش هست Trackig API. این لایه فقط در حین کار با Context همون ORM کار می‌کنه. اگر دو مورد رو با هم مخلوط کنید، قابل استفاده نیست، ترکیب پذیر نیستند. از این دست باز هم هست مثلاً در مورد نحوه تولید پروکسی‌هایی که برای lazy loading تولید می‌کنند و خیلی از مسایل دیگری از این دست. ضمن اینکه مدیریت چند Context فقط در یک لایه خودش یعنی نقض اصل تک مسئولیتی کلاس‌ها.

نویسنده: محسن
تاریخ: ۱۳۹۲/۰۱/۱۶ ۹:۱۵

سعی نکنید انتزاعی بحث کنید. چون در این حالت این حرف می‌تونه درست باشه یا حتی نباشه. اگر از ADO.NET استفاده می‌کنید، درسته. اگر از EF استفاده می‌کنید غلط هست. لازم هست منطق کار با ADO.NET رو یک سطح کپسوله کنیم. چون از تکرار کد جلوگیری می‌کنه و نهایتاً به یک کد یک دست خواهیم رسید. لازم نیست اعمال یک ORM رو در لایه‌ای به نام مخزن کپسوله کنیم، چون خودش کپسوله سازی ADO.NET رو به بهترین نحوی انجام داده. برای نمونه در همین مثال عینی بالا به هیچ مزیتی نرسیدیم. فقط یک تکرار کد است. فقط بازی با کدها است.

نویسنده: رام
تاریخ: ۱۳۹۲/۰۱/۱۶ ۱۶:۴۶

من منظور شما را خوب متوجه میشم ولی حرفام به بحث انتزاعی نیست چون پروژه عملی زیر دستم دارم که توی اون هم با پر کردن View Model کار میکنم.

مشکل از اینجا شروع میشه که شما فکر میکنید همیشه مدل ای که در EF ساختید را باید بدون تغییر در ساختارش به پوسته برنامه برسونید و از پوسته هم دقیقاً نمونه ای از همون را بگیرید و به لایه‌های پایین بفرستید ولی یکی از مهمترین کارهای View Model اینه که این قانون را از این سفتی و سختی در بیاره چون خیلی مواقع هست که شما در پوسته برنامه به شکل دیگه ای از داده‌ها (متفاوت با اونچه در Model تعریف کردید و EF باهاش کار میکنه) نیاز دارید. مثلاً فیلد تاریخ از نوع DateTime در Model و نوع String در پوسته و یا حتی اضافه و کم کردن فیلدهای یک Model و ایجاد ساختارهای متفاوتی از اون برای عملیات‌های Select, Update و Delete. لذا لایه سرویس قرار نیست فقط همون کار لایه مخزن را تکرار کنه (به قول شما GetAll). بلکه در زمان لزوم تغییرات لازم که نام بردم را هم رووش اعمال میکنه (که به نظر من آقای خوشبخت هم به خوبی از کلمه Convert در لایه سرویس استفاده کردند).

اما بحث اینکه ما در لایه مخزن روی EF یک سطح کپسوله میسازیم جای گفتگو داره هرچند من در اون مورد هم با وجد لایه مخزن بیشتر موافقم تا گفتگوی مستقیم لایه سرویس با چیزی مثل EF

نتیجه: فرقی نمیکنه شما از Asp.Net استفاده میکنید یا هر ORM مورد نظرتون. کلاس‌های مدل باید در ارتباط با لایه بالاتر خودشون به ویو مدل تبدیل بشند و در این الگو این کار در لایه سرویس انجام میشه.

نویسنده:

محسن

تاریخ:

۱۷:۱۰ ۱۳۹۲/۰۱/۱۶

- پیاده سازی الگوی مخزن در عمل (بر اساس بحث فعلی که در مورد کار با ORM ها است) به صورت کپسوله سازی ORM در همه جا مطرح میشه و اینکار اساسا اشتباه هست. چون هم شما رو محروم می‌کنه از قابلیت‌های پیشرفته ORM و هم ارزش افزوده‌ای رو به همراه نداره. دست آخر می‌بینید در لایه مخزن GetAll دارید در لایه سرویس هم GetAll دارید. این مساله هیچ مزیتی نداره. یک زمانی در ADO.NET برای GetAll کردن باید کلی کد شبیه به کدهای یک ORM نوشته می‌شد. خود ORM الان اومده این‌ها رو کپسوله کرده و لایه‌ای هست روی اون. اینکه ما مجدداً یک پوستره روی این بکشیم حاصلی نداره بجز تکرار کد. عده‌ای عنوان می‌کنند که حاصل اینکار امکان تعویض ORM رو ممکن می‌کنه ولی این‌ها هم بعد از یک مدت تجربه با ORM‌های مختلف به این نتیجه می‌رسند که ای بابا! حتی پیاده سازی LINQ این ORM‌ها یکی نیست چه برسه به قابلیت‌های پیشرفته‌ای که در یکی هست در دوتای دیگر نیست (واقع بینی، بجای بحث تئوری محض).

- اینکه این تبدیلات (پر کردن ViewModel از روی مدل) هم می‌تونه و بهتره که (نه الزاما) در لایه سرویس انجام بشه، نتیجه مناسبی هست.

نویسنده:

مجتبی آزاد

تاریخ:

۱۴:۳ ۱۳۹۴/۰۴/۱۰

بعد از پیاده سازی UOW و لایه‌بندی نرم‌افزار به این شکل که در مطلب فعلی توضیح داده شد، فرض کنید دو ویومدل زیر را داریم:

```
public class PersonFormViewModel
{
    public long Id { get; set; }
    public long RequestId { get; set; }
    [DisplayName("نام کاربری"), Required(ErrorMessage = "نام کاربری الزامی می‌باشد")]
    public string Username { get; set; }
    public bool Accepted { get; set; }
    [DisplayName("مدل")]
    public string DeviceModel { get; set; }
    public DateTime? ExpireDate { get; set; }
    public RequestViewModel RequestViewModel { get; set; }
}
```

```
public class RequestViewModel
{
    public long Id { get; set; }
    public string Username { get; set; }
    [DisplayName("توضیحات")]
    [DataType(DataType.MultilineText)]
    public string Description { get; set; }
    public DateTime CreateDate { get; set; }
    public Nullable<long> DeviceId { get; set; }
    public Nullable<long> ParentId { get; set; }
    public long RequestTypeId { get; set; }
    public bool IsFinalized { get; set; }
    public virtual PersonFormViewModel PersonFormViewModel { get; set; }
}
```

سناریو به این شکل است که ما فرمی داریم برای ایجاد یک درخواست (RequestViewModel) که با ایجاد آن در واقع اطلاعات شخص (PersonFormViewModel) را نیز دریافت می‌کنیم.

برای افزودن RequestViewModel به دیتابیس این دو روش قابل پیاده‌سازی است:
روش اول:

تنها RequestViewModel را از طریق RequestService اضافه می‌کنیم و به دلیل وجود PersonFormViewModel داخل RequestViewModel اطلاعات شخص به خودی خود داخل entity مربوطه اضافه می‌شود:

```
_requestService.Add(requestViewModel);
```



```
_uow.SaveChanges();
```

روش دوم:

ابتدا RequestViewModel را از طریق سرویس مربوطه اضافه می‌کنیم و بعد به طور جداگانه PersonViewModel را از طریق سرویس PersonFormService اضافه می‌کنیم:

```
var addedRequest = _requestService.Add(requestViewModel );  
var personViewModel = requestViewModel .PersonFormViewModel;  
_personFormService.Add(personViewModel);  
_uow.SaveChanges();
```

در حال حاضر روش درست کدام است؟

نویسنده: میثم خوشبخت

تاریخ: ۱۳/۰۴/۱۳۹۴ ۱۹:۲۴

اگر مشخصات شخص به همراه درخواست ثبت می‌شود و مدخل ورودی به سیستم درخواست می‌باشد، همان روش اول اتفاق می‌افتد.

یعنی با ثبت درخواست، شخص نیز به صورت خودکار ثبت خواهد شد.

به این نکته توجه داشته باشید که روش دوم به این دلایل غلط می‌باشند:

- 1- طبق تعریفی که در معماری سرویس گرا شده است، هیچ متدی از سرویس، نباید به متد سرویس دیگری وابسته باشد
- 2- چون ثبت این دو آیتم باید با هم انجام شود، پس بهتر است در یک متد دو موجودیت ثبت شوند
- 3- و همینطور چون باید در یک تراکنش قرار بگیرند و تجربه نشان داده در هر عملیات بهتر است DataContext نمونه سازی گردد پس چندین کار که قرار است در یک مرحله انجام شوند بهتر است در یک متد سرویس انجام گیرند

UI

در نهایت نوبت به طراحی و کدنویسی UI می‌رسد تا بتوانیم محصولات را به کاربر نمایش دهیم. اما قبل از شروع باید موضوعی را یادآوری کنم. اگر به یاد داشته باشید، در کلاس ProductService موجود در لایه Domain، از طریق یکی از روشهای الگوی Dependency Injection به نام Constructor Injection، فیلدی از نوع IProductRepository را مقداردهی نمودیم. حال زمانی که بخواهیم نمونه ای را از ProductService ایجاد نماییم، باید به عنوان پارامتر ورودی سازنده، شی ایجاد شده از جنس کلاس ProductRepository موجود در لایه Repository را به آن ارسال نماییم. اما از آنجایی که می‌خواهیم تفکیک پذیری لایه‌ها از بین نرود و UI بسته به نیاز خود، نمونه مورد نیاز را ایجاد نموده و به این کلاس ارسال کند، از ابزارهایی برای این منظور استفاده می‌کنیم. یکی از این ابزارها StructureMap می‌باشد که یک Inversion of Control Container یا به اختصار IoC Container نامیده می‌شود. با Inversion of Control در مباحث بعدی بیشتر آشنا خواهید شد. StructureMap ابزاری است که در زمان اجرا، پارامترهای ورودی سازنده کلاسهایی را که از الگوی Dependency Injection استفاده نموده اند و قطعاً پارامتر ورودی آنها از جنس یک Interface می‌باشد را، با ایجاد شی مناسب مقداردهی می‌نماید.

به منظور استفاده از StructureMap در Visual Studio 2012 باید بر روی پروژه UI خود کلیک راست نموده و گزینه‌ی Manage NuGet Packages را انتخاب نمایید. در پنجره ظاهر شده و از سمت چپ گزینه‌ی Online و سپس در کادر جستجوی سمت راست و بالای پنجره واژه‌ی structuremap را جستجو کنید. توجه داشته باشید که باید به اینترنت متصل باشید تا بتوانید Package مورد نظر را نصب نمایید. پس از پایان عمل جستجو، در کادر میانی structuremap ظاهر می‌شود که می‌توانید با انتخاب آن و فشردن کلید Install آن را بر روی پروژه نصب نمایید.

جهت آشنایی بیشتر با NuGet و نصب آن در سایر نسخه‌های Visual Studio می‌توانید به لینک‌های زیر رجوع کنید:

1. [آشنایی](#)

با [NuGet قسمت اول](#)

2. [آشنایی](#)

با [NuGet قسمت دوم](#)

3. [Installing](#)

[NuGet](#)

کلاسی با نام BootStrapper را با کد زیر به پروژه UI خود اضافه کنید:

```
using StructureMap;
using StructureMap.Configuration.DSL;
using SoCPatterns.Layered.Repository;
using SoCPatterns.Layered.Model;

namespace SoCPatterns.Layered.WebUI
{
    public class BootStrapper
    {
        public static void ConfigureStructureMap()
        {
            ObjectFactory.Initialize(x => x.AddRegistry<ProductRegistry>());
        }
    }
}
```

```
public class ProductRegistry : Registry
{
    public ProductRegistry()
    {
        For<IProductRepository>().Use<ProductRepository>();
    }
}
```

ممکن است یک WinUI ایجاد کرده باشید که در این صورت به جای فضای نام SoCPatterns.Layered.WebUI از SoCPatterns.Layered.WinUI استفاده نمایید.

هدف کلاس BootStrapper این است که تمامی وابستگی‌ها را توسط StructureMap در سیستم Register نماید. زمانی که کدهای کلاینت می‌خواهند به یک کلاس از طریق StructureMap دسترسی داشته باشند، StructureMap وابستگی‌های آن کلاس را تشخیص داده و بصورت خودکار پیاده سازی واقعی (Concrete Implementation) آن کلاس را، براساس همان چیزی که ما برایش تعیین کردیم، به کلاس تزریق می‌نماید. متد ConfigureStructureMap باید در همان لحظه ای که Application آغاز به کار می‌کند فراخوانی و اجرا شود. با توجه به نوع UI خود یکی از روالهای زیر را انجام دهید:

در WebUI :

فایل Global.asax را به پروژه اضافه کنید و کد آن را بصورت زیر تغییر دهید:

```
namespace SoCPatterns.Layered.WebUI
{
    public class Global : System.Web.HttpApplication
    {
        protected void Application_Start(object sender, EventArgs e)
        {
            BootStrapper.ConfigureStructureMap();
        }
    }
}
```

در WinUI :

در فایل Program.cs کد زیر را اضافه کنید:

```
namespace SoCPatterns.Layered.WinUI
{
    static class Program
    {
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
        }
    }
}
```

;(BootStrapper.ConfigureStructureMap

```
Application.Run(new Form1());
    }
}
```

سپس برای طراحی رابط کاربری، با توجه به نوع UI خود یکی از روالهای زیر را انجام دهید:

صفحه Default.aspx را باز نموده و کد زیر را به آن اضافه کنید:

```
<asp:DropDownList AutoPostBack="true" ID="ddlCustomerType" runat="server">
  <asp:ListItem Value="0">Standard</asp:ListItem>
  <asp:ListItem Value="1">Trade</asp:ListItem>
</asp:DropDownList>
<asp:Label ID="lblErrorMessage" runat="server" ></asp:Label>
<asp:Repeater ID="rptProducts" runat="server" >
  <HeaderTemplate>
    <table>
      <tr>
        <td>Name</td>
        <td>RRP</td>
        <td>Selling Price</td>
        <td>Discount</td>
        <td>Savings</td>
      </tr>
      <tr>
        <td colspan="5"><hr /></td>
      </tr>
    </HeaderTemplate>
    <ItemTemplate>
      <tr>
        <td><%= Eval("Name") %></td>
        <td><%= Eval("RRP") %></td>
        <td><%= Eval("SellingPrice") %></td>
        <td><%= Eval("Discount") %></td>
        <td><%= Eval("Savings") %></td>
      </tr>
    </ItemTemplate>
    <FooterTemplate>
      </table>
    </FooterTemplate>
  </asp:Repeater>
```

فایل Form1.Designer.cs را باز نموده و کد آن را بصورت زیر تغییر دهید:

```
#region Windows Form Designer generated code
/// <summary>
/// Required method for Designer support - do not modify
/// the contents of this method with the code editor.
/// </summary>
private void InitializeComponent()
{
    this.cmbCustomerType = new System.Windows.Forms.ComboBox();
    this.dgvProducts = new System.Windows.Forms.DataGridView();
    this.colName = new System.Windows.Forms.DataGridViewTextBoxColumn();
    this.colRrp = new System.Windows.Forms.DataGridViewTextBoxColumn();
    this.colSellingPrice = new System.Windows.Forms.DataGridViewTextBoxColumn();
    this.colDiscount = new System.Windows.Forms.DataGridViewTextBoxColumn();
    this.colSavings = new System.Windows.Forms.DataGridViewTextBoxColumn();
    ((System.ComponentModel.ISupportInitialize)(this.dgvProducts)).BeginInit();
    this.SuspendLayout();
    //
    // cmbCustomerType
    //
    this.cmbCustomerType.DropDownStyle =
System.Windows.Forms.ComboBoxStyle.DropDownList;
    this.cmbCustomerType.FormattingEnabled = true;
    this.cmbCustomerType.Items.AddRange(new object[] {
        "Standard",
        "Trade"});
    this.cmbCustomerType.Location = new System.Drawing.Point(12, 90);
```

```

        this.cmbCustomerType.Name = "cmbCustomerType";
        this.cmbCustomerType.Size = new System.Drawing.Size(121, 21);
        this.cmbCustomerType.TabIndex = 3;
        //
        // dgvProducts
        //
        this.dgvProducts.ColumnHeadersHeightSizeMode =
System.Windows.Forms.DataGridViewColumnHeadersHeightSizeMode.AutoSize;
        this.dgvProducts.Columns.AddRange(new System.Windows.Forms.DataGridViewColumn[] {
            this.colName,
            this.colRrp,
            this.colSellingPrice,
            this.colDiscount,
            this.colSavings});
        this.dgvProducts.Location = new System.Drawing.Point(12, 117);
        this.dgvProducts.Name = "dgvProducts";
        this.dgvProducts.Size = new System.Drawing.Size(561, 206);
        this.dgvProducts.TabIndex = 2;
        //
        // colName
        //
        this.colName.DataPropertyName = "Name";
        this.colName.HeaderText = "Product Name";
        this.colName.Name = "colName";
        this.colName.ReadOnly = true;
        //
        // colRrp
        //
        this.colRrp.DataPropertyName = "Rrp";
        this.colRrp.HeaderText = "RRP";
        this.colRrp.Name = "colRrp";
        this.colRrp.ReadOnly = true;
        //
        // colSellingPrice
        //
        this.colSellingPrice.DataPropertyName = "SellingPrice";
        this.colSellingPrice.HeaderText = "Selling Price";
        this.colSellingPrice.Name = "colSellingPrice";
        this.colSellingPrice.ReadOnly = true;
        //
        // colDiscount
        //
        this.colDiscount.DataPropertyName = "Discount";
        this.colDiscount.HeaderText = "Discount";
        this.colDiscount.Name = "colDiscount";
        //
        // colSavings
        //
        this.colSavings.DataPropertyName = "Savings";
        this.colSavings.HeaderText = "Savings";
        this.colSavings.Name = "colSavings";
        this.colSavings.ReadOnly = true;
        //
        // Form1
        //
        this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F);
        this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
        this.ClientSize = new System.Drawing.Size(589, 338);
        this.Controls.Add(this.cmbCustomerType);
        this.Controls.Add(this.dgvProducts);
        this.Name = "Form1";
        this.Text = "Form1";
        ((System.ComponentModel.ISupportInitialize)(this.dgvProducts)).EndInit();
        this.ResumeLayout(false);
    }
}
#endregion
private System.Windows.Forms.ComboBox cmbCustomerType;
private System.Windows.Forms.DataGridView dgvProducts;
private System.Windows.Forms.DataGridViewTextBoxColumn colName;
private System.Windows.Forms.DataGridViewTextBoxColumn colRrp;
private System.Windows.Forms.DataGridViewTextBoxColumn colSellingPrice;
private System.Windows.Forms.DataGridViewTextBoxColumn colDiscount;
private System.Windows.Forms.DataGridViewTextBoxColumn colSavings;

```

سپس در Code Behind ، با توجه به نوع UI خود یکی از روالهای زیر را انجام دهید:

وارد کد نویسی صفحه Default.aspx شده و کد آن را بصورت زیر تغییر دهید:

```
using System;
using System.Collections.Generic;
using SoCPatterns.Layered.Model;
using SoCPatterns.Layered.Presentation;
using SoCPatterns.Layered.Service;
using StructureMap;

namespace SoCPatterns.Layered.WebUI
{
    public partial class Default : System.Web.UI.Page, IProductListView
    {
        private ProductListPresenter _productListPresenter;
        protected void Page_Init(object sender, EventArgs e)
        {
            _productListPresenter = new
            ProductListPresenter(this, ObjectFactory.GetInstance<Service.ProductService>());
            this.ddlCustomerType.SelectedIndexChanged +=
                delegate { _productListPresenter.Display(); };
        }
        protected void Page_Load(object sender, EventArgs e)
        {
            if(!Page.IsPostBack)
                _productListPresenter.Display();
        }
        public void Display(IList<ProductViewModel> products)
        {
            rptProducts.DataSource = products;
            rptProducts.DataBind();
        }
        public CustomerType CustomerType
        {
            get { return (CustomerType) int.Parse(ddlCustomerType.SelectedValue); }
        }
        public string ErrorMessage
        {
            set
            {
                lblErrorMessage.Text =
                    String.Format("<p><strong>Error:</strong><br/>{0}</p>", value);
            }
        }
    }
}
```

وارد کدنویسی Form1 شوید و کد آن را بصورت زیر تغییر دهید:

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using SoCPatterns.Layered.Model;
using SoCPatterns.Layered.Presentation;
using SoCPatterns.Layered.Service;
using StructureMap;

namespace SoCPatterns.Layered.WinUI
{
    public partial class Form1 : Form, IProductListView
    {
        private ProductListPresenter _productListPresenter;
        public Form1()
        {
            InitializeComponent();
            _productListPresenter =
                new ProductListPresenter(this, ObjectFactory.GetInstance<Service.ProductService>());
            this.cmbCustomerType.SelectedIndexChanged +=
                delegate { _productListPresenter.Display(); };
            dgvProducts.AutoGenerateColumns = false;
        }
    }
}
```

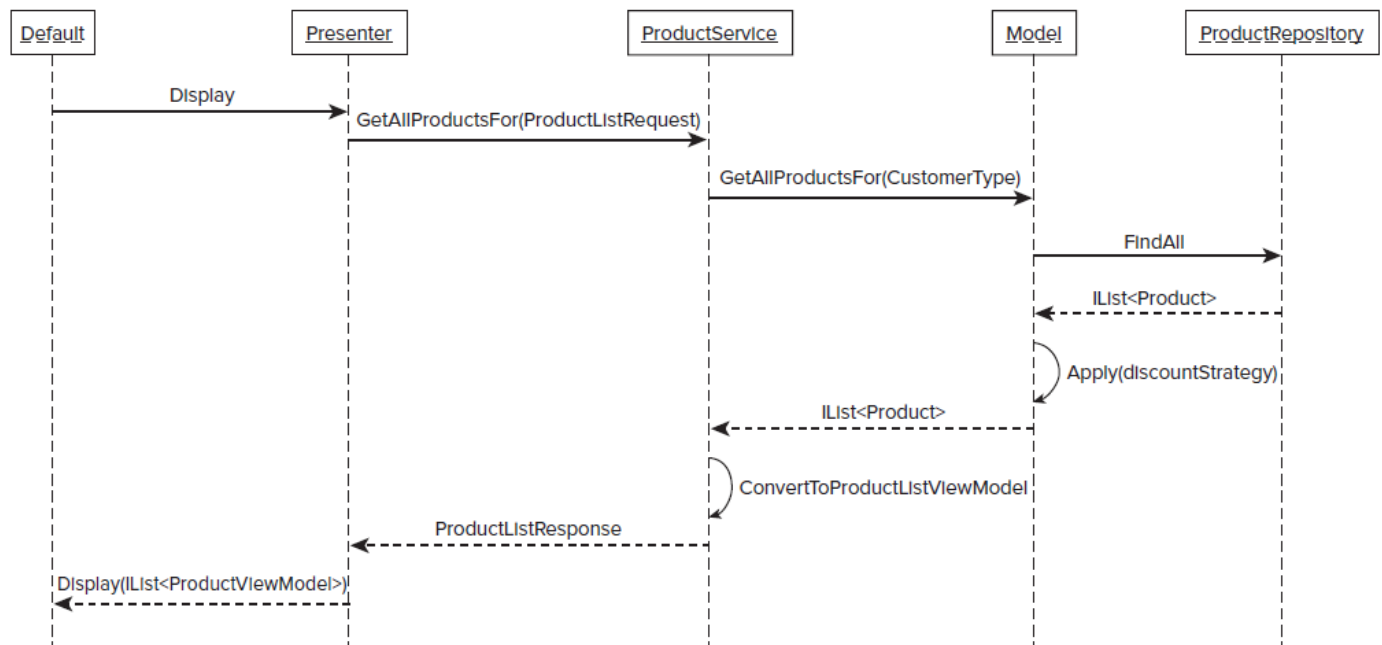
```

        cmbCustomerType.SelectedIndex = 0;
    }
    public void Display(IList<ProductViewModel> products)
    {
        dgvProducts.DataSource = products;
    }
    public CustomerType CustomerType
    {
        get { return (CustomerType)cmbCustomerType.SelectedIndex; }
    }
    public string ErrorMessage
    {
        set
        {
            MessageBox.Show(
                String.Format("Error:{0}{1}", Environment.NewLine, value));
        }
    }
}
}
}

```

با توجه به کد فوق، نمونه ای را از کلاس ProductListPresenter، در لحظه‌ی نمونه سازی اولیه‌ی کلاس UI، ایجاد نمودیم. با استفاده از متد ObjectFactory.GetInstance مربوط به StructureMap، نمونه ای از کلاس ProductService ایجاد شده است و به سازنده‌ی کلاس ProductListPresenter ارسال گردیده است. در مورد Structuremap در مباحث بعدی با جزئیات بیشتری صحبت خواهیم کرد. پیاده سازی معماری لایه بندی در اینجا به پایان رسید.

اما اصلاً نگران نباشید، شما فقط پرواز کوتاه و مختصری را بر فراز کدهای معماری لایه بندی داشته اید که این فقط یک دید کلی را به شما در مورد این معماری داده است. این معماری هنوز جای زیادی برای کار دارد، اما در حال حاضر شما یک Application با پیوند ضعیف (Loosely Coupled) بین لایه‌ها دارید که دارای قابلیت تست پذیری قوی، نگهداری و پشتیبانی آسان و تفکیک پذیری قدرتمند بین اجزای آن می‌باشد. شکل زیر تعامل بین لایه‌ها و وظایف هر یک از آنها را نمایش می‌دهد.



نظرات خوانندگان

نویسنده: حامد

تاریخ: ۱۴:۲۹ ۱۳۹۲/۰۱/۰۳

ممنون از مقاله خوبتون

به نظر شما امکانش هست برای این معماری یک generator بسازیم به طوری که فقط ما تمام جداول دیتابیس و رابطه‌ی آنها را بسازیم و بعد این generator از روی اون تمام لایه‌ها را بر اساس آن بسازه و بعد ما صرفا جاهایی که نیاز به جزییات داره را کامل کنیم

آیا نمونه ای از این برنامه‌ها هست که این معماری یا معماری‌های مشابه را بسازه؟

نویسنده: شاهین کیاست

تاریخ: ۱۵:۳۱ ۱۳۹۲/۰۱/۰۳

اگر با T4 آشنایی داشته باشید بر اساس هر قالبی می‌توانید کد تولید کنید.

نویسنده: حامد

تاریخ: ۱۶:۳۶ ۱۳۹۲/۰۱/۰۳

متأسفانه آشنایی ندارم میشه یه توضیح مختصر بدین و یا منبع معرفی کنید

نویسنده: محسن

تاریخ: ۲۳:۵۹ ۱۳۹۲/۰۱/۰۳

چون اینجا بحث طراحی مطرح شده یک اصل رو در برنامه‌های وب باید رعایت کرد:

هیچ وقت متن خطای حاصل رو به کاربر نمایش ندید (از لحاظ امنیتی). فقط به ذکر عبارت خطایی رخ داده بسنده کنید. خطا رو مثلا توسط ELMAH لاگ کنید برای بررسی بعدی برنامه نویس.

نویسنده: شاهین کیاست

تاریخ: ۱۰:۲۰ ۱۳۹۲/۰۱/۰۴

<http://codepanic.blogspot.com/2012/03/t4-enum.html>

نویسنده: M.Q

تاریخ: ۲۲:۱۵ ۱۳۹۲/۰۱/۰۴

دوست عزیز غیر از ELMAH ابزار دیگری برای لاگ گیری از خطاها وجود دارد که قابل اعتماد باشد؟

همچنین اگر ابزاری جهت لاگ گیری از عملیات کاربران (CRUD => حالا R خیلی مهم نیست) می‌شناسید معرفی نمائید.

با سپاس

نویسنده: محسن

تاریخ: ۰:۳۳ ۱۳۹۲/۰۱/۰۵

متد auditFields مطرح شده در [مطلب ردیابی اطلاعات](#) این سایت برای مقصود شما مناسب است.

نویسنده: صابر فتح الهی
تاریخ: ۱۴:۲۳ ۱۳۹۲/۰۱/۱۲

سلام با تشکر از شما
من نفهمیدم که توی ASP.NET MVC شما چگونه از الگوی MVP استفاده کردین؟
ظاهرا مثال این قسمت هم توی پست وجود نداره، اگر اشتباه می‌کنم لطفا تصحیح بفرمایید.

نویسنده: علی
تاریخ: ۱۶:۳ ۱۳۹۲/۰۱/۱۲

مثال وب فرم هست. page load و post back داره.

نویسنده: شاهین کیاست
تاریخ: ۱۶:۴ ۱۳۹۲/۰۱/۱۲

اگر توجه کنید از الگوی MVP در Web Forms استفاده شده و نه در MVC.

نویسنده: صابر فتح الهی
تاریخ: ۱۸:۳۰ ۱۳۹۲/۰۱/۱۲

آقای کیاست و علی آقا
می‌دونم که پروژه چی هست، یکی از UIهای ما قرار بود MVC باشه خواستم بدونم چطور می‌خوان استفاده کنن، اینجا (در این پست) که می‌دونم ASP.NET Web form هست و در MVC می‌دونم که Page_Load .. وجود نداره سوال من چیز دیگه بود دوستان

نویسنده: شاهین کیاست
تاریخ: ۱۸:۴۴ ۱۳۹۲/۰۱/۱۲

شما گفتید:

سلام با تشکر از شما
من نفهمیدم که توی ASP.NET MVC شما چگونه از الگوی MVP استفاده کردین؟
ظاهرا مثال این قسمت هم توی پست وجود نداره، اگر اشتباه می‌کنم لطفا تصحیح بفرمایید.
با خواندن کامنت شما برداشت کردم شما تصور کردید کدهای پست جاری مربوط به تکنولوژی ASP.NET MVC هست.

به نظر نویسنده هنوز برای MVC و WPF مثال‌ها را ایجاد نکرده و توضیح نداده اند.
اما برای استفاده از این نوع معماری در MVC کار خاصی لازم نیست انجام شود. همانطور که قبلا در مثال‌های آقای نصیری دیده ایم کافی است Service Layer در Controller مدل مناسب را تغذیه کند و برای View فراهم کند.

نویسنده: صابر فتح الهی
تاریخ: ۱۹:۲۶ ۱۳۹۲/۰۱/۱۲

من هم با توجه به مثال آقای نصیری و استفاده از الگوی کار گیج شدم، این معماری یک لایه Repository دارد، من الگوی کار توی این لایه پیاده کردم، با پیاده سازی در این لایه به نظر میاد لایه سرویس کاربردی از دست می‌ده توی پست‌های قبل هم از آقای خوشبخت سوال کردم اما ظاهرا هنوز وقت نکردن پاسخ بدن.

مورد دوم اینکه در این پست الگوی کار شرح داده شده و پیاده سازی شده، و در این پست گفته شده "حین استفاده از EF code first، الگوی واحد کار، همان DbContext است و الگوی مخزن، همان DbSet ها. ضرورتی به ایجاد یک لایه محافظ اضافی بر روی

این‌ها وجود ندارد. " با توجه به این مسائل کلا مسائل قاطی کردم متاسفانه آقای نصیری هم سرشون شلوغ و درگیر [دوره ها](#) است، که بحثی بر سر این معماری بشه.

نویسنده: شاهین کیاست
تاریخ: ۲۰:۴۶ ۱۳۹۲/۰۱/۱۲

روشی که در مثال آقای نصیری گفته شده با روش این سری مقالات کمی متفاوت هست. در آنجا از روکش اضافه برای Repository استفاده نشده همچنین از الگوی واحد کار استفاده شده. به علاوه این سری مقالات ممکن است هنوز تکمیل نشده باشند. به نظر من هر کس با توجه به میزان اطلاعاتی که دارد و درکی که از الگوها دارد با مقایسه‌ی روش‌ها و مقالات می‌تواند تصمیم بگیرد چه معماری به کار بگیرد.

نویسنده: صابر فتح الهی
تاریخ: ۲۱:۳ ۱۳۹۲/۰۱/۱۲

حرف شما کاملا متین هست

من قبلا معماری سه لایه کار می‌کردم، که نمونه اون توی همین سایت [بخش پروژه ها](#) گذاشتم، اما الان با EF , MVC کمی به مشکل بر خوردم و درست نتونستم تا حالا لایه‌های مورد نظر برای خودم در پروژه‌ها تفکیک کنم، این معماری به نظرم جالب اومد، خواستم که الگوی کار هم توی اون به کار ببرم که به مشکل بر خوردم (چون درک درستی از الگوی کار پیدا نکردم یا شایدم کلا دارم اشتباه می‌کنم). البته به قول شما شاید این معماری هنوز تکمیل نشده پروژه اش، در هر صورت از پاسخ‌های شما متشکرم.

نویسنده: شاهین کیاست
تاریخ: ۲۱:۷ ۱۳۹۲/۰۱/۱۲

خواهش می‌کنم.
فقط جهت یادآوری [مثال](#) روش آقای نصیری با پوشش MVC و EF قابل دریافت است.

نویسنده: ابوالفضل روشن ضمیر
تاریخ: ۱:۴۵ ۱۳۹۲/۰۱/۱۷

سلام
با تشکر فروان از شما ...
اگر امکان داره این مثال که در قالی یک پروژه نوشته شده برای دانلود قرار دهید ... تا بهتر بتوانیم برنامه را تجزیه و تحلیل کنیم
... ممنون

نویسنده: فرشید علی اکبری
تاریخ: ۱۵:۵۳ ۱۳۹۲/۰۱/۱۹

با سلام و تشکر از زحمات کلیه دوستان
با زحمتی که آقای خوشبخت تا اینجا کشیدن فکر کنم در صورتیکه خودمون مقاله مربوطه به این پروژه رو قدم به قدم بخونیم و طراحی کنیم خیلی بهتر متوجه میشیم تا اینکه اونو آماده دانلود کنیم. من با این روش پیش رفتم و برای ایجاد اون با step by step کردن مراحلش حدود 45 دقیقه وقت گذاشتم ولی درصد یادگیری خیلی بالاتر بود تا گرفتن فایل آماده...
درضمن لازمه بگم که بخاطر رفع شک و شبهه در سرعت پردازش وبلا اومدن اطلاعات، من تست این روش رو با تعداد 155 هزار رکورد انجام دادم که کمتر از سه ثانیه برام لود شد... باوجودیکه کامپوننت‌های دات نت بار مختلفی رو هم روی فرم قرار دادم که بیشتر به اهمیت لود اطلاعاتم در پروژه و فرمهای واقعی پی ببرم.
سؤال اینکه :

به نظر شما ما می‌تونیم روی این لایه‌ها الگوی واحد کار رو هم ایجاد کنیم یا نه؟ اصلا ضرورتی داره ؟

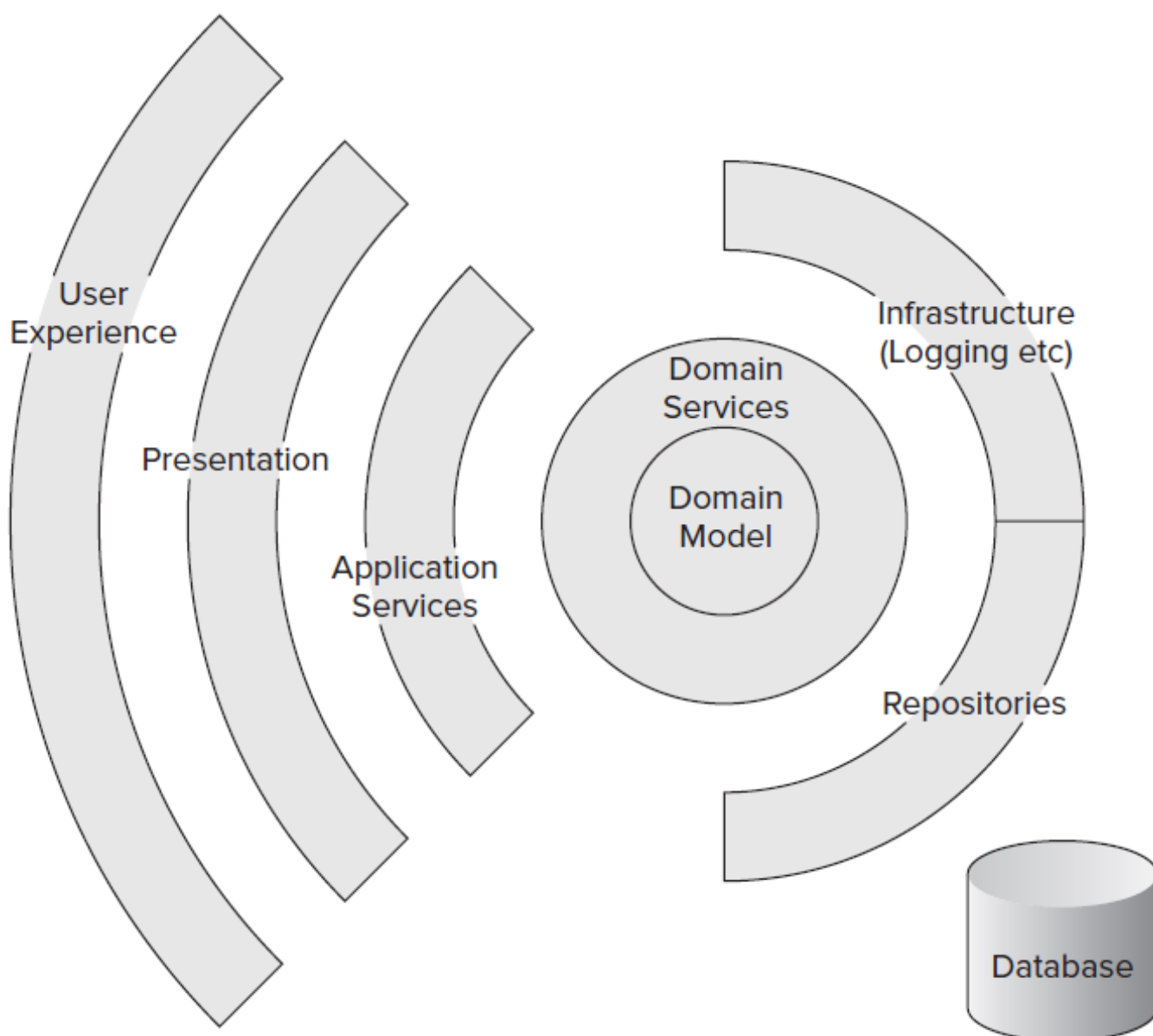
نویسنده: علیرضا کیانی مقدم
تاریخ: ۱۳۹۲/۰۱/۲۸

با تشکر از نویسنده مقاله و اهتمام ایشان به بررسی دقیق مفاهیم ،
از آنجا که flexible و reusable بودن برنامه‌ها را نمی‌توان نادیده گرفت تا آنجا که این تفکیک پذیری خود به مسئله‌ای بگرنج تبدیل نشده و تکرر داده‌ها و پاس دادن غیر ضرور آنها را موجب نشود تلاش در این باره مفید خواهد بود .
امروزه توسعه دهنده‌گان به سمت کم کردن لایه‌های فرسایشی و حذف پیچیدگی‌های غیر ضرور قدم بر می‌دارند. خلق عبارات لامبادا در دات نت و delegate ها نمونه‌هایی از تلاش بشر برنامه نویسی در این باره است .

نویسنده: مسعود 2
تاریخ: ۱۳۹۲/۰۲/۰۹

سلام

business Rule 1ها و validation-2ها در کجای این معماری اعمال میشوند؟



منظور از DomainService چیست؟

ممکنه منابع بیشتری معرفی نمایید؟
ممنون.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۲/۰۹ ۱۶:۲۶

[منبع برای مطالعه بیشتر](#)

نویسنده: محمدرضاتقی پور
تاریخ: ۱۳۹۴/۰۶/۰۷ ۱۵:۵

سلام
مرحله آخر که قراره اطلاعات را به کاربر نشون بدیم
اگر پروژه UI از نوع MCV باشه
چه تفاوتی خواهد کرد؟
مرسی

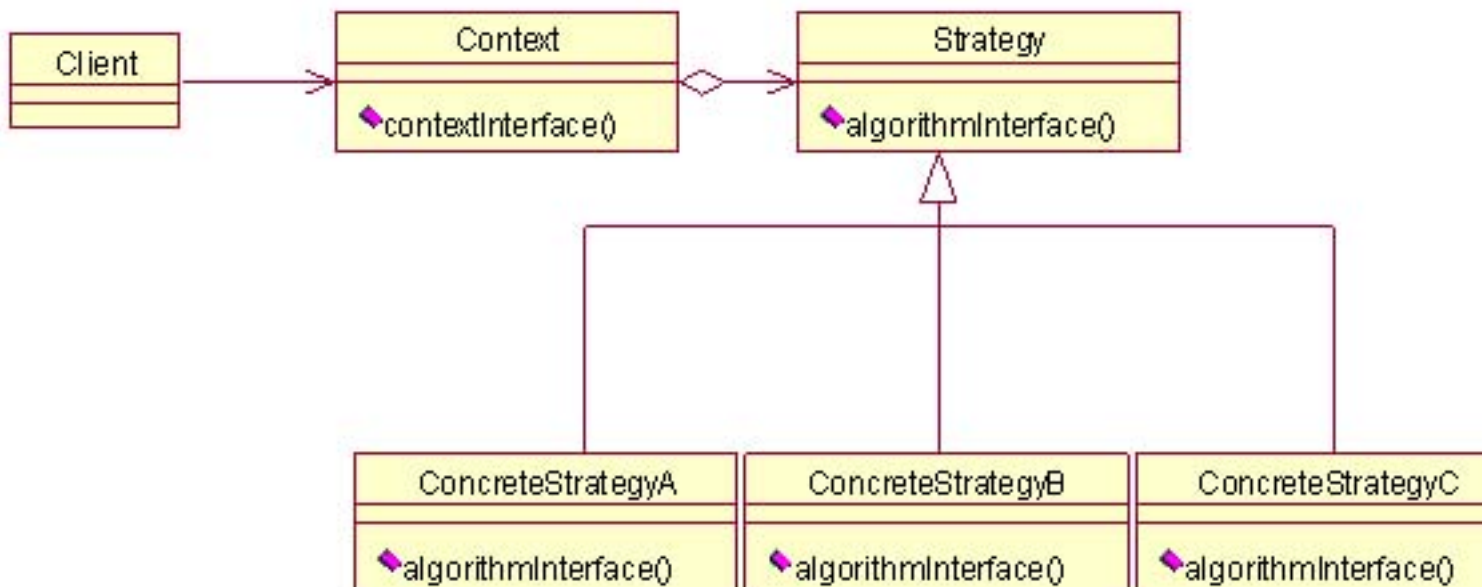
الگوی استراتژی (Strategy) اجازه می‌دهد که یک الگوریتم در یک کلاس بسته بندی شود و در زمان اجرا برای تغییر رفتار یک شیئی تعویض شود.

برای مثال فرض کنید که ما در حال طراحی یک برنامه مسیریابی برای یک شبکه هستیم. همانطوریکه می‌دانیم برای مسیر یابی الگوریتم‌های مختلفی وجود دارد که هر کدام دارای مزایا و معایبی هستند. و با توجه به وضعیت موجود شبکه یا عملی که قرار است انجام پذیرد باید الگوریتمی را که دارای بالاترین کارایی است انتخاب کنیم. همچنین این برنامه باید امکانی را به کاربر بدهد که کارائی الگوریتم‌های مختلف را در یک شبکه فرضی بررسی کنید. حالا طراحی پیشنهادی شما برای این مسئله چیست؟ دوباره فرض کنید که در مثال بالا در بعضی از الگوریتم‌ها نیاز داریم که گره‌های شبکه را بر اساس فاصله‌ی آنها از گره مبدا مرتب کنیم. دوباره برای مرتب سازی الگوریتم‌های مختلف وجود دارد و هر کدام در شرایط خاص، کارائی بهتری نسبت به الگوریتم‌های دیگر دارد. مسئله دقیقاً شبیه مسئله بالا است و این مسئله می‌تواند دارای طراحی شبیه مسئله بالا باشد. پس اگر ما بتوانیم یک طراحی خوب برای این مسئله ارائه دهیم می‌توانیم این طراحی را برای مسائل مشابه به کار ببریم.

هر کدام از ما می‌توانیم نسبت به درک خود از مسئله و سلیقه کاری، طراحی‌های مختلفی برای این مسئله ارائه دهیم. اما یک طراحی که می‌تواند یک جواب خوب و عالی باشد، الگوی استراتژی است که توانسته است بارها و بارها به این مسئله پاسخ بدهد.

الگوی استراتژی گزینه مناسبی برای مسائلی است که می‌توانند از چندین الگوریتم مختلف به مقصود خود برسند.

نمودار UML الگوی استراتژی به صورت زیر است :



اجازه بدهید، شیوه کار این الگو را با مثال مربوط به مرتب سازی بررسی کنیم. فرض کنید که ما تصمیم گرفتیم که از سه الگوریتم زیر برای مرتب سازی استفاده کنیم.

1 - الگوریتم مرتب سازی Shell Sort - الگوریتم مرتب سازی Quick Sort

3 - الگوریتم مرتب سازی Merge Sort

ما برای مرتب سازی در این برنامه دارای سه استراتژی هستیم. که هر کدام را به عنوان یک کلاس جداگانه در نظر می گیریم (همان کلاس های ConcreteStrategy). برای اینکه کلاس Client بتواند به سادگی یک از استراتژی ها را انتخاب کنید بهتر است که تمام کلاس های استراتژی دارای اینترفیس مشترک باشند. برای این کار می توانیم یک کلاس abstract تعریف کنیم و ویژگی های مشترک کلاس های استراتژی را در آن قرار دهیم و کلاس های استراتژی آنها را به ارث ببرند (همان کلاس Strategy) و پیاده سازی کنند.

در زیر کلاس Abstract که کل کلاس های استراتژی از آن ارث می برند را مشاهده می کنید :

```
abstract class SortStrategy
{
    public abstract void Sort(ArrayList list);
}
```

کلاس مربوط به QuickSort

```
class QuickSort : SortStrategy
{
    public override void Sort(ArrayList list)
    {
        // الگوریتم مربوطه
    }
}
```

کلاس مربوط به ShellSort

```
class ShellSort : SortStrategy
{
    public override void Sort(ArrayList list)
    {
        // الگوریتم مربوطه
    }
}
```

کلاس مربوط به MergeSort

```
class MergeSort : SortStrategy
{
    public override void Sort(ArrayList list)
    {
        // الگوریتم مربوطه
    }
}
```

و در آخر کلاس Context که یکی از استراتژی ها را برای مرتب کردن به کار می برد :

```
class SortedList
{
    private ArrayList list = new ArrayList();
    private SortStrategy sortstrategy;

    public void SetSortStrategy(SortStrategy sortstrategy)
    {
        this.sortstrategy = sortstrategy;
    }
    public void Add(string name)
```

```
{
    list.Add(name);
}
public void Sort()
{
    sortstrategy.Sort(list);
}
}
```

نظرات خوانندگان

نویسنده: علی

تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۲:۴۶

با سلام؛ لطفا کلاس آخری را بیشتر توضیح دهید.

نویسنده: محسن خان

تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۲:۵۵

کلاس آخری با یک پیاده سازی عمومی کار می‌کنه. دیگه نمی‌دونه نحوه مرتب سازی چطور پیاده سازی شده. فقط می‌دونه یک متد Sort هست که دراختیارش قرار داده شده. حالا شما راحت می‌تونن الگوریتم مورد استفاده رو عوض کنی، بدون اینکه نیاز داشته باشی کلاس آخری رو تغییر بدی. باز هست برای توسعه. بسته است برای تغییر. به این نوع طراحی رعایت open closed principle هم می‌گن.

نویسنده: SB

تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۴:۲۳

بنظر شما متد Sort کلاس اولیه، نباید از نوع Virtual باشد؟

نویسنده: محسن خان

تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۴:۴۸

نوع کلاسش abstract هست.

نویسنده: مجتبی شاطری

تاریخ: ۱۳۹۲/۰۶/۲۰ ۱۶:۴۷

در صورتی از virtual استفاده می‌کنیم که یک پیاده سازی از متد Sort در SortStrategy داشته باشیم، اما در اینجا طبق فرموده دوستمون کلاس ما فقط انتزاعی (Abstract) هست.

نویسنده: سید ایوب کوکبی

تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۱:۲۲

چرا استراتژی توسط Abstract پیاده سازی شده و از اینترفیس استفاده نشده؟

نویسنده: وحید نصیری

تاریخ: ۱۳۹۲/۰۶/۳۱ ۱۲:۵۱

تفاوت مهمی **نداره**؛ فقط اینترفیس ورژن پذیر نیست. یعنی اگر در این بین متدی رو به تعاریف اینترفیس خودتون اضافه کردید، تمام استفاده کننده‌ها مجبور هستند اون رو پیاده سازی کنند. اما کلاس Abstract می‌تونه شامل یک پیاده سازی پیش فرض متد خاصی هم باشه و به همین جهت ورژن پذیری بهتری داره. بنابراین کلاس Abstract یک اینترفیس است که می‌تواند پیاده سازی هم داشته باشه. همین مساله خاص نگارش پذیری، در طراحی ASP.NET MVC به کار گرفته شده: ([^](#)) برای من نوعی شاید این مساله اهمیتی نداشته باشه. اگر من قرارداد اینترفیس کتابخانه خودم را تغییر دادم، بالاخره شما با یک حداقل نق زدن مجبور به روز رسانی کار خودتان خواهید شد. اما اگر مایکروسافت چنین کاری را انجام دهد، هزاران نفر شروع خواهند کرد به بد گفتن از نحوه مدیریت پروژه تیم‌های مایکروسافت و اینکه چرا پروژه جدید آن‌ها با یک نگارش جدید MVC کامپایل نمی‌شود. بنابراین انتخاب بین این دو بستگی دارد به تعداد کاربر پروژه شما و استراتژی ورژن پذیری قرار دادهای کتابخانه‌ای که ارائه می‌دهید.

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳:۲۷ ۱۳۹۲/۰۶/۳۱

اطلاعات خوبی بود، ممنون، ولی با توجه به تجربه تون، در پروژه‌های متن باز فعلی تحت بستر دات نت بیشتر از کدام مورد استفاده میشه؟ اینترفیس روحیه نظامی خاصی به کلاس‌های مصرف کننده اش میده، یه همین دلیل من زیاد رقبت به استفاده از اون ندارم، آیا مواردی هست که چاره ای نباشه حتما از یکی از این دو نوع استفاده بشه؟

نویسنده: وحید نصیری
تاریخ: ۱۳:۴۹ ۱۳۹۲/۰۶/۳۱

- اگر پروژه خودتون هست، از اینترفیس استفاده کنید. تغییرات آن و نگارش‌های بعدی آن تحت کنترل خودتان است و build دیگران را تحت تاثیر قرار نمی‌دهد.
- در پروژه‌های سورس باز دات نت، عموماً از ترکیب این دو استفاده می‌شود. مواردی که قرار است در اختیار عموم باشند حتی دو لایه هم می‌شوند. مثلاً در MVC یک اینترفیس IController هست و بعد یک کلاس Abstract به نام Controller، که این اینترفیس را پیاده سازی کرده برای ورژن پذیری بعدی و کنترلرهای پروژه‌های عمومی MVC از این کلاس Abstract مشتق می‌شوند یا در پروژه RavenDB از کلاس‌های Abstract زیاد استفاده شده، مانند AbstractIndexCreationTask و AbstractMultiMapIndexCreationTask و غیره.

نویسنده: جمشیدی فر
تاریخ: ۱۶:۱۱ ۱۳۹۲/۰۷/۰۱

توابع abstract بطور ضمنی virtual هستند.

نویسنده: جمشیدی فر
تاریخ: ۱۸:۳۸ ۱۳۹۲/۰۷/۰۱

در کلاس abstract نیز می‌توان از پیاده سازی پیشفرض استفاده کرد. یکی از تفاوت‌های کلاس abstract با Interface همین ویژگی است که سبب ورژن پذیری آن شده است.

نویسنده: جمشیدی فر
تاریخ: ۹:۱۵ ۱۳۹۲/۰۸/۲۱

بهتر نیست در کلاس SortedList برای مشخص کردن استراتژی مرتب سازی، از روش تزریق وابستگی - Dependency Injection - استفاده بشه؟

نویسنده: محسن خان
تاریخ: ۹:۲۵ ۱۳۹۲/۰۸/۲۱

خوب، الان هم وابستگی کلاس یاد شده از طریق سازنده آن در اختیار آن قرار گرفته و داخل خود کلاس وهله سازی نشده. (در این مطلب طراحی بیشتر مدنظر هست تا اینکه حالا این وابستگی به چه صورتی و کجا قرار هست وهله سازی بشه و در اختیار کلاس قرار بگیره؛ این مساله ثانویه است)

نویسنده: جمشیدی فر
تاریخ: ۱۱:۴۳ ۱۳۹۲/۰۸/۲۱

از طریق سازنده کلاس SortedList؟ بنظر نمیداداز طریق سازنده انجام شده باشه. ولی ظاهراً این امکان هست که کلاس بالادستی که می‌خواهد از SortedList استفاده کند، بتواند از طریق تابع SetSortStrategy کلاس مورد نظر رادر اختیار SortedList قرار دهد. به نظر شبیه Setter Injection می‌شود.

الگوی طراحی Factory Method به همراه مثال

عناوین : تعریف Factory Method

• دیاگرام UML

• شرکت کنندگان در UML

• مثالی از Factory Pattern در C#

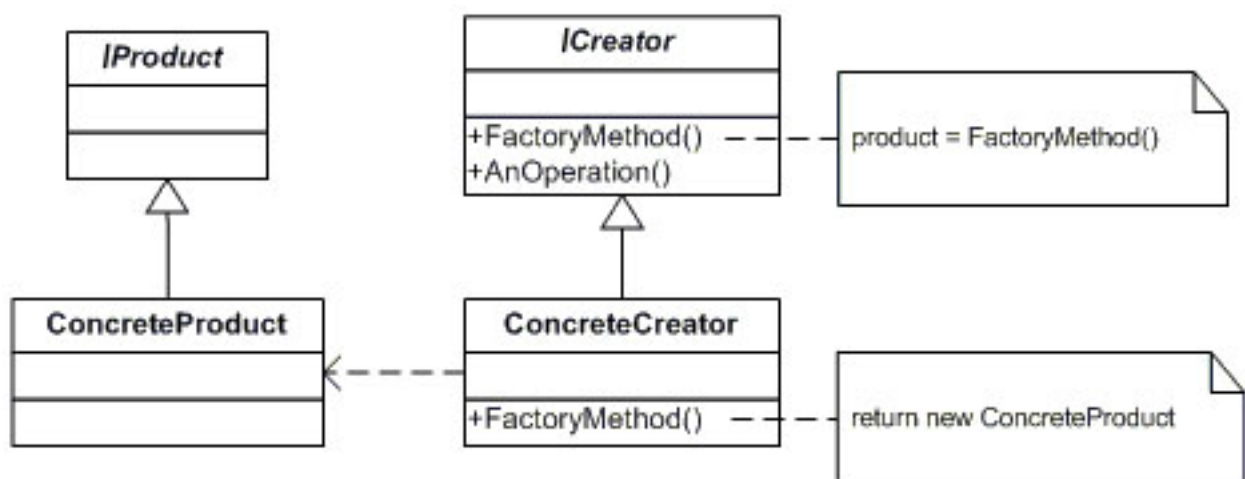
تعریف الگوی Factory Method :

این الگو پیچیدگی ایجاد اشیاء برای استفاده کننده را پنهان می‌کند. ما با این الگو می‌توانیم بدون اینکه کلاس دقیق یک شیء را مشخص کنیم آن را ایجاد و از آن استفاده کنیم. کلاینت (استفاده کننده) معمولاً شیء واقعی را ایجاد نمی‌کند بلکه با یک واسطه و یا کلاس انتزاعی (Abstract) در ارتباط است و کل مسئولیت ایجاد کلاس واقعی را به Factory Method می‌سپارد. کلاس Factory Method می‌تواند استاتیک باشد. کلاینت معمولاً اطلاعاتی را به متدی استاتیک از این کلاس می‌فرستد و این متد بر اساس آن اطلاعات تصمیم می‌گیرد که کدام یک از پیاده سازی‌ها را برای کلاینت برگرداند.

از مزایای این الگو این است که اگر در نحوه ایجاد اشیاء تغییری رخ دهد هیچ نیازی به تغییر در کد کلاینت‌ها نخواهد بود. در این الگو اصل DIP از اصول پنجگانه SOLID به خوبی رعایت می‌شود چون که مسئولیت ایجاد زیرکلاس‌ها از دوش کلاینت برداشته می‌شود.

دیاگرام UML :

در شکل زیر دیاگرام UML الگوی Factory Method را مشاهده می‌کنید.



شرکت کنندگان در این الگو به شرح زیر هستند :

- Iproduct یک واسطه است که هر کلاینت از آن استفاده می‌کند. در اینجا کلاینت استفاده کننده نهایی است مثلاً می‌تواند متد

main یا هر متدی در کلاسی خارج از این الگو باشد. ما می‌توانیم پیاده‌سازی‌های مختلفی بر حسب نیاز از واسط Iproduct ایجاد کنیم.

- ConcreteProduct یک پیاده‌سازی از واسط Iproduct است، برای این کار بایستی کلاس پیاده‌سازی (ConcreteProduct) از این واسط (IProduct) مشتق شود.

- Icreator واسطیست که Factory Method را تعریف می‌کند. پیاده‌ساز این واسط بر اساس اطلاعاتی دریافتی کلاس صحیح را ایجاد می‌کند. این اطلاعات از طریق پارامتر برایش ارسال می‌شوند. همانطور که گفتیم این عملیات بر عهده پیاده‌ساز این واسط است و ما در این نمودار این وظیفه را فقط بر عهده ConcreteCreator گذاشته ایم که از واسط Icreator مشتق شده است.

پیاده‌سازی UML به صورت زیر است:

در ابتدا کلاس واسط IProduct تعریف شده است.

```
interface IProduct
{
    // در اینجا بر حسب نیاز فیلدها و یا امضای متدها قرار می‌گیرند
}
```

در این مرحله ما پند پیاده‌سازی از IProduct انجام می‌دهیم.

```
class ConcreteProductA : IProduct
{
    // پیاده‌سازی A
}

class ConcreteProductB : IProduct
{
    // پیاده‌سازی B
}
```

در این مرحله کلاس انتزاعی Creator تعریف می‌شود.

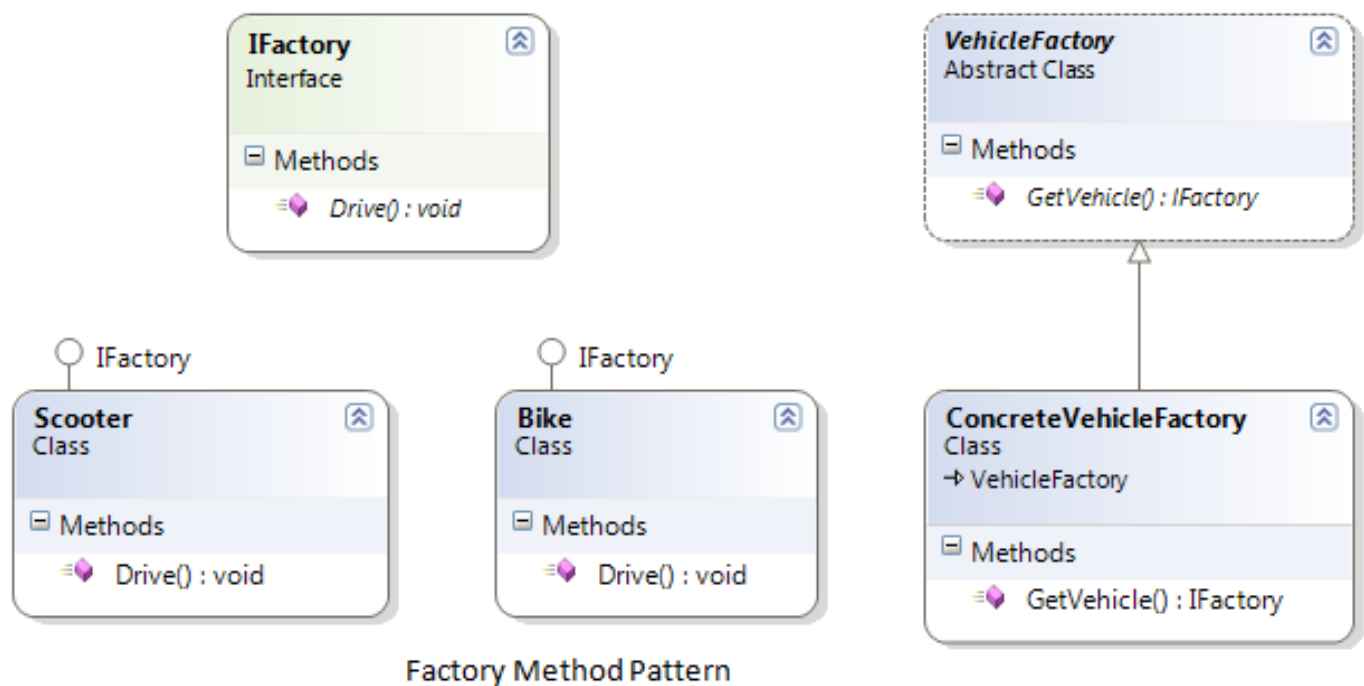
```
abstract class Creator
{
    // این متد بر اساس نوع ورودی انتخاب مناسب را انجام و باز می‌گرداند
    public abstract IProduct FactoryMethod(string type);
}
```

در این مرحله ما با ارث‌بری از Creator متد Abstract آن را به شیوه خودمان پیاده‌سازی می‌کنیم.

```
class ConcreteCreator : Creator
{
    public override IProduct FactoryMethod(string type)
    {
        switch (type)
        {
            case "A": return new ConcreteProductA();
            case "B": return new ConcreteProductB();
            default: throw new ArgumentException("Invalid type", "type");
        }
    }
}
```

مثالی از Factory Pattern در C# :

برای روشن‌تر شدن موضوع، یک مثال کاملتر ارائه داده می‌شود. در شکل زیر طراحی این برنامه نشان داده شده است.



Factory Method Pattern

کد برنامه به شرح زیر است :

```

using System;

namespace FactoryMethodPatternRealWordConsoleApp
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            VehicleFactory factory = new ConcreteVehicleFactory();

            IFactory scooter = factory.GetVehicle("Scooter");
            scooter.Drive(10);

            IFactory bike = factory.GetVehicle("Bike");
            bike.Drive(20);

            Console.ReadKey();
        }
    }

    public interface IFactory
    {
        void Drive(int miles);
    }

    public class Scooter : IFactory
    {
        public void Drive(int miles)
        {
            Console.WriteLine("Drive the Scooter : " + miles.ToString() + "km");
        }
    }

    public class Bike : IFactory
    {
        public void Drive(int miles)
        {

```

```
        Console.WriteLine("Drive the Bike : " + miles.ToString() + "km");
    }
}

public abstract class VehicleFactory
{
    public abstract IFactory GetVehicle(string Vehicle);
}

public class ConcreteVehicleFactory : VehicleFactory
{
    public override IFactory GetVehicle(string Vehicle)
    {
        switch (Vehicle)
        {
            case "Scooter":
                return new Scooter();
            case "Bike":
                return new Bike();
            default:
                throw new ApplicationException(string.Format("Vehicle '{0}' cannot be created",
Vehicle));
        }
    }
}
}
```

خروجی اجرای برنامه فوق به شکل زیر است :



```
Drive the Scooter : 10km
Drive the Bike : 20km
```

فایل این برنامه ضمیمه شده است، از لینک مقابل دانلود کنید [FactoryMethodPatternRealWordConsolApp.zip](#)

در مقالات بعدی مثال‌های کاربردی‌تر و جامع‌تری از این الگو و الگوهای مرتبط ارائه خواهیم کرد...

نظرات خوانندگان

نویسنده:

سید ایوب کوبی

تاریخ:

۱۹:۲۲ ۱۳۹۲/۰۷/۰۲

ممنونم بابت توضیحی که در مورد این الگو ارائه دادید و همچنین مثال خوبی که ارائه کردید، ولی چند تا سوال:

1- چرا کلاس VehicleFactory هم از نوع اینترفیس انتخاب نشده است؟ (آیا این موضوع سلیقه ای است؟)

2- استفاده از کلمه کلید string به جای نام کلاس String آیا تفاوتی در سرعت اجرا ایجاد میکند؟

3- چرا در دیاگرام uml رابطه بین ConcreteCreator و ConcreteProduct از نوع dependency است و از نوع Association نیست؟

4- چرا در ویژوال استودیو تولید خودکار uml از کد موجود متفاوت با دیاگرام فعلی است، مثلاً نوع روابط درست نمایش داده

همیشه و همچنین رابطه ای که در مورد 3 اشاره شد در اینجا وجود ندارد؟ آیا علتش نقص در این ابزار است؟ اگر بله، آیا ابزاری وجود دارد که دیاگرام رو دقیقتر جنریت کند؟

و یک نکته:

در کلاسهای Scooter و Bike نیازی به استفاده از متد ToString برای تبدیل مقدار عددی miles نیست چون با یک عبارت رشته ای دیگه جمع شده به صورت درونی این متد توسط CLR فراخوانی میشه. البته این مورد رو Resharper دوست داشتنی تذکر داد. بهتره قبل از ارائه سورس پیشنهادات Resharpe هم روی کد اعمال بشه تا کد در بهترین وضعیت ارائه بشه. ممنونم/.

نویسنده:

مجتبی شاطری

تاریخ:

۰۰:۳ ۱۳۹۲/۰۷/۰۳

جواب سوال اول :

بله کلاس VehicleFactory میتونه اینترفیس باشه. در اینجا سلیقه ای انجام شده. اما ممکنه در جایی نیاز باشه که ما بخواهیم ورژن پذیری را تو پروژمون لحاظ کنیم که از کلاس abstract استفاده می‌کنیم. ورژن پذیر بودن یعنی اینکه اگر شما متدی به اینترفیس اضافه کنید ، بایستی در تمام کلاسهایی که از آن اینترفیس ارث بری کردند پیاده سازی اون متد را انجام دهید. در کلاس abstract شما به راحتی متدی تعریف می‌کنید که نیاز نیست برای همه استفاده کننده‌ها اون متد را override کنید. این یعنی ورژن پذیری بهتر.

جواب سوال دوم :

string در واقع یک نام مستعار برای کلاس System.String هست. مثل int برای کلاس System.Int32. پس تفاوتی در سرعت ندارند و همیشه از کلاس String هم در اینجا استفاده کرد. چند نمونه برای مثال براتون میزارم :

```
string myagebyStringClass = String.Format("My age is {0}", 27);
```

معادل با :

```
string myagebystringType = string.Format("My age is {0}", 27);
```

و اینم چند نمونه دیگه :

```
object: System.Object
string: System.String
bool: System.Boolean
byte: System.Byte
sbyte: System.SByte
short: System.Int16
ushort: System.UInt16
int: System.Int32
uint: System.UInt32
```

```
long:    System.Int64
ulong:   System.UInt64
float:   System.Single
double:  System.Double
decimal: System.Decimal
char:    System.Char
```

جواب سوال سوم :

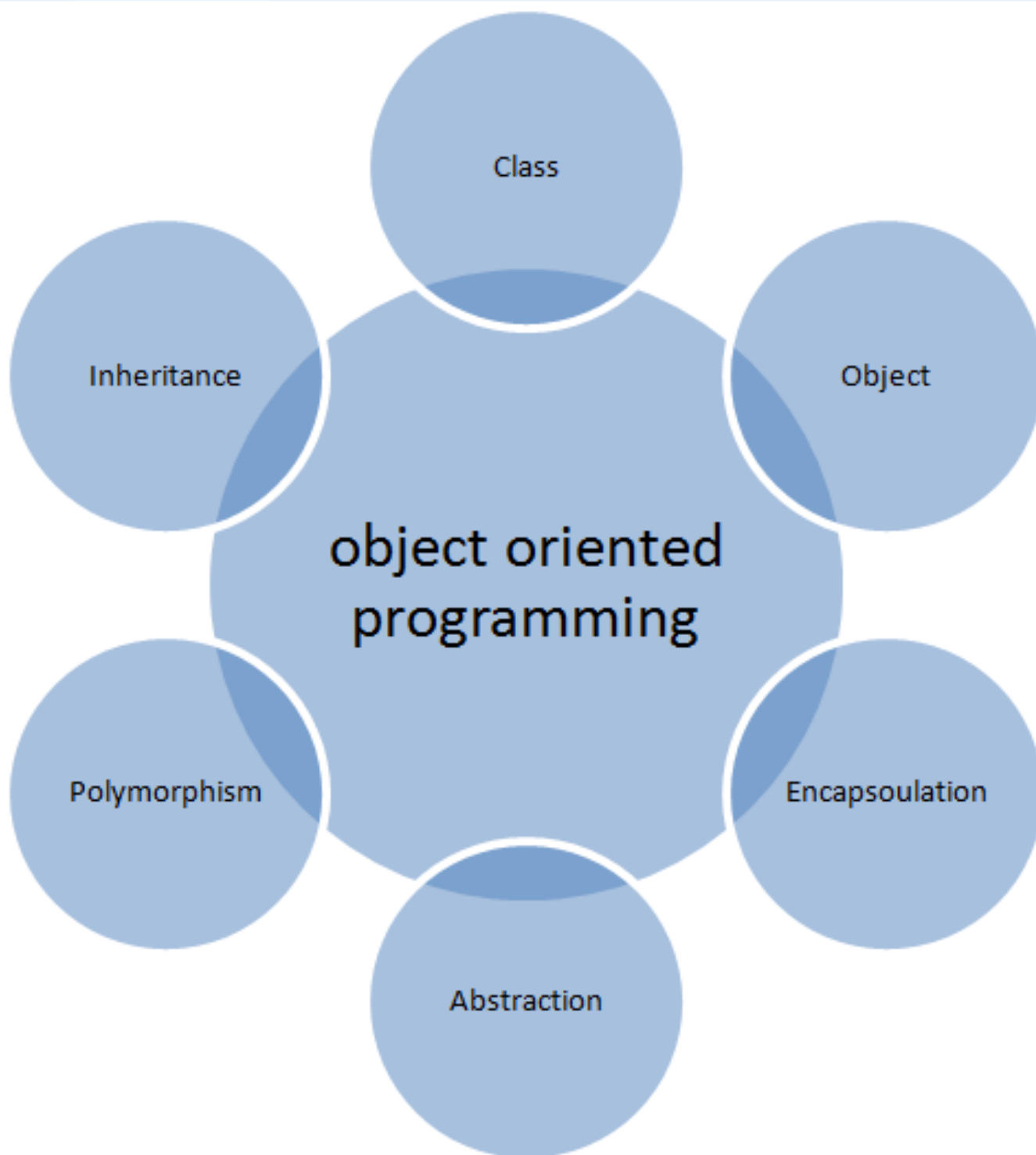
همونطور که میدونید رابطه Association (انجمنی) مربوط به ارتباطی یک به یک هستش. البته دو نوع هم داره که یکیش Aggregation (تجمع) و دیگری Composition (ترکیب) است. از اونجایی که نباید ConcreteProduct به ConcreteCreator وابسته باشه پس ما از رابطه Association در این مدل استفاده نمی‌کنیم. در مثال‌ها هم مشخص هست.

جواب سوال چهارم من نمی‌دونم.

درباره اون نکته Reshaper هم حرف شما صحیح هست . البته این یک مثال کلی هست. ممنون که این نکته رو یاد آوری کردید.

مخاطب چه کسی است؟

این مقاله برای کسانی در نظر گرفته شده است که حداقل پیش زمینه ای در مورد برنامه نویسی شی گرا داشته باشند. کسانی که تفاوت بین کلاس‌ها و اشیاء را میدانند و میتوانند در مورد ارکان پایه ای برنامه نویسی شی گرایی نظیر: کپسوله سازی (Encapsulation)، کلاس‌های انتزاعی (Abstraction)، چند ریختی (Polymorphism)، ارث بری (Inheritance) و... صحبت کنند.



مقدمه :

در جهان شی گرا ما فقط اشیاء را میبینیم که با یکدیگر در ارتباط هستند. کلاس ها، شی ها، ارث بری، کپسوله سازی، کلاس های انتزاعی و ... کلماتی هستند که ما هر روز در حرفه ای خودمان بارها آنها را می شنویم. در دنیای مدرن نرم افزار، بدون شک هر توسعه دهنده ی نرم افزار، یکی از انواع زبان های شی گرا را برای استفاده انتخاب میکند. اما آیا او واقعا میداند که برنامه نویسی شی گرا به چه معنی است؟ آیا او واقعا از قدرت شی گرایی استفاده میکند؟ در این مقاله تصمیم گرفته ایم پای خود را فراتر از ارکان پایه ای برنامه نویسی شی گرا قرار دهیم و بیشتر در مورد طراحی شی گرا صحبت کنیم. **طراحی شی گرا :**

طراحی شی گرا یک فرایند از برنامه ریزی یک سیستم نرم افزاری است که در آن اشیاء برای حل مشکلات خاص با یکدیگر در ارتباط هستند. در حقیقت یک طراحی شی گرای مناسب، کار توسعه دهنده را آسان میکند و یک طراحی نامناسب تبدیل به یک

فاجعه برای او میشود. هر کسی چگونه شروع میکند؟



وقتی کسانی شروع به ایجاد معماری نرم افزار میکنند، نشان میدهند که اهداف خوبی در سر دارند. آنها سعی میکنند از تجارب خود برای ساخت یک طراحی زیبا و تمیز استفاده کنند.



اما با گذشت زمان، نرم افزار کارایی خود را از دست میدهد و بلااستفاده میشود. با هر درخواست ایجاد ویژگی جدید در نرم افزار، به تدریج نرم افزار شکل خود را از دست میدهد و در نهایت ساده‌ترین تغییرات در نرم افزار موجب تلاش و دقت زیاد، زمان طولانی و مهمتر از همه بالا رفتن تعداد باگها در نرم افزار میشود.

چه کسی مقصر است؟

"تغییرات" یک قسمت جدایی ناپذیر از جهان نرم افزار هستند بنابراین ما نمیتوانیم "تغییر" را مقصر بدانیم و در حقیقت این طراحی ما است که مشکل دارد.

یکی از بزرگترین دلایل مخرب کننده‌ی نرم افزار، تعریف وابستگی‌های ناخواسته و بیخود در قسمت‌های مختلف سیستم است. در این گونه طراحی‌ها، هر قسمت از سیستم وابسته به چندین قسمت دیگر است، بنابراین تغییر یک قسمت، بر روی قسمت‌های دیگر نیز تاثیر میگذارد و باعث این چنین مشکلاتی میشود. ولی در صورتی که ما قادر به مدیریت این وابستگی باشیم در آینده خواهیم توانست از این سیستم نرم افزاری به آسانی نگهداری کنیم.

مثال:

هر گونه تغییری در لایه دسترسی به داده‌ها

بر روی لایه منطقی تاثیر میگذارد



راه حل : اصول، الگوهای طراحی و معماری نرم افزار معماری نرم افزار به عنوان مثال MVC, MVP, 3-Tire به ما میگویند که پروژه‌ها از چه ساختاری استفاده میکنند. الگوهای طراحی یک سری راه حل‌های قابل استفاده‌ی مجدد را برای مسائلی که به طور معمول اتفاق می‌افتند، فراهم میکند. یا به عبارتی دیگر الگوهای طراحی راه کارهایی را به ما معرفی میکنند که میتوانند برای حل مشکلات کد نویسی بارها مورد استفاده قرار بگیرند. **اصول به ما میگویند** اینها را انجام بده تا به آن دست پیدا کنی و اینکه چطور انجامش میدهی به خودت بستگی دارد. هر کس یک سری اصول را در زندگی خود تعریف میکند مانند : "من هرگز دروغ نمیگویم" یا "من هرگز سیگار نمی‌کشم" و از این قبیل. او با دنبال کردن این اصول زندگی آسانی را برای خودش ایجاد میکند. به همین شکل، طراحی شی گرا هم مملو است از اصولی که به ما اجازه میدهد تا با طراحی مناسب مشکلاتمان را مدیریت کنیم.

آقای رابرت مارتین (Robert Martin) این موارد را به صورت زیر طبقه بندی کرده است :

1- اصول طراحی کلاس‌ها که SOLID نامیده می‌شوند.

2- اصول انسجام بسته بندی

3- اصول اتصال بسته بندی

در این مقاله ما در مورد اصول SOLID به همراه مثال‌های کاربردی صحبت خواهیم کرد.

SOLID مخفی از 5 اصول معرفی شده توسط آقای مارتین است:

S -> Single responsibility Principle

O-> Open Close Principle

L-> Liskov substitution principle

I -> Interface Segregation principle

D-> Dependency Inversion principle

S - SRP - Single responsibility Principle (اصل 1)

به کد زیر توجه کنید :

```
public class Employee
{
    public string EmployeeName { get; set; }
    public int EmployeeNo { get; set; }

    public void Insert(Employee e)
    {
        //Database Logic written here
    }
    public void GenerateReport(Employee e)
    {
        //Set report formatting
    }
}
```

در کد بالا هر زمان تغییری در یک قسمت از کد ایجاد شود این احتمال وجود دارد که قسمت دیگری از آن مورد تاثیر این تغییر قرار بگیرد و به مشکل برخورد کنید. دلیل نیز مشخص است : هر دو در یک خانه‌ی مشابه و دارای یک والد یکسان هستند.

برای مثال با تغییر یک پراپرتی ممکن است متدهای هم خانه که از آن استفاده میکنند با مشکل مواجه شوند و باید این تغییرات را نیز در آنها انجام داد. در هر صورت خیلی مشکل است که همه چیز را کنترل کنیم. بنابراین تنها تغییر موجب دوبرابر شدن عملیات تست میشود و شاید بیشتر.

اصل SRP برای رفع این مشکل میگوید "هر ماژول نرم افزاری میبایست تنها یک دلیل برای تغییر داشته باشد".

(منظور از ماژول نرم افزاری همان کلاس‌ها ، توابع و ... است و عبارت "دلیل برای تغییر" همان مسئولیت است.) به عبارتی هر شی باید یک مسئولیت بیشتر بر عهده نداشته باشد. هدف این قانون جدا سازی مسئولیت‌های چسبیده به هم است. به عنوان مثال کلاسی که هم مسئول ذخیره سازی و هم مسئول ارتباط با واسط کاربر است، این اصل را نقض می‌کند و باید به دو کلاس مجزا تقسیم شود.

برای رسیدن به این منظور میتوانیم مثال بالا را به صورت 3 کلاس مختلف ایجاد کنیم :

Employee 1- : که حاوی خاصیت‌ها است.

EmployeeDB 2- : عملیات دیتابسی نظیر درج رکورد و واکشی رکوردها از دیتابیس را انجام میدهد.

EmployeeReport 3- : وظایف مربوط به ایجاد گزارش‌ها را انجام میدهد.

کد حاصل :

```
public class Employee
{
    public string EmployeeName { get; set; }
    public int EmployeeNo { get; set; }
}

public class EmployeeDB
{
    public void Insert(Employee e)
    {
        //Database Logic written here
    }
    public Employee Select()
    {
        //Database Logic written here
    }
}

public class EmployeeReport
{
    public void GenerateReport(Employee e)
```

```
{
    //Set report formatting
}
```

این روش برای متدها نیز صدق میکند به طوری که هر متد باید مسئولیت واحدی داشته باشد.
برای مثال قطعه کد زیر اصل SRP را نقض میکند :

```
//Method with multiple responsibilities - violating SRP
public void Insert(Employee e)
{
    string StrConnectionString = "";
    SqlConnection objCon = new SqlConnection(StrConnectionString);
    SqlParameter[] SomeParameters=null;//Create Parameter array from values
    SqlCommand objCommand = new SqlCommand("InertQuery", objCon);
    objCommand.Parameters.AddRange(SomeParameters);
    ObjCommand.ExecuteNonQuery();
}
```

این متد وظایف مختلفی را انجام میدهد مانند اتصال به دیتابیس ، ایجاد پارامترها برای مقادیر، ایجاد کوئری و در نهایت اجرای آن بر روی دیتابیس.
اما با توجه به اصل SRP میتوان آن را به صورت زیر بازنویسی کرد :

```
//Method with single responsibility - follow SRP
public void Insert(Employee e)
{
    SqlConnection objCon = GetConnection();
    SqlParameter[] SomeParameters=GetParameters();
    SqlCommand ObjCommand = GetCommand(objCon,"InertQuery",SomeParameters);
    ObjCommand.ExecuteNonQuery();
}

private SqlCommand GetCommand(SqlConnection objCon, string InsertQuery, SqlParameter[] SomeParameters)
{
    SqlCommand objCommand = new SqlCommand(InsertQuery, objCon);
    objCommand.Parameters.AddRange(SomeParameters);
    return objCommand;
}

private SqlParameter[] GetParaeters()
{
    //Create Paramter array from values
}

private SqlConnection GetConnection()
{
    string StrConnectionString = "";
    return new SqlConnection(StrConnectionString);
}
```

نظرات خوانندگان

نویسنده: حسن دهیاری
تاریخ: ۲۰:۲۷ ۱۳۹۲/۰۷/۲۳

با سلام.ضمن تشکر.یک سوال داشتم.آیا با توجه به اصل SRP نباید کلاس EmployeeDB تفکیک شود.چون که دو کار را انجام میدهد.ممنون میشم توضیح دهید.

نویسنده: محسن خان
تاریخ: ۲۳:۲۶ ۱۳۹۲/۰۷/۲۳

تنها دلیل تغییر کلی این کلاس در آینده، تغییر خاصیت‌های شیء کارمند است. بنابراین اصل تک مسئولیتی را نقض نمی‌کند. اگر این کلاس برای مثال دو Select داشت که یکی لیست کارمندان و دیگری لیست نقش‌های سیستم را بازگشت می‌داد، در این حالت تک مسئولیتی نقض می‌شد. ضمناً این نوع طراحی تحت عنوان الگوی مخزن یا لایه سرویس و امثال آن، یک طراحی پذیرفته شده و عمومی است. اگر قصد دارید که کوئری‌های خاص آن‌را طبقه بندی کنید می‌شود مثلاً از [Specification pattern](#) استفاده کرد.

نویسنده: فراز
تاریخ: ۰:۴۹ ۱۳۹۳/۰۵/۱۹

با سلام ممنون از مقاله ای که گذاشتین من 3 سال هست تقریباً به صورت دست پا شکسته دنبال OOAD می‌گردم که در عمل توضیح داده باشد که شما این کار رو انجام دادین با سپاس فراوان

در [قسمت قبل](#) در مورد اصل Single responsibility Principle یا به اختصار SRP صحبت شد. در این قسمت قصد داریم اصل دوم از اصول SOLID را مورد بررسی قرار دهیم.

اصل 2 (OCP – Open Close Principle)

فرض میکنیم که شما میخواهید یک طبقه بین طبقه اول و دوم خانه‌ی 2 طبقه‌ی خود اضافه کنید. فکرمیکنید امکان پذیر است؟



راه حل هایی که ممکن است به ذهن شما خطور کنند :

- 1- زمانی که برای اولین بار در حال ساخت خانه هستید آن را 3 طبقه بسازید و طبقه‌ی وسط را خالی نگه دارید. اینطوری هر زمان که شما بخواهید میتوانید از آن استفاده کنید. به هر حال این هم یک راه حل است.
 - 2- خراب کردن طبقه دوم و ساخت دو طبقه‌ی جدید که خوب اصلا معقول نیست.
- کد زیر را مشاهده کنید :

```
public class EmployeeDB
{
    public void Insert(Employee e)
    {
        //Database Logic written here
    }
    public Employee Select()
    {
        //Database Logic written here
    }
}
```

متد Select در کلاس EmployeeDB توسط کاربران و مدیر مورد استفاده قرار میگیرد. در این بین ممکن است مدیر نیاز داشته باشد تغییراتی را در آن انجام دهد. اگر مدیر این کار را برای برآورده کردن نیاز خود انجام دهد، روی دیگر قسمت‌ها نیز تاثیر میگذارد، به علاوه ایجاد تغییرات در راه حل‌های تست شده‌ی موجود ممکن است موجب خطاهای غیر منتظره ای شود. چگونه ممکن است که رفتار یک برنامه تغییر کند بدون اینکه کد آن ویرایش شود؟ چگونه می‌توانیم بدون تغییر یک موجودیت نرم افزاری کارکرد آن را تغییر دهیم؟

اصل OCP میگوید : "ماژول‌های نرم افزار باید برای تغییرات بسته و برای توسعه باز باشند."

راه حل هایی که OCP را نقض نمیکنند : 1- استفاده از وراثت (inheritance):

ایجاد یک کلاس جدید به نام EmployeeManagerDB که از کلاس EmployeeDB ارث بری کند و متد Select آن را جهت نیاز خود بازنویسی کند.

```
public class EmployeeDB
{
    public virtual Employee Select()
    {
        //Old Select Method
    }
}
public class EmployeeManagerDB : EmployeeDB
{
    public override Employee Select()
    {
        //Select method as per Manager
        //UI requirement
    }
}
```

این انتخاب خیلی خوبی است در صورتی که این تغییرات در زمان طراحی اولیه پیش بینی شده باشد و همکنون قابل استفاده باشند.

کد UI هم به شکل زیر خواهد بود :

```
//Normal Screen
EmployeeDB objEmpDb = new EmployeeDB();
Employee objEmp = objEmpDb.Select();

//Manager Screen
EmployeeDB objEmpDb = new EmployeeManagerDB();
Employee objEmp = objEmpDb.Select();
```

2- متدهای الحاقی (Extension Method):

اگر شما از NET 3.5. یا بالاتر از آن استفاده میکنید، دومین راه استفاده از متدهای الحاقی است که به شما اجازه میدهد بدون هیچ دست زدن به نوعهای موجود، متدهای جدیدی را به آنها اضافه کنید.

```
Public static class MyExtensionMethod{
    public static Employee managerSelect(this EmployeeDB employeeDB) {
        //Select method as per Manager
    }
}

//Manager Screen
Employee objEmp = EmployeeDB.managerSelect();
```

البته ممکن است راههای دیگری هم برای رسیدن به این منظور وجود داشته باشد. درقسمتهای بعدی قانونهای دیگر را بررسی خواهیم کرد.

بخش‌های پیشین: [اصول طراحی شی گرا SOLID - #بخش اول اصل SRP](#)

[اصول طراحی شی گرا SOLID - #بخش دوم اصل OCP](#)

اصل 3 (L - LSP - Liskov substitution principle

اصل LSP میگوید: "زیر کلاس‌ها باید بتوانند جایگزین نوع پایه‌ی خود باشند".

مقایسه با جهان واقعی:



شغل یک پدر تجارت املاک است درحالی که پسرش دوست دارد فوتبالیست شود.

یک پسر هیچگاه نمیتواند جایگزین پدرش شود، با اینکه که آنها به یک سلسله مراتب خانوادگی تعلق دارند.

در یک مثال عمومی تر بررسی میکنیم :

به طور معمول زمانی که ما در مورد اشکال هندسی صحبت میکنیم ، مستطیل را یک کلاس پایه برای مربع میدانیم. به کد زیر توجه کنید :

```
public class Rectangle
{
    public int Width { get; set; }
    public int Height { get; set; }
}

public class Square:Rectangle
{
    //codes specific to
    //square will be added
}
```

و میتوان گفت :

```
Rectangle o = new Rectangle();
o.Width = 5;
o.Height = 6;
```

بسیار خوب، اما با توجه به LSP باید قادر باشیم مستطیل را با مربع جایگزین کنیم. سعی میکنیم این کار را انجام دهیم :

```
Rectangle o = new Square();
o.Width = 5;
o.Height = 6;
```

موضوع چیست؟ مگر مربع می تواند طول و عرض نا برابر داشته باشد؟! امکان ندارد.

خوب این به چه معنی است؟ به این معنی که ما نمیتوانیم کلاس پایه را با کلاس مشتق شده جایگزین کنیم و باز هم این معنی را میدهد که ما داریم اصل LSP را نقض میکنیم.

آیا ما میتوانیم طول و عرض را در کلاس Square طبق کد زیر دوباره نویسی کنیم؟

```
public class Square : Rectangle
{
    public override int Width
    {
        get{return base.Width;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
    public override int Height
    {
        get{return base.Height;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```

باز هم اصل LSP نقض میشود چون ما داریم رفتار خاصیت های طول و عرض در کلاس مشتق شده را تغییر میدهیم. ولی با توجه به کد بالا یک مستطیل نمیتواند طول و عرض برابر داشته باشد چون در صورت برابری دیگر مستطیل نیست.

اما راه حل چیست؟

یک کلاس انتزاعی (abstract) را به شکل زیر ایجاد و سپس دو کلاس Square و Rectangle را از آن مشتق میکنیم :

```
public abstract class Shape
{
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }
}
```

همکنون ما دو کلاس مستقل از هم داریم. یکی Square و دیگری Rectangle که هر دو از کلاس Shape مشتق شده اند. حالا میتوانیم بنویسیم :

```
Shape o = new Rectangle();
o.Width = 5;
o.Height = 6;

Shape o = new Square();
o.Width = 5; //both height and width become 5
o.Height = 6; //both height and width become 6
```

زمانی که ما در مورد اشکال هندسی صحبت میکنیم ، هیچ قاعده‌ی خاصی جهت اندازه‌ی طول و عرض نیست. ممکن است برابر باشند یا نباشند.

در قسمت بعدی اصل ISP را مورد بررسی قرار خواهیم داد.

نظرات خوانندگان

نویسنده: فدورا

تاریخ: ۱۳۹۲/۰۷/۰۹ ۱۰:۳۸

سلام.

خیلی ممنون از بابت مقالات آموزشی خوبتون.

فقط سوالی برای من تو این بخش سوم پیش اومد و اون هم اینکه بعد از تعریف کلاس abstract تعریف کلاس‌های rectangle و square به چه شکل شد؟ لطفا کد اون کلاس‌ها رو هم اضافه کنید.
با تشکر

نویسنده: ناصر طاهری

تاریخ: ۱۳۹۲/۰۷/۰۹ ۱۳:۴۸

ممنون.

کلاس‌های Rectangle و Square هر دو به همون شکل باقی می‌مونند با این تفاوت که هر دو از کلاس Shape مشتق شده اند و میتوانند خاصیت‌های Width و Height را طبق نیاز خود دوباره نویسی کنند (override). کلاس Restangle:

```
public class Rectangle : Shape
{
    شما میتوانید خاصیت‌ها طول و عرض در کلاس پایه را در صورت نیاز دوباره نویسی کنید//
}
```

کلاس Square :

```
public class Square : Shape
{
    دوباره نویسی کردن خاصیت‌های طول و عرض در کلاس پایه جهت برابر کردن طول و عرض مربع
    public override int Width
    {
        get{return base.Width;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
    public override int Height
    {
        get{return base.Height;}
        set
        {
            base.Height = value;
            base.Width = value;
        }
    }
}
```

که با توجه به کدهای بالا ، کلاسهای مشتق شده‌ی Square و Restangle میتوانند جایگزین کلاس پایه خود یعنی Shape شوند :

```
Shape o = new Rectangle();
o.Width = 5;
o.Height = 6;

Shape o = new Square();
o.Width = 5; // میشوند 5
o.Height = 6; // میشوند 6
طول و عرض هر دو برابر 5 میشوند
عرض و طول هر دو برابر 6 میشوند
```

بخش‌های پیشین: اصول طراحی شی گرا SOLID - #بخش اول اصل SRP

اصول طراحی شی گرا SOLID - #بخش دوم اصل OCP اصول طراحی شی گرا SOLID - #بخش سوم اصل LSP

I - ISP- Interface Segregation principle (اصل 4)

مقایسه با دنیای واقعی:



بیایید فکر کنیم شما یک کامپیوتر دسکتاپ جدید خریداری کرده اید. شما یک زوج پورت USB، چند پورت سریال، یک پورت VGA و ... را پیدا میکنید. اگر شما بدنه‌ی کیس خود را باز کنید، تعدادی اسلات بر روی مادربورد خود که برای اتصال قطعات مختلف با یکدیگر هستند را مشاهده خواهید کرد که عمدتاً مهندسان سخت افزار در زمان متاثر از آنها استفاده میکنند. این اسلات‌ها تا زمانی که بدنه کیس را باز نکنید قابل مشاهده نخواهند بود. به طور خلاصه تنها رابطه‌های مورد نیازی که برای شما ساخته شده اند، قابل مشاهده خواهند بود.

و فرض کنید شما به یک توپ فوتبال نیاز دارید. به یک فروشگاه وسایل ورزشی میروید و فروشنده شروع به نشان دادن انواع توپ ها، کفش ها، لباس و گرم کن‌های ورزشی، لباس شنا و کاراته، زانوبند و ... کرده است. در نهایت ممکن است شما چیزی را خریداری کنید که اصلاً مورد نیازتان نبوده است یا حتی ممکن است فراموش کنیم که ما چرا اینجا هستیم! **ارائه مشکل:** ما می‌خواهیم یک سیستم مدیریت گزارشات را توسعه بدهیم. اولین کاری که انجام می‌دهیم ایجاد یک لایه منطقی که توسط سه UI مختلف دیگر مورد استفاده قرار میگیرد.

1- EmployeeUI : نمایش گزارش‌های مربوط با کارمند وارد شده‌ی جاری در سایت.

2- ManagerUI : نمایش گزارش‌های مربوط به خود و تیمی که به او تعلق دارد.

3- AdminUI : نمایش گزارش‌های مربوط به کارمندان، مربوط به تیم و مربوط به شرکت مانند گزارش سود و زیان و ...

```
public interface IReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();

    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();

    void GenerateProfitReport();
}
```

```

public class ReportBAL : IReportBAL
{
    public void GeneratePFReport()
    { /*.....*/ }

    public void GenerateESICReport()
    { /*.....*/ }

    public void GenerateResourcePerformanceReport()
    { /*.....*/ }

    public void GenerateProjectSchedule()
    { /*.....*/ }

    public void GenerateProfitReport()
    { /*.....*/ }
}

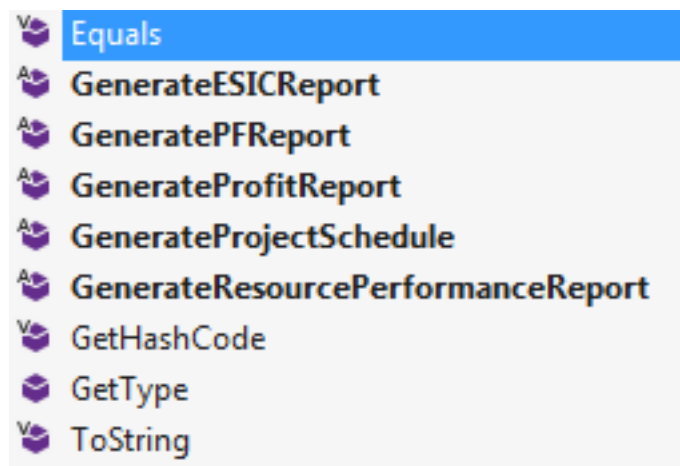
public class EmployeeUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}

public class ManagerUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport ();
        objBal.GenerateProjectSchedule ();
    }
}

public class AdminUI
{
    public void DisplayUI()
    {
        IReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
    }
}

```

حال زمانی که توسعه دهنده در هر UI عبارت objBal را تایپ کند، در لیست بازشوی آن (intellisense) به صورت زیر نمایش داده خواهد شد :



اما مشکل چیست؟ مشکل این است شخصی که روی لایه‌ی EmployeeUI کار میکند به تمام متدهاها به خوبی دسترسی دارد و این متدهای غیرضروری ممکن است باعث سردرگمی او شود.

این مشکل به این دلیل اتفاق می‌افتد که وقتی کلاسی یک واسط را پیاده‌سازی می‌کند، باید همه متدهای آن را نیز پیاده‌سازی کند. حالا اگر خیلی از این متدها توسط این کلاس استفاده نشود، می‌گوییم که این کلاس از مشکل واسط چاق رنج می‌برد.

اصل ISP می‌گوید: "کلاینتها نباید وابسته به متدهایی باشند که آنها را پیاده‌سازی نمی‌کنند."

برای رسیدن به این امر در مثال بالا باید آن واسط را به واسطه‌های کوچکتر تقسیم کرد. این تقسیم‌بندی باید بر اساس استفاده‌کنندگان از واسطها صورت گیرد.

ویرایش مثال بالا با در نظر گرفتن اصل ISP:

```
public interface IEmployeeReportBAL
{
    void GeneratePFReport();
    void GenerateESICReport();
}
public interface IManagerReportBAL : IEmployeeReportBAL
{
    void GenerateResourcePerformanceReport();
    void GenerateProjectSchedule();
}
public interface IAdminReportBAL : IManagerReportBAL
{
    void GenerateProfitReport();
}
public class ReportBAL : IAdminReportBAL
{
    public void GeneratePFReport()
    { /*.....*/ }

    public void GenerateESICReport()
    { /*.....*/ }

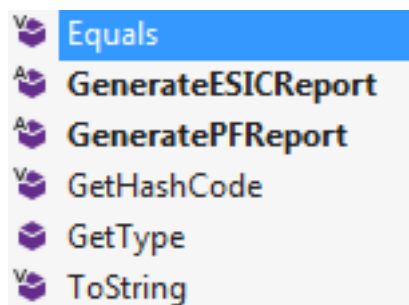
    public void GenerateResourcePerformanceReport()
    { /*.....*/ }

    public void GenerateProjectSchedule()
    { /*.....*/ }

    public void GenerateProfitReport()
    { /*.....*/ }
}
```

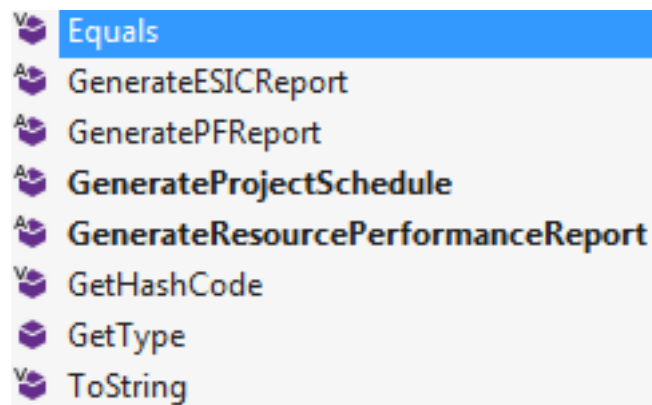
وضعیت نمایش:

```
public class EmployeeUI
{
    public void DisplayUI()
    {
        IEmployeeReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
    }
}
```

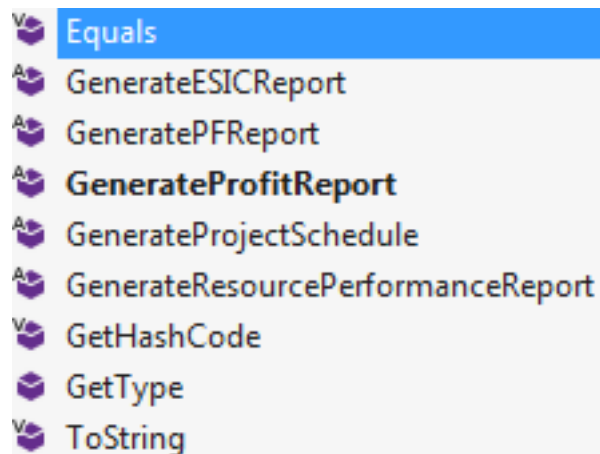


```
public class ManagerUI
{
    public void DisplayUI()
```

```
{
    IManagerReportBAL objBal = new ReportBAL();
    objBal.GenerateESICReport();
    objBal.GeneratePFReport();
    objBal.GenerateResourcePerformanceReport ();
    objBal.GenerateProjectSchedule ();
}
```



```
public class AdminUI
{
    public void DisplayUI()
    {
        IAdminReportBAL objBal = new ReportBAL();
        objBal.GenerateESICReport();
        objBal.GeneratePFReport();
        objBal.GenerateResourcePerformanceReport();
        objBal.GenerateProjectSchedule();
        objBal.GenerateProfitReport();
    }
}
```



DIP :

در قسمت بعدی آخرین اصل

را بررسی خواهیم کرد

بخش‌های پیشین : [اصول طراحی شی گرا SOLID - #بخش اول اصل SRP](#) [اصول طراحی شی گرا SOLID - #بخش دوم اصل OCP](#)

[اصول طراحی شی گرا SOLID - #بخش سوم اصل LSP](#)

[اصول طراحی شی گرا SOLID - #بخش چهارم اصل ISP](#)

اصل 5 - DIP - Dependency Inversion principle

مقایسه با دنیای واقعی:

همان مثال کامپیوتر را دوباره در نظر بگیرید. این کامپیوتر دارای قطعات مختلفی مانند RAM ، هارد دیسک، CD ROM و ... است که هر کدام به صورت مستقل به مادربرد متصل شده اند. این به این معنی است که اگر قسمتی از کار بیفتد میتوان آن را با یک قطعه‌ی جدید به آسانی تعویض کرد . حالا فقط تصور کنید که تمامی قطعات شدیداً به یکدیگر متصل شده اند آنوقت دیگر نمیتوانستیم قطعه ای را از مادربرد برداریم و به همین خاطر اگر مثلاً RAM از کار بیفتد ، باید یک مادربرد جدید خریداری کنید که برای شما گران تمام می‌شود.

به مثال زیر توجه کنید :

```
public class CustomerBAL
{
    public void Insert(Customer c)
    {
        try
        {
            //Insert logic
        }
        catch (Exception e)
        {
            FileLogger f = new FileLogger();
            f.LogError(e);
        }
    }
}

public class FileLogger
{
    public void LogError(Exception e)
    {
        //Log Error in a physical file
    }
}
```

در کد بالا کلاس CustomerBAL مستقیماً به کلاس FileLogger وابسته است که استثناءهای رخ داده را بر روی یک فایل فیزیکی لاگ میکند. حالا فرض کنید که چند روز بعد مدیریت تصمیم میگیرد که از این به بعد استثناءها بر روی یک Event Viewer لاگ شوند. اکنون چه میکنید؟ با تغییر کدها ممکن است با خطاهای زیادی روبرو شوید(درصورت تعداد بالای کلاسهایی که از کلاس FileLogger استفاده میکنند و فقط تعداد محدودی از آنها نیاز دارند که بر روی Event Viewer لاگ کنند). **DIP** به ما میگوید : " **ماژول‌های سطح بالا نباید به ماژولهای سطح پایین وابسته باشند، هر دو باید به انتزاعات وابسته باشند. انتزاعات نباید وابسته به جزئیات باشند، بلکه جزئیات باید وابسته به انتزاعات باشند.** "

در طراحی ساخت یافته، ماژولهای سطح بالا به ماژولهای سطح پایین وابسته بودند. این مسئله دو مشکل ایجاد می‌کرد:

1- ماژول‌های سطح بالا (سیاست گذار) به ماژول‌های سطح پایین (مجری) وابسته هستند. در نتیجه هر تغییری در ماژول‌های سطح پایین ممکن است باعث اشکال در ماژول‌های سطح بالا گردد.

2- استفاده مجدد از ماژول‌های سطح بالا در جاهای دیگر مشکل است، زیرا وابستگی مستقیم به ماژول‌های سطح پایین دارند. راه حل با توجه به اصل DIP :

```
public interface ILogger
{
    void LogError(Exception e);
}

public class FileLogger:ILogger
```

```

{
    public void LogError(Exception e)
    {
        //Log Error in a physical file
    }
}
public class EventViewerLogger : ILogger
{
    public void LogError(Exception e)
    {
        //Log Error in a Event Viewer
    }
}
public class CustomerBAL
{
    private ILogger _objLogger;
    public CustomerBAL(ILogger objLogger)
    {
        _objLogger = objLogger;
    }

    public void Insert(Customer c)
    {
        try
        {
            //Insert logic
        }
        catch (Exception e)
        {
            _objLogger.LogError(e);
        }
    }
}

```

در اینجا وابستگی‌های کلاس CustomerBAL از طریق سازنده آن در اختیارش قرار گرفته است. یک اینترفیس ILogger تعریف شده است به همراه دو پیاده سازی مختلف از آن مانند FileLogger و EventViewerLogger. یکی از انواع فراخوانی آن نیز می‌تواند به شکل زیر باشد:

```

var customerBAL = new CustomerBAL (new EventViewerLogger());
customerBAL.LogError();

```

اطلاعات بیشتر در دوره آموزشی "[بررسی مفاهیم معکوس سازی وابستگی‌ها و ابزارهای مرتبط با آن](#)".

نظرات خوانندگان

نویسنده: سعید سلیمانی فر
تاریخ: ۱۳۹۲/۰۷/۰۹ ۹:۳۱

خیلی مطلب خوبی بود! لذت بردیم متشکرم (:

نویسنده: بهزاد علی محمدزاده
تاریخ: ۱۳۹۲/۰۷/۱۹ ۱۶:۲۲

اقای طاهری با تشکر . امکان داره منبع رو معرفی کنید . به دنبال یه کتاب یا منبع آموزشی خوب در این زمینه هستم که البته نمونه ها رو با #C انجام داده باشه .

نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۱۹ ۱۷:۴۱

چند مقاله ای که من اونها رو مطالعه کردم : [اصول طراحی SOLID SOLIDify your software design concepts through SOLID](#)
[Articles by Christian Vos](#) [Object Oriented Design Principles](#) [SOLID by example](#) [SOLID Agile Development](#) [Articles Of SOLID](#) [SOLID Principles in C# - An Overview](#)