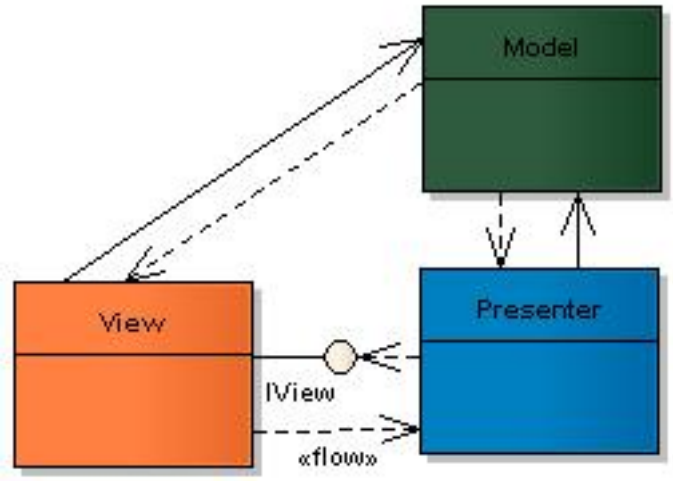
عنوان: **آشنایی با الگوی MVP**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۵/۲۸ ۱۲:۴۳:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

پروژه‌های زیادی را می‌توان یافت که اگر سورس کدهای آن‌ها را بررسی کنیم، یک اسپاگتی کد تمام عیار را در آن‌ها می‌توان مشاهده نمود. منطق برنامه، قسمت دسترسی به داده‌ها، کار با رابط کاربر، غیره و غیره همگی درون کدهای یک یا چند فرم خلاصه شده‌اند و آنچنان به هم گره خورده‌اند که هر گونه تغییر یا اعمال درخواست‌های جدید کاربران، سبب از کار افتادن قسمت دیگری از برنامه می‌شود.
همچنین از کدهای حاصل در یک پروژه، در پروژه‌های دیگر نیز نمی‌توان استفاده کرد (به دلیل همین در هم تنیده بودن قسمت‌های مختلف). حداقل نتیجه یک پروژه برای برنامه نویس، باید یک یا چند کلاس باشد که بتوان از آن به عنوان ابزار تسریع انجام پروژه‌های دیگر استفاده کرد. اما در یک اسپاگتی کد، باید مدتی طولانی را صرف کرد تا بتوان یک متد را از لابلای قسمت‌های مرتبط و گره خورده با رابط کاربر استخراج و در پروژه‌ای دیگر استفاده نمود. برای نمونه آیا می‌توان این کدها را از یک برنامه ویندوزی استخراج کرد و آن‌ها را در یک برنامه تحت وب استفاده نمود؟

یکی از الگوهایی که شیوه‌ی صحیح این جدا سازی را ترویج می‌کند، الگوی MVP یا Model-View-Presenter می‌باشد. خلاصه‌ی این الگو به صورت زیر است:



Model :
من می‌دانم که چگونه اشیاء برنامه را جهت حصول منطقی خاص، پردازش کنم.
من نمی‌دانم که چگونه باید اطلاعاتی را به شکلی بصری به کاربر ارائه داد یا چگونه باید به رخدادها یا اعمال صادر شده از طرف کاربر پاسخ داد.

View :
من می‌دانم که چگونه باید اطلاعاتی را به کاربر به شکلی بصری ارائه داد.
من می‌دانم که چگونه باید اعمالی مانند data binding و امثال آن را انجام داد.
من نمی‌دانم که چگونه باید منطق پردازشی موارد ذکر شده را فراهم آورم.

Presenter :

من می‌دانم که چگونه باید درخواست‌های رسیده کاربر به View را دریافت کرده و آن‌ها را به Model انتقال دهم.
من می‌دانم که چگونه باید اطلاعات را به Model ارسال کرده و سپس نتیجه‌ی پردازش آن‌ها را جهت نمایش در اختیار View قرار دهم.
من نمی‌دانم که چگونه باید اطلاعاتی را ترسیم کرد (مشکل View است نه من) و نمی‌دانم که چگونه باید پردازشی را بر روی اطلاعات انجام دهم. (مشکل Model است و اصلا ربطی به اینجانب ندارد!)

یک مثال ساده از پیاده سازی این روش
برنامه‌ای وبی را بنویسید که پس از دریافت شعاع یک دایره از کاربر، مساحت آن‌را محاسبه کرده و نمایش دهد.
یک تکست باکس در صفحه قرار خواهیم داد (txtRadius) و یک دکمه جهت دریافت درخواست کاربر برای نمایش نتیجه حاصل در یک برچسب به نام lblResult

الف) پیاده سازی به روش متداول (اسپاگتی کد)

```
protected void btnGetData_Click(object sender, EventArgs e)
      {
          lblResult.Text = (Math.PI * double.Parse(txtRadius.Text) *
double.Parse(txtRadius.Text)).ToString();
      }
```

بله! کار می‌کنه!
اما این مشکلات را هم دارد:
- منطق برنامه (روش محاسبه مساحت دایره) با رابط کاربر گره خورده.
- کدهای برنامه در پروژه‌ی دیگری قابل استفاده نیست. (شما متد یا کلاسی را این‌جا با قابلیت استفاده مجدد می‌توانید پیدا می‌کنید؟ آیا یکی از اهداف برنامه نویسی شیءگرا تولید کدهایی با قابلیت استفاده مجدد نبود؟)
- چگونه باید برای آن آزمون واحد نوشت؟

ب) بهبود کد و جدا سازی لایه‌ها از یکدیگر

در روش MVP متداول است که به ازای هر یک از اجزاء ابتدا یک interface نوشته شود و سپس این اینترفیس‌ها پیاده سازی گردد.

پیاده سازی منطق برنامه:

1- ایجاد Model :
یک فایل جدید را به نام CModel.cs به پروژه اضافه کرده و کد زیر را به آن خواهیم افزود:

```
using System;

namespace MVPTest
{
  public interface ICircleModel
  {
      double GetArea(double radius);
  }

  public class CModel : ICircleModel
  {
      public double GetArea(double radius)
      {
          return Math.PI * radius * radius;
      }
  }
}
```

همانطور که ملاحظه می‌کنید اکنون منطق برنامه از موارد زیر اطلاعی ندارد:

- خبری از textbox و برچسب و غیره نیست. اصلا نمی‌داند که رابط کاربری وجود دارد یا نه.
- خبری از رخدادهای برنامه و پاسخ دادن به آن‌ها نیست.
- از این کد می‌توان مستقیما و بدون هیچ تغییری در برنامه‌های دیگر هم استفاده کرد.
- اگر باگی در این قسمت وجود دارد، تنها این کلاس است که باید تغییر کند و بلافاصله کل برنامه از این بهبود حاصل شده می‌تواند بدون هیچگونه تغییری و یا به هم ریختگی استفاده کند.
- نوشتن آزمون واحد برای این کلاس که هیچگونه وابستگی به UI ندارد ساده است.

-2 ایجاد View :
فایل دیگری را به نام CView.cs را به همراه اینترفیس زیر به پروژه اضافه می‌کنیم:

```
namespace MVPTest
{
  public interface IView
  {
      string RadiusText { get; set; }
      string ResultText { get; set; }
  }
}
```

کار View دریافت ابتدایی مقادیر از کاربر توسط RadiusText و نمایش نهایی نتیجه توسط ResultText است البته با یک اما.
View نمی‌داند که چگونه باید این پردازش صورت گیرد. حتی نمی‌داند که چگونه باید این مقادیر را به Model جهت پردازش برساند یا چگونه آن‌ها را دریافت کند (به همین جهت از اینترفیس برای تعریف آن استفاده شده).

-3 ایجاد Presenter :
در ادامه فایل جدیدی را به نام CPresenter.cs با محتویات زیر به پروژه خواهیم افزود:

```
namespace MVPTest
{
  public class CPresenter
  {
      IView _view;

      public CPresenter(IView view)
      {
          _view = view;
      }

      public void CalculateCircleArea()
      {
          CModel model = new CModel();
          _view.ResultText = model.GetArea(double.Parse(_view.RadiusText)).ToString();
      }
  }
}
```

کار این کلاس برقراری ارتباط با Model است.
می‌داند که چگونه اطلاعات را به Model ارسال کند (از طریق _view.RadiusText) و می‌داند که چگونه نتیجه‌ی پردازش را در اختیار View قرار دهد. (با انتساب آن به _view.ResultText)
نمی‌داند که چگونه باید این پردازش صورت گیرد (کار مدل است نه او). نمی‌داند که نتیجه‌ی نهایی را چگونه نمایش دهد (کار View است نه او).
روش معرفی View به این کلاس به constructor dependency injection معروف است.

اکنون کد وب فرم ما که در قسمت (الف) معرفی شده به صورت زیر تغییر می‌کند:

```
using System;
```

```
namespace MVPTest
{
  public partial class _Default : System.Web.UI.Page, IView
  {
      protected void Page_Load(object sender, EventArgs e)
      {
      }

      public string RadiusText
      {
          get { return txtRadius.Text; }
          set { txtRadius.Text = value; }
      }
      public string ResultText
      {
          get { return lblResult.Text; }
          set { lblResult.Text = value; }
      }

      protected void btnGetData_Click(object sender, EventArgs e)
      {
          CPresenter presenter = new CPresenter(this);
          presenter.CalculateCircleArea();
      }
  }
}
```

در این‌جا یک وهله از Presenter برای برقراری ارتباط با Model ایجاد می‌شود. همچنین کلاس وب فرم ما اینترفیس View را نیز پیاده سازی خواهد کرد.

نظرات خوانندگان

نویسنده: زوزو
تاریخ: ۱۳۸۸/۰۵/۲۸ ۱۷:۵۶:۰۸

آیا این همون مدل 3 لایه برنامه نویسی هست که در اینجا بدون استفاده از دیتابیس به این شکل مطرح شده است؟

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۵/۲۸ ۱۸:۳۰:۲۱

سلام
احتمالا tier 3 را شنیده‌اید که به این صورت مطرح کردید.
n-tier نوعی معماری است که به شما در تهیه برنامه‌های توزیع شده کمک می‌کند و مهم‌ترین مزیت آن قابلیت بسط پذیری سیستم است. Tiering در مورد تخصیص منابع و نحوه‌ی توزیع آن‌ها بحث می‌کند. برای مثال دیتابیس سرور شما جدا است، منطق برنامه در سروری دیگر توسط یک وب سرویس قابل دسترسی است و سروری دیگر کار دریافت و ارائه این اطلاعات را به عهده خواهد داشت.
MVC که در ابتدا پدید آمد و بعد از آن MVP ، یک نوع الگوی برنامه نویسی شیءگرا هستند که به شما کمک خواهند کرد تا برنامه‌ی n-tier ایی با حداقل گره خوردگی و به هم پیچیدگی که اصطلاحا به آن Loosely coupled نیز گفته می‌شود، تولید کنید.

نویسنده: میثم جوادی
تاریخ: ۱۳۸۸/۰۵/۲۹ ۰۱:۵۲:۵۳

سلام، جناب نصیری اگه برنامه بزرگ بشه نیاز به الگوی Facade بیشتر نمیشه؟منظورم استفاده اش تو این الگو.
یه ازای همه کلاس ها باید اینترفیس تعریف کنیم؟
(حتی واسه گوگل کلمه MVP تازه است به طوری که did you mean... گوگل کلمه MVC رو پیشنهاد میکنه!)

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۵/۲۹ ۱۰:۲۵:۲۶

سلام
MVP جدید نیست و اولین مقاله در مورد آن به سال 1996 بر می‌گرده
http://www.wildcrest.com/Potel/Portfolio/mvp.pdf

- در این روش Presenter شما می‌تونه با یک Facade یا Service object جهت دریافت اطلاعات Model نیز در ارتباط باشه.

نویسنده: حسن
تاریخ: ۱۳۸۸/۰۵/۳۰ ۰۲:۴۸:۱۱

از کدهایی که نوشته بودید هیچ سر در نیاوردم (.net بود؟) ولی با MVC در phpآشنام، اینا فرق دارن یا فقط اسماشون در محیط‌های مختلف متفاوته؟

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۵/۳۰ ۱۱:۳۴:۵۵

فرق دارند (و البته در انحصار پلتفرم خاصی هم نیستند چون یک نوع الگوی برنامه نویسی‌اند).
سیر تکاملی این‌ها رو در تصویر زیر می‌تونید مشاهده کنید:
http://vahid.nasiri.googlepages.com/mvcmvp.png

تفاوت‌ها:

در MVP
view و model کاملا از هم جدا شده‌اند.
presenter کار رخدادگردانی عناصر UI را انجام می‌دهد
presenter کار به روز رسانی view را از طریق فراخوانی اینترفیس آن انجام می‌دهد

در MVC
view و model کاملا از هم جدا نیستند.
View کار رخدادگردانی عناصر UI را انجام می‌دهد
controller مدل را به view ارسال کرده و سپس view بر این اساس خودش را به روز می‌کند

نویسنده: مسعود
تاریخ: ۱۳۸۸/۰۵/۳۰ ۱۳:۰۴:۴۳

آقای نصیری اول که خسته نباشید.

دوم اینکه من اینو درک نمی کنم که وقتی من اینتر فیس برای بخشی نوشتم و اونوimplements کردم و از این نوع معاری استفاده کردم ، چرا بهتره.

من سوالم اینجاست که اگه من اینترفیس رو متناسب با نیازی که بعدا به وجود میاد تغییراتی بدم کم ترین کاری که باید بکنم اینه که کلاس مربوطه رو دست کاری کنم.
این که زمان بیشتری میگیره...

البته منکر reusable شدن و ... این مدل نیستم...

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۵/۳۰ ۱۴:۱۸:۱۳

کیوان نیری یک دمو در مورد ASP.Net MVC درست کرده و روش متداول و روش جدید را در ابتدای این دمو با هم مقایسه کرده (حداقل برای MVC الان یک فریم ورک خوب هست).
http://nayyeri.net/files/media/file/Talks/ASPNETMVC10Presentation.pptx

مزایا و معایب هر کدام را توضیح داده که بد نیست یک نگاهی بیندازید.

نویسنده: افشار محبی
تاریخ: ۱۳۸۸/۰۵/۳۰ ۱۶:۵۴:۳۶

MVP به نوعی با SOA (معماری سرویس‌گرا) هم شبیه است

نویسنده: ahmad
تاریخ: ۱۳۸۹/۰۴/۱۸ ۰۰:۰۲:۳۳

سلام
بالاخره در Windows Application از کدام روش استفاده کنیم؟ MVP یا n-tier ؟

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۴/۱۸ ۰۱:۱۰:۳۶

شما در یک سیستم n-tier می‌تونید از MVP هم استفاده کنید.

نویسنده: Mohsen
تاریخ: ۱۳۸۹/۰۶/۱۸ ۰۴:۳۵:۲۳

واقعا این چند لایه آدم رو کلافه می کنه
10 جا یک تیکه کد رو با کمی تغییر باید بنویسی

MVC رو تست کردم عالی بود

MVP رو هنوز تست نکردم
برای MVP در دات نت فریمورک وبی داریم مثل MVC

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۶/۱۸ ۱۰:۱۰:۲۸

Better Web Forms with the MVP Pattern

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۹/۲۸ ۰۰:۳۷:۵۰

Model View Presenter Pattern Implementation in ASP.NET

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۹/۲۸ ۰۰:۴۰:۲۳

ASP.Net MVP Framework

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۹/۲۸ ۰۱:۰۷:۳۵

Web Forms MVP

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۱/۱۰ ۱۷:۲۸:۰۹

یک مثال جالب دیگر در این مورد: (+)

نویسنده: shahin kiassat
تاریخ: ۱۳۹۰/۰۱/۱۱ ۲۳:۱۴:۰۸

سلام.
آقای نصیری هنگامی که می خواستم این آدرس رو باز کنم صحفه منتظر پاسخ از بلاگر می مونه :
http://www.dotnettips.info/2009/08/mvp.html#disqus_thread

در مورد این لینک جدید ام وی پی من برای دانلود فریمورک
webformsmvp
به این صحفه رسیدم
http://nuget.org/List/Packages/WebFormsMvp
اما با وجود ثبت نام در این سایت موفق به دانلود نمیشم.
یعنی هیچ جایی برای دانلود پیدا نکردم.
در سایت پروژه در کد پلکس هم چیزی برای دانلود نبود.
اگر فرصت داشتید راهنمایی کنید.

ممنون
پ ن : قبلا می شد کامنت راست به چپ نوشت الان انگار این امکان وجود نداره؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۱/۱۱ ۲۳:۵۵:۰۶

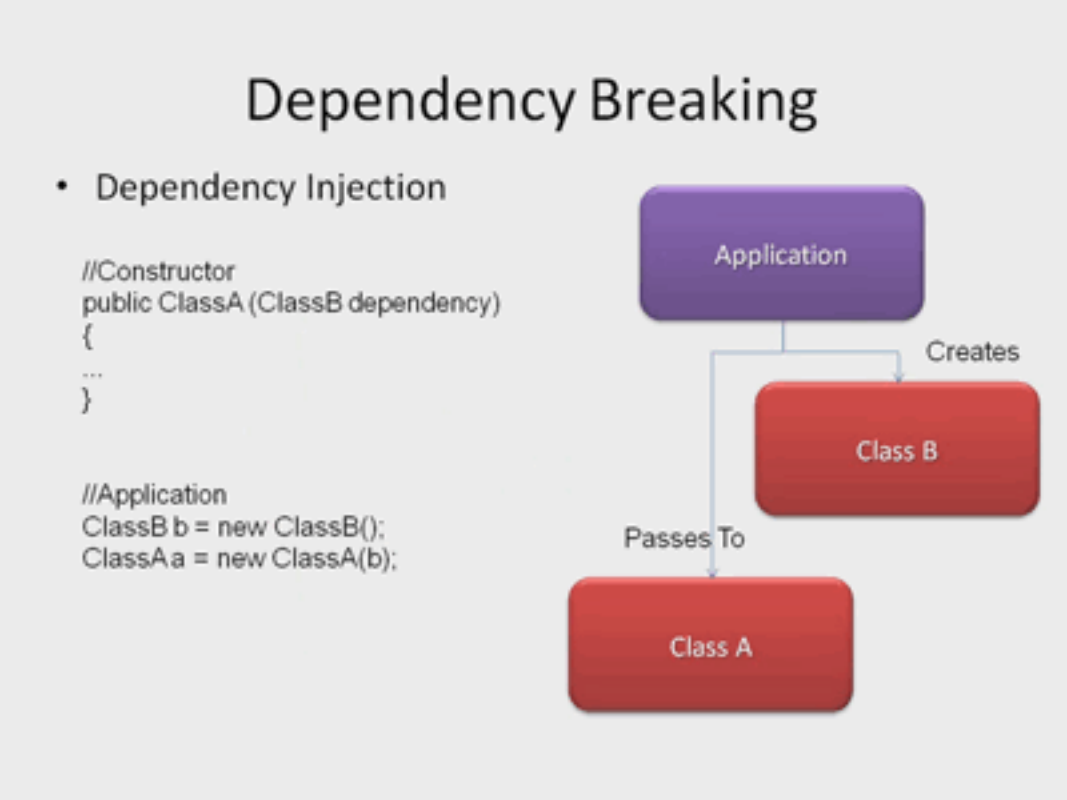
در مورد آدرس مشکلی نبود الان حالا شاید در اون لحظه مشکل ارتباطی وجود داشته

این پکیج رو برای نوگت کامپایل کردن ولی اگر علاقمند بودید می‌شود سورس را از آدرس زیر دریافت و سپس خودتون کامپایل کنید
http://webformsmvp.codeplex.com/SourceControl/list/changesets

این راست به چپ فعلا در دیسکاس نیست یا من ندیدم حالا تا بعد

عنوان: **در هم تنیدگی کدهای خود را کمتر کنید**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۶/۰۶ ۱۹:۱۶:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

مطلب " آشنایی با الگوی MVP " مقدمه‌ی کوتاهی بود بر یکی از روش‌هایی که توسط آن می‌توان گره خوردگی کدهای خود را کمتر، نگهداری طولانی مدت و اعمال تغییرات بعدی به آن‌ها را ساده‌تر کرده و همچنین امکان استفاده مجدد از کدهای موجود را فراهم آورد. در همین ارتباط ویدیویی تحت عنوان Decoupling Your Code, By Example را می‌توانید از آدرس زیر دریافت کنید:



دریافت (90Mb, 44mins)

ماخذ

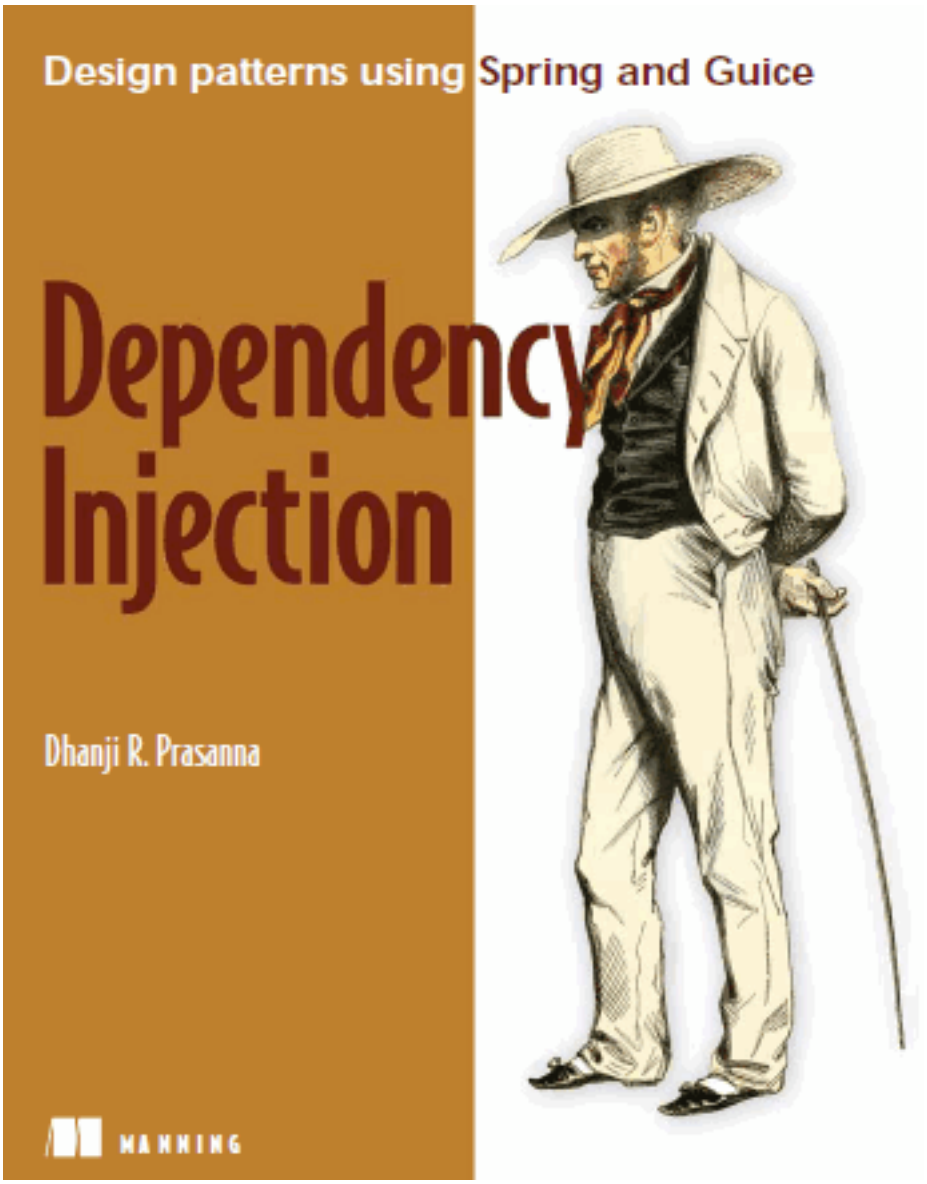
نظرات خوانندگان

نویسنده: Anonymous
تاریخ: ۱۳۸۸/۰۶/۰۷ ۰۶:۴۷:۳۲

ممنون از ارسال فیلم آموزشی. حتما میبنیمش این دو سه روزه . منم یه تاپیک در همین راستا! زده بودم توی دات نت سورس.

عنوان: **Dependency Injection**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۶/۱۴ ۱۹:۱۱:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

در ادامه مباحث بهتر کد بنویسیم و الگوهایی که در این رابطه معرفی شدند، اخیرا کتابی از انتشارات manning منتشر شده تحت عنوان Dependency Injection . هر چند به ظاهر این کتاب برای جاوا کارها تهیه شده اما قسمت عمده‌ای از آن برای سایر زبان‌های برنامه نویسی دیگر نیز قابل استفاده است.



DESCRIPTION

In object-oriented programming, a central program normally controls other objects in a module, library, or framework. With dependency injection, this pattern is inverted—a reference to a service is placed directly into

the object which eases testing and modularity. Spring or Google Guice use dependency injection so you can focus on your core application and let the framework handle infrastructural concerns.

Dependency Injection explores the DI idiom in fine detail, with numerous practical examples that show you the payoffs. You'll apply key techniques in Spring and Guice and learn important pitfalls, corner-cases, and design patterns. Readers need a working knowledge of Java but no prior experience with DI is assumed.

:WHAT'S INSIDE

- ◊ How to apply it (Understand it first!)
- ◊ Design patterns and nuances
- ◊ Spring, Google Guice, PicoContainer, and more
- ◊ How to integrate DI with Java frameworks

نظرات خوانندگان

نویسنده: Novin
تاریخ: ۱۳۸۸/۰۶/۱۵ ۰۰:۴۳:۵۹

منظورتون کتاب کاغذی است؟ کدام کتاب فروشی؟

نویسنده: Anonymous
تاریخ: ۱۳۸۸/۰۶/۱۵ ۰۶:۵۵:۵۶

دمت گرم استاد وحید.
آره والا . منم حالا که 26 سالم شده فهمیدم از روز اول اشتباه انتخاب کرده بودم . هرچی مفاهیم حرفه ای بوده توی جاوا بوده .
نه اون دلفی لعنت الله علیه و ویبی . حالا مجبوریم به سی شارپ قناعت کنیم . دیگه عمر جاوای ما سر اومده

نویسنده: مهدی یوسفی
تاریخ: ۱۳۸۸/۰۶/۱۵ ۰۸:۱۷:۵۸

قیمت کتابهای لاتین واقعا زیاد است مثلا Asp.net C#3.5 در حدود 60 هزارتومان

من که فقط نسخه های دانلودی را ترجیح می دهم
وب سایت جدید PersianDevelopers هم یک بخش برای دانلود قرار داده که لیست آخرین Ebook های برنامه نویسی رو داره.

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۶/۱۵ ۱۳:۰۴:۴۴

@ناشناس
"دلفی لعنت الله" نیست، مرحومه، رحمت اله علیه است ...
دلفی همانند کودک باهوشی بود که از زمان خودش بسیار جلوتر بود اما زود مرحوم شد.

نویسنده: Mohammad Shams Javi
تاریخ: ۱۳۸۸/۰۶/۲۱ ۱۹:۳۴:۳۸

سلام.

خیلی جالب گفتی که، "مرحومه، رحمت اله علیه است"، با این حساب بنده و خیلی از اعضای دیگر سایت برنامه‌نویس، در تالار مربوطه، هنوز بر سر قبر این مرحومه در حال زاری و مویه هستیم. :)

جالب بود - تشکر

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۶/۲۱ ۱۹:۵۳:۲۶

مشکلی که ما داریم دید ادغام وزارت ICT و وزارت راه و برداشت‌هایی در رده‌های بالا در این حد و اندازه است که نهایتا منجر به عدم وجود صنعت برنامه نویسی به شکلی که در کشورهای دیگر هست شده است. با این اوصاف وقتی برنامه‌ها در حد چند سفارش کوچک خلاصه می‌شود یا عموما تک کاربره یا یکی دو کاربره هستند، شاید زیاد تفاوتی نکند که ابزار شما VB6 باشد یا دلفی یا دات نت (همچنین بحث پشتیبانی سیستم‌های قدیمی هم مطرح است).
اما زمانیکه تعداد کاربران شما بالای 200 نفر همزمان بودند و در یک شرکت باید این‌ها رو جمع و جور و پشتیبانی می‌کردید، استفاده از دلفی و دید برنامه نویسی دسکتاپ فقط در حد یک شوخی قابل طرح بود (فقط یکبار این تصور را بکنید که برنامه شما باید در طی روز حداقل سه بار بر اساس درخواست‌های رسیده به روز شود. اگر تونستید ادمینی رو پیدا کنید که 200 تا کامپیوتر

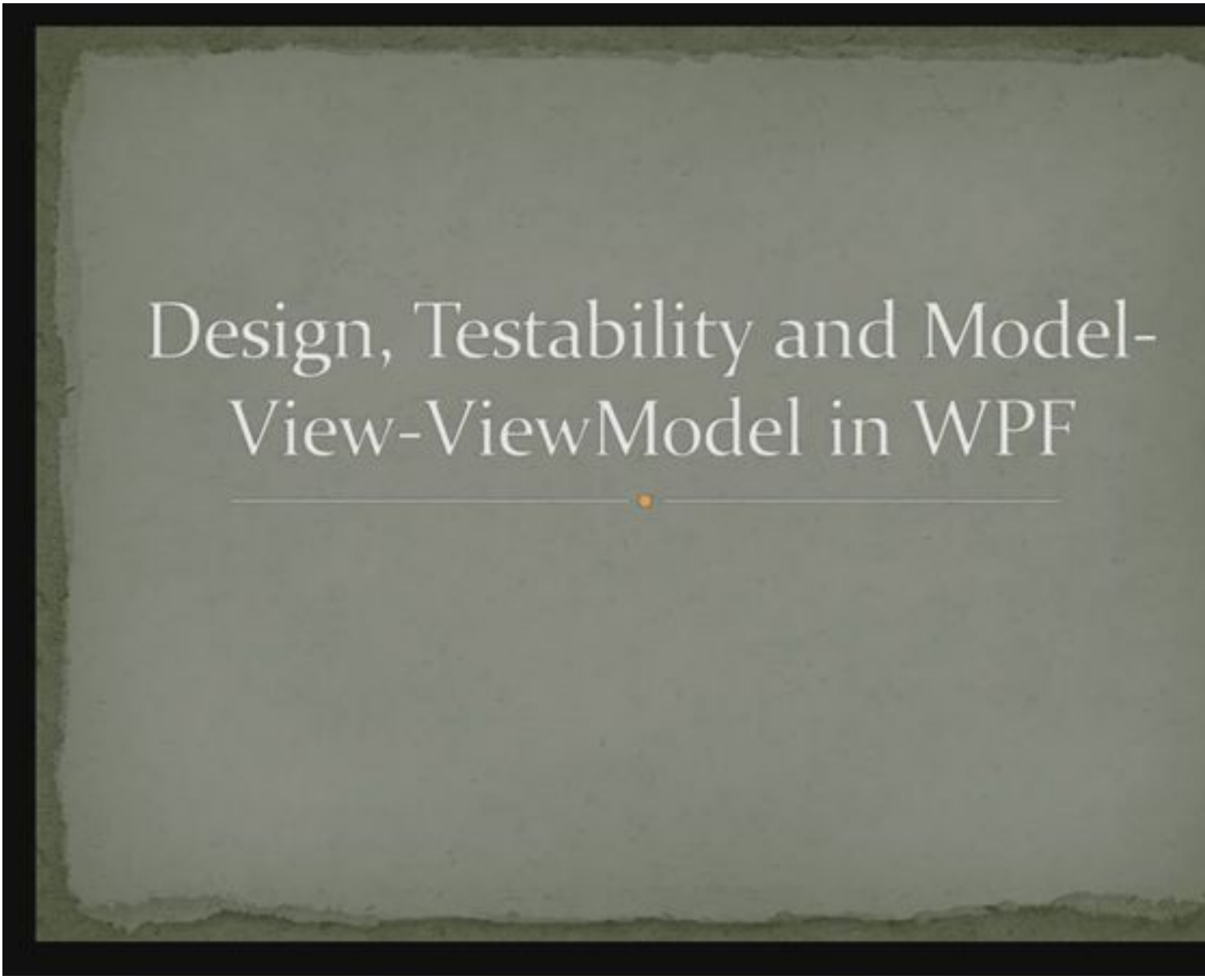
رو برای شما روزی سه بار به روز کند به من خبر بدید)

عنوان: **آشنایی با الگوی MVVM**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۹/۰۴ ۲۱:۱۷:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

حدود یک سال قبل الگوی MVVM زیاد معروف نبود (Model-View-ViewModel pattern). اما در 6 ماه اخیر، این الگو به یک متدولوژی جدی توسعه برنامه‌های WPF و سیلورلایت تبدیل شده. نمی‌شود به یک وبلاگ خوب WPF سر زد و خبری از این روش نباشد. حتی فریم ورک‌هایی هم برای آن طراحی شده که لیست آن‌ها را در این مقاله می‌توانید مشاهده نمائید.

مزایای این الگو چیست؟
جدا سازی Model و View
تولید کدهایی با قابلیت تست بالا
فایل‌های code-behind ایی با حداقل کد
و ...

اگر علاقمند به آشنایی با این الگوی طراحی باشید ویدیوی آموزشی زیر در طی یک ساعت و نیم به توضیح این مطلب پرداخته است.



دریافت

ماخذ

عنوان: آشنایی با الگوی IOC یا Inversion of Control (واگذاری مسئولیت)
نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۹/۲۵ ۱۱:۴۷:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

کلاس Kid را با تعریف زیر در نظر بگیرید. هدف از آن نگهداری اطلاعات فرزندان یک شخص خاص می‌باشد:

```
namespace IOCBeginnerGuide
{
  class Kid
  {
      private int _age;
      private string _name;

      public Kid(int age, string name)
      {
          _age = age;
          _name = name;
      }

      public override string ToString()
      {
          return "KID's Age: " + _age + ", Kid's Name: " + _name;
      }
  }
}
```

اکنون کلاس والد را با توجه به اینکه در حین ایجاد این شیء، فرزندان او نیز باید ایجاد شوند؛ در نظر بگیرید:

```
using System;

namespace IOCBeginnerGuide
{
  class Parent
  {
      private int _age;
      private string _name;
      private Kid _obj;

      public Parent(int personAge, string personName, int kidsAge, string kidsName)
      {
          _obj = new Kid(kidsAge, kidsName);
          _age = personAge;
          _name = personName;
      }

      public override string ToString()
      {
          Console.WriteLine(_obj);
          return "ParentAge: " + _age + ", ParentName: " + _name;
      }
  }
}
```

و نهایتا مثالی از استفاده از آن توسط یک کلاینت:

```
using System;

namespace IOCBeginnerGuide
{
  class Program
  {
      static void Main(string[] args)
      {
          Parent p = new Parent(35, "Dev", 6, "Len");
```

```
            Console.WriteLine(p);

            Console.ReadKey();
            Console.WriteLine("Press a key...");
        }
    }
}
```

که خروجی برنامه در این حالت مساوی سطرهای زیر می‌باشد:

```
KID's Age: 6, Kid's Name: Len
ParentAge: 35, ParentName: Dev
```

مثال فوق نمونه‌ای از الگوی طراحی ترکیب یا composition می‌باشد که به آن Object Dependency یا Object Coupling نیز گفته می‌شود. در این حالت ایجاد شیء والد وابسته است به ایجاد شیء فرزند.

مشکلات این روش:
-1 با توجه به وابستگی شدید والد به فرزند، اگر نمونه سازی از شیء فرزند در سازنده‌ی کلاس والد با موفقیت روبرو نشود، ایجاد نمونه‌ی والد با شکست مواجه خواهد شد.
-2 با از بین رفتن شیء والد، فرزندان او نیز از بین خواهند رفت.
-3 هر تغییری در کلاس فرزند، نیاز به تغییر در کلاس والد نیز دارد (اصطلاحا به آن Dangling Reference هم گفته می‌شود. این کلاس آویزان آن کلاس است!).

چگونه این مشکلات را برطرف کنیم؟
بهتر است کار وهله سازی از کلاس Kid به یک شیء، متد یا حتی فریم ورک دیگری واگذار شود. به این واگذاری مسئولیت، delegation و یا IOC - inversion of control نیز گفته می‌شود.

بنابراین IOC می‌گوید که:
-1 کلاس اصلی (یا همان Parent) نباید به صورت مستقیم وابسته به کلاس‌های دیگر باشد.
-2 رابطه‌ی بین کلاس‌ها باید بر مبنای تعریف کلاس‌های abstract باشد (و یا استفاده از interface ها).

تزریق وابستگی یا Dependency injection
برای پیاده سازی IOC از روش تزریق وابستگی یا dependency injection استفاده می‌شود که می‌تواند بر اساس constructor injection ، setter injection و یا interface-based injection باشد و به صورت خلاصه پیاده سازی یک شیء را از مرحله‌ی ساخت وهله‌ای از آن مجزا و ایزوله می‌سازد.

مزایای تزریق وابستگی‌ها:
-1 گره خوردگی اشیاء را حذف می‌کند.
-2 اشیاء و برنامه را انعطاف پذیرتر کرده و اعمال تغییرات به آن‌ها ساده‌تر می‌شود.

روش‌های متفاوت تزریق وابستگی به شرح زیر هستند:

تزریق سازنده یا constructor injection :
در این روش ارجاعی از شیء مورد استفاده، توسط سازنده‌ی کلاس استفاده کننده از آن دریافت می‌شود. برای نمونه در مثال فوق از آنجائیکه کلاس والد به کلاس فرزندان وابسته است، یک ارجاع از شیء Kid به سازنده‌ی کلاس Parent باید ارسال شود.
اکنون بر این اساس تعاریف، کلاس‌های ما به شکل زیر تغییر خواهند کرد:

```
//IBuisnessLogic.cs
namespace IOCBeginnerGuide
{
  public interface IBuisnessLogic
  {
  }
}
```

```
//Kid.cs
namespace IOCBeginnerGuide
{
  class Kid : IBuisnessLogic
  {
      private int _age;
      private string _name;

      public Kid(int age, string name)
      {
          _age = age;
          _name = name;
      }

      public override string ToString()
      {
          return "KID's Age: " + _age + ", Kid's Name: " + _name;
      }
  }
}
```

```
//Parent.cs
using System;

namespace IOCBeginnerGuide
{
  class Parent
  {
      private int _age;
      private string _name;
      private IBuisnessLogic _refKids;

      public Parent(int personAge, string personName, IBuisnessLogic obj)
      {
          _age = personAge;
          _name = personName;
          _refKids = obj;
      }

      public override string ToString()
      {
          Console.WriteLine(_refKids);
          return "ParentAge: " + _age + ", ParentName: " + _name;
      }
  }
}
```

```
//CIOC.cs
using System;

namespace IOCBeginnerGuide
{
  class CIOC
  {
      Parent _p;

      public void FactoryMethod()
      {
          IBuisnessLogic objKid = new Kid(12, "Ren");
          _p = new Parent(42, "David", objKid);
      }

      public override string ToString()
```

```
        {
            Console.WriteLine(_p);
            return "Displaying using Constructor Injection";
        }
    }
}
```

```
//Program.cs
using System;

namespace IOCBeginnerGuide
{
  class Program
  {
      static void Main(string[] args)
      {
          CIOC obj = new CIOC();
          obj.FactoryMethod();
          Console.WriteLine(obj);

          Console.ReadKey();
          Console.WriteLine("Press a key...");
      }
  }
}
```

توضیحات:
ابتدا اینترفیس IBuisnessLogic ایجاد خواهد شد. تنها متدهای این اینترفیس در اختیار کلاس Parent قرار خواهند گرفت.
از آنجائیکه کلاس Kid توسط کلاس Parent استفاده خواهد شد، نیاز است تا این کلاس نیز اینترفیس IBuisnessLogic را پیاده سازی کند.
اکنون سازنده‌ی کلاس Parent بجای ارجاع مستقیم به شیء Kid ، از طریق اینترفیس IBuisnessLogic با آن ارتباط برقرار خواهد کرد.
در کلاس CIOC کار پیاده سازی واگذاری مسئولیت وهله سازی از اشیاء مورد نظر صورت گرفته است. این وهله سازی در متدی به نام Factory انجام خواهد شد.
و در نهایت کلاینت ما تنها با کلاس IOC سرکار دارد.

معایب این روش:
- در این حالت کلاس business logic، نمی‌تواند دارای سازنده‌ی پیش فرض باشد.
- هنگامیکه وهله‌ای از کلاس ایجاد شد دیگر نمی‌توان وابستگی‌ها را تغییر داد (چون از سازنده‌ی کلاس جهت ارسال مقادیر مورد نظر استفاده شده است).

تزریق تنظیم کننده یا Setter injection
این روش از خاصیت‌ها جهت تزریق وابستگی‌ها بجای تزریق آن‌ها به سازنده‌ی کلاس استفاده می‌کند. در این حالت کلاس Parent می‌تواند دارای سازنده‌ی پیش فرض نیز باشد.

مزایای این روش:
- از روش تزریق سازنده بسیار انعطاف پذیرتر است.
- در این حالت بدون ایجاد وهله‌ای می‌توان وابستگی اشیاء را تغییر داد (چون سر و کار آن با سازنده‌ی کلاس نیست).
- بدون نیاز به تغییری در سازنده‌ی یک کلاس می‌توان وابستگی اشیاء را تغییر داد.
- تنظیم کننده‌ها دارای نامی با معناتر و با مفهوم‌تر از سازنده‌ی یک کلاس می‌باشند.

نحوه‌ی پیاده سازی آن:
در اینجا مراحل ساخت Interface و همچنین کلاس Kid با روش قبل تفاوتی ندارند. همچنین کلاینت نهایی استفاده کننده از IOC

نیز مانند روش قبل است. تنها کلاس‌های IOC و Parent باید اندکی تغییر کنند:

```
//Parent.cs
using System;

namespace IOCBeginnerGuide
{
  class Parent
  {
      private int _age;
      private string _name;

      public Parent(int personAge, string personName)
      {
          _age = personAge;
          _name = personName;
      }

      public IBuisnessLogic RefKID {set; get;}

      public override string ToString()
      {
          Console.WriteLine(RefKID);
          return "ParentAge: " + _age + ", ParentName: " + _name;
      }
  }
}
```

```
//CIOC.cs
using System;

namespace IOCBeginnerGuide
{
  class CIOC
  {
      Parent _p;

      public void FactoryMethod()
      {
          IBuisnessLogic objKid = new Kid(12, "Ren");
          _p = new Parent(42, "David");
          _p.RefKID = objKid;
      }

      public override string ToString()
      {
          Console.WriteLine(_p);
          return "Displaying using Setter Injection";
      }
  }
}
```

همانطور که ملاحظه می‌کنید در این روش یک خاصیت جدید به نام RefKID به کلاس Parent اضافه شده است که از هر لحاظ نسبت به روش تزریق سازنده با مفهوم‌تر و خود توضیح دهنده‌تر است. سپس کلاس IOC جهت استفاده از این خاصیت اندکی تغییر کرده است.

ماخذ

نظرات خوانندگان

نویسنده: gg
تاریخ: ۱۳۸۸/۱۲/۰۷ ۱۹:۲۹:۵۶

خوب بود . موضوع پروژه منم همین است . خوشحال میشم بازم در این مورد مطلب بنوسید.

عنوان: **تزریق وابستگی (dependency injection) به زبان ساده**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۹/۲۷ ۱۸:۴۵:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

این مطلب در ادامه‌ی " آشنایی با الگوی IOC یا Inversion of Control (واگذاری مسئولیت) " می‌باشد که هر از چندگاهی یک قسمت جدید و یا کاملتر از آن ارائه خواهد شد.

==========

به صورت خلاصه ترزیق وابستگی و یا dependency injection ، الگویی است جهت تزریق وابستگی‌های خارجی یک کلاس به آن، بجای استفاده مستقیم از آن‌ها در درون کلاس.
برای مثال شخصی را در نظر بگیرید که قصد خرید دارد. این شخص می‌تواند به سادگی با کمک یک خودرو خود را به اولین محل خرید مورد نظر برساند. حال تصور کنید که 7 نفر عضو یک گروه، با هم قصد خرید دارند. خوشبختانه چون تمام خودروها یک اینترفیس مشخصی داشته و کار کردن با آن‌ها تقریبا شبیه به یکدیگر است، حتی اگر از یک ون هم جهت رسیدن به مقصد استفاده شود، امکان استفاده و راندن آن همانند سایر خودروها می‌باشد و این دقیقا همان مطلبی است که هدف غایی الگوی تزریق وابستگی‌ها است. بجای این‌که همیشه محدود به یک خودرو برای استفاده باشیم، بنابر شرایط، خودروی متناسبی را نیز می‌توان مورد استفاده قرار داد.
در دنیای نرم افزار، وابستگی کلاس Driver ، کلاس Car است. اگر موارد ذکر شده را بدون استفاده از تزریق وابستگی‌ها پیاده سازی کنیم به کلاس‌های زیر خواهیم رسید:

```
//Person.cs
namespace DependencyInjectionForDummies
{
   class Person
   {
       public string Name { get; set; }
   }
}
```

```
//Car.cs
using System;
using System.Collections.Generic;

namespace DependencyInjectionForDummies
{
   class Car
   {
       List<Person> _passengers = new List<Person>();

       public void AddPassenger(Person p)
       {
           _passengers.Add(p);
           Console.WriteLine("{0} added!", p.Name);
       }

       public void Drive()
       {
           foreach (var passenger in _passengers)
               Console.WriteLine("Driving {0} ...!", passenger.Name);
       }
   }
}
```

```
//Driver.cs
using System.Collections.Generic;

namespace DependencyInjectionForDummies
{
```

```
   class Driver
   {
       private Car _myCar = new Car();

       public void DriveToMarket(IList<Person> passengers)
       {
           foreach (var passenger in passengers)
               _myCar.AddPassenger(passenger);

           _myCar.Drive();
       }
   }
}
```

```
//Program.cs
using System.Collections.Generic;
using System;

namespace DependencyInjectionForDummies
{
   class Program
   {
       static void Main(string[] args)
       {
           new Driver().DriveToMarket(
               new List<Person>
               {
                   new Person{ Name="Ali" },
                   new Person{ Name="Vahid" }
               });

           Console.WriteLine("Press a key ...");
           Console.ReadKey();
       }
   }
}
```

توضیحات:
کلاس شخص (Person) جهت تعریف مسافرین، اضافه شده؛ سپس کلاس خودرو (Car) که اشخاص را می‌توان به آن اضافه کرده و سپس به مقصد رساند، تعریف گردیده است. همچنین کلاس راننده (Driver) که بر اساس لیست مسافرین، آن‌ها را به خودروی خاص ذکر شده هدایت کرده و سپس آن‌ها را با کمک کلاس خودرو به مقصد می‌رساند؛ نیز تعریف شده است. در پایان هم یک کلاینت ساده جهت استفاده از این کلاس‌ها ذکر شده است.
همانطور که ملاحظه می‌کنید کلاس راننده به کلاس خودرو گره خورده است و این راننده همیشه تنها از یک نوع خودروی مشخص می‌تواند استفاده کند و اگر روزی قرار شد از یک ون کمک گرفته شود، این کلاس باید بازنویسی شود.

خوب! اکنون اگر این کلاس‌ها را بر اساس الگوی تزریق وابستگی‌ها (روش تزریق در سازنده که در قسمت قبل بحث شد) بازنویسی کنیم به کلاس‌های زیر خواهیم رسید:

```
//ICar.cs
using System;

namespace DependencyInjectionForDummies
{
   interface ICar
   {
       void AddPassenger(Person p);
       void Drive();
   }
}
```

```
//Car.cs
using System;
using System.Collections.Generic;
```

```
namespace DependencyInjectionForDummies
{
   class Car : ICar
   {
       همانند قسمت قبل//
   }
}
```

```
//Van.cs
using System;
using System.Collections.Generic;

namespace DependencyInjectionForDummies
{
   class Van : ICar
   {
       List<Person> _passengers = new List<Person>();

       public void AddPassenger(Person p)
       {
           _passengers.Add(p);
           Console.WriteLine("{0} added!", p.Name);
       }

       public void Drive()
       {
           foreach (var passenger in _passengers)
               Console.WriteLine("Driving {0} ...!", passenger.Name);
       }
   }
}
```

```
//Driver.cs
using System.Collections.Generic;

namespace DependencyInjectionForDummies
{
   class Driver
   {
       private ICar _myCar;

       public Driver(ICar myCar)
       {
           _myCar = myCar;
       }

       public void DriveToMarket(IList<Person> passengers)
       {
           foreach (var passenger in passengers)
               _myCar.AddPassenger(passenger);

           _myCar.Drive();
       }
   }
}
```

```
//Program.cs
using System.Collections.Generic;
using System;

namespace DependencyInjectionForDummies
{
   class Program
   {
       static void Main(string[] args)
       {
           Driver driver = new Driver(new Van());
           driver.DriveToMarket(
               new List<Person>
               {
                   new Person{ Name="Ali" },
```

```
                    new Person{ Name="Vahid" }
                });

            Console.WriteLine("Press a key ...");
            Console.ReadKey();
        }
    }
}
```

توضیحات:

در اینجا یک اینترفیس جدید به نام ICar اضافه شده است و بر اساس آن می‌توان خودروهای مختلفی را با نحوه‌ی بکارگیری یکسان اما با جزئیات پیاده سازی متفاوت تعریف کرد. برای مثال در ادامه، یک کلاس ون با پیاده سازی این اینترفیس تشکیل شده است. سپس کلاس راننده‌ی ما بر اساس ترزیق این اینترفیس در سازنده‌ی آن بازنویسی شده است. اکنون این کلاس دیگر نمی‌داند که دقیقا چه خودرویی را باید مورد استفاده قرار دهد و از وابستگی مستقیم به نوعی خاص از آن‌ها رها شده است؛ اما می‌داند که تمام خودروها، اینترفیس مشخص و یکسانی دارند. به تمام آن‌ها می‌توان مسافرانی را افزود و سپس به مقصد رساند. در پایان نیز یک راننده جدید بر اساس خودروی ون تعریف شده، سپس یک سری مسافر نیز تعریف گردیده و نهایتا متد DriveToMarket فراخوانی شده است.

به این صورت به یک سری کلاس اصطلاحا loosely coupled رسیده‌ایم. دیگر راننده‌ی ما وابسته‌ی به یک خودروی خاص نیست و هر زمانی که لازم بود می‌توان خودروی مورد استفاده‌ی او را تغییر داد بدون اینکه کلاس راننده را بازنویسی کنیم.

یکی دیگر از مزایای تزریق وابستگی‌ها ساده سازی unit testing کلاس‌های برنامه توسط mocking frameworks است. به این صورت توسط این نوع فریم‌ورک‌ها می‌توان رفتار یک خودرو را تقلید کرد بجای اینکه واقعا با تمام ریز جرئیات آن‌ها بخواهیم سروکار داشته باشیم (وابستگی‌ها را به صورت مستقل می‌توان آزمایش کرد).

نظرات خوانندگان

نویسنده: MDP
تاریخ: ۱۳۸۸/۰۹/۲۸ ۰۹:۴۰:۵۲

سلام ، خسته نباشید.خیلی جالب بود.

جناب نصیری من با یه قسمت این تزریق وابستگی و کلا دیزان پترن های مختلف مشکل دارم.

توی بیشتر دیزاین پترن ها از جمله همین تزریق وابستگی و یا ریپتازیتوری از اینتر فیس های استفاده میشه.

چه طوری به جای خود آبجکت کلاس با اینترفیسش کار میکنن.اینتر فیس که هیچ امپلیمنتی از کلاس نداره.

اصلا حکمت این کار چیه؟

خوش حال میشم یک آموزش در زمینه Repository بنویسید.چون توی ASP.NET MVC کاربرد زیادی می تونه داشته باشه.

ممنون (:

نویسنده: وحید نصیری
تاریخ: ۱۳۸۸/۰۹/۲۸ ۱۰:۱۴:۴۷

سلام،
در مورد repository قبلا مطلب نوشتم:
http://www.dotnettips.info/2009/10/nhibernate_17.html

نویسنده: Rahman Mohammadi
تاریخ: ۱۳۸۸/۱۰/۰۲ ۰۹:۳۶:۲۲

سلام
خسته نباشید ، مثل هیمشه مطالبتون عالی بود

عنوان: **ویدیوهای آموزشی MVVM**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۱/۱۳ ۲۲:۱۴:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

یک سری ویدیوی رایگان آموزشی MVVM از مایکروسافت و همچنین شرکت Infragistics در دسترس هستند که جهت سهولت، لیست آن‌ها را ادامه می‌توانید مشاهده نمائید:

Understanding the Model-View-ViewModel Pattern

Build Your Own MVVM Framework

Implementing the M-V-VM Pattern

Implementing Model-View-ViewModel in Silverlight

Implementing Model-View-ViewModel in WPF

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۱/۱۵ ۰۹:۴۶:۵۰

یک مورد دیگر

Rocky Lhotka on the MVVM Pattern in CSLA .NET 3.8

http://www.dnrtv.com/default.aspx?showNum=161

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۱/۱۶ ۲۲:۵۲:۳۸

دو مورد دیگر

Silverlight TV 13: MVVM Light Toolkit

/http://channel9.msdn.com/shows/SilverlightTV/Silverlight-TV-13-MVVM-Light-Toolkit

MVVM, a WPF UI Design Pattern

/http://channel9.msdn.com/shows/Continuum/MVVM

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۱/۲۲ ۲۱:۴۳:۴۴

دو مورد دیگر:

Advanced Topics for Building Large-Scale Applications with Microsoft Silverlight

http://microsoftpdc.com/Sessions/CL22

Developing Testable Silverlight Applications

http://microsoftpdc.com/Sessions/CL32

عنوان: **انجام پی در پی اعمال Async به کمک Iterators - قسمت اول**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۴/۱۰ ۲۰:۵۶:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

تقریبا تمام اعمال کار با شبکه در Silverlight از مدل asynchronous programming پیروی می‌کنند؛ از فراخوانی یک متد وب سرویس تا دریافت اطلاعات از وب و غیره. اگر در سایر فناوری‌های موجود در دات نت فریم ورک برای مثال جهت کار با یک وب سرویس هر دو متد همزمان و غیرهمزمان در اختیار برنامه نویس هستند اما اینجا خیر. اینجا فقط روش‌های غیرهمزمان مرسوم هستند و بس. خیلی هم خوب. یک چارچوب کاری خوب باید روش استفاده‌ی صحیح از کتابخانه‌های موجود را نیز ترویج کند و این مورد حداقل در Silverlight اتفاق افتاده است.
برای مثال فراخوانی‌های زیر را در نظر بگیرید:

```
private int n1, n2;

private void FirstCall()
{
  Service.GetRandomNumber(10, SecondCall);
}

private void SecondCall(int number)
{
  n1 = number;
  Service.GetRandomNumber(n1, ThirdCall);
}

private void ThirdCall(int number)
{
  n2 = number;
  // etc
}
```

عموما در اعمال Async پس از پایان عملیات در تردی دیگر، یک متد فراخوانی می‌گردد که به آن callback delegate نیز گفته می‌شود. برای مثال توسط این سه متد قصد داریم اطلاعاتی را از یک وب سرویس دریافت و استفاده کنیم. ابتدا FirstCall فراخوانی می‌شود. پس از پایان کار آن به صورت خودکار متد SecondCall فراخوانی شده و این متد نیز یک عملیات Async دیگر را شروع کرده و الی آخر. در نهایت قصد داریم توسط مقادیر بازگشت داده شده منطق خاصی را پیاده سازی کنیم. همانطور که مشاهده می‌کنید این اعمال زیبا نیستند! چقدر خوب می‌شد مانند دوران synchronous programming (!) فراخوانی‌های این متدها به صورت ذیل انجام می‌شد:

```
private void FetchNumbers()
{
 int n1 = Service.GetRandomNumber(10);
 int n2 = Service.GetRandomNumber(n1);
}
```

در برنامه نویسی متداول همیشه عادت داریم که اعمال به صورت A -> B -> C انجام شوند. اما در Async programming ممکن است ابتدا C انجام شود، سپس A و بعد B یا هر حالت دیگری صرفنظر از تقدم و تاخر آن‌ها در حین معرفی متدهای مرتبط در یک قطعه کد. همچنین میزان خوانایی این نوع کدنویسی نیز مطلوب نیست. مانند مثال اول ذکر شده، یک عملیات به ظاهر ساده به چندین متد منقطع تقسیم شده است. البته به کمک lambda expressions مثال اول را به شکل زیر نیز می‌توان در طی یک متد ارائه داد اما اگر تعداد فراخوانی‌ها بیشتر بود چطور؟ همچنین آیا استفاده از عدد n2 بلافاصله پس از عبارت ذکر شده مجاز است؟ آیا عملیات واقعا به پایان رسیده و مقدار مطلوب به آن انتساب داده شده است؟

```
private void FetchNumbers()
{
  int n1, n2;

  Service.GetRandomNumber(10, result =>
                                {
```

```
                                    n1 = result;
                                    Service.GetRandomNumber(n1, secondResult =>
                                                                {
                                                                    n2 = secondResult;
                                                                });
                                });
}
```

به عبارتی می‌خواهیم کل اعمال انجام شده در متد FetchNumbers هنوز Async باشند (ترد اصلی برنامه را قفل نکنند) اما پی در پی انجام شوند تا مدیریت آن‌ها ساده‌تر شوند (هر لحظه دقیقا بدانیم که کجا هستیم) و همچنین کدهای تولیدی نیز خواناتر باشند.
روش استانداری که توسط الگوهای برنامه نویسی برای حل این مساله پیشنهاد می‌شود، استفاده از الگوی coroutines است.
توسط این الگو می‌توان چندین متد Async را در حالت معلق قرار داده و سپس در هر زمانی که نیاز به آن‌ها بود عملیات آن‌ها را از سر گرفت.
دات نت فریم ورک حالت ویژه‌ای از coroutines را توسط Iterators پشتیبانی می‌کند (از C# 2.0 به بعد) که در ابتدا نیاز است از دیدگاه این مساله مروری بر آن‌ها داشته باشیم. مثال بعد یک enumerator را به همراه yield return ارائه داده است:

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

namespace CoroutinesSample
{
  class Program
  {
      static void printAll()
      {
          foreach (int x in integerList())
          {
              Console.WriteLine(x);
          }
      }

      static IEnumerable<int> integerList()
      {
          yield return 1;
          Thread.Sleep(1000);
          yield return 2;
          yield return 3;
      }

      static void Main()
      {
          printAll();
      }
  }
}
```

کامپایلر سی شارپ در عمل یک state machine را برای پیاده سازی این عملیات به صورت خودکار تولید خواهد کرد:

```
private bool MoveNext()
{
  switch (this.<>1__state)
  {
      case 0:
          this.<>1__state = -1;
          this.<>2__current = 1;
          this.<>1__state = 1;
          return true;

      case 1:
          this.<>1__state = -1;
          Thread.Sleep(0x3e8);
          this.<>2__current = 2;
          this.<>1__state = 2;
          return true;

      case 2:
```

```
            this.<>1__state = -1;
            this.<>2__current = 3;
            this.<>1__state = 3;
            return true;

        case 3:
            this.<>1__state = -1;
            break;
    }
    return false;
}
```

در حین استفاده از یک IEnumerator ابتدا در وضعیت شیء Current آن قرار خواهیم داشت و تا زمانیکه متد MoveNext آن فراخوانی نشود هیچ اتفاق دیگری رخ نخواهد داد. هر بار که متد MoveNext این enumerator فرخوانی گردد (برای مثال توسط یک حلقه‌ی foreach) اجرای متد integerList ادامه خواهد یافت تا به yield return بعدی برسیم (سایر اعمال تعریف شده در حالت تعلیق قرار دارند) و همینطور الی آخر.
از همین قابلیت جهت مدیریت اعمال Async پی در پی نیز می‌توان استفاده کرد. State machine فوق تا پایان اولین عملیات تعریف شده صبر می‌کند تا به yield return برسد. سپس با فراخوانی متد MoveNext به عملیات بعدی رهنمون خواهیم شد. به این صورت دیدگاهی پی در پی از یک سلسه عملیات غیرهمزمان حاصل می‌گردد.

خوب ما الان نیاز به یک کلاس داریم که بتواند enumerator ایی از این دست را به صورت خودکار مرحله به مرحله آن هم پس از پایان واقعی عملیات Async قبلی (یا مرحله‌ی قبلی)، اجرا کند. قبل از اختراع چرخ باید متذکر شد که دیگران اینکار را انجام داده‌اند و کتابخانه‌های رایگان و یا سورس بازی برای این منظور موجود است.

ادامه دارد ...

نظرات خوانندگان

نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۸/۰۸ ۱۶:۳۳:۴۸

خبر خوش اینکه انجام امور async در سی شارپ 5 به کمک واژه کلیدی await ، همانند مقصود دو مقاله فوق به سادگی در اختیار و کنترل برنامه نویس‌ها خواهد بود.

نویسنده: ابراهیم بیاگوی
تاریخ: ۱۳۹۱/۰۵/۱۵ ۱۸:۳۴

واقعاً عالی‌است!

عنوان: **انجام پی در پی اعمال Async به کمک Iterators - قسمت دوم**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۴/۱۱ ۰۰:۰۱:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

در قسمت قبل ایده‌ی اصلی و مفاهیم مرتبط با استفاده از Iterators مطرح شد. در این قسمت به یک مثال عملی در این مورد خواهیم پرداخت.

چندین کتابخانه و کلاس جهت مدیریت Coroutines در دات نت تهیه شده که لیست آن‌ها به شرح زیر است:

1) Using C# 2.0 iterators to simplify writing asynchronous code
2) Wintellect's Jeffrey Richter's PowerThreading Library
3) Rob Eisenberg's Build your own MVVM Framework codes

و ...

مورد سوم که توسط خالق اصلی کتابخانه‌ی Caliburn (یکی از فریم ورک‌های مشهور MVVM برای WPF و Silverlight) در کنفرانس MIX 2010 ارائه شد، این روزها در وبلاگ‌های مرتبط بیشتر مورد توجه قرار گرفته و تقریبا به یک روش استاندارد تبدیل شده است. این روش از یک اینترفیس و یک کلاس به شرح زیر تشکیل می‌شود:

```
using System;

namespace SLAsyncTest.Helper
{
   public interface IResult
   {
       void Execute();
       event EventHandler Completed;
   }
}
```

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

namespace SLAsyncTest.Helper
{
   public class ResultEnumerator
   {
       private readonly IEnumerator<IResult> _enumerator;

       public ResultEnumerator(IEnumerable<IResult> children)
       {
           _enumerator = children.GetEnumerator();
       }

       public void Enumerate()
       {
           childCompleted(null, EventArgs.Empty);
       }

       private void childCompleted(object sender, EventArgs args)
       {
           var previous = sender as IResult;

           if (previous != null)
               previous.Completed -= childCompleted;

           if (!_enumerator.MoveNext())
               return;

           var next = _enumerator.Current;
           next.Completed += childCompleted;
           next.Execute();
```

```
        }
    }
}
```

توضیحات:
مطابق توضیحات قسمت قبل، برای مدیریت اعمال همزمان به شکلی پی در پی، نیاز است تا یک IEnumerable را به همراه yield return در پایان هر مرحله از کار ایجاد کنیم. در اینجا این IEnumerable را از نوع اینترفیس IResult تعریف خواهیم کرد. متد Execute آن شامل کدهای عملیات Async خواهند شد و پس از پایان کار رخداد Completed صدا زده می‌شود. به این صورت کلاس ResultEnumerator به سادگی می‌تواند یکی پس از دیگری اعمال Async مورد نظر ما را به صورت متوالی فراخوانی نمائید. با هر بار فراخوانی رخداد Completed، متد MoveNext صدا زده شده و یک مرحله به جلو خواهیم رفت.
برای مثال کدهای ساده WCF Service زیر را در نظر بگیرید.

```
using System.ServiceModel;
using System.ServiceModel.Activation;
using System.Threading;

namespace SLAsyncTest.Web
{
    [ServiceContract(Namespace = "")]
    [AspNetCompatibilityRequirements(RequirementsMode
        = AspNetCompatibilityRequirementsMode.Allowed)]
    public class TestService
    {
        [OperationContract]
        public int GetNumber(int number)
        {
            Thread.Sleep(2000);//Simulating a log running operation
            return number * 2;
        }
    }
}
```

قصد داریم در طی دو مرحله متوالی این WCF Service را در یک برنامه‌ی Silverlight فراخوانی کنیم. کدهای قسمت فراخوانی این سرویس بر اساس پیاده سازی اینترفیس IResult به صورت زیر درخواهند آمد:

```
using System;
using SLAsyncTest.Helper;

namespace SLAsyncTest.Model
{
    public class GetNumber : IResult
    {
        public int Result { set; get; }
        public bool HasError { set; get; }

        private int _num;
        public GetNumber(int num)
        {
            _num = num;
        }

        #region IResult Members
        public void Execute()
        {
            var srv = new TestServiceReference.TestServiceClient();
            srv.GetNumberCompleted += (sender, e) =>
            {
                if (e.Error == null)
                    Result = e.Result;
                else
                    HasError = true;

                Completed(this, EventArgs.Empty); //run the next IResult
            };
            srv.GetNumberAsync(_num);
        }
```

```
        public event EventHandler Completed;
        #endregion
    }
}
```

در متد Execute کار فراخوانی غیرهمزمان WCF Service به صورتی متداول انجام شده و در پایان متد Completed صدا زده می‌شود. همانطور که توضیح داده شد، این فراخوانی در کلاس ResultEnumerator یاد شده مورد استفاده قرار می‌گیرد.
اکنون قسمت‌های اصلی کدهای View Model برنامه به شکل زیر خواهند بود:

```
private void doFetch(object obj)
        {
            new ResultEnumerator(executeAsyncOps()).Enumerate();
        }

        private IEnumerable<IResult> executeAsyncOps()
        {
            FinalResult = 0;
            IsBusy = true; //Show BusyIndicator

            //Sequential Async Operations
            var asyncOp1 = new GetNumber(10);
            yield return asyncOp1;

            //using the result of the previous step
            if(asyncOp1.HasError)
            {
                IsBusy = false; //Hide BusyIndicator
                yield break;
            }

            var asyncOp2 = new GetNumber(asyncOp1.Result);
            yield return asyncOp2;

            FinalResult = asyncOp2.Result; //Bind it to the UI

            IsBusy = false; //Hide BusyIndicator
        }
```

در اینجا یک IEnumerable از نوع IResult تعریف شده است و در طی دو مرحله‌ی متوالی اما غیرهمزمان کار دریافت اطلاعات از WCF Service صورت می‌گیرد. ابتدا عدد 10 به WCF Service ارسال می‌شود و خروجی 20 خواهد بود. سپس این عدد در مرحله‌ی بعد مجددا به WCF Service ارسال گردیده و حاصل نهایی که عدد 40 می‌باشد در اختیار سیستم Binding قرار خواهد گرفت.
اگر از این روش استفاده نمی‌شد ممکن بود به این جواب برسیم یا خیر. ممکن بود مرحله‌ی دوم ابتدا شروع شود و سپس مرحله‌ی اول رخ دهد. اما با کمک Iterators و yield return به همراه کلاس ResultEnumerator موفق شدیم تا عملیات دوم همزمان را در حالت تعلیق قرار داده و پس از پایان اولین عملیات غیر همزمان، مرحله‌ی بعدی فراخوانی را بر اساس مقدار حاصل شده از WCF Service آغاز کنیم.
این روش برای برنامه نویس‌ها آشناتر است و همان سیستم فراخوانی A->B->C را تداعی می‌کند اما کلیه اعمال غیرهمزمان هستند و ترد اصلی برنامه قفل نخواهد شد.

کدهای کامل این مثال را از اینجا می‌توانید دریافت کنید.

نظرات خوانندگان

نویسنده: Majid325
تاریخ: ۱۳۸۹/۰۴/۱۲ ۱۳:۳۸:۲۱

خیلی عالی بود ، اتفاقا همین مشکل هفته گذشته واسه من به وجود امده بود.

عنوان: **MEF و الگوی Singleton**
نویسنده: وحید نصیری
تاریخ: ۱۳۸۹/۰۴/۱۴ ۱۹:۱۹:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

در مورد معرفی مقدماتی MEF می‌توانید به این مطلب مراجعه کنید و در مورد الگوی Singleton به اینجا .

کاربردهای الگوی Singleton عموما به شرح زیر هستند:
1) فراهم آوردن دسترسی ساده و عمومی به DAL (لایه دسترسی به داده‌ها)
2) دسترسی عمومی به امکانات ثبت وقایع سیستم در برنامه logging -
3) دسترسی عمومی به تنظیمات برنامه
و موارد مشابهی از این دست به صورتیکه تنها یک روش دسترسی به این اطلاعات وجود داشته باشد و تنها یک وهله از این شیء در حافظه قرار گیرد.

با استفاده از امکانات MEF دیگر نیازی به نوشتن کدهای ویژه تولید کلاس‌های Singleton نمی‌باشد زیرا این چارچوب کاری دو نوع روش وهله سازی از اشیاء (PartCreationPolicy) را پشتیبانی می‌کند: Shared و NonShared . حالت Shared دقیقا همان نام دیگر الگوی Singleton است. البته لازم به ذکر است که حالت Shared ، حالت پیش فرض تولید وهله‌ها بوده و نیازی به ذکر صریح آن همانند ویژگی زیر نیست:

```
[PartCreationPolicy(CreationPolicy.Shared)]
```

مثال:
فرض کنید قرار است از کلاس زیر تنها یک وهله بین صفحات یک برنامه‌ی Silverlight توزیع شود. با استفاده از ویژگی Export به MEF اعلام کرده‌ایم که قرار است سرویسی را ارائه دهیم :

```
using System;
using System.ComponentModel.Composition;

namespace SlMefTest
{
   [Export]
   public class WebServiceData
   {
       public int Result { set; get; }

       public WebServiceData()
       {
           var rnd = new Random();
           Result = rnd.Next();
       }
   }

}
```

اکنون برای اثبات اینکه تنها یک وهله از این کلاس در اختیار صفحات مختلف قرار خواهد گرفت، یک User control جدید را به همراه یک دکمه که مقدار Result را نمایش می‌دهد به برنامه اضافه خواهیم کرد. دکمه‌ی دیگری را نیز به همین منظور به صفحه‌ی اصلی برنامه اضافه می‌کنیم.
کدهای صفحه اصلی برنامه (که از یک دکمه و یک Stack panel جهت نمایش محتوای یوزر کنترل تشکیل شده) به شرح بعد هستند:

```
<UserControl x:Class="SlMefTest.MainPage"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

```
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
   mc:Ignorable="d" d:DesignHeight="300" d:DesignWidth="400">
   <StackPanel>
       <Button Content="MainPageButton" Height="23"
               HorizontalAlignment="Left"
               Margin="10,10,0,0" Name="button1"
               VerticalAlignment="Top" Width="98" Click="button1_Click" />
       <StackPanel Name="panel1" Margin="5"/>
   </StackPanel>
</UserControl>
```

```
using System.ComponentModel.Composition;
using System.Windows;

namespace SlMefTest
{
   public partial class MainPage
   {
       [Import]
       public WebServiceData Data { set; get; }

       public MainPage()
       {
           InitializeComponent();
           this.Loaded += mainPageLoaded;
       }

       void mainPageLoaded(object sender, RoutedEventArgs e)
       {
           CompositionInitializer.SatisfyImports(this);
           panel1.Children.Add(new SilverlightControl1());
       }

       private void button1_Click(object sender, RoutedEventArgs e)
       {
           MessageBox.Show(Data.Result.ToString());
       }
   }
}
```

با استفاده از ویژگی Import به MEF اعلام می‌کنیم که به اطلاعاتی از نوع شیء WebServiceData نیاز داریم و توسط متد CompositionInitializer.SatisfyImports کار وهله سازی و پیوند زدن export و import های همانند صورت می‌گیرد. سپس استفاده‌ی مستقیم از Data.Result مجاز بوده و مقدار آن null نخواهد بود.

کدهای User control ساده اضافه شده به شرح زیر هستند:

```
<UserControl x:Class="SlMefTest.SilverlightControl1"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
   xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
   mc:Ignorable="d"
   d:DesignHeight="300" d:DesignWidth="400">

   <Grid x:Name="LayoutRoot" Background="White">
       <Button Content="UserControlButton"
               Height="23"
               HorizontalAlignment="Left"
               Margin="10,10,0,0"
               Name="button1"
               VerticalAlignment="Top"
               Width="125"
               Click="button1_Click" />
   </Grid>
</UserControl>
```

```
using System.ComponentModel.Composition;
```

```
using System.Windows;

namespace SlMefTest
{
   public partial class SilverlightControl1
   {
       [Import]
       public WebServiceData Data { set; get; }

       public SilverlightControl1()
       {
           InitializeComponent();
           this.Loaded += silverlightControl1Loaded;
       }

       void silverlightControl1Loaded(object sender, RoutedEventArgs e)
       {
           CompositionInitializer.SatisfyImports(this);
       }

       private void button1_Click(object sender, RoutedEventArgs e)
       {
           MessageBox.Show(Data.Result.ToString());
       }
   }
}
```

اکنون قبل از شروع برنامه یک break point را در سازنده‌ی کلاس WebServiceData قرار دهید. سپس برنامه را آغاز نمائید. تنها یکبار این سازنده فراخوانی خواهد شد (هر چند در دو کلاس کار Import اطلاعات WebServiceData صورت گرفته است). همچنین با کلیک بر روی دو دکمه‌ای که اکنون در صفحه‌ی اصلی برنامه ظاهر می‌شوند، فقط یک عدد مشابه نمایش داده می‌شود (با توجه به اینکه اطلاعات هر دکمه در یک وهله‌ی جداگانه قرار دارد؛ یکی متعلق است به صفحه‌ی اصلی و دیگری متعلق است به user control اضافه شده).

عنوان: **آشنایی با Fluent interfaces**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۳/۰۴ ۱۳:۲۱:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

تعریف مقدماتی fluent interface در ویکی پدیا به شرح زیر است: (+)

In software engineering, a fluent interface (as first coined by Eric Evans and Martin Fowler) is a way of implementing an object oriented API in a way that aims to provide for more readable code.

به صورت خلاصه هدف آن فراهم آوردن روشی است که بتوان متدها را زنجیر وار فراخوانی کرد و به این ترتیب خوانایی کد نوشته شده را بالا برد. پیاده سازی آن هم شامل دو نکته است:
الف) نوع متد تعریف شده باید مساوی با نام کلاس جاری باشد.
ب) در این حالت خروجی متدهای ما کلمه کلیدی this خواهند بود.

برای مثال:

```
using System;

namespace FluentInt
{
 public class FluentApiTest
 {
   private int _val;

   public FluentApiTest Number(int val)
   {
       _val = val;
       return this;
   }

   public FluentApiTest Abs()
   {
       _val = Math.Abs(_val);
       return this;
   }

   public bool IsEqualTo(int val)
   {
       return val == _val;
   }
 }
}
```

مثالی هم از استفاده‌ی آن به صورت زیر می‌تواند باشد:

```
if (new FluentApiTest().Number(-10).Abs().IsEqualTo(10))
{
 Console.WriteLine("Abs(-10)==10");
}
```

که در آن توانستیم تمام متدها را زنجیر وار و با خوانایی خوبی شبیه به نوشتن جملات انگلیسی در کنار هم قرار دهیم.
خوب! این مطلبی است که همه جا پیدا می‌کنید و مطلب جدیدی هم نیست. اما موردی را که سخت می‌شود یافت این است که طراحی کلاس فوق ایراد دارد. برای مثال شما می‌توانید ترکیب‌های زیر را هم تشکیل دهید و کار می‌کند؛ یا به عبارتی برنامه کامپایل می‌شود و این خوب نیست:

```
if(new FluentApiTest().Abs().Number(-10).IsEqualTo(10)) ...
if (new FluentApiTest().Abs().IsEqualTo(10)) ...
```

می‌شود در کدهای برنامه یک سری throw new exception را هم قرار داد که ... هی! اول باید اون رو فراخوانی کنی بعد این رو! ولی ... این روش هم صحیح نیست. از ابتدای کار نباید بتوان متد بی‌ربطی را در طول این زنجیره مشاهده کرد. اگر قرار نیست استفاده گردد، نباید هم در intellisense ظاهر شود و پس از آن هم نباید قابل کامپایل باشد.

بنابراین صورت مساله به این ترتیب اصلاح می‌شود:
می‌خواهیم پس از نوشتن FluentApiTest و قرار دادن یک نقطه، در intellisense فقط Number ظاهر شود و نه هیچ متد دیگری. پس از ذکر متد Number فقط متد Abs یا مواردی شبیه به آن مانند Sqrt ظاهر شوند. پس از انتخاب مثلا Abs آنگاه متد IsEqualTo توسط Intellisense قابل دسترسی باشد. در روش اول فوق، به صورت دوستانه همه چیز در دسترس است و هر ترکیب قابل کامپایلی را می‌شود با متدها ساخت که این مورد نظر ما نیست.
اینبار پیاده سازی اولیه به شرح زیر تغییر خواهد کرد:

```
using System;

namespace FluentInt
{
 public class FluentApiTest
 {
   public MathMethods<FluentApiTest> Number(int val)
   {
       return new MathMethods<FluentApiTest>(this, val);
   }
 }

 public class MathMethods<TParent>
 {
   private int _val;
   private readonly TParent _parent;

   public MathMethods(TParent parent, int val)
   {
       _val = val;
       _parent = parent;
   }

   public Restrictions<MathMethods<TParent>> Abs()
   {
       _val = Math.Abs(_val);
       return new Restrictions<MathMethods<TParent>>(this, _val);
   }
 }

 public class Restrictions<TParent>
 {
   private readonly int _val;
   private readonly TParent _parent;

   public Restrictions(TParent parent, int val)
   {
       _val = val;
       _parent = parent;
   }

   public bool IsEqualTo(int val)
   {
       return _val == val;
   }
 }
}
```

در اینجا هم به همان کاربرد اولیه می‌رسیم:

```
if (new FluentApiTest().Number(-10).Abs().IsEqualTo(10))
{
 Console.WriteLine("Abs(-10)==10");
}
```

با این تفاوت که intellisense هربار فقط یک متد مرتبط در طول زنجیره را نمایش می‌دهد و تمام متدها در همان ابتدای کار قابل انتخاب نیستند.

در پیاده سازی کلاس MathMethods از Generics استفاده شده به این جهت که بتوان نوع متد Number را بر همین اساس تعیین کرد تا متدهای کلاس MathMethods در Intellisense (یا به قولی در طول زنجیره مورد نظر) ظاهر شوند. کلاس Restrictions نیز به همین ترتیب معرفی شده است و از آن جهت تعریف نوع متد Abs استفاده کردیم. هر کلاس جدید در طول زنجیره، توسط سازنده خود به وهله‌ای از کلاس قبلی به همراه مقادیر پاس شده دسترسی خواهد داشت. به این ترتیب زنجیره‌ای را تشکیل داده‌ایم که سازمان یافته است و نمی‌توان در آن متدی را بی‌جهت پیش یا پس از دیگری صدا زد و همچنین دیگر نیازی به بررسی نحوه‌ی فراخوانی‌های یک مصرف کننده نیز نخواهد بود زیرا برنامه کامپایل نمی‌شود.

نظرات خوانندگان

نویسنده: Saber Soleymani
تاریخ: ۱۳۹۰/۰۳/۰۴ ۱۵:۰۰:۳۲

مقاله خوبی بود. اما درست است که استفاده از فراخوانی زنجیره‌ای با این روش ساده‌تر و ایمن‌تر می‌شود، اما خوانایی کد را (در مقایسه با روش اول) پایین‌تر می‌آورد.
در کل روش جالبی بود.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۳/۰۴ ۱۶:۲۱:۱۹

بله روش دوم ساده نیست اما نتیجه نهایی آن برای کسی که قرار است از API شما استفاده کند یکی است و به همان اندازه ساده. در کل طراحی API خوب کار مشکلی است. برای نمونه ما از LINQ لذت می‌بریم (به عنوان استفاده کننده نهایی) ولی واقعا پیاده سازی اون مشکل بوده و پشت صحنه ساده‌ای نداره.

نویسنده: Alidoosti
تاریخ: ۱۳۹۰/۰۳/۰۵ ۱۴:۲۴:۲۳

Constructor کلاس Restrictions به صورت (Public Restriction(MathMethods parent, int valنیز درست است ؟؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۳/۰۶ ۰۲:۱۸:۵۴

خیر؛ یک پارامتر جنریک به عنوان ورودی یک پارامتر جنریک قابل تعریف نیست.

پ.ن.
بلاگر در ارسال یک سری کاراکترها حساسیت دارد حتی در کامنت‌ها. همان کاراکترهای غیرمجاز در XML که باید به صورت escape شده معرفی شوند و گرنه خیلی ساده یا آن‌ها را نمایش نمی‌دهد یا حذف می‌کند (مشکلی که به نظر با تعریف جنریک‌ها داشتید و کاراکترها حدف شده بود؛ علتش این مبحث است).

نویسنده: امیرحسین
تاریخ: ۱۳۹۱/۰۶/۰۶ ۳:۴۵

سلام
به نظر من این امکانات فقط برای پروژه‌های مانند LINQ خوبه و واقعا در محیطهای واقعی قابل پیاده سازی نیست.
چون این کار خودش نیاز به تحلیل جدا داره ... و پیاده ساز ی اون هم وقت گیره.
مگر در مواردی که قرار لایه بندی برنامه در بالاترین صطح و کیفیت قرار داشته باشه ، که باز هم بعید می‌دونم در این حد نیاز باشه.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۶/۰۶ ۹:۱۱

- نمونه پیاده سازی شده اون رو در پروژه نسبتا بزرگ fluent nhibernate می‌تونید مشاهده کنید.
- پروژه بزرگ دیگری که از این روش استفاده می‌کنه ASP.NET MVC Extensions شرکت telerik است (برای طراحی API نهایی قابل استفاده از آن).
- همچنین اکثر افزونه‌ها و کتابخانه‌های کمکی طراحی شده برای ASP.NET MVC از روش Fluent interfaces استفاده می‌کنند. مثلا fluent validation ، fluent security و غیره.
- اخیرا هم اعضای تیم Entity framework، قسمتی از کار تنظیم نگاشت‌ها را توسط روشی به نام Fluent API طراحی کرده‌اند(در

.(EF Code first

نویسنده: حسام
تاریخ: ۱۳۹۱/۰۶/۰۶ ۱۰:۳۴

در این روش ضرورت استفاده از genericها چیست ؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۶/۰۶ ۱۰:۳۷

لطفا در مطلب فوق از قسمت «بنابراین صورت مساله به این ترتیب اصلاح می‌شود» را مطالعه کنید. هدف مقید کردن استفاده کننده از API به انتخاب متدهایی خاص است و نه هر متد ممکنی در طول یک زنجیره.

عنوان: **کمپین ضد IF !**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۶/۰۴ ۰۰:۵۸:۰۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

بکارگیری بیش از حد If و خصوصا Switch برخلاف اصول طراحی شیءگرا است؛ تا این حد که یک کمپین ضد IF هم وجود دارد!

البته سایت فوق بیشتر جنبه تبلیغی برای سمینارهای گروه مذکور را دارد تا اینکه جنبه‌ی آموزشی/خود آموزی داشته باشد.

یک مثال کاربردی:
فرض کنید دارید یک سیستم گزارشگیری را طراحی می‌کنید. به جایی می‌رسید که نیاز است با Aggregate functions سروکار داشته باشید؛ مثلا جمع مقادیر یک ستون را نمایش دهید یا معدل امتیازهای نمایش داده شده را محاسبه کنید و امثال آن. طراحی متداول آن به صورت زیر خواهد بود:

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

namespace CircularDependencies
{
    public enum AggregateFunc
    {
        Sum,
```

```
        Avg
    }

    public class AggregateFuncCalculator
    {
        public decimal Calculate(IList<decimal> list, AggregateFunc func)
        {
            switch (func)
            {
                case AggregateFunc.Sum:
                    return getSum(list);
                case AggregateFunc.Avg:
                    return getAvg(list);
                default:
                    return 0m;
            }
        }

        private decimal getAvg(IList<decimal> list)
        {
            if (list == null || !list.Any()) return 0;
            return list.Sum() / list.Count;
        }

        private decimal getSum(IList<decimal> list)
        {
            if (list == null || !list.Any()) return 0;
            return list.Sum();
        }
    }
}
```

در کلاس AggregateFuncCalculator یک متد Calculate داریم که توسط آن قرار است روی list دریافتی یک سری عملیات انجام شود. عملیات پشتیبانی شده هم توسط یک enum معرفی شده؛ برای مثال اینجا فقط جمع و میانگین پشتیبانی می‌شوند.
و مشکل طراحی این کلاس، همان switch است که برخلاف اصول طراحی شیءگرا می‌باشد. یکی از اصول طراحی شیءگرا بر این مبنا است که:
یک کلاس باید جهت تغییر، بسته اما جهت توسعه، باز باشد.

یعنی چی؟
داستان طراحی Aggregate functions که فقط به جمع و میانگین خلاصه نمی‌شود. امروز می‌گویند واریانس چطور؟ فردا خواهند گفت حداقل و حداکثر چطور؟ پس فردا ...

به عبارتی این کلاس جهت تغییر بسته نیست و هر روز باید بر اساس نیازهای جدید دستکاری شود.

چکار باید کرد؟
آیا می‌توانید در کلاس AggregateFuncCalculator یک الگوی تکراری را تشخیص دهید؟ الگوی تکراری موجود، محاسبات بر روی یک لیست است. پس می‌شود بر اساس آن یک اینترفیس عمومی را تعریف کرد:

```
public interface IAggregateFunc
{
     decimal Calculate(IList<decimal> list);
}
```

اکنون هر کدام از پیاده سازی‌های موجود در کلاس AggregateFuncCalculator را به یک کلاس جدا منتقل خواهیم کرد تا یک اصل دیگر طراحی شیءگرا نیز محقق شود:
هر کلاس باید تنها یک کار را انجام دهد.

```
public class Sum : IAggregateFunc
{
        public decimal Calculate(IList<decimal> list)
        {
            if (list == null || !list.Any()) return 0;
            return list.Sum();
        }
}

public class Avg : IAggregateFunc
{
        public decimal Calculate(IList<decimal> list)
        {
            if (list == null || !list.Any()) return 0;
            return list.Sum() / list.Count;
        }
}
```

تا اینجا 2 هدف مهم حاصل شده است:
- کم کم کلاس AggregateFuncCalculator دارد خلوت می‌شود. قرار است هر کلاس یک کار را بیشتر انجام ندهد.
- برنامه از بسته بودن جهت توسعه هم خارج شده است (یکی دیگر از اصول طراحی شیءگرا). اگر تعاریف توابع محاسباتی را تماما در یک کلاس قرار دهیم صاحب اول و آخر آن کتابخانه خودمان خواهیم بود. این کلاس بسته است جهت تغییر. اما با معرفی IAggregateFunc، من امروز 2 تابع را تعریف کرده‌ام، شما فردا توابع خاص خودتان را تعریف کنید. باز هم برنامه کار خواهد کرد. نیازی نیست تا من هر روز یک نگارش جدید از کتابخانه را ارائه دهم که در آن فقط یک تابع دیگر اضافه شده است.

اکنون یکی از چندین و چند روش بازنویسی کلاس AggregateFuncCalculator به صورت زیر می‌تواند باشد

```
public class AggregateFuncCalculator
{
        public decimal Calculate(IList<decimal> list, IAggregateFunc func)
        {
            return func.Calculate(list);
        }
}
```

بله! دیگر سوئیچی در کار نیست. این کلاس تنها یک کار را انجام می‌دهد. همچنین دیگر نیازی به تغییر هم ندارد (محاسبات از آن خارج شده) و باز است جهت توسعه (شما نگارش‌های دلخواه IAggregateFunc دیگر خود را توسعه داده و استفاده کنید).

نظرات خوانندگان

نویسنده: Meysam Javadi
تاریخ: ۱۳۹۰/۰۶/۰۴ ۱۰:۱۷:۲۹

استادانه به مثال رسوندید.http://en.wikipedia.org/wiki/Strategy_pattern

نویسنده: جلال
تاریخ: ۱۳۹۰/۰۶/۰۶ ۰۱:۱۲:۵۴

به این روش، Dependency Injection گفته میشه که برای حالت های پیشرفتش فریم ورک هم طراحی شده! مثل Ninject یا اون یکی مال خود مایکروسافت به اسم Unity

لیست کاملش http://www.hanselman.com/blog/ListOfNETDependencyInjectionContainersIOC.aspx

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۶/۰۶ ۰۸:۱۵:۳۵

توصیه می‌کنم مطالب زیر رو مطالعه کنید:

dependency-injection

ioc-inversion-of-control

بعد تفاوت این‌ها مثلا با الگوی استراتژی بهتر مشخص می‌شود.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۰/۰۶/۰۶ ۰۹:۰۴:۴۱

توضیحات تکمیلی:
سؤال : آیا refactoring صورت گرفته در مطلب فوق از نوع تزریق وابستگی‌ها (dependency injection) بود؟
پاسخ: خیر.
پیاده سازی الگوی تزریق وابستگی‌ها زمانی معنا پیدا می‌کند که شما حداقل 2 کلاس داشته باشید (مطلب فوق با یک کلاس شروع شد)، همچنین این دو کلاس ارجاعی به یکدیگر داشته باشند و اصطلاحا به هم گره خورده باشند.

سؤال : چگونه در یک پروژه بزرگ می‌توان نیاز به پیاده سازی الگوی تزریق وابستگی‌ها را تشخیص داد؟
پاسخ:
آیا نسخه‌ی ultimate ویژوال استودیوی 2010 بر روی سیستم شما نصب است؟
اگر بله: (نصب است)
برای نمونه به مطلب Discovering Circular References مراجعه کنید.

اگر خیر: (نصب نیست)
در این حالت از ابزار رایگانی به نام .NET Architecture Checker می‌توانید استفاده کنید. همان نمودارهای نسخه‌ی ultimate ویژوال استودیو را برای شما ترسیم خواهد کرد.

سؤال : آیا می‌توان از کتابخانه‌های تزریق وابستگی‌ها و فریم ورک‌های مرتبط، جهت مدیریت ساده‌تر قسمت آخر مطلب فوق یعنی تامین پیاده سازی‌های اینترفیس‌هایی که قرار است در زمان اجرا استفاده شوند، کمک گرفت؟
پاسخ: بله.
این مورد یکی از کاربردهای متداول این ابزارها است (برای مثال ساخت برنامه‌های افزونه پذیر و همچنین ساده‌تر کردن Object composition و وهله سازی‌های مرتبط) و ... این مورد را نباید با اصل refactoring صورت گرفته در مثال جاری اشتباه گرفت.

نویسنده: Kamyar
تاریخ: ۱۳۹۰/۰۶/۰۶ ۱۵:۳۶:۲۳

من شباهتی بین مطلب این مقاله و Dependency Injection نمی بینم.
مطلب بالا دقیقا پیاده سازی الگوی طراحی Strategy هست. جایی که رفتارها (عملیات محاسبه Aggregate) از رفتار کننده (محاسبه گر، ماشین حساب) جدا شده و در کلاسهای خودشان که یک اینترفیس مشترک را پیاده سازی می کنند، تعریف می شوند.

عنوان: **پیاده سازی رخداد گردانهای یک شیء در الگوی MVP**
نویسنده: امیر هاشم زاده
تاریخ: ۱۳۹۱/۰۳/۲۹ ۰:۱۴
آدرس: www.dotnettips.info
برچسب‌ها: ASP.Net, Design patterns, MVP, Telerik, Grid, RadGrid

یکی از ضروریات دنیای برنامه نویسی امروز، داشتن یک الگوی مناسب می‌باشد. یکی از الگوهای مناسب برای وب فرم‌ها، استفاده از الگوی MVP است.
اگر در خلال پیاده سازی، گاهی اوقات نیاز به handle کردن رخدادها را داشته باشید بدین منظور به روش زیر عمل می‌کنیم:
(توجه: شیء مورد نظر ما در این پست RadGrid از کنترلهای Telerik در نظر گرفته شده است.)

```
// ASPX page
<telerik:RadGrid ID="RadGrid1" runat="server"></telerik:RadGrid>
// Asp.Net Code Behind
protected void Page_Load(object sender, EventArgs e)
{
    GridPresenter presenter = new GridPresenter(this);
}
// view interface
public interface IGridView
{
    Telerik.Web.UI.RadGrid myGrid { get; }
}
// presenter
protected readonly IGridView _view;
public GridPresenter(IGridView view)
{
    _view = view;
    _view.myGrid.UpdateCommand += new Telerik.Web.UI.GridCommandEventHandler(onUpdateCommand);
    _view.myGrid.InsertCommand += new Telerik.Web.UI.GridCommandEventHandler(onInsertCommand);
    _view.myGrid.EditCommand += new Telerik.Web.UI.GridCommandEventHandler(onEditCommand);
}
private void onUpdateCommand(object sender, Telerik.Web.UI.GridCommandEventArgs e)
{
    // Code for updating
}
private void onInsertCommand(object sender, Telerik.Web.UI.GridCommandEventArgs e)
{
    // Code for inserting
}
private void onEditCommand(object sender, Telerik.Web.UI.GridCommandEventArgs e)
{
    // Code for editcommand
}
```

نظرات خوانندگان

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۳/۲۹ ۰:۴۹

خوبه
اما من یه قسمت رو متوجه نشدم.ببینید ما میگیم MVP (Model View Presenter).
خب حالا سوال اینجاست که در بخش Asp.net Code Behind از چه کدی استفاده میکنیم؟
یعنی کدهای View و Presenter چه موقع فراخوانی میشوند؟
از کجا اجرا میشود و به کجا ختم میشود؟
مدل ما کجاست؟
و ...

نویسنده: امیر هاشم زاده
تاریخ: ۱۳۹۱/۰۳/۲۹ ۱۱:۱

ابتدا وب فرم مورد نظر اینترفیس IGridView را پیاده سازی می‌کند و سپس در Code behind در Page_Load یک وهله از presenter با پارامتر this ایجاد می‌کنید، در نتیجه کلیه رخدادها در presenter مدیریت می‌شوند.
نکته در این است که چون هر بارگذاری صفحه نیاز به متصل کردن رخدادها به شی هستیم مجبوریم در Page_Load یک وهله از presenter ایجاد کنیم.
در این پست هدف پیاده سازی مدل نبوده است، ولی شما می‌توانید مدل خود را با توجه به الگوی MVP طراحی کنید و از آن در presenter استفاده کنید.

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۳/۲۹ ۱۴:۲۶

مرسی
مشکل قطعه کد زیر بود که جا افتاد بود و باعث شده بود که نحوه استفاده رو متوجه نشم!

```
GridPresenter presenter = new GridPresenter(this);
```

در مورد Model هم شما درست میگید.هدف این پست استفاده از Model نبود.

نویسنده: جلال
تاریخ: ۱۳۹۱/۰۷/۲۲ ۱۲:۴۱

سلام،
با این قضیه ارجاع مستقیم به یک کنترل خاص در IView می‌تونم کنار بیام، ولی با ارجاع به اون در CPresenter نه!
دلیل خاصی داشته؟ آخه معمولا برای رخدادگردانی، EventHandlerها رو در IView معرفی می‌کنن و در CPresenter استفاده می‌کنند.

البته در این مقاله ، کلا منکر استفاده از EventHandlerها در IView میشه و ترجیح میده فقط تابع Presenter رو از Code Behind فراخوانی کنه!

نویسنده: امیر هاشم زاده
تاریخ: ۱۳۹۱/۰۸/۱۶ ۱۸:۴۸

یکی از دلایل استفاده از این روش، تمیز نگه داشته داشتن ویو (با توسعه برنامه پارامترهای لازم برای 4عمل اصلی اضافه می‌شد

که نگهداری پارامترهای رخداد گردانها پیچیده و دشوار می‌شد) بود، زمانی که داشتم این پیاده سازی رو در پروژه انجام می‌دادم به این نتیجه رسیده بودم بهتر است کلیه عملیات‌های افزودن، حذف، ویرایش و حذف شی تلریک گرید ویو رو در presenter انجام بدم چون قرار بود یک سری عملیات منطقی بر روی 4عمل اصلی انجام بدم.

عنوان: **چرا به الگوهای طراحی JavaScript نیاز داریم ؟**
نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۳/۳۰ ۲۱:۵۵
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, JavaScript

امروزه اهمیت یادگیری JavaScript بر هیچ کس پوشیده نیست ، APIهای جدید HTML 5 و امکانات جدید وب مثل Geo Location Offline Web applications ، Canvas ، Offline storage ، و ... همه نیاز به دانش JavaScript دارند. در نتیجه نیاز است علاوه بر یادگیری JavaScript به تمیز کد نوشتن جهت سهولت نگهداری آگاه بود. همانطور که در کدهای سمت سرور مثل C# و یا PHP نیاز به استفاده از الگوهای طراحی (Design Patterns)است در JavaScript هم اوضاع به همین منوال است.
الگوی طراحی یک راه حل قابل استفاده مجدد است که برای حل مشکلات متداول در طراحی نرم افزار به کار می‌رود.
چرا به الگوی‌های طراحی JavaScript نیازمندیم ؟
می خواهیم کد هایی با قابلیت استفاده‌ی مجدد بنویسیم ، استفاده از عملکردهای مشابه در سطح صفحات یک Web application یا چند Web Application.
می خواهیم کد هایی با قابلیت نگهداری بنویسیم ، هر چه قدر در فاز توسعه کدهای با کیفیت بنویسیم در فاز نگهداری از آن بهره می‌بریم. باید کد هایی بنویسیم که قابل Debug و خواندن توسط دیگر افراد تیم باشند.
کدهای ما نباید با توابع و متغیرهای دیگر پلاگین‌ها تداخل نامگزاری داشته باشند. در برنامه‌های امروزی بسیار مرسوم است که از پلاگین‌های Third party استفاده شود. می‌خواهیم با رعایت Encapsulation and modularization در کدهایمان از این تداخل جلوگیری کنیم.

معمولا کدهای JavaScript که توسط اکثر ما نوشته می‌شود یک سری تابع پشت سرهم هست ، بدون هیچ کپسوله سازی :

```
function getDate() {
    var now = new Date();
    var utc = now.getTime() + (now.getTimezoneOffset() * 60000);
    var est;
    est = new Date(utc + (3600000 * -4));
    return dateFormat(est, "dddd, mmmm dS, yyyy, h:MM:ss TT") + " EST";

}
function initiate_geolocationToTextbox() {
    navigator.geolocation.getCurrentPosition(handle_geolocation_queryToTextBox);
}

function handle_geolocation_queryToTextBox(position) {
    var longitude = position.coords.longitude;
    var latitude = position.coords.latitude;
    $("#IncidentLocation").val(latitude + " " + longitude);
}
```

به این روش کدنویسی Function Spaghetti Code گفته می‌شود که معایبی دارد :
توابع و متغیرها به Global scope برنامه افزوده می‌شوند.
کد Modular نیست.
احتمال رخ دادن Conflict در اسامی متغیرها و توابع بالا می‌رود.
نگهداری کد به مرور زمان سخت می‌شود.

با شبیه سازی یک مثال مشکلات احتمالی را بررسی می‌کنیم :

```
// file1.js
function saveState(obj) {
    // write code here to saveState of some object
    alert('file1 saveState');
}
// file2.js (remote team or some third party scripts)
function saveState(obj, obj2) {
     // further code...
    alert('file2 saveState");
}
```

همانطور که می‌بینید در این مثال در 2 فایل متفاوت در برنامه مان از 2 تابع با اسامی یکسان و امضای متفاوت استفاده کرده ایم .
اگر فایل‌ها را اینگونه در برنامه آدرس دهی کنیم :

```
<script src="file1.js" type="text/javascript"></script>
<script src="file2.js" type="text/javascript"></script>
```

متد saveState در فایلی که دیرتر آدرس داده شده (file2.js) ، متد saveState در file1.js را Override می‌کند ، در نتیجه عملکردی که از متد saveState در فایل اول انتظار داریم اتفاق نمی‌افتد.
در پست بعدی به راه حل این مشکلات و کپسوله سازی خواهم پرداخت.
برای مطالعه‌ی بیشتر کتاب (Learning JavaScript Design Patterns) را از دست ندهید.

نظرات خوانندگان

نویسنده: NargesM
تاریخ: ۱۳۹۱/۰۳/۳۱ ۶:۸

سلام.. ممنون

عنوان: **Closure در JavaScript**
نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۴/۰۴ ۷:۱۸
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, JavaScript, Closure

در قسمت قبلی درباره علت نیاز به الگوهای طراحی در JavaScript و Function Spaghetti code صحبت شد. در این قسمت Closure در JavaScript مورد بررسی قرار می‌گیرد.
در JavaScript می‌توان توابع تو در تو نوشت (nested functions) ، زمانی که یک تابع درون تابع دیگر تعریف می‌شود تابع درونی به تمام متغیرها و توابع تابع بیرونی (Parent) دسترسی دارد.
Douglas Crockford
برای تعریف Closure می‌گوید :

```
an inner function always has access to the vars and parameters of its outer function, even after the
outer



function has returned
```

یک تابع درونی (nested) همیشه به متغیرها و پارامترها تابع بیرونی دسترسی دارد ، حتی اگر تابع بیرونی مقدار برگردانده باشد.

تابع زیر را در نظر بگیرید :

```
// The getDate() function returns a nested function which refers the 'date' variable defined
// by outer function getDate()
function getDate() {
   var date = new Date();    // This variable stays around even after function returns

   // nested function
   return function () {
      return date.getMilliseconds();
   }
}
```

اکنون اگر به صورت زیر تابع getDate فراخوانی شود مشاهده می‌شود که تابع درونی (با کامنت nested function مشخص شده است.) به شیء date دسترسی دارد.

```
// Once getDate() is executed the variable date should be out of scope and it is, but since
// the inner function
// referenes date, this value is available to the inner function.
var dt = getDate();

alert(dt());
alert(dt());
```

خروجی هر 2 alert یک مقدار خواهد بود.
اگر از فردی که به تازگی رو به JavaScript آورده است خواسته شود تابعی بنویسد که میلی ثانیه‌ی زمان جاری را برگداند احتمالا همچین کدی تحویل می‌دهد :

```
function myNonClosure() {
            var date = new Date();
            return date.getMilliseconds();
        }
```

در کد بالا پس از اجرای myNonClosure متغیر date از بین خواهد رفت ، این مسئله در دنیای JavaScript طبیعی هست.
این مثال را در نظر بگیرید :

```
var MyDate = function () {
    var date = new Date();
    var getMilliSeconds = function () {
        return date.getMilliseconds();
    }
}

var dt = new MyDate();

alert(dt.getMilliSeconds());  // This will throw error as getMilliSeconds is not accessible.
```

در صورت اجرای مثال بالا خطایی با این مضمون دریافت خواهد شد که getMilliSeconds دستیابی پذیر نیست. (کپسوله شده)
برای اینکه آن را دستیابی پذیر کنیم کد را به این صورت تغییر می‌دهیم :

```
// This is closure
var MyDate = function () {
    var date = new Date();   // variable stays around even after function returns
    var getMilliSeconds = function () {
        return date.getMilliseconds();
    };
    return {
        getMs : getMilliSeconds
    }
}
```

آنچه در تابع بالا انجام شده کپسوله سازی همه‌ی منطق کار (منطق کار در اینجا برگرداندن میلی ثانیه زمان جاری می‌باشد) در یک فضای نام به نام MyDate می‌باشد. همچنین فقط متدهای عمومی در اختیار استفاده کننده این تابع قرار داده شده است. برای استفاده می‌توان بدین صورت عمل کرد :

```
var dt = new MyDate();
alert(dt.getMs());  // This should work.
```

در کد بالا برای توابع و متغیرهای درونی یک container ایجاد کردیم که باعث جلوگیری از تداخل در نام متغیرها با دیگر کدها خواهد شد . (برای مشاهده‌ی تداخل‌ها به قسمت قبلی توجه کنید.)
اگر بخواهیم Closure را تشبیه کنیم ، Closure شبیه به کلاس‌ها در C# یا Java هست.
Closure یک حوزه (scope) برای متغیرها و توابع درونی خودش ایجاد می‌کند.
jQuery بهترین مثال کاربردی برای Closure می‌باشد :

```
(function($) {

    // $() is available here

})(jQuery);
```

در ادامه این مفاهیم بیشتر توضیح داده می‌شودند ، اکنون می‌خواهیم مشکلی که در قسمت قبلی مطرح کردیم به کمک Closure حل کنیم :

در آن مثال گفته شد که اگر :

```
// file1.js
function saveState(obj) {
    // write code here to saveState of some object
    alert('file1 saveState');
}
// file2.js (remote team or some third party scripts)
function saveState(obj, obj2) {
```

```
     // further code...
    alert('file2 saveState");
}
```

اگر تابعی به نام saveState در 2 فایل مختلف داشته باشیم و این 2 فایل را بدین صورت در برنامه آدرس دهیم :

```
<script src="file1.js" type="text/javascript"></script>
<script src="file2.js" type="text/javascript"></script>
```

تابع saveState در فایل دوم تابع saveState فایل اول را override می‌کند. یک از توابع بالا را به صورت زیر باز نویسی می‌کنیم و منطق کار را کپسوله می‌کنیم :

```
function App() {
    var save = function (o) {
        // write code to save state here..
        // you have acces to 'o' here...
        alert(o);
    };

    return {
        saveState: save
    };
}
```

بدون نگرانی تداخل saveState با بقیه saveStateها در هر پلاگین یا فایل دیگری می‌توان از saveState اینگونه استفاده کرد :

```
var app = new App();

app.saveState({ name: "rajesh"});
```

برای اطلاعات بیشتر در مورد Closure ها این لینک را بررسی کنید.

نظرات خوانندگان

نویسنده: MBE
تاریخ: ۱۳۹۱/۰۴/۰۴ ۲۰:۳۱

ممنون عالی بود . اما این بحث واقعا جای کار داره . من که منتظر مقالات بیشتری در مورد Closure هستم .

نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۴/۰۴ ۲۱:۳۱

خواهش می‌کنم. در ادامه درباره الگوهای Prototype و Module مطلب می‌نویسم و سعی می کنم با یک مثال بحث را بیشتر باز کنم.

نویسنده: صالح
تاریخ: ۱۳۹۱/۰۴/۰۴ ۲۳:۴۸

نکته بسیار خوب و کاربردی در جاوا اسکریپت بود. ممون

نویسنده: پوران
تاریخ: ۱۳۹۲/۱۱/۱۹ ۱۵:۵۵

واقعا مرسی ... عالی بود ...

عنوان: **نحوه استفاده صحیح از لوسین در ASP.NET**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۴/۲۹ ۹:۳۵
آدرس: www.dotnettips.info
برچسب‌ها: ASP.Net, Design patterns, Lucene.NET

بر مبنای پیاده سازی متداولی که در n هزار سایت اینترنتی می‌توان یافت، نحوه کار با جستجوگر لوسین حدودا به این شکل است:

```
var directory = FSDirectory.Open(new DirectoryInfo(Environment.CurrentDirectory + "\\LuceneIndex"));
using (var searcher = new IndexSearcher(directory, readOnly: true))
{
      //do something ...

       searcher.Close();
       directory.Close();
}
```

و ... اینکار به این شکل غلط است!
مطابق مستندات رسمی لوسین، این کتابخانه thread-safe است. به این معنا که در آن واحد چندین و چند کاربر می‌توانند از یک وهله از شیءهای Reader و Searcher استفاده کنند و نباید به ازای هر جستجو، یکبار این اشیاء را ایجاد و تخریب کرد. البته در اینجا تنها یک Writer در آن واحد می‌تواند مشغول به کار باشد.
مشکلاتی که به همراه باز و بسته کردن بیش از حد IndexSearcher وجود دارد، مصرف بالای حافظه است (به ازای هر کاربر مراجعه کننده، یکبار باید ایندکس‌ها در حافظه بارگذاری شوند) و همچنین تاخیر اولیه این بارگذاری و کندی آن‌را نیز باید مدنظر داشت.

نتیجه گیری:
برای کار با جستجوگر لوسین نیاز است از الگوی Singleton استفاده شود و تنها یک وهله از این اشیاء بین تردهای مختلف به اشتراک گذاشته شود.

نظرات خوانندگان

نویسنده: حسین غلامی
تاریخ: ۱۳۹۱/۱۰/۱۱ ۱:۲۹

آقای نصیری میشه نمونه ای رو با استفاده از این الگو ، مثال بزنید؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۱۰/۱۱ ۱۰:۹

مراجعه کنید به مثال Auto Complete .

عنوان: **ایجاد یک Repository در پروژه برای دستورات EF**
نویسنده: میثم ثوامری
تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۱:۴۷
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, Design patterns, Entity framework, VB.NET

ایجاد یک Pattern در پروژتون میتونه نظم, سرعت و زیبایی خاصی به کدتون بده. با وجود frameworkهای و Patternهایی مسه MVC و MVVM برنامه نویسان را وادار کنه که همه Actionهای یک پروژه رو به سمت کلاینت ببرن.تو یک فرصت دیگه در مورد فریمورک Knockout حتما تاپیک میزارم.امروز میخوام یک Pattern با استفاده از یک Interface و codefirst model براتون بزارم.

گام اول:ایجاد که class property

```
Public Class Employee
    Public Property ID As Integer
    Public Property Fname As String
    Public Property Bdate As DateTime
End Class
```

گام دوم:ایجاد بانک با استفاده از CodeFirst

```
Imports System.Data.Entity
Public Class EmployeeDBContext : Inherits DBContext
    Public Property Employees As DbSet(Of Employee)
End Class
```

گام سوم: ایجاد repository با استفاده از interface

```
Interface EmployeeRepository
    ReadOnly Property All As List(Of Employee)
    Function Find(id As Integer) As Employee
    Sub InsertOrUpdate(p As Employee)
    Sub Delete(id As Integer)
    Sub Save()
End Interface
```

گام چهارم: تعریف کلاس برای implement کردن از iInterface

```
Public Class EmployeeClass : Implements EmployeeRepository
    Private DB As New EmployeeDBContext
    Public ReadOnly Property All As List(Of Employee) Implements EmployeeRepository.All
        Get
            Return DB.Employees.ToList()
        End Get
    End Property

    Public Sub Delete(id As Integer) Implements EmployeeRepository.Delete
        Dim query = DB.Employees.Single(Function(q) q.ID = id)
        DB.Employees.Remove(query)
    End Sub

    Public Function Find(id As Integer) As Employee Implements EmployeeRepository.Find
        Return DB.Employees.Where(Function(q) q.ID = id)
    End Function

    Public Sub InsertOrUpdate(p As Employee) Implements EmployeeRepository.InsertOrUpdate
        If p.ID = Nothing Then
            DB.Employees.Add(p)
        Else
            DB.Entry(p).State = Data.EntityState.Modified
        End If
    End Sub

    Public Sub Save() Implements EmployeeRepository.Save
        DB.SaveChanges()
    End Sub
```

```
End Class
```

برای استفاده تو پروژه براحتی میتونید یک instance از classتون ایجاد کنید و ..

```
Dim cls As New EmployeeClass
```

```
Public Sub BindGrid()
        GridView1.DataSource = cls.All
        GridView1.DataBind()
    End Sub
```

موفق باشید

نظرات خوانندگان

نویسنده: علیرضا صالحی
تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۲:۱۳

برای مواردی که خروجی یک لیست (تعدادی آیتم) باشد از Property استفاده نمی‌شود. مثلا برای All باید از Method استفاده کنید. Properties vs. Methods
بهتر است برای خروجی متد هایی مانند All نیز به جای لیست از IEnumerable یا IQueryable استفاده کنید.
متدهای Update و Insert نیز به طور جداگانه تعریف شوند. (قرار است هر متد تنها یک وظیفه داشته باشد)

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۲:۲۱

- در مورد آرایه بحث شده در MSDN. ضمن اینکه استفاده از متد عموما برای حالتیکه عملیات قابل توجهی در بدنه آن قرار است صورت گیرد، توصیه می‌شود . البته در اینجا چون عملیات دریافت اطلاعات از بانک اطلاعاتی می‌تواند سنگین درنظر گرفته شود، استفاده از متد ارجحیت دارد. خواص نمایانگر اطلاعاتی سبک و با دسترسی سریع هستند.
- خروجی لیست بهتر است. (^) + اگر ReSharper جدید را نصب کنید استفاده از IEnumerable را نیز توصیه نمی‌کند ؛ چون ممکن است چندین بار رفت و برگشت به بانک اطلاعاتی در این بین صورت گیرد.
- مشکلی ندارد. خود EF Code first چنین متدی را دارد. (^) بحث کلاس تک وظیفه‌ای متفاوت است با متدی که نهایتا قرار است اطلاعات یک رکورد را در بانک اطلاعاتی تغییر دهد (اگر نبود ثبتش کند؛ اگر بود فقط همان رکورد مشخص را به روز رسانی کند).

نویسنده: ناشناس
تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۲:۳۷

با سلام
دلیل استفاده از Interface EmployeeRepository چیه؟
دقیقا دلیل استفاده Interface اینجا چیه؟
با تشکر از مطلب خوبتون.

نویسنده: ناشناس
تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۲:۴۵

اینجا شاید استفاده از IQueryable بهتر باشه.
شاید کاربر بخواد قبل از نمایش اطلاعات اونو فیلتر کنه یا اینکه بهتره دو متد Find داشته باشی یکی با خروجی یک آیتم و دیگری با خروجی چندین آیتم.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۲:۴۸

خیر (^). طراحی یک لایه سرویس که خروجی IQueryable دارد نشتی دار درنظر گرفته شده و توصیه نمی‌شود. اصطلاحا leaky abstraction هم به آن گفته می‌شود؛ چون طراح نتوانسته حد و مرز سیستم خودش را مشخص کند و همچنین نتوانسته سازوکار درونی آن‌را به خوبی کپسوله سازی و مخفی نماید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۲:۵۰

به دو دلیل:
- استفاده از امکان تزریق وابستگی‌ها

- امکان نوشتن ساده‌تر آزمون‌های واحد با فراهم شدن زیر ساخت mocking اشیاء

نویسنده: ناشناس
تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۳:۲

در همین پست خود شما تعداد زیاد رکوردها رو مثال زدیدو این پیاده سازی از این موضوع رنج میبره. و در مورد IQueryable قبول دارم و گفتم که بهتر است از دو یا چند متد find استفاده کنید.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۳:۲۵

منظور از آن مطلب این بود که از ابزاری که در اختیار دارید درست استفاده کنید. اگر قرار است دو یا چند جستجو را انجام دهید، اینکارها بله باید با IQueryable داخل یک متد انجام شود، اما خروجی متد فقط باید لیست حاصل باشد؛ نه IQueryable ایی که انتهای آن باز است و سبب نشتی لایه سرویس شما در لایه‌های دیگر خواهد شد.

نویسنده: میثم ثوامری
تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۹:۵۴

برای برنامه نویسا پیدا کردن یک property راحتره در ضمن از property برای تزریق یا بازیابی اطلاعات از یک object استفاده میکنن.
IQueryable در واقع توسعه یافته IEnumerable.تفاوت عمدشون در LINQ operators که در IQueryable استفاده میشه.اگر هم بخوایم دلیل پیشنهادی داده باشیم اونم اینه که مدیریت حافظه در IQueryable رعایت شده در حالی که Lisrtها کامپایلرو مجاب به اجنام دستور تا انتها میکنن.

نویسنده: میثم ثوامری
تاریخ: ۱۳۹۱/۰۵/۱۹ ۱۹:۵۹

مهندس با نظر دوستمون موافقم
IQueryable بهترین انتخاب برای remote data source که میشه به database یا webserviceها اشاره کرد.بطور کل اگر شما از ORM مسه linqtosql استفاده میکنید
IQueriable : کوئری شمارو به دستورات sql در database server تبدیل میکنه
IEnumerable: همه رکوردهای شما قبل از اینکه بسمت دیتابیس برن بصورت object در memory نگهداری میشن.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۵/۱۹ ۲۰:۱۱

IQueryable فقط یک expression است. هنوز اجرا نشده. (expose آن از طریق وب سرویس اشتباه است و به مشکلات serialization برخواهید خورد.)
زمانیکه First، ToList و امثال آن روی این عبارت فراخوانی شود تبدیل به SQL شده و سپس بر روی بانک اطلاعاتی اجرا می‌شود. به این deferred execution یا اجرای به تعویق افتاده گفته می‌شود.
اگر این عبارت را در اختیار لایه‌های دیگر قرار دهید، یعنی انتهای کار را بازگذاشته‌اید و حد و حدود سیستم شما مشخص نیست. شما اگر IQueryable بازگشت دهید، در لایه‌ای دیگر می‌شود یک join روی آن نوشت و اطلاعات چندین جدول دیگر را استخراج کرد؛ درحالیکه نام متد شما GetUsers بوده. بنابراین بهتر است به صورت صریح اطلاعات را به شکل List بازگشت دهید، تا انتهای کار باز نمانده و طراحی شما نشتی نداشته باشد.

نویسنده: محمد عامریان
تاریخ: ۱۳۹۱/۰۸/۱۸ ۱۷:۶

با سلام من یک معماری طراحی کردم به شکل زیر
ابتدا یک اینترفیس به شکل زیر دارم

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

namespace Framework.Model
{
    public interface IContext
    {
        T Get<T>(Func<T, bool> prediction) where T : class;
        IEnumerable List<T>(Func<T, bool> prediction) where T : class;
        void Insert<T>(T entity) where T : class;
        int Save();
    }
}
```

بعد یک کلاس ازش مشتق شده

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data;
using System.Data.Entity;
using System.Linq;
using System.Text;

namespace Framework.Model
{
    public class Context : IContext
    {
        private readonly DbContext _dbContext;

        public Context(DbContext context)
        {
            _dbContext = context;
        }

        public T Get<T>(Func<T,bool> prediction) where T : class
        {
            var dbSet = _dbContext.Set<T>();
            if (dbSet!= null)
                return dbSet.Single(prediction);

            throw new Exception();
        }

        public void Insert<T>(T entity) where T : class
        {
            var dbSet = _dbContext.Set<T>();
            if (dbSet != null)
            {
                _dbContext.Entry(entity).State = EntityState.Added;
            }
        }

        public int Save()
        {
            return _dbContext.SaveChanges();
        }


        IEnumerable IContext.List<T>(Func<T, bool> prediction)
        {
            var dbSet = _dbContext.Set<T>();
            if (dbSet != null)
                return dbSet.Where(prediction).ToList();

            throw new Exception();
        }
    }
}
```

سپس یک کلاش context دارم که مستقیما از dbcontext مشتق شده

```
using System.Data.Entity;
using DataModel;

namespace Model
{
    public class EFContext : DbContext
    {
        public EFContext(string db): base(db)
        {

        }

        public DbSet<Product> Products { get; set; }
    }
}
```

و سپس کلاس دارم که اومده پیاده سازی کرده context که خودم ساختمو

```
using System;
using System.Collections.Generic;
using System.Data.Entity;
using System.Linq;
using System.Text;

namespace Model
{
    public class Context : Framework.Model.Context
    {
        public Context(string db): base(new EFContext(db))
        {

        }
    }
}
```

در پروژه دیگری اومدم یک کلاس context جدید ساختم

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

namespace Biz
{
    public class Context : Model.Context
    {
        public Context(string db) : base(db)
        {

        }
    }
}
```

و در کنترلر هم به این شکل ازش استفاده کردم

```
using System.Web.Mvc;
using Framework.Model;

namespace ProductionRepository.Controllers
{
    public class BaseController : Controller
    {
        public IContext DataContext { get; set; }

        public BaseController()
        {
            DataContext = new
Biz.Context(System.Configuration.ConfigurationManager.ConnectionStrings["Database"].ConnectionString);
        }
    }
}
```

```
using System.Web.Mvc;
using DataModel;
using System.Collections.Generic;

namespace ProductionRepository.Controllers
{
    public class ProductController : BaseController
    {
        public ActionResult Index()
        {
            var x = DataContext.List<Product>(s => s.Name != null);
            return View(x);
        }

    }
}
```

و این هم تست

```
using NUnit.Framework;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Web.Mvc;

namespace TestUnit
{
    [TestFixture]
    public class Test
    {
        [Test]
        public void IndexShouldListProduct()
        {
            var repo = new Moq.Mock<Framework.Model.IContext>();
            var products = new List<DataModel.Product>();
            products.Add(new DataModel.Product { Id = 1, Name = "asdasdasd" });
            products.Add(new DataModel.Product { Id = 2, Name = "adaqwe" });
            products.Add(new DataModel.Product { Id = 4, Name = "qewqw" });
            products.Add(new DataModel.Product { Id = 5, Name = "qwe" });
            repo.Setup(x => x.List<DataModel.Product>(p => p.Name !=
null)).Returns(products.AsEnumerable());
            var controller = new ProductionRepository.Controllers.ProductController();
            controller.DataContext = repo.Object;
            var result = controller.Index() as ViewResult;
            var model = result.Model as List<DataModel.Product>;
            Assert.AreEqual(4, model.Count);
            Assert.AreEqual("", result.ViewName);

        }
    }
}
```

نظرتون چیه آقای نصیری

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۸/۱۸ ۱۷:۱۳

موارد 1 و 2 عنوان شده در این مطلب رو تکرار کرده: (^)

نویسنده: مجید پارسا
تاریخ: ۱۳۹۳/۰۷/۱۲ ۱۲:۹

با سلام؛ سوالی که وجود داره اینه که با استفاده از repository pattern چطور میتونیم join بزنیم. با توجه به نظرات قبلی توصیه شده است که از خروجی IQueryable نباید برای لایه داده استفاده شود.
در این صورت در هنگام نوشتن دستورات join ابتدا تمامی رکوردهای جداول مورد نظر توسط الگوی repository به حافظه load می‌شود، با توجه به ماهیت linq to object بودن کوئری مورد نظر(join) اجرای برنامه به لحاظ زمانی و مصرف حافظه از

کارایی خوبی برخوردار نخواهد بود.
در این حالت یا می‌باست از خیر کارایی بالاتر گذشت یا از خروجی IQueryable استفاده کرد که در تضاد با پیشنهاد دوستان گرامی می‌باشد.
آیا در این حالت منطقی است joinهای پر استفاده را با خروجی IEnumerable در repository مربوط به خودش نوشت یا راهکار دیگری وجود دارد؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۷/۱۲ ۱۲:۱۹

- الگوی مخزن عمومی (Generic repository pattern)، لایه داده برنامه نیست. زمانیکه از یک ORM استفاده می‌کنید، لایه داده برنامه همان ORM است.
- الگوی مخزن عمومی، عمده‌ی کارش مخفی کردن ساز و کار ORM مورد استفاده از لایه سرویس برنامه است (^).
- اگر از الگوی عمومی مخزن استفاده می‌کنید، سطح دسترسی آن‌را internal تعریف کنید تا محدود شود به لایه سرویس برنامه. داخل لایه سرویس برنامه به هر نحوی که علاقمندید از آن استفاده کنید. نهایتا این لایه سرویس است که خروجی IList یا IEnumerable نهایی را در اختیار مصرف کننده قرار می‌دهد.

نویسنده: مجید پارسا
تاریخ: ۱۳۹۳/۰۷/۱۲ ۱۶:۸

با تشکر، از آنجا که من اولین بار است که به شکل حرفه‌ای برنامه نویسی سه لایه را تجربه می‌کنم با توجه به توضیحات شما این طور متوجه شدم که پیاده سازی کلاس‌های Repository در لایه سرویس صورت گیرد اگر اشتباه نکنم.

در صورت امکان بیشتر موضوع رو باز کنید (منظورم آماتوری تره)
نمونه برنامه‌های سه لایه موجود در اینترنت پیدا کردم در حد CRUD ساده و با استفاده از الگوی مخزن عمومی بوده. مانند مثال‌های سایت asp.net در صورت معرفی نمونه کاملتر و واقعی‌تر ممنون میشوم.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۷/۱۲ ۱۷:۲۹

مراجعه کنید به مسیر راه EF Code first ، انتهای مطلب، قسمت لایه بندی پروژه‌های EF Code first

عنوان: **تزریق وابستگی (Dependency Injection) و توسعه پذیری**
نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۶/۰۲ ۳:۱۹
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

دانستن اینکه چگونه یک نرم افزار با قابلیت نگهداری بالا بنویسیم مهم است ، برای اکثر سیستم‌های سازمانی زمانی که در فاز نگهداری صرف می‌شود بیشتر از زمان فاز توسعه می‌باشد. به عنوان مثال تصور کنید در حال توسعه یک سیستم مالی هستید ، این سیستم احتمالا بین شش ماه تا یک زمان برای توسعه نیاز دارد و بقیه‌ی دوره‌ی پنج ساله صرف نگهداری سیستم خواهد شد. در فاز نگهداری زمان صرف رفع باگ ، افزودن امکانات جدید و یا تغییر عملکرد ویژگی‌های فعلی می‌شود. مهم است که این تغییرات راحت و سریع صورت پذیرد.
اطمینان از اینکه کدها قابلیت نگهداری دارند به توسعه دهندگان احتمالی که در آینده به پروژه اضافه می‌شوند کمک می‌کند سریع کدهای فعلی را درک کنند و مشغول کار شوند. روش‌های زیادی برای افزایش قابلیت نگهداری کدها وجود دارد ، مانند نوشتن آزمون‌های واحد ، شکستن قسمت‌های بزرگ سیستم به قسمت‌های کوچک‌تر و ... در این مورد که ما از یکی از زبان‌های شئ گرا مانند C# استفاده می‌کنیم در حالت معمول کلاس‌ها باید با مسئولیت‌های مستقل و منحصر به فرد طراحی شوند به جای آنکه تمام مسئولیت‌ها از قبیل پردازش ورودی‌های کاربر ، رندر کردن HTML و حتی Query زدن به دیتابیس را به یک کلاس سپرد (مثلا Controller در MVC) باید برای هر مقصود کلاسی مجزا طراحی کرد. با این روش نتیجه اینگونه خواهد بود که می‌توان هر قسمت از عملکرد را بدون نیاز به تغییر بقیه‌ی قسمت‌های Codebase تغییر داد.
در این مطلب قصد داریم به کمک تزریق وابستگی (Dependency Injection) قسمت‌های مستقلتری توسعه دهیم. تکنیک تزریق وابستگی را نمی‌توان در یک مطلب وبلاگ و حتی یک فصل کامل از یک کتاب کامل تشریح کرد ، اگر جستجو کنید کتاب‌ها و آموزش‌های ویدویی زیادی هستند که فقط روی این تکنیک بحث و آموزش دارند. برای بیان مفهوم DI مثالی از یک سیستم ساده‌ی "چاپ اسناد" ارائه می‌کنیم ، این سیستم ممکن است کارهای متفاوتی انجام دهد :
این سیستم ابتدا باید یک سند را تحویل بگیرد ، سپس باید آن را به فرمت قابل چاپ در آورد و در انتها باید عمل اصلی چاپ را انجام دهد. برای اینکه سیستم ما ساختار خوبی داشته باشد می‌توان هر وظیفه را به کلاسی مجزا سپرد :
کلاس Document : این کلاس اطلاعات سندی که قرار است چاپ شود را نگه می‌دارد.
کلاس DocumentRepository : این کلاس وظیفه‌ی بازیابی سند از فایل سیستم (یا هر منبع دیگری) را دارد.
کلاس DocumentFormatter : یک وهله از سند را جهت چاپ آماده می‌کند.
کلاس Printer : مسئولیت ارتباط با سخت افزار Printer را دارد.
کلاس DocumentPrinter : مسئولیت سازماندهی اجزا سیستم را بر عهده دارد.
در این مطلب پیاده سازی بدنه‌ی کلاس‌های بالا اهمیتی ندارد :

```
public class DocumentPrinter
{
  public void PrintDocument(string documentName)
  {
    var repository = new DocumentRepository();
    var formatter = new DocumentFormatter();
    var printer = new Printer();
    var document = repository
      .GetDocumentByName(documentName);
    var formattedDocument = formatter.Format(document);
    printer.Print(formattedDocument);
  }
}
```

همانطور که مشاهده می‌کنید در بدنه‌ی کلاس DocumentPrinter ابتدا وابستگی‌ها نمونه سازی شده اند ، سپس یک سند بر اساس نام دریافت شده و سند پس از آماده شدن به فرمت چاپ به چاپگر ارسال شده است. کلاس DocumentPrinter به تنهایی قادر به چاپ سند نیست و برای انجام این کار نیاز به نمونه سازی همه‌ی وابستگی‌ها دارد .
استفاده از این API اینگونه خواهد بود :

```
var documentPrinter = new DocumentPrinter();
documentPrinter.PrintDocument(@"c:\doc.doc");
```

در حال حاضر کلاس DocumentPrinter از DI استفاده نمی‌کند این کلاس Loosely coupled نیست. به طور مثال لازم است که API سیستم به گونه ای تغییر پیدا کند که سند به جای فایل سیستم از دیتابیس بازیابی شود ، باید کلاس جدیدی به نام DatabaseDocumentRepository تعریف شود و به جای DocumentRepository اصلی در بدنه‌ی DocumentPrinter استفاده شود ، در نتیجه با تغییر دادن یک قسمت از برنامه مجبور به تغییر در قسمت دیگر شده ایم.(tightly coupled است یعنی به دیگر قسمت‌ها چفت شده است.)

DI به ما کمک می‌کند که این چفت شدگی (coupling) را از بین ببریم.

استفاده از constructor injection:

اولین قدم برای از بین بردن این چفت شدگی Refactor کردن کلاس DocumentPrinter هست ، پس از این Refactoring وظیفه‌ی وهله سازی مستقیم اشیاء از این کلاس گرفته می‌شود و نیازمندی‌های این کلاس از طریق سازنده به این کلاس تزریق می‌شود و فیلدهای کلاس نگهداری می‌شود . به کد زیر توجه کنید :

```
public class DocumentPrinter
{
  private DocumentRepository _repository;
  private DocumentFormatter _formatter;
  private Printer _printer;
  public DocumentPrinter(
    DocumentRepository repository,
    DocumentFormatter formatter,
    Printer printer)
  {
    _repository = repository;
    _formatter = formatter;
    _printer = printer;
  }
  public void PrintDocument(string documentName)
  {
    var document = _repository.GetDocumentByName(documentName);
    var formattedDocument = _formatter.Format(document);
    _printer.Print(formattedDocument);
  }
}
```

اکنون برای استفاده از این کلاس باید نیازمندی هایش را قبل از ارسال به سازنده نمونه سازی کرد :

```
var repository = new DocumentRepository();
var formatter = new DocumentFormatter();
var printer = new Printer();
var documentPrinter = new DocumentPrinter(repository, formatter, printer);
documentPrinter.PrintDocument(@"c:\doc.doc");
```

بله هنوز طراحی خوبی نیست اما این یک مثال ساده از DI می‌باشد. هنوز مشکلاتی در این طراحی هست ، به طور مثال کلاس DocumentPrinter به یک پیاده سازی مشخص از وابستگی هایش چفت شده است. (هنوز برای استفاده از DatabaseDocumentRepository باید DocumentPrinter را تغییر داد) پس این طراحی هنوز انعطاف پذیر نیست و نمی‌توان به سادگی برای آن آزمون واحد نوشت.

برای حل این مشکلات از Interfaceها کمک می‌گیریم. اگر به مثال قبلی بازگردیم نگرانی هر دو کلاس DocumentRepository و DatabaseDocumentRepository دریافت سند می‌باشد ، تنها پیاده سازی تفاوت دارد ، پس می‌توان یک Interface تعریف کرد

```
public interface IDocumentRepository
{
  Document GetDocumentByName(string documentName);
}
```

حال ما 2 کلاس داریم که هر دو یک Interface را پیاده سازی کرده اند می‌توان این کار را برای بقیه‌ی وابستگی‌های کلاس DocumentPrinter نیز انجام داد ، حالا باید DocumentPrinter را به گونه ای Refactor کنیم که وابستگی‌ها را بر اساس Interface دریافت کند :

```
public class DocumentPrinter
{
  private IDocumentRepository _repository;
```

```
  private IDocumentFormatter _formatter;
  private IPrinter _printer;
  public DocumentPrinter(
    IDocumentRepository repository,
    IDocumentFormatter formatter,
    IPrinter printer)
  {
    _repository = repository;
    _formatter = formatter;
    _printer = printer;
  }
  public void PrintDocument(string documentName)
  {
    var document = _repository.GetDocumentByName(documentName);
    var formattedDocument = _formatter.Format(document);
    _printer.Print(formattedDocument);
  }
}
```

حالا به سادگی می‌توان پیاده سازی‌های متفاوتی را از وابستگی‌های DocumentPrinter انجام داد و به آن تزریق کرد. همچنین اکنون نوشتن آزمون واحد هم ممکن شده است ، می‌توان یک پیاده سازی جعلی از هر کدام از Interfaceها انجام داد و جهت اهداف Unit testing از آن استفاده کرد. به طور مثال می‌توان یک پیاده سازی جعلی از IPrinter انجام داد و بدون نیاز به ارسال صفحه به پرینتر عملکرد سیستم را تست کرد.
با وجودی که موفق شدیم چفت شدگی میان DocumentPrinter و وابستگی هایش را از بین ببریم اما اکنون استفاده از آن پیچیده شده است ، هربار که قصد نمونه سازی شیء را داریم باید به یاد آوریم کدام پیاده سازی از Interface مورد نیاز است ؟ این پروسه را می‌توان به کمک یک DI Container اتوماسیون کرد.
DI Container یک Factory هوشمند است ، مانند بقیه‌ی کلاس‌های Factory وظیفه‌ی نمونه سازی اشیاء را بر عهده دارد.
هوشمندی آن در اینجا هست که می‌داند چطور وابستگی‌ها را نمونه سازی کند . DI Containerهای زیادی برای .NET وجود دارند یکی از محبوب‌ترین آنها StructureMap می‌باشد که قبلا در سایت درباره آن صحبت شده است .

برای مثال جاری پس از افزودن StructureMap به پروژه کافی است در ابتدای شروع برنامه به آن بگوییم برای هر Interface کدام شیء را وهله سازی کند :

```
ObjectFactory.Configure(cfg =>
{
  cfg.For<IDocumentRepository>().Use<FilesystemDocumentRepository>();
  cfg.For<IDocumentFormatter>().Use<DocumentFormatter>();
  cfg.For<IPrinter>().Use<Printer>();
});
```

نظرات خوانندگان

نویسنده: بهروز راد
تاریخ: ۱۳۹۱/۰۶/۰۲ ۹:۴۶

برادر، بسیار خوب و روان توضیح دادی. از معدود مقالات فارسی بود که از خواندنش لذت بردم.

موفق باشی.

نویسنده: محسن
تاریخ: ۱۳۹۱/۰۶/۰۲ ۱۰:۴۶

خیلی جالب بود . بخصوص قسمت DI Container .

نویسنده: مجتبی حسینی
تاریخ: ۱۳۹۱/۰۶/۰۲ ۲۲:۵۸

بسیار شیوا و رسا بود.
تاکید بر این نکته نیز خالی از لطف نیست که با توجه به مطلب خط آخر به جای مثلاً:

```
IDocumentFormatter documentformatter = new DocumentFormatter();
```

باید نوشت:

```
IDocumentFormatter documentformatter = ObjectFactory.GetInstance<IDocumentFormatter>();
```

نویسنده: Alex
تاریخ: ۱۳۹۱/۰۶/۰۳ ۱۴:۵۴

واقعن ساده و روان توضیحش دادین. البته Spring.net هم یکی از موارد خوبی هستش که میشه برای DI استفاده کرد.

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۶/۰۳ ۲۰:۱۶

مرسی. واقعا عالیه
موفق باشید

نویسنده: ایلیا اکبری فرد
تاریخ: ۱۳۹۱/۰۶/۱۳ ۱۹:۳۵

عالی بود . عالی. در صورت به مقالاتی که در این زمینه هست بیشتر بپردازین .

نویسنده: مسعود رمضانی
تاریخ: ۱۳۹۱/۰۸/۲۰ ۱۶:۳۳

با سلام و خسته نباشی

بابت مطلب خوبتون تشکر میکنم.

نویسنده: بهار قدمی
تاریخ: ۱۳۹۲/۰۴/۱۵ ۱۲:۲۰

با سلام و عرض خسته نباشید
آیا در همه جای پروژه باید ازاین روش استفاده کرد.منظورم استفاده از یک DI Container است.آیا anti - pattern ای در این مورد هم وجود دارد؟مثلا من یک پروژه‌ی ماژوالار بزرگ دارم آیا فقط در قسمت اتصال کلاس‌ها به لایه‌ی UI یا همون MVC از DI Container استفاده کنم یا هیچ جای پروژه ام new() نداشته باشم و همیشه از DI Container اسم کلاسم رو بگیرم؟ با توجه به بزرگی پروژه ام آیا Performance از دست نمیدم؟

عنوان: **روش نامگذاری Smurf ایی!**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۶/۱۱ ۱۱:۱
آدرس: www.dotnettips.info
برچسب‌ها: C#, Design patterns, Naming

اگر به یک سری از کتابخانه‌ها دقت کنید، تمام کلاس‌های آن‌ها دارای یک پیشوند تکراری هستند؛ مثلا **Smurf** XMLDataRow، **Smurf** XMLElement و الی آخر در مورد تمام کلاس‌های موجود در پروژه. به این رویه « Smurf Naming Convention » گفته می‌شود! در این نوع کتابخانه‌ها زمانیکه کاربری بر روی دکمه‌ای کلیک می‌کند، **Smurf** AccountView اطلاعات **Smurf** AccountDTO را به **Smurf** AccountController منتقل می‌کند. در ادامه از خاصیت **Smurf** ID دریافتی، مقدار **Smurf** OrderHistory دریافت شده و به **Smurf** HistoryReportingView جهت نمایش ارسال خواهد شد. اگر استثنای **Smurf** ErrorEvent رخ دهد، توسط **Smurf** ErrorLogger در فایلی به نام **smurf** / **smurf** log.log /log ثبت خواهد شد.

کلمه Smurf هم از شخصیتی کارتونی به همین نام اخذ شده است که در زبان مخصوص آن‌ها اکثر افعال و نام‌ها از کلمه Smurf مشتق می‌شود! برای مثال در مورد ماهیگیری کردن در یک رودخانه عنوان می‌کنند «We're going **smurfing** on the River **Smurf** today».

خوب، چکار باید کرد؟ روش صحیح معرفی نام یک شرکت در حین طراحی و نامگذاری کلاس‌های یک کتابخانه چیست؟
در مطلب بسیار جامع و عالی « اصول و قراردادهای نام‌گذاری در دات‌نت » عنوان شده است که اساس نام‌گذاری فضاهای نام باید از قاعده زیر پیروی کند:

```
<Company>.<Technology|Produt|Project>[.<Feature>][.<SubNamespace>]
```

مثلا مایکروسافت یکبار فضای نام Microsoft.Reporting.WebForms را تعریف کرده است و ... همین! دیگر به ابتدای هر کلاسی در این کتابخانه، پیشوند Microsoft یا MS و امثال آن اضافه نشده است تا بر روی اعصاب و روان استفاده کننده تاثیر منفی داشته باشد.

نظرات خوانندگان

نویسنده: رحمت اله رضایی
تاریخ: ۱۳۹۱/۰۶/۱۱ ۱۸:۳۸

مشکل اینجاست که در پروژه ای، کلاسهایی هم اسم کلاسهای دات نت داشته باشیم. همیشه باید فضای نام را در ابتدای کلاسها نوشت.
مثلا فرض کنید کنترلهای TextBox و Button و ... را در یک پروژه وب فرم یا ویندوز فرم سفارشی کرده باشیم. در این حالت چکار باید بکنیم؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۱/۰۶/۱۱ ۱۹:۲۰

- برنامه FxCop می‌تونه اسمبلی‌های شما رو آنالیز کنه و دقیقا گزارش بده که چه مواردی هم نام کلاس‌های پایه دات نت هستند و بهتر است تغییر نام پیدا کنند. بنابراین به این صورت می‌تونید خیلی سریع حجم بالایی از کدها رو بررسی و رفع اشکال کنید.
- به علاوه زمانیکه طراح شما هستید، محدودیتی در نامگذاری نهایی وجود ندارد. مثلا نام کلاس مشتق شده را NumericTextBox قرار دهید و مواردی مانند این که بیانگر عملکرد سفارشی و ویژه کلاس مشتق شده جدید هستند:

```
public class RequiredTextBox : TextBox
```

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

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

برای مثال فرض کنید مدلی به شکل زیر داریم

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

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

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

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

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

فرض کنید پیاده سازی توابع به شکل زیر باشه

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

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

نحوه استفاده

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

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

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

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

توابع رو داخل یک دیکشنری ذخیره میکنیم

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

نحوه استفاده

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

عنوان: **معماری لایه بندی نرم افزار #1**
نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۱/۱۲/۲۸ ۱۸:۵
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, SoC, Separation of Concerns, ASP.Net, Domain Driven Design, DDD, SOLID Principals, C#, MVC, WPF, N-Layer Architecture

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 مطمئنا 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)

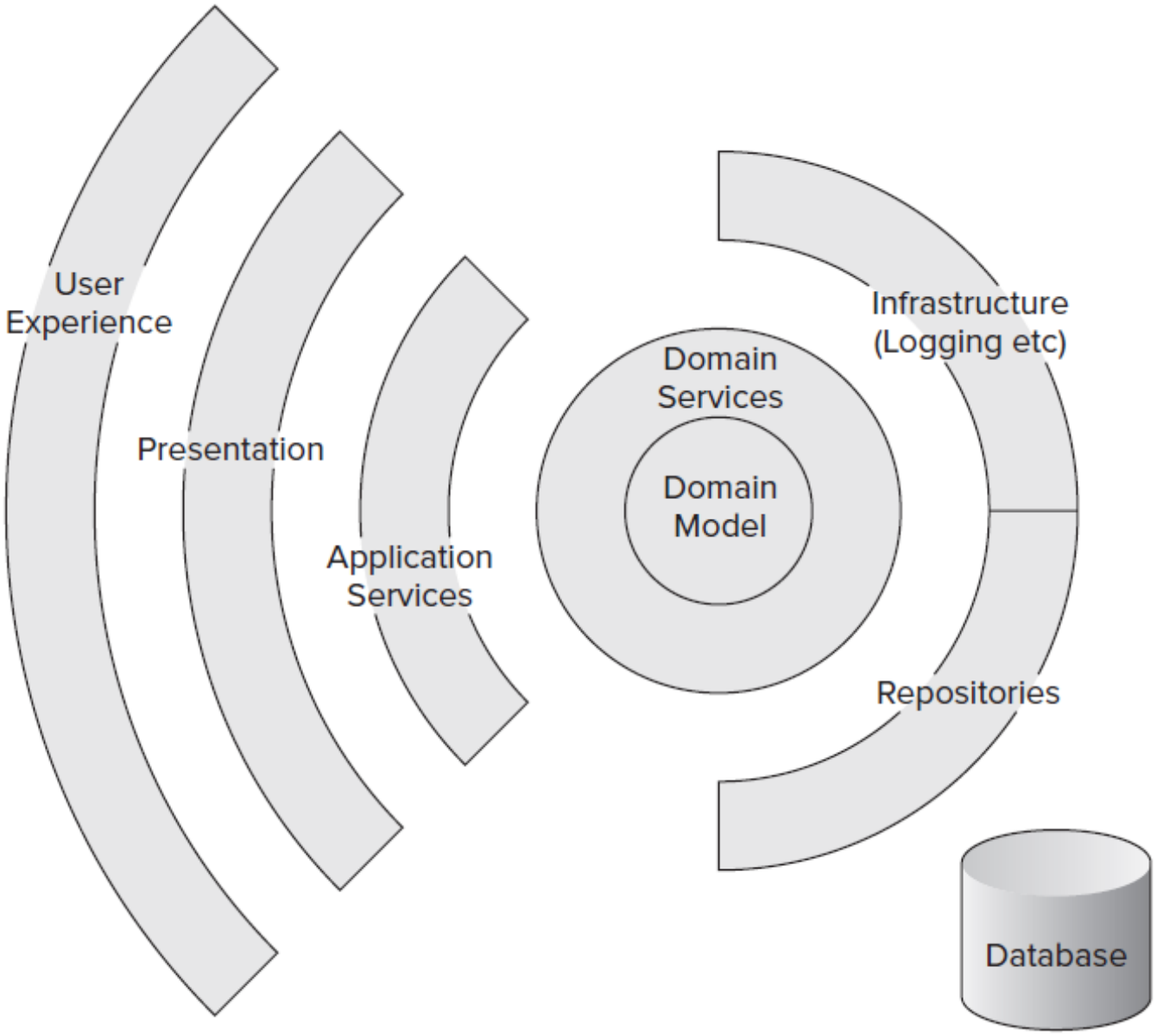
با استفاده از Visual Studio یا به اختصار VS ، می‌توانید برنامه‌های کاربردی را به راحتی تولید نمایید. طراحی رابط کاربری به آسانی عمل کشیدن و رها کردن (Drag & Drop) کنترل‌ها بر روی رابط کاربری قابل انجام است. همچنین در پشت رابط کاربری (Code Behind) تمامی عملیات مربوط به مدیریت رویدادها، دسترسی به داده ها، 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 New Project > Add 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.Model و SoCPatterns.Layered.Repository ارجاع دهید.

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

1. UI
§ SoCPatterns.Layered.WebUI

2. Presentation Layer
§ SoCPatterns.Layered.Presentation

3. Service Layer
§ SoCPatterns.Layered.Service

4. Domain Layer
§ SoCPatterns.Layered.Model

5. Data Layer
§ SoCPatterns.Layered.Repository

توجه داشته باشید که پوشه بندی برای مرتب سازی لایه‌ها و دسترسی راحت‌تر به آنها می‌باشد.

پیاده سازی ساختار لایه بندی برنامه به صورت کامل انجام شد. حال به پیاده سازی کدهای مربوط به هر یک از لایه‌ها و بخش‌ها می‌پردازیم و از لایه Domain شروع خواهیم کرد.

نظرات خوانندگان

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

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: ۱۳۹۱/۱۲/۲۸ ۲۲:۵

آیا صرفا 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: ۱۳۹۱/۱۲/۲۹ ۰:۵۳

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 مرزبندی لایه‌ها هم صحبت می‌کنم و مفهوم هر کدام را دقیقا توضیح میدم.

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

اگر مقاله فوق رو با دقت بخونید متوجه میشید 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 پوشه ها، فضای نام و یا پروژه‌ها از هم جدا کنید

نویسنده: حسن
تاریخ: ۱۳۹۱/۱۲/۲۹ ۱۰:۱۴

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 صرفا تعریف چند ماژول جدید می‌تواند ضمانتی باشد بر رفع مشکل code behind یا اینکه با این ماژول‌ها هم نهایتا همان مشکل قبل پابرجا است یا می‌تواند پابرجا باشد.

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

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

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
```

Application.Common

با تشکر

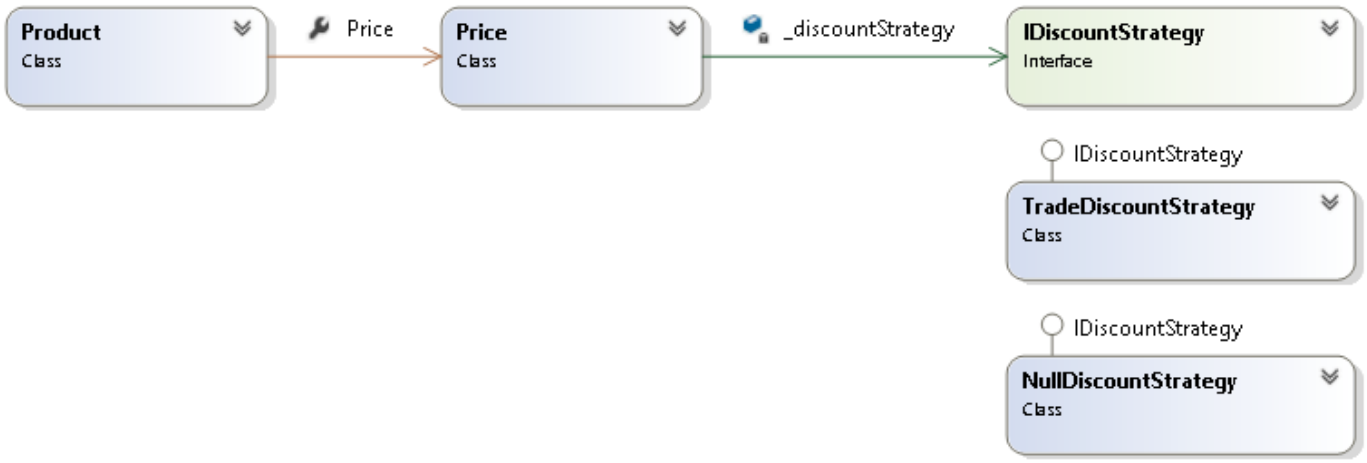
عنوان: **معماری لایه بندی نرم افزار #2**
نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۱/۱۲/۳۰ ۱:۴۵
آدرس: www.dotnettips.info
برچسب‌ها: ASP.Net, C#, Design patterns, MVC, WPF, SoC, Separation of Concerns, Domain Driven Design, DDD, SOLID Principals, N-Layer Architecture

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 خواهم 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 خواهم کرد. استفاده از این الگو نیز به این دلیل بود که این الگو مختص الگوریتم هایی 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 خواهم 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 خواهم 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 اتباط 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 خواهم کرد.

لایه‌ی سرویس با برقراری ارتباط با منبع داده ای، داده‌های مورد نیاز خود را بر می‌گرداند. برای این منظور از الگوی Repository استفاده می‌کنیم. از آنجایی که لایه Model باید مستقل از منبع داده ای عمل کند و نیازی به شناسایی نوع منبع داده ای ندارد، جهت پیاده سازی 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 انباره ای از موجودیت‌های تجاری عمل می‌کند که نسبت به زیر بنای ساختاری منبع داده ای کاملا مستقل می‌باشد.

کلاس سرویس باید بتواند استراتژی تخفیف را بر روی مجموعه ای 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 پیاده سازی واقعی کاملا نامطلع می‌باشد و می‌تواند برنامه نویس را به سمت Abstraction و Dependency Inversion به جای پیاده سازی 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 داده ای خاصی وابسته نیست و به جای پیاده سازی کدهای منبع داده ای، از Interface ها به منظور برقراری ارتباط با پایگاه داده استفاده شده است. پیاده سازی کدهای منبع داده ای را به لایه‌ی Repository واگذار نمودیم که در بخش‌های بعدی نحوه پیاده سازی آن را مشاهده خواهید کرد. این امر موجب می‌شود تا لایه Model درگیر پیچیدگی‌ها و کد نویسی‌های منبع داده ای 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. اصلا جای نگرانی نیست، چون در مباحث بعدی به صورت مفصل در مورد آنها صحبت خواهم 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 پیاده سازی 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 معماری‌ها و الگوها و بهترین‌های 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 میخواهم که من رو راهنمایی کنید که چه مقدماتی در این زمینه‌ها نیاز دارم
باز هم ممنون.

نویسنده: علي
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 می‌تونید پیگیری کنید.

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

فشردگی این مباحث بخاطر این بود که میخواستم فعلا یک نمونه پروژه رو آموزش بودم تا یک شمای کلی از کاری که می‌خواهیم انجام بدیم رو ببینید. در مباحث بعدی این مباحث رو بازتر می‌کنم. خود من برای مطالعه و جمع بندی این مباحث منابع زیادی رو مطالعه کردم. واقعا برای بعضی مباحث نمیشه 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 کلاس‌های 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 IList<Product> است. اگر این متد تعریف نمی‌شد شما باید در کلاس سرویس حلقه foreach رو قرار می‌دادید. البته با این حال در قسمت هایی از طراحی کلاسها که الگوهای طراحی را زیر سوال نمی‌برد و تست پذیری را دچار مشکل نمی‌کند، طراحی سلیقه ای است. مقاله من هم آیه‌ی نازل شده نیست که دستخوش تغییرات نشود. شما می‌توانید با سلیقه و دید فنی خود تغییرات مورد نظر رو اعمال کنید. ولی اگر نظر من را بخواهید این طراحی مناسب‌تر 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: ۱۳۹۲/۰۲/۲۹ ۱۷:۱۰

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: ۱۳۹۲/۰۲/۳۰ ۰:۴۰

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

عنوان: **معماری لایه بندی نرم افزار #3**
نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۲/۰۱/۰۲ ۱۷:۵۵
آدرس: www.dotnettips.info
برچسب‌ها: ASP.Net, C#, Design patterns, MVC, WPF, SoC, Separation of Concerns, Domain Driven Design, DDD, SOLID Principals, N-Layer Architecture

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 داده ای 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 خواهم کرد.

الگوی Facade یک Interface ساده را به منظور کنترل دسترسی به مجموعه ای از Interface ها و زیر سیستم‌های پیچیده ارائه می‌کند. در مباحث بعدی در مورد آن بیشتر صحبت خواهم 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:

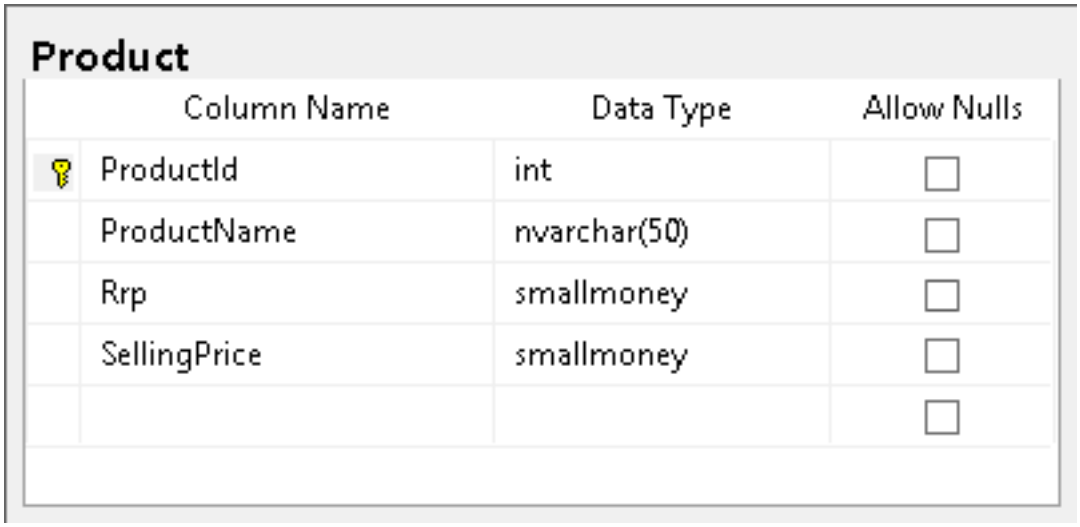
```
public class ProductService
{
    private Model.ProductService _productService;
    public ProductService(Model.ProductService ProductService)
    {
        _productService = ProductService;
    }
    public ProductListResponse GetAllProductsFor(
        ProductListRequest productListRequest)
    {
        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.

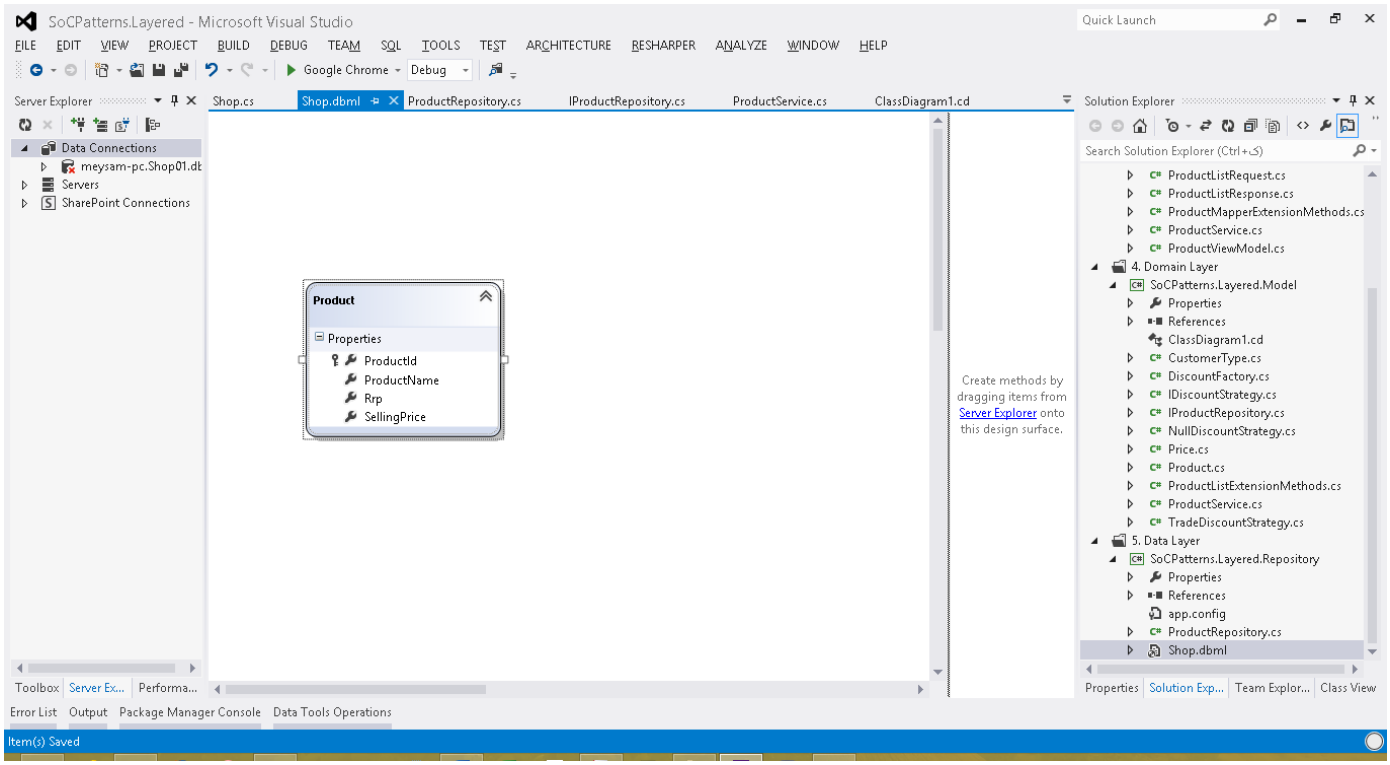
Data Layer

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



برای اینکه کدهای بانک اطلاعاتی را سریعتر تولید کنیم از روش Linq to SQL در Data Layer استفاده می‌کنم. 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 لایه‌ی Data هم به پایان رسید و در ادامه به کدنویسی لایه‌ی 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 خواهم 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 واکشی داده ها، 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: ۱۳۹۲/۰۱/۰۲ ۱۸:۲۹

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 مثلا 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:
- مثلا 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 اصلا 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: ۱۳۹۲/۰۱/۰۲ ۲۳:۳۵

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 اشیا Request و Response هم باید خدمتتان عرض کنم که برای درخواست و پاسخ به درخواست استفاده می‌شوند که چون پروژه ای که مثال زدم کوچک بوده ممکنه کاملا 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 کاملا 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 بعدا صحبت میکنم.

نویسنده: ایلیا
تاریخ: ۱۳۹۲/۰۱/۰۳ ۰:۴۳

آقای خوشبخت خداقوت.

مرسی از مطالب خوبتون.

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

لطفا برای اینکه نظرات حالت فنی‌تر و غنای بیشتری پیدا کنند، از ارسال پیام‌های 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 ۱:۰

- من در عمل تفاوتی بین لایه مخزن و سرویس شما مشاهده نمی‌کنم. یعنی لایه مخزن داره GetAll می‌کنه، بعد لایه سرویس هم داره همون رو به یک شکل دیگری بر می‌گردونه. 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 می‌کنند. فرضشون هم این است که این روش ما رو از تماس مستقیم با ORM برحذر می‌داره (شاید فکر می‌کنند ایدز می‌گیرند اگر مستقیم کار کنند!). ولی عرض کردم این روکش در واقعیت فقط شاید با EF یا L2S قابل تعویض باشه نه با ORMهای دیگر با روش‌های 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 کوئری‌ها یا روش‌های 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 می‌کنید.
-پیشنهادی که من دارم اینه که لایه‌ی 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 همه‌ی منطق‌ها مانند GetBlaBla به Service منتقل می‌شود.
-یک پیشنهاد دیگر اینکه استفاده از کلمه‌ی New در Presentation Layer را به حداقل رساند و همه جا نیاز مندی‌ها را به صورت وابستگی به کلاس‌های استفاده کننده تزریق شود تا در زمان نوشتن تست‌ها همه‌ی 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 می‌کنین 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: ۱۳۹۲/۰۱/۱۶ ۵:۲۹

محسن جان، چیزی که من از این الگو در مورد واکشی 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 میدند.

بعد

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

بنابراین
کار در **"لایه مخزن"** روی **"مدل ها"** انجام میگیره
و
کار در **"لایه سرویس"** روی **"ویـــومدل ها"** 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: ۱۳۹۲/۰۱/۱۶ ۵:۴۹

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: ۱۳۹۲/۰۱/۱۶ ۹:۳

- 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 مثلا 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 DAL شما اسمش رو گذشتید 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 مثلا یک نمونه: ORMها لایه‌ای دارند به نام سطح اول کش که مثلا 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 مثلا 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 نهایتا 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 یه 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 دقیقا 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. مثلا 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 ویـــو مدل تبدیل بشند و در این الگو این کار در لایه سرویس انجام میشه.

نویسنده: محسن
تاریخ: ۱۳۹۲/۰۱/۱۶ ۱۷:۱۰

- پیاده سازی الگوی مخزن در عمل (بر اساس بحث فعلی که در مورد کار با ORMها 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 مجددا یک پوسته 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.

عنوان: **معماری لایه بندی نرم افزار #4**
نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۲/۰۱/۰۳ ۱۲:۲۵
آدرس: www.dotnettips.info
گروه‌ها: ASP.Net, C#, Design patterns, MVC, WPF, SoC, Separation of Concerns, Domain Driven Design, DDD, SOLID Principals, N-Layer Architecture

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 لایه‌ی 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 سازنده‌ی کلاسهایی را که از الگوی Dependency Injection استفاده نموده اند و قطعا 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 NuGetقسمت اول

2. آشنایی با NuGetقسمت 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:

در WebUI :

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>
```

در WinUI :

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:

در WebUI :

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);
            }
        }
    }
}
```

در WinUI :

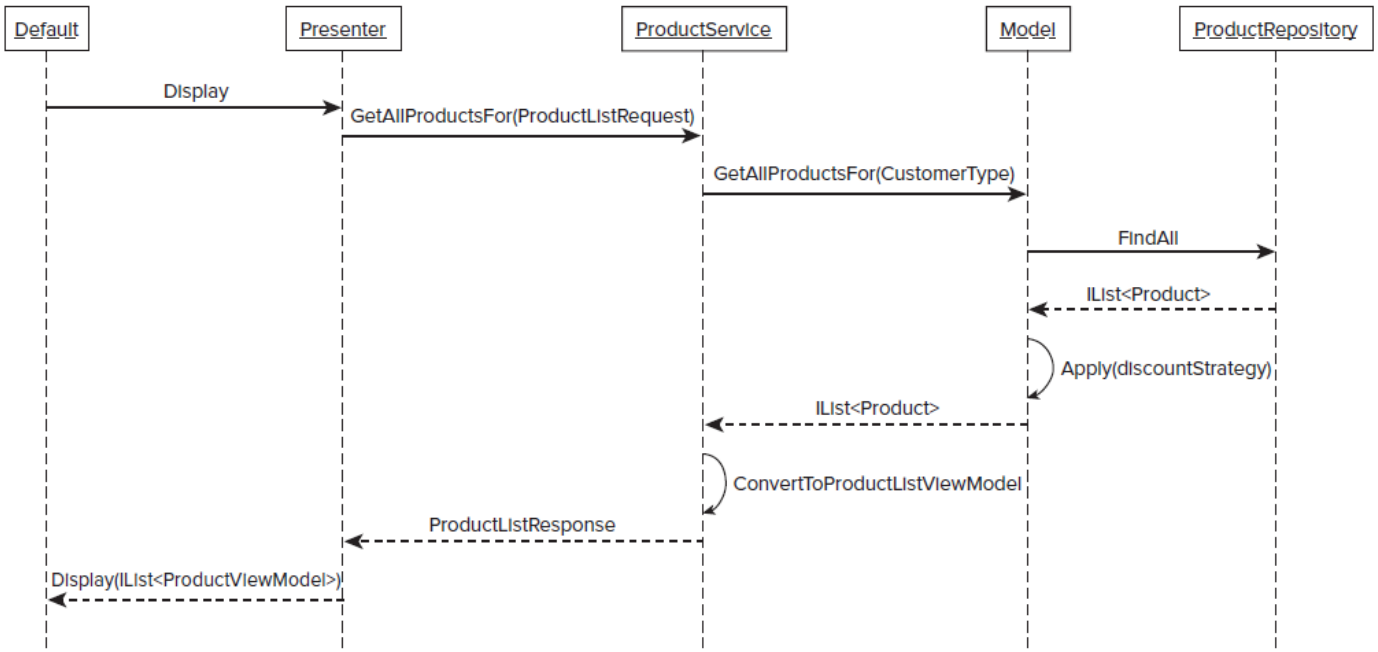
وارد کدنویسی Form1 شوید و کد آن را بصورت زیر تغییر دهید:

```
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 خواهم کرد. پیاده سازی معماری لایه بندی در اینجا به پایان رسید.

اما اصلا 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 Applicaion 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: ۱۳۹۲/۰۱/۰۳ ۱۴:۲۹

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: ۱۳۹۲/۰۱/۰۳ ۱۵:۳۱

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

نویسنده: حامد
تاریخ: ۱۳۹۲/۰۱/۰۳ ۱۶:۳۶

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

نویسنده: محسن
تاریخ: ۱۳۹۲/۰۱/۰۳ ۲۳:۵۹

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: ۱۳۹۲/۰۱/۰۴ ۱:۲۰

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

نویسنده: M.Q
تاریخ: ۱۳۹۲/۰۱/۰۴ ۲۲:۱۵

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: ۱۳۹۲/۰۱/۰۵ ۰:۳۳

متد auditFields مطرح شده در مطلب ردیابی اطلاعات این سایت برای مقصود شما مناسب است.

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

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: علي
تاریخ: ۱۳۹۲/۰۱/۱۲ ۱۶:۳

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

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

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

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

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 و.. وجود نداره سوال من چیز دیگه بود دوستان

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

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: ۱۳۹۲/۰۱/۱۲ ۱۹:۲۶

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 کاربردش از دست میده توی پست‌های قبل هم از آقای خوشبخت سوال کردم اما طاهرا 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 DbSetها. 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: ۱۳۹۲/۰۱/۱۲ ۲۰:۴۶

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: ۱۳۹۲/۰۱/۱۲ ۲۱:۳

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 MVC , EF 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: ۱۳۹۲/۰۱/۱۲ ۲۱:۷

خواهش می‌کنم.
فقط جهت یادآوری مثال روش آقای نصیری با پوشش MVC و EF قابل دریافت است.

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

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: ۱۳۹۲/۰۱/۱۹ ۱۵:۵۳

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 وطراحی کینم 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 یادگیریش خیلی بالاتر بود تا گرفتن فایل آماده...
درضمن لازمه بگم که بخاطر رفع شک وشبهه در سرعت پردازش وبالا 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 فرمم قرار دادم که بیشتر به اهمیت لود اطلاعاتم در پروژه و فرمهای واقعی پی ببرم.
سئوال اینکه :

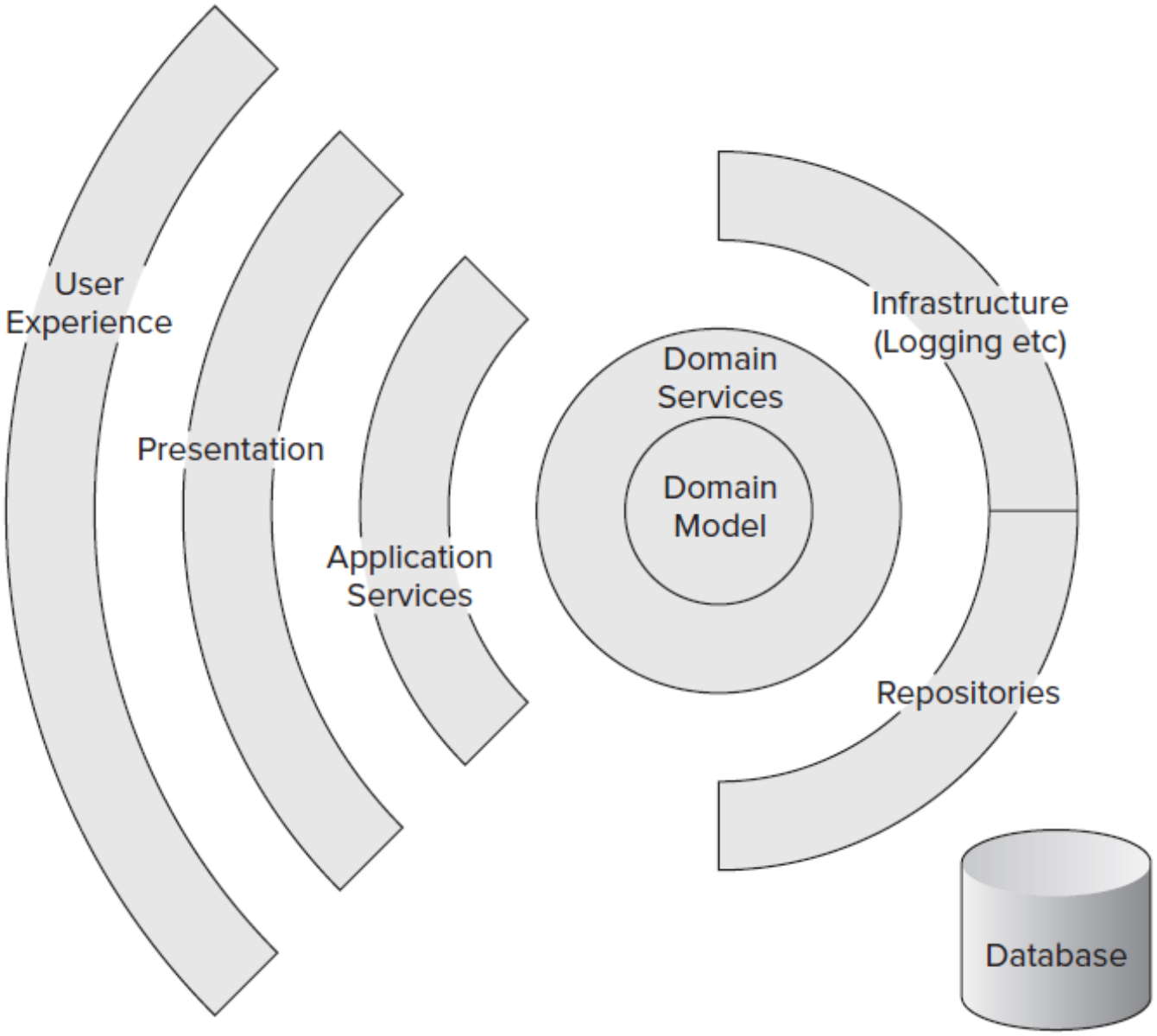
به نظر شما ما می‌تونیم روی این لایه‌ها الگوی واحد کار رو هم ایجاد کنیم یا نه؟ اصلا ضرورتی داره ؟

نویسنده: علیرضا کیانی مقدم
تاریخ: ۱۳۹۲/۰۱/۲۸ ۱۲:۸

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 مسئله ای بغرنج تبدیل نشده و تکرر داده‌ها و پاس دادن غیر ضرور آنها را موجب نشود تلاش در این باره مفید خواهد بود .
امروزه توسعه دهنده گان به سمت کم کردن لایه‌های فرسایشی و حذف پیچیدگی‌های غیر ضرور قدم بر می‌دارند. خلق عبارات لامبادا در دات نت و delegate ها نمونه هایی از تلاش بشر برنامه نویس در این باره است .

نویسنده: مسعود2
تاریخ: ۱۳۹۲/۰۲/۰۹ ۹:۱۲

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



منظور از DomainService چیست؟

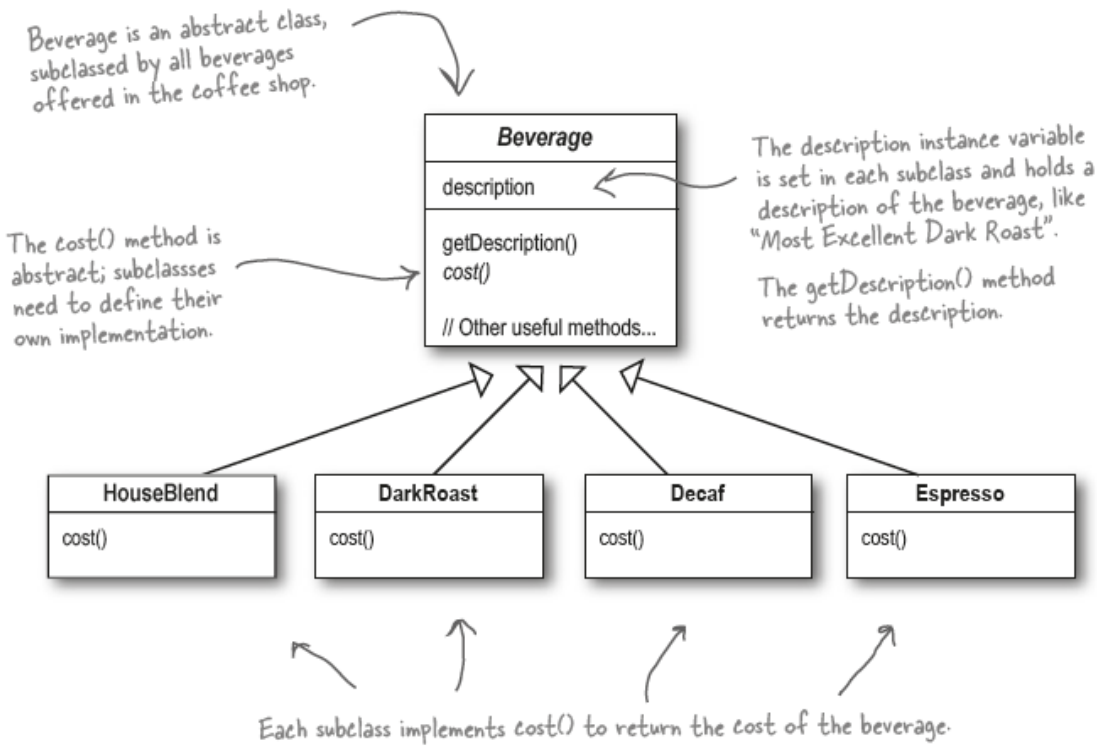
ممکنه منابع بیشتری معرفی نمایید؟
ممنون.

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

منبع برای مطالعه بیشتر

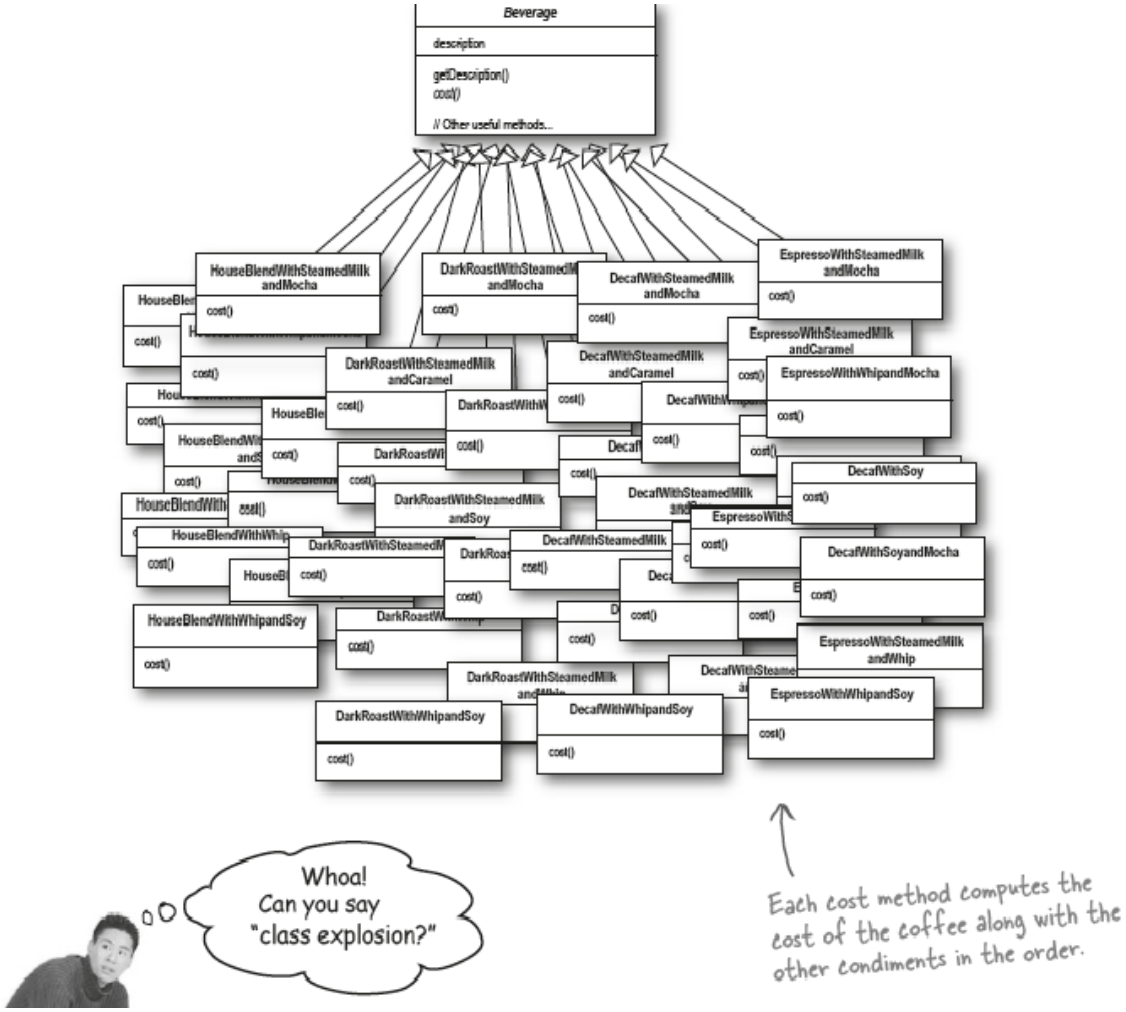
عنوان: معرفی Decorator Design Pattern
نویسنده: حامد صمدی
تاریخ: ۱۳۹۲/۰۱/۲۳ ۸:۳۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

فرض کنید که می‌خواهیم یک برنامه برای یک فروشگاه نوشیدنی (مانند coffee shop) بنویسیم ، این فروشگاه در ابتدای کار ممکن است ، منوی ساده‌ای جهت ارائه به مشتری داشته باشد. برای مثال ممکن است که فقط 3 یا 4 محصول داشته باشد.
بنابراین ممکن است ما برنامه‌ای را که می‌خواهیم برای این مشتری بنویسیم به صورت زیر طراحی کنیم:

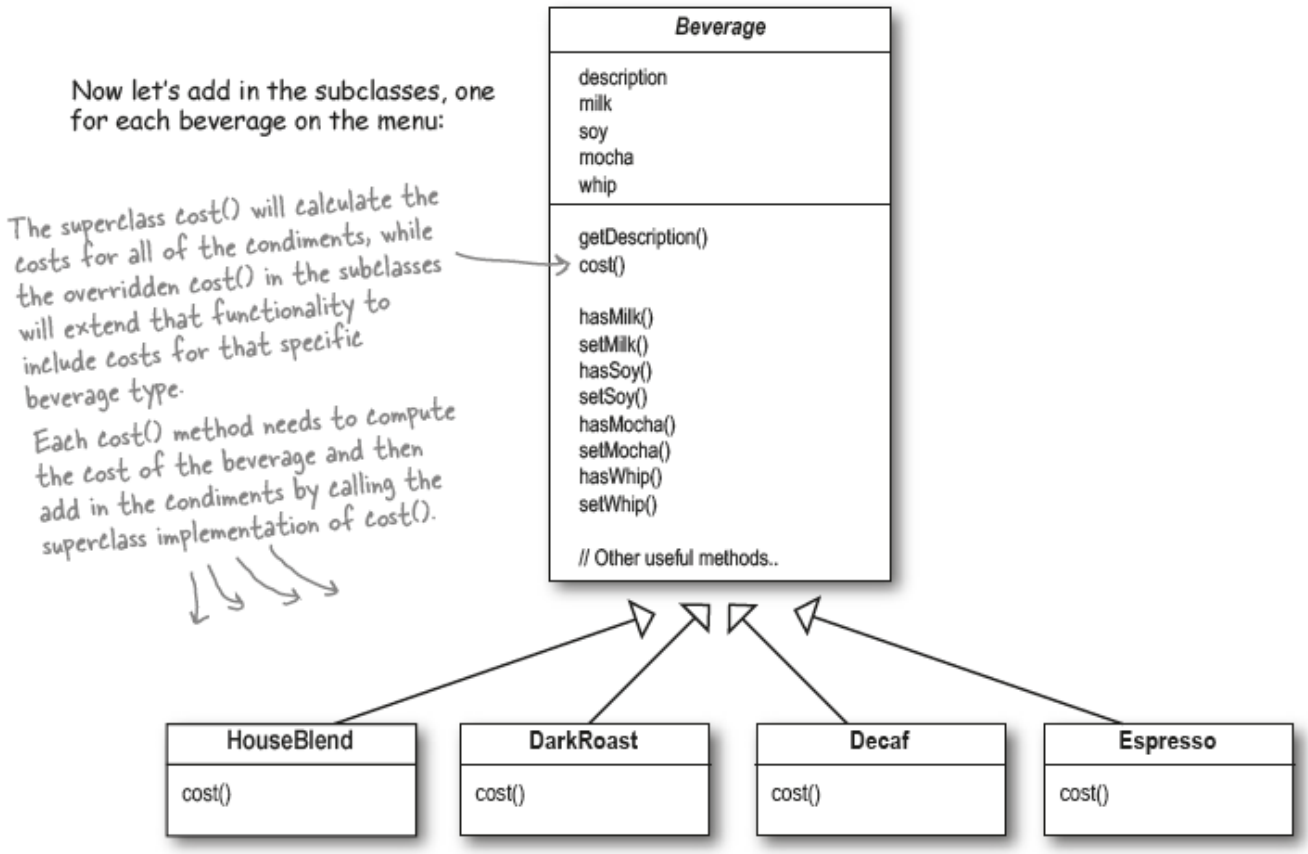


که بسیار طبیعی و درست می‌باشد. اما حالا در نظر بگیرید که این فروشگاه در آینده ممکن است محصولات خود را افزایش بدهد و یا حالاتی که ممکن است این محصولات با هم ادغام شوند را در نظر بگیرید. برای مثال ممکن است شما بخواهید که قهوه‌تان را با شیر نوش جان کنید و یا ....

بنابراین تعداد این حالات را در نظر بگیرید که در آینده ممکن است چقدر زیاد بشود:



خوب پس چه‌کاری ما می‌توانیم برای نگهداری این برنامه انجام بدهیم؟ یکی از راه‌هایی که ممکن است به فکر ما برسد این است که روش بالا روش احمقانه ای است. چرا ما باید به همه‌ی این کلاس‌ها نیاز داشته باشیم. ما می‌توانیم که چاشنی‌ها را در کلاس اصلی نگه‌داری کنیم و کلاس محصولاتمان را از کلاس اصلی به ارث ببریم اجازه دهید تا این کار را با هم انجام بدهیم



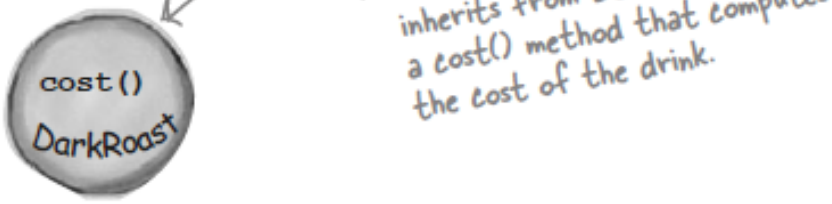
خوب با این روش ما n کلاس تشکیل شده در رویکرد اول را فقط به 5 کلاس تبدیل کردیم. خوب این روشی بسیار ایده‌ال به نظر می‌رسد. اما ممکن است در آینده که تعداد چاشنی‌های ما بالا می‌رود و همچنین تعداد محصولاتمان نیز ممکن است بیشتر شود مجبور شویم که تعداد این کلاس‌ها را بیشتر کنیم، و یا فکر کنید که ما می‌خواهیم هریک از چاشنی‌هایمان، یک قیمت را نسبت بدهیم. بنابراین مجبوریم که تمامی این‌ها را در کلاس پایه اضافه کنیم؛ بله درست است، ما با کلاس پایه‌ی حجیمی روبرو می‌شویم که بیشتر خواص و یا متدهای آن برای زیر کلاس‌های دیگر مناسبت نیستند. خوب آیا روش بهتری برای جلوگیری از این مشکل داریم؟ بلی.

خوب ما به این مسئله به این صورت نگاه می‌کنیم که شروع می‌کنیم با نوشیدنی‌ها و آن‌ها را با چاشنی‌ها در زمان اجرا تزیین (Decorate) می‌کنیم؛ نه کامپایل.

برای مثال اگر مشتری ما یک نوشیدنی DarkRoast با Mocha و Whip خواست، سپس ما :
-1 یک شی از DarkRoast ایجاد می‌کنیم .
-2 آن را با یک شی از Mocha تزئین می‌کنیم.
-3 آن را با یک شی از Whip - تزیین می‌کنیم.
-4 متد Cost() را صدا می‌زنیم و یک Delegation را برای اضافه کردن قیمت چاشنی‌ها در نظر می‌گیریم.

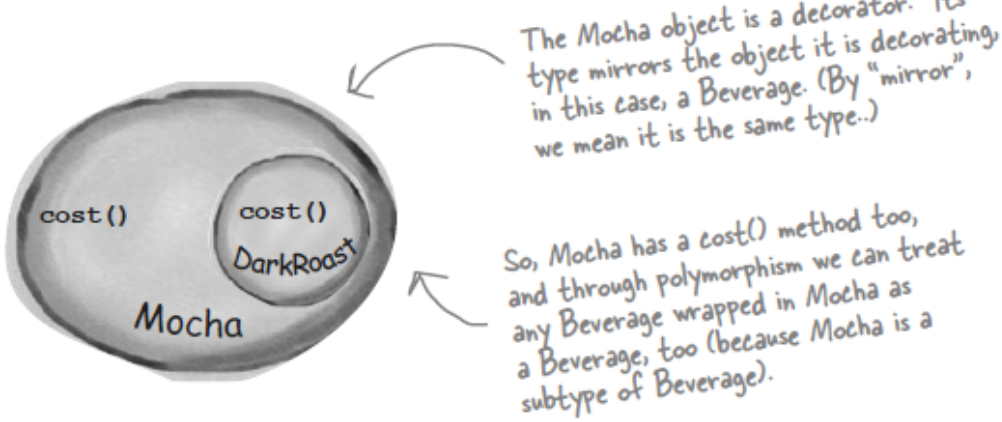
بسیار خوب؛ اما ما عملیات تزئین یک شی را چگونه انجام می‌دهیم و delegation ما چگونه عمل می‌کند .
یک اشاره : به شیء تزئین کننده، مانند یک Wrappers فکر کنید. اجاز بدهید ببینم که چه طور این کار را انجام می‌دهیم.
-1 یک شی از DarkRoast ایجاد می‌کنیم.

1 We start with our DarkRoast object.

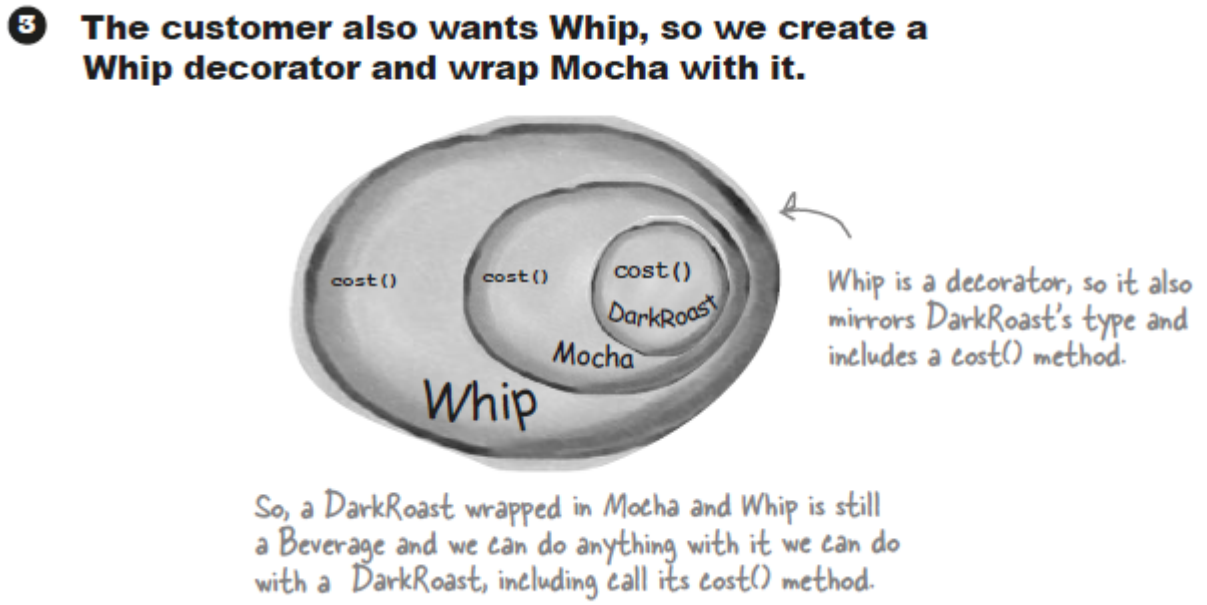


-2 آن را با یک شی از Mocha تزئین می‌کنیم.

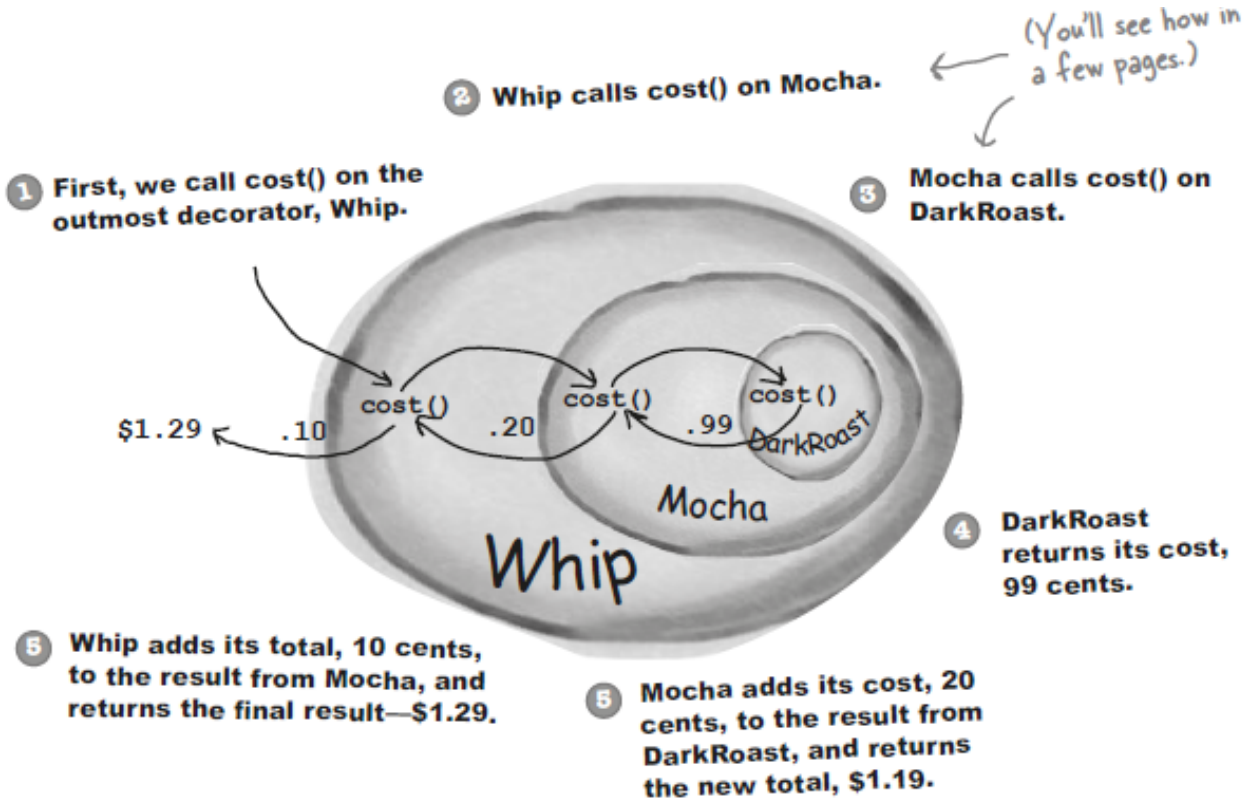
2 The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.

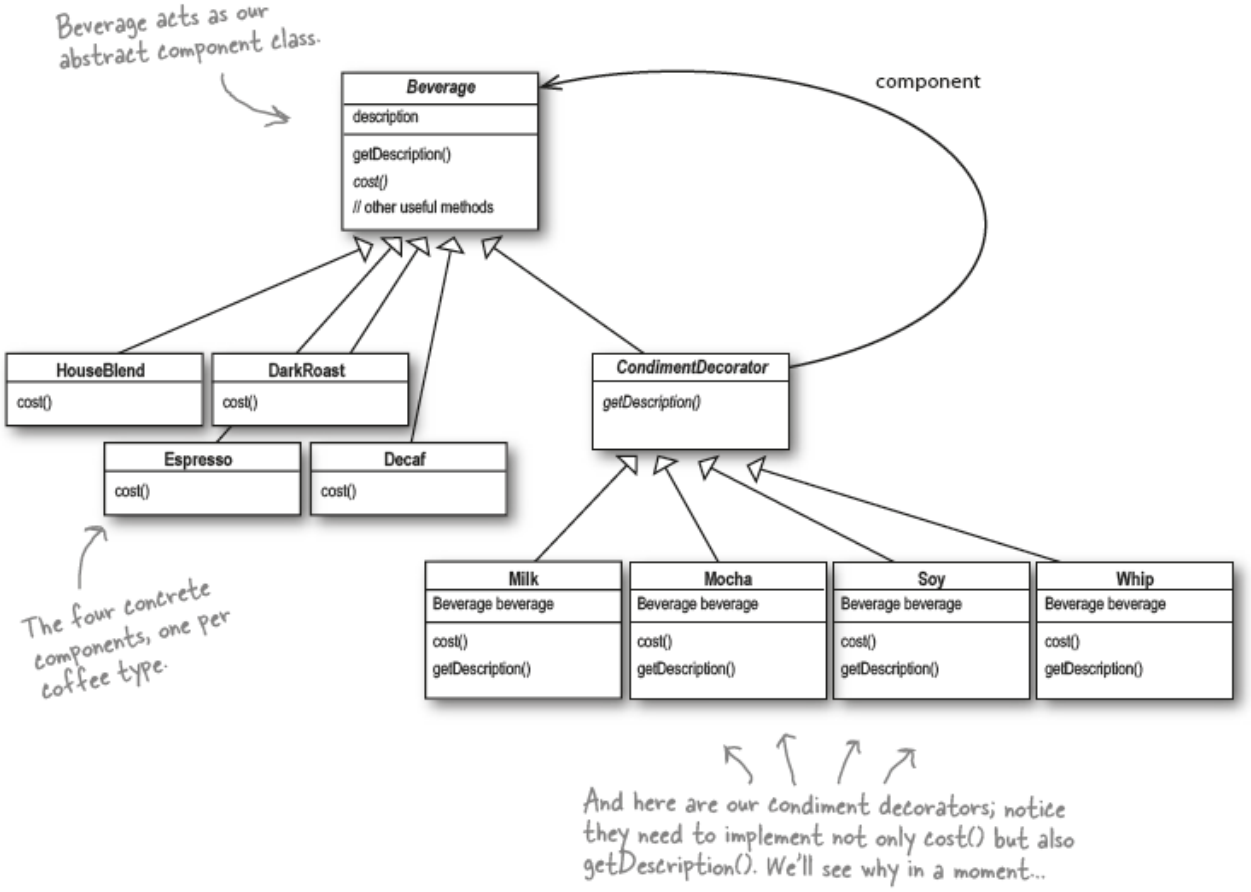


-3 آن را با یک شی از Whip تزیین می‌کنیم



4- حالا زمان محاسبه قیمت محصول برای مشتری فرا رسیده است. ما این کار را را با صدا زدن بیرونی‌ترین Decorator(Whip) انجام می‌دهیم و شی whip به کمک Delegate مابقی توابع cost را صدا می‌زند.





در آخر شما می‌توانید پیاده سازی این برنامه را به زبان جاوا در زیر مشاهده نمایید.

```
public abstract class Beverage
{
     string description ="unknow beverage";
     public String getDescription(){
         return description;
     }
     public abstract double cost();
}

public abstract class CondimentDecorator extends Beverage {
    public abstract string getDescription();
}

public class Espersso extends Beverage{
   public Espersso()
   {
      description="Espersso";
   }
   public double cost(){
     return 1.99;
   }
}

public class HouseBelend extends Beverage {
   public HouseBelend()
   {
     description="HouseBelend";
   }
   public double cost()
   {
     return .89;
   }
}
```

```
public class mocha extends condimateDecorator {
  Beverage beverage;
  public mocha(Beverage beverage)
  {
     this.beverage=beverage;
  }
   public string getDescription(){
      return beverage.getdescription() + "Mocha";
   }
   public double cost(){
     return .20 +beverage.cost
   }
}


// Now Use These classes in Final form
Beverage beverage=new Espersso();
//Customers want a coffe with double milk and whip
beverage=new mocha(beverage);
beverage=new mocha(meverage);
beverage=new whip(beverage);

system.out.println(beverage.getDescription() + "$" +beverage.cost());
```

نظرات خوانندگان

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳۹۲/۰۱/۲۳ ۹:۵۳

به نظرم ترجمه بخشی از کتاب Head First Design Pattern باشه. کتاب خوبیه.

نویسنده: حامد صمدی
تاریخ: ۱۳۹۲/۰۱/۲۳ ۱۲:۳۹

بله آقای کوکبی ترجمه ای از کتاب Head First Design Pattern است

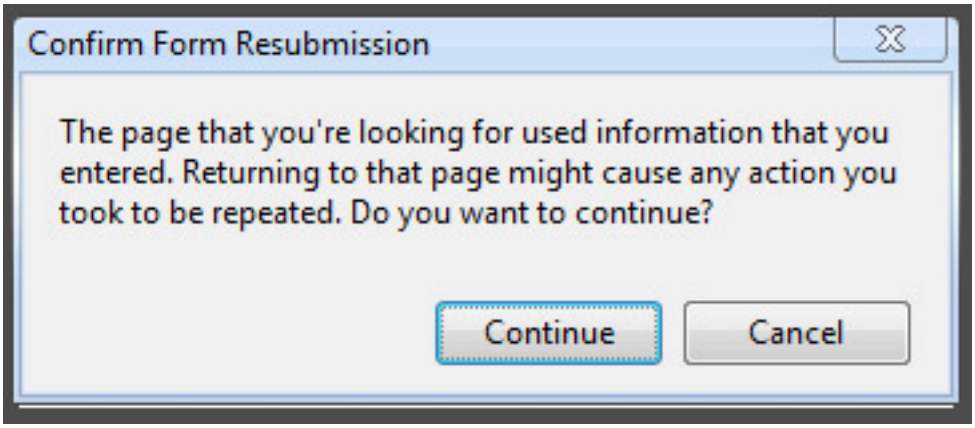
نویسنده: توحید عزیزی
تاریخ: ۱۳۹۲/۰۱/۲۴ ۱:۱۲

ضمن تشکر از مقاله بسیار مفید شما، در مثال آخر مقاله، نوشته شده که مشتری اسپرسو را با دو شات «شیر» و [یک] شات «ویپ» می‌خواهد، اما کد نوشته شده، ۲ شات «موکا» اضافه کرده است.
از دقت خودم در قضایای شکمی، شرمنده ام :)

```
//Customers want a coffe with double milk and whip
beverage=new mocha(beverage);
beverage=new mocha(meverage);
```

عنوان: **الگوی PRG در ASP.NET MVC**
نویسنده: محسن اسماعیل پور
تاریخ: ۱۳۹۲/۰۳/۰۶ ۱۱:۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, MVC, ASP.Net MVC

تا حالا با این پنجره حتما مواجه شدین:



دارید اطلاعات یک فرم داخل صفحه رو به سمت سرور میفرستید و پس از اتمام عملیات، صفحه دوباره نمایش داده میشه. در این حالت اگه دکمه F5 یا دکمه Refresh مرور گر رو بزنید، با این پنجره مواجه میشید که میگه دارید اطلاعات قبلی رو دوباره به سمت سرور میفرستید. بعضی وقت‌ها کاربران به هر دلیل دوباره صفحه رو Refresh میکنند و با این پنجره روبرو میشن بدون اینکه بدونن جریان از چه قراره، دوباره اطلاعات رو به سمت سرور میفرستن و این کار باعث ثبت اطلاعات تکراری میشه. برای جلوگیری از این کار الگویی به نام Post/Redirect/Get هست که راه حلی رو برای اینکار پیشنهاد میده.

راه حل به این صورت هست که پس از پست شدن فرم به سمت سرور و انجام عملیات، بجای اینکه صفحه، دوباره با استفاده از متد GET به کاربر نشون داده بشه، کاربر Redirect بشه به صفحه. برای توضیح این مسئله به سراغ AccountController که بصورت پش فرض وقتی یک پروژه ASP.NET MVC رو از نوع Internet ایجاد میکنید، وجود داره.

اکشن Register از نوع GET صفحه ثبت نام کاربر رو نمایش میده.

```
[HttpGet]
[AllowAnonymous]
public ActionResult Register()
{
      return View();
}
```

پس از اینکه کاربر اطلاعات داخل فرم رو پر کرد و به سمت سرور فرستاد و صحت اطلاعات فرستاده معتبر و عمل ثبت موفقیت آمیز بود برای ادامه کار به دو روش میتوان عمل کرد:
۱- کاربر به صفحه دیگری منتقل بشه و در اون صفحه پیام موفقیت آمیز بودن عملیات نشون داده بشه. مثلا معمولا پس از انجام عمل ثبت نام، کاربر به صفحه شخصی یا صفحه اصلی سایت منتقل میشه و یا در موقع ویرایش اطلاعات پش از انجام عمل ویرایش کاربر به صفحه دیگری که لیستی از آیتمها که کاربر یکی از آنها را ویرایش کرده باز گردانده میشه.

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public ActionResult Register(RegisterModel model)
{
    if (ModelState.IsValid)
```

```
    {
        // Attempt to register the user
        try
        {
            WebSecurity.CreateUserAndAccount(model.UserName, model.Password);
            WebSecurity.Login(model.UserName, model.Password);
            ViewBag.Message = "Successfully Registered!";

            // PRG has been maintained
            return RedirectToAction("Index", "Home");
        }
        catch (MembershipCreateUserException e)
        {
            ModelState.AddModelError("", ErrorCodeToString(e.StatusCode));
        }
    }
    // If we got this far, something failed, redisplay form
    return View(model);
}
```

۲- نمایش دوباره صفحه ولی با تغییر هدر صفحه به کد 303 . کد 303 به مروگر اعلام میکنه صفحه ریدایرکت شده است

```
[HttpPost]
[AllowAnonymous]
[ValidateAntiForgeryToken]
public ActionResult Register(RegisterModel model)
{
    if (ModelState.IsValid)
    {
        // Attempt to register the user
        try
        {
            WebSecurity.CreateUserAndAccount(model.UserName, model.Password);
            WebSecurity.Login(model.UserName, model.Password);
            ViewBag.Message = "Successfully Registered!";

            // PRG has been maintained
            return RedirectToAction("Register");
        }
        catch (MembershipCreateUserException e)
        {
            ModelState.AddModelError("", ErrorCodeToString(e.StatusCode));
        }
    }
    // If we got this far, something failed, redisplay form
    return View(model);
}
```

در این حالت دوباره صفحه ثبت نام نمایش داده میشه ولی با زدن دکمه رفرش، اطلاعات دوباره به سمت سرور فرستاده نمیشه

نظرات خوانندگان

نویسنده: محمد رضا ابراهیم راد
تاریخ: ۱۳۹۲/۰۳/۰۷ ۱۲:۲

با تشکر از مطلبی که نوشتید ولی من تو این کدها که 303 نمی‌بینم.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۲/۰۳/۰۷ ۱۳:۳۲

البته روش توضیح داده شده همان روش متداول PRG است. اگر حتما نیاز به 303 دارید به روش زیر باید عمل کرد:

```
using System.Web.Mvc;

namespace TestMvcPRG.Helper
{
    public class Redirect303 : ActionResult
    {
        private string _url;

        public Redirect303(string url)
        {
            _url = url;
        }

        public override void ExecuteResult(ControllerContext context)
        {
            context.HttpContext.Response.StatusCode = 303; // redirect using GET
            context.HttpContext.Response.RedirectLocation = _url;
        }
    }

    public abstract class BaseController : Controller
    {
        public Redirect303 Redirect303(string actionName)
        {
            return new Redirect303(Url.Action(actionName));
        }

        public Redirect303 Redirect303(string actionName, object routeValues)
        {
            return new Redirect303(Url.Action(actionName, routeValues));
        }

        public Redirect303 Redirect303(string actionName, string controllerName)
        {
            return new Redirect303(Url.Action(actionName, controllerName));
        }

        public Redirect303 Redirect303(string actionName, string controllerName, object routeValues)
        {
            return new Redirect303(Url.Action(actionName, controllerName, routeValues));
        }
    }
}
```

و بعد برای استفاده:

```
using System.Web.Mvc;
using TestMvcRPG.Helper;

namespace TestMvcPRG.Controllers
{
    public class HomeController : BaseController
    {
        [HttpGet]
        public ActionResult Index()
        {
            return View();
        }
```

```
        [HttpPost]
        public ActionResult Index(string data)
        {
            if (ModelState.IsValid)
            {
                return Redirect303("Index"); // post-redirect-get
            }
            return View();
        }
    }
}
```

عنوان: **استفاده از SimpleIoc به عنوان یک IoC Container**
نویسنده: مهدی ع
تاریخ: ۱۳۹۲/۰۳/۱۱ ۱۶:۲۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, MVVM, Dependency Injection, SimpleIoc, Ioc, mvvm light

SimpleIoc به صورت پیش فرض در پروژه های MVVM Light موجود می‌باشد. قطعه کد پایین به صورت پیش فرض در پروژه‌های MVVM Lightایجاد می‌شود.
در کلاس ViewModelLocator ما تمام میانجی(Interface)ها و اشیا(Objects)ی مورد نیازمان را ثبت(register) می‌کنیم.
در ادامه اجزای مختلف آن را شرح می‌دهیم.

```
class ViewModelLocator
{
    static ViewModelLocator()
    {
        ServiceLocator.SetLocatorProvider(() => SimpleIoc.Default);
        if (ViewModelBase.IsInDesignModeStatic)
        {
            SimpleIoc.Default.Register<IDataService, Design.DesignDataService>();
        }
        else
        {
            SimpleIoc.Default.Register<IDataService, DataService>();
        }
        SimpleIoc.Default.Register<MainViewModel>();
        SimpleIoc.Default.Register<SecondViewModel>();
    }

    public MainViewModel Main
    {
        get
        {
            return ServiceLocator.Current.GetInstance<MainViewModel>();
        }
    }
}
```

1) هر شیء که به صورت پیش فرض ایجاد می‌شود با الگوی Singlton ایجاد می‌شود.

```
SimpleIoc.Default.GetInstance<MainViewModel>(Guid.NewGuid().ToString());
```

2) جهت ثبت یک کلاس مرتبط با میانجی آن از روش زیر استفاده می‌شود.

```
SimpleIoc.Default.Register<IDataService, Design.DesignDataService>();
```

3) جهت ثبت یک شی مرتبط با میانجی از روش زیر استفاده می‌شود.

```
SimpleIoc.Default.Register<IDataService>(myObject);
```

4) جهت ثبت یک نوع (Type) به طریق زیر عمل می‌کنیم.

```
SimpleIoc.Default.Register<MainViewModel>();
```

5) جهت گرفتن وهله (Instance) از یک میانجی خاص، از روش زیر استفاده می‌کنیم.

```
SimpleIoc.Default.GetInstance<IDataService>();
```

6) جهت گرفتن وهله ای به صورت مستقیم، 'ایجاد و وضوح وابستگی(dependency resolution)' از روش زیر استفاد می‌کنیم.

```
SimpleIoc.Default.GetInstance();
```

7) برای ایجاد داده‌های زمان طراحی از روش زیر استفاده می‌کنیم.

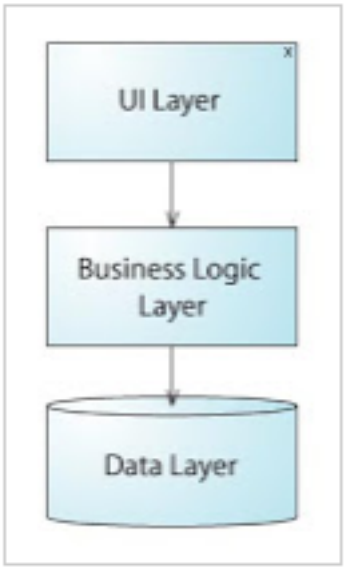
```
if (ViewModelBase.IsInDesignModeStatic)
        {
            SimpleIoc.Default.Register<IDataService, Design.DesignDataService>();
        }
        else
        {
            SimpleIoc.Default.Register<IDataService, DataService>();
        }
```

در حالت زمان طراحی، سرویس‌های زمان طراحی به صورت خودکار ثبت می‌شوند. و می‌توان این داده‌ها را در ViewModelها و Viewها حین طراحی مشاهده نمود.

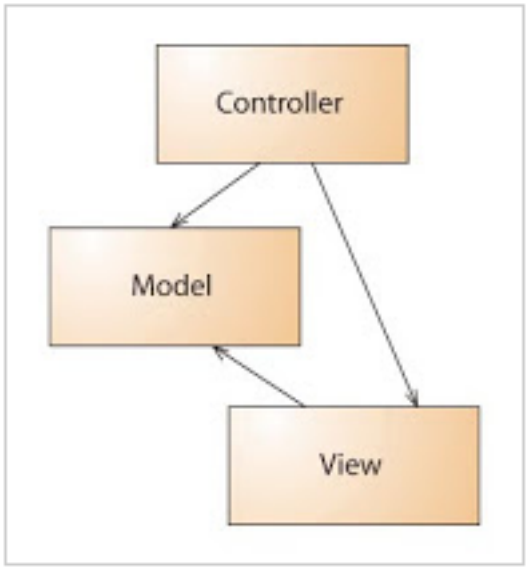
منبع

عنوان: MVC vs 3-Tier Pattern
نویسنده: مهرداد اشکانی
تاریخ: ۱۳۹۲/۰۳/۲۶ ۱۴:۰
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, MVC, Architecture, معماری سه لایه, MVC معماری

من تا به حال برنامه نویس‌های زیادی را دیده‌ام که می‌پرسند «چه تفاوتی بین الگوهای معماری MVC و Three-Tier وجود دارد؟» قصد من روشن کردن این سردرگمی، بوسیله مقایسه هردو، با کنار هم قرار دادن آنها می‌باشد. حداقل در این بخش ، من اعتقاد دارم، منبع بیشتر این سردرگمی‌ها در این است که هر دوی آنها، دارای سه لایه متمایز و گره، در دیاگرام مربوطه‌اشان هستند.



Three-Tier



MVC

اگر شما به دقت به دیاگرام آنها نگاه کنید، پیوستگی را خواهید دید. بین گره‌ها و راه اندازی آنها، کمی تفاوت است.

معماری سه لایه

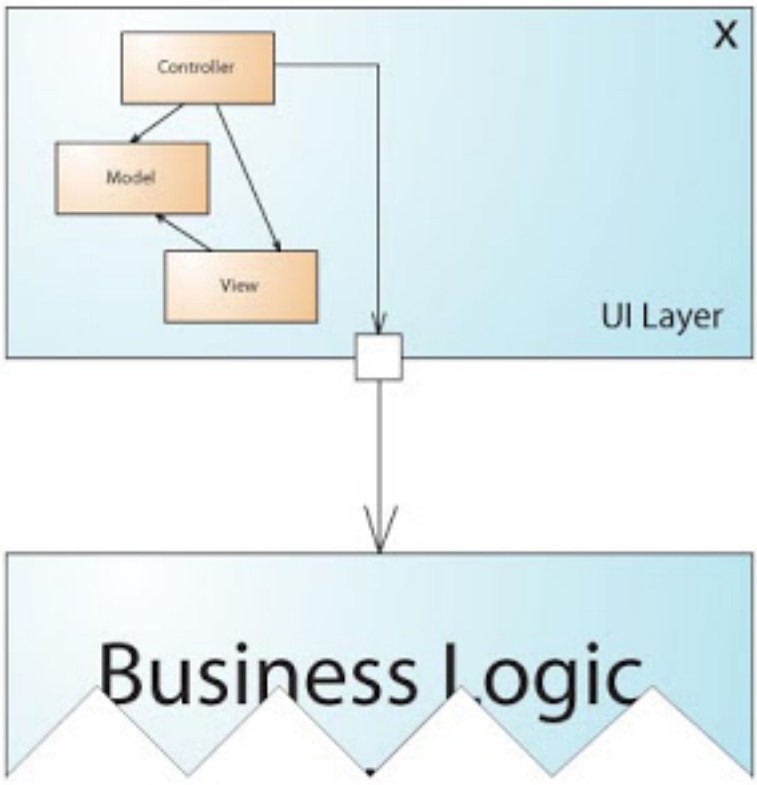
سیستم‌های سه لایه، واقعاً لایه‌ها را می‌سازند: لایه UI به لایه Business logic دسترسی دارد و لایه Business logic به لایه Data دسترسی دارد. اما لایه UI دسترسی مستقیمی به لایه Data ندارد و باید از طریق لایه Business logic و روابط آنها عمل کند. بنابراین می‌توانید فکر کنید که هر لایه، بعنوان یک جزء، آزاد است؛ همراه با قوانین محکم طراحی دسترسی بین لایه ها.

MVC

در مقابل، این Pattern ، لایه‌های سیستم را نگهداری نمی‌کند. کنترلر به مدل و View (برای انتخاب یا ارسال مقادیر) دسترسی دارد. View نیز دسترسی دارد به مدل . دقیقاً چطور کار می‌کند؟ کنترلر در نهایت نقطه تصمیم گیری منطقی است. چه نوع منطقی؟ نوعاً، کنترلر، ساخت و تغییر مدل را در اکشن‌های مربوطه، کنترل خواهد کرد . کنترلر سپس تصمیم گیری می‌کند که برای منطق داخلیش، کدام View مناسب است. در آن نقطه، کنترلر مدل را به View ارسال می‌کند. من در اینجا چون هدف بحث مورد دیگه‌ای می‌باشد، مختصر توضیح دادم.

چه موقع و چه طراحی را انتخاب کنم؟

اول از همه، هر دو طراحی قطعاً و متقابلاً منحصر بفرد نیستند. در واقع طبق تجربه‌ی من، هر دو آنها کاملاً هماهنگ هستند. اغلب ما از معماری چند لایه استفاده می‌کنیم مانند معماری سه لایه، برای یک ساختار معماری کلی. سپس من در داخل لایه UI ، از MVC استفاده می‌کنم، که در زیر دیاگرام آن را آورده ام.



نظرات خوانندگان

نویسنده: محسن اسماعیل پور
تاریخ: ۱۳۹۲/۰۳/۲۶ ۲۲:۳۳

3-Layer در واقع Architecture Style هست اما MVC یک Design Pattern هست پس مقایسه مستقیم نمیدونم کاری دست باشد یا نه اما میتونیم به این شکل نتیجه گیری کنیم:
Data Access: شامل کلاسهای ADO.NET یا EF برای کار با دیتابیس.
Business Logic: یا همان Domain logic که میتوان Model رو به عنوان Business entity در این لایه بکار برد.
UI Layer: بکارگیری Controller و View در این لایه

نویسنده: یزدان
تاریخ: ۱۳۹۲/۰۳/۲۷ ۱:۳۳

در برنامه نویسی 3لایه کار Business Logic به طور واضح و شفاف چی هست و چه کارهایی در این لایه لحاظ میشه ؟

نویسنده: fss
تاریخ: ۱۳۹۲/۰۳/۲۷ ۸:۴۱

منم یه مدت دچار این ابهام بودم. ولی الان اینطور نتیجه می‌گیرم:

mvc کلا در لایه UI قرار داره. یعنی اگر شما لایه BL و DAL رو داشته باشید، حالا میتونید لایه UI رو با یکی از روش ها، مثلا سیلورلایت، asp.net mvc یا asp.net web form پیاده کنید.

نویسنده: محسن خان
تاریخ: ۱۳۹۲/۰۳/۲۷ ۸:۴۹

همون لایه UI هم نیاز به جداسازی کدهای نمایشی از کدهای مدیریت کننده آن برای بالابردن امکان آزمایش کردن و یا حتی استفاده مجدد قسمت‌های مختلف اون داره. در این حالت شما راحت نمی‌تونید MVC و Web forms رو در یک سطح قرار بدی (که اگر اینطور بود اصلا نیازی به MVC نبود؛ نیازی به MVVM برای سیلورلایت یا WPF نبود و یا نیازی به MVP برای WinForms یا Web forms نبود).

نویسنده: fss
تاریخ: ۱۳۹۲/۰۳/۲۷ ۹:۱۳

دوست عزیز من متوجه منظور شما نشدم. حرف من اینه که MVC، MVVM، MVP و .. در سطح UI پیاده میشن.

نویسنده: مهرداد اشکانی
تاریخ: ۱۳۹۲/۰۳/۲۷ ۹:۱۸

لایه business Logic در واقع لایه پیاده سازی Business پروژه شما می‌باشد با یک مثال عرض می‌کنم فرض کنید در لایه UI شما لازم دارید یک گزارش از لیست مشتریانی که بالاترین خرید را در 6 ماه گذشته داشته اند و لیست تراکنش مالی آنها را بدست آورید.برای این مورد شما توسط کلاسهای و متدهای لازم ، در لایه Business Logic این عملیات را پیاده سازی می‌کنید.

نویسنده: مهرداد اشکانی
تاریخ: ۱۳۹۲/۰۳/۲۷ ۹:۳۸

این طور نیست دوست عزیز شما می‌تونید حتی برای Model هم لایه در نظر بگیرید که براحتی توسط لایه Business و کلا لایه‌های دیگر در دسترس باشد.که این مورد الان در MVC خیلی کاربرد دارد.مواردی که من عرض کردم برای رفع ابهام بین معماری چند لایه و Pattern MVC بود.

نویسنده: داود
تاریخ: ۱۳۹۲/۰۳/۲۷ ۱۷:۱۴

به بنظر بنده هم معماری رو نمی‌شود با الگو مقایسه کرد به هر حال خود الگوی mvcها یک سری لایه داره و تا اونجایی هم که می‌دونم فرق tier با layer اینه که tierها رو از لحاظ فیزیکی هم جدا می‌کنند

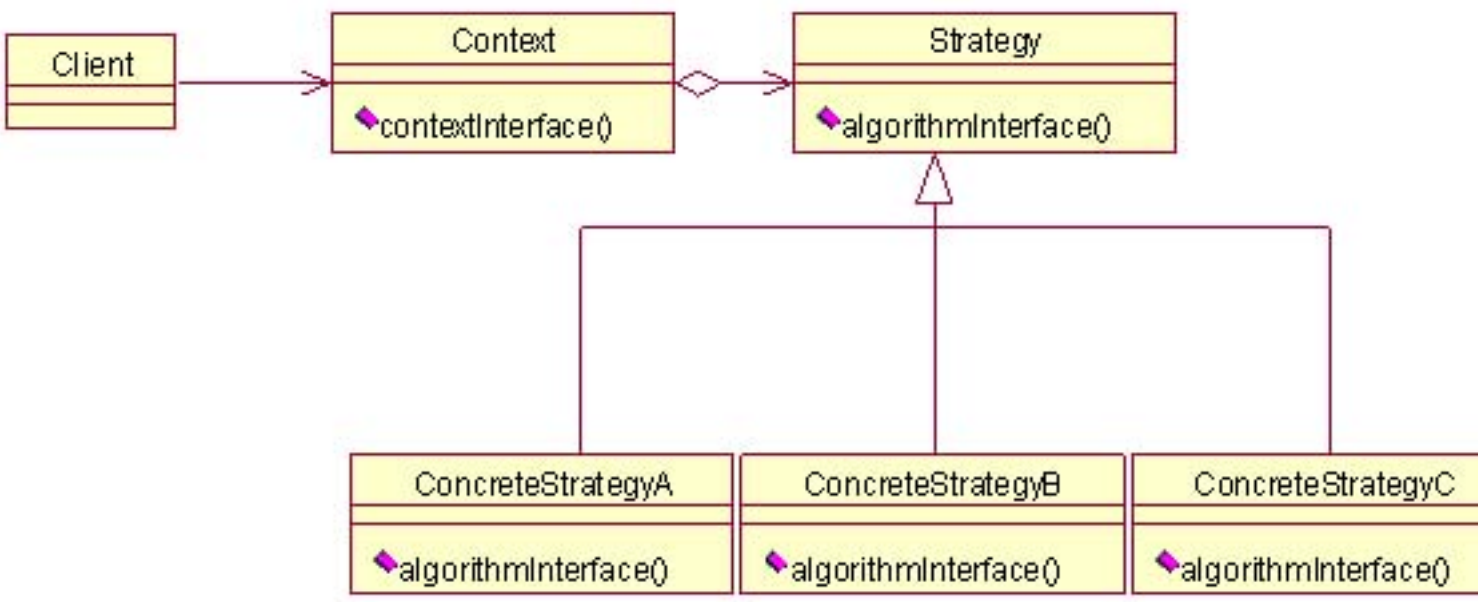
عنوان: **الگوی استراتژی - Strategy Pattern**
نویسنده: مجتبی شاطری
تاریخ: ۱۳۹۲/۰۶/۲۰ ۰:۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, Dependency Injection, SOLID Principals, Inheritance Strategies, SOLID, Dependency Inversion

الگوی استراتژی (Strategy) اجازه می-دهد که یک الگوریتم در یک کلاس بسته بندی شود و در زمان اجرا برای تغییر رفتار یک شیئ 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 کارائی 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 مبداء مرتب کنیم. دوباره برای مرتب سازی الگوریتم‌های مختلف وجود دارد و هر کدام در شرایط خاص، کارائی بهتری نسبت به الگوریتم‌های دیگر دارد. مسئله دقیقا شبیه مسئله بالا است و این مسله می‌توانند دارای طراحی شبیه مسله 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، طراح‌های 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 الگویتم زیر برای مرتب سازی استفاده کنیم.

1 - الگوریتم مرتب سازی Shell Sort 2 - 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 می‌گیریم (همان کلاس‌های ConcreteStrategy). برای اینکه کلاس Client بتواند به سادگی یک از استراتژی‌ها را انتخاب کنید بهتر است که تمام کلاس‌های استراتژی دارای اینترفیس مشترک باشند. برای این کار می‌توانیم یک کلاس abstract تعریف کنیم و ویژگیهای مشترک کلاس‌های استراتژی را در آن قرار دهیم و کلاس‌های استراتژی آنها را به ارث ببرند(همان کلاس Strategy) و پیاده سازی کنند.

در زیل کلاس Abstract که کل کلاس‌های استراتژی از آن ارث می‌برند را مشاهده می‌کنید :

```
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 استراتژی‌ها را برای مرتب کردن به کار می‌برد :

```
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: علي
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 می‌تونی 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 کوکبي
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 Abstact یک اینترفیس است که می‌تواند پیاده سازی هم داشته باشد.
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 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: ۱۳۹۲/۰۶/۳۱ ۱۳:۲۷

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: ۱۳۹۲/۰۶/۳۱ ۱۳:۴۹

- 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، عموما از ترکیب این دو استفاده می‌شود. مواردی که قرار است در اختیار عموم باشند حتی دو لایه هم می‌شوند. مثلا 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: ۱۳۹۲/۰۷/۰۱ ۱۶:۱۱

توابع abstract بطور ضمنی virtual هستند.

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

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: ۱۳۹۲/۰۸/۲۱ ۹:۱۵

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

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

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: ۱۳۹۲/۰۸/۲۱ ۱۱:۴۳

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

عنوان: **الگوی طراحی Factory Method به همراه مثال**
نویسنده: مجتبی شاطری
تاریخ: ۱۳۹۲/۰۶/۲۸ ۰:۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, Domain Service, Domain Driven Design, SOLID Principals

الگوی طراحی Factory Method به همراه مثال

عناوین :
· تعریف Factory Method
· دیاگرام UML
· شرکت کنندگان در UML
· مثالی از Factory Pattern در C#

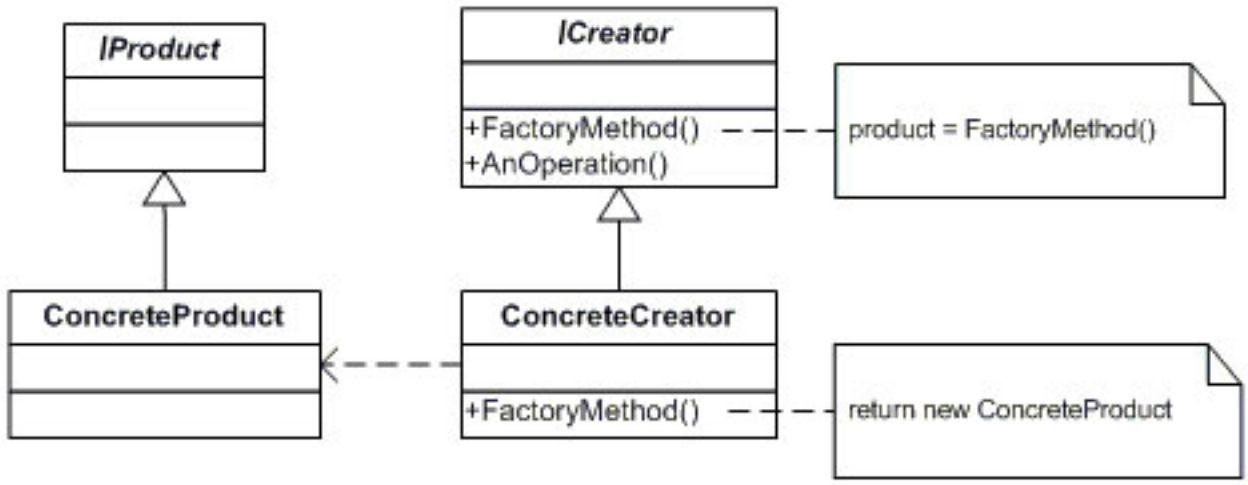
تعریف الگوی Factory Method :

این الگو پیچیدگی ایجاد اشیاء برای استفاده کننده را پنهان می‌کند. ما با این الگو میتوانیم بدون اینکه کلاس دقیق یک شیئ را مشخص کنیم آن را ایجاد و از آن استفاده کنیم. کلاینت (استفاده کننده) معمولا شیئ واقعی را ایجاد نمی‌کند بلکه با یک واسط 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 معمولا اطلاعاتی را به متدی استاتیک از این کلاس می‌فرستد و این متد بر اساس آن اطلاعات تصمیم می‌گیرید 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 زیل هستند :

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

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

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

- Icreator واسطیست که Factory Method را تعریف می‌کند. پیاده ساز 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 پیاده ساز این واسط است و ما در این نمودار این وظیفه را فقط بر عهده ConcreteCreator گذاشته ایم که از واسط Icreator مشتق شده است.

پیاده سازی UMLفوق به صورت زیر است:

در ابتدا کلاس واسط IProduct تعریف شده است.

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

در این مرحله ما پند پیاده سازی از IProduct انجام می‌دهیم.

```
class ConcreteProductA : IProduct
{
     // پیاده سازی A
}

class ConcreteProductB : IProduct
{
     // پیاده سازی B
}
```

در این مرحله کلاس انتزاعی Creator تعریف می‌شود.

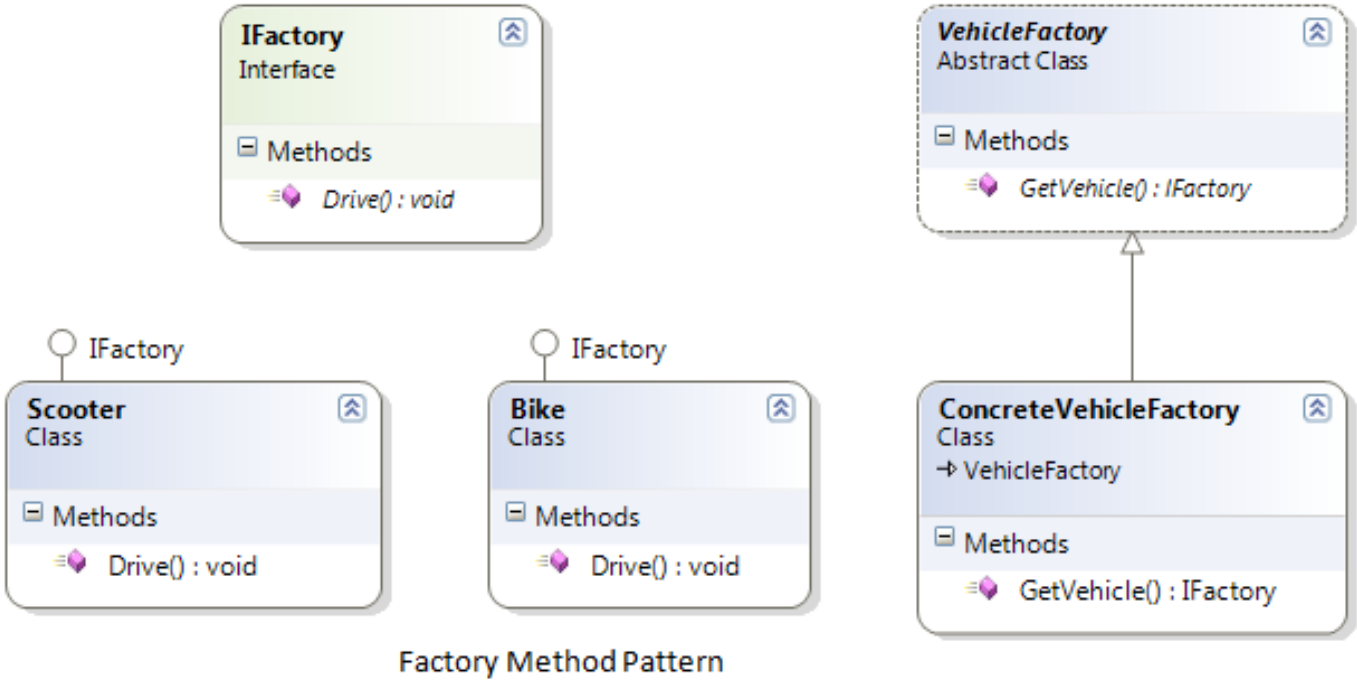
```
abstract class Creator
{
         // این متد بر اساس نوع ورودی انتخاب مناسب را انجام و باز می‌گرداند
          public abstract IProduct FactoryMethod(string type);
}
```

در این مرحله ما با ارث بری از Creator متد Abstract آن را به شیوه خودمان پیاده سازی 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 زیل است :

```
using System;

namespace FactoryMethodPatternRealWordConsolApp
{
    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 :

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 خواهم کرد...

نظرات خوانندگان

نویسنده: سید ایوب کوکبی
تاریخ: ۱۳۹۲/۰۷/۰۲ ۱۹:۲۲

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؟ یعنی در مثال رابطه بین ConcreteVehicleFactory و یکی از کلاس‌های Bike و یا Scooter
-4چرا در ویژوال استودیو تولید خودکار uml از کد موجود متفاوت با دیاگرام فعلی است، مثلا نوع روابط درست نمایش داده نمیشه و همچنین رابطه ای که در مورد 3 اشاره شد در اینجا وجود نداره؟ آیا علتش نقص در این ابزار است؟ اگر بله، آیا ابزاری وجود داره که دیاگرام رو دقیقتر جنریت کنه؟
و یک نکته:
در کلاس‌های Scooter و Bike نیازی به استفاده از متد ToStringبرای 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: ۱۳۹۲/۰۷/۰۳ ۰:۳

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 اگرشما 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 میشه 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. درمثال‌ها 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.

عنوان: **استفاده از الگوی Adapter در تزریق وابستگی‌ها**
نویسنده: محمد رضا منشادی
تاریخ: ۱۳۹۲/۰۸/۱۹ ۱۸:۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, Dependency Injection, Ioc, Dependency Inversion

در بعضی از مواقع ممکن است که در هنگام استفاده از اصل تزریق وابستگی‌ها، با یک مشکل روبرو شویم و آن این است که اگر از کلاسی استفاده می‌کنیم که به سورس آن دسترسی نداریم، نمی‌توانیم برای آن یک Interface تهیه کنیم و اصل (Depend on abstractions, not on concretions) از بین می‌رود، حال چه باید کرد.
برای اینکه موضوع تزریق وابستگی‌ها (DI) به صورت کامل در قسمتهای دیگر سایت توضیح داده شده است، دوباره آن را برای شما بازگو نمی‌کنیم .
لطفا به کدهای ذیل توجه کنید:

کد بدون تزریق وابستگی ها

به سازنده کلاس ProductService و تهیه یک نمونه جدید از وابستگی مورد نیاز آن دقت نمائید:

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

namespace ASPPatterns.Chap2.Service
{
    public class Product
    {
    }

    public class ProductRepository
    {
        public IList<Product> GetAllProductsIn(int categoryId)
        {
            IList<Product> products = new List<Product>();
            // Database operation to populate products …
            return products;
        }
    }

    public class ProductService
    {
        private ProductRepository _productRepository;
        public ProductService()
        {
            _productRepository = new ProductRepository();
        }

        public IList<Product> GetAllProductsIn(int categoryId)
        {
            IList<Product> products;
            string storageKey = string.Format("products_in_category_id_{0}", categoryId);
            products = (List<Product>)HttpContext.Current.Cache.Get(storageKey);
            if (products == null)
            {
                products = _productRepository.GetAllProductsIn(categoryId);
                HttpContext.Current.Cache.Insert(storageKey, products);
            }
            return products;
        }
    }
}
```

همان کد با تزریق وابستگی

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

```
namespace ASPPatterns.Chap2.Service
{
    public interface IProductRepository
    {
        IList<Product> GetAllProductsIn(int categoryId);
    }

    public class ProductRepository : IProductRepository
    {
        public IList<Product> GetAllProductsIn(int categoryId)
        {
            IList<Product> products = new List<Product>();
            // Database operation to populate products …
            return products;
        }
    }

    public class ProductService
    {
        private IProductRepository _productRepository;
        public ProductService(IProductRepository  productRepository)
        {
            _productRepository = productRepository;
        }

        public IList<Product> GetAllProductsIn(int categoryId)
        {
            //…
        }
    }
}
```

همانطور که ملاحظه می‌کنید به علت دسترسی به سورس، به راحتی برای استفاده از کلاس ProductRepository در کلاس ProductService، از تزریق وابستگی‌ها استفاده کرده‌ایم.

اما از این جهت که شما دسترسی به سورس Http context class را ندارید، نمی‌توانید به سادگی یک Interface را برای آن ایجاد کنید و سپس یک تزریق وابستگی را مانند کلاس ProductRepository برای آن تهیه نمائید.

خوشبختانه این مشکل قبلا حل شده است و الگویی که به ما جهت پیاده سازی آن کمک کند، وجود دارد و آن الگوی آداپتر (Adapter Pattern) می‌باشد.

این الگو عمدتا برای ایجاد یک Interface از یک کلاس به صورت یک Interface سازگار و قابل استفاده می‌باشد. بنابراین می‌توانیم این الگو را برای تبدیل HTTP Context caching API به یک API سازگار و قابل استفاده به کار ببریم.

در ادامه می‌توان Interface سازگار جدید را در داخل productservice که از اصل تزریق وابستگی‌ها (DI) استفاده می‌کند تزریق کنیم.

یک اینترفیس جدید را با نام ICacheStorage به صورت ذیل ایجاد می‌کنیم:

```
public interface ICacheStorage
{
    void Remove(string key);
    void Store(string key, object data);
    T Retrieve<T>(string key);
}
```

حالا که شما یک اینترفیس جدید دارید، می‌توانید کلاس produceservic را به شکل ذیل به روز رسانی کنید تا از این اینترفیس، به جای HTTP Context استفاده کند.

```
public class ProductService
{
    private IProductRepository _productRepository;
    private ICacheStorage _cacheStorage;
    public ProductService(IProductRepository  productRepository,
    ICacheStorage cacheStorage)
    {
        _productRepository = productRepository;
        _cacheStorage = cacheStorage;
```

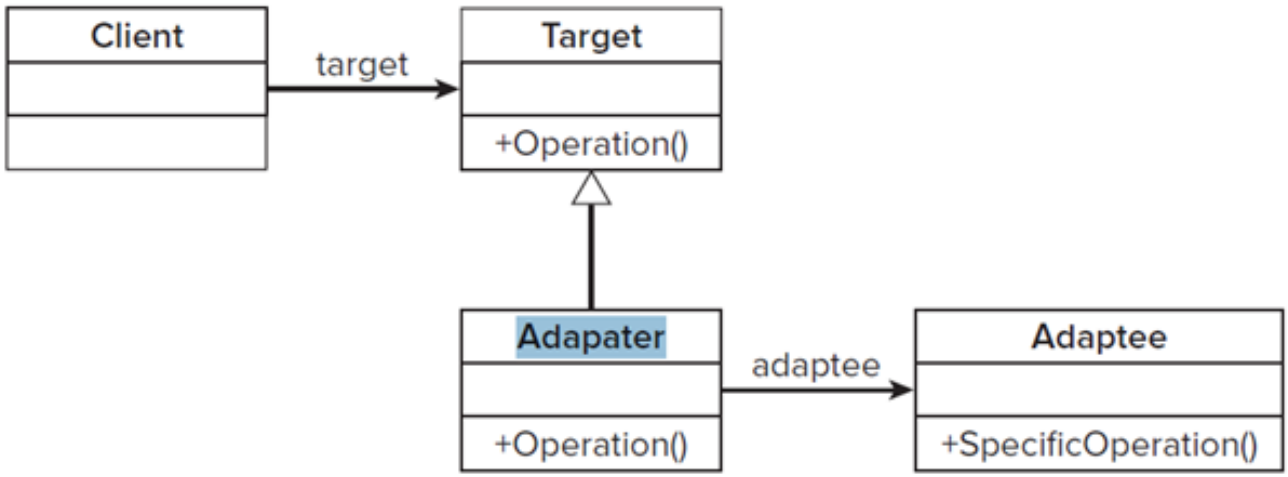
```
    }

    public IList<Product> GetAllProductsIn(int categoryId)
    {
        IList<Product> products;
        string storageKey = string.Format("products_in_category_id_{0}", categoryId);
        products = _cacheStorage.Retrieve<List<Product>>(storageKey);
        if (products == null)
        {
            products = _productRepository.GetAllProductsIn(categoryId);
            _cacheStorage.Store(storageKey, products);
        }
        return products;
    }
}
```

مسئله ای که در اینجا وجود دارد این است که HTTP Context Cache API صریحا نمی‌تواند Interface ایی که ما ایجاد کرده‌ایم را اجرا کند.

پس چگونه الگوی Adapter می‌تواند به ما کمک کند تا از این مشکل خارج شویم؟

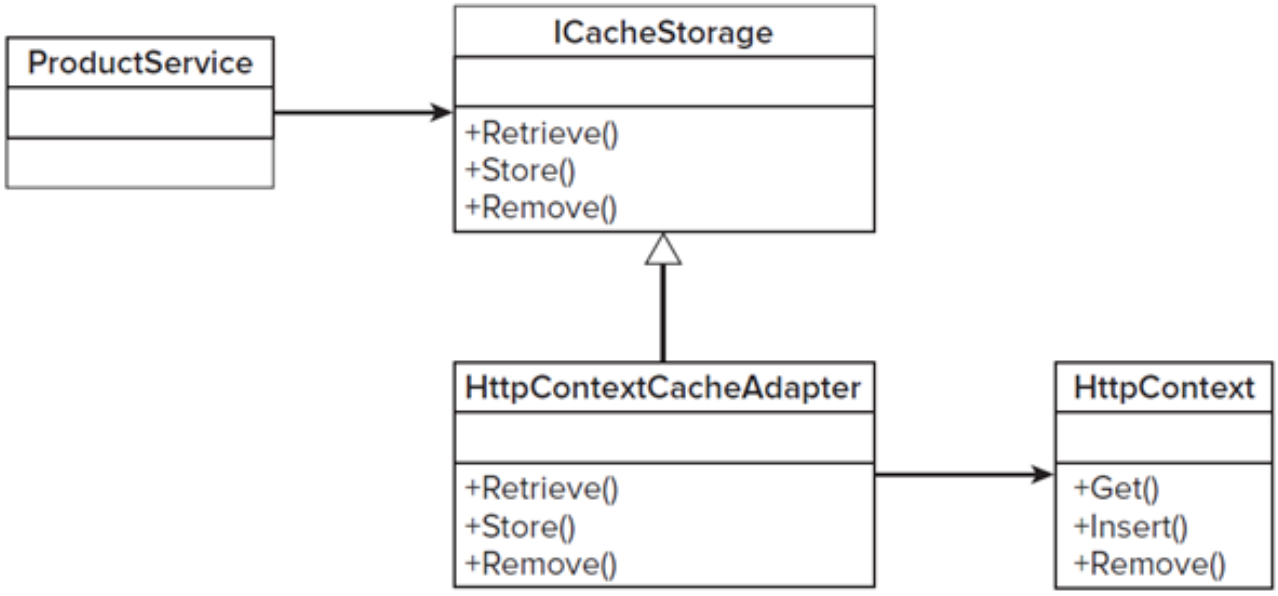
هدف این الگو به صورت ذیل در GOF مشخص شده است .«تبدیل Interface از یک کلاس به یک Interface مورد انتظار Client»

تصویر ذیل، مدل این الگو را به کمک UML نشان می‌دهد:



همانطور که در این تصویر ملاحظه می‌کنید، یک Client ارجاعی به یک Abstraction در تصویر (Target) دارد (ICacheStorage در کد نوشته شده). کلاس Adapter اجرای Target را بر عهده دارد و به سادگی متدهای Interface را نمایندگی می‌کند. در اینجا کلاس Adapter، یک نمونه از کلاس Adaptee را استفاده می‌کند و در هنگام اجرای قراردادهای Target، از این نمونه استفاده خواهد کرد.

اکنون کلاس‌های خود را در نمودار UML قرار می‌دهیم که به شکل ذیل آنها را ملاحظه می‌کنید.



در شکل ملاحظه می‌نمایید که یک کلاس جدید با نام HttpContextCacheAdapter مورد نیاز است. این کلاس یک کلاس روکش (محصور کننده یا Wrapper) برای متدهای HTTP Context cache است. برای اجرای الگوی Adapter کلاس HttpContextCacheAdapter را به شکل ذیل ایجاد می‌کنیم:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Web;
namespace ASPPatterns.Chap2.Service
{
    public class HttpContextCacheAdapter : ICacheStorage
    {
        public void Remove(string key)
        {
            HttpContext.Current.Cache.Remove(key);
        }

        public void Store(string key, object data)
        {
            HttpContext.Current.Cache.Insert(key, data);
        }

        public T Retrieve<T>(string key)
        {
            T itemStored = (T)HttpContext.Current.Cache.Get(key);
            if (itemStored == null)
                itemStored = default(T);
            return itemStored;
        }
    }
}
```

حال به سادگی می‌توان یک caching solution دیگر را پیاده سازی کرد بدون اینکه در کلاس ProductService اثر یا تغییری ایجاد کند .

منبع : Professional Asp.Net Design Pattern

عنوان: **آشنایی با الگوی Adapter**
نویسنده: فرهاد فرهمندخواه
تاریخ: ۱۳۹۲/۰۹/۰۴ ۲۱:۲۰
آدرس: www.dotnettips.info
گروه‌ها: Design patterns

قبل از آشنایی با الگوی Adapter،ابتدا با تعریف الگوهای ساختاری آشنا می‌شویم که به شرح ذیل می‌باشد:

الگوهای ساختاری (Structural Patterns):

از الگوهای ساختاری برای ترکیب کلاسها و اشیاء (Objects)،در جهت ایجاد ساختارهای بزرگتر استفاده می‌شود.به بیان ساده‌تر الگوهای ساختاری با ترکیب کلاسها و آبجکتها،قابلیت‌های کلاسهای غیر مرتبط را در قالب یک Interface(منظور ظاهر) در اختیار Client (منظور کلاس یا متد استفاده کننده می‌باشد) قرار می‌دهند.الگوهای ساختاری با استفاده از ارث بری به ترکیب Interfaceها پرداخته و آنها را پیاده سازی می‌نمایند.
استفاده از الگوهای ساختاری برای توسعه کتابخانه هایی (Library) که مستقل از یکدیگر می‌باشند،اما در کنار هم مورد استفاده قرار می‌گیرند،بسیار مفید است.

در ادامه به الگوی Adapter که یکی از الگوهای ساختاری است،می پردازیم. الگوی Adapter انواع مختلفی دارد که فهرست آنها به شرح ذیل می‌باشد:
1- Class Adapter 2- Object Adapter 3- Two way Adapter 4- Pluggable Adapter

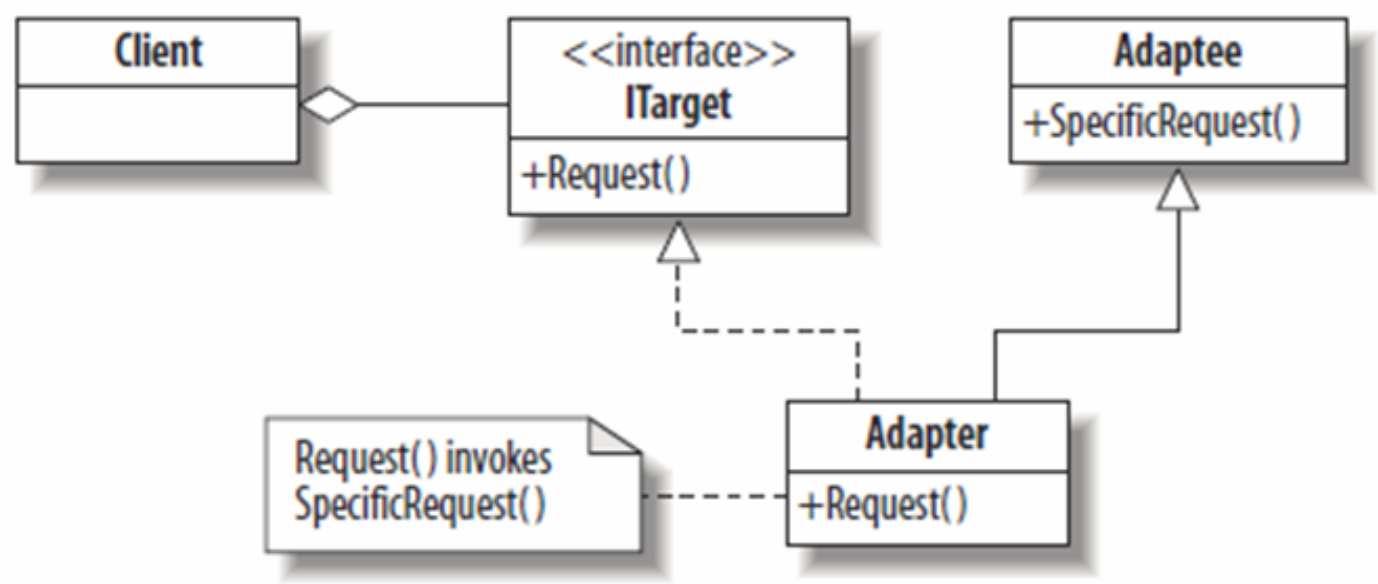
در این مقاله Class Adapter و Object Adapter را مورد بررسی قرار می‌دهیم و اگر عمری باقی باشد در مقاله بعدی Two-way Adapter و Pluggable Adapter را بررسی می‌کنیم.
قبل از پرداختن به هر یک از Adapterها با یکسری واژه آشنا می‌شویم،که در سرتاسر مقاله ممکن است از آنها استفاده شود.
Interface: منظور از Interface در اینجا، ظاهر یا امکاناتی است که یک کلاس می‌تواند ارائه دهد.
Client: منظور متد یا کلاسی است که از Interface مورد انتظار،استفاده می‌نمایید.

Intent (هدف)
هدف از ارائه الگوی Adapter ،تبدیل Interface یک Class به Interface ی که مورد انتظار Client است، می‌باشد.در واقع الگوی Adapter روشی است که بوسیله آن می‌توان کلاسهای با Interface متفاوت را در یک سیستم کنار یکدیگر مورد استفاده قرار داد. به بیان ساده‌تر هرگاه بخواهیم از کلاسهای ناهمگون یا نامنطبق (کلاسهای غیر مرتبط) در یک سیستم استفاده کنیم،راه حل مناسب استفاده از الگوی Adapter می‌باشد.

Adapter را به عنوان Wrapper می‌شناسند.الگوی Adapter از سه Component مهم تشکیل شده است،که عبارتند از:
Target،Adapter و Adaptee.
Target:کلاس یا Interface ی است که توسط Client مورد استفاده قرار می‌گیرد، و Client از طریق آن درخواستهای خود را بیان می‌کند. در واقع Functionality موجود در کلاس Target به جهت پاسخگویی به درخواست‌های Client فراهم گردیده است.
Adaptee: کلاسی است، دارای قابلیتهای مورد نیاز Client بطوریکه Interface اش با Interface مورد انتظار Client (یعنی Target)سازگار نیست. و Client برای استفاده از امکانات کلاس Adaptee و سازگاری با Interface مورد انتظارش نیاز به یک Wrapper همانند کلاسAdapter دارد.
Adapter: کلاسی است که قابلیتها و امکانات کلاس Adaptee را با Interface مورد انتظار Client یعنی Target سازگار می‌کند، تا Client بتواند از امکانات کلاس Adaptee جهت رفع نیازهای خود استفاده نماید. به بیان ساده‌تر Adapter کلاسی هست که برای اتصال دو کلاس نامتجانس (منظور دو کلاسی که هم جنس نمی‌باشند یا از نظر Interface بطور کامل با یکدیگر غیر مرتبط هستند) مورد استفاده قرار می‌گیرد.

در ادامه به بررسی اولین الگوی Adapter یعنی Class Adapter می‌پردازیم:
Class Adapter:

در این روش کلاس Adapter از ارث بری چند گانه استفاده می‌کند و Interface مرتبط به Adaptee را به Interface مرتبط به Target سازگار می‌نماید.
برای درک تعریف بالا مثالی را بررسی می‌کنیم، در ابتدا شکل زیر را مشاهده نمایید:



در شکل ملاحظه می‌کنید، متد SpecificationRequet واقع در Adaptee می‌تواند نیاز Client را برطرف نماید، اما Client،چیزی را که مشاهده می‌کند اینترفیس Itarget می‌باشد، به عبارتی Client بطور مستقیم نمی‌تواند با Adaptee ارتباط برقرار کند، بنابراین اگر بخواهیم از طریق Itarget نیاز Client را برطرف نماییم، لازم است کلاسی بین Itarget و Adaptee به جهت تبادل اطلاعات ایجاد کنیم، که Adapter نامیده می‌شود. حال در روش Class Adapter، کلاس Adapter جهت تبادل اطلاعات بین ITarget و Adaptee هر دو را در خود Implement می‌نماید، به عبارتی از هر دو مشتق (Inherit) می‌شود.
در ادامه شکل بالا را بصورت کد پیاده سازی می‌نماییم.

```
class Adaptee
    {
        public void SpecificationRequest()
        {
            Console.WriteLine("SpecificationRequest() is called");
        }
    }
```

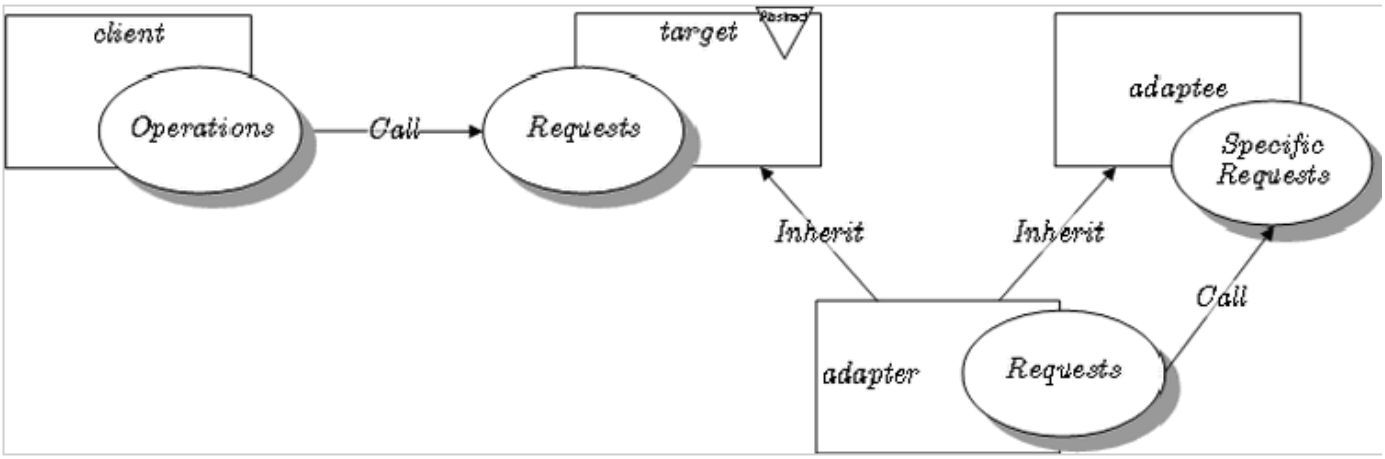
```
interface ITarget
    {
        void Request();

    }
```

```
class Adapter:Adaptee, ITarget
    {
        public void Request()
        {
            SpecificationRequest();
        }
    }
```

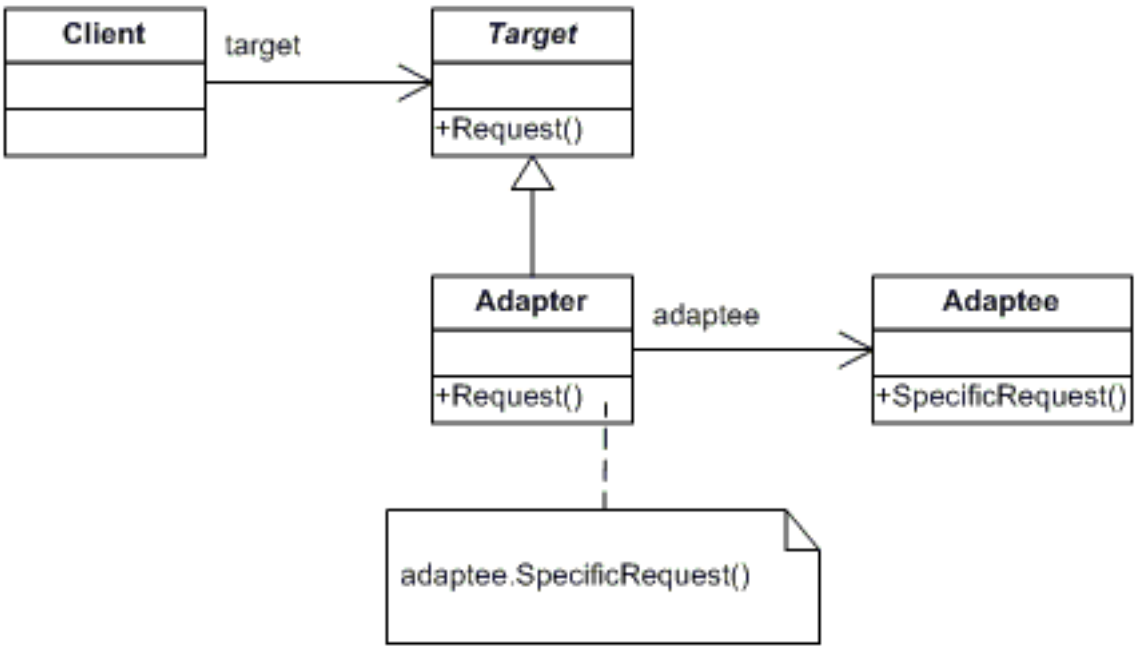
```
class MainApp
{
    static void Main()
    {
        ITarget target = new Adapter();
        target.Request();
```

```
        Console.ReadKey();
    }
}
```

سادگی کد، روش Class Adapter را قابل درک می‌نماید،نکته مهم در کد بالا،متد Request در کلاس Adapter و نحوه فراخوانی متد SpecificationRequest در آن می‌باشد.
شکل زیر که از سایت Wikipedia گرفته شده است،به خوبی نحوه فراخوانی را مشخص می‌نماید:



روش Object Adapter:
می دانیم در زبان برنامه نویسی C# هر کلاس فقط می‌تواند از یک کلاس دیگر Inherit شود، به طوری که هر کلاس نمی‌تواند بیش از یک کلاس Parent داشته باشد، بنابراین اگر Client شما بخواهد از امکانات و قابلیت‌های چندین کلاس Adaptee استفاده نماید، روش Class Adapter نمی‌تواند پاسخگوی نیازتان باشد، بلکه می‌بایست از روش Object Adapter استفاده نمایید.
شکل زیر بیانگر روش Object Adapter می‌باشد:



همانطور که در شکل ملاحظه می‌کنید، در این روش کلاس Adapter به جای Inherit نمودن از کلاس Adaptee، آبجکتی از کلاس Adaptee را در خود ایجاد می‌نماید، بنابراین با این روش شما می‌توانید به چندین Adaptee از طریق کلاس Adapter دسترسی داشته باشید.

پیاده سازی کدی شکل بالا به شرح ذیل می‌باشد:

```
class Adaptee
    {
        public void SpecificRequest()
        {
            MessageBox.Show("Called SpecificRequest()");
        }
    }
```

```
interface ITarget
    {
        void Request();

    }
```

```
class Adapter: ITarget
    {
        private Adaptee _adptee = new Adaptee();

        public void Request()
        {
            _adptee.SpecificationRequest();
        }
    }
```

```
class MainApp
{
    static void Main()
    {
        ITarget target = new Adapter();
        target.Request();

        Console.ReadKey();
```

```
    }
}
```

برای درک تفاوت Class Adapter و Object Adapter ، پیاده سازی کلاس Adapter را مشاهده نمایید، که در کد بالا به جای Inherit نمودن از کلاس Adaptee ، آبجکت آن را ایجاد نمودیم. واضح است که Object Adapter انعطاف پذیرتر نسبت به Class Adapter می‌باشد.
امیدوارم مطلب فوق مفید واقع شود

نظرات خوانندگان

نویسنده: حسین کهزادی
تاریخ: ۱۳۹۲/۱۱/۱۴ ۱۵:۸

باتشکر از مطلب بسیار خوبتون
اگر مطلب رو مانند الگوی composite با یک مثال ساده و کوچک توضیح میدادید بسیار قابل فهم‌تر میشد

عنوان: **الگوی Composite**
نویسنده: فرهاد فرهمندخواه
تاریخ: ۱۳۹۲/۰۹/۲۴ ۱۱:۲۰
آدرس: www.dotnettips.info
گروه‌ها: Design patterns

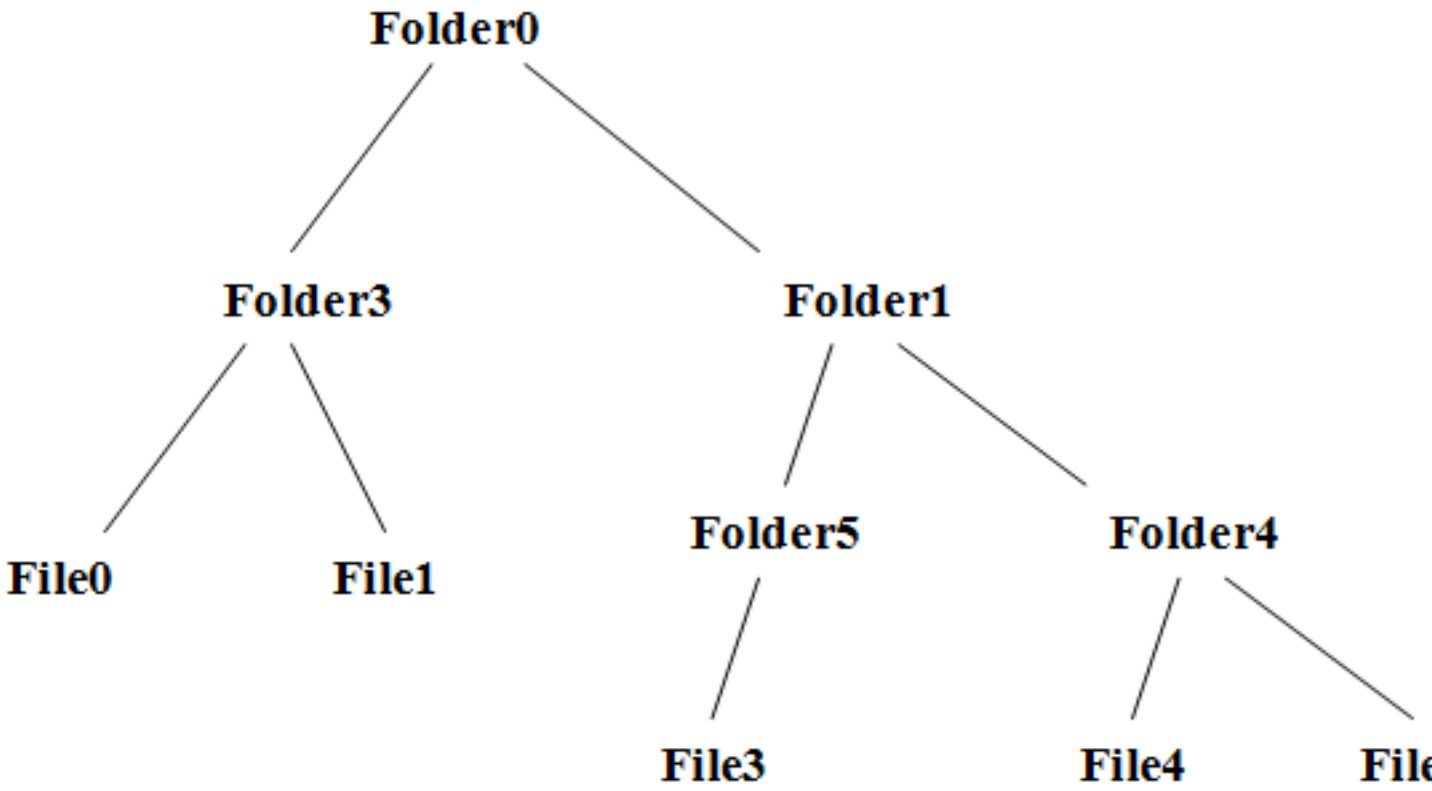
الگوی Composite یکی دیگر از الگوهای ساختاری می‌باشد که قصد داریم در این مقاله آن را بررسی نماییم.
الگوی Composite در عمل یک Collection Pattern (الگوی مجموعه ای) است. که می‌توان در درون آن ترکیبی از زیر مجموعه‌های مختلف را قرار داد و سپس هر زیر مجموعه را به نوبه خود فراخوانی نمود.به بیان دیگر الگوی Composite به ما کمک می‌کند که در یک ساختار درختی بتوانیم مجموعه ای (Collection ی)،از بخشی از آبجکتهای سلسله مراتبی را نمایش دهیم. این الگو به Client اجازه می‌دهد، که رفتار یکسانی نسبت به یک Collection ی از آبجکتها یا یک آبجکت تنها داشته باشد.

مثالهای متعددی می‌توان از الگوی Composite زد، که در ذیل به چند نمونه از آنها می‌پردازیم:

نمونه اول: همانطور که می‌دانیم یک سازمان از بخشهای مختلفی تشکیل شده است، که بصورت سلسله مراتبی با یکدیگر در ارتباط می‌باشند، چنانچه بخواهیم بخشها و زیر مجموعه‌های تابعه آنها را بصورت آبجکت نگهداری نماییم، یکی از بهترین الگوهای پیشنهاد شده الگوی Composite می‌باشد.

نمونه دوم: در بحث حسابداری،یک حساب کل از چندین حساب معین تشکیل شده است و هر حساب معین نیز از چندین سرفصل حسابداری تشکیل می‌شود. بنابراین برای نگهداری آبجکتهای معین مرتبط به حساب کل، می‌توان آنها را در یک Collection قرار داد. و هر حساب معین را می‌توان،در صورت داشتن چندین سرفصل در مجموعه خود به عنوان یک Collection در نظر گرفت. برای دسترسی به هر حساب معین و سرفصل‌های زیر مجموعه آن نیز می‌توان از الگوی Composite استفاده نمود.

نمونه سوم: یک File System را در نظر بگیرید،که ساختارش از File و Folder تشکیل شده است. و می‌تواند یک ساختار سلسله مراتبی داشته باشد.بطوریکه درون هر Folder می‌تواند یک یا چند File یا Folder قرار گیرد. و در درون Folderهای زیر مجموعه می‌توان چندین File یا Folder دیگر قرار داد.اگر بخواهیم به عنوان نمونه شکل ساختار درختی File و فولدر را نمایش دهیم بصورت زیر خواهد بود:

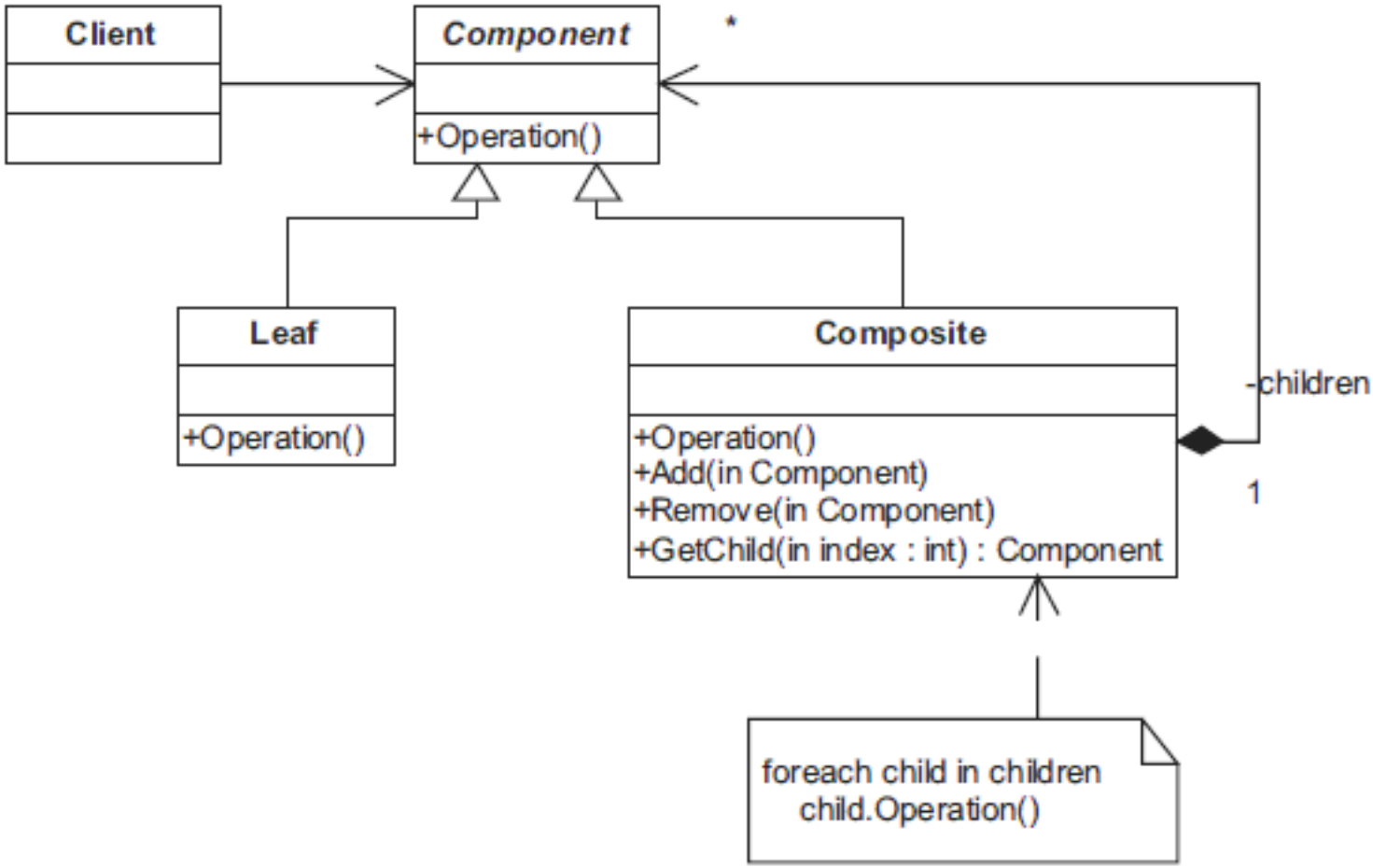


در ساختار درختی به Folder شاخه یا Branch گویند، چون می‌تواند زیر شاخه‌های دیگری نیز در خود داشته باشد. و به File برگ یا Leaf گویند.برگ نمی‌تواند زیر مجموعه ای داشته باشد. در واقع برگ (Leaf) بیانگر انتهای یک شاخه می‌باشد.

نمونه آخر:می توان به ساختار منوها در برنامه‌ها اشاره نمود.هر منو می‌تواند شامل چندین زیر منو باشد. و همان زیر منوها می‌توانند از چندین زیر منوی دیگر تشکیل شوند. این ساختار نیز یک ساختار سلسله مراتبی می‌باشد، و برای نگهداری آبجکتهای یک مجموعه می‌توان از الگوی Composite استفاده نمود.

الگوی Composite از سه Component اصلی تشکیل شده است،که یکایک آنها را بررسی می‌کنیم:
Component: کلاس پایه ای است که در آن متدها یا Functionalityهای مشترک تعریف می‌گردد. Component می‌تواند یک Abstract Class یا Interface باشد.
Leaf : به آبجکتهای گفته می‌شود که هیچ Child ی ندارند. و فقط یک آبجکت مستقل تنها می‌باشد. کلاس Leaf متدهای مشترک تعریف شده در Component را پیاده سازی می‌کند.اگر مثال File و Folder را بخاطر آورید،File یک آبجکت از نوع Leaf است چون نمی‌تواند هیچ فرزندی داشته باشد و یک آبجکت تنها می‌باشد.
Composite: کلاس فوق Collection ی از آبجکتها را در خود نگهداری می‌کند، به عبارتی در Composite می‌توان بخشی از ساختار درختی را قرار داد، که این ساختار می‌تواند ترکیبی از آبجکتهای Leaf و Composite باشد. در مثال File و Folder، یک Folder را می‌توان به عنوان Composite در نظر گرفت،زیرا که یک Folder می‌تواند چندین File یا Folder را در خود جای دهد. در کلاس Composite معمولا متدهایی همچون Add (افزودن Child)،Remove (حذف یک Child) و غیرو... وجود دارد.

کلاس Leaf و کلاس Composite از کلاس Component ارث بری (Inherit) می‌شوند.
شکل زیر بیانگر الگوی Composite می‌باشد:



توصیف شکل: طبق تعاریف گفته شده، دو کلاس Leaf و Composite از Component، Inherit شده اند. و Client نیز فقط متدهای مشترک تعریف شده در Component را مشاهده می‌کند، به عبارتی Client رفتار یکسانی نسبت به Leaf و Composite خواهد داشت.

برای درک بیشتر الگوی Composite مثالی را بررسی می‌کنیم، فرض کنید در کلاس Component متدی به نام Display را تعریف می‌کنیم،بطوریکه نام آبجکت را نمایش دهد.بنابراین خواهیم داشت:

اینترفیسی را برای Component در نظر می‌گیریم، و متد Display را در آن تعریف می‌کنیم:

```
public interface Icomponent
    {
        void Display(int depth);
    }
```

در کلاس Leaf، اینترفیس IComponent را پیاده سازی می‌نماییم:

```
public class Leaf:Icomponent
    {
        private String name = string.Empty;
        public Leaf(string name)
        {
            this.name = name;
        }

        public void Display(int depth)
        {

            Console.WriteLine(new String('-', depth) + ' ' + name);
```

```
        }
    }
```

در کلاس Composite نیز اینترفیس IComponent را پیاده سازی می‌نماییم، با این تفاوت که متدهای Add و Remove را نیز در کلاس Composite اضافه می‌کنیم، چون قبلا هم گفته بودیم، Composite در حکم یک Collection می‌باشد، بنابراین می‌بایست قابلیت حذف و اضافه نمود آبجکت در خود را داشته باشد. پیاده سازی متد Display در آن بصورت Recursive (بازگشتی) میباشد. و علتش این است که بتوانیم ساختار سلسله مراتبی را بازیابی نماییم.

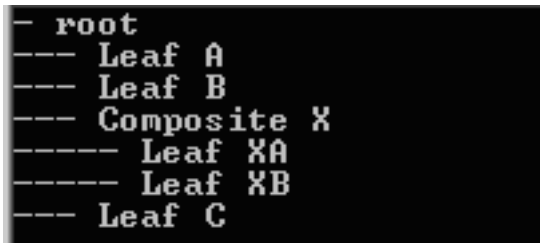
```
public class Composite:Icomponent
    {
        private List<Icomponent> _children = new List<Icomponent>();
        private String name = String.Empty;

        public Composite(String sname)
        {
            this.name = sname;
        }

        public void Add(Icomponent component)
        {

            _children.Add(component);

        }


        public void Remove(Icomponent component)
        {

            _children.Remove(component);

        }


        public void Display(int depth)
        {

            Console.WriteLine(new String('-', depth) + ' ' + name);



            // Recursively display child nodes

            foreach (Icomponent component in _children)
            {

                component.Display(depth + 2);

            }

        }
    }
```

در ادامه بوسیله چندین آبجکت Leaf و Composite یک ساختار درختی را ایجاد می‌کنیم.

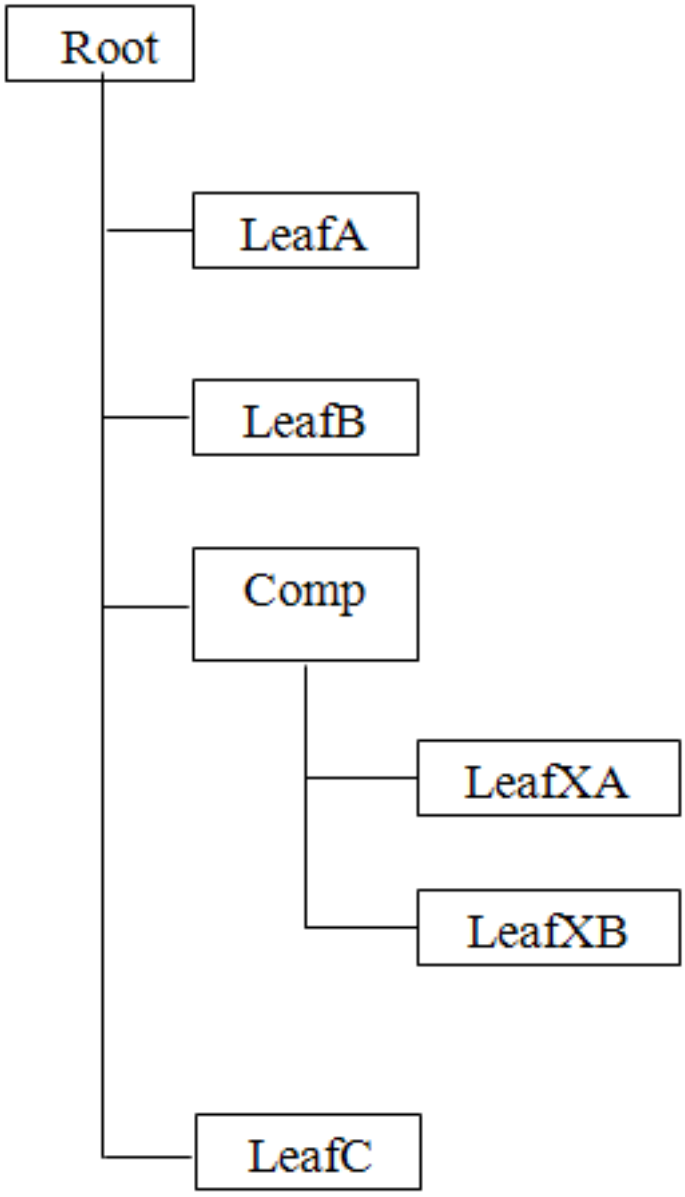
```
class Program
    {
        static void Main(string[] args)
        {
            // Create a tree structure

            Composite root = new Composite("root");

            root.Add(new Leaf("Leaf A"));

            root.Add(new Leaf("Leaf B"));



            Composite comp = new Composite("Composite X");

            comp.Add(new Leaf("Leaf XA"));
```

```
            comp.Add(new Leaf("Leaf XB"));

            root.Add(comp);
            root.Add(new Leaf("Leaf C"));

            // Add and remove a leaf
            Leaf leaf = new Leaf("Leaf D");
            root.Add(leaf);
            root.Remove(leaf);

            // Recursively display tree
            root.Display(1);
            Console.ReadKey();
        }
    }
```

در ابتدا یک آبجکت Composite ایجاد می‌کنیم و آن را به عنوان ریشه در نظر گرفته و نام آن را Root قرار می‌دهیم. سپس دو آبجکت LeafA و LeafB را به آن می‌افزاییم، در ادامه آبجکت Composite دیگری به نام Comp ایجاد می‌کنیم، که خود دارای دو فرزند به نامهای LeafXA و LeafXB می باشد. و سر آخر هم یک آبجکت LeafC ایجاد می‌کنیم.
آبجکت LeafD صرفا جهت نمایش افزودن و حذف کردن آن در یک آبجکت Composite نوشته شده است. برای این که بتوانیم ساختار سلسله مراتبی کد بالا را مشاهده نماییم، متد Root.Display را اجرا می‌کنیم و خروجی آن بصورت زیر خواهد بود:



اگر بخواهیم،شکل درختی آن را تصور کنیم بصورت زیر خواهد بود:



درپایان باید بگویم،که نمونه کد بالا را می‌توان به ساختار File و Folder نیز تعمیم داد، بطوریکه متدهای مشترک بین File و Folder را در اینترفیس IComponent تعریف می‌کنیم و بطور جداگانه در کلاسهای Composite و Leaf پیاده سازی می‌کنیم.
امیدوارم توضیحات داده شده در مورد الگوی Composite مفید واقع شود.

عنوان: **Design Pattern: Factory**
نویسنده: مهسا حسن کاشی
تاریخ: ۱۳۹۲/۱۱/۲۶ ۲۲:۱۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Design patterns, SOLID, Object Oriented Design, ServiceFactory, Patterns, Software architecture

الگوهای طراحی، سندها و راه حلهای از پیش تعریف شده و تست شده‌ای برای مسائل و مشکلات روزمره‌ی برنامه نویسی می‌باشند که هر روزه ما را درگیر خودشان می‌کنند. هر چقدر مقیاس پروژه وسیعتر و تعداد کلاسها و اشیاء بزرگتر باشند، درگیری برنامه نویس و چالش برای مرتب سازی و خوانایی برنامه و همچنین بالا بردن کارآیی و امنیت افزون‌تر می‌شود. از همین رو استفاده از ساختارهایی تست شده برای سناریوهای یکسان، امری واجب تلقی می‌شود.

الگوهای طراحی از لحاظ سناریو، به سه گروه عمده تقسیم می‌شوند:

-1 تکوینی: هر چقدر تعداد کلاسها در یک پروژه زیاد شود، به مراتب تعداد اشیاء ساخته شده از آن نیز افزوده شده و پیچیدگی و درگیری نیز افزایش می‌یابد. راه حل‌هایی از این دست، تمرکز بر روی مرکزیت دادن به کلاسها با استفاده از رابطها و کپسوله نمودن (پنهان سازی) اشیاء دارد.

-2 ساختاری: گاهی در پروژه‌ها پیش می‌آید که می‌خواهیم ارتباط بین دو کلاس را تغییر دهیم. از این رو امکان از هم پاشی اجزایِ دیگر پروژه پیش می‌آید. راه حلهای ساختاری، سعی در حفظ انسجام پروژه در برابر این دست از تغییرات را دارند.

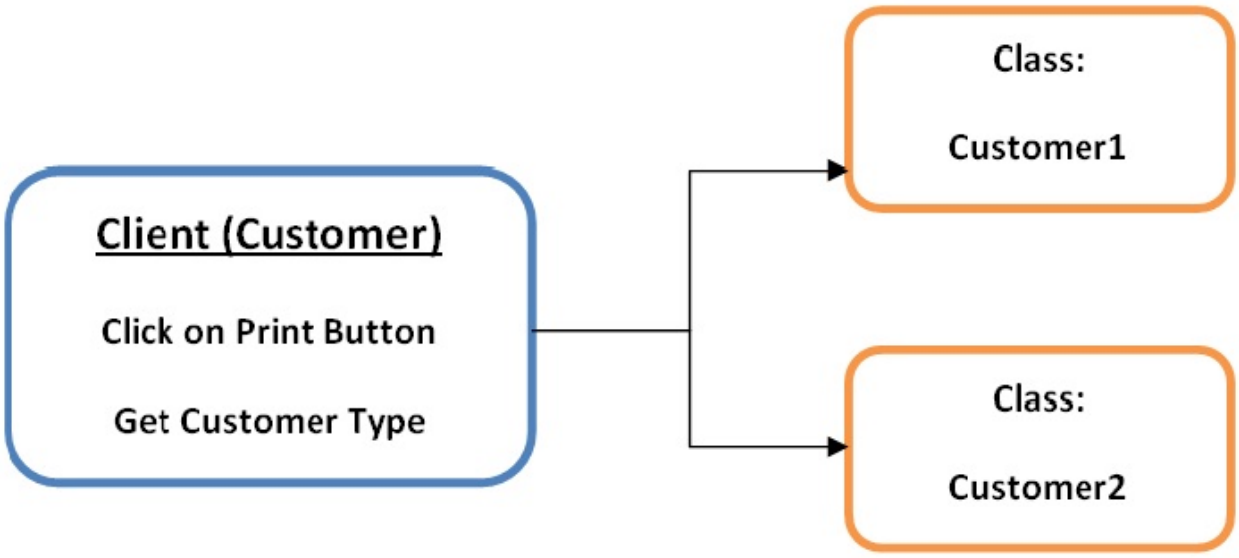
-3 رفتاری: گاهی بنا به مصلحت و نیاز مشتری، رفتار یک کلاس می‌بایستی تغییر نماید. مثلا چنانچه کلاسی برای ارائه صورتحساب داریم و در آن میزان مالیات 30% لحاظ شده است، حال این درصد باید به عددی دیگر تغییر کند و یا پایگاه داده به جای مشاهده‌ی تعدادِ معدودی گره از درخت، حال می‌بایست تمام گره‌ها را ارائه نماید.

الگوی فکتوری:

الگوی فکتوری در دستهء اول قرار می‌گیرد. من در اینجا به نمونه‌ای از مشکلاتی که این الگو حل می‌نماید، اشاره می‌کنم:

فرض کنید یک شرکت بزرگ قصد دارد تا جزییات کامل خرید هر مشتری را با زدن دکمه چاپ ارسال نماید. چنین شرکت بزرگی بر اساس سیاستهای داخلی، بر حسب میزان خرید، مشتریان را به چند گروه مشتری معمولی و مشتری ممتاز تقسیم می‌نماید. در نتیجه نمایش جزییات برای آنها با احتساب میزان تخفیف و به عنوان مثال تعداد فیلدهایی که برای آنها در نظر گرفته شده است، تفاوت دارد. بنابراین برای هر نوع مشتری یک کلاس وجود دارد.

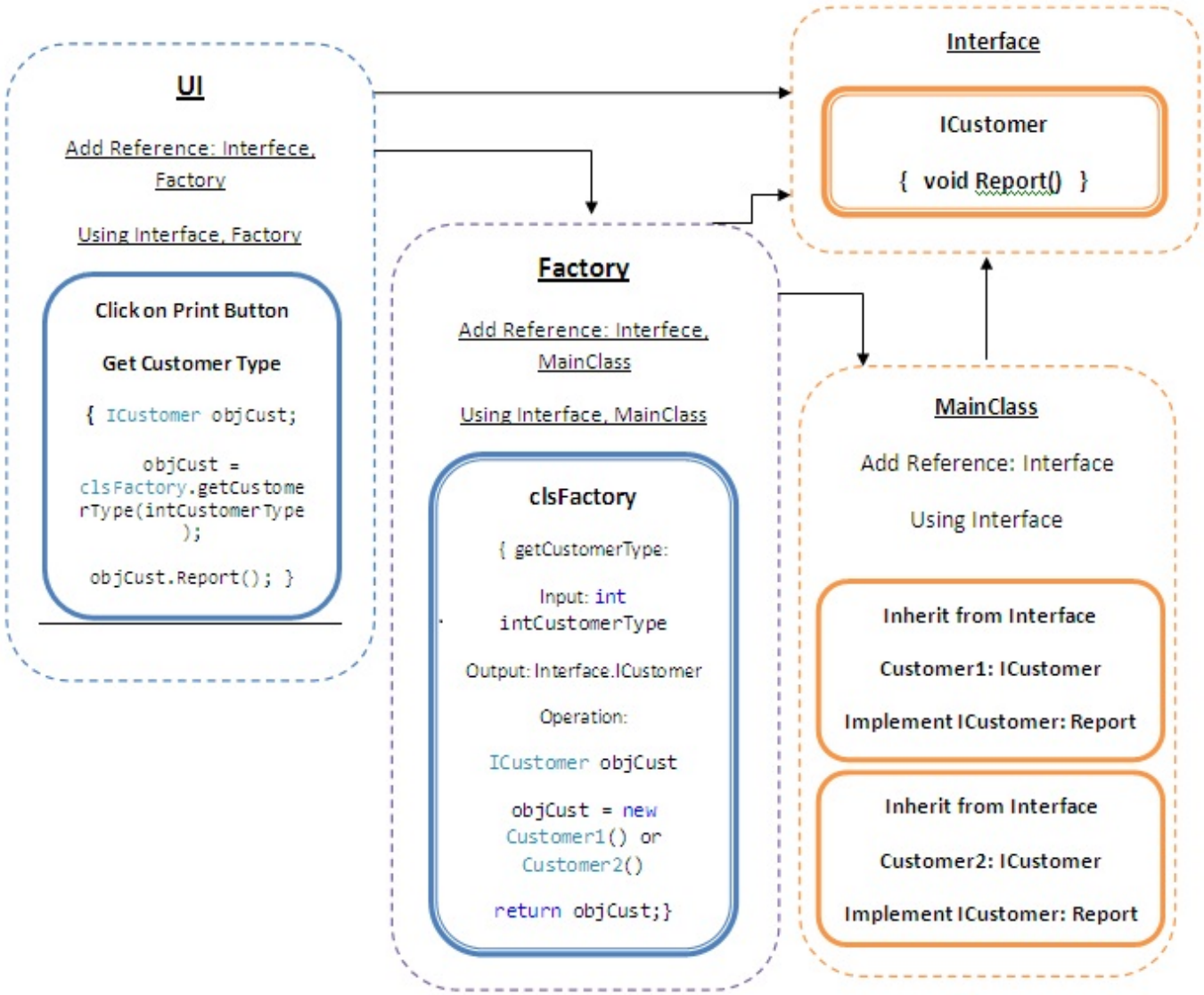
یک راه این است که با کلیک روی دکمه‌ی چاپ، نوع مشتری تشخیص داده شود و به ازای نوع مشتری، یک شیء از کلاس مشخص شده برای همان نوع ساخته شود.




```
// Get Customer Type from Customer click on Print Button
            int customerType = 0;

            // Create Object without instantiation
            object obj;


            //Instantiate obj according to customer Type
            if (customerType == 1)
            {
                obj = new Customer1();
            }
            else if (customerType == 2)
            {
                obj = new Customer2();
            }
            // Problem:
            //          1: Scattered New Keywords
            //          2: Client side is aware of Customer Type
```

همانگونه که مشاهده می‌نمایید در این سبک کدنویسی غیرحرفه‌ای، مشکلاتی مشهود است که قابل اغماض نیستند. در ابتدا سمت کلاینت دسترسی مستقیم به کلاسها دارد و همانگونه که در شکل بالا قابل مشاهده است کلاینت مستقیما به کلاس وصل است. مشکل دوم عدم پنهان سازی کلاس از دید مشتری است.

راه حل: این مشکل با استفاده از الگوی فکتوری قابل حل است. با استناد به الگوی فکتوری، کلاینت تنها به کلاس فکتوری و یک اینترفیس دسترسی دارد و کلاسهای فکتوری و اینترفیس، حق دسترسی به کلاسهای اصلی برنامه را دارند.



گام نخست: در ابتدا یک class library به نام Interface ساخته و در آن یک کلاس با نام ICustomer می سازیم که متد Report() را معرفی می‌نماید.

Interface//

```
namespace Interface
{
    public interface ICustomer
    {
        void Report();
    }
}
```

گام دوم: یک class library به نام MainClass ساخته و با Add Reference کلاس Interface را اضافه نموده، در آن دو کلاس با نام Customer1, Customer2 می‌سازیم و using Interface را Import می‌نماییم. هر دو کلاس از ICustomer ارث می‌برند و سپس متد Report() را در هر دو کلاس Implement می‌نماییم.

```
// Customer1
using System;
```

```
using Interface;

namespace MainClass
{
    public class Customer1 : ICustomer
    {
        public void Report()
        {
            Console.WriteLine("این گزارش مخصوص مشتری نوع اول است");
        }
    }
}

//Customer2
using System;
using Interface;

namespace MainClass
{
   public class Customer2 : ICustomer
    {
        public void Report()
        {
            Console.WriteLine("این گزارش مخصوص مشتری نوع دوم است");
        }
    }
}
```

گام سوم: یک class library به نام FactoryClass ساخته و با Add Reference کلاس Interface, MainClass را اضافه نموده، در آن یک کلاس با نام clsFactory می سازیم و using Interface, using MainClass را Import می‌نماییم. پس از آن یک متد با نام getCustomerType ساخته که ورودی آن نوع مشتری از نوع int است و خروجی آن از نوع Interface-ICustomer و بر اساس کد نوع مشتری object را از کلاس Customer1 و یا Customer2 می‌سازیم و آن را return می نماییم.

```
//Factory
using System;
using Interface;
using MainClass;

namespace FactoryClass
{
    public class clsFactory
    {
        static public ICustomer getCustomerType(int intCustomerType)
        {
            ICustomer objCust;
            if (intCustomerType == 1)
            {
                objCust = new Customer1();
            }
            else if (intCustomerType == 2)
            {
                objCust = new Customer2();
            }
            else
            {
                return null;
            }
            return objCust;
        }
    }
}
```

گام چهارم (آخر): در قسمت UI Client، کد نوع مشتری را از کاربر دریافت کرده و با Add Reference کلاس Interface, FactoryClass را اضافه نموده (دقت نمایید هیچ دسترسی به کلاس‌های اصلی وجود ندارد)، و using Interface, using FactoryClass را Import می‌نماییم. از clsFactory تابع getCustomerType را فراخوانی نموده (به آن کد نوع مشتری را پاس می‌دهیم) و خروجی آن را که از نوع اینترفیس است به یک object از نوع ICustomer نسبت می‌دهیم. سپس از این object متد Report را فراخوانی می‌نماییم. همانطور که از شکل و کدها مشخص است، هیچ رابطه ای بین UI(Client) و کلاسهای اصلی برقرار نیست.

```
//UI (Client)
using System;
using FactoryClass;
using Interface;

namespace DesignPattern
{
    class Program
    {
        static void Main(string[] args)
        {
            int intCustomerType = 0;
            ICustomer objCust;
            Console.WriteLine("نوع مشتری را وارد نمایید");
            intCustomerType = Convert.ToInt16(Console.ReadLine());
            objCust = clsFactory.getCustomerType(intCustomerType);
            objCust.Report();
            Console.ReadLine();
        }
    }
}
```

عنوان: **آموزش LightInject IoC Container - قسمت 1**
نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۳/۰۲/۰۸ ۱۶:۳۰
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, Dependency Injection, Ioc, Dependency Inversion, LightInject

LightInject در حال حاضر یکی از قدرتمندترین IoC Containerها است که از لحاظ سرعت و کارآیی در بالاترین جایگاه در میان IoC Containerهای موجود قرار دارد. جهت بررسی کارایی IoC Containerها می‌توانید به این لینک مراجعه کنید . LightInject یک IoC Container فوق العاده سبک وزن می‌باشد که تمامی قابلیت‌های متداولی که از یک Service Container انتظار می‌رود را شامل می‌شود. تنها شامل یک فایل .cs می‌باشد که تمامی کدهای آن در همین یک فایل نوشته شده‌اند. در پروژه‌های کوچک تا بزرگ بدون از دست دادن کارآیی، با بالاترین سرعت ممکن عمل تزریق وابستگی را انجام می‌دهد. در این مجموعه مقالات به بررسی کامل این IoC Container می‌پردازیم و تمامی قابلیت‌های آن را آموزش می‌دهیم.

نحوه نصب و راه اندازی LightInject

در پنجره Package Manager Console می‌توانید با نوشتن دستور ذیل، نسخه باینری آن را نصب کنید که به فایل .dll آن Reference میدهد.

```
PM> Install-Package LightInject
```

همچنین می‌توانید توسط دستور ذیل فایل .cs آن را به پروژه اضافه نمایید.

```
PM> Install-Package LightInject.Source
```

آماده سازی پروژه نمونه

قبل از شروع کار با LightInject، یک پروژه Windows Forms Application را با ساختار کلاس‌های ذیل ایجاد نمایید. (در مقالات بعدی و پس از آموزش کامل LightInject نحوه استفاده از آن را در ASP.NET MVC نیز آموزش می‌دهیم)

```
public class PersonModel
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public string Family { get; set; }
        public DateTime Birth { get; set; }
    }

    public interface IRepository<T> where T:class
    {
        void Insert(T entity);
        IEnumerable<T> FindAll();
    }

    public interface IPersonRepository:IRepository<PersonModel>
    {
    }

    public class PersonRepository:IPersonRepository
    {
        public void Insert(PersonModel entity)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<PersonModel> FindAll()
        {
            throw new NotImplementedException();
        }
    }

    public interface IPersonService
    {
        void Insert(PersonModel entity);
        IEnumerable<PersonModel> FindAll();
```

```
    }

    public class PersonService:IPersonService
    {
        private readonly IPersonRepository _personRepository;

        public PersonService(IPersonRepository personRepository)
        {
            _personRepository = personRepository;
        }

        public void Insert(PersonModel entity)
        {
            _personRepository.Insert(entity);
        }

        public IEnumerable<PersonModel> FindAll()
        {
            return _personRepository.FindAll();
        }
    }
```

توضیحات PersonModel: ساختار داده ای جدول Person در سمت Application، که در لایه Domain Model ایجاد می‌گردد. **توجه:** جهت سهولت تست و تسریع کدنویسی از لایه بندی و از کلاس‌های ViewModel استفاده نکردیم. **IRepository:** یک Interface عمومی برای تمامی Interfaceهای مربوط به Repository که عملیات مربوط به پایگاه داده مثل بروزرسانی و واکشی اطلاعات را انجام می‌دهند. **IPersonRepository:** واسط بین لایه Service و لایه Repository می‌باشد. **PersonRepository:** پیاده سازی واقعی عملیات مربوط به پایگاه داده برای PersonModel می‌باشد. به کلاسهایی که حاوی پیاده سازی واقعی کد می‌باشند Concrete Class می‌گویند. **IPersonService:** واسط بین رابط کاربری و لایه سرویس می‌باشد. رابط کاربری به جای دسترسی مستقیم به PersonService از IPersonService استفاده می‌کند. **PersonService:** دریافت درخواست‌های رابط کاربری و بررسی قوانین تجاری، سپس ارسال درخواست به لایه Repository در صورت صحت درخواست، و در نهایت ارسال پاسخ دریافتی به رابط کاربری. در واقع واسطی بین Repository و UI می‌باشد.

پس از ایجاد ساختار فوق کد مربوط به Form1 را بصورت زیر تغییر دهید.

```
public partial class Form1 : Form
    {
        private readonly IPersonService _personService;
        public Form1(IPersonService personService)
        {
            _personService = personService;
            InitializeComponent();
        }
    }
```

توضیحات

در کد فوق به منظور ارتباط با سرویس از IPersonService استفاده نمودیم که به عنوان پارامتر ورودی برای سازنده Form1 تعریف شده است. حتما با Dependency Inversion و انواع Dependency Injection آشنا هستید که به سراغ مطالعه این مقاله آمدید و علت این نوع کدنویسی را هم می‌دانید. بنابراین توضیح بیشتری در این مورد نمی‌دهم.

حال اگر برنامه را اجرا کنید در Program.cs با خطای عدم وجود سازنده بدون پارامتر برای Form1 مواجه می‌شوید که کد آن را باید به صورت زیر تغییر می‌دهیم.

```
static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            var container = new ServiceContainer();
            container.Register<IPersonService, PersonService>();
            container.Register<IPersonRepository, PersonRepository>();
            Application.Run(new Form1(container.GetInstance<IPersonService>()));
        }
```

توضیحات

کلاس ServiceContainer وظیفه‌ی Register کردن یک کلاس را برای یک Interface دارد. زمانی که می‌خواهیم Form1 را نمونه سازی نماییم و Application را راه اندازی کنیم، باید نمونه ای را از جنس IPersonService ایجاد نموده و به سازنده‌ی Form1 ارسال نماییم. با رعایت اصل DIP، نمونه سازی واقعی یک کلاس لایه دیگر، نباید در داخل کلاس‌های لایه جاری انجام شود. برای این منظور از شیء container استفاده نمودیم و توسط متد GetInstance، نمونه‌ای از جنس IPersonService را ایجاد نموده و به

Form1 پاس دادیم. حال container از کجا متوجه می‌شود که چه کلاسی را برای IPersonService نمونه سازی نماید؟
در خطوط قبلی توسط متد Register، کلاس PersonService را برای IPersonService ثبت نمودیم. container نیز برای نمونه سازی به کلاس هایی که برایش Register نمودیم مراجعه می‌نماید و نمونه سازی را انجام می‌دهد. جهت استفاده از PersonService به پارامتر ورودی IPersonRepository برای سازنده‌ی آن نیاز داریم که کلاس PersonRepository را برای IPersonRepository ثبت کردیم.
حال اگر برنامه را اجرا کنید، به درستی اجرا خواهد شد. برنامه را متوقف کنید و به کد موجود در Program.cs مراجعه نموده و دو خط مربوط به Register را Comment نمایید. سپس برنامه را اجرا کنید و خطای تولید شده را ببینید. این خطا بیان می‌کند که امکان نمونه سازی برای IPersonService را ندارد. چون قبلا هیچ کلاسی را برای آن Register نکرده ایم. **Named Services**
در برخی مواقع، بیش از یک کلاس وجود دارند که ممکن است از یک Interface ارث بری نمایند. در این حالت و در زمان Register، باید به ServiceContainer بگوییم که کدام کلاس را باید نمونه سازی نماید. برای بررسی این موضوع، کلاسهای زیر را به ساختار پروژه اضافه نمایید.

```
public class WorkerModel:PersonModel
    {
        public ManagerModel Manager { get; set; }
    }

    public class ManagerModel:PersonModel
    {
        public IEnumerable<WorkerModel> Workers { get; set; }
    }

    public class WorkerRepository:IPersonRepository
    {
        public void Insert(PersonModel entity)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<PersonModel> FindAll()
        {
            throw new NotImplementedException();
        }
    }

    public class ManagerRepository:IPersonRepository
    {
        public void Insert(PersonModel entity)
        {
            throw new NotImplementedException();
        }

        public IEnumerable<PersonModel> FindAll()
        {
            throw new NotImplementedException();
        }
    }

    public class WorkerService:IPersonService
    {
        private readonly IPersonRepository _personRepository;

        public WorkerService(IPersonRepository personRepository)
        {
            _personRepository = personRepository;
        }

        public void Insert(PersonModel entity)
        {
            var worker = entity as WorkerModel;
            _personRepository.Insert(worker);
        }

        public IEnumerable<PersonModel> FindAll()
        {
            return _personRepository.FindAll();
        }
    }

    public class ManagerService:IPersonService
    {
        private readonly IPersonRepository _personRepository;
```

```
        public ManagerService(IPersonRepository personRepository)
        {
            _personRepository = personRepository;
        }

        public void Insert(PersonModel entity)
        {
            var manager = entity as ManagerModel;
            _personRepository.Insert(manager);
        }

        public IEnumerable<PersonModel> FindAll()
        {
            return _personRepository.FindAll();
        }
    }
```

توضیحات

دو کلاس Manager و Worker به همراه سرویس‌ها و Repository هایشان اضافه شده اند که از IPersonService و IPersonRepository مشتق شده اند.

حال کد کلاس Program را به صورت زیر تغییر می‌دهیم

```
...
 var container = new ServiceContainer();
            container.Register<IPersonService, PersonService>();
            container.Register<IPersonService, WorkerService>();
            container.Register<IPersonRepository, PersonRepository>();
            container.Register<IPersonRepository, WorkerRepository>();
            Application.Run(new Form1(container.GetInstance<IPersonService>()));
```

توضیحات

در کد فوق، چون WorkerService بعد از PersonService ثبت یا Register شده است، LightInject در زمان ارسال پارامتر به Form1، نمونه ای از کلاس WorkerService را ایجاد میکند. اما اگر بخواهیم از کلاس PersonService نمونه سازی نماید باید کد را به صورت زیر تغییر دهیم.

```
...
            container.Register<IPersonService, PersonService>("PersonService");
            container.Register<IPersonService, WorkerService>();
            container.Register<IPersonRepository, PersonRepository>();
            container.Register<IPersonRepository, WorkerRepository>();
            Application.Run(new Form1(container.GetInstance<IPersonService>("PersonService")));
```

همانطور که مشاهده می‌نمایید، در زمان Register نامی را به آن اختصاص دادیم که در زمان نمونه سازی از این نام استفاده شده است.

اگر در زمان ثبت، نامی را به نمونه‌ی مورد نظر اختصاص داده باشیم، و فقط یک Register برای آن Interface معرفی نموده باشیم، در زمان نمونه سازی، LightInject آن نمونه را به عنوان سرویس پیش فرض در نظر می‌گیرد.

```
container.Register<IPersonService, PersonService>("PersonService");
  Application.Run(new Form1(container.GetInstance<IPersonService>()));
```

در کد فوق، چون برای IPersonService فقط یک کلاس برای نمونه سازی معرفی شده است، با فراخوانی متد GetInstance، حتی بدون ذکر نام، نمونه ای را از کلاس PersonService ایجاد می‌کند. **<IEnumerable<T>**

زمانی که چند کلاس را که از یک Interface مشتق شده اند، با هم Register می‌نمایید، LightInject این قابلیت را دارد که این کلاس‌های Register شده را در قالب یک لیست شمارشی برگردانید.

```
container.Register<IPersonService, PersonService>();
            container.Register<IPersonService, WorkerService>("WorkerService");
            var personList = container.GetInstance<IEnumerable<IPersonService>>();
```

در کد فوق لیستی با دو آیتم ایجاد می‌شود که یک آیتم از نوع PersonService و دیگری از نوع WorkerService می‌باشد. همچنین از کد زیر نیز می‌توانید استفاده کنید:

```
container.Register<IPersonService, PersonService>();
            container.Register<IPersonService, WorkerService>("WorkerService");
            var personList = container.GetAllInstances<IPersonService>();
```

به جای متد GetInstance از متد GetAllInstances استفاده شده است.

LightInject از Collectionهای زیر نیز پشتیبانی می‌نماید:

Array

<ICollection<T

<IList<T

<IReadOnlyCollection<T

<IReadOnlyList<T

Values توسط LightInject می‌توانید مقادیر ثابت را نیز تعریف کنید

```
container.RegisterInstance<string>("SomeValue");
            var value = container.GetInstance<string>();
```

متغیر value با رشته "SomeValue" مقداردهی می‌گردد. اگر چندین ثابت رشته ای داشته باشید می‌توانید نام جداگانه ای را به هر کدام اختصاص دهید و در زمان فراخوانی مقدار به آن نام اشاره کنید.

```
container.RegisterInstance<string>("SomeValue","String1");
            container.RegisterInstance<string>("OtherValue","String2");
            var value = container.GetInstance<string>("String2");
```

متغیر value با رشته "OtherValue" مقداردهی می‌گردد.

نظرات خوانندگان

نویسنده: احمد زاده
تاریخ: ۱۳۹۳/۰۲/۲۱ ۱:۲۷

ممنون از مطلب خوبتون
من یه مقایسه دیگه دیدم که اونجا گفته بود Ligth Inject از Instance Per Request پشتیبانی نمی‌کنه
میخواستم جایگزین Unity کنم برای حالتی که unit of work داریم و DBContext for per request اگر راهنمایی کنید، ممنون میشم

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۲/۲۱ ۱۰:۳۲

از حالت طول عمر PerRequestLifetime پشتیبانی می‌کند.

نویسنده: میثم خوشبخت
تاریخ: ۱۳۹۳/۰۲/۲۱ ۱۱:۱۴

خواهش می‌کنم
همانطور که آقای نصیری نیز عنوان کردند، از PerRequestLifeTime استفاده می‌شود که در مقاله بعدی در مورد آن صحبت خواهم کرد.

عنوان: Repository ها روی UnitOfWork ایده خوبی نیستند
نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۳/۰۲/۰۸ ۱۹:۳۰
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, Entity framework, ORM

در دنیای دات نت گرایشی برای تجزیه (abstract) کردن EF پشت الگوی Repository وجود دارد. این تمایل اساسا بد است و در ادامه سعی می‌کنم چرای آن را توضیح دهم.

پایه و اساس

عموما این باور وجود دارد که با استفاده از الگوی Repository می‌توانید (در مجموع) دسترسی به داده‌ها را از لایه دامنه (Domain) تفکیک کنید و "داده‌ها را بصورت سازگار و استوار عرضه کنید".

اگر به هر کدام از پیاده سازی‌های الگوی Repository در کنار UnitOfWork (EF) دقت کنید خواهید دید که تفکیک (decoupling) قابل ملاحظه ای وجود ندارد.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data;
using ContosoUniversity.Models;

namespace ContosoUniversity.DAL
{
    public class StudentRepository : IStudentRepository, IDisposable
    {
        private SchoolContext context;

        public StudentRepository(SchoolContext context)
        {
            this.context = context;
        }

        public IEnumerable<Student> GetStudents()
        {
            return context.Students.ToList();
        }

        public Student GetStudentByID(int id)
        {
            return context.Students.Find(id);
        }

        //<snip>
        public void Save()
        {
            context.SaveChanges();
        }
    }
}
```

این کلاس بدون SchoolContext نمی‌تواند وجود داشته باشد، پس دقیقا چه چیزی را در اینجا decouple کردیم؟ **هیچ چیز را!**

در این قطعه کد - از MSDN - چیزی که داریم یک پیاده سازی مجدد از LINQ است که مشکل کلاسیک Repository APIهای بی انتها را بدست می‌دهد. منظور از Repository APIهای بی انتها، متدهای جالبی مانند GetStudentById, GetStudentByBirthday, GetStudentByOrderNumber و غیره است.

اما این مشکل اساسی نیست. مشکل اصلی روتین ()Save است. این متد یک دانش آموز (Student) را ذخیره می‌کند .. اینطور بنظر می‌رسد. دیگر چه چیزی را ذخیره می‌کند؟ آیا می‌توانید حدس بزنید؟ من که نمی‌توانم .. بیشتر در ادامه.

UnitOfWork تراکنشی است یک UnitOfWork همانطور که از نامش بر می‌آید برای **انجام کاری** وجود دارد. این کار می‌تواند به

سادگی واکشی اطلاعات و نمایش آنها، و یا به پیچیدگی پردازش یک سفارش جدید باشد. هنگامی که شما از EntityFramework استفاده می‌کنید و یک DbContext را وهله سازی می‌کنید، در واقع یک UnitOfWork می‌سازید.

در EF می‌توانید با فراخوانی ()SubmitChanges تمام تغییرات را فلاش کرده و بازنشانی کنید (flush and reset). این کار بیت‌های مقایسه change tracker را تغییر می‌دهد. افزودن رکوردهای جدید، بروز رسانی و حذف آنها. هر چیزی که تعیین کرده باشید. و تمام این دستورات در یک تراکنش یا Transaction انجام می‌شوند.

یک Repository مطلقا یک UnitOfWork نیست

هر متد در یک Repository قرار است فرمانی اتمی (Atomic) باشد - چه واکشی اطلاعات و چه ذخیره آنها. مثلا می‌توانید یک Repository داشته باشید با نام SalesRepository که اطلاعات کاتالوگ شما را واکشی می‌کند، و یا یک سفارش جدید را ثبت می‌کند. منظور از فرمان‌های اتمیک این است، که هر متد تنها یک دستور را باید اجرا کند. تراکنشی وجود ندارد و امکاناتی مانند ردیابی تغییرات و غیره هم جایی ندارند.

یکی دیگر از مشکلات استفاده از Repositoryها این است که بزودی و به آسانی از کنترل خارج می‌شوند و نیاز به ارجاع دیگر مخازن پیدا می‌کنند. به دلیل اینکه مثلا نمی‌دانستید که SalesRepository نیاز به ارجاع ReportRepository داشته است (یا چیزی مانند این).

این مشکل به سرعت مشکل ساز می‌شود، و نیز به همین دلیل است که به UnitOfWork تمایل پیدا می‌کنیم.

بدترین کاری که می‌توانید انجام دهید: Repository<T> این الگو دیوانه وار است. این کار عملا انتزاعی از یک انتزاع دیگر است (abstraction of an abstraction). به قطعه کد زیر دقت کنید، که به دلیلی نامشخص بسیار هم محبوب است.

```
public class CustomerRepository : Repository < Customer > {
  public CustomerRepository(DbContext context){
    //a property on the base class
    this.DB = context;
  }

  //base class has Add/Save/Remove/Get/Fetch
}
```

در نگاه اول شاید بگویید مشکل این کلاس چیست؟ همه چیز را کپسوله می‌کند و کلاس پایه Repository هم به کانتکست دسترسی دارد. پس مشکل کجاست؟

مشکلات عدیده اند .. بگذارید نگاهی بیاندازیم.

آیا می‌دانید این DbContext از کجا آمده است؟

خیر، نمی‌دانید. این آبجکت به کلاس تزریق (Inject) می‌شود، و نمی‌دانید که چه متدی آن را باز کرده و به چه دلیلی. ایده اصلی پشت الگوی Repository استفاده مجدد از کد است. بدین منظور که مثلا برای عملیات CRUD از کلاسی پایه استفاده کنید تا برای هر موجودیت و فرمی نیاز به کدنویسی مجدد نباشد. برگ برنده این الگو نیز دقیقا همین است. مثلا اگر بخواهید از کدی در چند فرم مختلف استفاده کنید از این الگو استفاده میشد.

الگوی UnitOfWork همه چیز در نامش مشخص است. اگر قرار باشد آنرا بدین شکل تزریق کنید، نمی‌توانید بدانید که از کجا آمده است.

شناسه مشتری جدید را نیاز داشتم

کد بالا در CustomerRepository را در نظر بگیرید - که یک مشتری جدید را به دیتابیس اضافه می‌کند. اما CustomerID جدید چه می‌شود؟ مثلا به این شناسه نیاز دارید تا یک log بسازید. چه می‌کنید؟ گزینه‌های شما اینها هستند:

متد ()SubmitChanges را صدا بزنید تا تغییرات ثبت شوند و بتوانید به CustomerID جدید دسترسی پیدا کنید
CustomerRepository خود را باز کنید و متد پایه Add را بازنویسی (override) کنید. بدین منظور که پیش از بازگشت دادن، متد ()SubmitChanges را فراخوانی کند. این راه حلی است که MSDN به آن تشویق می‌کند، و بمبی ساعتی است که در انتظار انفجار است
تصمیم بگیرید که تمام متدهای Add/Remove/Save در مخازن شما باید ()SubmitChanges را فراخوانی کنند

مشکل را می‌بینید؟ مشکل در خود پیاده سازی است. در نظر بگیرید که چرا New Customer ID را نیاز دارید؟ احتمالا برای استفاده از آن در ثبت یک سفارش جدید، و یا ثبت یک ActivityLog.

اگر بخواهیم از StudentRepository بالا برای ایجاد دانش آموزان جدید پس از خرید آنها از فروشگاه کتاب مان استفاده کنیم چه؟ اگر DbContext خود را به مخزن تزریق کنید و دانش آموز جدید را ذخیره کنید .. اوه .. تمام تراکنش شما فلاش شده و از بین رفته!

حالا گزینه‌های شما اینها هستند: 1) از StudentRepository استفاده نکنید (از OrderRepository یا چیز دیگری استفاده کنید). و یا 2) فراخوانی ()SubmitChanges را حذف کنید و به باگ‌های متعددی اجازه ورود به کد تان را بدهید.

اگر تصمیم بگیرید که از StudentRepository استفاده نکنید، حالا کدهای تکراری (duplicate) خواهید داشت.

شاید بگویید که برای دستیابی به شناسه رکورد جدید نیازی به ()SubmitChanges نیست، چرا که خود EF این عملیات را در قالب یک تراکنش انجام می‌دهد!

دقیقا درست است، و نکته من نیز همین است. در ادامه به این قسمت باز خواهیم گشت.

متدهای Repositories قرار است اتمیک باشند

به هر حال تئوری اش که چنین است. چیزی که در Repositoryها داریم حتی اصلا Repository هم نیست. بلکه یک abstraction برای عملیات CRUD است که هیچ کاری مربوط به منطق تجاری اپلیکیشن را هم انجام نمی‌دهد. مخازن قرار است روی دستورات مشخصی تمرکز کنند (مثلا ثبت یک رکورد یا واکشی لیستی از اطلاعات)، اما این مثال‌ها چنین نیستند.

همانطور که گفته شده استفاده از چنین رویکردهایی به سرعت مشکل ساز می‌شوند و با رشد اپلیکیشن شما نیز مشکلات عدیده ای برایتان بوجود می‌آورند.

خوب، راه حل چیست؟

برای جلوگیری از این abstractionهای غیر منطقی دو راه وجود دارد. اولین راه استفاده از Command/Query Separation است که ممکن است در ابتدا کمی عجیب و بنظر برسند اما لازم نیست کاملا CQRS را دنبال کنید. تنها از سادگی انجام کاری که مورد نیاز است لذت ببرید، و نه بیشتر.

آبجکت‌های Command/Query

Jimmy Bogard مطلب خوبی در اینباره نوشته است و با تغییراتی جزئی برای بکارگیری Properties کدی مانند لیست زیر خواهیم داشت. مثلا برای مطالعه بیشتر درباره آبجکت‌های Command/Query به این لینک سری بزنید.

```
public class TransactOrderCommand {
  public Customer NewCustomer {get;set;}
  public Customer ExistingCustomer {get;set;}
  public List<Product> Cart {get;set;}
  //all the parameters we need, as properties...
  //...

  //our UnitOfWork
  StoreContext _context;
  public TransactOrderCommand(StoreContext context){
    //allow it to be injected - though that's only for testing
```

```
    _context = context;
  }

  public Order Execute(){
    //allow for mocking and passing in... otherwise new it up
    _context = _context ?? new StoreContext();

    //add products to a new order, assign the customer, etc
    //then...
    _context.SubmitChanges();

    return newOrder;
  }
}
```

همین کار را با یک آبجکت Query نیز می‌توانید انجام دهید. می‌توانید پست Jimmy را بیشتر مطالعه کنید، اما ایده اصلی این است که آبجکت‌های Query و Command برای دلیل مشخصی وجود دارند. می‌توانید آبجکت‌ها را در صورت نیاز تغییر دهید و یا mock کنید.

DataContext خود را در آغوش بگیرید ایده ای که در ادامه خواهید دید را شخصا بسیار می‌پسندم (که توسط Ayende معرفی شد). چیزهایی که به آنها نیاز دارید را در قالب یک فیلتر wrap کنید و یا از یک کلاس کنترلر پایه استفاده کنید (با این فرض که از اپلیکیشن‌های وب استفاده می‌کنید).

```
using System;
using System.Web.Mvc;

namespace Web.Controllers
{
  public class DataController : Controller
  {
    protected StoreContext _context;

    protected override void OnActionExecuting(ActionExecutingContext filterContext)
    {
      //make sure your DB context is globally accessible
      MyApp.StoreDB = new StoreDB();
    }

    protected override void OnActionExecuted(ActionExecutedContext filterContext)
    {
      MyApp.StoreDB.SubmitChanges();
    }
  }
}
```

این کار به شما اجازه می‌دهد که از DataContext خود در خلال یک درخواست واحد (request) استفاده کنید. تنها کاری که باید بکنید این است که از این کلاس پایه ارث بری کنید. این بدین معنا است که هر درخواست به اپلیکیشن شما یک UnitOfWork خواهد بود. که بسیار هم منطقی و قابل قبول است. در برخی موارد هم شاید این فرض درست یا کارآمد نباشد، که در این هنگام می‌توانید از آبجکت‌های Command/Query استفاده کنید.

ایده‌های بعدی: چه چیزی بدست آوردیم؟ چیزهای متعددی بدست آوردیم.

تراکنش‌های روشن و صریح : دقیقا می‌دانیم که DbContext ما از کجا آمده و در هر مرحله روی چه UnitOfWork ای کار می‌کنیم. این امر هم الان، و هم در آینده بسیار مفید خواهد بود

انتزاع کمتر == شفافیت بیشتر : ما Repositoryها را از دست دادیم، که دلیلی برای وجود داشتن نداشتند. به جز اینکه یک abstraction از abstraction دیگر باشند. رویکرد آبجکت‌های Command/Query تمیزتر است و دلیل وجود هرکدام و مسئولیت آنها نیز روشن‌تر است

شانس کمتر برای باگ ها : رویکردهای مبتنی بر Repository باعث می‌شوند که با تراکنش‌های ناموفق یا پاره ای (partially-executed) مواجه شویم که نهایتا به یکپارچگی و صحت داده‌ها صدمه می‌زند. لازم به ذکر نیست که خطایابی و رفع چنین مشکلاتی شدیدا زمان بر و دردسر ساز است

برای مطالعه بیشتر

ایجاد Repositories بر روی UnitOfWork
به الگوی Repository در لایه DAL خود نه بگویید!
پیاده سازی generic repository یک ضد الگو است
نگاهی به generic repositories
بدون معکوس سازی وابستگی‌ها، طراحی چند لایه شما ایراد دارد

نظرات خوانندگان

نویسنده: شهروز جعفری
تاریخ: ۱۳۹۳/۰۲/۰۸ ۱۹:۵۴

سلام آرمین جان ممنون از مطلبت
به نظرم جای یک بحثی خالی اونم تست پذیری کد.

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۲/۰۸ ۲۱:۱۳

ممنون.
با بیشتر مطالب شما موافقم ولی Repositoryها نیز دلیلی برای وجود دارند.
»الگوی Repository بسیار پروژه را تست پذیر می‌کند. به راحتی با استفاده از کتابخانه‌های Mock می‌توان بخش دسترسی به داده را تست کرد.
»اگر منظور شما از StoreContext ، کلاسی است که مستقیم از DBContext ارث برده است، در نتیجه امکان استفاده از دستوراتی نظیر Entry of T و Set of T یا مواردی مربوط به Change Tracking نیز به صورت مستقیم حتی در الگوی CQRS نیز وجود دارد.
چگونه می‌توانید دستوراتی این چنینی را Mock کنید؟ استفاده از کتابخانه‌های Mock نظیر Moq برای تست دستوراتی نظیر Entry Of T و SetCurrentValues و GetCurrentValues کمکی به شما نمی‌کند. (برای DbSet کتابخانه ای نظیر FakeDbSet وجود دارد ولی برای سایر دستورات خیر...)
»اگر از روش توصیه شده در این جا استفاده کنید باز برای Mock آبجکت IUnitOfWork به مشکل بر خواهید خورد. در این حالت برای تست لایه‌های دسترسی بهتر است از کتابخانه هایی نظیر Effort استفاده نمایید.

» در بخش **شناسه مشتری جدید را نیاز داشتم** یک راه حل را فراموش کردید و آن استفاده از GUID برای تعریف Id هر entity است در نتیجه دیگر نیازی به واکشی مجدد رکورد نخواهید داشت.

»بهتر است متد Save را نیز در Repository قرار ندهید. متد Save باید توسط UnitOfWork به اشتراک گذاشته شده فراخوانی شود.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۲/۰۸ ۲۱:۴۰

- How EF6 Enables Mocking DbSets more easily
- Testing with a mocking framework - EF6 onwards

+ شخصا اعتقادی به Unit tests درون حافظه‌ای، در مورد لایه دسترسی به داده‌ها ندارم . به قسمت « Limitations of EF in-memory test doubles » مراجعه کنید؛ توضیحات خوبی را ارائه داده‌است.
تست درون حافظه‌ی LINQ to Objects با تست واقعی LINQ to Entities که روی یک بانک اطلاعاتی واقعی اجرا می‌شود، الزاما نتایج یکسانی نخواهد داشت (به دلیل انواع قیود بانک اطلاعاتی، پشتیبانی از SQL خاص تولید شده تا بارگذاری اشیاء مرتبط و غیره) و نتایج مثبت آن به معنای درست کار کردن برنامه در دنیای واقعی نخواهد بود. در اینجا Integration tests بهتر جواب می‌دهند و نه Unit tests.

نویسنده: جلال
تاریخ: ۱۳۹۳/۰۲/۰۸ ۲۱:۵۷

خدا از دهنت بشنوفه. مدت هاست منم به همین نتیجه رسیدم تازه وقتی فهمیدم بدون اون بازم میشه قابلیت تست پذیری رو داشت. کافیه یه واسط از خود DbContext برنامه بسازی .
ولی الگوی Repository توی استفاده از کلاس‌های پایه ADO.NET مثل DbCommand و DbConnection کارایی خوبی داره.

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۲/۰۸ ۲۲:۲

ممنونم جناب نصیری. دلیل اشاره من به عدم تست پذیری قابل قبول در حالت استفاده مستقیم از Context به خاطر وجود دستوراتی نظیر Entry of T یا موارد مربوط به ChangeTracking است که با تست درون حافظه ای نتیجه مطلوب حاصل نمی‌شود، در نتیجه بهتر است از Effort برای تست لایه دسترسی استفاده شود که عملیات را در قالب یک دیتابیس SqlCE تست می‌کند و نسخه Effort.Ef6 آن نیز از Entity Framework 6 به خوبی پشتیبانی می‌کند.

نویسنده: Ara
تاریخ: ۱۳۹۳/۰۲/۰۸ ۲۳:۱۳

با توجه به متن قضاوتتون عجولانه است !

تو پروژه‌های Huge که توصیه خود مایکروسافت استفاده از Domain Driven و CQRS می‌باشد ، Repository یکی از اصول Domain Driven و Enterprise Application Pattern می‌باشد !

نویسنده: آرمین ضیاء
تاریخ: ۱۳۹۳/۰۲/۰۸ ۲۳:۵۹

با تشکر از همگی دوستان

شخصا نظرم به نظر جناب نصیری نزدیک‌تر است. جناب پاکدل هم به نکات خوبی اشاره فرمودند. اما صرفا توصیه‌های مایکروسافت و دیگران دال بر درستی یا کارآمدی یک رویکرد نمی‌تواند باشد. مطلب پست شده مبتنی بر چندین پست از توسعه دهندگان مطرح دنیای دات نت ترجمه و تالیف شده. مسلما هیچ راه حل نهایی (silver-bullet) ای وجود ندارد و توسعه ساختار پروژه بر اساس نیازها و تعاریف اپلیکیشن‌ها به پیش می‌رود. اما در کل می‌توان اینگونه نتیجه گیری کرد که استفاده از الگوی Repository در کنار فریم ورک‌های ORM مانند EF که مبتنی بر UnitOfWork کار می‌کنند ایده خوبی نیست. برای مطالعات بیشتر به چند لینک نمونه زیر مراجعه شود.

^ ، ^ ، ^ ، ^ ، ^ ، ^

نویسنده: محسن موسوی
تاریخ: ۱۳۹۳/۰۲/۰۹ ۱:۲

در پروژه http://nopcommerce.codeplex.com استفاده از Repository جهت اجبار به رویکرد Command/Query بوده است.(البته اینطور برداشت میشود) جهت مطالعه
http://nopcommerce.codeplex.com/SourceControl/latest#src/Libraries/Nop.Data/EfRepository.cs
و همینطور جهت تست پذیری پروژه، راه حل‌های اشاره شده را پیاده سازی کرده.

نویسنده: محسن موسوی
تاریخ: ۱۳۹۳/۰۲/۰۹ ۱:۱۱

آقای پاکدل لطفا راجع به این جمله بیشتر توضیح بدید:

»بهتر است متد Save را نیز در Repository قرار ندهید. متد Save باید توسط UnitOfWork به اشتراک گذاشته شده فراخوانی شود. در پروژه http://nopcommerce.codeplex.com/SourceControl/latest اینگونه عمل شده
http://nopcommerce.codeplex.com/SourceControl/latest#src/Libraries/Nop.Data/EfRepository.cs
در نهایت این لایه سرویس است که باید اطمینان از انجام عملیات درخواستی و یا عدم انجام آنرا بدهد و برای عملیات‌های

پیچیده‌تر نیز بایستی سیاست خودرا بسط دهد.منظور انجام عملیات Save و ادامه عملیات میباشد.
لایه UI وظیفه فراهم آوری اطلاعات را دارد و مابقی مسائل در لایه سرویس پوشش داده میشوند.
الزام این کار هم به وظیفه این لایه برمیگردد که یا این کار را میتوانم انجام دهم و یا خیر.
پیاده سازی ارائه شده نقضی بر جمله‌ی نقل قول شده میباشد؟

نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۲/۰۹ ۱۰:۱

به طور کلی هدف اصلی از الگوی واحد کار یا UnitOfWork به اشتراگ گذاشتن یک Context بین همه نمونه‌های ساخته شده از Repository یا سرویس‌های برنامه است. فرض کنید در یک کنترلر شما از دو یا سه نمونه از سرویس‌ها یا Repositoryها وهله سازی کرده اید. اگر قرار باشد برای اعمال تغییرات، مجبور به فراخوانی متد Save هر Repository باشیم چرا اصلا الگوی واحد کار را به کار بردیم؟ فراخوانی SaveChanges الگوی واحد کار معادل است با فراخوانی متدهای Save تمام Repositoryهای وهله سازی شده در طی یک درخواست.

نویسنده: آرایه
تاریخ: ۱۳۹۳/۰۲/۰۹ ۱۰:۲۰

دلایل منطقی هستند و کد ارائه شده در مثال‌ها واقعاً مشکل دارد. بعضی آثار را شاید بتوان کاهش داد. مثلاً برای رفع Repository APIهای بی‌انتها شاید استفاده از متدی که IQueryable برگرداند و بعد ادامه دادن کوئری در خروجی آن متد کمک کند.
یک پروژه برای پیاده سازی Generic از Repository و Unit Of Work اینجا هست که مشکلات کمتری دارد.

نویسنده: محسن موسوی
تاریخ: ۱۳۹۳/۰۲/۰۹ ۱۰:۲۸

صد در صد درست.ولی فکر میکنم این مسئله باید در لایه سرویس حل بشه. در یک Application انتظار چندین و چند عملیات در طی یک Request میره. برای نمونه میگم:
- در یک کنترلر قراره یک مشتری تعریف بشه.از طرفی هم لاگ گیری‌های عمومی سیستم نیز باید انجام باشه که اصولا در بعضی از عملیات‌ها مستقل از همدیگه باید باشند.پس باید چند بار SaveChanges فراخوانی بشه.
عملیات لاگ گیری سیستم حتما باید انجام بشه ولی عملیات تعریف یک مشتری میتونه دارای خطایی باشه.(استقلال بعضی از عملیات‌های سیستم در UOW)
-عملیات هایی که در طی یک Action در کنترلر انجام میشه:بایستی تمام اینها به لایه‌ی سرویس منتقل بشه و اونجا در طی یک SaveChange عملیات مورد نظر نتیجه بده.(رویکرد Command/Query)
- الگوی واحد کار هدف‌های بیشتری داره.
-مسئولیت هر متد در لایه سرویس مشخصه و نتیجه بازگشتی از لایه سرویس عملا بایستی دلالت بر نتیجه‌ی عملیات رو داشته باشه.نه اینکه در یک متد در لایه سرویس عملیات درج رو انجام بده و بعد در UI عملیات خطا بده.
-جدا سازی منطق لایه‌ها در این کار مشخص نیست.(تا حدی)
*مدیریت پیچیده وظایف در لایه سرویس به درستی انجام بشه SaveChangesها با کمترین سربار و بهترین کارایی انجام میشه.
البته فکر میکنم در پروژه اشاره شده نیز به همین مسئله دلالت داره.
http://nopcommerce.codeplex.com/SourceControl/latest#src/Libraries/Nop.Data/EfRepository.cs
و اینکه در بعضی از مسائل نیز باید تغییراتی صورت بگیره .مانند عملیات‌های گروهی.
و در نهایت صحبت آقای ضیا دلالت بر تفکرات متفاوت درستره.

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۰۲/۰۹ ۱۱:۳

This is a leaky abstraction

نویسنده: Ara
تاریخ: ۱۳۹۳/۰۲/۱۳ ۰:۵۳

تو مبحث DDD دلیل اصلی که Repository وارد داستان شده Persistence Ignorance می‌باشد ،
همونطور که میدونید ، این قضیه می‌گه که شما تو Domain نبایید بگید EF این طوری Select می‌زنه Nhibernate یک نوع دیگر ، NoSql یک نحو دیگر (NoSql ها هم بخاطر اینکه میتونند براحتی یک Aggregate رو ذخیره کنند میتونند ابزار خوبی برای DDD باشند!)
چون Domain نباید به تکنولوژی وابسته باشد ! نباید رفرنسی به دیتا اکسس یا EF و یا ... داشته باشد فقط یک سری Interface تعریف می‌کند ، که یکی که بعدا به نام لایه دیتا اکسس می‌باشد باید این اینترفیس رو Implement کتد !
در مورد CQRS هم چون معمولا Application Layer بر روی Rest هاست می‌شوند پس هر Request فقط شامل یک Command می‌باشد که Unit Of work رو هم فقط روی همان Command ایجاد می‌کنند
جالبه براتون بگم که در Domain Driven Design اصل بر این هست که شما در هر ترانزاکشن فقط یک Aggregate رو باید ذخیره کنید و تغییر در Aggregate های دیگه بوسیله Event Source ها Publish می‌شه

و از توصیه‌های اولیه DDD اینه که برای پروژه‌های Complex و Huge استفاده بشه ،پس قطعا برای یک پروژه که از این متد استفاده نمی‌شه و یا در ابعاد کوچکتر می‌باشد کاملا حرف شما درست باشد و از پیچیده شدن برنامه جلوگیری می‌کند

نویسنده: Ara
تاریخ: ۱۳۹۳/۰۲/۱۳ ۱:۳

پیاده سازی خوبیه
البته زمانی که از DDD استفاده می‌شه استفاده از IQueryable در IRepository به عنوان نشت اطلاعات خوانده می‌شود و تاکیدا نباید استفاده شود !

این Repository های Generic میتوانند داخل یک کلاس که IRepository را Implement کرده استفاده شوند و یا به عنوان کلاس Base ان باشند

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۰۲/۱۳ ۱:۱۸

این generic repository الان از امکانات async در EF 6 داره استفاده می‌کنه. برای مثال NH چنین توانمندی async ایی رو در حال حاضر نداره. آیا در این حالت Persistence Ignorance تامین شده؟ یعنی راحت میشه زیر ساخت این مخزن رو عوض کرد و سوئیچ کرد به یک ORM دیگه؟ و اگر نخواهیم از async استفاده کنیم، خوب یک ORM داریم که توانمندی‌های جدیدش رو باید ازش صرفنظر کرد. خروجی IQueryable آن که جای خودش. ORMهای مختلف متدهای الحاقی خاص خودشون رو دارند و پیاده سازی یکسانی از LINQ رو ندارند. یعنی اگر با EF کار کردید و متد Include آن توسط این generic repository بخاطر خروجی IQueryable در دسترس بود، معادلی در سایر ORMها نداره (متدهای الحاقی اون‌ها فرق می‌کنه). یا مثلا NH سطح دوم کش رو با متد الحاقی Cacheable پیاده سازی کرده. فرض کنید این رو در generic repository قرار دادیم (یک روکش روی این متد تا به ظاهر مستقیما در دسترس نباشه). خوب، الان فلان ORM دیگه که متد Cacheable رو نداره چکار باید باهاش کرد؟ این برنامه و سیستم به این سادگی‌ها قابل تبدیل به یک ORM دیگه نیست. رسیدن به Persistence Ignorance در دنیای واقعی کار ساده‌ای نیست مگر اینکه از توانمندی‌های خوب ORM انتخاب شده صرفنظر کنیم و به قولی دست و پاشو ببریم تا قد بقیه بشه.
گذشته از این‌ها بحث مدل سازی هم هست. نگاشت‌های کلاس‌ها و خواص اون‌ها به جداول بانک اطلاعاتی در ORMهای مختلف 100 درصد با هم متفاوت هست. حداقل EF و NH روش‌های خاص خودشون رو دارند که انطباقی با هم ندارن. یعنی این Persistence Ignorance محدود نیست به روکش کشیدن روی insert/update/delete. اینجا صحبت از یک سیستم هست که اجزای هماهنگ زیادی داره که باید درنظر گرفته بشه؛ از نگاشت‌ها تا اعتبارسنجی‌های خاص تا قابلیت‌های ویژه و صددرصد اختصاصی. به این می‌گن تا خرخره فرو رفتن!

نویسنده: Ara
تاریخ: ۱۳۹۳/۰۲/۱۳ ۷:۴۱

در مورد async راست می‌گید ! باید ببینم راهی داره یا نه ، در ضمن در لایه دیتا اکسس هر جور که می‌خواهید می‌تونید include و غیره بزنید مشکلی وجود نداره ، چون رو اینتفریس از شما می‌خواهند که یک entity چه چیزهایی همراهش باشه یا نباشه
خوب اگه به cqrs نگاه کنید در سمت Command شما قسمت اصلی و insert , Update , delete و Get رو دارید و برخی مواقع getall که خیلی کمه ولی سمت query کاملا دستتون بازه هر جور که کار کنید کلا پشت query سرویس هر جور که راحتی با هر چی که راحتی کار کن !

نویسنده: Ara
تاریخ: ۱۳۹۳/۰۲/۱۳ ۱۷:۱۷

مثل اینکه async کردن متدهای Repository زیاد پیچیده نمی‌باشد !

پس می‌شه query های NH رو هم Async کرد ، پس روی IRepository می‌تونیم هم متد Async هم متد Sync رو با هم داشته باشیم

نویسنده: علیرضا
تاریخ: ۱۳۹۳/۰۲/۲۶ ۱۲:۵۷

-1 من دقیقا متوجه نشدم منظور شما از decoupling اول مقاله چیه؟ منظورتون تفکیک Domain از DAL هست؟ اگر اینطوره چه ربطی به UoW و انواع پیاده سازی اون داره؟
اگر منظورشما انفکاک بین EFContext و Repository هست، توجه شما رو به این نکته جلب میکنم که StudentRepository که در اول مقاله آورده شده در حقیقت یک پیاده سازی برپایه EF هست به عبارتی EFStudentRepository اسم مناسب‌تری میتونه باشه. بنابراین تزریق Context با هیچ اصلی مغایر نیست. چرا که این Repository یک پیاده سازی خاص از IStudentRepository است.

-2 وجود متد Save در Repository؟ نه تنها قابل قبول نیست که اصلا اگر قرار باشه هر Repository مستقلا Save رو صدا بزنه که مفهوم Transaction از بین میره یا حداقل سخت میشه بهش رسید.

-3 با شما موافقم که Generic Repository ایده خوبی نیست. البته فقط تا اینجا موافقم که این الگو برای Expose کردن Interface یک Repository مناسب نیست. چه بسا Repository هایی که فقط SELECT میکنند. ولی اگر پیاده سازی خاصی از یک Repository مد نظر دارید (مثلا پیاده سازی برپایه EF یا NHibernate) اونوقت دقیقا چیزی که به کمک شما میاد همین Generic Repository برای جلوگیری از کدهای تکراریه.

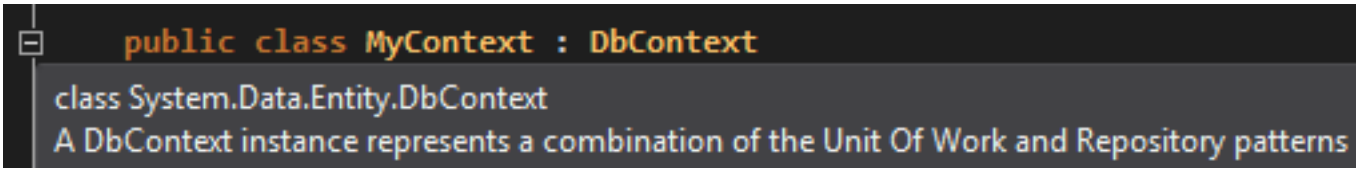
-4 اصولا Repository برای اینکه منطق برنامه (یا به قول شما منطق تجاری) رو پیاده سازی کنه نیست. در حقیقت لایه ای که استفاده کننده مستقیم از Repository است میداند که چه موقع به چه Repository فرمانهای CRUD بده تا منطق برنامه پیاده سازی بشه.

-5 در واقع استفاده از امکانات هر ORM تا حد بینهایتی امکان پذیره به شرطی که ORM و توانمندیهاشو در همون لایه DAL محصور کنید مثلا IQueriable و Cachable و گرنه Leaky Abstraction به طور خزنده و ساکتی کل برنامه رو مثل سرطان در خودش میکشه.

نهایتا اینکه نمیشه یک پیاده سازی مشکل دار از مفهوم Repository + UoW رو بدون درنظر گرفتن مفاهیم مهمی مثل Service Layer و Domain Model نقد کرد و بعدا نتیجه گرفت که این الگوها صحیح نیستند. ضمن اینکه این موضوع بسته به تجربه و نظر هر برنامه نویس و معماری میتونه پیاده سازی خاص خودشو داشته باشه که من شخصا هنوز موارد جالب و جدیدی که یک برنامه نویس باهوش برداشت کرده رو میبینم و نتیجه میگیرم که مفهوم Repository + UoW در بین ماها هنوز به یک تعریف جهانشمول نرسیده.

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۲/۲۶ ۱۳:۱۸

- مباحث الگوی مخزن، در حالت کلی درست هستند؛ یک بحث انتزاعی، بدون درنظر گرفتن فناوری پیاده سازی کننده‌ی آن.
- در مورد EF به خصوص (در این مطلب)، DbSet و DbContext آن پیاده سازی کننده‌ی الگوهای Repository و Uow هستند (و منکر *آن نیستند*). به همین جهت عنوان می‌کنند که روی Repository آن، دوباره یک Repository درست نکنید. در بحث هم اشاره به «یک abstraction از abstraction دیگر» همین مطلب است.



تصویری است از قرار دادن کرسر ماوس بر روی DbContext در VS.NET که به *صراحت* در آن از پیاده سازی الگوی مخزن یاد شده

اینترفیس IDbSet معروف در EF دقیقا یک abstraction است و بیانگر ساختار الگوی مخزن. کاملا هم قابلیت mocking دارد؛ از نگارش 6 به بعد EF البته (^ و ^ و ^).

- راه حل‌های ارائه شده به دلیل اینکه Uow را تزریق نمی‌کنند مشکل دارند. اساسا هرگونه لایه بندی بدون تزریق وابستگی‌ها مشکل دارد؛ نمی‌شود یک وهله از یک شیء را بین چندین کلاس درگیر به اشتراک گذاشت (مباحث مدیریت طول عمر در IoC Containerها). مثلا در راه حل آخر ارائه شده فقط آغاز و پایان اجرای یک متد از یک کنترلر مشخص تحت نظر هستند. واقعیت این است که تا اجرای یک اکشن متد به پایان برسد، در طول یک درخواست، پردازش referrer رسیده هم در کلاسی دیگر به موازت آن باید انجام شود (در یک HTTP Module مجزا) و امثال آن. در این حالت چون یک وهله از Uow به اشتراک گذاشته نشده، مدام باید وهله سازی شود؛ بجای اینکه از آن تا پایان درخواست، استفاده‌ی مجدد شود. برای حل آن، در متن ذکر شده مطمئن شوید که «globally accessible» است. این مورد و راه حل‌های استاتیک (مانند نحوه‌ی فراخوانی MyApp آن) و singleton در برنامه‌های وب تا حد ممکن باید پرهیز شود. چون به معنای به اشتراک گذاری آن در بین تمام کاربران سایت. این مورد تخریب اطلاعات را به همراه خواهد داشت. چون DbContext جاری در حال استفاده توسط کاربر الف است و در همان زمان کاربر ب هم چون دسترسی عمومی به آن تعریف شده، مشغول به استفاده از آن خواهد شد. در این بین عملا تراکنش تعریف شده بی‌معنا است چون اطلاعات آن خارج از حدود متدهای مدنظر توسط سایر کاربران تغییر کرده‌اند.
همچنین به دلیل عدم تزریق وابستگی‌ها، پیاده سازی‌های آن تعویض پذیر نیستند و قابلیت آزمایش واحد پایینی خواهند داشت. برای مثال در بحث mocking که مطرح شد، می‌توانید بگویید بجای این متد خاص از کلاس اصلی، نمونه‌ی آزمایشی من را استفاده کن.

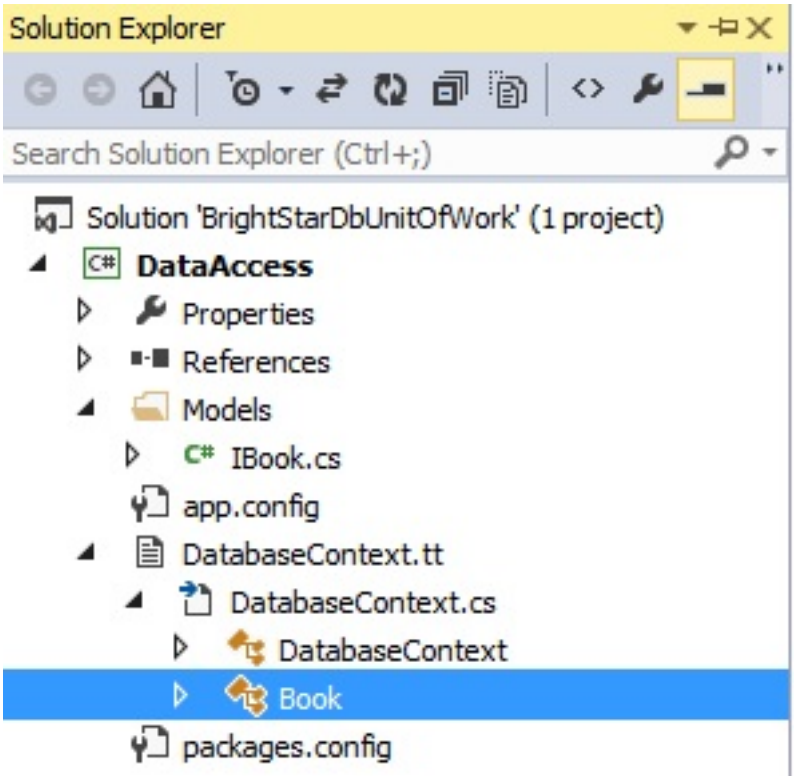
عنوان: **پیاده سازی UnitOfWork برای BrightStarDb**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۳/۰۲/۲۶ ۲۰:۰
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, NoSQL, UnitOfWork, BrightstarDB

در این پست با BrightStarDb و مفاهیم اولیه آن آشنا شدید. همان طور که پیش‌تر ذکر شد BrightStarDb از تراکنش‌ها جهت ذخیره اطلاعات پشتیبانی می‌کند. قصد داریم روش شرح داده شده در اینجا را بر روی BrightStarDb فعال کنیم. ابتدا بهتر است با روش ساخت مدل در B*Db آشنا شویم.
*یکی از پیش نیازهای این پست مطالعه این دو مطلب (^) و (^) می‌باشد.
فرض می‌کنیم در دیتابیس مورد نظر یک Store به همراه یک جدول به صورت زیر داریم:

```
[Entity]
    public interface IBook
    {
        [Identifier]
        string Id { get; }

        string Title { get; set; }

        string Isbn { get; set; }
    }
```

بر روی پروژه مورد نظر کلیک راست کرده و گزینه Add new Item را انتخاب نمایید. از برگه Data گزینه BrightStar Entity Context را انتخاب کنید

بعد از انتخاب گزینه بالا یک فایل با پسوند tt به پروژه اضافه خواهد شد که وظیفه آن جستجو در اسمبلی مورد نظر و پیدا کردن تمام اینترفیس هایی که دارای EntityAttribute هستند و همچنین ایجاد کلاس‌های متناظر جهت پیاده سازی اینترفیس‌های بالا است. در نتیجه ساختار پروژه تا این جا به صورت زیر خواهد شد.



واضح است که فایلی به نام Book به عنوان پیاده سازی مدل IBook به عنوان زیر مجموعه فایل DatabaseContext.tt به پروژه اضافه شده است.

تا اینجا برای استفاده از Context مورد نظر باید به صورت زیر عمل نمود:

```
DatabaseContext context = new DatabaseContext();
  context.Books.Add(new Book());
```

Context پیش فرض ساخته شده توسط B*Db از Generic DbSetهای معادل EF پشتیبانی نمی‌کند و از طرفی IUnitOfWork مورد نظر به صورت زیر است

```
public interface IUnitOfWork
    {
        BrightstarEntitySet<T> Set<T>() where TEntity : class;
        void DeleteObject(object obj);
         void SaveChanges();
    }
```

در اینجا فقط به جای IDbSet از BrightStarDbSet استفاده شده است. همان طور که در این مقاله توضیح داده شده است، برای پیاده سازی مفهوم UnitOfWork؛ نیاز است تا کلاس DatabaseContext که نماینده BrightStarDbContext پروژه است، از اینترفیس IUnitOfWork طراحی شده ارث بری کند. جهت انجام این مهم و همچنین جهت اضافه کردن قابلیت ایجاد Generic DbSetها نیز باید کمی در فایل Template Generator تغییر ایجاد نماییم. این تغییرات را قبلا در طی یک پروژه ایجاد کرده‌ام و شما می‌توانید آن را از اینجا دریافت کنید. بعد از دانلود کافیست فایل DatabaseContext.tt مورد نظر را در پروژه خود کپی کرده و گزینه Run Custom Tools را فراخوانی نمایید.

نکته: برای حذف یک آبجکت از Store، باید از متد DeleteObject تعبیه شده در Context استفاده نماییم. در نتیجه متد مورد نظر نیز در اینترفیس بالا در نظر گرفته شده است.

استفاده از IOC Container جهت رجیستر کردن IUnitOfWrok

در این قدم باید IUnitOfWork را در یک IOC container رجیستر کرده تا در جای مناسب عملیات وهله سازی از آن میسر باشد.
من در اینجا از Castle Windsor Container استفاده کردم. کلاس زیر این کار را برای ما انجام خواهد داد:

```
public class DependencyResolver
    {
        public static void Resolve(IWindsorContainer container)
        {
            var context = new
DatabaseContext("type=embedded;storesdirectory=c:\brightstar;storename=test ");
            container.Register(Component.For<IUnitOfWork>().Instance(context).LifestyleTransient());
        }
    }
```

حال کافیست در کلاس‌های سرویس برنامه UnitOfWork رجیستر شده را به سازنده آن‌ها تزریق نماییم.

```
public class BookService
    {
        public BookService(IUnitOfWork unitOfWork)
        {
            UnitOfWork = unitOfWork;
        }

        public IUnitOfWork UnitOfWork
        {
            get;
            private set;
        }

        public IList<IBook> GetAll()
        {
            return UnitOfWork.Set<IBook>().ToList();
        }

        public void Add()
        {
            UnitOfWork.Set<IBook>().Add(new Book());
        }

        public void Remove(IBook entity)
        {
            UnitOfWork.DeleteObject(entity);
        }
    }
```

سایر موارد دقیقا معادل مدل EF آن است.
نکته: در حال حاضر امکان جداسازی مدل‌های برنامه (تعاریف اینترفیس) در قالب یک پروژه دیگر(نظیر مدل CodeFirst در EF) در B*Db امکان پذیر نیست.
نکته : برای اضافه کردن آیتم جدید به Store نیاز به وهله سازی از اینترفیس IBook داریم. کلاس Book ساخته شده توسط DatabaseContext.tt در عملیات Insert و update کاربرد خواهد داشت.

عنوان: **پشتیبانی توکار از ایجاد کلاس‌های Singleton از دات نت 4 به بعد**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۸/۰۲ ۱۲:۴۵
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, Threading

روش‌های زیادی برای ایجاد یک وهله‌ی Singleton وجود دارند. وهله‌ای که در طول عمر یک برنامه، تنها یکبار ایجاد شده و حفظ می‌شود. برای مثال شاید متداول‌ترین حالت آن که در بسیاری از کدها دیده می‌شود، تعریف یک متغیر استاتیک در کلاس، غیرعمومی تعریف کردن سازنده‌ی کلاس و وهله سازی این فیلد استاتیک در صورت نال بودن آن است:

```
public class WrongSingleton
    {
        static WrongSingleton _instance;

        WrongSingleton()
        {
        }

        public static WrongSingleton Instance
        {
            get { return _instance ?? (_instance = new WrongSingleton()); }
        }
    }
```

هرچند این روش کار می‌کند اما thread-safe نیست. به این معنا که ممکن است دو ترد در آن واحد به بررسی قسمت ?? _instance بپردازند و چون هنوز نال است، دوبار وهله سازی کلاس، با فراخوانی new WrongSingleton صورت خواهد گرفت و هر ترد در آن لحظه به وهله‌ی متفاوتی دسترسی خواهد داشت.
راه حل‌های زیادی برای رفع این مشکل با اعمال مباحث قفل گذاری تا نکات ریز مربوط به کامپایلر وجود دارند که لیست آن‌ها را در اینجا می‌توانید مطالعه کنید.

از دات نت 4 به بعد با معرفی الگوی Lazy ، امکان پیاده سازی lazy thread safe singletons به صورت توکار در دسترس می‌باشد. نمونه‌ای از آن در کدهای IoC Container معروفی به نام StructureMap بکار رفته‌است :

```
public class Container
    {
        // ...
    }

    public static class ObjectFactory
    {
        private static readonly Lazy<Container> _containerBuilder =
            new Lazy<Container>(defaultContainer, LazyThreadSafetyMode.ExecutionAndPublication);

        public static Container Container
        {
            get { return _containerBuilder.Value; }
        }

        private static Container defaultContainer()
        {
            return new Container();
        }
    }
```

در اینجا کلاس ObjectFactory یک وهله از کلاس Container را در اختیار مصرف کننده قرار می‌دهد؛ با این شرایط:
- چون این وهله توسط کلاس Lazy محصور شده‌است، صرفا در اولین بار دسترسی به آن، نمونه سازی خواهد شد. این مورد سبب کاهش مصرف حافظه‌ی برنامه و همچنین بالا رفتن سرعت برپایی اولیه‌ی آن می‌شود. بسیاری از اشیایی که در یک برنامه تعریف می‌شوند، شاید الزاما جهت ارائه راه حلی برای مساله‌ای خاص، مورد استفاده قرار نگیرند. تعریف آن‌ها به صورت Lazy، سربار نمونه سازی الزامی آن‌ها را حذف خواهد کرد و آن‌را به اولین بار استفاده از شیء مورد نظر، به تعویق خواهد انداخت.
- با استفاده از LazyThreadSafetyMode.ExecutionAndPublication، چندین ترد می‌توانند به خاصیت Container دسترسی پیدا کنند، اما تنها یکی از آن‌ها موفق به وهله سازی این کلاس خواهد شد. البته حالت ExecutionAndPublication، حالت پیش فرض

است و الزاما نیازی به ذکر آن نیست.

اینبار بازنویسی کلاس ابتدای بحث با توجه به نکات ذکر شده به صورت زیر خواهد بود:

```
public sealed class LazySingleton
    {
        private static readonly Lazy<LazySingleton> _instance =
            new Lazy<LazySingleton>(() => new LazySingleton(),
LazyThreadSafetyMode.ExecutionAndPublication);

        private LazySingleton()
        {
        }

        public static LazySingleton Instance
        {
            get { return _instance.Value; }
        }
    }
```

- در آن سازنده‌ی کلاس، خصوصی تعریف شده‌است.
- تنها وهله‌ی در دسترس کلاس، به صورت استاتیک و نمونه سازی کلاس، توسط کلاس Lazy با پارامتر LazyThreadSafetyMode.ExecutionAndPublication انجام می‌شود.
- علت استفاده از lambda در سازنده‌ی کلاس Lazy، امکان دسترسی به اعضای private کلاس، از طریق یک خاصیت static است.

عنوان: **ایجاد یک Abstract Factory با استفاده از جنریک‌ها**
نویسنده: احمد نواصری
تاریخ: ۱۳۹۳/۰۸/۱۱ ۱۵:۲۵
آدرس: www.dotnettips.info
گروه‌ها: C#, Design patterns

همان طور که میدانید از الگوی Factory به عنوان روشی برای کاهش وابستگی اجزای یک سیستم استفاده میشود. در این مقاله میخواهیم با استفاده از جنریک‌ها، الگوی Abstract Factory را پیاده سازی کنیم.
1) ایجاد یک کلاس به نام AbstractFactory و یک متد جنریک به نام CreateObject

```
public class AbstractFactory
    {
        public static T CreateObject<T>() where T : class , new()
        {
            return new T();
        }
    }
```

2) ساخت کلاسهای مورد نظر

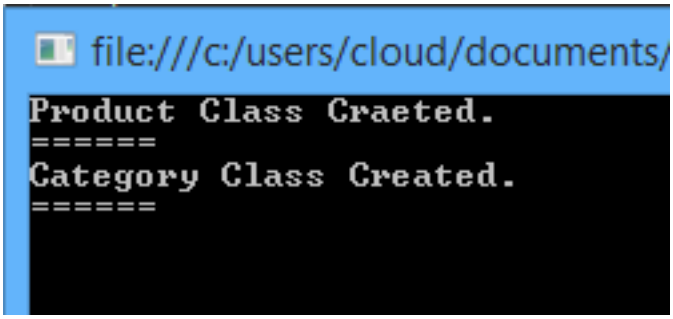
```
public class Product
    {
        public void DisplayInfo()
        {
            Console.WriteLine("Product Class Craeted. ");
        }
    }
```

```
public class Category
    {
        public void DisplayInfo()
        {
            Console.WriteLine("Category Class Created.");
        }
    }
```

3) حال در یک برنامه‌ی کنسول ویندوز، از کلاس AbstractFactory به شکل زیر استفاده میکنیم

```
static void Main(string[] args)
        {
            var p = AbstractFactory.CreateObject< Product>();
            p.DisplayInfo();
            Console.WriteLine("======");

            var c = AbstractFactory.CreateObject<Category>();
            c.DisplayInfo();
            Console.WriteLine("======");

            Console.ReadKey();
        }
```

خروجی کد بالا


file:///c:/users/cloud/documents/
Product Class Craeted.
======
Category Class Created.
======

نظرات خوانندگان

نویسنده: سعید
تاریخ: ۱۳۹۳/۰۸/۱۲ ۲۱:۱۶

تو این حالت میشه به صورت مستقیم هم از کلاس Product، آبجکت تعریف کرد. پس در این حالت کاهش وابستگی دقیقا به چه صورتی هست؟ چه فرقی هست بین AbstractFactory.CreateObject<Product>() و new Product ()؟

نویسنده: محسن خان
تاریخ: ۱۳۹۳/۰۸/۱۲ ۲۱:۳۵

مطلب فوق کمی خلاصه شده هست. یک مثال عملی اون رو برای کاهش وابستگی‌ها، می‌تونید اینجا مطالعه کنید: استفاده از Factories برای حذف Service locators در برنامه‌های WinForms

نویسنده: احمد نواصری
تاریخ: ۱۳۹۳/۰۸/۲۲ ۱۰:۲۸

درسته . اما در این حالت دیگر کلاینت شما مستقیما به کار ساخت Object را انجام نمیدهد و کار را به کلاس دیگری واگذار کرده.

عنوان: **نحوه ایجاد الگوی Singleton به صورت جنریک**
نویسنده: احمد نواصری
تاریخ: ۱۳۹۳/۰۸/۲۲ ۱۳:۰
آدرس: www.dotnettips.info
گروه‌ها: C#, Design patterns

در برخی از مواقع، ایجاد یک وهله از یک کلاس کاری هزینه بر می‌باشد. بنابراین نیاز است تا فقط یک وهله از آن کلاس را ایجاد و تا آخر اجرای برنامه از آن استفاده کرد. این راه حل در قالب یک الگوی طراحی به نام Singleton معرفی شده است. حال می‌خواهیم با استفاده از امکانات جنریک، کلاسی را طراحی کنیم تا عملیات ساخت وهله‌ها را انجام دهد.
نکاتی که در طراحی یک الگوی Singleton باید مد نظر داشت این است که:
دسترسی سازنده کلاس Singleton را از نوع Private تعیین کنیم.
یک فیلد استاتیک از نوع کلاس Singleton تعریف کنیم.
یک خاصیت از نوع استاتیک فقط خواندنی (یعنی فقط get داشته باشد) تعریف کرده تا فیلد استاتیک را مقداردهی و Return کند. به جای پروپرتی میتوان از یک متد استاتیک نیز استفاده کرد.

```
public class SingletonClassCreator<T> where T:class , new()
    {
        private static T _singletoneInstance;
        private static readonly object Lock = new object();

        public static T SingletoneInstance
        {
            get
            {
                lock (Lock)
                {
                    if (_singletoneInstance == null)
                    {
                        _singletoneInstance = new T();
                    }
                }
                return _singletoneInstance;
            }
        }

        private SingletonClassCreator()
        {
        }
    }
```

برای ایجاد حالت Tread-Safe در برنامه هایی که امکان دسترسی همزمان به یک شیء (مثلا در برنامه‌های وب) وجود دارد، از یک بلاک Lock استفاده شده است تا در هر لحظه فقی یک نخ قادر به ایجاد Singleton شود.
حال برای ایجاد وهله‌های Singleton از کلاسهای مورد نظر به صورت زیر عمل میکنیم

```
public class FirstSingleton
    {
        public int Square(int input)
        {
            return input*input;
        }
    }
```

```
static void Main(string[] args)
        {
            var firstSingletone = SingletonClassCreator<FirstSingleton>.SingletoneInstance ;
            Console.WriteLine(firstSingletone.Square(12));
            Console.ReadKey();
        }
```

در خط اول، با تعریف یک متغیر و قرار دادن وهله استاتیک که بوسیله پروپرتی استاتیک SingletoneInstance برگشت داده میشود، یک شی Singleton از کلاس FirstSingleton را ایجاد میکنیم.

نظرات خوانندگان

نویسنده: مصطفی
تاریخ: ۱۳۹۳/۰۸/۲۷ ۹:۳۳

یه روش بهتر برای استفاده در حالت Thread Safe که به نظرم بهینه‌تر هستش در زمان اجرا, بهینه سازی کد به این شکل هستش

```
if (instance == null)
         {
            lock (Lock)
            {
               if (instance == null)
                  instance = new Singleton();
            }
         }
```

با این حالت در صورتی که شی قبلا ایجاد شده باشه هیچ کدوم از Threadها رو بلوک نمیکنه.

نویسنده: مصطفی
تاریخ: ۱۳۹۳/۰۸/۲۷ ۹:۳۴

همچنین میشد این کلاس رو با این لینک تلفیق کرد که یه خروجی جالبتری به وجود بیاد.
با تشکر از زحمت شما