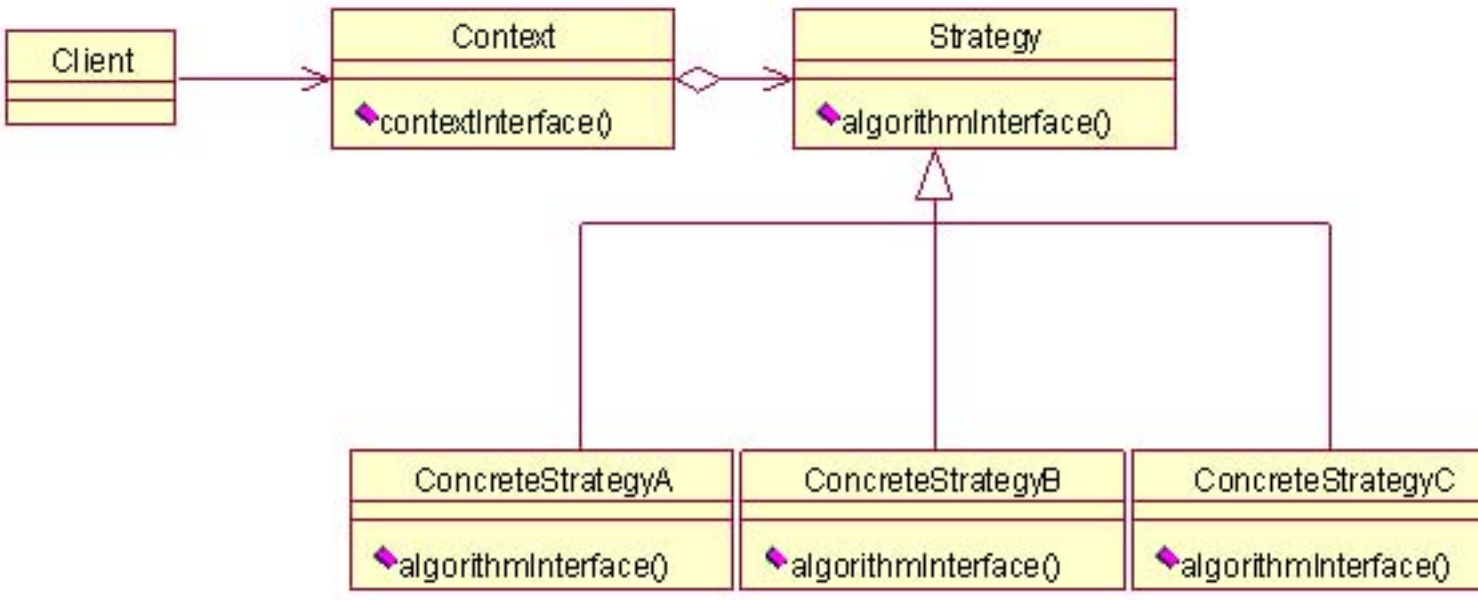
عنوان: **الگوی استراتژی - Strategy Pattern**
نویسنده: مجتبی شاطری
تاریخ: ۱۳۹۲/۰۶/۲۰ ۰:۰
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns, Dependency Injection, SOLID Principals, Inheritance Strategies, SOLID, Dependency Inversion

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: سيد ايوب كوكبي
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: **اصول طراحی شی گرا SOLID - #بخش پنجم اصل DIP**
نویسنده: ناصر طاهری
تاریخ: ۱۳۹۲/۰۷/۰۶ ۲۰:۵۰
آدرس: www.dotnettips.info
گروه‌ها: OOP, SOLID Principals, Dependency Inversion, Object Oriented Design

بخش‌های پیشین : اصول طراحی شی گرا SOLID - #بخش اول اصل SRP اصول طراحی شی گرا SOLID - #بخش دوم اصل OCP
اصول طراحی شی گرا SOLID - #بخش سوم اصل LSP
اصول طراحی شی گرا SOLID - #بخش چهارم اصل ISP

اصل 5) D – DIP– Dependency Inversion principle

مقایسه با دنیای واقعی:

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

به مثال زیر توجه کنید :

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

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

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

در طراحی ساخت یافته، ماژولهای سطح بالا به ماژولهای سطح پایین وابسته بودند. این مسئله دو مشکل ایجاد می‌کرد:
1- ماژول‌های سطح بالا (سیاست گذار) به ماژول‌های سطح پایین (مجری) وابسته هستند. در نتیجه هر تغییری در ماژول‌های سطح پایین ممکن است باعث اشکال در ماژول‌های سطح بالا گردد.
2- استفاده مجدد از ماژول‌های سطح بالا در جاهای دیگر مشکل است، زیرا وابستگی مستقیم به ماژول‌های سطح پایین دارند. **راه حل با توجه به اصل DIP :**

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

public class FileLogger:ILogger
```

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

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

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

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

اطلاعات بیشتر در دوره آموزشی " بررسی مفاهیم معکوس سازی وابستگی‌ها و ابزارهای مرتبط با آن ".

نظرات خوانندگان

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

خیلی مطلب خوبی بود! لذت بردیم متشکرم :)

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

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

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

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

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;
```

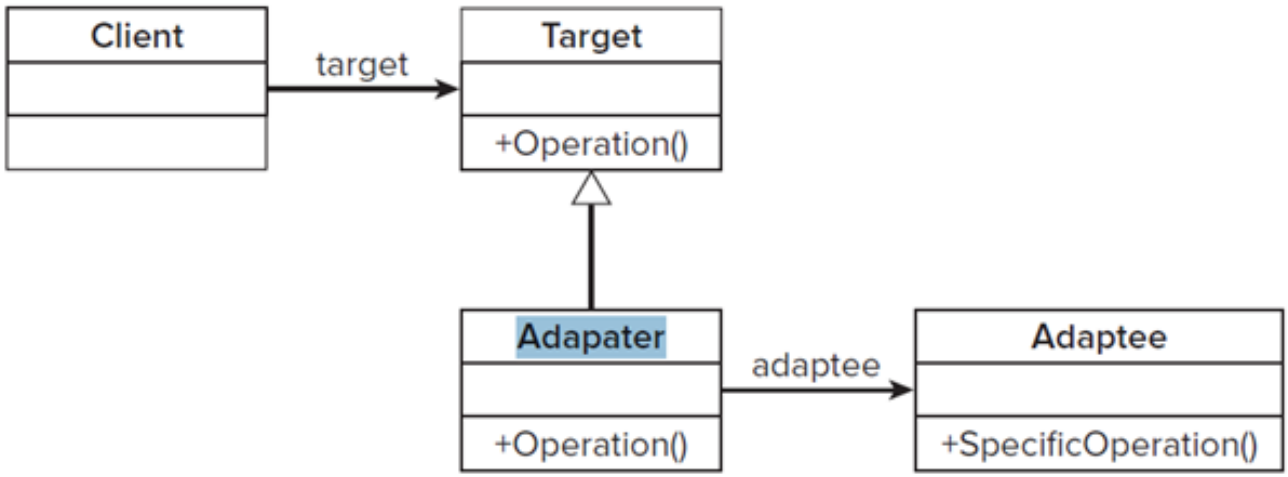
```
    }

    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؟

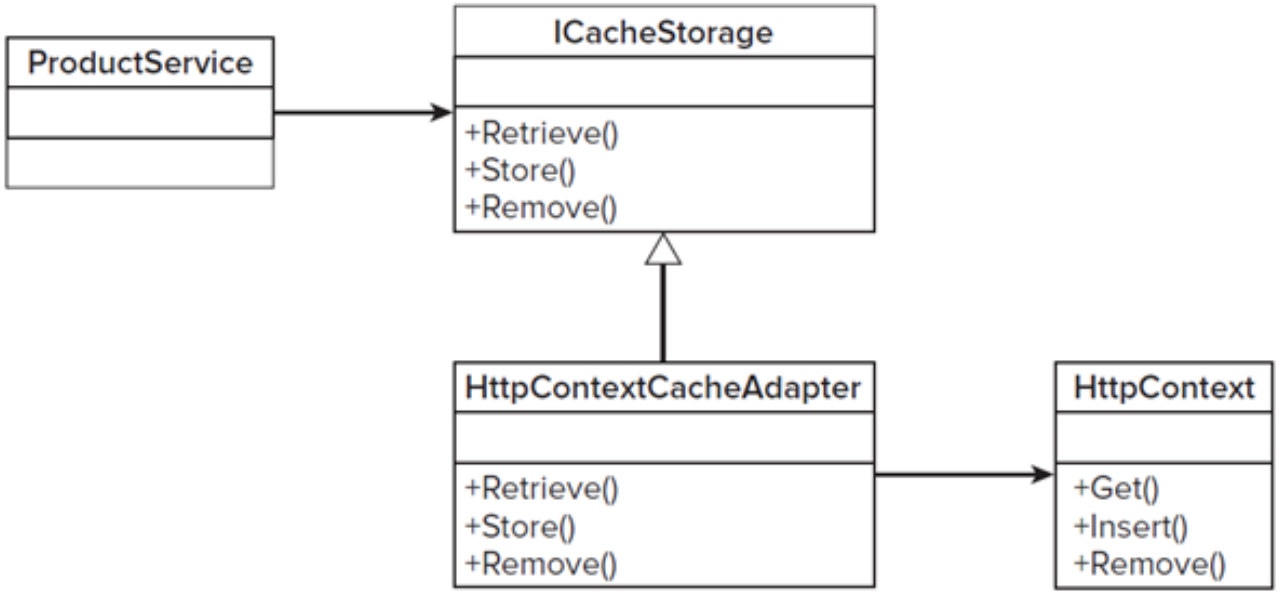
هدف این الگو به صورت ذیل در GOF مشخص شده است .«تبدیل Interface از یک کلاس به یک Interface مورد انتظار Client»

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: **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.