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

http://en.wikipedia.org/wiki/Strategy_patternاستادانه به مثال رسوندید.

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.