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