عنوان: **پیاده سازی ServiceLocator با استفاده از Microsoft Unity**
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۱/۱۲/۰۳ ۱۶:۳۰
آدرس: www.dotnettips.info
گروه‌ها: Dependency Injection, UnityContainer, ServiceLoctor

در این پست قصد دارم روش استفاه از ServiceLoctor رو به وسیله یک مثال ساده بهتون نمایش بدم. Microsoft Unity روش توصیه شده Microsoft برای پیاده سازی Dependecy Injecttion و ServiceLocator Pattern است. یک ServiceLocator در واقع وظیفه تهیه Instanceهای مختلف از کلاس‌ها رو برای پیاده سازی Dependency Injection بر عهده داره.
برای شروع یک پروژه از نوع Console Application ایجاد کنید و یک ارجاع به Assemblyهای زیر رو در برنامه قرار بدید.

Microsoft.Practices.ServiceLocation

Microsoft.Practices.Unity

Microsoft.Practices.EnterpriseLibrary.Common

اگر Assemblyهای بالا رو در اختیار ندارید می‌تونید اون‌ها رو از اینجا دانلود کنید. Microsoft Enterprise Library یک کتابخانه تهیه شده توسط شرکت Microsoft است که شامل موارد زیر است و بعد از نصب می‌تونید در قسمت‌های مختلف برنامه از اون‌ها استفاده کنید.
Enterprise Library Caching Application Block : یک CacheManager قدرتمند در اختیار ما قرار می‌ده که می‌تونید از اون برای کش کردن داده‌ها استفاده کنید.

Enterprise Library Exception Handling Application Block : یک کتابخانه مناسب و راحت برای پیاده سازی یک Exception Handler در برنامه‌ها است.

Enterprise Library Loggin Application Block : برای تهیه یک Log Manager در برنامه استفاده می‌شود.

Enterprise Library Validation Application Block : برای اجرای Validation برای Entityها با استفاده از Attribute می‌تونید از این قسمت استفاده کنید.

Enterprise Library DataAccess Application Block : یک کتابخانه قدرتمند برای ایجاد یک DataAccess Layer است با Performance بسیار بالا.
Enterprise Library Shared Library: برای استفاده از تمام موارد بالا در پروژه باید این Dll رو هم به پروژه Reference بدید. چون برای همشون مشترک است.

برای اجرای مثال ابتدا کلاس زیر رو به عنوان مدل وارد کنید.

```
public class Book
    {
        public string Title { get; set; }
        public string ISBN { get; set; }
    }
```

حالا باید Repository مربوطه رو تهیه کنید. ایتدا یک Interface به صورت زیر ایجاد کنید.

```
public interface IBookRepository
    {
        List<Book> GetBooks();
    }
```

سپس کلاسی ایجاد کنید که این Interface رو پیاده سازی کنه.

```
public class BookRepository : IBookRepository
    {
        public List<Book> GetBooks()
        {
            List<Book> listOfBooks = new List<Book>();

            listOfBooks.AddRange( new Book[]
            {
                new Book(){Title="Book1" , ISBN="123"},
                new Book(){Title="Book2" , ISBN="456"},
                new Book(){Title="Book3" , ISBN="789"},
                new Book(){Title="Book4" , ISBN="321"},
                new Book(){Title="Book5" , ISBN="654"},
            } );

            return listOfBooks;
        }
    }
```

کلاس BookRepository یک لیست از Book رو ایجاد میکنه و اونو برگشت می‌ده.
در مرحله بعد باید Service مربوطه برای استفاده از این Repository ایجاد کنید. ولی باید Repository رو به Constructor این کلاس Service پاس بدید. اما برای انجام این کار باید از ServiceLocator استفاده کنیم.

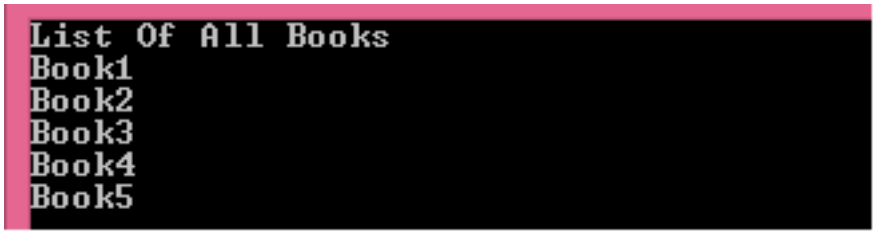
```
public class BookService
    {
        public BookService()
            : this( ServiceLocator.Current.GetInstance<IBookRepository>() )
        {
        }

        public BookService( IBookRepository bookRepository )
        {
            this.BookRepository = bookRepository;
        }

        public IBookRepository BookRepository
        {
            get;
            private set;
        }

        public void PrintAllBooks()
        {
            Console.WriteLine( "List Of All Books" );

            BookRepository.GetBooks().ForEach( ( Book item ) =>
            {
                Console.WriteLine( item.Title );
            } );
        }
    }
```

همان طور که می‌بینید این کلاس دو تا Constructor داره که در حالت اول باید یک IBookRepository رو به کلاس پاس داد و در حالت دوم ServiceLocator این کلاس رو برای استفاده دز اختیار سرویس قرار میده.
متد Print هم تمام کتاب‌های مربوطه رو برامون چاپ می‌کنه.
در مرحله آخر باید ServiceLocator رو تنظیم کنید. برای این کار کدهای زیر رو در کلاس Program قرار بدید.

```
class Program
    {
        static void Main( string[] args )
        {
            IUnityContainer unityContainer = new UnityContainer();

            unityContainer.RegisterType<IBookRepository, BookRepository>();

            ServiceLocator.SetLocatorProvider( () => new UnityServiceLocator( unityContainer ) );

            BookService service = new BookService();
```

```
            service.PrintAllBooks();
            Console.ReadLine();
        }
    }
```

در این کلاس ابتدا یک UnityContainer ایجاد کردم و اینترفیس IBookRepository رو به کلاس BookRepository؛ Register کردم تا هر جا که به IBookRepository نیاز داشتم یک Instance از کلاس BookRepository ایجاد بشه. در خط بعدی ServiceLocator برنامه رو ست کردم و برای این کار از کلاس UnityServiceLocator استفاده کردم .
بعد از اجرای برنامه خروجی زیر قابل مشاهده است.



نظرات خوانندگان

نویسنده: sunn
تاریخ: ۱۳۹۳/۰۳/۲۰ ۱۱:۱۶

سلام اول از همه ممنون بابت این همه تلاش،
دوم چرا بدون این همه کد نویسی نمیاییم از یه دستور linq ساده استفاده کنیم، نام‌ها رو بگیریم و با یک Foreach ساده پاس بدیم و این همه راه رفتیم و الان MVC از این روشها و استفاده از اینترفیسها و تزریقات وابستگی و ... که من نمیدونم این تزریقات وابستگی چیه استفاده میکنیم ، ممنون میشم توضیح بدین یا به جایی ارجاء بدین منو

نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۰۳/۲۰ ۱۱:۴۲

جهت مطالعه مباحث مقدماتی تزریق وابستگی‌ها، مراجعه کنید به دوره « بررسی مفاهیم معکوس سازی وابستگی‌ها و ابزارهای مرتبط با آن ».

عنوان: AOP با استفاده از Microsoft Unity
نویسنده: مسعود پاکدل
تاریخ: ۱۳۹۲/۰۸/۰۶ ۱۳:۵
آدرس: www.dotnettips.info
برچسب‌ها: UnityContainer, AOP

چند روز پیش فرصتی پیش آمد تا بتوانم مروری بر مطلب منتشر شده درباره AOP داشته باشم. به حق مطلب مورد نظر، بسیار خوب و مناسب شرح داده شده بود و همانند سایر مقالات جناب نصیری چیزی کم نداشت. اما امروز قصد پیاده سازی یک مثال AOP، با استفاده از Microsoft Unity Application Block را به عنوان IOC Container دارم. اگر شما هم، مانند من از UnityContainer به عنوان IOC Container در پروژه‌های خود استفاده می‌کنید نگران نباشید. این کتابخانه به خوبی از مباحث Interception پشتیبانی می‌کند. در ادامه طی یک مقاله این مورد را با هم بررسی می‌کنیم.
برای دوستانی که با AOP آشنایی ندارند پیشنهاد می‌شود ابتدا مطلب مورد نظر را یک بار مطالعه نمایند.
برای شروع یک پروژه در VS.Net بسازید و ارجاع به اسمبلی‌های زیر را در پروژه فراموش نکنید:

«Microsoft.Practices.EnterpriseLibrary.Common
«Microsoft.Practices.Unity
«Microsoft.Practices.Unity.Configuration
«Microsoft.Practices.Unity.Interception
«Microsoft.Practices.Unity.Interception.Configuration

یک اینترفیس به نام IMyOperation بسازید:

```
public interface IMyOperation
    {
        void DoIt();
    }
```

کلاسی می‌سازیم که اینترفیس بالا را پیاده سازی نماید:

```
public void DoIt()
  {
     Console.WriteLine( "this is main block of code" );
  }
```

قصد داریم با استفاده از AOP یک سری کد مورد نظر خود(در این مثال کد لاگ کردن عملیات در یک فایل مد نظر است) را به کدهای متدهای مورد نظر تزریق کنیم. یعنی با فراخوانی این متد کدهای لاگ عملیات در یک فایل ذخیره شود بدون تکرار یا فراخوانی دستی متد لاگ.
ابتدا یک کلاس برای لاگ عملیات می‌سازیم:

```
public class Logger
    {
        const string path = @"D:\Log.txt";

        public static void WriteToFile( string methodName )
        {
            object lockObject = new object();
            if ( !File.Exists( path ) )
            {
                File.Create( path );
            }
            lock ( lockObject )
            {
                using ( TextWriter writer = new StreamWriter( path , true ) )
                {
                    writer.WriteLine( string.Format( "{0} at {1}" , methodName , DateTime.Now ) );
                }
            }
        }
    }
```

حال نیاز به یک Handler برای مدیریت فراخوانی کدهای تزریق شده داریم. برای این کار یک کلاس می‌سازیم که اینترفیس ICallHandler را پیاده سازی نماید.

```
public class LogHandler : ICallHandler
    {
        public IMethodReturn Invoke( IMethodInvocation input , GetNextHandlerDelegate getNext )
        {
            Logger.WriteToFile( input.MethodBase.Name );

            var methodReturn = getNext()( input , getNext );

            return methodReturn;
        }

        public int Order { get; set; }
    }
```

کلاس بالا یک متد به نام Invoke دارد که فراخوانی متدهای مورد نظر برای تزریق کد را در دست خواهد گرفت. در این متد ابتدا عملیات لاگ در فایل مورد نظر ثبت می‌شود(با استفاده از Logger.WriteToFile). سپس با استفاده از getNext که از نوع GetNextHandlerDelegate است، اجرا را به کدهای اصلی برنامه منتقل می‌کنیم.

```
var methodReturn = getNext()( input , getNext );
```

برای مدیریت بهتر عملیات لاگ یک Attribute می‌سازیم که فقط متد هایی که نیاز به لاگ کردن دارند را مشخص کنیم. به صورت زیر:

```
public class LogAttribute : HandlerAttribute
    {
        public override ICallHandler CreateHandler( Microsoft.Practices.Unity.IUnityContainer container
)
        {
            return new LogHandler();
        }
    }
```

فقط دقت داشته باشید که کلاس مورد نظر به جای ارث بری از کلاس Attribute باید از کلاس HandlerAttribute که در فضای نام Microsoft.Practices.Unity.InterceptionExtension تعبیه شده است ارث ببرد(خود این کلاس از کلاس Attribute ارث برده است). کافیست در متد CreateHandler آن که Override شده است یک نمونه از کلاس LogHandler را برگشت دهیم.
برای آماده سازی Ms Unity جهت عملیات Interception باید کدهای زیر در ابتدا برنامه قرار داده شود:

```
var  unityContainer = new UnityContainer();

 unityContainer.AddNewExtension<Interception>();

  unityContainer.Configure<Interception>().SetDefaultInterceptorFor<IMyOperation>( new
InterfaceInterceptor() );

  unityContainer.RegisterType<IMyOperation, MyOperation>();
```

توضیح چند مطلب:
بعد از نمونه سازی از کلاس UnityContainer باید Interception به عنوان یک Extension به این Container اضافه شود. سپس با استفاده از متد Configure برای اینترفیس IMyOperation یک Interceptor پیش فرض تعیین می‌کنیم. در پایان هم به وسیله متد RegisterType کلاس MyOperation به اینترفیس IMyOperation رجیستر می‌شود. از این پس هر گاه درخواستی برای اینترفیس IMyOperation از unityContainer شود یک نمونه از کلاس MyOperation در اختیار خواهیم داشت.
به عنوان نکته آخر متد DoIt در اینترفیس بالا باید دارای LogAttribute باشد تا عملیات مزین سازی با کدهای لاگ به درستی انجام شود.

یک نکته تکمیلی:
در هنگام مزین سازی متد set خاصیت ها، به دلیل اینکه اینترفیسی برای این کار وجود ندارد باید مستقیما عملیات AOP به خود کلاس اعمال شود. برای این کار باید به صورت زیر عمل نمود:

```
var container = new UnityContainer();
container.RegisterType<Book , Book>();

container.AddNewExtension<Interception>();

 var policy = container.Configure<Interception>().SetDefaultInterceptorFor<Book>( new
VirtualMethodInterceptor() ).AddPolicy( "MyPolicy" );

  policy.AddMatchingRule( new PropertyMatchingRule( "*" , PropertyMatchingOption.Set ) );
  policy.AddCallHandler<Handler.NotifyChangedHandler>();
```

همان طور که مشاهده می‌کنید عملیات Interception مستقیما برای کلاس پیکر بندی می‌شود و به جای InterfaceInterceptor از VirtualMethodInterceptor برای تزریق کد به بدنه متدها استفاده شده است. در پایان نیز با تعریف یک Policy می‌توانیم به راحتی(با استفاده از "*") متد Set تمام خواص کلاس را به NotifyChangedHandler مزین نماییم.

سورس کامل مثال بالا

عنوان: **استفاده از Unity در پیاده سازی الگوی Service locator**
نویسنده: میثم نوایی
تاریخ: ۱۳۹۳/۰۷/۱۷ ۲۳:۵۰
آدرس: www.dotnettips.info
گروه‌ها: Dependency Injection, UnityContainer, Ioc, Service Factory, Service locator

یکی از راهکارهای پیاده سازی IOC یا همان Inversion Of Control در پروژه‌های MVC استفاده از Unity و معرفی آن به DependencyResolver خود دات نت است.
برای آشنایی با Unity و قابلیت‌های آن میتوانید به اینجا و اینجا سر بزنید.
اما برای استفاده از Unity در پروژه‌های MVC کافی است در Global یا فایل راه انداز (bootstrapper) تک تک انتزاع‌ها (Interface) را به کلاس‌های مرتبط شان معرفی کنید.

```
var container = new UnityContainer();
```

```
container.RegisterType<ISomeService, SomeService>(new PerRequestLifetimeManager());
container.RegisterType<ISomeBusiness, SomeBusiness>(new PerRequestLifetimeManager());
container.RegisterType<ISomeController, SomeController>(new PerRequestLifetimeManager());
```

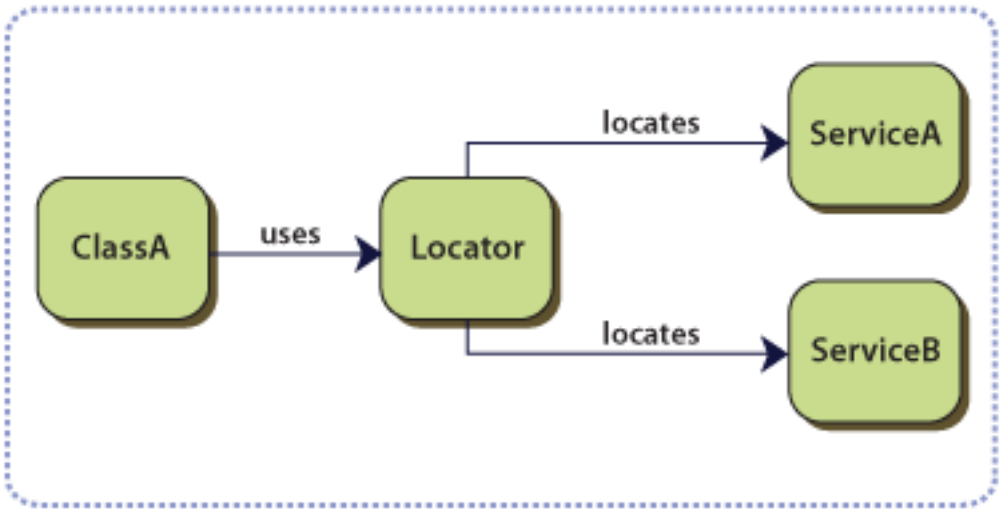
و بعد از ایجاد container از نوع UnityContainer میتوانیم آنرا به MVC معرفی کنیم:

```
DependencyResolver.SetResolver(new UnityDependencyResolver(container));
```

تا به اینجا به‌راحتی میتوانید از سرویس‌های معرفی شده در پروژه MVC استفاده کنید.

```
var someService=(ISomeService)DependencyResolver.Current.GetService(typeof(ISomeService));
var data=someService.GetData();
```

اما اگر بخواهیم از کلاس‌های معرفی شده در Unity در لایه‌های دیگر (مثلا Business) استفاده کنیم چه باید کرد؟
برای هر این مشکل راهکارهای متفاوتی وجود دارد. من در لایه سرویس از Service locator بهره برده ام. برای آشنایی با این الگو اینجا را بخوانید. اکثر برنامه نویسان الگوهای IOC و Service Locator را با هم اشتباه میگیرند یا آنها را اشتباها بجای هم بکار میبرند.
برای درک تفاوت الگوی IOC و Service locator اینجا را بخوانید.



در لایه سرویس یک کلاس Service Factory داریم که قرار است همه سرویس‌ها، برای برقراری ارتباط با یکدیگر از آن استفاده

کنند.این کلاس معمولا در لایه سرویس به اشکال گوناگونی پیاده سازی میشود که کارش وهه سازی از Interfaceهای درخواستی است. اما برای یکپارچه کردن آن با Unity من آنرا به شکل زیر پیاده سازی کرده ام

```
public class ServiceFactory : MarshalByRefObject
    {
        static IUnityContainer uContainer = new UnityContainer();
        public static Type DataContextType { get; set; }

        public static void Initialise(IUnityContainer unityContainer, Type dbContextType)
        {
            uContainer = unityContainer;
            DataContextType = dbContextType;
            uContainer.RegisterType(typeof(BaseDataContext), DataContextType, new
HierarchicalLifetimeManager());
        }
        public static T Create<T>()
        {
            return (T)Activator.CreateInstance<T>();
        }
        public static T Create<T>(string fullTypeName)
        {
            return (T)System.Reflection.Assembly.GetExecutingAssembly().CreateInstance(fullTypeName);
        }
        public static T Create<T>(Type entityType)
        {
            return (T)Activator.CreateInstance(entityType);
        }
        public static dynamic Create(Type entityType)
        {
            return Activator.CreateInstance(entityType);
        }

        public static T Get<T>()
        {
            return uContainer.Resolve<T>();
        }
        public static object Get(Type type)
        {
            return uContainer.Resolve(type);
        }
    }
```

در این کلاس ما بجای ایجاد داینامیک آبجکت‌ها، از Unity استفاده کرده‌ایم. در همان ابتدا که برنامه‌ی وب ما برای اولین بار اجرا میشود و بعد از Register کردن کلاس‌ها، می‌توانیم container را به صورت پارامتر سازنده به کلاس Service Factory ارسال کنیم. به این ترتیب برای استفاده از سرویس‌ها در لایه Business از Unity بهره میبریم.

البته استفاده از Unity برای DataContext خیلی منطقی نیست و بهتر است نوع DataContext را در ابتدا بگیریم و هرجا نیاز داشتیم با استفاده از متد Create از آن وهله سازی بکنیم.

عنوان: **پیاده سازی INotifyPropertyChanged با استفاده از Unity Container**
نویسنده: برات جوادی
تاریخ: ۱۳۹۴/۰۱/۲۴ ۱۴:۲۵
آدرس: www.dotnettips.info
گروه‌ها: WPF, UnityContainer, AOP

AOP چیست

AOP یکی از فناوری‌های مرتبط با توسعه نرم افزار محسوب می‌شود که توسط آن می‌توان اعمال مشترک و متداول موجود در برنامه را در یک یا چند ماژول مختلف قرار داد (که به آن‌ها Aspects نیز گفته می‌شود) و سپس آن‌ها را به مکان‌های مختلفی در برنامه متصل ساخت. عموما Aspects، قابلیت‌هایی را که قسمت عمده‌ای از برنامه را تحت پوشش قرار می‌دهند، کپسوله می‌کنند. اصطلاحا به این نوع قابلیت‌های مشترک، تکراری و پراکنده مورد نیاز در قسمت‌های مختلف برنامه، Cross cutting concerns نیز گفته می‌شود؛ مانند اعمال ثبت وقایع سیستم، امنیت، مدیریت تراکنش‌ها و امثال آن. با قرار دادن این نیازها در Aspects مجزا، می‌توان برنامه‌ای را تشکیل داد که از کدهای تکراری عاری است.

پیاده سازی INotifyPropertyChanged یکی از این مسائل می‌باشد که می‌توان آن را در یک Aspect محصور و در ماژول‌های مختلف استفاده کرد.

مسئله:

کلاس زیر مفروض است:

```
public class Foo
{
        public virtual int Id { get; set; }

        public virtual string Name { get; set; }
}
```

اکنون می‌خواهیم کلاس Foo را به INotifyPropertyChanged مزین، و یک Subscriber به قسمت set پراپرتی‌های کلاس تزریق کنیم.

راه حل:

ابتدا پکیج‌های Unity را از Nuget دریافت کنید:

```
PM> Install-Package Unity.Interception
```

این پکیج وابستگی‌های خود را که Unity و CommonServiceLocator هستند نیز دریافت می‌کند.

حال یک Interceptor که اینترفیس IInterceptionBehavior را پیاده سازی می‌کند، می‌نویسیم:

```
namespace NotifyPropertyChangedInterceptor.Interceptions
{
    using System;
    using System.Collections.Generic;
    using System.ComponentModel;
    using System.Reflection;
    using Microsoft.Practices.Unity.InterceptionExtension;

    class NotifyPropertyChangedBehavior : IInterceptionBehavior
    {
        private event PropertyChangedEventHandler PropertyChanged;

        private readonly MethodInfo _addEventMethodInfo =
            typeof(INotifyPropertyChanged).GetEvent("PropertyChanged").GetAddMethod();

        private readonly MethodInfo _removeEventMethodInfo =
            typeof(INotifyPropertyChanged).GetEvent("PropertyChanged").GetRemoveMethod();


        public IMethodReturn Invoke(IMethodInvocation input, GetNextInterceptionBehaviorDelegate
getNext)
        {
            if (input.MethodBase == _addEventMethodInfo)
            {
                return AddEventSubscription(input);
            }
```

```
            if (input.MethodBase == _removeEventMethodInfo)
            {
                return RemoveEventSubscription(input);
            }

            if (IsPropertySetter(input))
            {
                return InterceptPropertySet(input, getNext);
            }

            return getNext()(input, getNext);
        }

        public bool WillExecute
        {
            get { return true; }
        }

        public IEnumerable<Type> GetRequiredInterfaces()
        {
            yield return typeof(INotifyPropertyChanged);
        }

        private IMethodReturn AddEventSubscription(IMethodInvocation input)
        {
            var subscriber = (PropertyChangedEventHandler)input.Arguments[0];
            PropertyChanged += subscriber;

            return input.CreateMethodReturn(null);
        }

        private IMethodReturn RemoveEventSubscription(IMethodInvocation input)
        {
            var subscriber = (PropertyChangedEventHandler)input.Arguments[0];
            PropertyChanged -= subscriber;

            return input.CreateMethodReturn(null);
        }

        private bool IsPropertySetter(IMethodInvocation input)
        {
            return input.MethodBase.IsSpecialName && input.MethodBase.Name.StartsWith("set_");
        }

        private IMethodReturn InterceptPropertySet(IMethodInvocation input,
GetNextInterceptionBehaviorDelegate getNext)
        {
            var propertyName = input.MethodBase.Name.Substring(4);

            var subscribers = PropertyChanged;
            if (subscribers != null)
            {
                subscribers(input.Target, new PropertyChangedEventArgs(propertyName));
            }

            return getNext()(input, getNext);
        }
    }
}
```

متد Invoke : این متد Behavior مورد نظر را پردازش می‌کند (در اینجا، تزریق یک Subscriber در قسمت set پراپرتی ها). **متد GetRequiredInterfaces** : یک روش است برای یافتن کلاس هایی که با اینترفیس IInterceptionBehavior مزین شده‌اند.
پراپرتی WillExecute : ابن پراپرتی به Unity می‌گوید که این Behavior اعمال شود یا نه. اگر مقدار برگشتی آن false باشد، متد Invoke اجرا نخواهد شد.
همانطور که در متد Invoke مشاهد می‌کنید، شرط هایی برای افزودن و حذف یک Subscriber و چک کردن متد set نوشته شده و در غیر این صورت کنترل به متد بعدی داده می‌شود.

اتصال Interceptor به کلاس ها

در ادامه Unity را برای ساخت یک نمونه از کلاس پیکربندی می‌کنیم:

```
var container = new UnityContainer();

container.RegisterType<Foo, Foo>(
                new AdditionalInterface<INotifyPropertyChanged>(),
                new Interceptor<VirtualMethodInterceptor>(),
                new InterceptionBehavior<NotifyPropertyChangedBehavior>())
                .AddNewExtension<Interception>();
```

توسط متد RegisterType یک Type را با پیکربندی دلخواه به Unity معرفی می‌کنیم. در اینجا به ازای درخواست Foo (اولین پارامتر جنریک)، یک Foo (دومین پارامتر جنریک) برگشت داده می‌شود. این متد تعدادی InjetctionMember (بصورت params) دریافت می‌کند که در این مثال سه InjetctionMember به آن پاس داه شده است:

Interceptor : اطلاعاتی در مورد IInterceptor و نحوه‌ی Intercept یک شیء را نگه داری می‌کند. در اینجا از VirtualMethodInterceptor برای تزریق کد استفاده شده.

InterceptionBehavior : این کلاس Behavior مورد نظر را به کلاس تزریق می‌کند.

AddintionalInterface : کلاس target را مجبور به پیاده سازی اینترفیس دریافتی از پارامتر می‌کند. اگر کلاس behavior، متد GetRequiredInterfaces اینترفیس INotifyPropertyChanged را برمی گرداند، نیازی نیست از AddintionalInterface در پارامتر متد فوق استفاده کنید.

نکته : کلاس VirtualMethodInterceptor فقط اعضای virtual را تحت تاثیر قرار می‌دهد.

اکنون نحوه‌ی ساخت یک نمونه از کلاس Foo به شکل زیر است:

```
var foo = container.Resolve<Foo>();
(foo as INotifyPropertyChanged).PropertyChanged += FooPropertyChanged;
private void FooPropertyChanged (object sender, PropertyChangedEventArgs e)
 {
      // Do some things.......
 }
```

نکته‌ی تکمیلی

طبق مستندات MSDN ، کلاس VirtualMethodInterceptor یک کلاس جدید مشتق شده از کلاس target (در اینجا Foo) می‌سازد. بنابراین اگر کلاس‌های شما دارای Data annotation و یا در کلاس‌های Mapper یک ORM استفاده شده‌اند (مانند کلاس‌های لایه Domain)، بجای VirtualMethodInterceptor از TransparentProxyInterceptor استفاده کنید. سرعت اجرای VirtualMethodInterceptor سریعتر است ؛ اما به یاد داشته که برای استفاده از TransparentProxyInterceptor باید کلاس target از کلاس MarshalByRefObject ارث بری کند. دریافت مثال کامل این مقاله

نظرات خوانندگان

نویسنده: جلال
تاریخ: ۱۳۹۴/۰۱/۲۴ ۲۰:۲۳

این روش به همه‌ی Property Setterهای کلاس بدون در نظر گرفتن نیازهای کاربر/برنامه نویس، فراخوانی PropertyChanged رو اضافه می‌کنه. همینطور ممکنه کاربر بخواد با فراخوانی یه PropertyChanged برای یه Property، بعدش مجددا این رویداد رو برای یه Property دیگه فراخوانی کنه. به نظرم بهتره از روش‌های Attribute Base مثل این استفاده بشه.

نویسنده: برات جوادی
تاریخ: ۱۳۹۴/۰۱/۲۵ ۸:۵۴

- این Interceptor فقط کار تزریق یک Subscriber برای PropertyChanged را به عهده دارد و به سایر نیازها کاری ندارد. ضمن اینکه نیازهای کاربر/برنامه نویس اینجا کمی نامفهوم است!

- هنگام تشخیص متد set در Interceptor میتوان یک شرط دیگر گذاشت و اینکار را انجام داد.
- بسته به سناریو می‌توان از attribute هم استفاده کرد. در اینجا قصدم تزریق برای همه پراپرتی‌ها بوده، درصورتی که تزریق فقط برای برخی از انها باشد، میشه Attribute هم تعریف کرد .