عنوان: **تزریق وابستگی‌ها در حالتی‌که از یک اینترفیس چندین کلاس مشتق شده‌اند**
نویسنده: وحید نصیری
تاریخ: ۱۳۹۳/۱۰/۱۷ ۲۳:۱۷
آدرس: www.dotnettips.info
گروه‌ها: Design patterns, Dependency Injection, IoC

حین کار با ASP.NET Identity به اینترفیسی به نام IIdentityMessageService شبیه به اینترفیس ذیل می‌رسیم:

```
namespace SameInterfaceDifferentClasses.Services.Contracts
{
  public interface IMessageService
  {
   void Send(string message);
  }
}
```

فرض کنید از آن دو پیاده سازی در برنامه برای ارسال پیام‌ها توسط ایمیل و همچنین توسط SMS، وجود دارد:

```
public class EmailService : IMessageService
{
  public void Send(string message)
  {
   // ...
  }
}

public class SmsService : IMessageService
{
  public void Send(string message)
  {
   //todo: ...
  }
}
```

اکنون کلاس مدیریت کاربران برنامه، در سازنده‌ی خود نیاز به دو وهله، از این سرویس‌های متفاوت، اما در اصل مشتق شده‌ی از یک اینترفیس دارد:

```
public interface IUsersManagerService
{
  void ValidateUserByEmail(int id);
}

public class UsersManagerService : IUsersManagerService
{
  private readonly IMessageService _emailService;
  private readonly IMessageService _smsService;

  public UsersManagerService(IMessageService emailService, IMessageService smsService)
  {
   _emailService = emailService;
   _smsService = smsService;
  }

  public void ValidateUserByEmail(int id)
  {
   _emailService.Send("Validated.");
  }
}
```

در این حالت صرف تنظیمات ابتدایی انتساب یک اینترفیس، به یک کلاس مشخص کافی نیست:

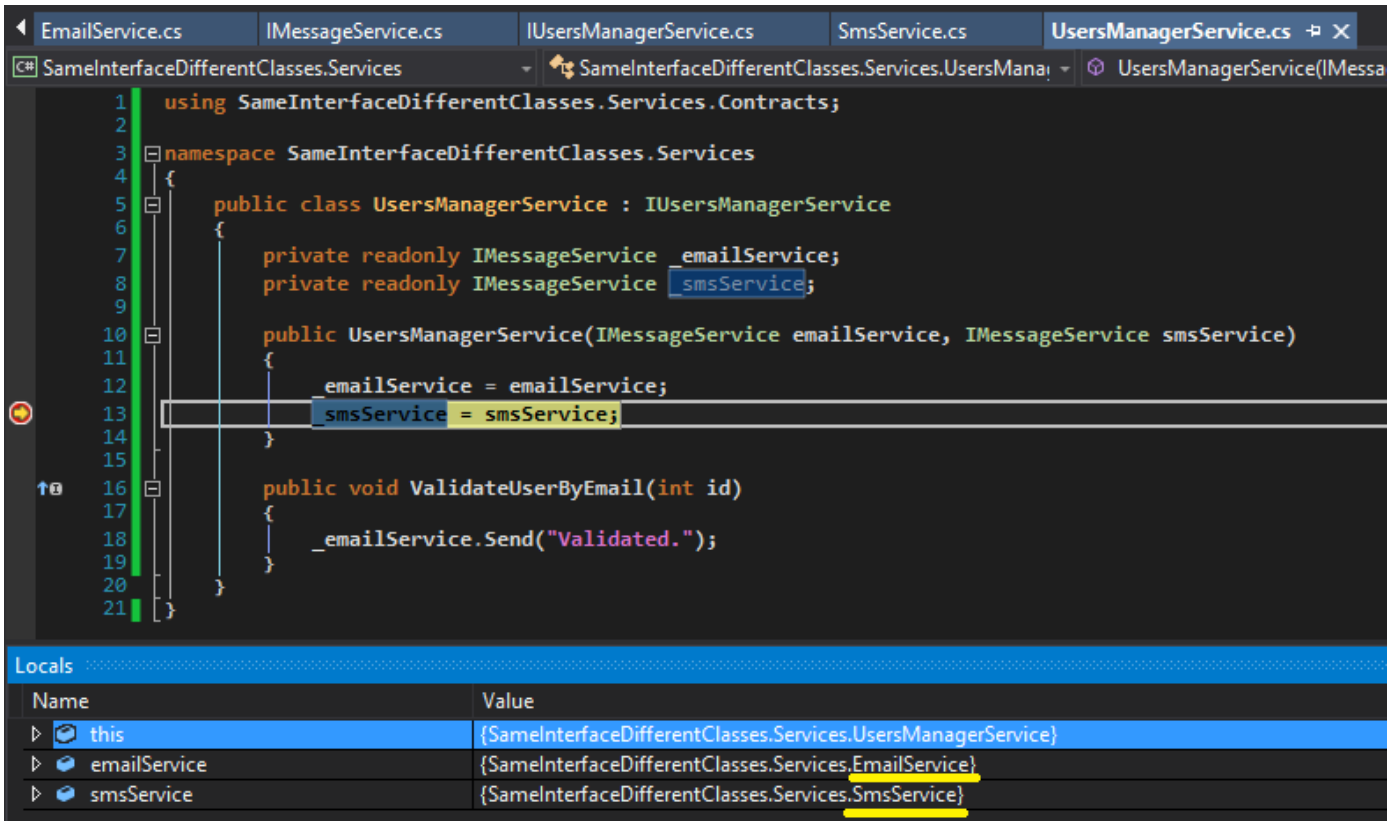
```
ioc.For<IMessageService>().Use<SmsService>();
ioc.For<IMessageService>().Use<EmailService>();
```

از این جهت که در سازنده‌ی کلاس UsersManagerService دقیقا مشخص نیست، پارامتر اول باید سرویس SMS باشد یا ایمیل؟
برای حل این مشکل می‌توان به نحو ذیل عمل کرد:

```
public static class SmObjectFactory
{
  private static readonly Lazy<Container> _containerBuilder =
   new Lazy<Container>(defaultContainer, LazyThreadSafetyMode.ExecutionAndPublication);

  public static IContainer Container
  {
   get { return _containerBuilder.Value; }
  }

  private static Container defaultContainer()
  {
   return new Container(ioc =>
   {
    // map same interface to different concrete classes
    ioc.For<IMessageService>().Use<SmsService>();
    ioc.For<IMessageService>().Use<EmailService>();

    ioc.For<IUsersManagerService>().Use<UsersManagerService>()
     .Ctor<IMessageService>("smsService").Is<SmsService>()
     .Ctor<IMessageService>("emailService").Is<EmailService>();
   });
  }
}
```

در اینجا توسط متد Ctor که مخفف Constructor یا سازنده‌ی کلاس است، مشخص می‌کنیم که اگر به پارامتر smsService رسیدی، از کلاس SmsService استفاده کن و در مورد کلاس سرویس ایمیل نیز به همین ترتیب. اینبار اگر برنامه را اجرا کنیم:

```
var usersManagerService = SmObjectFactory.Container.GetInstance<IUsersManagerService>();
usersManagerService.ValidateUserByEmail(id: 1);
```



همانطور که در تصویر مشخص است، هر کدام از پارامترها، توسط کلاس‌های متفاوتی مقدار دهی شده‌اند؛ هرچند از یک اینترفیس

مشخص استفاده می‌کنند.

کدهای کامل این مثال را از اینجا می‌توانید دریافت کنید:

Dependency-Injection-Samples/DIO9

نظرات خوانندگان

نویسنده: احسان شیروان
تاریخ: ۱۳۹۴/۰۳/۱۳ ۹:۴۱

با سلام
کلاس UsersManagerService که داره اینترفیس IUsersManagerService رو پیاده سازی می‌کنه ، اگر روزی فرضا یه سرویس دیگه مثل Sms و ایمیلی که الان توی کلاس هست به سیستم اضافه شد به نظر میاد که باید اون رو هم مثل اینها توی سازنده کلاس اضافه کرد و در هر حال کلاس ما دچار تغییر میشه .این آیا نقض OpenClose Responsibility و SRP نیست ؟

نویسنده: وحید نصیری
تاریخ: ۱۳۹۴/۰۳/۱۳ ۱۰:۹

- کلاس UsersManagerService اطلاعی از نحوه‌ی پیاده سازی IMessageService ندارد. بنابراین تغییر پیاده سازی IMessageService تاثیری در کدهای فعلی این کلاس نخواهد داشت. فقط تنظیمات IoC Container ابتدای بحث اندکی تغییر خواهد کرد و نه کدهای اصلی برنامه. بنابراین بسته‌است برای تغییر (کدهای فعلی آن نیازی به تغییر ندارند) و باز است برای توسعه (می‌توان انواع پیاده سازی‌ها را جهت این اینترفیس‌ها ارائه داد).
- همچنین اگر برنامه نیاز به سرویس‌های بیشتری از نوع IMessageService داشته باشد، بدیهی است باید کدهای متناظری هم از آن به کلاس UsersManagerService اضافه شوند و طراحی این کلاس تغییر کند. مانند این است که کنترلری امروز نیاز به لیست کاربران و سرویس کاربران دارد. روز بعد شاید نیاز به سرویس ارسال ایمیل به آن‌ها را هم پیدا کند. در این حالت طراحی این کنترلر باید تغییر کند و این تغییر ناقض اصلی نیست. صرفا برآورده کردن نیاز کاری است. حتی این تغییر هم ناقض Open Closed Principle نیست؛ چون باز است جهت تعویض پیاده سازی سرویس ایمیل و بسته‌است جهت تغییرات آتی، از این جهت که اطلاعی از جزئیات پیاده سازی اینترفیس و سرویس ایمیل ندارد.
- این مثال صرفا جهت حل مساله‌ی ASP.NET Identity ارائه شد و استفاده‌ی از یک اینترفیس برای تمام کارها. اگر قرار بود من آن‌را طراحی کنم، برای ارسال ایمیل یک اینترفیس و برای ارسال SMS یک اینترفیس دیگر ایجاد می‌کردم. یک طراحی خوب باید دارای حداقل ابهام باشد.