عنوان: **تزریق وابستگی (Dependency Injection) و توسعه پذیری**
نویسنده: شاهین کیاست
تاریخ: ۱۳۹۱/۰۶/۰۲ ۳:۱۹
آدرس: www.dotnettips.info
برچسب‌ها: Design patterns

دانستن اینکه چگونه یک نرم افزار با قابلیت نگهداری بالا بنویسیم مهم است ، برای اکثر سیستم‌های سازمانی زمانی که در فاز نگهداری صرف می‌شود بیشتر از زمان فاز توسعه می‌باشد. به عنوان مثال تصور کنید در حال توسعه یک سیستم مالی هستید ، این سیستم احتمالا بین شش ماه تا یک زمان برای توسعه نیاز دارد و بقیه‌ی دوره‌ی پنج ساله صرف نگهداری سیستم خواهد شد. در فاز نگهداری زمان صرف رفع باگ ، افزودن امکانات جدید و یا تغییر عملکرد ویژگی‌های فعلی می‌شود. مهم است که این تغییرات راحت و سریع صورت پذیرد.
اطمینان از اینکه کدها قابلیت نگهداری دارند به توسعه دهندگان احتمالی که در آینده به پروژه اضافه می‌شوند کمک می‌کند سریع کدهای فعلی را درک کنند و مشغول کار شوند. روش‌های زیادی برای افزایش قابلیت نگهداری کدها وجود دارد ، مانند نوشتن آزمون‌های واحد ، شکستن قسمت‌های بزرگ سیستم به قسمت‌های کوچک‌تر و ... در این مورد که ما از یکی از زبان‌های شئ گرا مانند C# استفاده می‌کنیم در حالت معمول کلاس‌ها باید با مسئولیت‌های مستقل و منحصر به فرد طراحی شوند به جای آنکه تمام مسئولیت‌ها از قبیل پردازش ورودی‌های کاربر ، رندر کردن HTML و حتی Query زدن به دیتابیس را به یک کلاس سپرد (مثلا Controller در MVC) باید برای هر مقصود کلاسی مجزا طراحی کرد. با این روش نتیجه اینگونه خواهد بود که می‌توان هر قسمت از عملکرد را بدون نیاز به تغییر بقیه‌ی قسمت‌های Codebase تغییر داد.
در این مطلب قصد داریم به کمک تزریق وابستگی (Dependency Injection) قسمت‌های مستقلتری توسعه دهیم. تکنیک تزریق وابستگی را نمی‌توان در یک مطلب وبلاگ و حتی یک فصل کامل از یک کتاب کامل تشریح کرد ، اگر جستجو کنید کتاب‌ها و آموزش‌های ویدویی زیادی هستند که فقط روی این تکنیک بحث و آموزش دارند. برای بیان مفهوم DI مثالی از یک سیستم ساده‌ی "چاپ اسناد" ارائه می‌کنیم ، این سیستم ممکن است کارهای متفاوتی انجام دهد :
این سیستم ابتدا باید یک سند را تحویل بگیرد ، سپس باید آن را به فرمت قابل چاپ در آورد و در انتها باید عمل اصلی چاپ را انجام دهد. برای اینکه سیستم ما ساختار خوبی داشته باشد می‌توان هر وظیفه را به کلاسی مجزا سپرد :
کلاس Document : این کلاس اطلاعات سندی که قرار است چاپ شود را نگه می‌دارد.
کلاس DocumentRepository : این کلاس وظیفه‌ی بازیابی سند از فایل سیستم (یا هر منبع دیگری) را دارد.
کلاس DocumentFormatter : یک وهله از سند را جهت چاپ آماده می‌کند.
کلاس Printer : مسئولیت ارتباط با سخت افزار Printer را دارد.
کلاس DocumentPrinter : مسئولیت سازماندهی اجزا سیستم را بر عهده دارد.
در این مطلب پیاده سازی بدنه‌ی کلاس‌های بالا اهمیتی ندارد :

```
public class DocumentPrinter
{
  public void PrintDocument(string documentName)
  {
    var repository = new DocumentRepository();
    var formatter = new DocumentFormatter();
    var printer = new Printer();
    var document = repository
      .GetDocumentByName(documentName);
    var formattedDocument = formatter.Format(document);
    printer.Print(formattedDocument);
  }
}
```

همانطور که مشاهده می‌کنید در بدنه‌ی کلاس DocumentPrinter ابتدا وابستگی‌ها نمونه سازی شده اند ، سپس یک سند بر اساس نام دریافت شده و سند پس از آماده شدن به فرمت چاپ به چاپگر ارسال شده است. کلاس DocumentPrinter به تنهایی قادر به چاپ سند نیست و برای انجام این کار نیاز به نمونه سازی همه‌ی وابستگی‌ها دارد .
استفاده از این API اینگونه خواهد بود :

```
var documentPrinter = new DocumentPrinter();
documentPrinter.PrintDocument(@"c:\doc.doc");
```

در حال حاضر کلاس DocumentPrinter از DI استفاده نمی‌کند این کلاس Loosely coupled نیست. به طور مثال لازم است که API سیستم به گونه ای تغییر پیدا کند که سند به جای فایل سیستم از دیتابیس بازیابی شود ، باید کلاس جدیدی به نام DatabaseDocumentRepository تعریف شود و به جای DocumentRepository اصلی در بدنه‌ی DocumentPrinter استفاده شود ، در نتیجه با تغییر دادن یک قسمت از برنامه مجبور به تغییر در قسمت دیگر شده ایم.(tightly coupled است یعنی به دیگر قسمت‌ها چفت شده است.)

DI به ما کمک می‌کند که این چفت شدگی (coupling) را از بین ببریم.

استفاده از constructor injection:

اولین قدم برای از بین بردن این چفت شدگی Refactor کردن کلاس DocumentPrinter هست ، پس از این Refactoring وظیفه‌ی وهله سازی مستقیم اشیاء از این کلاس گرفته می‌شود و نیازمندی‌های این کلاس از طریق سازنده به این کلاس تزریق می‌شود و فیلدهای کلاس نگهداری می‌شود . به کد زیر توجه کنید :

```
public class DocumentPrinter
{
  private DocumentRepository _repository;
  private DocumentFormatter _formatter;
  private Printer _printer;
  public DocumentPrinter(
    DocumentRepository repository,
    DocumentFormatter formatter,
    Printer printer)
  {
    _repository = repository;
    _formatter = formatter;
    _printer = printer;
  }
  public void PrintDocument(string documentName)
  {
    var document = _repository.GetDocumentByName(documentName);
    var formattedDocument = _formatter.Format(document);
    _printer.Print(formattedDocument);
  }
}
```

اکنون برای استفاده از این کلاس باید نیازمندی هایش را قبل از ارسال به سازنده نمونه سازی کرد :

```
var repository = new DocumentRepository();
var formatter = new DocumentFormatter();
var printer = new Printer();
var documentPrinter = new DocumentPrinter(repository, formatter, printer);
documentPrinter.PrintDocument(@"c:\doc.doc");
```

بله هنوز طراحی خوبی نیست اما این یک مثال ساده از DI می‌باشد. هنوز مشکلاتی در این طراحی هست ، به طور مثال کلاس DocumentPrinter به یک پیاده سازی مشخص از وابستگی هایش چفت شده است. (هنوز برای استفاده از DatabaseDocumentRepository باید DocumentPrinter را تغییر داد) پس این طراحی هنوز انعطاف پذیر نیست و نمی‌توان به سادگی برای آن آزمون واحد نوشت.

برای حل این مشکلات از Interfaceها کمک می‌گیریم. اگر به مثال قبلی بازگردیم نگرانی هر دو کلاس DocumentRepository و DatabaseDocumentRepository دریافت سند می‌باشد ، تنها پیاده سازی تفاوت دارد ، پس می‌توان یک Interface تعریف کرد

```
public interface IDocumentRepository
{
  Document GetDocumentByName(string documentName);
}
```

حال ما 2 کلاس داریم که هر دو یک Interface را پیاده سازی کرده اند می‌توان این کار را برای بقیه‌ی وابستگی‌های کلاس DocumentPrinter نیز انجام داد ، حالا باید DocumentPrinter را به گونه ای Refactor کنیم که وابستگی‌ها را بر اساس Interface دریافت کند :

```
public class DocumentPrinter
{
  private IDocumentRepository _repository;
```

```
  private IDocumentFormatter _formatter;
  private IPrinter _printer;
  public DocumentPrinter(
    IDocumentRepository repository,
    IDocumentFormatter formatter,
    IPrinter printer)
  {
    _repository = repository;
    _formatter = formatter;
    _printer = printer;
  }
  public void PrintDocument(string documentName)
  {
    var document = _repository.GetDocumentByName(documentName);
    var formattedDocument = _formatter.Format(document);
    _printer.Print(formattedDocument);
  }
}
```

حالا به سادگی می‌توان پیاده سازی‌های متفاوتی را از وابستگی‌های DocumentPrinter انجام داد و به آن تزریق کرد. همچنین اکنون نوشتن آزمون واحد هم ممکن شده است ، می‌توان یک پیاده سازی جعلی از هر کدام از Interfaceها انجام داد و جهت اهداف Unit testing از آن استفاده کرد. به طور مثال می‌توان یک پیاده سازی جعلی از IPrinter انجام داد و بدون نیاز به ارسال صفحه به پرینتر عملکرد سیستم را تست کرد.

با وجودی که موفق شدیم چفت شدگی میان DocumentPrinter و وابستگی هایش را از بین ببریم اما اکنون استفاده از آن پیچیده شده است ، هربار که قصد نمونه سازی شیء را داریم باید به یاد آوریم کدام پیاده سازی از Interface مورد نیاز است ؟ این پروسه را می‌توان به کمک یک DI Container اتوماسیون کرد.

DI Container یک Factory هوشمند است ، مانند بقیه‌ی کلاس‌های Factory وظیفه‌ی نمونه سازی اشیاء را بر عهده دارد.

هوشمندی آن در اینجا هست که می‌داند چطور وابستگی‌ها را نمونه سازی کند . DI Containerهای زیادی برای .NET وجود دارند یکی از محبوب‌ترین آنها StructureMap می‌باشد که قبلا در سایت درباره آن صحبت شده است .

برای مثال جاری پس از افزودن StructureMap به پروژه کافی است در ابتدای شروع برنامه به آن بگوییم برای هر Interface کدام شیء را وهله سازی کند :

```
ObjectFactory.Configure(cfg =>
{
  cfg.For<IDocumentRepository>().Use<FilesystemDocumentRepository>();
  cfg.For<IDocumentFormatter>().Use<DocumentFormatter>();
  cfg.For<IPrinter>().Use<Printer>();
});
```

نظرات خوانندگان

نویسنده: بهروز راد
تاریخ: ۱۳۹۱/۰۶/۰۲ ۹:۴۶

برادر، بسیار خوب و روان توضیح دادی. از معدود مقالات فارسی بود که از خواندنش لذت بردم.

موفق باشی.

نویسنده: محسن
تاریخ: ۱۳۹۱/۰۶/۰۲ ۱۰:۴۶

خیلی جالب بود . بخصوص قسمت DI Container .

نویسنده: مجتبی حسینی
تاریخ: ۱۳۹۱/۰۶/۰۲ ۲۲:۵۸

بسیار شیوا و رسا بود.
تاکید بر این نکته نیز خالی از لطف نیست که با توجه به مطلب خط آخر به جای مثلاً:

```
IDocumentFormatter documentformatter = new DocumentFormatter();
```

باید نوشت:

```
IDocumentFormatter documentformatter = ObjectFactory.GetInstance<IDocumentFormatter>();
```

نویسنده: Alex
تاریخ: ۱۳۹۱/۰۶/۰۳ ۱۴:۵۴

واقعن ساده و روان توضیحش دادین. البته Spring.net هم یکی از موارد خوبی هستش که میشه برای DI استفاده کرد.

نویسنده: حسین مرادی نیا
تاریخ: ۱۳۹۱/۰۶/۰۳ ۲۰:۱۶

مرسی. واقعا عالیه
موفق باشید

نویسنده: ایلیا اکبری فرد
تاریخ: ۱۳۹۱/۰۶/۱۳ ۱۹:۳۵

عالی بود . عالی. در صورت به مقالاتی که در این زمینه هست بیشتر بپردازین .

نویسنده: مسعود رمضانی
تاریخ: ۱۳۹۱/۰۸/۲۰ ۱۶:۳۳

با سلام و خسته نباشی

بابت مطلب خوبتون تشکر میکنم.

نویسنده: بهار قدمی
تاریخ: ۱۳۹۲/۰۴/۱۵ ۱۲:۲۰

با سلام و عرض خسته نباشید
آیا در همه جای پروژه باید ازاین روش استفاده کرد.منظورم استفاده از یک DI Container است.آیا anti - pattern ای در این مورد هم وجود دارد؟مثلا من یک پروژه‌ی ماژوالار بزرگ دارم آیا فقط در قسمت اتصال کلاس‌ها به لایه‌ی UI یا همون MVC از DI Container استفاده کنم یا هیچ جای پروژه ام new() نداشته باشم و همیشه از DI Container اسم کلاسم رو بگیرم؟ با توجه به بزرگی پروژه ام آیا Performance از دست نمیدم؟