Mark Monster is Software Engineer bij Rubicon

Dependency Injection: waarom afhankelijk zijn?

GEBRUIK VAN DI VOOR EEN BETERE APPLICATIEARCHITECTUUR

Gebruik van Dependency Injection kan voor een betere applicatiearchitectuur zorgen en maakt een einde aan de afhankelijkheden die gecreëerd worden bij het gebruik van componenten.

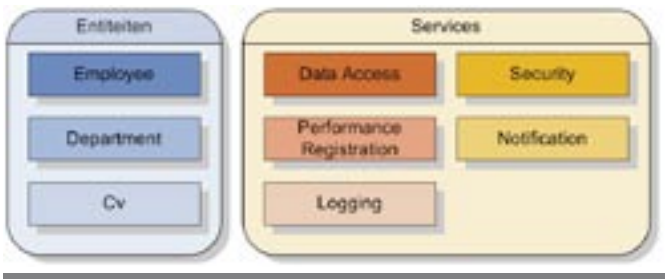
Als basis biedt Inversion of Control (IoC) en Dependency Injection (DI) een simpel mechanisme voor het leveren van componentafhankelijkheden en het beheren van de levensduur van deze afhankelijkheden. Wanneer een object andere objecten nodig heeft, bestaan er afhankelijkheden tussen deze objecten. IoC biedt services waarmee toegang kan worden verschaft tot deze afhankelijkheden. Dependency Injection is een manier om inversion of control te bereiken. In dit artikel beschrijft de auteur wat de voordelen zijn van dependency injection en laat hij zien hoe DI in de praktijk werkt.

Waarom gebruikmaken van dependency injection?

In eerste instantie is er geen grote noodzaak om gebruik te maken van dependency injection. Het is immers mogelijk om zelf met eigen middelen een vergelijkbaar resultaat te krijgen. Design patterns als Abstract Factory, Decorator, Adapter, Proxy en Facade kunnen toch ook voor een betere applicatie zorgen? Dat kunnen ze inderdaad, maar niet al deze patterns zorgen voor een eenvoudige switch tussen de afhankelijkheden. Wanneer we gebruikmaken van DI hoeven we deze design patterns geen gedag te zeggen, we kunnen ze nog steeds gebruiken in combinatie met DI. Verderop in het artikel komen we het Decorator Design Pattern tegen voor het gebruik van een interessante dienst die door een Dependency Injection Container wordt geleverd.

In tweede instantie blijkt er genoeg reden te zijn om gebruik te maken van dependency injection. Zo zorgt DI voor betere testbaarheid van de code. Door componenten los te trekken van het gebruik ervan, kunnen deze componenten individueel goed getest worden. Hetgeen ook voor unit-testing van belang is, omdat we daarmee een enkele unit willen testen. Het gebruik van een component is dan te vervangen door een Mock-object op de locaties waar de component gebruikt wordt. Op deze manier is het ook eenvoudiger te achterhalen in welke component zich een probleem bevindt. We testen immers niet zozeer de volledige keten, wat overigens ook moet gebeuren maar bij unit-testing minder van belang is, maar slechts de losse componenten. Bij gebruik van DI staat het contract, de interface, centraal. Omdat afhankelijkheden alleen aan het contract vastzitten, is een vervanging van de implementatie een kleine stap. Dit zou zelfs runtime kunnen door bijvoorbeeld een nieuwe dll met de nieuwe component te plaatsen en daarnaast de configuratie aan te passen. Hoewel 'Loose Coupling' meestal in de context van SOA en webservices wordt genoemd, kan dit ook toegepast worden op componenten.

Daarnaast biedt deze manier van werken veel flexibiliteit. Het contract is het enige dat vaststaat, hierdoor is het goed mogelijk een nieuwe implementatie voor de component in te zetten. Zo kan het heel goed dat de oude implementatie gebruikmaakte van een C++ dll, die op termijn vervangen zal worden door een webservice. Natuurlijk kan de vertaalslag die nodig is groot zijn, maar als de webservice qua functionaliteiten biedt wat er gevraagd wordt, zal een vertaalslag naar het contract mogelijk moeten zijn. Omdat er bij de ontwikkeling van een component moet worden nagedacht over de functionaliteiten die door het contract geboden worden, komen de componenten meer op zichzelf te staan. Hergebruik is op die manier beter mogelijk. Componenten die veel gebruikt worden binnen de organisatie kunnen evolueren naar een facility. Dit gaat echter te ver voor dit artikel.



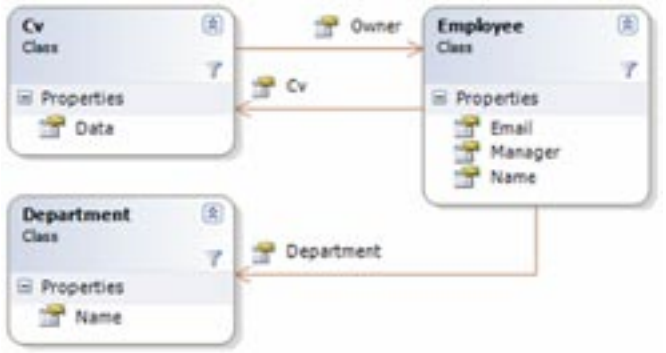
Afbeelding 1. Versimpeld model van de voorbeeldapplicatie

Aan de slag met dependency injection

Om met DI aan de slag te gaan is een tool/container noodzakelijk die daarbij helpt. Nu is het heel goed mogelijk een eigen dependency injection container te schrijven, maar waarom het wiel opnieuw uitvinden? Er zijn genoeg DI-containers te vinden die

Service	Betekenis
Data Access	Voor data access is het uitgangspunt dat de entiteiten rechtstreeks kunnen worden opgeslagen door een geleverde API, zoals bij Db4o en NHibernate het geval is.
Performance Registration	Performanceregistratie houdt in dit geval in dat data voor performance counters worden gezet. Deze zou dan met PerfMon in de gaten kunnen worden gehouden.
Logging	Logging wordt gebruikt om iedere aanroep naar data access te loggen.
Security	Security heeft in dit voorbeeld slechts zijn uitwerking op het uitlezen van een willekeurige CV. Dit is namelijk alleen toegestaan voor medewerkers binnen de verkoopafdeling.
Notification	Iedere keer dat een CV wordt opgeslagen, wordt de manager hiervan op de hoogte gebracht.

Tabel 1. Uitleg van de betekenis van de verschillende services die in de voorbeeldapplicatie aanwezig zijn.

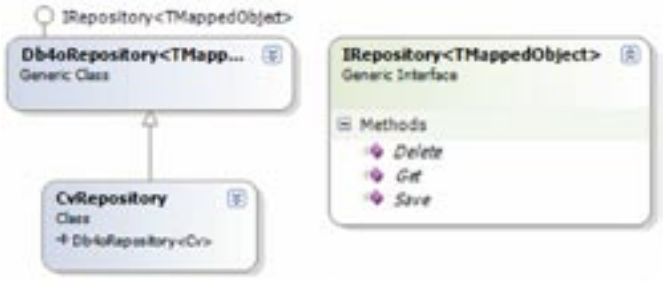


Afbeelding 2. Domeinmodel van de voorbeeldapplicatie

gebruikt kunnen worden: StructureMap; Spring.NET, bekend van Spring voor Java; Castle Windsor, die in dit artikel wordt gebruikt. Naast de gereedschapskist is er nog een doel/applicatie nodig om te verwezenlijken. Dat zal een applicatie zijn voor het beheren van curricula vitae, een veelvoorkomend aspect in bijvoorbeeld detachering. In afbeelding 1 staat een versimpeld model van deze applicatie. De user-interface negeer ik verder, meestal wordt deze door een Model View Controller-framework afgehandeld.
De services staan verder beschreven in tabel 1, hierbij wordt niet ingegaan op de implementatie, maar worden ze gebruikt als referentie in deze voorbeeldapplicatie.
Het domeinmodel staat beschreven in afbeelding 2. Hier is niets nieuws aan ten opzichte van dependency injection.
Nu een aantal onderdelen is besproken, volgt het belangrijkste onderdeel waarbij dependency injection gebruikt wordt. Het gaat hierbij om de repository-toegang, waarvoor een generieke implementatie is gemaakt die door de entiteiten Department en Employee gebruikt gaan worden, en een specifieke implementatie voor de CV-entiteit. Naast opslag wordt in CvRepository de manager op de hoogte gebracht van een wijziging. Afbeelding 3 toont de relaties tussen IRepository, Db4oRepository en CvRepository.
De code die normaal gesproken nodig is om bijvoorbeeld een nieuwe medewerker op te slaan, staat in codevoorbeeld 1. Waarbij in regel 7 de code rechtstreeks afhankelijk wordt van de implementatie van de repository, iets wat eigenlijk niet wenselijk is. Dit geldt ook voor de voor CvRepository in regel 14, omdat deze afwijkt van de generieke repository. Deze afhankelijkheden staan hard in de source-code. En dit terwijl we ons alleen afhankelijk willen maken van het generieke contract IRepository<TMappedObject>.
Laten we nu eens de slag gaan met de Windsor-container om dit probleem door middel van dependency injection op te lossen.
Laten we met simpele configuratie beginnen: codevoorbeeld 2.
De eerste component die wordt geregistreerd in de configuratie is de generieke repository. Met service wordt aangegeven aan welk contract voldaan wordt voor de afhankelijkheid, waarbij met type

```
1  Employee mark = new Employee();
2  mark.Name = "Mark Monster";
3  mark.Email = "m.monster@rubicongroup.biz";
4  mark.Department = servicesDepartment;
5  mark.Manager = manager;
6
7  IRepository<Employee> employeeRepository = new Db4oRepository
   <Employee>();
8  employeeRepository.Save(mark);
9
10 Cv cvMark = new Cv();
11 cvMark.Data = cvData;
12 cvMark.Owner = mark;
13
14 IRepository<Cv> cvRepository = new CvRepository();
15 cvRepository.Save(cvMark);
```

Codevoorbeeld 1. Standaard implementatie van toegang tot de Repository.



Afbeelding 3. Model van de repository voor de voorbeeldapplicatie

de onderliggende implementatie wordt bedoeld. De tweede component die we registreren is de CvRepository. Zoals te zien is, moet aan hetzelfde contract worden voldaan, alleen dan specifiek voor de Cv-implementatie van de generieke IRepository-interface. De brackets in de servicedefinitie voor de CvRepository wijken af van de C#-notatie voor een gesloten generieke type, deze notatie komt overeen met de CLR (Common Language Runtime) notatie. De configuratie kan worden opgenomen in de app.config of de web.config, maar kan ook in een andere xml-file staan.
Om nu gebruik te kunnen maken van de Windsor-container moet deze eerste geïnitieerd worden met de juiste configuratie: codevoorbeeld 3 regel 1-2. Daarna vragen we aan de container naar de componenten die geregistreerd zijn: codevoorbeeld 3 regel 11 en 19. Zoals duidelijk is te zien, is er alleen een afhankelijkheid naar het contract IRepository. Er staat niet langer een referentie naar Db4oRepository en CvRepository in de code. Het enige dat er gebeurt, is een verzoek om een implementatie van IRepository, de Windsor-container achterhaalt de implementatie en geeft deze ons.

```
<component
  id="generic.repository"
  service="Rubicon.DI.WindowsUI.Services.IRepository`1, Rubicon.I.WindowsUI"
  type="Rubicon.DI.WindowsUI.Services.Db4oRepository`1, Rubicon.DI.WindowsUI" />

<component
  id="cv.repository"
  service="Rubicon.DI.WindowsUI.Services.IRepository`1[[Rubicon.
  DI.WindowsUI.Entities.Cv, Rubicon.DI.WindowsUI]], Rubicon
  DI.WindowsUI"
  type="Rubicon.DI.WindowsUI.Services.CvRepository, Rubicon.DI.WindowsUI"
 />
```

Codevoorbeeld 2. Initiële configuratie van Windsor voor de repositories

```
1  IWindsorContainer container = new WindsorContainer(
2     new XmlInterpreter(new ConfigResource("castle")));
3
4  Employee mark = new Employee();
5  mark.Name = "Mark Monster";
6  mark.Email = "m.monster@rubicongroup.biz";
7  mark.Department = servicesDepartment;
8  mark.Manager = manager;
9
10 IRepository<Employee> employeeRepository =
11    container.Resolve<IRepository<Employee>>();
12 employeeRepository.Save(mark);
13
14 Cv cvMark = new Cv();
15 cvMark.Data = cvData;
16 cvMark.Owner = mark;
17
18 IRepository<Cv> cvRepository =
19    container.Resolve<IRepository<Cv>>();
20 cvRepository.Save(cvMark);
```

Codevoorbeeld 3. Windsor-implementatie van toegang tot de repository

```
1  public class LoggingRepository<TMappedObject> : IRepository<TMappedObject>
2  {
3     private readonly IRepository<TMappedObject> innerRepository;
4
5     public LoggingRepository(IRepository<TMappedObject> innerRepository) {
6        this.innerRepository = innerRepository;
7     }
8  }
```

Codevoorbeeld 4. Implementatie van een generieke repository met een generieke repository als constructor-parameter

```
1  IRepository<Cv> cvRepository =
2     new SecurityRepository(
3        new LoggingRepository<Cv>(
4           new PerformanceCountingRepository<Cv>(
5              new CvRepository())));
6  cvRepository.Save(cvMark);
```

Codevoorbeeld 5. Implementatie van een decorator-keten.

```
1  IRepository<Cv> cvRepository =
2     container.Resolve<IRepository<Cv>>();
3  cvRepository.Save(cvMark);
```

Codevoorbeeld 6. Windsor-implementatie van toegang tot de repository decorator-keten

In het geval IRepository<Employee> voldoet deze aan de generieke Db4oRepository, terwijl het verzoek om een IRepository<Cv> de CvRepository oplevert. Wanneer later blijkt dat de standaard repository voor medewerkers niet voldoet, kan deze simpelweg vervangen worden door de configuratie van Windsor aan te passen met de nieuwe implementatie. De bestaande code hoeft daar niet voor worden aangepast. De code richt zich nu meer op wat er gebeurt in plaats van hoe dat gebeurt.

Een bijzondere eigenschap van dependency injection met generics is de mogelijkheid dat de generieke afhankelijkheden zelf ook generieke afhankelijkheden kunnen bevatten via de constructor of een property. Dit klinkt vreemd, maar illustreert zich het beste aan de hand van codevoorbeeld 4.

Een oplettende lezer ziet waarschijnlijk al een Decorator-design pattern in codevoorbeeld 4, in dit geval een decorator voor het loggen van een aanroep naar IRepository. Daarnaast zijn er nog decorators voor security en performance registration. Deze decorators kunnen gebruikt worden voor een decorator-keten zoals in afbeelding 4.

De realisatie van de decorator-keten zorgt voor een grote hoeveelheid afhankelijkheden in een omgeving zonder DI; zie codevoorbeeld 5. Natuurlijk kan codevoorbeeld 5 herschreven worden door gebruik te maken van DI. Codevoorbeeld 6 laat het resultaat zien, code zonder afhankelijkheden naar de implementatie, alleen een afhankelijkheid naar het contract.

Natuurlijk vergt dit iets meer van Windsor, de configuratie is iets uitgebreider, maar het concept blijft gelijk zoals te zien is in codevoorbeeld 7. De configuratie is ook goed aan te passen, zo kan de LoggingRepository heel eenvoudig uit de decorator-keten worden gehaald. Zoals te zien is, wordt de innerRepository gezet door middel van een referentie naar de juiste component. Deze referentie heeft de volgende structuur: ${*componentid*}. De container is intelligent genoeg om te bepalen welke keten er moet ontstaan. Wanneer er fouten in de configuratie zitten, zal dit duidelijk worden tijdens het opstarten van de Windsor-container.

Afbeelding 4. Decorater-keten: Security, logging, performance registration, data access

```
<component id="cv.security.repository"
   service="Rubicon.DI.WindowsUI.Services.IRepository`1[[Rubicon.
   DI.WindowsUI.Entities.Cv, Rubicon.DI.WindowsUI]], Rubicon.DI.WindowsUI"
   type="Rubicon.DI.WindowsUI.Services.SecurityRepository,
   Rubicon.DI.WindowsUI">
   <parameters>
      <innerRepository>${cv.logging.repository}</innerRepository>
   </parameters>
</component>

<component id="cv.logging.repository"    service="Rubicon.DI.WindowsUI.
   Services.IRepository`1[[Rubicon.DI.WindowsUI.Entities.Cv,
   Rubicon.DI.WindowsUI]], Rubicon.DI.WindowsUI"
   type="Rubicon.DI.WindowsUI.Services.LoggingRepository`1[[Rubicon.DI.
WindowsUI.Entities.Cv, Rubicon.DI.WindowsUI]], Rubicon.DI.WindowsUI">
   <parameters>
      <innerRepository>${cv.performancecounting.repository}</innerRepository>
   </parameters>
</component>

<component id="cv.performancecounting.repository"
   service="Rubicon.DI.WindowsUI.Services.IRepository`1[[Rubicon.
   DI.WindowsUI.Entities.Cv, Rubicon.DI.WindowsUI]], Rubicon.DI.WindowsUI"
   type="Rubicon.DI.WindowsUI.Services.PerformanceCountingRepository`
   1[[Rubicon.DI.WindowsUI.Entities.Cv, Rubicon.DI.WindowsUI]], Rubicon.
   DI.WindowsUI">
   <parameters>
      <innerRepository>${cv.repository}</innerRepository>
   </parameters>
</component>

<component id="cv.repository" service="Rubicon.DI.WindowsUI.Services.
IRepository`1[[Rubicon.DI.WindowsUI.Entities.Cv, Rubicon.DI.WindowsUI]],
 Rubicon.DI.WindowsUI"
   type="Rubicon.DI.WindowsUI.Services.CvRepository, Rubicon.
DI.WindowsUI" />
```

Codevoorbeeld 7. Windsor-configuratie van de CvRepository decorator-keten

Afhankelijkheden de wereld uit

Nu er een manier is gevonden om afhankelijkheden de wereld uit te helpen, kunnen we dit gaan toepassen. De toepassing in dit artikel is natuurlijk een voorbeeld, er zijn nog vele andere toepassingen mogelijk. Dit kan bijna tot in het oneindige, wat ook geldt voor objectoriëntatie. Het is natuurlijk belangrijk om hier de gulden middenweg in te vinden, zodat er voordeel uit DI gehaald kan worden. In het voorbeeld in dit artikel is de Castle Windsor-container gebruikt, waardoor de oplossing hier afhankelijk van wordt. De voordelen zijn bekend: betere testbaarheid, loose coupling, flexibiliteit en herbruikbaarheid. Voor wie meer wil weten over IoC en DI verwijs ik graag naar Martin Fowler die een uitstekend artikel heeft geschreven over IoC en DI.

Mark Monster is Software Engineer bij Rubicon (www.rubicongroup.biz). Hij houdt zich bezig met Custom Development in .NET.

Referenties

Inversion of Control Containers and the Dependency Injection pattern: www.martinfowler.com/articles/injection.html

Castle Project – MicroKernel / Windsor Container: www.castleproject.org/container

Castle Project - Container Getting Started: www.castleproject.org/container/gettingstarted