Dependency Injection in .NET

Een wereld zonder afhankelijkheden

Willem Boeré en Timo Swanenberg

Het gebruik van technieken en ontwerppatronen onder de verzamelnaam 'Dependency injection' faciliteert het bouwen van een zogenaamde 'loosely coupled' applicatie. We leggen dat in dit artikel uit met gebruikmaking van voorbeelden op basis van een concrete applicatie.

Een applicatie die volgens het object-georiënteerde paradigma is ontworpen en gebouwd bestaat dikwijls uit meerdere klassen. Objecten van deze klassen werken vaak niet geïsoleerd, in de meeste gevallen hebben ze andere objecten nodig om hun taak naar behoren te kunnen uitvoeren. Tegenwoordig worden veel applicaties 'modulair' opgezet, met de doelstelling om applicatie onderdelen los van elkaar te kunnen bouwen, wijzigen, testen en natuurlijk deployen. Helaas wordt deze doelstelling niet altijd (volledig) bereikt.
De oorzaak hiervan is dat doorgaans diverse applicatie onderdelen (c.q. lagen) direct aan elkaar gekoppeld zijn en daarmee sterk afhankelijk worden van elkaar ('tightly coupled'). Het schrijven van 'tightly coupled' code heeft tot gevolg dat verschillende applicatie onderdelen moeilijk los van elkaar kunnen worden gewijzigd, kunnen functioneren of geïsoleerd kunnen worden getest.

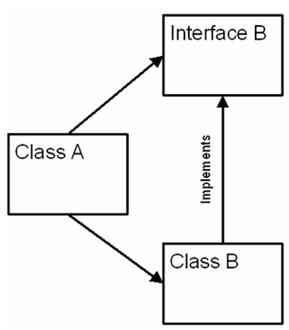
Dependency Injection theorie

Dependency Injection (DI) is typisch een techniek waar de één niet meer zonder kan, terwijl de ander er nog nooit van heeft gehoord of het nog nooit heeft toegepast. Reden genoeg om deze techniek eens van dichterbij te bekijken.
DI is een verzameling technieken en patronen die het mogelijk maken ´loosely coupled´ code te schrijven. Dit houdt in dat objecten data kunnen uitwisselen zonder dat hun relatie hard (in de broncode) is vastgelegd, althans niet door de programmeurs van de originele classes.
Traditioneel gebeurt dat vaak wel, met als gevolg dat hergebruik van deze classes moeilijk is.

Wanneer een object een ander object nodig heeft om te functioneren, ontstaat er een afhankelijkheid tussen deze objecten. Martin Fowler introduceerde de term Dependency Injection om specifiek te verwijzen naar het managen van afhankelijkheden binnen het kader van Inversion of Control (IoC). IoC is een veel bredere term die DI omvat (maar er zich niet toe beperkt) en DI is dan ook een manier om IoC te bereiken. Inversion of control (IoC) kenmerkt zich dan ook door 'services' waarmee toegang kan worden verkregen tot de eerder genoemde afhankelijkheden. Voordat we dieper ingaan op het onderwep, eerst een aantal voordelen van DI op een rijtje:

- DI geeft de mogelijkheid om hergebruik van code eenvoudiger te maken.
- DI maakt het eenvoudig mogelijk om een specifieke implementatie te vervangen door een ander.
- DI geeft de mogelijkheid om teams parallel te laten werken aan functionaliteit.
- DI maakt het eenvoudiger om code te onderhouden.
- DI maakt het (unit)testen van code eenvoudiger.

Middels DI is het mogelijk om onderdelen los van elkaar te ontwikkelen en op een later tijdstip te verbinden. Er zijn meerdere manieren om dit te bereiken. Zo kan men de logica zelf implementeren of er kan gebruik worden gemaakt van een framework dat dit voor je doet. Zo´n framework of tool wordt binnen het DI paradigma ook wel een container genoemd .
Tijdens het programmeren van software lopen we tegen verscheidene uitdagingen aan. Zo zijn er enerzijds aparte componenten die samen moeten werken en anderzijds willen we die onderdelen zo onafhankelijk mogelijk van elkaar ontwikkelen en testen zodat ze op zoveel mogelijk plaatsen kunnen worden gebruikt.
Door componenten (classes) los te trekken van het gebruik ervan, kunnen deze componenten ook individueel goed worden getest. Het zal daarom geen verrassing zijn dat DI populair is onder de aanhangers van 'Test Driven Development'. DI maak het mogelijk een enkele unit in isolatie te testen.
Een component dat tijdens het normale functioneren van de class



FIGUUR 1: ZONDER DI

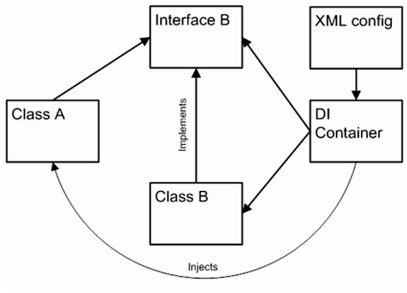
wordt gebruikt, is bij gebruik van DI tijdens het unittesten eenvoudig te vervangen door een Mock-object . Op deze manier is het eenvoudiger te achterhalen in welke class een probleem zich bevindt.

Concepten

Wanneer classes sterk zijn gekoppeld kan een wijziging in class A aanleiding geven tot een golf van wijzigingen in andere classes. Daarom streeft men er naar om de koppeling tussen classes, en dus objecten, zoveel mogelijk te vermijden, zie figuur 1.

Een goede manier om dit te bereiken, is door te ontwerpen of te programmeren tegen interfaces. Dit houdt in dat je er voor zorgt dat objecten zoveel mogelijk met elkaar 'praten' in termen van interfaces en niet via concrete types. Hierdoor wordt het bijvoorbeeld eenvoudiger om een class te vervangen door een andere, zonder dat overige klassen gewijzigd moeten worden, zie figuur 2.

FIGUUR 2: MET DI



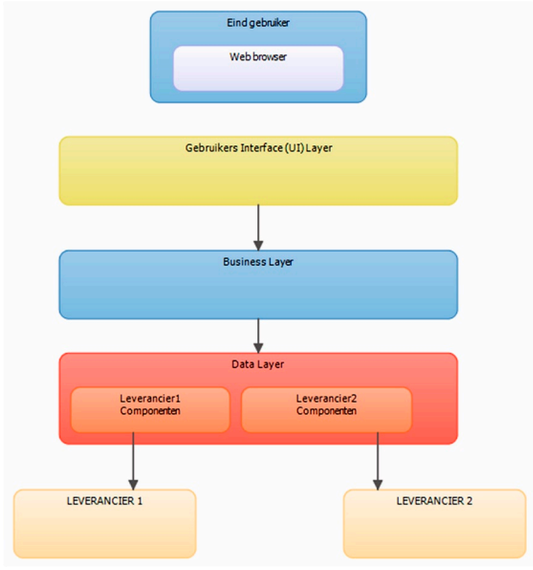
Hier valt te zien dat de afhankelijkheid tussen class A & B niet langer in de code aanwezig is, maar in dit specifieke geval is verplaatst naar een XML configuratie file. De gebruikte DI container ziet dat Class A een concreet type van Interface B nodig heeft om te opereren, er wordt in de XML configuratie file gekeken of er een concreet type beschikbaar is voor deze interface. Wanneer er een class beschikbaar is wordt deze door de container geïnstantieerd en geïnjecteerd in Class A. Deze functionaliteit van de container wordt ook wel 'object composition' genoemd.
Er zijn drie verschillende smaken van object composition, te weten: Constructor injection, Setter injection en Interface injection. Deze drie vormen van injectie zijn ieder op zich bijzonder nuttig en richten zich op een specifiek 'probleem'.
Elke vorm heeft echter ook zijn eigen nadelen; Interface injection heeft het nadeel dat bij classes met veel verschillende afhankelijkheden ook veel interfaces moeten worden geïmplementeerd. Constructor injection heeft als nadeel dat de constructor veel parameters kan krijgen bij vele afhankelijkheden.
Naast object composition kan middels de configuratie van een gemiddelde DI container veel meer worden geconfigureerd. Zo kan de 'lifetime' van objecten worden bepaald, men kiest bijvoorbeeld om een specifieke mapping de singleton lifetime te geven. Het kiezen hiervan heeft tot gevolg dat wanneer de container (binnen dezelfde applicatie context) meerdere malen een aanvraag krijgt voor het retourneren van een implementatie van een specifieke afhankelijkheid, hetzelfde object wordt geretourneerd. Daarnaast kan bijvoorbeeld ook worden gekozen voor een modus waarbij bij iedere aanvraag een nieuw object wordt geïnstantieerd.

De derde bekende feature van de meeste DI containers luistert naar de naam 'Interception'. Interception is nog het best te vergelijken met het decorator design pattern. Stel je voor dat we beschikken over een specifieke repository welke we injecteren in een service die hier gebruik van maakt. Wanneer we later de wens krijgen om al de calls naar onze repository te loggen, kunnen we simpelweg een nieuwe klasse schrijven welke al deze calls logt en vervolgens de correcte methode op de customer repository aanroept. De DI container faciliteert vervolgens dat deze decorator klasse wordt gebruikt. Hierdoor kunnen we logging toevoegen en ons toch blijven conformeren aan het Single Responsibility Principle.
Om een goed voorbeeld te geven van hoe Dependency Injection (DI) werkt, geven we een aantal voorbeelden hoe bepaalde praktijksituaties werken zonder DI en hoe dit anders zou kunnen door te werken met DI.
In het volgende voorbeeld wordt weergegeven hoe een applicatie ontworpen kan worden waarbij rekening wordt gehouden met applicatie onderdelen die een grote kans hebben om in de toekomst te wijzigen. Bij het ontwerpen worden DI en IoC ontwerppatronen toegepast om de eerder genoemde problemen het hoofd te kunnen bieden.

De case

Functionele input:
De klant vraagt om een website waar tuinmeubilair van verschillende leveranciers c.q. fabrikanten gekocht kan worden. De leveranciers bieden ieder op hun eigen manier hun product- en prijsinformatie aan. Het is zeer waarschijnlijk dat in de maanden na de oplevering van de website er nieuwe leveranciers bijkomen, die ieder weer hun eigen manier hebben van het aanleveren van product- en prijsinformatie. De klant wil dat nieuwe leveranciers snel, goedkoop en gemakkelijk gekoppeld kunnen worden aan de website.

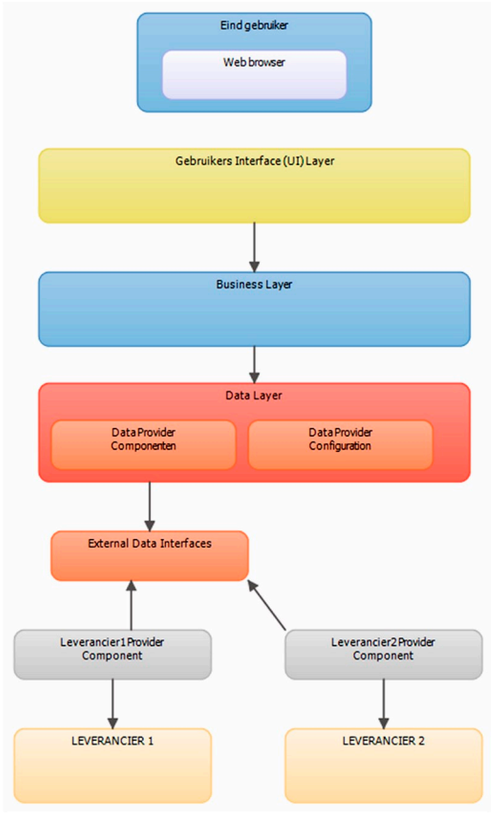
De architectuur:
Er is voor de website een gekozen voor een 3-layered application pattern. Dit houdt in dat er 3 applicatie lagen zijn met ieder een eigen verantwoordelijkheid: De gebruikers interface (UI) laag, de business laag en de data laag. Normaliter zou deze keuze in onderstaande afbeelding resulteren:



FIGUUR 3

In figuur 3 is te zien dat er tussen de businesslaag en de datalaag een harde verwijzing staat. Dit is opzich goed, er is echter binnen de datalaaglogica opgenomen (de verschillende leverancier com-

ponenten) die een grote kans hebben om te wijzigen. Wat uiteraard in het voordeel van zowel de opdrachtgever als opdrachtnemer werkt is dat, wanneer er een nieuwe leverancier moet worden toegevoegd, er weinig wijzigingen hoeven te worden gemaakt en dat de applicatie niet helemaal opnieuw dient te worden getest. Om te kunnen voldoen aan deze wens, met de concepten van DI in het achterhoofd, kijken we eens naar figuur 4.



FIGUUR 4

In figuur 4 is te zien dat de datalaag aanzienlijk is gewijzigd. Binnen de datalaag zijn zogenoemde dataprovidercomponenten gedefinieerd. Deze componenten zijn verantwoordelijk voor het ophalen van leveranciers product- en prijsinformatie uit de leverancierscomponenten. De leveranciersinformatie wordt altijd gecommuniceerd volgens een interface gedefinieerd in de externe data interfaces. Er is geen harde afhankelijkheid tussen de datalaag en de leveranciersprovidercomponenten.

De dataprovider configuratie is verantwoordelijk voor het bijhouden van welke leveranciers gekoppeld zijn, of in werkelijke zin, welke leverancierscomponenten er zijn gekoppeld.
Voor elke leverancier wordt een providercomponent geschreven. Elk component heeft een afhankelijkheid naar de externe data interfaces, die verplicht moeten worden gebruikt.
Wanneer er een nieuwe leverancier bijkomt, hoeft er alleen een nieuw leveranciersprovidercomponent te worden geïmplementeerd, geconfigureerd en getest. Dit alles zonder de gehele applicatie te hoeven aanraken. Uiteraard geldt dit ook voor het aanpassen van bestaande leveranciersprovidercomponenten.

In-depth

Laten we een tastbaar voorbeeld nemen. Stel dat we een eenvoudig programma willen maken om klantgegevens mee op te halen.

De specifieke logica voor het ophalen van de klanten (bv. uit een SQL database, of via de SharePoint BCS) zal worden ingekapseld in een gespecialiseerde klasse. In dit voorbeeld wordt de klant uit een database opgehaald:

```
public class DbCustomerRepository : ICustomerRepository
    {
        public Customer RetrieveCustomer(int userId)
        {
            // get customer from database by id and return it
        }
    }
```

Ook maken we een klasse die de business logica bevat en onze specifieke service gebruikt:

```
public class CustomerService
    {
        private ICustomerRepository _customerRepository;

        public CustomerService()
        {
            _customerRepository = new DbCustomerRepository();
        }

        public Customer GetCustomer(int userId)
        {
            var customer = _customerRepository.
RetrieveCustomer(userId);

            //Do something with the customer

            return customer;
        }
    }
```

Zoals is te zien, is de 'CustomerService' klasse afhankelijk van de 'DbCustomerRepository'.
Nu is het niet ondenkbaar dat na verloop van tijd wordt besloten om de klanten niet langer uit de SQL database te halen, maar om bijvoorbeeld gebruik te maken van SharePoint 2010 BCS.
De nieuwe implementatie van onze 'customer repository' brengen we onder in een nieuwe klasse:

```
public class BcsCustomerRepository : ICustomerRepository
    {
        public Customer RetrieveCustomer(int userId)
        {
            // get customer through SharePoint 2010 BCS by id
and return it
        }
    }
```

Waar we nu tegenaan lopen is dat de 'CustomerService' klasse moet worden aangepast als we onze nieuwe klasse willen gebruiken. In het bovenstaande voorbeeld is al netjes gebruik gemaakt van interfaces, maar hier duikt een ander probleem op. Er bestaat namelijk geen polymorfisme voor constructors. Men kan niet zoiets als new ICustomerRepository() opschrijven en verwachten dat er, afhankelijk van de situatie, een object van de klasse 'DbCustomerRepository' of 'BcsCustomerRepository' wordt geïnstantieerd. Ergens in de code moet worden aangeven welke implementatie er precies wordt gebruikt, maar de vraag is waar?

Een traditionele manier om te realiseren dat de keuze voor de concrete klasse gecentraliseerd is, is het toepassen van het factory design pattern. Dit houdt in dat er een afzonderlijke klasse wordt gedefinieerd die louter verantwoordelijk is voor het maken van concrete instanties van de interfaces die we nodig hebben. Dit lost het probleem deels op, de rest van de applicatie kan nu program-

meren tegen interfaces en wordt niet 'vervuild' met het aanmaken van concrete implementaties.
Door de bovenstaande voorgestelde factory hebben we 'loose coupling' tussen de 'CustomerService' en de 'ICustomerRepository' weten te realiseren. Echter, er is nu wel een sterke band met de factory klasse, die op zijn beurt dan weer gekoppeld is met een specifieke implementatie. Het probleem is hiermee dus nog niet volledig verholpen, hooguit gecentraliseerd.

Dit scheelt al moeite wanneer de applicatie dient te worden aangepast, zoals in dit scenario. De factory heeft nog steeds een harde afhankelijkheid op de concrete repository klasse; wat als we de logica van de Customer Service klasse willen (unit)testen zonder daarbij onvermijdelijk ook de repository laag mee te nemen? We kunnen het namelijk eerder een regressie test dan een unit test noemen, wanneer de onderliggende database aanwezig moet zijn en op zijn beurt valide data moet retourneren.

Een oplossing voor het eerder geschetste probleem kan gevonden worden door gebruik te maken van de eerder genoemde techniek dependency injection, in dit voorbeeld wordt gebruik gemaakt van de specifieke vorm 'constructor injection'. De 'CustomerService' klasse kan hiermee worden herschreven naar:

```
public class CustomerService
    {
        private ICustomerRepository _customerRepository;

        public CustomerService(ICustomerRepository
customerRepository)
        {
            _customerRepository = customerRepository;
        }
    //And the rest of the class
```

Via de constructor wordt een concrete variant van de customer repository geïnjecteerd. Wanneer we nu de Customer Service geisoleerd willen unit testen, kunnen we een 'test variant' van de repository injecteren.

```
[TestClass]
public class CustomerServiceUnitTests
{
    [TestMethod]
    public void TestCustomerServiceReturnsCorrectUser()
    {
        var cs = new CustomerService(new
TestCustomerRepository());

        Assert.IsNotNull(cs); //Check if the service is
created correctly

        var userId = 1;
        var customer = cs.GetCustomer(userId);

        Assert.IsNotNull(customer); //Check if the service
returns a
customer
        Assert.AreEqual(userId, customer.id); //Check if the
service returns the expected customer
    }
}
```

We kunnen nu tevens een stap verder gaan, er kan ook gebruik worden gemaakt van een mock object. In het bovenstaande voorbeeld zijn we namelijk helemaal niet geïnteresseerd in de werking van de specifieke TestCustomerRepository, de enige functie die het object vervult is het kunnen testen van de CustomerService.

Gebruikmakend van mocking framework 'Moq' kunnen we bijvoorbeeld dit schrijven:

```
[TestClass]
    public class CustomerServiceTest
    {
        private Mock<ICustomerRepository> _customerRepository;
        private CustomerService _customerService;

        [TestInitialize]
        public void Initialize()
        {
            _customerRepository = new Mock<ICustomerRepository>();

_customerService = new CustomerService(_customerRepository.
Object);
        }

        [TestMethod, Description("Must get the customer by
using the
customer repository.")]
        public void GetCustomer01()
        {
            int userId = 123;

            _customerService.GetCustomer(userId);


_customerRepository.Verify(c => c.RetrieveCustomer(userId),
Times.Once());
        }
    }
```

Bovenstaand codevoorbeeld toont hoe we de customer solution geïsoleerd testen. We maken gebruik van een mock van de ICustomerRepository om de afhankelijkheid op de repository te vervullen. Vervolgens testen we in dit voorbeeld of bij het aanroepen van de GetCustomer methode van de service er ook daadwerkelijk een call wordt gedaan op de onderliggende repository met de meegegeven userId. De flexibiliteit van mocking frameworks maakt het eenvoudig geïsoleerde stukken code te testen, zonder specifieke test classes te hoeven aanmaken.

Conclusie

Het is erg eenvoudig om tightly coupled code te schrijven, dit gebeurt dan ook te pas en te onpas. Vaak wordt het argument gebruikt dat het opdelen van applicaties in meerdere libraries een applicatie modulair maakt. Echter, wanneer de verschillende libraries doordrongen zijn van onderlinge afhankelijkheden, heeft deze opdeling verder weinig praktisch nut. Strikt genomen, het opdelen in zoveel mogelijk libraries maakt een applicatie niet modulair. Het aantal en de richting van de onderlinge afhankelijkheden zijn de bepalende factoren. Dependency injection geeft ons de mogelijkheid om deze afhankelijkheden te centraliseren en configureerbaar te maken. Het komt kort gezegd neer op het programmeren tegen een abstractie, waarbij er vanuit mag worden gegaan dat tijdens runtime de concrete invulling zal worden aangeleverd. De voordelen hiervan zijn dat de geschreven code eenvoudiger is te testen, gemakkelijker is te onderhouden en doorgaans beter is te hergebruiken.

Een uitgewerkt voorbeeldproject is te downloaden van www.codingarchitects.com.

Willem Boeré en Timo Swanenberg, zijn beide werkzaam bij Coding Architects. Ze zijn te bereiken op wboere@codingarchitects.com en tswanenberg@codingarchitects.com.