Aspect Oriented Programming onder de loep

BEHEER SYSTEEMBREDE VERANTWOORDELIJKHEDEN

Componenten zijn goed van elkaar los te koppelen door gebruik te maken van Dependency Injection. De systeembrede verantwoordelijkheden koppelen we los van de core-code door middel van Aspect Oriented Programming (AOP). In dit artikel nemen we in vogelvlucht de mogelijkheden van AOP door. Dit gebeurt aan de hand van twee voorbeelden die in de meeste software-ontwikkeltrajecten aan de orde komen.

Het identificeren en afzonderen van verantwoordelijkheden in een systeem is een best practice tijdens het ontwikkelen. Door middel van Dependency Injection is het mogelijk afhankelijkheden deels of volledig automatisch aan elkaar te knopen. In .NET Magazine #18 staat een artikel van mijn hand waarin Dependency Injection wordt uitgelegd. Een aantal verantwoordelijkheden is echter lastig los te koppelen via Dependency Injection. Dit zijn bijvoorbeeld de non-functionele eisen in het traditionele functioneel ontwerp. Een dergelijke eis is dat iedere fout die optreedt gelogd moet worden.
De technische implementatie hiervan kan bestaan uit een try-catch-blok in iedere methode. Dit blok zal in het algemeen geen complexe set aan statements zijn, maar het is arbeidsintensief om deze statements aan iedere methode toe te voegen. Door het toepassen van AOP kunnen we de non-functionele eis minder arbeidsintensief implementeren.

Wat is Aspect Oriented Programming?

Met Aspect Oriented Programming is het mogelijk systeembrede verantwoordelijkheden, ook wel crosscutting concerns, te beheren. Dit betekent enerzijds dat je systeembrede verantwoordelijkheden kunt scheiden van de code met businesswaarde, anderzijds biedt AOP de mogelijkheid een gescheiden verantwoordelijkheid, ook wel aspect genoemd, op de juiste plaatsen te 'mengen' met de code. Dit proces wordt ook wel 'weaving' genoemd.
In de praktijk betekent het dat er door AOP minder regels code nodig zijn voor de non-functionele eisen. Denk maar eens aan aspecten als logging, caching, security, validatie en transactiemanagement. Dit zijn vaak eisen die in een functioneel ontwerp in slechts één pagina worden beschreven. Echter, het is zonder AOP bijna niet in te schatten hoeveel tijd de implementatie van een non-functionele eis in beslag neemt. Maar als een aspect los te koppelen is van de core-code, dan kunnen we een betere inschatting doen. Daarnaast dient een aspect declaratief toegepast te kunnen worden. Het resultaat: minder code, minder afhankelijkheden en betere inschattingen.

Hoe werkt AOP?

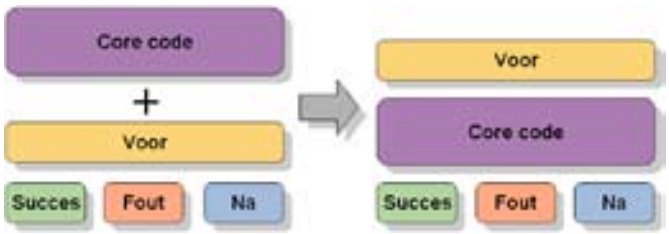
Een AOP-tool biedt meestal een variëteit aan punten waarop een ontwikkelaar kan inhaken voor het uitvoeren van code. Standaard worden mogelijkheden geboden om voor en na de uitvoer van de core-code in te haken. Maar ook het inhaken op een specifiek succesvol of foutief einde, behoort tot de mogelijkheden. Een aspect bestaat dan ook uit één of meer blokken code die voor of na de core-code worden uitgevoerd. De AOP-tool zal de core-code en de aspecten of runtime of compile-time 'mengen'; zie afbeelding 1.

Join points

De join points zijn de punten waarop het aspect en de core-code samen komen. Het is volledig afhankelijk van de gekozen AOP-oplossing hoe deze join points worden ingesteld. Er zijn oplossingen waarbij in een configuratie-file de join points worden beschreven die tijdens runtime worden uitgelezen. Dit kan erg krachtig zijn wanneer in productie een wijziging in de join points dient te worden gemaakt. Ander oplossingen hebben een eigen 'taal' waarin de join points kunnen worden beschreven. Soms worden de join points beschreven in dezelfde taal waarin ook het aspect zelf wordt geschreven. De verschillende mogelijkheden die een AOP-oplossing biedt voor het beschrijven van de join points bepaalt mede welke AOP-oplossing het geschiktst is. Voor het beschrijven van de join points worden meestal uitgebreide mogelijkheden geboden door middel van wildcards of reguliere expressies.

AOP-oplossingen in .NET

Er zijn verschillende oplossingen beschikbaar die het Aspect Oriented Programming in .NET ondersteunen. Een aantal heeft specifiek als doel ondersteuning van AOP, zoals Aspect DNG, Loom.NET en PostSharp. Een aantal Inversion of Control-containers, zoals Castle Windsor en Spring.NET bieden naast Dependency Injection ook ondersteuning voor AOP.
In dit artikel worden twee voorbeeldaspecten uitgewerkt op basis van de technologie van PostSharp.



Afbeelding 1. Van core-code en aspect naar runtime-flow

```
1  public void CoreCodeMethod()
2  {
3     Console.Out.WriteLine("Aanroep Core Code.");
4  }
```

Codevoorbeeld 1. De core-code zonder aspect

PostSharp als AOP-oplossing

De AOP-oplossing van PostSharp bestaat grotendeels uit een compile-time-mechanisme. PostSharp genereert code met aanroepen naar de aspecten op basis van beschreven join points. Dit is goed te zien in de verschillen tussen codevoorbeeld 1 en codevoorbeeld 2. Beide codevoorbeelden zijn na compilatie opgevraagd met Reflector. Codevoorbeeld 1 is de code zonder dat er een aspect is toegepast. Na toepassing van een aspect wordt deze code aangevuld zoals te zien is in codevoorbeeld 2. De code die rond regel 14 staat (regel 3-13 en 15-36), is de gegenereerde code die noodzakelijk is voor de aanroep van het aspect. Ongeacht welk aspect er wordt toegepast, zal deze gegenereerde code grotendeels gelijk zijn. De join points worden compile-time bepaald. Hierdoor hoeft runtime slechts de aanroep naar het aspect te worden doorgespeeld. De overhead van deze code, zoals te zien in codevoorbeeld 2, is minimaal, omdat er voornamelijk aanroepen naar het aspect worden gedaan. Verder kun je de objecten na toepassing van aspecten zoals voorheen blijven gebruiken. Inversion of Control-containers bepalen in sommige gevallen pas runtime de join points, waarbij een object met toegepaste aspecten alleen via de container zijn te verkrijgen

```
1  public void CoreCodeMethod()
2  {
3     MethodExecutionEventArgs ~laosEventArgs~1;
4     try
5     {
6        ~laosEventArgs~1 = new MethodExecutionEventArgs(
7           methodof(SimpleTraceAttributeTest.CoreCodeMethod,
8           SimpleTraceAttributeTest),
9           this, null);
10       ~PostSharp~Laos~Implementation.SimpleTraceAttribute~7.
11          OnEntry(~laosEventArgs~1);
12       if (~laosEventArgs~1.FlowBehavior != FlowBehavior.Return)
13       {
14          Console.Out.WriteLine("Aanroep Core Code.");
15          ~PostSharp~Laos~Implementation.SimpleTraceAttribute~7.
16             OnSuccess(~laosEventArgs~1);
17       }
18    }
19    catch (Exception ~exception~0)
20    {
21       ~laosEventArgs~1.Exception = ~exception~0;
22       ~PostSharp~Laos~Implementation.SimpleTraceAttribute~7.
23          OnException(~laosEventArgs~1);
24       switch (~laosEventArgs~1.FlowBehavior)
25       {
26          case FlowBehavior.Continue:
27          case FlowBehavior.Return:
28             return;
29       }
30       throw;
31    }
32    finally
33    {
34       ~PostSharp~Laos~Implementation.SimpleTraceAttribute~7.
35          OnExit(~laosEventArgs~1);
36    }
37 }
```

Codevoorbeeld 2. De core-code na toepassing van een aspect

Aspect Soort	Bruikbaar voor	In te haken op
OnMethodBoundaryAspect	Tracing of transactiemanagement.	OnEntry, OnSucces, OnException, OnExit
OnExceptionAspect	Foutafhandeling	OnException
OnFieldAccessAspect	Validatie	OnGetValue, OnSetValue
OnMethodInvocationAspect	Caching	OnInvocation
ImplementMethodAspect	Implementatie van een abstracte method	OnExecution
CompositionAspect	Laat een object een interface implementeren door te delegeren	
CompoundAspect	Samenstellen van meerdere aspecten	

Tabel 1. Verschillende soorten aspecten ondersteund binnen PostSharp

en het aanroepen van de constructor niet langer mogelijk is vanuit code.

PostSharp laat de code van het aspect met rust tijdens de compilatie, de aspectcode wordt runtime aangeroepen. Deze aanroepen zijn gelijk aan de verschillende inhaakmomenten van het aspect en zijn te zien in de regels 11, 16, 23 en 35 van codevoorbeeld 2. Eventuele breakpoints die in het aspect aanwezig zijn gaan af, zodat ook debuggen van een aspect goed te doen is. Het is dan wel van belang dat het aspect is toegepast op een object, anders wordt het aspect niet aangeroepen.

Het build-proces heeft een extra stap gekregen, zoals te zien is in afbeelding 2, waarin PostSharp wordt aangeroepen. PostSharp past in deze stap de ruwe assembly aan, zodat de aspecten worden aangeroepen op de juiste join points.

Soorten aspecten

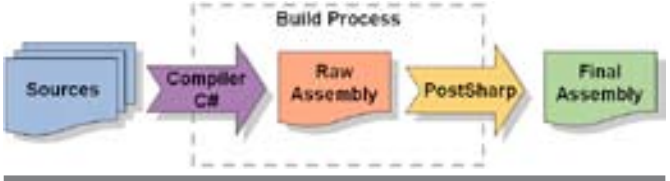
Er zijn veel verschillende soorten aspecten en elke soort heeft zijn eigen doel. De verschillende soorten die door PostSharp worden ondersteund, worden in tabel 1 opgesomd, waarbij is aangegeven waar elke soort voor is te gebruiken. Het is echter heel goed mogelijk dat een aspectsoort andere bruikbaarheden kent. Zo is het OnMethodBoundaryAspect ook goed te gebruiken voor validatie van properties, een property bestaat in Intermediate Language immers uit een 'set_' en een 'get_' method.

Join points met PostSharp

De implementatie van een aspect met PostSharp is in de basis een attribuut met extra eigenschappen. Het weaven van een aspect met de core-code gaat declaratief in C#. Zo is het bijvoorbeeld mogelijk het trace-aspect dicht bij een methode te plaatsen als hier specifiek voor getraced dient te worden. Het is echter ook mogelijk een aspect breder in te zetten door het attribuut als assembly-attribuut te plaatsen in bijvoorbeeld de AssemblyInfo. Eventuele filtering kan bijvoorbeeld door middel van de eigenschappen AttributeTargetMembers en AttributeTargetTypes, waarbij de wildcards en reguliere expressies worden ondersteund.

Nadelen van PostSharp

Je kunt de join points van een aspect in de AssemblyInfo plaatsen. Dit zorgt mogelijk voor een ondoorzichtige appli-



Afbeelding 2. Van source naar final assembly

```
1  [Serializable]
2  public class SimpleTraceAttribute : OnMethodBoundaryAspect
3  {
4    private string methodName;
5
6    public override void CompileTimeInitialize(MethodBase method)
7    {
8      methodName = method.DeclaringType.Name + "/" + method.Name;
9    }
10
11   public override void OnEntry(MethodExecutionEventArgs eventArgs)
12   {
13     Trace.WriteLine(string.Format("Entering - {0}.", methodName));
14     Trace.Indent();
15   }
16
17   public override void OnExit(MethodExecutionEventArgs eventArgs)
18   {
19     Trace.Unindent();
20     Trace.WriteLine(string.Format("Leaving - {0}.", methodName));
21   }
22
23   public override void OnException(MethodExecutionEventArgs eventArgs)
24   {
25     Trace.WriteLine(string.Format("Error - {0}",
       eventArgs.Exception.Message));
26   }
27 }
```

Codevoorbeeld 3. Tracing-aspect

catie, aangezien het aspect 'ver' van de originele code staat. Zorgvuldig te werk gaan, blijft belangrijk. Het eenvoudig kunnen toepassen van een aspect betekent niet dat een aspect ondoordacht moet worden toegepast. Verder is in PostSharp de ondersteuning van ASP.NET op het moment van schrijven alleen nog experimenteel. Dit komt doordat de compilatie van ASP.NET niet via MSBuild loopt. Dit in tegenstelling tot de compilatie van de andere soorten .NET-assemblies. Runtime hebben de aspecten twee libraries nodig, namelijk PostSharp.Public en PostSharp.Laos. Ondanks het feit dat deze libraries binnen LGPL worden uitgegeven, kan open-source voor klanten een drempel zijn in het gebruik van PostSharp.

Testen van een aspect

Het testen van een aspect vergt extra aandacht. Om onnodige foutsituaties te voorkomen, is het verstandig unit-tests te schrijven voor een aspect met honderd procent codecoverage. Gebruik voor het unit-testen mock-objecten waar je een aspect op toepast. Controleer dan via unit-tests of het gedrag dat ontstaat het verwachte gedrag is. Ondanks een wijziging van de dll is het tijdens compilatie mogelijk zowel in de originele code als in het aspect te debuggen. Dit is niet alleen belangrijk voor het tackelen van mogelijke problemen, maar op deze manier krijg je ook een beter gevoel over de werking van een specifiek aspect.

Voorbeeldaspect: Tracing

Het eerste voorbeeldaspect dat in dit artikel aan de orde komt, is een trace-aspect. Dit aspect bevat simpele compile-time initialisatie, zodat er runtime minder performanceverlies zal zijn. De compile-time initialisatieregels 6-9 van codevoorbeeld 3 plaatsen de naam van het object en de methode in een string voor runtime-gebruik. Dit laatste biedt performancewinst ten opzichte van het runtime opvragen van de namen. Dit aspect overerft de OnMethodBoundaryAspect, zodat het mogelijk is runtime in te haken op de 'OnEntry', 'OnExit' en 'OnException' van een methode. Verder is het

```
1  //Tracing voor alle methoden in alle objecten
2  [assembly: SimpleTrace]
3  //Tracing voor alle methoden in OrderService
4  [SimpleTrace]
5  public class OrderService
6  { }
7  //Tracing voor alle methoden in de objecten welke het woord
   'Service' bevatten
8  [assembly: SimpleTrace(AttributeTargetTypes = "*Service*")]
```

Codevoorbeeld 4. Tracing join points

ook mogelijk in te haken op de 'OnSucces' van een methode, wat in dit voorbeeld niet wordt gebruikt.

Nu is het goed mogelijk dit aspect op letterlijk iedere methode in de core-code toe te passen. Regel 2 van codevoorbeeld 4 beschrijft het join point voor iedere methode in alle objecten in de desbetreffende assembly. Maar het is ook mogelijk heel specifiek voor een object te tracen, zoals in regel 4 van codevoorbeeld 4. Algemener kan ook met behulp van wildcards: regel 8 beschrijft dat ieder object met 'Service' in de naam getraced moet worden.

Voorbeeldaspect: Validatie

In het voorbeeldaspect van tracing zal in veel gevallen een zeer algemeen join point worden gebruikt. Bij validatie wordt in de meeste gevallen een specifiek veld gecontroleerd. Hierbij is het zaak de validatie dichtbij het veld te plaatsen, zodat het join point in de code op de juiste plaats wordt gedocumenteerd. Een validatieaspect join point is dan ook het beste te plaatsen zoals in codevoorbeeld 6 regel 3 te zien is, vóór de property. Het validatieaspect dat is uitgewerkt, is een controle dat een veld geen null-waarde krijgt. Deze controle wordt uitgevoerd tijdens de 'OnEntry' door het eerste argument met 'null' te vergelijken. Deze controle dient alleen uitgevoerd te worden op de set van een property en niet op de get van een property, vandaar dat dit compile-time wordt gecontroleerd, zoals te zien is in regel 6 van codevoorbeeld 5. PostSharp genereert in dit geval alleen code voor de aanroep van het aspect binnen de set van een property. Dit is slechts één validatieaspect, maar het is natuurlijk mogelijk

```
1  [Serializable]
2  public class NotNullValidation : OnMethodBoundaryAspect
3  {
4    public override bool CompileTimeValidate(MethodBase method)
5    {
6      return method.Name.StartsWith("set_");
7    }
8
9    public override void OnEntry(MethodExecutionEventArgs eventArgs)
10   {
11     if (eventArgs.GetArguments()[0] == null)
12       throw new Exception(string.Format("Value should not be
         null."));
13     base.OnEntry(eventArgs);
14   }
15 }
```

Codevoorbeeld 5. Not null-validatieaspect

```
1  public class Customer
2  {
3    [NotNullValidation]
4    public string Name { get; set; }
5  }
```

Codevoorbeeld 6. Validatie join point

een hele bibliotheek aan te leggen van validaties zoals reguliere expressievalidaties.

Losgekoppeld

Door gebruik te maken van Aspect Oriented Programming kunnen we systeembrede verantwoordelijkheden loskoppelen van de core-code. AOP kan klein beginnen door bijvoorbeeld tracing in te zetten, maar groot eindigen door het beheren van transacties volledig aan aspecten over te laten. Dit artikel is niet uitputtend, maar is een eerste aanzet in Aspect Oriented Programming. Probeer het eens uit en laat me weten wat de ervaringen zijn.

Mark Monster is Software Engineer bij Rubicon (www.rubicongroup.biz). Hij houdt zich bezig met Custom Development in .NET. Voor vragen en opmerkingen is hij te bereiken via m.monster@rubicongroup.biz of via zijn blog op mark.mymonster.nl.

Referenties

.NET Magazine #18, Artikel Mark Monster over Dependency Injection: www.microsoft.nl/netmagazine18
PostSharp: www.postsharp.org
Castle Windsor: www.castleproject.org/container
Aspect DNG: aspectdng.tigris.org/nonav/doc/index.html
Loom.NET: www.dcl.hpi.uni-potsdam.de/research/loom
Spring.NET: springframework.net
Reflector: www.aisto.com/roeder/dotnet