Aspect georiënteerd programmeren met Unity

De oplossing van technische crosscutting concerns

Jonne Kats

Mede dankzij de opmars van Test Driven Development wordt steeds meer gebruik gemaakt van Dependency Injection containers. Eén voordeel hiervan wordt echter vaak over het hoofd gezien, namelijk de AOP (Aspect Oriented Programming) mogelijkheden. Eindelijk een oplossing voor die slecht onderhoudbare regels logging- en foutafhandeling-code!

AOP staat voor Aspectgeoriënteerd programmeren en is bedoeld om crosscutting concerns op een eenduidige manier op te lossen. Crosscutting concerns zijn handelingen die door het hele systeem moeten worden uitgevoerd en waarvoor een object-georiënteerde aanpak geen efficiënte oplossing biedt. Bekende voorbeelden hiervan zijn logging, security en foutafhandeling. Deze code komt voor in alle lagen en componenten van het systeem en zorgt vaak voor ruis, waardoor de code minder leesbaar wordt. Daarnaast moet deze code, wanneer er iets wijzigt in het betreffende concern, overal worden aangepast.

Met AOP is het mogelijk om de code voor het betreffende crosscutting concern op één locatie te definiëren, waarna deze automatisch op de verschillende locaties wordt uitgevoerd. Niet alleen wordt de code hierdoor beter onderhoudbaar, het voorkomt ook een hoop repeterend werk.
De definitie van de uit te voeren code heet in AOP termen een advice en de locaties waar de code uitgevoerd moet worden joinpoints. De combinatie van een advice en bijbehorende joinpoints vormt samen een aspect. Waarschijnlijk is de meest bekende vorm van AOP het inweven van de uit te voeren code tijdens het compileren van programmacode, dit noemt men statisch weven. Een bekende oplossing voor .NET waarmee dit mogelijk is, is PostSharp. Met PostSharp kan met een attribuut worden aangegeven waar de code moet worden ingeweven. Het inweven zelf gebeurt echter na het compileren, waarbij de Intermediate Language code wordt aangepast. De andere vorm van AOP is het inweven van de code tijdens het uitvoeren van de code, het zogenaamde dynamisch weven. Deze vorm van AOP is mogelijk met de meeste gangbare Dependency Injection (DI) containers, zoals Unity, een DI container van Microsoft Patterns & Practices.

AOP met een DI container

Door gebruik te maken van een DI container, kunnen componenten met een afhankelijkheid van elkaar worden losgekoppeld. Dit brengt een aantal voordelen met zich mee, waaronder een betere testbaarheid van de verschillende onafhankelijke units. Een DI container werkt over het algemeen als volgt:

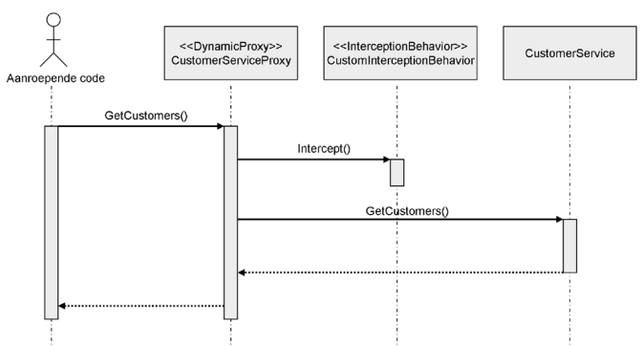
- In plaats van een concrete afhankelijkheid van een component op een ander component, wordt er een interface geïntroduceerd.
- Op een centrale locatie wordt geregistreerd welke implementatie voor de betreffende interface moet worden gebruikt (zie codevoorbeeld 1).

```
Container.Register<ICustomerService, CustomerServiceImpl>();
```

CODEVOORBEELD 1: REGISTEREN VAN EEN AFHANKELIJKHEID BIJ DE CONTAINER.

- Componenten met een afhankelijkheid op de ICustomerService interface, communiceren vervolgens met een instantie van de CustomerServiceImpl.

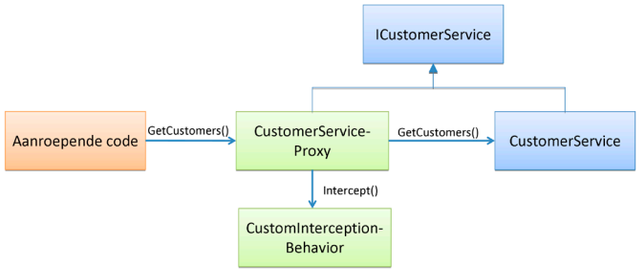
Doordat de componenten door de DI container worden geïnstantieerd, is dit de ideale locatie om dynamisch weven toe te passen. In plaats van het rechtstreeks instantiëren van het betreffende component, kan deze namelijk worden vervangen door een dynamische proxy. Dankzij deze proxy kunnen de verschillende berichten (of methode aanroepen) worden onderschept en kan



FIGUUR 1: HET INTERCEPTION PROCES IN EEN SEQUENCE DIAGRAM.

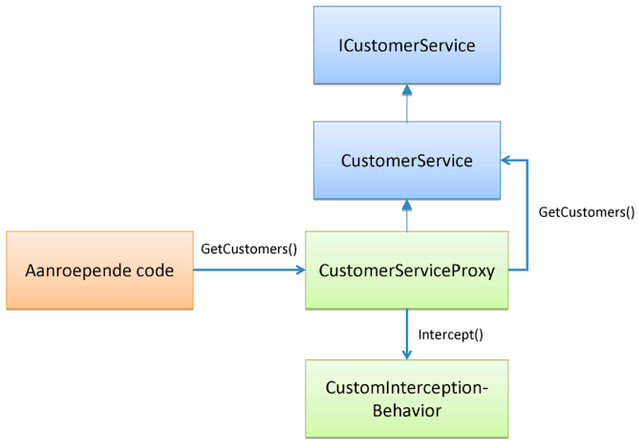
er additionele code worden uitgevoerd. Het onderscheppen van berichten heet in Unity termen Interception en een advice een Interception Behavior.

Unity kent twee soorten interception, namelijk instance interception en type interception. Door middel van instance interception wordt er dynamisch een proxy aangemaakt, welke alle calls doorgeeft aan de daadwerkelijke instantie (zie figuur) . In het voorbeeld heeft de aanroepende code een afhankelijkheid op de ICustomerService interface, welke door de CustomerService klasse wordt geïmplementeerd. Unity maakt echter dynamisch een proxy aan en geeft deze aan de aanroepende code, in plaats van de CustomerService. De aanroepende code communiceert hierdoor met de proxy alsof het de daadwerkelijke instantie zelf is. De proxy voert eventuele aanvullende code uit en roept vervolgens de daadwerkelijke instantie aan, alsof er niets anders is gebeurd.



FIGUUR 2: SCHEMATISCHE WEERGAVE INSTANCE INTERCEPTION.

Type interception maakt in tegenstelling tot instance interception geen gebruik van een tussenliggende proxy, maar een proxy welke van het doelobject overerft. Er wordt dynamisch een afgeleide class van het doelobject gemaakt. Deze onderschept alle berichten, voert eventuele additionele code uit en geeft vervolgens de berichten door aan de base class. In het voorbeeld wordt er door Unity een dynamische proxy gemaakt, welke is afgeleid van de CustomerService. Deze proxy overschrijft de GetCustomers methode, voert additionele functionaliteit uit en roept uiteindelijk de GetCustomers methode op de CustomersService zelf aan. Het grote voordeel hiervan is de winst in performance ten opzichte van instance interception. Het nadeel is dat de te onderscheppen berichten virtual moeten zijn, omdat de proxy deze van de base klasse moet kunnen overschrijven.



FIGUUR 3: SCHEMATISCHE WEERGAVE TYPE INTERCEPTION.

Instance interception en type interception worden in Unity geïmplementeerd door middel van een drietal interceptors:

- **Interface interceptor**

Een instance interceptor die alleen met interfaces werkt. Hierbij kan er slechts één interface worden geïmplementeerd en de gegenereerde proxy kan niet worden gecast naar het doelobject (Dit komt niet vaak voor, maar het zou een showstopper kunnen zijn).

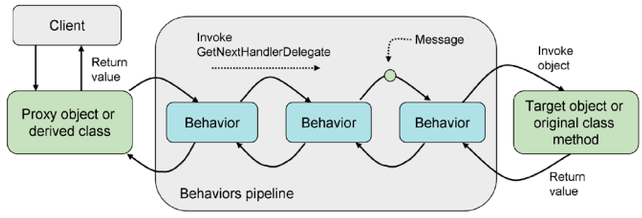
- **Transparent proxy interceptor**

Een instance interceptor die gebruik maakt van onderdelen van het .NET Remoting framework. Dit heeft als gevolg dat de doelobjecten van de MarshalByRefObject base klasse moeten afleiden om hier gebruik van te kunnen maken. Voordeel hiervan is dat deze interceptor het breedst toepasbaar is. Het nadeel is dat deze het traagst is.

- **Virtual method interceptor**

Dit is een type interceptor en kan zowel interfaces als klassen implementeren. De methodes in de klasse moeten echter wel virtual zijn. Doordat het een type interceptor is , kan deze niet worden toegepast op reeds geïnstantieerde objecten. De doelobjecten moeten dus door de Unity container zijn geïnstantieerd, dit is voor de andere twee interceptors geen vereiste.

De keuze van het type interceptor hangt af van een aantal factoren. Bij een behavior voor logging kan de vraag worden gesteld of het volstaat om alleen de methodeaanroepen naar interfaces te loggen. Hierover later in dit artikel meer. Het type interceptor kan eenvoudig worden geconfigureerd en heeft geen gevolgen voor de implementatiedetails van het aspect. Zoals eerdergenoemd wordt een advice in Unity gedefinieerd in een interception behavior. Om het gebruik van meerdere interception behaviors te faciliteren, kent Unity de zogenaamde behaviors pipeline.



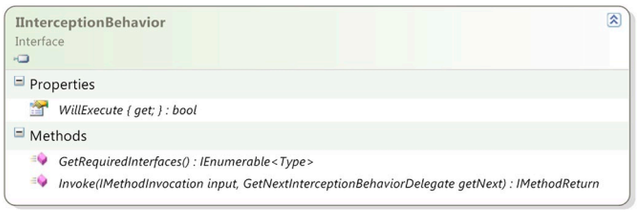
FIGUUR 4: DE UNITY BEHAVIORS PIPELINE. (BRON: UNITY DOCUMENTATIE).

Deze behaviors pipeline zorgt ervoor dat verschillende behaviors hun additionele functionaliteit zowel voor- als achteraf de methodeaanroep kunnen uitvoeren. De proxy start de pipeline wanneer een bericht wordt onderschept, de pipeline roept vervolgens één voor één de behaviors aan en uiteindelijk de methode op het doelobject.

Het maken van een interception behavior

Een interception behavior moet worden afgeleid van de IInterceptionBehavior interface (zie figuur 5).

In de Invoke methode gebeurt al het werk. Deze is onderdeel van de behavior pipeline en wordt aangeroepen zodra een bericht wordt onderschept. Deze methode heeft als input twee parameters, de eerste van het type IMethodInvocation en de tweede van het type GetNextInterceptionBehaviorDelegate. In de IMethodInvocation parameter zit algemene informatie over het bericht dat wordt onderschept, zoals de naam van de property of metho-



FIGUUR 5: KLASSEDIAGRAM VAN DE IINTERCEPTIONBEHAVIOR INTERFACE.

de en de waarden van de argumenten. Met de GetNextInterceptionBehaviorDelegate delegate wordt de volgende behavior in de pipeline aangeroepen, waardoor uiteindelijk ook het doelobject zal worden aangeroepen. Het resultaat van de GetNextInterceptionBehaviorDelegate bevat het resultaat van het daadwerkelijke bericht en kan worden gebruikt voor bijvoorbeeld foutafhandeling. De WillExecute property van de IInterceptionBehavior interface wordt gebruikt voor optimalisatie. Als tijdens het uitvoeren van de code blijkt dat de behavior om een bepaalde reden niet hoeft worden uitgevoerd, dan zal Unity ook geen proxy maken. De GetRequiredInterfaces property wordt gebruikt als de behavior alleen maar toepasbaar is op objecten die een bepaalde interface implementeren. Bijvoorbeeld een behavior die berichten onderschept op doelobjecten die INotifyPropertyChanged implementeren en automatisch het PropertyChanged event afvuurt.
In codevoorbeeld 2 staat een voorbeeld voor een interception behavior voor het loggen van berichten.

```
    public class LoggingInterceptor : IInterceptionBehavior
    {
        public System.Collections.Generic.IEnumerable<Type>
GetRequiredInterfaces()
        {
            return Type.EmptyTypes;
        }

        public IMethodReturn Invoke(IMethodInvocation input,
GetNextInterceptionBehaviorDelegate getNext)
        {
            string logMessage = FormatLogMessage(input);

            Console.WriteLine(logMessage);


IMethodReturn methodReturn = getNext()(input, getNext); //
Volgende in de behavior pipeline

            Console.WriteLine("Einde method, return waarde:
{0}", methodReturn.ReturnValue);

            return methodReturn;
        }

        private static string FormatLogMessage(IMethodInvocation
 input)
        {
            var sb = new StringBuilder();

            string methodName = input.MethodBase.Name;


sb.AppendLine(string.Format("Method {0} called, with parameters: ",
 methodName));


for (int parameterIndex = 0; parameterIndex < input.Arguments.
Count; parameterIndex++)
            {
                string parameterName = input.Arguments.
ParameterName(parameterIndex);

string parameterValue = input.Arguments[parameterIndex].ToString();
                sb.AppendLine(string.Format("Name: {0}, value: {1}",
parameterName, parameterValue));
            }

            return sb.ToString();
        }

        public bool WillExecute
        {
            get { return true; }
        }
    }
```

CODEVOORBEELD 2: EEN INTERCEPTION BEHAVIOR VOOR HET LOGGEN VAN BERICHTEN.

De logging behavior is niet afhankelijk van een bepaalde interface en dus geeft de GetRequiredInterfaces property een lege lijst terug. De behavior wordt altijd uitgevoerd waardoor de WIllExecute property true terug geeft. Voordat de volgende behavior wordt aangeroepen, wordt eerst de naam van de methode en de gebruikte parameters naar de console gelogd. Vervolgens wordt door middel van de getNext() delegate de eventuele volgende behavior in de pipeline aangeroepen en uiteindelijk de methode op het doelobject zelf. Na deze aanroep wordt het resultaat van de methode op het doelobject gelogd.

AOP leent zich vooral voor het oplossen van technische crosscutting concerns.

Het advice gedeelte van het aspect is nu gedefinieerd, maar hoe zit het met het joinpoints? Oftwel, hoe wordt de interception behavior aan locaties in de code gekoppeld?

Configureren van interception

De joinpoints kunnen met Unity op verschillende manieren worden gedefinieerd. In de meest simpele vorm wordt dit per afhankelijkheid gedaan tijdens het registeren in de container. Per afhankelijkheid wordt dan het type interceptor en de gewenste interception behavior aangegeven. Stel dat alle methodeaanroepen op de IOrderService uit codevoorbeeld 3 moeten worden gelogd.

```
    public interface IOrderService
    {
        object GetOrder(string orderNumber);
    }

    public class OrderService : IOrderService
    {
        public object GetOrder(string orderNumber)
        {
            return new object();
        }
    }
```

CODEVOORBEELD 3: DE ORDERSERVICE.

Allereerst moet de Unity interception extension worden geregistreerd bij de Unity container. Unity kent een extension mechanisme waarmee gemakkelijk uitbreidingen kunnen worden gemaakt, het interception mechanisme is als een dergelijke extension geïm-

plementeerd. Na het registreren van deze extension, wordt de orderservice geregistreerd. Hierbij wordt het type interceptor en de toe te passen behavior aangegeven. In dit geval de interface interceptor en de logging behavior, zie codevoorbeeld 4.

```
var container = new UnityContainer();
            container.AddNewExtension<Interception>();

            container.RegisterType<IOrderService, OrderService>
(new Interceptor<InterfaceInterceptor>(),
                                                new InterceptionB
ehavior<LoggingInterceptor>());

            var orderService = container.Resolve<IOrderService>();

            orderService.GetOrder("34");
```

CODEVOORBEELD 4: CONFIGUREREN VAN UNITY INTERCEPTION.

Omdat alleen methodeaanroepen op de IOrderService interface moeten worden gelogd, volstaat de InterfaceInterceptor. Op dit moment zal bij het aanroepen van een methode op de orderservice de logging behavior worden gebruikt en zal de aanroep worden gelogd. Het lijkt omslachtig om voor elk component waarin iets gelogd moet worden, de specifieke interceptor en behavior te definiëren. Helaas biedt Unity hiervoor (nog) geen mooi mechanisme, wel is er integratie met het Policy Injection Application Block (PIAB) van Patterns & Practises. Hiermee kunnen door middel van matchregels automatisch behaviors (of callhandlers in PIAB termen) aan componenten worden gekoppeld. Zo is er bijvoorbeeld een matchregel die matched aan de hand van de namespace.
Om een behavior in combinatie met PIAB te kunnen gebruiken moet de behavior worden afgeleid van ICallHandler in plaats van IInterceptionBehavior. Dit is omdat de PIAB integratie zelf als interception behavior is geïmplementeerd. De ICallHandler interface definieert geen WillExecute en GetRequiredInterfaces properties, de Invoke method is hetzelfde. Als voor bijvoorbeeld alle componenten in een bepaalde assembly de logging behavior moet worden toegepast, dan kan de policy worden gedefinieerd zoals in codevoorbeeld 5:

```
            container.Configure<Interception>()
                .AddPolicy("MyPolicy")
                .AddMatchingRule(new NamespaceMatchingRule
("InterceptionDemo"))
                .AddCallHandler<LoggingCallHandler>();
```

CODEVOORBEELD 5: CONFIGUREREN VAN INTEGRATIE MET HET POLICY INJECTION APPLICATION BLOCK.

De integratie met het Policy Injection Application Block is als Interception Behavior geïmplementeerd, dit betekent dat het type interceptor alsnog per component moet worden geregistreerd bij de Unity container. De reden dat het registreren van een interceptor type altijd per component gebeurt is omdat geen enkel interceptor type voor alle soorten componenten geschikt is, vandaar dat dit per component zou moet worden afgewogen. Toch zijn er scenario's te bedenken waarbij dit wel mogelijk zou zijn. Indien veel gebruik wordt gemaakt van interfaces zou het de keuze kunnen zijn om alleen alle methodeaanroepen op interface niveau te loggen. Gelukkig biedt Unity genoeg uitbreidingsmogelijkheden en met een eigen Unity extension is dit alsnog eenvoudig te realiseren, zie codevoorbeeld 6.

```
    public class ConfigureLoggingInterceptionExtension :
UnityContainerExtension
    {
        protected override void Initialize()
        {
            Context.Registering += (sender, e) =>
ConfigureLoggingInterceptor(e.TypeFrom);
            Context.RegisteringInstance += (sender, e) =>
ConfigureLoggingInterceptor(e.RegisteredType);
        }

        private void ConfigureLoggingInterceptor(Type
typeToIntercept)
        {
            if (typeToIntercept != null && typeToIntercept.
IsInterface)
            {
                if (typeToIntercept.Namespace ==
"InterceptionDemo")
                {
                    Container.Configure<Interception>()
                        .SetInterceptorFor(typeToIntercept,
new InterfaceInterceptor());
                }
            }
        }
    }
```

CODEVOORBEELD 6: EEN UNITY EXTENSIONS VOOR AUTOMATISCH CONFIGUREREN VAN HET TYPE INTERCEPTOR.

Met deze extension worden alle componenten die een interface implementeren en zich in de InterceptionDemo namespace bevinden, automatisch gekoppeld aan een InterfaceInterceptor.

Conclusie

Aspectgeoriënteerd programmeren is een krachtige techniek en een waardevolle toevoeging aan de gereedschapskist van de ontwikkelaar. Omdat dependency injection containers steeds vaker gebruikt worden en de meeste hiervan AOP faciliteiten aanbieden, is de stap snel gemaakt. Zeker in vergelijking met het toepassen van statisch weven, waarvoor het compileerproces van een applicatie moet worden uitgebreid. Wel zal statisch weven altijd een betere performance hebben, dit is echter te verwaarlozen wanneer het systeem in ontwikkeling een Line of Business applicatie betreft met een gemiddeld aantal transacties. Een algemeen mogelijk nadeel van AOP is dat de code soms moeilijk te doorgronden is, omdat er tijdens het uitvoeren iets anders gebeurt dan in de code staat beschreven. Dit is zeker niet wenselijk wat betreft domein logica, AOP leent zich dan ook vooral voor het oplossen van technische crosscutting concerns.

Referenties

- Dependency Injection: http://nl.wikipedia.org/wiki/Dependency_injection
- Aspectgeoriënteerd programmeren: http://nl.wikipedia.org/wiki/Aspectgeori%C3%ABnteerd_programmeren
- Unity: http://unity.codeplex.com/
- PostSharp: http://www.sharpcrafters.com/postsharp

Jonne Kats, is .NET consultant bij Aviva Solutions. Jonne is te bereiken via jonne.kats@avivasolutions.nl