Modulaire Silverlight applicaties bouwen

In combinatie met MEF en PRISM

Ralph Jansen, Arjan Hordijk en Arne Wauters

Zowel PRISM als het Managed Extensibility Framework (MEF) kunnen helpen bij het bouwen van modulaire applicaties. Alvorens daar dieper op in te gaan, eerst wat meer informatie bij de term zelf. Modulaire applicaties zijn applicaties die niet als één groot monolithisch blok worden ontwikkeld. Het zijn applicaties die zijn samengesteld uit onafhankelijke en uitwisselbare modules.

Deze modules zijn autonoom en kunnen los van elkaar en gelijktijdig worden ontwikkeld. Het kader waarin deze modules draaien, meestal de Shell genoemd, voorziet in de infrastructuur die er voor zorgt dat de modules samenkomen als één geïntegreerd geheel.
Grote line-of-business (LOB) applicaties zijn vaak gebaat bij een modulaire aanpak. Aangezien modules los van elkaar kunnen evolueren, verwijderd of toegevoegd worden, is het gemakkelijker om te ontwikkelen, testen, deployen en uit te breiden. Daarboven komt dat de Shell ook infrastructuur kan bevatten voor cross-cutting concerns zoals logging, auditing, etc. Door het delen van deze abstracte implementaties kunnen modules snel en uniform worden ontwikkeld.

Situering van PRISM

PRISM werd ontwikkeld door Microsoft Patterns and Practices en was vroeger bekend als 'The Composite Application Guidance for WPF and Silverlight'. Een framework en set van klassen die de ontwikkelaar helpt om modulaire applicaties uit te bouwen voor WPF, Silverlight en Windows Phone 7.
PRISM voorziet in een uitgebreid Application Model waarmee je je eigen modulaire applicatie kan bouwen. Door gebruik te maken van verschillende Design Patterns zoals Event Aggregation, Service Location en Inversion of Control (IoC) laat het je toe om losgekoppelde componenten te introduceren die toch met elkaar kunnen communiceren.

Hier stopt het echter niet. Naast het modulaire Framework ondersteunt PRISM ook:

- het ontdekken en registeren van modules;
- het indelen van de user interface;
- navigatie;
- het scheiden van business logica en presentatie aan de hand van het Model View ViewModel (MVVM) Design Pattern.

PRISM is een verzameling van Design Patterns en Best Practices om hedendaagse grote client applicaties mee te creëren.

Situering van MEF

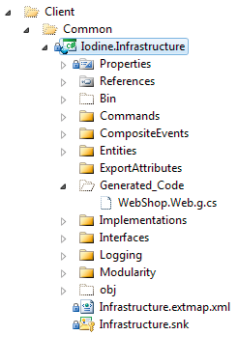
Hoewel het Managed Extensibility Framework kan worden gebruikt voor Dependency Injection, is het geen traditionele IoC container. Waar met een traditionele container meestal een Reference wordt gelegd naar een object van een op voorhand bekend type, richt MEF zich meer op de mogelijkheid om aan de hand van een sleutel of contract één of meerdere References te verkrijgen naar types die op voorhand nog niet bekend zijn.
Zo kan MEF in runtime nieuwe componenten ontdekken en automatisch lijsten die alle geregistreerde elementen met een bepaald contract bevatten vernieuwen! Voor Silverlight is dit concept nog iets concreter uitgewerkt met de ingebouwde functionaliteit om een XAP in runtime asynchroon te downloaden en vervolgens de nieuw ontdekte onderdelen in de container te laden.
Met een krachtig kant-en-klaar modulair framework dat expliciet gebruik maakt van de vele voordelen die MEF biedt boven een traditionele IoC container, is het bouwen van een responsieve, dynamische en onderhoudbare client een stuk eenvoudiger geworden. Om deze stelling kracht bij te zetten volgt hieronder een hands-on approach voor het creëren van een modulaire Silverlight applicatie.

Creatie met MEF en PRISM

In april 2010 is de nieuwste versie van Silverlight gelanceerd. Microsoft bestempelde versie 4 als de eerste echte versie van het Framework dat geschikt is voor het bouwen van Business Applications. Grote Business Applications zijn vaak in stukken verdeeld om het product in modules te verkopen. De testbaarheid van grote applicaties is vaak een probleem. Door de applicatie modulair op te bouwen wordt de testbaarheid en betrouwbaarheid van het product vergroot.
De volgende hoofdstukken beschrijven in hoofdlijnen hoe een Silverlight applicatie modulair kan worden opgebouwd. Om niet meteen in het diepe te springen wordt er eerst ingegaan in de opzet van de beschreven voorbeeld applicatie.
De applicatie bestaat uit een Client en Server gedeelte, en is opge-

zet met behulp van een Silverlight Business Application Template. Deze template maakt het WebShop Client en het WebShop.Web Server project. De overige projecten in de Client zijn aangemaakt met de Silverlight Application Template van Visual Studio.
De Server is verantwoordelijk voor het hosten van een aantal Rich Internet Application (RIA) Services gekoppeld aan een Entity FrameWork (EF) 4.0 datamodel en de Silverlight Client applicatie.
De Services voorzien de Client van data en toegang tot applicatie logica.

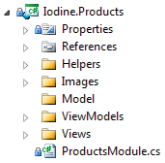
De Client is opgedeeld in drie onderdelen:
- Common;
- Modules;
- De Silverlight Client applicatie (WebShop).



Het Common gedeelte bevat alles dat beschikbaar moet zijn voor de Modules en de Client applicatie. Zo worden hier bijvoorbeeld style elementen en Base Classes gedefinieerd die door meerdere Modules worden gebruikt.
Het Infrastructure project heeft ook de rol om het Proxy EF4.0 datamodel beschikbaar te stellen aan de Modules.
De Modules kunnen door de Common Infrastructure de Server datalaag via de RIA Services aanspreken. Het proxy model, Webshop.Web.g.cs, wordt automatisch gegenereerd. In de afbeelding is dat zichtbaar in de folder Generated_Code.
De Modules, de CompositeEvents en Commands worden uitgelegd in "Communiceren tussen modules" en in "Communiceren binnen de module".
De Silverlight Client, WebShop, is het hart van de applicatie. Hier start de applicatie en de Bootstrapper. De Bootstrapper wordt geïnstantieerd door de applicatie en is verantwoordelijk voor de volgende onderdelen:
- initialiseren van de applicatie;
- tonen van de Shell (de hoofdpagina van de applicatie);
- opbouwen van de Module Catalog;
- laden van Modules.

De opbouw van de applicatie heeft als doelstelling dat de losse modules snel worden geladen. De eindgebruiker moet niet worden lastiggevallen met lange downloadtijden. Onze applicatie is een webshop; hierin staat performance voorop, anders verlies je de aandacht. De techniek die zorgt voor het downloaden van de modules staat beschreven in "Module Downloading".
Losgekoppelde module

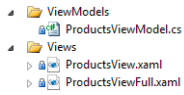
Zoals hierboven in de architectuur is beschreven, bestaat de applicatie uit meerdere modules. Een module heeft een specifieke taak en of functionaliteit binnen de applicatie. In deze applicatie zijn dat de functionaliteiten 'producten tonen' en 'winkelwagen'. Door de functionaliteit van een module af te bakenen kan elke module apart leven. Om een module op het scherm te krijgen dient deze te worden geregistreerd. In het voorbeeld hieronder gebeurt dat in de ProductsModule Class.



Elke module heeft zijn eigen PRISM Region als container. Een module kan via de RegionManager op de betreffende Region worden geplaatst.

```
this._regionManager.RegisterViewWithRegion("contentProducts",
typeof(Iodine.Products.Views.ProductsView));
```

In het voorbeeld wordt de ProductsView geplaatst in de Region 'contentProducts'. Elke view heeft weer zijn eigen logica. Dit zal in het volgende hoofdstuk worden uitgelegd.



Communiceren binnen de module gebeurt op een andere manier dan het communiceren tussen modules. Het voorbeeld project is opgezet door middel van het Model View ViewModel (MVVM) Design Pattern. Dit Pattern is bedoeld voor het loskoppelen van de Business logica en de presentatielaag. De View of Views zijn er alleen om de informatie weer te geven. Het ViewModel zorgt voor alle Business logica. Het voordeel hiervan is dat meerdere soorten Views aan dezelfde ViewModel kunnen worden gekoppeld. In het voorbeeld van de lijst met producten kan er meerdere Views worden gebruikt. Een lijst van producten kan op verschillende manieren worden weergeven. Bijvoorbeeld door middel van alleen miniaturen te gebruiken, lijst van producten in tabel lay-out of een lijst van producten met plaatje er naast.
Het doorgeven van de informatie tussen de View en het ViewModel gebeurt door middel van Commands. Commands implementeren de interface ICommand. In de applicatie wordt hiervoor het AddToCartCommand voor gebruikt. Dit Command verwacht een product als parameter om deze vervolgens door te geven. In de View wordt de

```
public class RelayCommand<T> : ICommand
        where T : Entity

public class AddToCartCommand : RelayCommand<Product>
```

Command gebruikt op een knop en wordt het betreffende product meegegeven als parameter.

```
<Button Command="{Binding DataContext.AddToCartCommand}"
CommandParameter="{Binding}" />
```

In het ViewModel is er een public property die een methode aanroept met als parameter het product dat is doorgegeven uit de View.

```
        public ICommand AddToCartCommand
        {
            get
            {
                if ( _addToCartCommand == null)
                    _addToCartCommand = new AddToCartCommand(OnA
ddToCartCommand);

                return _addToCartCommand;
            }
        }
```

```
void OnAddToCartCommand(Product product)
{
    _eventAggregator.GetEvent<AddToCartEvent>().Publish(product);
}
```

Koppelen View en ViewModel door middel van MEF, zorgt er voor dat er zo min mogelijk code is en niets hard is gelinkt. Het enige wat nodig is, is het exporteren van een ViewModel en het importeren van een

```
[Export(ExportNames.ProductsViewVM)]
public class ProductsViewModel : ViewModelBase
```

ViewModel in de View. Het ViewModel wordt meteen gekoppeld aan de DataContext van de View, zodat alle public properties in de ViewModel direct kunnen worden aangesproken. In de Code Behind van de View staat naast deze koppeling geen andere code. De View wordt geladen door de RegionManager en de geladen View laadt vervolgens automatisch zijn bijbehorende ViewModel.

```
[Import(ExportNames.ProductsViewVM)]
public object DataContextImported { set { this.DataContext =
value; } }
```

Communiceren tussen modules is nodig indien de modules afhankelijk zijn van elkaar. Dit onderdeel is niet verplicht om te implementeren, maar is alleen van toepassing als de modules afhankelijk zijn van elkaar. In ons voorbeeld project hebben we een lijst van producten en een winkelwagen waar de producten in kunnen worden geplaatst. Hoewel het één gehele applicatie is, zijn de twee onderdelen toch aparte modules. Deze modules kunnen los van elkaar leven, los van elkaar worden geïnstalleerd maar dienen toch te kunnen communiceren zonder deze aan elkaar te Referencen. In PRISM is hiervan de oplossing Event Aggregation gebruikt. Event Aggregation is gebaseerd op het Publish/Subscribe Design Pattern. Als voorbeeld gebruiken we hier het toevoegen aan de winkelwagen vanuit de productenlijst als Publish moment.

```
void OnAddToCartCommand(Product product)
{
    _eventAggregator.GetEvent<AddToCartEvent>().Publish(product);
}
```

```
namespace Iodine.Infrastructure.CompositeEvents
{
    public class AddToCartEvent : CompositePresentationEvent
<Product>
    {

    }
}
```

Aan dit AddToCartEvent kunnen andere modules zich inschrijven (Subscriben). In het voorbeeld heeft de ShoppingCartModule dat gedaan. Aangezien het AddToCartEvent een Product object verwacht zal deze worden doorgegeven, zodat dit product in de winkelwagen kan worden gestopt en worden weggeschreven naar de database.

```
private void SubscribeToEvents()
{
    _eventAggregator.GetEvent<AddToCartEvent>().
Subscribe(OnAddToCart);
}
```

```
public void OnAddToCart(Product product)
{
//Add new line to the collection
Cart.ShoppingCartLines.Add(new ShoppingCartLine() { Product =
product });

//Save changes to the database
if ( _ctx.CurrentDomainContext.HasChanges)
_ctx.CurrentDomainContext.SubmitChanges();

//Give a notification to the UI
ShoppingCartLinesChanged();
}
```

Module downloading zorgt ervoor dat functionaliteit beschikbaar komt aan de Client applicatie met als doelstelling dat de Module beschikbaar komt als de eindgebruiker er gebruik van gaat maken. Module downloading vindt zijn oorsprong in de Bootstrapper van de Silverlight applicatie.

```
private void Application_Startup(object sender, StartupEventArgs
 e)
{
    WebShopBootStrapper bootstrapper = new WebShopBootStrapper();
    bootstrapper.Run();
}
```

De gebruikte Bootstrapper in dit voorbeeld project is een Mef-Bootstrapper. De UnityBootstrapper kan ook worden gebruikt. De MEF variant is gekozen omdat MEF een vast onderdeel is geworden van het .Net Framework 4.0. Daarnaast komt MEF ook terug in de koppeling tussen View en ViewModel door middel van de Import en Export Attributes. Het automatisch discoveren van Assemblies en Classes op een Managed Code manier geeft MEF een meerwaarde boven Unity.

```
public class WebShopBootStrapper : MefBootstrapper
```

Zoals gezegd is de Bootstrapper verantwoordelijk voor het creëren van een Module Catalog. In het onderstaande code voorbeeld wordt door middel van PRISM de Module Catalog opgebouwd op basis van een Xaml file.

```
protected override IModuleCatalog CreateModuleCatalog()
{
    return Microsoft.Practices.Prism.Modularity.ModuleCatalog.
CreateFromXaml(new Uri(
                    "/WebShop;component/ModulesCatalog.xaml",
                     UriKind.Relative));
}
```

In de ModulesCatalog.Xaml file staan de modules beschreven.

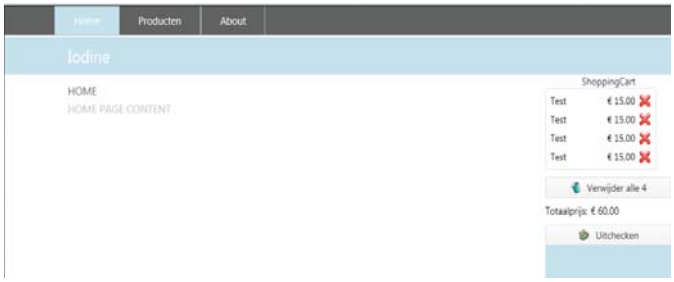
```
<Modularity:ModuleCatalog xmlns="http://schemas.microsoft.com/
winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
                            xmlns:sys="clr-namespace:System;assembly
```

```
=mscorlib"

xmlns:Modularity="clr-namespace:Microsoft.Practices.Prism.
Modularity;assembly=Microsoft.Practices.Prism">
    <Modularity:ModuleInfoGroup InitializationMode="OnDemand">
        <Modularity:ModuleInfo
            Ref="Products.xap"
            ModuleName="ProductsModule"
            ModuleType="Products.ProductsModule, ProductsModule,
Version=1.0.0.0, Culture=neutral, PublicKeyToken=null"
            />
    </Modularity:ModuleInfoGroup>
    <Modularity:ModuleInfoGroup InitializationMode="WhenAvailabl
e">
        <Modularity:ModuleInfo
            Ref="ShoppingCarts.xap"
            ModuleName="ShoppingCartsModule"
            ModuleType="ShoppingCarts.ShoppingCartsModule,
ShoppingCartsModule, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null"
            />
    </Modularity:ModuleInfoGroup>
</Modularity:ModuleCatalog>
```

Afhankelijk van de InitializationMode worden de modules On-Demand of WhenAvailable geladen. In geval van WhenAvailable wordt de module altijd gedownload naar de Client. Voor de gebruiker is de applicatie beschikbaar, maar de applicatie moet nog wachten op de module. Het is belangrijk de modules zo compact mogelijk te houden, zodat deze snel worden gedownload.



Bij OnDemand komt de module naar de Client wanneer de gebruiker daar om vraagt. In dit voorbeeld wordt de ProductsModule geladen wanneer de gebruiker op de Producten tab klikt.

Line-of-Business applications

De modulaire opbouw van Silverlight projecten is een belangrijke stap om Silverlight geschikt te maken om gebruikt te worden in Business Applications. De modulaire opbouw maakt het mogelijk om alleen delen te laden die de eindgebruiker daadwerkelijk nodig heeft. Grote applicaties kunnen op deze manier blijven performen door de korte downloadtijd.
Voor de start van een Silverlight applicatie moet er altijd worden nagegaan hoe groot de applicatie gaat worden. Voor kleinschalige oplossingen is het niet noodzakelijk de applicatie op te delen in meerdere XAP files. De complexiteit van deze technologie weegt dan niet op tegen de minimale performance winst. Het gebruik van de Module Catalog en het downloaden van de XAP files kan achterwege worden gelaten. Het gebruiken van MVVM als Design Pattern voor het scheiden van de business- en presentatie logica kan behouden blijven.

Conclusie

De modulaire opbouw van Silverlight projecten is een belangrijke stap om Silverlight te gebruiken in Business Applications. De modulaire opbouw maakt het mogelijk om alleen delen te laden die de eindgebruiker daadwerkelijk nodig heeft. Grote applicaties kunnen op deze manier blijven performen door de korte downloadtijd. In dit artikel is, ondersteund met een aantal codevoorbeelden, ingegaan op deze techniek.
Dit artikel is bedoeld als startpunt voor een succesvolle Silverlight applicatie. Voor de start van een Silverlight applicatie moet er altijd worden nagegaan hoe groot de applicatie gaat worden. Voor kleinschalige oplossingen is het niet noodzakelijk de applicatie op te delen in meerdere XAP files. De complexiteit van deze technologie weegt dan niet op tegen de minimale performance winst. Het gebruik van de Module Catalog en het downloaden van de XAP files kan achterwege gelaten worden. MVVM, meerdere Views op een ViewModel en het delen van Commands blijven ongewijzigd. Hieronder nog enkele links om zelf aan de slag aan te gaan!

Links

http://compositewpf.codeplex.com/ - Project pagina voor Prism 4.0
http://mef.codeplex.com/ - Project pagina MEF
http://www.silverlight.net/getstarted/ - Spreekt voor zich
http://silverlight.codeplex.com/ - Silverlight Toolkit
http://channel9.msdn.com/Shows/SilverlightTV - Enorme bron met instructie video's gehost door John Papa.
http://mtaulty.com/communityserver/blogs/mike_taultys_blog/default.aspx - Mike Taulty heeft heel veel voorbeeld code en kan perfect uitleggen.
http://msdn.microsoft.com/en-us/library/gg406140.aspx - Uitleg over PRISM 4.0

Ralph Jansen, is als Software Ontwikkelaar werkzaam bij Centric IT Solutions. Hij is bereikbaar via ralph.jansen@centric.eu en op zijn blog http://locktar.wordpress.com

Arjan Hordijk, is als Software Architect werkzaam bij Centric IT Solutions. Hij is bereikbaar via arjan.hordijk@centric.eu en op twitter via @ahordijk

Arne Wauters, is als Software Ontwikkelaar werkzaam bij Centric Retail Solutions. Hij is bereikbaar via arne.wauters@centric.eu en op twitter via @AWT_CentricBEL