Demystifying MVVM

Code beter onderhouden en testen in XAML applicaties

Kevin Dockx

XAML bestaat al verscheidene jaren (sinds WPF, 2005), maar is slechts de laatste 18 maanden aan een steile opgang begonnen dankzij snel op elkaar volgende releases van Silverlight. Aangezien ook de nieuwe Windows Mobile Phone 7 series gebruik zal maken van Silverlight, en dus XAML, om applicaties in te bouwen, wordt XAML meer en meer de standaard manier om een UI in te bouwen met Microsoft technologieën.

XAML is echter meer dan UI-taal, ook de manier van werken, de manier waarop je je C# code zal gaan schrijven, kun je wijzigen zodat je gebruik maakt van de sterke features die het framework je biedt. Enter MVVM, ofwel Model-View-ViewModel: het de facto standaard design pattern voor zij die WPF/Silverlight-applicaties ontwikkelen, en dat ervoor zorgt dat zaken als DataContext, Data Binding en het Observer Pattern optimaal gebruikt kunnen (moeten!) worden.

MVVM: wat, en waarom?

MVVM zal al snel bekend voorkomen voor zij die al gebruik hebben gemaakt van Martin Fowlers' Model-View-Presenter pattern, en ook zij die ervaring hebben met Model-View-Controller zullen zich snel thuis voelen. Het pattern vertoont sterke gelijkenissen met deze twee patterns, en heeft als grote voordeel dat het zorgt voor separation of concerns, wat op zijn beurt weer leidt tot betere testability en code maintainability. Het belangrijkste voordeel is echter dat het ervoor zorgt dat je als developer gebruik zal moeten maken van Data Binding & DataContext om je UI operationeel te maken – wat er op zijn beurt weer voor zorgt dat er een striktere scheiding kan zijn tussen het werk dat een UX/UI developer (de Devigner) en een andere developer zal uitvoeren: de developer schrijft zijn code in het ViewModel/achter het Model (niet meer in de code behind file van de View), zodat de Devigner zich kan bezighouden met het uitwerken van de View.

Hoe ziet dit er nu in de praktijk uit? Bekijk even figuur 1. Hierop zie je de verschillende onderdelen van ons pattern: het Model, ofwel: de representatie van je data. In sommige implementaties is dit Model niet meer dan een gegenereerde proxy class of de gegenereerde code van WCF RIA Services, in andere implementaties wordt er dan weer een client-side business-laag rond gebouwd, om de DTO's terug om te zetten naar bruikbare objecten. Welke aanpak je ook kiest, je model blijft een object-based representatie van de data. Dan heb je de View, ofwel de UI. Deze bevat alle nodige UI elementen, en wordt typisch ontworpen door een Devigner. Ten derde is er het ViewModel, wat zorgt voor de link tussen het Model en de View: het ViewModel bevat typisch een aantal properties (bvb: een lijst van winkels: Stores), die in hun setter via INotifyPropertyChanged de UI verwittigen dat ze gewijzigd zijn. De waarden voor deze properties komen onder andere uit het Model, aangepast zodat ze bruikbaar zijn voor de View. De View zal dan syntax als "ItemsSource = {Binding Stores}" moeten bevatten, om deze lijst te tonen. Kortom: het ViewModel is de DataContext van de View, bevat properties waaraan UI elementen in de View gebind worden, en deze properties zijn vaak terug te koppelen naar het Model, omgevormd door het ViewModel om bruikbaar te zijn in de View. Sommigen noemen het ViewModel dan ook wel eens een 'converter on steroids'.

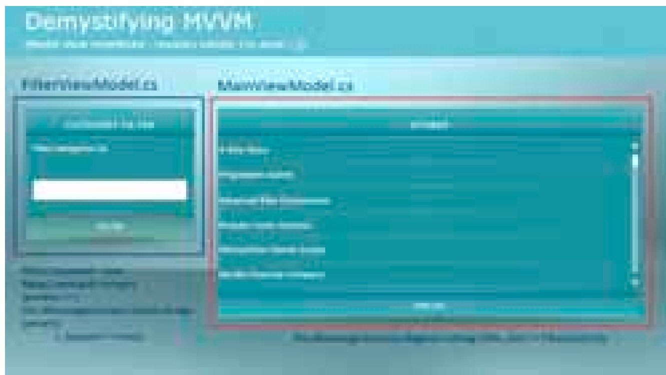
FIGUUR 1

Voorbeeldapplicatie

De voorbeeldapplicatie die we bouwen om MVVM te illustreren bestaat uit 3 Views: een InfoView, een FilterView en een MainView. MainView zal een lijst van Stores tonen, die vanuit FilterView (TextBox en Button) gefilterd zal kunnen worden. InfoView bevat een loading-animatie. Elke View heeft z'n ViewModel, en onze Models komen uit de gegenereerde code van WCF RIA Services, die achterliggend via Entity Framework verbindt naar een databank.

ViewModel base

Om MVVM mogelijk te maken in een applicatie zal je een aantal zaken moeten bouwen, of een bestaand framework gebruiken, zoals MVVM Light, PRISM of Caliburn. Als basisvoorbeeld zullen we zelf een ViewModel base class bouwen.
De basisimplementatie van een ViewModel base class, waar alle ViewModels van zullen overerven, hoeft niet meer te zijn dan een class die INotifyPropertyChanged implementeert (zodat vanuit



HET MAINVIEWMODEL.

het ViewModel notificaties kunnen gestuurd worden naar de UI bij wijziging van een van z'n properties). In dit voorbeeld gaan we echter net iets verder: we maken een generische ViewModel<T> base class aan, waarbij T van type ViewModel<T> moet zijn. Dit doen we om type-safe Property Changed events te kunnen afvuren (zodat eventuele errors al bovenkomen at compile time in plaats van at run time). Dankzij dit stukje code kunnen we een Stores-property maken in MainViewModel zoals in codevoorbeeld 1, welke we opvullen met data via WCF RIA Services.

```
private ObservableCollection<Store> _stores;
/// <summary>
/// The Stores property
/// </summary>
public ObservableCollection<Store> Stores
{
    get
    {
        return _stores;
    }
    set
    {
        _stores = value;
        RaisePropertyChanged((vm) => vm.Stores);
    }
}
```

CODEVOORBEELD 1

Aangezien MainViewModel de DataContext van MainView.xaml wordt, kunnen we in MainView de ItemsSource van een ListBox binden aan Stores met volgende syntax: <ListBox ItemsSource={Binding Stores}>.

View en ViewModel met elkaar verbinden: View first, of ViewModel first?

Als we nu de applicatie opstarten zal er nog niks gebeuren: MainView is immers nog niet gekoppeld aan MainViewModel. Er zijn verschillende manieren mogelijk om dit op te lossen. Twee approaches worden meteen duidelijk: de View first approach, of de ViewModel first approach. De eerste houdt in dat de View geïnstantieerd wordt door de applicatie – bvb door de View op MainPage.xaml te plaatsen – en dat deze zal instaan voor het creëren van z'n ViewModel. Dit resulteert vaak in een aanpak waarbij de DataContext gezet wordt in XAML, en een verwijzing bevat naar een ViewModel property die beschikbaar gemaakt wordt via een class die in een Resource Dictionary (bvb App.xaml) komt, waardoor je er via {StaticResource} aankan – het zogeheten Locator pattern.

De tweede approach is net omgekeerd, en verloopt vaak met behulp van een IoC container: het ViewModel wordt gecreëerd, en deze ontvangt een identificatie van de View die bij dat ViewModel hoort, waarna de IoC container verantwoordelijk is voor instantiatie.

Er is geen 'betere' approach, maar belangrijk is wel dat je je View niet beschikbaar maakt vanuit je ViewModel, omdat op die manier de mogelijkheid bestaat om rechtstreeks UI elementen te wijzigen vanuit een ViewModel, wat niet de bedoeling is.

View en ViewModel met elkaar verbinden: MEF

Een handige mogelijkheid om een View aan een ViewModel te koppelen is MEF, ofwel: het Managed Extensibility Framework (meegeleverd met .NET 4.0 / Silverlight 4). Met dit framework kunnen we de verantwoordelijkheid om voor een instantie van een ViewModel te zorgen bij MEF leggen, niet bij de View. We doen dit door MainViewModel te exporteren als iets van het type IMainViewModel (een lege, marker interface, die zich bevindt in het .Contracts-project), met het attribuut [Export(typeof(IMain ViewModel))]. Met het attribuut [PartCreationPolicy(CreationP olicy.Shared)] duiden we dan weer aan dat we willen dat MEF, indien er reeds een instantie bestaat, diezelfde instantie zal teruggeven in plaats van een andere aan te maken.

In onze MainView moeten we dan nog aangeven dat deze 'iets' van het type IMainViewModel verwacht (Import), en dat dit als DataContext moet dienen (zie codevoorbeeld 2). Met CompositionInitializer.SatisfyImports(this) zorgen we er ten slotte voor dat aan alle imports van MainView voldaan wordt, en op deze manier worden MainView en MainViewModel aan elkaar gekoppeld. Als we de applicatie nu opnieuw opstarten, zien we dat de ListBox met Stores opgevuld wordt.

```
public partial class MainView : Page, IView
{
    public MainView()
    {
        InitializeComponent();

        if (!(ViewModelBase.IsInDesignModeStatic))
        {
            // load VM through MEF
            CompositionInitializer.SatisfyImports(this);
        }
    }

    [Import(typeof(IMainViewModel))]
    public object ViewModel
    {
        set
        {
            this.DataContext = value;
        }
    }
}
```

CODEVOORBEELD 2

Code behind in een View?

Dit brengt ons meteen bij de vraag: kan dat wel, code in de code behind file van een View? Zondigt dit niet met het MVVM pattern? Er zijn enkele hoofddoelen aan MVVM: seperation of concerns, mogelijkheid tot samenwerking tussen de UX designer en developers, en de mogelijkheden van het framework optimaal benutten. Dit houdt niet in dat een View geen code in z'n code behind mag bevatten: het houdt in dat de View niet meer verantwoordelijkheid mag bezitten dan strikt noodzakelijk, en dat deze geen conditionele, te testen code mag bevatten. Met MEF ziet de code behind van een View er uit als in codevoorbeeld 2. We zien hier een ViewModel property, maar de View is niet verantwoordelijk voor het instantiëren van deze property: MEF zorgt er voor dat we een instantie van het juiste ViewModel terugkrijgen. Daarnaast bevat deze geen te testen conditionele code (zoals code die achter een

event handler zou kunnen steken), en blijft de scheiding tussen UX design / development gewaarborgd door de check op het al dan niet in design mode zijn alvorens MEF toe te laten te voldoen aan de import van het ViewModel. We zondigen dus niet tegen de hoofddoelen van MVVM.

Daarnaast is een volledige afscheiding tussen View en ViewModel quasi onmogelijk, noch de bedoeling: per definitie zijn een View en een ViewModel volgens conventie reeds aan elkaar gebonden (hoewel meerdere Views bij verschillende ViewModels kunnen horen, en omgekeerd) – immers, als de properties waaraan je in je View bindt niet bestaan in je ViewModel, zal de applicatie niet werken.

Commanding

Het volgende dat we moeten aanpakken is Event Handling: standaard zal bvb een Button Click Event Handler in de code behind van onze Views terechtkomen, en dit willen we natuurlijk niet. We willen dit Click Event kunnen afhandelen in het ViewModel – met andere woorden: we willen dit kunnen binden aan iets in ons ViewModel. De oplossing hiervoor heet Commanding.
Silverlight 4 bevat de ICommand interface. Elke implementatie hiervan kan als command gebruikt worden (zie code voor een implementatie, een gewone en eentje die een type-safe parameter ontvangt). Daarnaast bevat Silverlight 4 ook Command- en CommandParameter-properties op controls die overerven van ButtonBase. Echter, dit volstaat niet voor de meeste applicaties: met deze manier van werken kunnen we immers niet gaan binden aan, bvb, een SelectionChanged event.
Om dit wel mogelijk te maken kunnen we Event Triggers gebruiken: de System.Windows.Interactivity assembly (meegeleverd met Blend) bevat de mogelijkheid om een Event Trigger te gaan koppelen aan eender welk event. Als we dan een eigen TriggerAction<FrameworkElement> definiëren, CommandTriggerAction (zie meegeleverde code), kunnen we deze als Event Trigger gebruiken. In codevoorbeeld 3 kan je zien dat we het Click event van de Filter Button in FilterView.xaml op deze manier koppelen aan het FilterCommand, en dat we als parameter de ingevulde tekst uit txtFilter meegeven. Dit Command zelf wordt gedefinieerd in het FilterViewModel, zoals je kan zien in codevoorbeeld 4.

```
<Button Content="FILTER">
  <i:Interaction.Triggers>
    <i:EventTrigger EventName="Click">
      <mvvm:CommandTriggerAction Command="{Binding FilterCommand}"
      CommandParameter="{Binding ElementName=txtFilter, Path=Text}" />
    </i:EventTrigger>
  </i:Interaction.Triggers>
</Button>
```

CODEVOORBEELD 3

```
FilterCommand = new RelayCommand<string>((param) => {
                    // send a msg to MainView
                     this.MessengerInstance.Send<string>(param);
                }
                    , (param) => true);
```

CODEVOORBEELD 4

Als we nu op deze knop klikken, zullen we terechtkomen in het gedefinieerde FilterCommand in FilterViewModel.

Communiceren tussen ViewModels

Op deze manier komen we naadloos bij het laatste grote blok betreffende MVVM: communiceren tussen verschillende ViewModels. Immers: bij het drukken op de Filter knop moet de lijst van Stores gefilterd worden. Maar: het FilterCommand is gedefinieerd op FilterViewModel, terwijl de lijst van Stores gedefinieerd is op MainViewModel. Bij de MVVM Light Toolkit via een Messenger. Het is deze Messenger die we ook gebruiken in onze demo-applicatie (code: zie meegeleverde democode). Onze ViewModel base class zal een verwijzing naar een static instantie van deze class bevatten, zodat deze voor de applicatie beschikbaar is. Een Messenger werkt volgens het Subscribe/Send-principe: een bepaald ViewModel zal zich registreren om berichten van van een bepaald type te ontvangen. Als je dan code schrijft om via die Messenger een bericht van dat bepaald type te sturen, zal het ViewModel dat zich geregistreerd heeft hiervoor dat bericht ontvangen.
Meer specifiek: in MainViewModel schrijven we code om berichten van het type string te ontvangen via de default Messenger, zoals in codevoorbeeld 5. In het FilterCommand in FilterViewModel gebruiken we diezelfde Messenger om een bericht te sturen van type string, zoals je kan zien in codevoorbeeld 4.

```
this.MessengerInstance.Register<string>(this, (str) => Filter-
List(str));
```

CODEVOORBEELD 5

Bij het klikken op de Filter Button zal MainViewModel een bericht ontvangen met daarin de meegegeven filter, waarna deze aan de hand hiervan de lijst van Stores kan beperken.
Je kan natuurlijk ook Messengers gaan definiëren die enkel tussen bepaalde ViewModels werken in plaats van de algemene Messenger te gebruiken.

Dialogs, Animaties, State, Navigatie…

Enkele vaak voorkomende vragen over/bezwaren omtrent MVVM klinken als volgt: "Je kan geen storyboard starten vanuit een ViewModel", "Hoe kan ik een Dialog Window openen vanuit een ViewModel", ... Deze problematiek is in de meeste gevallen zeer gelijkaardig, en komt neer op het nodig hebben van toegang tot elementen die gedefinieerd zijn in de UI (StoryBoard gedefinieerd in XAML, ChildWindow instantiëren...), terwijl dat niet kan bij een correcte implementatie van MVVM.
Het antwoord op deze problemen is: abstractie. Het komt er op neer dat je de verantwoordelijkheid voor, bvb, het starten van een Storyboard of het tonen van een popup, niet meer bij het ViewModel legt, maar bij een service die je zelf schrijft.
Laten we als voorbeeld een Dialog Window nemen. Als je werkt met Rich Applications is het eigenlijk een goed idee om dit zo veel mogelijk te vermijden: popups blokkeren een gebruiker vaak, en zorgen voor onderbrekingen in een goede applicatieflow. Ze zou-

VIEWMODELMESSAGING.

den zelden nodig moeten zijn, maar soms kan je misschien niet anders. Maar je mag natuurlijk geen instantie van een ChildWindow gaan toevoegen aan View vanuit een viewModel.
Om dit toch mogelijk te maken voorzien we een IDialogService interface, met 1 gedefinieerde methode, ShowDialog, met een specifieke implementatie, DialogService (zie voorbeeld X). Deze implementatie is verantwoordelijk om het ChildWindow te creëren. In MainViewModel voeren we voorgenoemde ShowDialog methode uit, eventueel met het meegeven van een callback methode.
Op deze manier hebben we voor abstractie gezorgd, en kunnen we toch een popup tonen. Door het feit dat onze ViewModels los staan van de Views, kunnen we ze heel makkelijk gaan unit testen. In de Silverlight Toolkit kan je een Unit Test Project Template vinden. Zo'n project aanmaken en de nodige referenties leggen naar het project met de ViewModels, DemystifyingMVVM.VM, en naar het project met de base classes, DemystifyingMVVM.Base, volstaat om deze te kunnen testen. In codevoorbeeld 6 kan je zo'n unit test zien.

```
[TestMethod]
public void TestVMInit()
{
    MainViewModel vm = new MainViewModel();

    Assert.IsInstanceOfType(vm, typeof(MainViewModel));
}
```

CODEVOORBEELD 6

Mocking is ook mogelijk, dankzij MEF. In de democode zal je een .Mocks-project vinden, wat mocks bevat van de gebruikte ViewModels. Deze ViewModels krijgen een Export-attribute volgens hetzelfde contract (bvb IMainViewModel) als onze echte ViewModels. Door MEF te vertellen dat hij z'n catalog moet bouwen door gebruik te maken van de DemystifyingMVVM.Mocks-assembly in plaats van de DemystifyingMVVM.VM-assembly zal de applicatie de mocks gebruiken in plaats van de echte ViewModels. Dit kan je zien in codevoorbeeld 7, uit App.xaml.cs.

```
var mainCatalog = new AssemblyCatalog(Assembly.GetExecuting-
Assembly());
var vmCatalog = new AssemblyCatalog(Assembly.Load("Demystifying-
MVVM.Mocks, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null"));
```

CODEVOORBEELD 7

Conclusie

Model-View-ViewModel is een aan te raden design pattern om te gebruiken in XAML applicaties, aangezien het zorgt voor een verbeterde onderhoudbaarheid en testbaarheid van je code. Ook zorgt het dat de kracht van het framework optimaal wordt gebruikt. Het is echter niet 'set in stone': er bestaan verschillende implementaties van, waarbij de ene niet noodzakelijk beter is dan de andere, maar misschien wel beter voor een specifieke situatie. Kies de implementatie die het best geschikt is voor het project waar je aan aan het werken bent.

Kevin Dockx, is technisch specialist/projectleider voor .NET webapplications en solution manager voor rich applications.

(Advertentie)

everything 4dotnet

The one-stop company for .NET development

Kies de juiste architectuur!

..door u uitgebreid te laten adviseren door één van onze .NET software architecten!

Bovendien, goed advies hoeft niet duur te zijn! (€ 750 per dag)

Ga naar www.4dotnet.nl of bel voor een afspraak: 0522-241448

Data Management Solutions
Learning Solutions
Custom Development Solutions

4DotNet bv • Paradijsweg 2 • 7942 HB Meppel • t. 0522-24 14 48 • info@4dotnet.nl • www.4dotnet.nl