NorthwindTradersMVVM Demo - Notes, Tips & Tricks

How to name your Files

View

Simply append “View”.

ViewModel

Just append “Model” to the View name, like CustomerEditView → CustomerEditView*Model*.
If the View name is called e.g. MainWindow, append “ViewModel” → MainWindow*ViewModel*.

Model

Use the domain object names like Product, Patient, Employee, …

Where to Store your Files

1. Folder by Type: View(s), ViewModel(s), …
2. Folder by Feature: Products, Patients, Employees with all Views and ViewModels together per feature

In a larger application *Folder by Type* will result in a *lot of files* inside a single folder and it takes more time to find the corresponding file in the other folder. Using *Folder by Feature* there are usually max. related 10 files per feature.

Models and DataServices could best be stored in separate folders or Class Library Projects to be reused/shared.

Window Management and Navigation

In this demo the so-called ViewModel-first approach by means of Data Templates is used. The parent ViewModel acts as the central hub for navigation and communication.

The Chicken or the Egg; who is created First

View-First

The MainWindow itself uses the View-First approach to ‘marry’ its ViewModel:

```
<Window x:Class="Northwind.GUI.MainWindow"
        ...
        xmlns:local="clr-namespace:Northwind.GUI"
        ... >

    <Window.DataContext>
        <local:MainWindowViewModel/>
    </Window.DataContext>
```

ViewModel-First

The ‘child’ windows are implemented as *UserControls* and are loaded into the MainWindow using *implicit* DataTemplates (so without x:Key) to implement the ViewModel-First approach.

```
class MainWindowViewModel : BindableBase
{
    private BindableBase _currentViewModel;

    // Set by method OnNavigateChanged
    public BindableBase CurrentViewModel
    {
        get { return _currentViewModel; }
        set { SetProperty(ref _currentViewModel, value); }
    }
}
```

```
<Window x:Class="Northwind.GUI.MainWindow"
        ...
        xmlns:employees="clr-namespace:Northwind.GUI.Employees"
        xmlns:orders="clr-namespace:Northwind.GUI.Orders"
```

```
        ... >

    <Window.Resources>
        <!-- the following says: if something of type XYZ needs to be rendered, use this ...View -->
        <DataTemplate DataType="{x:Type employees:EmployeeListViewModel}">
            <employees:EmployeeListView/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type orders:OrderViewModel}">
            <orders:OrderView/>
        </DataTemplate>
    </Window.Resources>

    <ContentControl Content="{Binding CurrentViewModel}"/>

</Window>
```

Behaviors

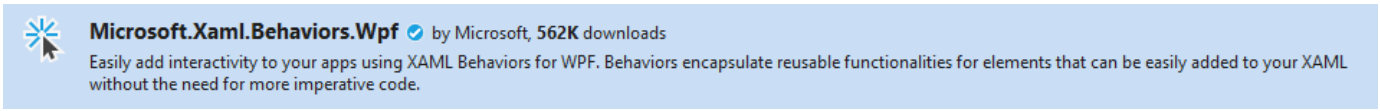
Behaviors were first introduced with *Expression Blend* version 3 in 2009, to encapsulate pieces of functionality into a *reusable* component. These components than can be *attached to controls* to give them an *additional behavior*. The ideas behind behaviors are to give the *interaction designer* more flexibility to design complex user interactions *without* writing any code. Examples of a behaviors are play sounds, start/stop/pause animations, call methods, invoke commands, ...

In MVVM they can be used to enable communication between the View and ViewModel where *Commands* and *PropertyChanged* notifications do not suffice.

Originally present in the *Blend SDK*, the since late 2018 open sourced *XAML Behaviors for WPF* can be added to our projects as a *NuGet* Package.

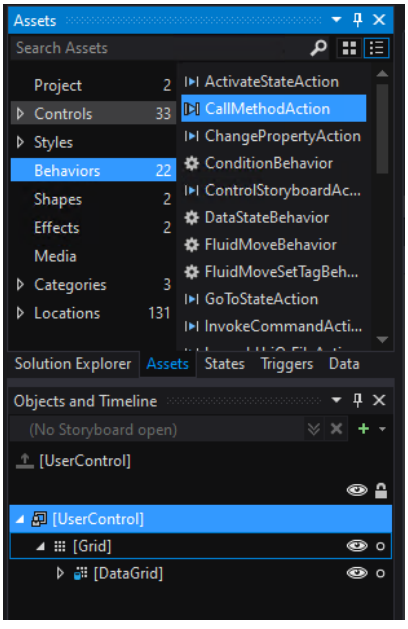
Visual Studio part

1. In Visual Studio go to *menu* Tools > NuGet Package Manager > Manage NuGet Package for Solutions…
2. Select *Browse* and search for *behaviors*
3. Select the latest version of **Microsoft.Xaml.Behaviors.Wpf**



4. Make sure only ☑ Northwind.GUI is selected
5. Press the [Install] button

Blend for Visual Studio part



1. In Visual Studio go to *menu* View > Design in Blend…
2. In *Blend*, Solution Explorer is at the left side. Select the View for which a *Behavior* is needed.
3. In panel *Objects and Timeline* (below the Solution Explorer) select the [UserControl] root.
4. In tab *Assets*, select *Behaviors* and double-click *CallMethodAction*.
   The necessary XAML *namespace* and *EventTrigger* have been added and will be configured in Visual Studio.
5. *Save All* and *Close* Blend

result (after some property editing) in XAML:

```
<UserControl x:Class="Northwind.GUI.Employees.EmployeeListView"
             ...
             xmlns:b="http://schemas.microsoft.com/xaml/behaviors"
             ...>

    <b:Interaction.Triggers>
```

```
        <b:EventTrigger EventName="Loaded">
            <!-- TargetObject="{Binding}" is used to get a reference to the current ViewModel -->
            <b:CallMethodAction TargetObject="{Binding}" MethodName="LoadEmployees"/>
        </b:EventTrigger>
    </b:Interaction.Triggers>
```

Hocus Pocus View Creation

How To: Scaffold an Edit View in Visual Studio

- Visual Studio menu View > Other Windows > Data Sources
- Click “Add New Data Source...” link or button
- In dialog “Choose a Data Source Type” select **Object**
- Find and select the desired *model* class
- Open root object to select (e.g. DataGrid or Details)
- Expand and select which props should be drawn and in what form (e.g. TextBox or Label)
- Drag & Drop at your heart's desire

Validation

Next to using *Exceptions*, implementing *IDataErrorInfo* and using *ValidationRules* (as seen in *demos* 🕶 *WPF104 > BindingValidationDemo and IDataErrorInfoImplementationDemo*), this demo uses **INotifyDataErrorInfo** which was introduced in WPF in .NET 4.5 (Summer 2012). Implementing *INotifyDataErrorInfo* supports querying the object for errors associated with properties and it fixes some of the shortcomings of the other options. For example, it allows asynchronous validation and it allows properties to have more than one error associated with them.

Validation takes place by using *attributes* from the *System.ComponentModel.DataAnnotations* Namespace. Some examples: `[Required]`, `[StringLength(…)]`, `RegularExpression(…)]`, `[CreditCard]`, `[EmailAddress]`.

Dependency Injection & IoC Containers

Inversion of Control (*IoC*) and Dependency Injection (*DI*) are two design patterns that are closely related.

- *IoC* is a generic term meaning that objects do not create other objects on which they rely to do their work. Instead, they get the necessary *implementation objects* from an outside service or framework.
- *DI* is a form of IoC, where *implementation objects* are passed into an object - usually by a framework component that passes constructor parameters and set properties – on which the object will 'depend' in order to behave correctly.

A container is infrastructure code that does both of those patterns for you, and takes responsibility for the following actions:

- *Constructs an object when asked*
  So instead of instantiating objects yourself, you ask the container to produce an object for you.
- *Determines what that object depends on*
  As part of that process, the container will determine what that object depends on, and depending on the container you use, it can do that based on parameterized constructors, properties or methods.
- *Constructs the desired dependencies*
  To provide those dependencies, the container will then construct those dependencies.
- *Injects those dependencies into the object being constructed*
  The dependencies being constructed may also have dependencies of their own, and so on. So you have to *recursively* do this process until an entire object graph is built up from the object you asked for down through all of its dependencies. A container can also manage different instancing patterns and produce *singleton* objects where it constructs them on the first try and then just hands out a reference to them each other time some object takes a dependency on them.

There are many open source or commercial containers available for .NET, for example Unity, AutoFac, Ninject, StructureMap.