# TUTORIALREIHE WPF LERNEN

## WPF verständlich und anhand von Praxisbeispielen erklärt

**Sascha-Heinz Patschka**

### Inhaltsverzeichnis (inkrementell)

## 4.0

# Eine WPF Telefonbuch App

Jetzt kommen wir zum praktischen Teil. Wir erstellen nun eine kleine App unter Verwendung des bisher gelerntem. Wir werden noch nicht mit MVVM oder ähnlichem Pattern in Berührung kommen, das ist aber gewollt so. Es ist besser wenn wir uns noch nicht mit komplizierten Pattern Beschäftigen und erstmal lernen wie wir nun z.b. von Binding profitieren können.

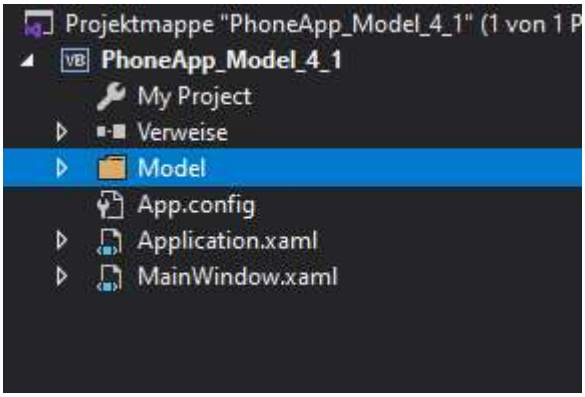
Wir haben ja gelernt dass wir wunderbar auf Klassen Binden können und hier ein wirklich Leistungsstarkes System an die Hand bekommen um schnell und unkompliziert Daten mit Controls zu verbinden. Und genau das versuchen wir nun in der Praxis.

Es wird eine Telefonbuch App.
Da ich ungerne Beispiele mache welche nicht wirklich funktionsfähig sind soll die App - auch wenn sie nur diesem Tutorial dienen soll - Daten speichern können. Also soll auch die persistierung funktionieren, damit wir hier flexibel sind werde ich Klassen haben welche schlicht Serialisiert werden. Wir könnten in diese „Modelklassen“ natürlich theoretisch INotifyPropertyChanged implementieren um diese direkt als eine Art „ViewModel“ zu missbrauchen. Ich bin aber kein Freund davon. Warum werden wir im Laufe dieses Kapitel noch näher erfahren. Nun gut, fangen wir mit dem Model eines Telefonbuchs an.

## 4.1

# Das Model erstellen / was soll die App können

Wir erstellen eine neue WPF Solution in welcher wir erstmal einen Order „Model“ anlegen. In diesem landen unsere Modelklassen.




Wir legen eine neue Klasse „Contact“ an und fügen folgende typische Properties ein.

## 1

```
Namespace Model

    Public Class Contact

        Public Property FirstName As String
        Public Property LastName As String
        Public Property ImagePath As String
        Public Property Birthday As Date?
        Public Property Note As String

    End Class
End Namespace
```

Ich denke diese erklären sich von selbst und sind typisch für einen Kontakt eines Telefonbuchs.

Nun hätten wir gerne noch eine Eigenschaft um die Telefonnummern oder Mailadressen zu hinterlegen. Das können ja mehrere je Kontakt sein. Also ein Kontakt kann Beispielsweise zwei Mailadressen und eine Telefonnummer haben. Aber vielleicht auch eine Website und einen SocialMedia Account wie Facebook oder Instagram.
Wir benötigen hierfür eine Auflistung. Also legen wir uns eine Eigenschaft vom Typ List(Of T) an. Dafür benötigen wir aber erstmal eine neue Klasse. Nennen wir diese ContactData.

```
Namespace Model
    Public Class ContactData

        Public Property Data As String
        Public Property DataType As EnumContactDataType

    End Class

    Public Enum EnumContactDataType
        Phonenumber = 0
        Mail = 1
        SocialMedia = 2
        Fax = 3
        Website = 4
    End Enum

End Namespace
```

Diese Klasse hat zwei Eigenschaften. Zum einen die Daten vom Typ String. Denn es sollen sowohl Telefonnummern aber auch z.b. Mailadressen gespeichert werden können. Da ist String der einzige Datentyp der hierfür geeignet ist. Um aber später unterschieden zu können um welche Art von Daten es sich hierbei handelt habe ich einen Enumerator erstellt und eine Eigenschaft welche den Wert dieses Enumerators hält. So können wir später die Daten richtig zuordnen.

Nun können wir diese als Auflistung in unsere Contact Klasse einbinden.

```
Namespace Model

    Public Class Contact

        Public Property FirstName As String
        Public Property LastName As String
        …
        Public Property ContactData As List(Of ContactData)

    End Class
End Namespace
```

Tja, ich würde sagen das reicht uns fürs erste erstmal als Model für unser Telefonbuch. Ich weiß das Kapitel war etwas kurz, aber keine Sorge, beim nächsten geht's ran an den Speck.

Hier geht's zum Video: https://youtu.be/2eWaAs1w9ok

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

## 4.2

# Die erste View erstellen

In diesem Kaptitel werden wir unsere erste View mit einer Klasse verbinden. Wie im letzten Kapitel bereits erwähnt werde ich hier nicht direkt mit den Modelklassen arbeiten, ihr werdet im Verlauf der nächsten Kapitel – aber auch in diesem – sehen warum dies von Vorteil sein kann.

Als legen wir uns einen Order für unsere „ViewModel"-Klassen an. Ich nenne diese ViewModel-Klassen auch wenn es hier nicht um MVVM geht. Das eine hat mit dem anderen ja nicht viel zu tun. Jede Klasse welche mit einem View verbunden wird kann als ViewModel-Klasse bezeichnet werden.

Als erste Klasse lege ich eine Basisklasse an damit wir nicht für jede einzelne Klasse INotifyPropertyChanged implementieren müssen sondern eine Basisklasse haben von welcher wir immer wieder erben können. Das spart uns viel Tipparbeit. Die ViewModel-Klasse packe ich alle in einen seperaten Namespace „ViewModel".

## 3

```
Namespace ViewModel
    Public MustInherit Class ViewModelBase
        Implements INotifyPropertyChanged

        Protected Overridable Sub RaisePropertyChanged(<CallerMemberName> Optional
prop As String = "")
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(prop))
        End Sub

        Public Event PropertyChanged As PropertyChangedEventHandler Implements
INotifyPropertyChanged.PropertyChanged
    End Class
End Namespace
```

Nun können wir bereits beginnen die erste ViewModel-Klasse anzulegen. Die ContactViewModel-Klasse. Diese Klasse hat die Aufgabe die Eigenschaften der Model-Klasse „Contact“ für die View aufzubereiten. Wie schon erwähnt machen es viele so das sie INotifyPropertyChanged in der Modelklasse implementieren um sich so diese Zwischenklasse zu „sparen“. Das ist natürlich ein Weg den man gehen kann um dem *dont repeat yourself* Grundsatz zu folgen, ich habe aber eben die Erfahrung gemacht das sich diese zusätzlicher Arbeit durchaus auszahlt und sogar eher komplexität herausnimmt – auch wenn dies auf den ersten Blick nicht so aussieht - speziell auf Hinblick auf späteres MVVM, aber das ist ein späteres Thema, ihr werdet auch in diesem Kapitel bereits sehen das dies von Vorteil sein kann.

Erstellen wir erstmal eine Klasse, ich erkläre hierzu gleich ein paar Dinge:

```
Namespace ViewModel
    Public Class ContactViewModel
        Inherits ViewModelBase

        Private ReadOnly _modelContact As Model.Contact

        Public Sub New()
            _modelContact = New Model.Contact
            ContactData = New ObservableCollection(Of ContactDataViewModel)
        End Sub

        Friend Sub New(model As Model.Contact)
            _modelContact = model
            ContactData = New ObservableCollection(Of ContactDataViewModel)
            model.ContactData.ForEach(Sub(x) ContactData.Add(New ContactDataViewModel(x)))
        End Sub

        Public Property FullName() As String
            Get
                Return $"{_modelContact.FirstName} {_modelContact.LastName}"
            End Get
            Set(ByVal value As String)
                Dim fullname = value.Split(" ")
                _modelContact.FirstName = fullname.First
                If fullname.Length > 1 Then _modelContact.LastName = fullname.Last
                RaisePropertyChanged()
            End Set
        End Property

        Public Property ImagePath As String
            Get
                Return _modelContact.ImagePath
            End Get
            Set(value As String)
                _modelContact.ImagePath = value
                RaisePropertyChanged()
            End Set
        End Property
        Public Property Birthday As Date?
            Get
                Return _modelContact.Birthday
            End Get
            Set(value As Date?)
                _modelContact.Birthday = value
                RaisePropertyChanged()
            End Set
        End Property
        Public Property Note As String
            Get
                Return _modelContact.Note
            End Get
            Set(value As String)
                _modelContact.Note = value
                RaisePropertyChanged()
            End Set
        End Property

        Private _contactData As ObservableCollection(Of ContactDataViewModel)
        Public Property ContactData() As ObservableCollection(Of ContactDataViewModel)
            Get
                Return _contactData
            End Get
            Set(ByVal value As ObservableCollection(Of ContactDataViewModel))
                _contactData = value
                RaisePropertyChanged()
            End Set
        End Property

    End Class
End Namespace
```

OK, das sieht erstmal verwirrend aus. Gebe ich zu.
Fangen wir oben an. Wir haben eine Private Schreibgeschützte Variable vom Typ Model.Contact.
Diese Variable dient dazu unser „Modelobjekt“ zu halten. Wie an den beiden Konstuktoren zu sehen, wird entweder eine neue Instanz eines Contact erstellt falls beim Instaziieren unseres ViewModels keine Instanz hereingereicht wird, oder es wird die Instanz an die Variable übergeben. Wir haben somit immer ein Model-Object in dieser Variable.

In späterer folge haben wir dann eine Eigenschaft FullName. Hier ist schon mal ein Fall für den eine ViewModel Klasse eben praktisch ist. In unserem Model ist definiert das wir FirstName und LastName getrennt speichern werden. Wie Ihr es vom Handy evtl. kennt kann man einen neuen Kontakt aber meißt vereinfacht eingeben indem man einfach den Vor und den Zunamen in einer Zeile eingibt. Tja, unser Telefonbuch soll dies auch können, gespeichert soll aber trotzdem getrennt werden. Also wenn der User „Sascha Patschka“ in das Feld eingibt soll unsere App hier „Sascha“ als Vornamen und „Patschka“ als Nachnamen speichern. Das ist bereits der Vorteil der Trennung zwischen einem *ViewModel* und einem *Model*. Wir können frei bestimmen und sind flexibel wie die View aussehen soll bzw. wie Daten sowohl angezeigt als auch eingegeben werden können.

Im Getter der Eigenschaft setzen wir die Model-Eigenschaften FirstName und LastName zusammen und geben diese mit einem Leehrzeichen getrennt aus.
Im Setter lesen wir den String aus und versuchen diesen zu Splitten und anschliessend getrennt an das Modelobjekt zu übergeben . Die Art wie dies geschied ist im Moment nur sehr rudimentär implementiert und hat noch so einige Fehler, reicht uns aber fürs erste um einfach hier nur eine Idee zu bekommen was wir hier vorhaben.

Nun sehen wir uns die nächsten Eigenschaften an. Diese reichen im Grunde einfach nur durch.

```
Public Property ImagePath As String
     Get
         Return _modelContact.ImagePath
     End Get
     Set(value As String)
        _modelContact.ImagePath = value
        RaisePropertyChanged()
    End Set
End Property
```

Im Getter wir der ImagePath des ModelObjekts einfach durchgereicht und im Setter wird ImagePath des Modelobjekts gesetzt. Man kann dies als Wrapper bezeichnen.

Im Grunde geht es einfach darum eine Eigenschaft zu haben welche nach dem setzen des Wertes das Event PropertyChanged wirft und das macht unsere Eigenschaft im Setter mit der Methode RaisePropertyChanged().
Kommen wir zur letzten Eigenschaft „ContactData“ vom Typ ObservableCollection(Of ContactDataViewModel).
Diese in Verbindung mit der dritten Zeile des Konstruktors ist etwas speziell und kann vielleicht verwirrend sein. Aber keine Sorge, wir gehen das langsam durch.

Erstmal benötigen wir den Typ ContactDataViewModel.

```
Namespace ViewModel
    Public Class ContactDataViewModel
        Inherits ViewModelBase

        Private ReadOnly _modelContactData As ContactData

        Public Sub New()
            _modelContactData = New ContactData()
        End Sub

        Friend Sub New(model As Model.ContactData)
            _modelContactData = model
        End Sub

        Public Property Data As String
            Get
                Return _modelContactData.Data
            End Get
            Set(value As String)
                _modelContactData.Data = value
                RaisePropertyChanged()
            End Set
        End Property
        Public Property DataType As EnumContactDataType
            Get
                Return _modelContactData.DataType
            End Get
            Set(value As EnumContactDataType)
                _modelContactData.DataType = value
                RaisePropertyChanged()
            End Set
        End Property

    End Class
End Namespace
```

Diese Klasse sollte soweit verständlich sein, auch diese dient als „Wrapper“ für ein Model-Objekt.

Zurück zur ContactViewModel-Klasse…
Wir bekommen ja unser Model-Objekt in die private Variable. Um nun die Auflistung an ContactData des Model-Objekts in die Eigenschaft ContactData des ViewModels zu bekommen können wir diesmal nicht einfach „durchreichen“ da die Typen sich unterschieden. Wir müssen

die Typen also konvertieren. Das machen wir am einfachsten indem wir in einer Schleife die Auflistung durchgehen.

Da wir in der Klasse ContactDataViewModel einen Konstruktor haben welchem wir ein Model-Objekt übergeben können ist dies auch einfach. Wir erstellen schlicht für jedes Objekt in der Auflistung des Model-Objekts eine neue Instanz von ContactDataViewModel und verwenden den Konstruktor mit der Überladung welcher wir ein Modelobjekt mitgeben können was in der dritten Zeile des Konstruktors von ContactViewModel passiert.

Schon haben wir dafür gesorgt dass die Auflistung mit Daten gefüllt ist.

Nun muss nur noch schnell ein View her um das mal zu testen.
Wir öffnen die CodeBehind des MainWindow und erstellen erstmal eine Eigenschaft welche eine Hand voll Testdaten halten soll. Eine ObservableCollection(Of ContactViewModel)

```
Class MainWindow

    Private _allContacts As ObservableCollection(Of ContactViewModel)
    Public Property AllContacts() As ObservableCollection(Of ContactViewModel)
        Get
            Return _allContacts
        End Get
        Set(ByVal value As ObservableCollection(Of ContactViewModel))
            _allContacts = value
        End Set
    End Property

End Class
```

OK, aber wir müssen die WPF für Binding wieder Benachrichtigen. Die Basisklasse für unsere ViewModels können wir bei einem Window aber nicht erben, wir müssen hier also INotifyPropertyChanged implementieren.

```
Class MainWindow
    Implements INotifyPropertyChanged

    Private _allContacts As ObservableCollection(Of ContactViewModel)
    Public Property AllContacts() As ObservableCollection(Of ContactViewModel)
        Get
            Return _allContacts
        End Get
        Set(ByVal value As ObservableCollection(Of ContactViewModel))
            _allContacts = value
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(NameOf(AllContacts)))
        End Set
    End Property


    Public Event PropertyChanged As PropertyChangedEventHandler Implements
INotifyPropertyChanged.PropertyChanged
End Class
```

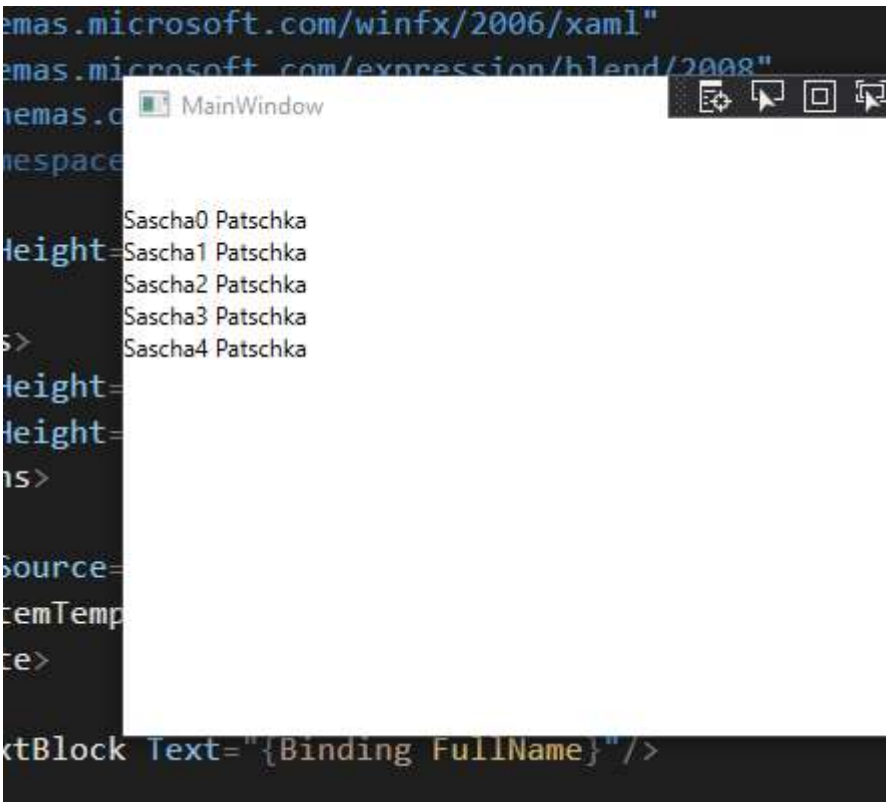
Nun Abonnieren wir das Loaded Event des Window um zum einen ein paar Testdaten zu erstellen und zum anderen die View an die CodeBehind zu Binden.

```
Class MainWindow
    Implements INotifyPropertyChanged

    Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles Me.Loaded
        AllContacts = New ObservableCollection(Of ContactViewModel)

        For i As Integer = 0 To 4
            AllContacts.Add(New ContactViewModel(New Model.Contact() With {.FirstName = "Sascha" & i,
.LastName = "Patschka",
                                                 .Birthday = New Date(1983, 9, 12), .Note = "Eine
Notiz", .ContactData = New List(Of Model.ContactData) From {
                                                   New Model.ContactData() With {.Data = "+43 664
1234567", .DataType = Model.EnumContactDataType.Phonenumber}})))
        Next

        Me.DataContext = Me
    End Sub


    Private _allContacts As ObservableCollection(Of ContactViewModel)
    Public Property AllContacts() As ObservableCollection(Of ContactViewModel)
        Get
            Return _allContacts
        End Get
        Set(ByVal value As ObservableCollection(Of ContactViewModel))
            _allContacts = value
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(NameOf(AllContacts)))
        End Set
    End Property


    Public Event PropertyChanged As PropertyChangedEventHandler Implements
INotifyPropertyChanged.PropertyChanged
End Class
```

Wir füllen also AllContacts mit Testdaten in einer Schleife. Nun können wir versuchen diese Testdaten mal in einem View anzuzeigen.

```xaml
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:_4_2_PhoneBook"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="47*"/>
            <RowDefinition Height="372*"/>
        </Grid.RowDefinitions>

        <ItemsControl ItemsSource="{Binding AllContacts}" Grid.Row="1">
            <ItemsControl.ItemTemplate>
                <DataTemplate>

                        <TextBlock Text="{Binding FullName}"/>

                </DataTemplate>
            </ItemsControl.ItemTemplate>
        </ItemsControl>


    </Grid>
</Window>
```

Na das sieht doch gut aus. Nicht von der Optik her, sondern das wir korrekt gebunden haben und unser Code bis hierher erstmal funktioniert.



Hier geht's zum Video: https://youtu.be/t-0bwri7wVk

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

## 4.3

# Die erste View besser strukturieren

Nun werden wir ein wenig Ordnung in den XAML reinbringen. Ich habe abseites des Tutorials ein wenig die Optik der kleinen App verbessert, nichts besonderes aber ein wenig XAML ist entstanden. Gewisse Dinge wurden in Resourcen ausgelagert und eingebunden aber auch DataTemplates wurden erstellt.

Diesen XAML Code werden wir besser Strukturieren indem wir UserControls erstellen um den Code auszulagern. Ziel ist es in der MailWindow etwas Odnung zu schaffen.

Da dies nun nicht wirklich Codearbeit ist oder mit Binding sehr viel zu tun hat ist dies hier in Textform etwas schwer zu zeigen weshalb ich mich für dieses Kapitel auf das Video beschränken werde. Ich hoffe ich könnt mir das verzeihen.

Hier geht's zum Video: https://youtu.be/OdRGPPofn64

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

## 4.4

# Filtermöglichkeit einbinden (oder doch mehr?)

Wir werden nun eine Filtermöglichkeit für die Kontakte einbinden. Der Benutzer soll ja schließlich nach einem Kontakt suchen können. Ideal wäre hier wenn der Benutzer nach dem Nachnamen, dem Vornamen oder den Informationen in den Notizen suchen könnte.

Genau das implementieren wir nun. Wir haben ja bereits in Kapitel 4.1.4.6 gelernt das uns hier die ICollectionView zur verfügung steht um zu Filtern und zu Sortieren.

Wir werden also unsere MainWindow-CodeBehind anpassen und so umbauen das wir eine ICollectionView haben auf welche wir Binden können.

Als erstes erstellen wir uns diese und geben unsere bereits vorhandene ObservableCollection als Datenquelle an.

```
AllContactsView = CollectionViewSource.GetDefaultView(AllContacts)
AllContactsView.Filter = AddressOf Contacts_Filter
```

Wir haben nun auch gleich eine Filtermethode angegeben wie in Kapitel 4.1.4.6 ja bereits gelernt.

In  dieser Methode welche ein Boolean als Rückgabewert besitzt, soll nun für jedes Objekt True zurückgeben in welchem der Name oder die Notiz den eingegebenen Filtertext enthält. Aber hierfür benötigt es noch ein Property für den Filterstring. An diese Eigenschaft binden wir in unserem UserControl uclMainHeader die dafür vorgesehene Textbox. Wir dürfen aber nicht vergessen im Setter dann die CollectionView zu aktualisieren.

```
Private _filterString As String = ""
Public Property FilterString() As String
    Get
        Return _filterString
    End Get
    Set(ByVal value As String)
        _filterString = value
       AllContactsView.Refresh()
        RaiseEvent PropertyChanged(Me, New
PropertyChangedEventArgs(NameOf(FilterString)))
    End Set
End Property
```

```
TextBox x:Name="txtSearch" Text="{Binding
FilterString,UpdateSourceTrigger=PropertyChanged}"
                 FontSize="{StaticResource DefaultFontSize}"
VerticalContentAlignment="Center" BorderThickness="0"/>
```

Wir ihr sehen könnt habe ich zusätzlich beim Binding den UpdateSourceTrigger auf PropertyChanged gesetzt damit im Code bei jedem Tastenanschlag der Setter durchlaufen wird und wir darauf reagieren können. So ist die Suche für den User weitaus angenehmer.

Nun fehlt uns nur noch die Methode für den eigentlichen Filter zu erstellen.

```
Private Function Contacts_Filter(obj As Object) As Boolean
    Dim c As ContactViewModel = CType(obj, ContactViewModel)
    Return c.FullName.ToLower.Contains(FilterString.ToLower()) Or
c.Note.ToLower.Contains(FilterString.ToLower())
End Function
```

Nichts wildes, aber effektiv. Jetzt noch das Binding des ItemsControl anpassen da wir nun nicht mehr an die ObservableCollection sondern an unsere neue CollectionView binden möchten.

```
<ItemsControl ItemsSource="{Binding AllContactsView}" Grid.Row="1">
```

Starten wir nun die App können wir bereits Filtern wie es uns gefällt.


Was wir auch noch möchten ist, mit dem X das Suchfeld zu leeren. OK, wir müssen also im Grunde nur die Eigenschaft FilterString auf einen Leerstring setzen. Gut. Reagieren wir mal auf einen Klick auf das X.

In unserem *UserControl* haben wir ein *ContentControl*. Wenn wir dieses Klicken soll der Text in der *Textbox* geleert werden. Wer schon ein paar Dinge mit der WPF gemacht hat wird bereits wissen das wenn wir nun auf das *ContentControl* welches als Content einen Path besitzt einen Event verarbeiten wollen ist das unschön. Warum? Der User müsste wirklich genau auf das X klicken. Weil nur wenn die Maus genau über einem Path ist – also genau nur im schwarzen Teil – wird das *Event* verarbeitet. Das ist von der WPF aber so gewollt und ist bei einige Szenarien auch recht wichtig. In unserem Fall möchten wir das in diesem Beispiel aber nicht weshalb wir uns einen Border um das *ContentContol* machen und dessen Background explizit auf Transparent setzen. Tun wir das nicht ist der *Border* erst gar nicht klickbar. In dem *Border* können wir nun das MouserLeftButtonDown Event abonnieren und mit *F12* in die CodeBehind des UserControls springen.

Unser *DockPanel* sieht nun wie folgt aus:

```
<DockPanel LastChildFill="True">
    <ContentControl Margin="5" DockPanel.Dock="Left"
              Content="{StaticResource IconMagnify}"/>
       <Border DockPanel.Dock="Right" MouseLeftButtonDown="Cross_MouseLeftButtonDown"
       Background="Transparent">
        <ContentControl Margin="5" Content="{StaticResource IconCross}"/>
    </Border>
    <TextBox x:Name="txtSearch"
       Text="{Binding FilterString,UpdateSourceTrigger=PropertyChanged}"
       FontSize="{StaticResource DefaultFontSize}" VerticalContentAlignment="Center"
       BorderThickness="0"/>
</DockPanel>
```

In unserer *CodeBehind* haben wir nun folgende Prozedur:

```
Private Sub Cross_MouseLeftButtonDown(sender As Object, e As MouseButtonEventArgs)


End Sub
```

So. Und was jetzt? Wir sind ja in einem *UserControl*. Nicht in dem *MainWindow*. Haben wir uns nun ein Bein gestellt indem wir den XAML ausgelagert haben? Wir können nun die Eigenschaft FilterString in der *CodeBehind* der *MainWindow* ja gar nicht setzen? OK, dann geben wir der TextBox einen Namen und setzen die Text-Eigenschaft dieser TextBox einfach auf einen Nullstring. **Halt, Stopp.**

Nein, wir wollen mit Binding arbeiten. Also fangen wir uns das erst gar nicht an.
Wir wissen dass dieses *UserControl* auf jeden Fall als Datenkontext eine *MainWindow*-Instanz besitzt. Also werden wir uns diese Instanz holen. Und schon haben wir Zugriff.

```
Dim mw As MainWindow = CType(Me.DataContext, MainWindow)
mw.FilterString = ""
```

Nun funktioniert bereits das Filtern und das leeren des Filters. Super.

Aber wäre es nicht toll nun noch eine Gruppierung zu haben. Bei Handys ist es ja so dass man immer beim Scrollen den Anfangsbuchstaben sieht bei welchem man gerade ist. Also wenn man gerade bei allen Namen welche mit *B* beginnen vorbei ist das dann eine Art Überschrift mit einem *C* kommt. Moment, dafür bietet sich ja eine Gruppierung in Verbindung mit einem Group-Style für das *ItemsControl* an. Gut, legen wir uns einen Style an:

```
<ItemsControl ItemsSource="{Binding AllContacts}" Grid.Row="1">
   <ItemsControl.ItemTemplate>
       <DataTemplate>
           <local:uclContactListItemTemplate ContactClicked="ContactItem_Click"/>
       </DataTemplate>
    </ItemsControl.ItemTemplate>
    <ItemsControl.GroupStyle>
        <GroupStyle>
           <GroupStyle.ContainerStyle>
               <Style TargetType="GroupItem">
                    <Setter Property="Template">
                        <Setter.Value>
                            <ControlTemplate TargetType="GroupItem">
                                <GroupBox Header="{Binding Name}" >
                                    <ItemsPresenter />
                                </GroupBox>
                            </ControlTemplate>
                        </Setter.Value>
                    </Setter>
                </Style>
            </GroupStyle.ContainerStyle>
        </GroupStyle>
    </ItemsControl.GroupStyle>
</ItemsControl>
```

Jetzt nur noch wie wir es gelernt haben eine GroupDescription mit einer PropertyGroupDescription für unsere *CollectionView* erstellen und schon haben wir Gruppiert.

Aber für welche Eigenschaft? Wir haben keine Eigenschaft zum Gruppieren nach Anfangsbuchstaben. Wir erstellen uns also eine schreibgeschützte Eigenschaft in unserem ContactViewModel.

```
Public ReadOnly Property FirstLastNameLetter As Char
    Get
        Return FullName.First()
    End Get
End Property
```

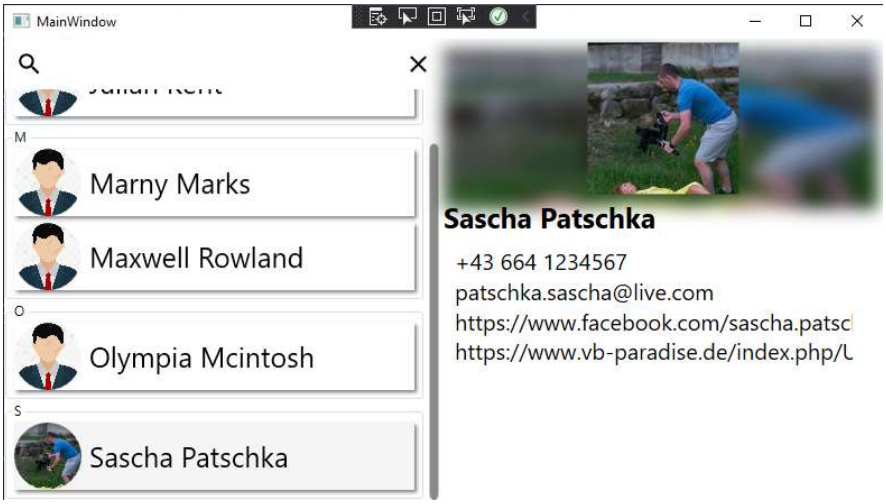
Da die WPF aber auch für schreibgeschützte Eigenschaften über Änderungen benachrichtigt werden will tun wir das in der Eigenschaft FullName.

```
Public Property FullName() As String
    Get
        Return $"{_modelContact.FirstName} {_modelContact.LastName}"
    End Get
    Set(ByVal value As String)
        Dim fullname = value.Split(CChar(" "))
        _modelContact.FirstName = fullname.First
        If fullname.Length > 1 Then _modelContact.LastName = fullname.Last
        RaisePropertyChanged()
        RaisePropertyChanged(NameOf(FirstLastNameLetter))
    End Set
End Property
```

Gut, jetzt können wir unsere PropertyGroupDescription in der MainWindow.vb einfügen. Die komplette MainWindow_Loaded sieht nun wie folgt aus:

```
Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles
Me.Loaded
    AllContacts = New ObservableCollection(Of ContactViewModel)

    Dim dp As New ContactsDataProvider()
    dp.GetAllContacts(True).ToList.ForEach((Sub(x) AllContacts.Add(New
ContactViewModel(x))))

    AllContactsView = CollectionViewSource.GetDefaultView(AllContacts)
    AllContactsView.Filter = AddressOf Contacts_Filter
    AllContactsView.GroupDescriptions.Add(New
PropertyGroupDescription("FirstLastNameLetter"))

    SelectedContact = AllContacts.FirstOrDefault()

    Me.DataContext = Me
End Sub
```

Das Ergebnis kann sich durchaus sehen lassen.




Natürlich zeige ich all das auch wieder im Video wobei ich hier noch auf ein paar kleine Details mehr eingehe. Viel Spaß damit.

Hier geht's zum Video: https://youtu.be/rmpb6fv4LNA

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

## 4.4

# Einträge bearbeiten und speichern

In diesem Kapitel möchten wir Einträge bearbeiten und speichern. Dies ist ein übliches Scenario in einer jeden CRUD Anwendung. Ich entscheide mich in diesem Fall Daten in einem Dialog zu bearbeiten.

***Info:*** *Im Video spreche ich davon das wir die Funktion Einträge bearbeiten zu können im nächsten Video noch ausbauen, ich habe mich aber dazu entschieden bereits etwas früher zum Thema MVVM zu springen. Dies wurde von einigen auch so gewünscht.*

Erst benötigen wir in unserem MainWindow einen Button welcher dann Dialog dann öffnet. Diesen definiere ich innerhalb der Grids und vergebe auch einen Handler „btnEditContact“ für das Click-Event:

```
<Button Content="Eintrag bearbeiten" Grid.Column="1" HorizontalAlignment="Right"
                VerticalAlignment="Top" Margin="5" Padding="5"
Click="btnEditContact"/>

Private Sub btnEditContact(sender As Object, e As RoutedEventArgs)
End Sub
```

Wir erinnern uns dass wir in der MainWindow_Loaded Methode sowohl alle Variablen und Eigenschaften initialisieren als auch die Daten laden. Den Code zum Laden der Daten müssen wir in eine neue Methode auslagern, denn wir möchten ja später sicher nach dem speichern der Daten auch diese wieder neu in die Liste laden, das geht zwar natürlich auch nur mit diesem einen Eintrag, aber der Einfachheit halber laden wir in diesem Beispiel einfach alles neu in die Liste.

***Tipp:*** *Wie man in Visual Studio sehr einfach über die Refactoring-Funktion Code auslagert seht ihr in dem Video zu diesem Kapitel.*

```
Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles
Me.Loaded
        AllContacts = New ObservableCollection(Of ContactViewModel)

        LoadData()
        Me.DataContext = Me
    End Sub

    Private Sub LoadData()
        _allContactsModels = dp.GetAllContacts(True).ToList()
        _allContactsModels.ForEach((Sub(x) AllContacts.Add(New ContactViewModel(x))))

        AllContactsView = CollectionViewSource.GetDefaultView(AllContacts)
        AllContactsView.Filter = AddressOf Contacts_Filter
        AllContactsView.GroupDescriptions.Add(New
PropertyGroupDescription("FirstLastNameLetter"))

        SelectedContact = AllContacts.FirstOrDefault()
    End Sub
```

Gut, in unserem Handler welcher den „Bearbeiten-Dialog“ öffnen soll müssen wir nun einen Dialog öffnen. Den bauen wir uns schnell mal. Ich baue gerne UserControls anstatt Fenster da ich damit flexibler bin wenn ich eine Komponente mal austauschen möchte.

```
<UserControl x:Class="uclEditContact"
...
             d:DesignHeight="450" d:DesignWidth="800" Width="400">
    <Grid Margin="10">
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition Height="50"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="20"/>
            <RowDefinition Height="30"/>
        </Grid.RowDefinitions>
        <Label FontSize="20" Grid.ColumnSpan="2">Eintrag bearbeiten</Label>
        <Label Grid.Row="1">Name</Label>
        <Label Grid.Row="2">Notiz</Label>
        <TextBox Grid.Row="1" Grid.Column="1" Text="{Binding FullName}"/>
        <TextBox Grid.Row="2" Grid.Column="1" Text="{Binding Note}"/>

        <Button Grid.Row="10" Grid.Column="1" Content="Speichern"
Click="btnSaveClick"></Button>
    </Grid>
</UserControl>
```

Wie Ihr seht, ist in diesem Dialog nicht viel Spezielles enthalten. Zwei TextBoxen und zwei dazugehörige Labels sowie ein Speicher-Button. Die TextBoxen sind mittels Binding gebunden. Aber an was sich diese gebunden? Woher weis die WPF das dieser Dialog einen Telefonbucheintrag bearbeitet?

Dazu kommen wir gleich, erstmal hauchen wir dem Speichern-Button leben ein indem wir für diesem einen Handler erstellen und folgendes in die Methode packen:

```
Private Sub btnSaveClick(sender As Object, e As RoutedEventArgs)
    Dim win = Window.GetWindow(Me)
    win.DialogResult = True
    win.Close()
End Sub
```

OK, ein wenig habe ich gelogen, es ist ja im Grunde gar kein Button welcher Speichert. Er schließt nur den Dialog, denn es ist einfacher wenn wir von „draußen“ speichern, also im MainWindow, dort haben wir all unsere Daten und können hier leichter speichern. Der Dialog soll und am besten ein DialogResult zurückgeben damit wir außen wissen ob der User auf Speichern geklickt hat oder ob er das Fenster (z.b. mit dem X) geschlossen hat.
Das machen wir mittels DialogResult.

Da es sich hier um ein UserControl handelt und nicht um ein Fenster können wir das Fenster in welchem es sich später befinden wird nicht einfach so aufrufen und dieses schließen.
Mit Window.GetWindow(Me) bekommen wir eine Methode des Frameworks zur Hand welches uns die Instanz des Fensters zurückgibt in welchem sich das Objekt befindet welcher wir unser UserControl als Parameter übergeben.
Wir setzen dann nur den DialogResult und schließen das Fenster.

Aber wie kommt nun die Bindung zustande? Das machen wir im selben Schritt in welchem wir auch das Fenster öffnen lassen. Und zwar in unserer Methode btnEditContact in unserem MainWindow-CodeBehind.

```
Private Sub btnEditContact(sender As Object, e As RoutedEventArgs)
    Dim dlgWin As New Window()
    dlgWin.SizeToContent = SizeToContent.WidthAndHeight
    dlgWin.Content = New uclEditContact()
    dlgWin.DataContext = SelectedContact
    If dlgWin.ShowDialog() Then
        dp.SaveContacts(_allContactsModels)
    Else
        AllContacts.Clear()
        LoadData()
    End If
End Sub
```

OK, was machen wir hier. Wir Instanziieren hier ein neues Window und setzen den Content. In diesem Fall unser soeben erstelltes UserControl. Den DatenContext des Window (welchen das UserControl wie wir ja wissen erbt) setzen wir auf den aktuell selektierten Eintrag unserer Auflistung. Durch die Bindung werden die Eingaben ja direkt Syncronisiert und wir müssen einfach nur die ganze Liste speichern falls der DialogResult = True ist. Falls dem nicht so ist leeren wir unsere Liste und laden diese neu von der Platte. Wie schon erwähnt ist das nicht der schönste Weg, aber für dieses Tutorial und um die Funktionsweise zu verstehen der einfachste, da wir uns mit komplizierten Lade und Speichermechanismen im Moment nicht aufhalten wollen. Das hält die Beispiele einfach und übersichtlich.

Wenn wir die Anwendung nun starten haben wir die neuen Funktionalitäten zur Verfügung und können Telefonbucheinträge bearbeiten und speichern.

Ich hoffe das war verständlich und wir haben nun genug Vorkenntnisse um zu einem viel spannenderen Thema kommen. Dem MVVM Pattern.

Hier geht's zum Video: https://youtu.be/Ragc3JIrGHM

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

4

# Das MVVM Pattern

Von vielen schon lange gewünscht kommen wir nun zum richtig spannenden Teil, dem MVVM Muster. Dieses Muster ist ein Pattern welches die Trennung zwischen Logik und UI ermöglicht wodurch viele Vorteile erlangt werden können, aber wie immer wenn etwas Vorteile hat, hat es auch Nachteile. Auch diese werde ich aufzeigen.

Aber was ist das MVVM Pattern genau? Im Video erkläre ich es genauer und erläutere einige Textpassagen des dazugehörigen Wikipedia Artikeln zum MVVM Pattern.

Kurz und knapp die Vorteile und Nachteile von welchen ich denke das sie für die meisten relevant sein dürften.

Vorteile:

- Die Geschäftslogik kann unabhängig von der Darstellung bearbeitet werden
- Die Testbarkeit verbessert sich, da die ViewModel die UI-Logiken enthalten und unabhängig von der View instanziiert werden können
- Views können von reinen UI-Designern gestaltet werden, während Entwickler unabhängig davon die Models und ViewModels implementieren

Nachteile:

- Für kleinere Projekte eher „overkill"
- Für Anfänger eher nicht geeignet da es schnell komplex wirken kann

Wie Datenbindung an sich und wie diese im MVVM Pattern funktioniert ist auch im verlinkten Wiki Artikel enthalten, ich gehe aber auch auf diesen Part im Video genauer ein, sorry das ich das hier nicht nochmals aufschreibe, ich denke was die Theorie angeht gibt es ja genug Artikel im Netz. Dennoch – im Video gibt's ein paar Zusatzinfos.

Hier geht's zum Video: https://youtu.be/Mw9CJJhpTuA

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

## 5.2

# Wann MVVM und wann nicht?

Das ist eine Frage die nicht so leicht zu beantworten ist. Zum einen sollte der eigene Wissensstand auf einem gewissen Level sein. Doch auf welchem.
Für MVVM ist einiges notwendig, hierfür sollten Dinge wie Vererbung, Interfaces, generische Klassen und XAML Binding schon recht gut sitzen. Ist dies nicht der Fall hält man sich gut und gerne Wochenlang auf wenn man auf ein Problem stößt. Denn die sind Techniken welche immer wieder zum Einsatz kommen und sollten schon sitzen.

Aber nicht nur das. Immer wieder lese ich in Foren dass man unter WPF unbedingt mit dem MVVM Pattern arbeiten muss. Muss? Why?
Das kann ich so nicht unterschreiben. Es kommt immer auf das Projekt an. Habe ich nur vor ein kleines „Helferlein" zu proggen und ich weis im Vorfeld das wird nur eine kleine App mit ein bis zwei Funktionen und evtl. zwei bis drei Fenstern, warum soll ich mir dann diesen Overhead antun?

Solche Anwendungen sind sicher nicht so komplex das ich von den Vorteilen des Patterns profitieren würde, und die Testbarkeit ist bei solchen Apps auch irrelevant.
Habe ich allerding vor eine App zu schreiben welche evtl. in Zukunft wächst, wo sich Abhängigkeiten in Zukunft ändern, wo ich im Vorfeld weis das die Anwendung größer wird und die Komplexität immer weiter steigen wird, dann macht MVVM durchaus Sinn da uns das Pattern

einfach in gewisser Weise auch zwingt eine gewisse Struktur einzuhalten und somit die Komplexität mit wachsender Anwendung immer weiter fällt.

Das hört sich komisch an aber je größer die Anwendung, desto weniger komplex wird diese durch das Pattern.
Anfangs wirkt eine Anwendung mit MVVM sehr wirr, sehr viele Projekte, sehr viele Klassen, Unmengen an Namespaces und alles befindet sich wo anders. Brainfuck? Zum Teil ja – Anfangs!

Mit zunehmender Anzahl an Logikklassen und immer wachsender Anwendung freut man sich immer mehr über die Trennung und findet alles sofort, denn ohne diese „Ordnung“ hat man irgendwann keinen Überblick mehr. Zugegeben, hält man Ordnung ist alles gut, aber mal ehrlich, wer hält wirklich immer Ordnung, wer weis schon immer genau welcher Code und welche Logik wo genau ist? Bei MVVM stellt sich die Frage oft erst gar nicht.

Wann und wo Ihr MVVM einsetzt entscheidet Ihr. Unsicher? Frag mich einfach, evtl. kann ich dir weiterhelfen.

Hier geht’s zum Video: https://youtu.be/wjKwCn0Myqo

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

## 5.3

# Welchen Mehrwert kann ich aus dem Pattern gewinnen?

Ja, für viele stellt sich die Frage was man davon hat die durchaus vorhandenen Hürden auf sich zu nehmen und jede Einzelne versuchen zu meistern. Wenn man von den bereits angesprochenen Vorteilen im letzten Kapitel mal absieht stellt sich die Frage was man davon hat.

Die wenigsten welche mit MVVM anfangen werden sicher gleich Unittests schreiben, was ja eines der großen Vorteile sind wenn man das MVVM-Pattern korrekt anwendet. Was habe ich noch davon? OK, Designtime-Support ist ja ganz nett und lustig, aber auch das wäre den Aufwand nicht Wert. Bessere und saubere Struktur bei größeren Projekten und den damit verbundenen geringerem Wartungsaufwand ist ein schon recht guter Grund, aber jetzt auch nicht der überhammer.
Wie wäre es dann mit der Austauschbarkeit von Modulen? Also von Teilen wie z.b. der View welche man komplett tauschen könnte ohne etwas am Code ändern zu müssen.
Wenn man dies nicht vor hat auch kein sonderlich guter Grund. Jaaa, aber ein Designer könnte die View ja auch unabhängig vom Entwickler erstellen, warten und testen. Gut, das betrifft nur Teams.
OK, ich habe einen Grund. Weil alles davon reden und überall steht das man MVVM machen soll? Haha, sicher nicht.

Aber was ist nun ein guter Grund? Die Antwort ist so schlicht. All diese kleinen Gründe zusammen ergeben einfach ein „Rundum Paket“ das es uns ermöglicht eine saubere, gut gewartete und getestete Anwendung sowohl als alleiniger Entwickler, als auch im Team zu erstellen. Und das ist es im Endeffekt was wir wollen.

Hier geht’s zum Video: https://youtu.be/dSNQkO4qvRI

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

## 5.4

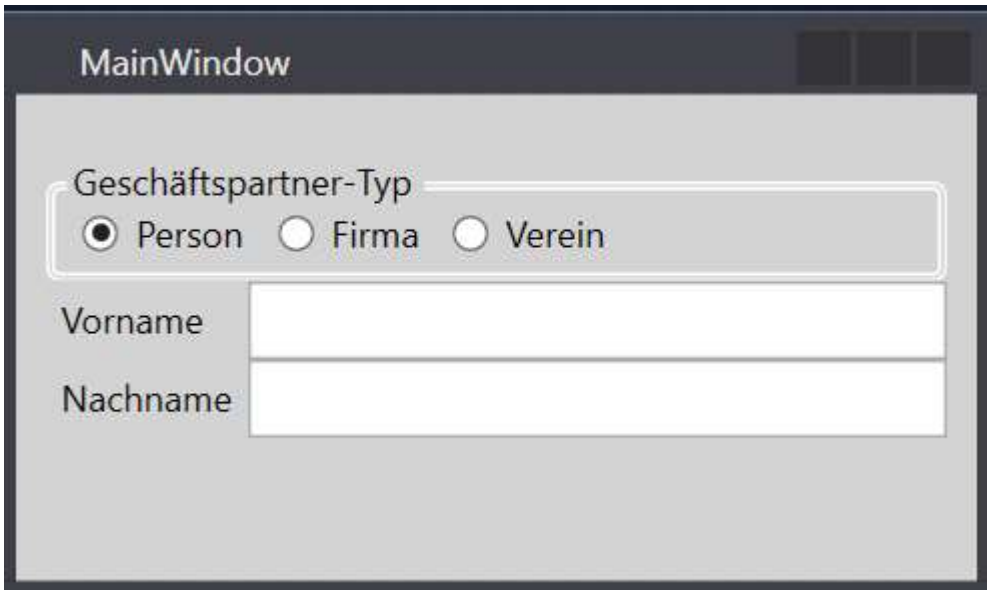
# MVVM und CodeBehind – Verboten?

Das ist eine sehr stark diskutierte Frage und in diversen Foren ist immer die rede davon das bei MVVM CodeBehind verboten ist. Meine Meinung: Nein, das stimmt so nicht!

Erstmal muss klar definiert werden um welchen Code es sich handelt. Es ist ganz klar, dass Code, welcher die Logik oder Daten betrifft nicht in die UI Schicht gehört. Aber Code welche rein die UI betrifft kann sehr wohl in die Code-Behind Datei.  Denn Code, welcher die Logik betrifft ist somit nicht mehr Testbar und auch eine unabhängige Entwicklung wäre damit nur noch sehr schwer

machbar. Betrifft der Code wirklich nur die UI oder deren Manipulation gilt dennoch zu beachten das, wenn eine View „ausgetauscht“ wird, dieser Code auch migriert werden muss. Demnach sollte man immer versuchen alternative Wege zu gehen und ggf. ein AttachedProperty für solch einen Fall zu schreiben, denn so kann die View einfachst ausgetauscht und wiederverwendet werden. In den meisten Fällen kann ein AttachedProperty oder Trigger schon helfen die Code-Behind Datei zu umgehen es gibt aber immer wieder Fälle, in denen das nicht möglich oder nur sehr umständlich möglich wäre.

Exemplarisch sehen wir hier folgende Ausgangssituation:



```
<Grid Margin="10">
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="Auto"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition Height="10"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="Auto"/>
        <RowDefinition Height="*"/>
    </Grid.RowDefinitions>
    <GroupBox Header="Geschäftspartner-Typ" Grid.ColumnSpan="2" Grid.Row="1">
        <StackPanel Orientation="Horizontal">
            <RadioButton Name="radioPerson" Content="Person" Margin="5,2"
Checked="radioChecked" IsChecked="True"/>
            <RadioButton Name="radioCompany" Content="Firma" Margin="5,2"
Checked="radioChecked"/>
            <RadioButton Name="radioClub" Content="Verein" Margin="5,2"
Checked="radioChecked"/>
        </StackPanel>
    </GroupBox>
    <Label Name="lblFirstName" Grid.Row="2">Vorname</Label>
    <Label Name="lblLastName" Grid.Row="3">Nachname</Label>
    <TextBox Name="txtFirstName" Grid.Row="2" Grid.Column="1" Text="{Binding
FirstName}"/>
    <TextBox Name="txtLastName" Grid.Row="3" Grid.Column="1" Text="{Binding
LastName}"/>
</Grid>
```

Angenommen wir möchten das ausschließlich bei einer Person beide Felder (Vor und Nachname) sichtbar sind, im Falle einer Firma jedoch nur ein Feld und die Beschriftung soll in diesem Fall auf „Name“ anstatt auf „Vorname“ geändert werden. Zudem soll die aktuelle Auswahl gespeichert werden, wählt der Benutzer also eine Firma soll in der Datenbank gespeichert werden das es sich um eine Firma handelt.

**Hier mal ein Beispiel was NICHT in die Code-Behind soll:**

```
Private Sub radioChecked(sender As Object, e As RoutedEventArgs)
    If DirectCast(sender, RadioButton).Name = "radioPerson" Then
        Dim currentViewModel As PersonViewModel = Me.DataContext
        currentViewModel.BusinessPartnerType = BpType.Person
    End If
End Sub
```

**Hier ein Beispiel was gerne in die Code-Behind darf:**

```
Private Sub radioChecked(sender As Object, e As RoutedEventArgs)
  If txtFirstName IsNot Nothing AndAlso lblFirstName IsNot Nothing Then
       Dim personSelected = DirectCast(sender, RadioButton).Name =
"radioPerson"

       txtFirstName.Visibility = If(personSelected, Visibility.Visible,
Visibility.Collapsed)
       lblFirstName.Visibility = txtFirstName.Visibility
       lblFirstName.Content = If(personSelected, "Vorname", "Name")
    End If
End Sub
```

Beim ersten Beispiel haben wir die Daten verändert. Das ist Aufgabe des ViewModels, ob dieses weiter delegiert zu einer Businesslogik, einem Repository oder ob die Logik im Model-Layer ist, ist egal, wichtig ist das es nicht über die View passiert.

Beim zweiten Beispiel betrifft der Code rein die View, es werden nur Controls Manipuliert, hätten die Textboxen oder die Radiobuttons nun ein Binding auf eine ViewModel Klasse hätte dies keinerlei Auswirkungen auf die Daten. Wir setzen lediglich die Visibility von Controls bzw. die Beschriftung eines Labels.

Ich hoffe es wir so etwas klarer was quasi mehr oder weniger „erlaubt“ ist. Natürlich gibt es auch genug Ansätze gewisse Daten an von der View aus zu manipulieren, damit nimmt man sich allerdings gewisse Vorteile und eben die lose Kopplung, mal ganz abgesehen davon das diese Logik nicht mehr testbar ist.

Hier geht’s zum Video: https://youtu.be/p7P2QCgx44M

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

## 5.5

# Model – View – ViewModel | Wars das?

Jetzt haben wir ja verinnerlicht das wir hier drei Layer haben welche entkoppelt sind und mehr oder weniger unabhängig sind. Wir wissen dass unsere Logik im ViewModel abläuft. Aber ist das gut?

In kleineren Projekten ist dies noch recht übersichtlich, das MVVM Pattern gibt und aber durch diese Layertrennung auch die Möglichkeit das wir hier einfach noch einen Layer vor das ViewModel schieben wie z.b. eine Businesslogic. Also ein Layer welcher sich um die Logik kümmert, so müssen wir im ViewModel alles nur noch so aufbereiten wie wir es für die Anzeige benötigen. Das macht das ganze bei größeren Projekten nochmals um einiges übersichtlicher.

Lasst uns das Anhand eines Beispiels durchgehen.

Mal angenommen wir möchten dem User die Möglichkeit geben das dieser das Passwort ändert. Natürlich geben wir gewisse Richtlinien vor wie z.b. das ein Kennwort eine Mindestlänge besitzen muss, das gleiche Kennwort nicht nochmals verwendet werden darf, evtl. dürfen die letzten X Kennwörter nicht verwendet werden, es muss x Sonderzeichnen enthalten, darf Vor- und Zunahme nicht enthalten usw.

Hier gibt es sehr viele Möglichkeiten. Solche eine Logik kann schnell mal komplex werden.

Hier mal ein kleines Beispiel mit einem Pseudocode:

```
Public Function ChangePassword(currentUser As User, oldPassword As String,
newPassword As String)
        Dim errorMessage As String = Nothing
        'Altes Passwort korrekt?
        If CryptPassword(oldPassword) <> db.Users.Where(Function(x) x.Username
= currentUser.Username).Password Then
            errorMessage = "Das alte Passwort war nicht korrekt!"
            Return errorMessage
        End If
        'Neues Passwort gleich wie altes
        If CryptPassword(newPassword) = CryptPassword(oldPassword) Then
            errorMessage = "Das alte Passwort darf nicht gleich wie das neue
sein"
            Return errorMessage
        End If
        'Wieviele der letzten Passwörter dürfen laut Richtlinie nicht verwendet
werden
        Dim numberOfVorbittenPasswordsFromHistory As Integer =
CInt(settings.GetSettingValue("NumberVorbittenPasswords").Value)
        Dim vorbittenPasswords As List(Of String) = db.Users.Where(Function(x)
x.Username =
currentUser.Username).Passwordhistory.OrderByDescending(Function(x)
x.CreationTimestamp).Take(10).Select(Function(x) x.Password).Tolist()
        If vorbittenPasswords.Any(Function(x) CryptPassword(newPassword)) Then
            errorMessage = $"Das neue Passwort darf nicht gleich mit einem der
letzten {numberOfVorbittenPasswordsFromHistory} Passwörtern sein."
            Return errorMessage
        End If
        'Muss passwort Sonderzeichen enthalten laut Richtlinie
        Dim passwordMustHaveSpecialCharacter As Boolean =
CBool(settings.GetSettingValue("NumberVorbittenPasswords").Value)
        If passwordMustHaveSpecialCharacter AndAlso Not
ContainsSpecialChars(newPassword) Then
            errorMessage = "Das Passwort entspricht nicht den Richtlinien das
ein Passwort Sonderzeichen enthalten muss"
            Return errorMessage
        End If
        'Mindestpasswortlänge
        Dim minimumPasswortLength As Integer =
CInt(settings.GetSettingValue("MinimumPasswortLength").Value)
        If newPassword.Length < minimumPasswortLength Then
            errorMessage = $"Das neue Passwort muss mindestens
{minimumPasswortLength} Zeichen lang sein"
            Return errorMessage
        End If
        'Muss Mindestens x Buchstaben enthalten

        'Muss Mindestens x Ziffern enthalten

        'Darf nicht gleich dem Usernamen sein

        'Darf keine Wörter der Blacklist enthalten (Firmennamen, Vornamen,
```

Packen wir dies alles in unsere ViewModel-Klasse wird schnell ungemütlich. OK, jetzt würden wir Anfangen das wir eine Helper-Klasse erstellen usw.

Was aber wenn wir einen eigenen Layer hätten in dem wir das alles haben? Wäre doch klasse. Kein Problem, geht ganz Easy und werden wir in späteren Kapiteln auch machen, ich möchte nur das ich jetzt schon wisst das ihr nicht alle in die ViewModel-Klassen packen müsst.

Wer Lust hat kann sich das WPFNote2 Projekt von mir auf vb-paradise.de ansehen. In diesem habe ich unter anderem eine Businesslogik implementiert.

Im Video gehe ich auch noch etwas tiefer.

Hier geht's zum Video: https://youtu.be/EvQZI3xW5CI

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

## 5.6

# Erstellen einer korrekten MVVM - Projektmappe unter Visual Studio

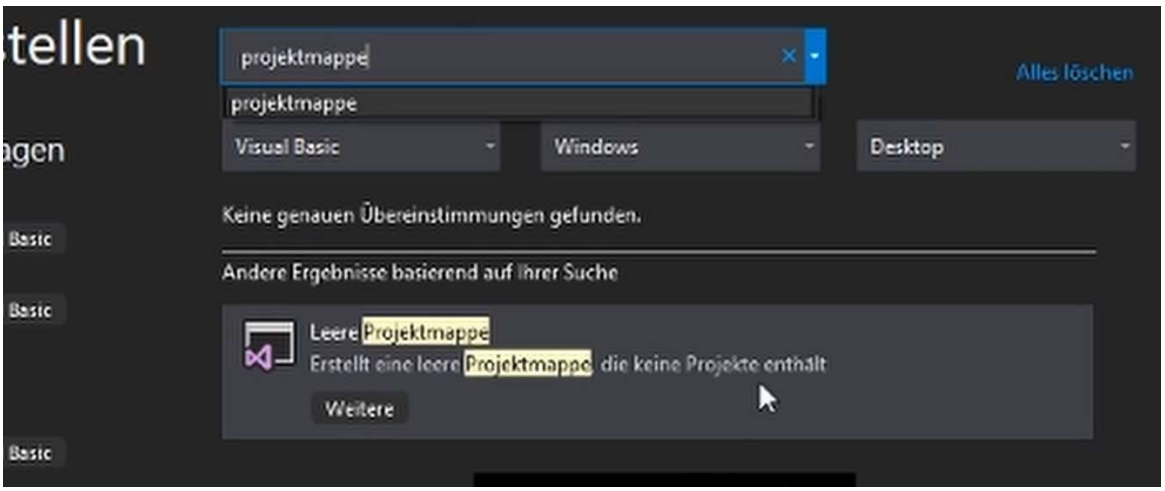
Wir erstellen nun eine Projektmappe unter Visual Studio welche „MVVM Gerecht" sein soll. Das bedeutet das alle Layer (Model – View – ViewModel) in getrennte Assemblys verteilt werden.

Warum dies wichtig ist habe ich zwar bereits erwähnt, gehe aber gerne nochmals darauf ein. Da es wichtig ist das wir uns keine Abhängigkeiten in unser ViewModel holen, da wir dieses später ja evtl. testen möchten, ist es wichtig dass wir dies nicht irgendwann unabsichtlich tun. Ist unser Projekt einfach eine WPF Anwendung und wir erstellen lediglich einfach unterschiedliche Namespaces für die einzelnen „Layer" ist dies nicht gewährleistet.

In einer Projektmappe mit mehreren getrennten Projekten kann uns das nicht mehr so leicht passieren da wir hierfür explizit einen Verweis auf ein Projekt setzen müssten. Spätestens an diesem Punkt müsste uns unser Fehler dann auffallen.

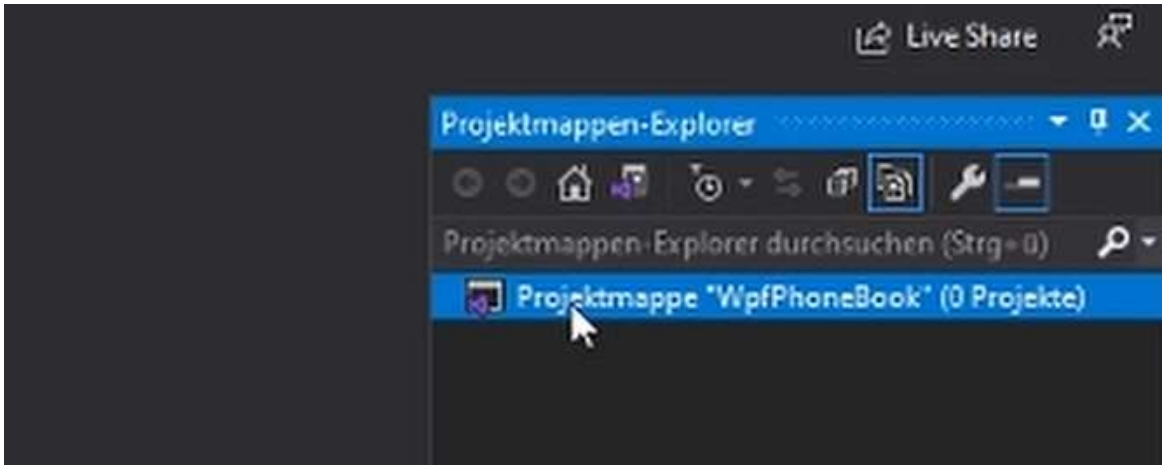
Aber auch für die widerverwendbarkeit, denn das Model oder gar das ViewModel können wir sicher in einer anderen Anwendung evtl. auch benötigen, insofern ist die Trennung auch für solch einen Fall ganz Praktisch.

Erstmal erstelle ich immer eine leere Projektmappe welcher ich den Namen des Projekts gebe, in diesem Fall ist dies „WpfPhoneBook“.



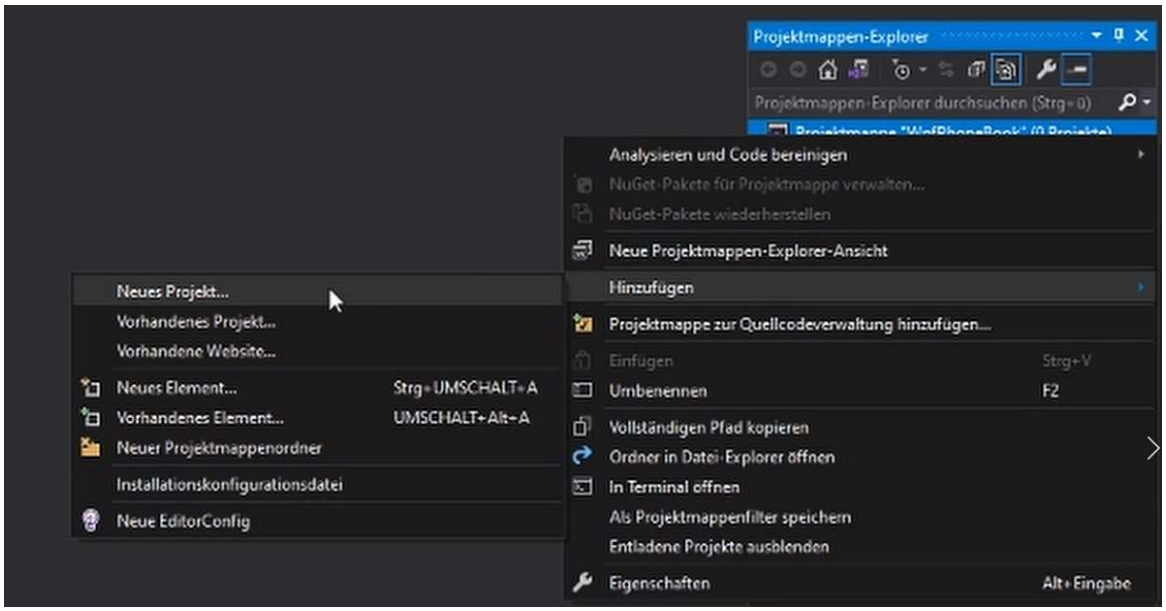

Wir suchen die Projektvorlage „Leere Projektmappe“ im Assistenten für neue Projekte.


Benennen diese Projektmappe nun „WpfPhoneBook“, wählen einen Speicherort und bestätigen die Angaben.



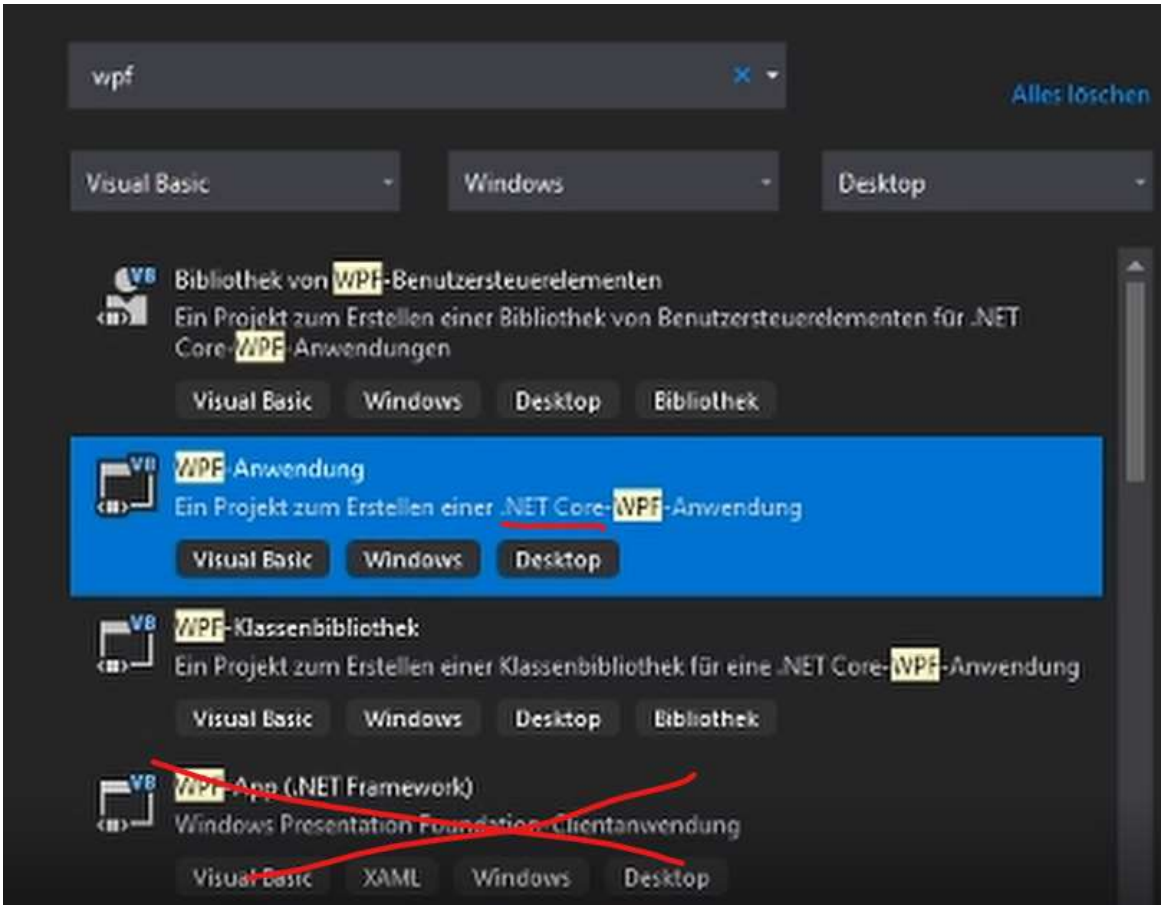

Daraufhin erhalten wie eine leere Projektmappe in welche sich keine Dateien befinden. Diese dient nur dafür verschiedene Typen von Projekten zu beherbergen.

Und genau das wollen wir ja. Wir erstellen also auch gleich das erste Projekt und fangen mit der App selbst an…




Dieses Projekt ist vom Typ **WPF .NET Core Anwendung**. Achtung, erstellt ihr die Anwendung unter .Net Core/.NET 5 müssen alle später folgenden Projekte unserer MVVM Anwendung auch .Net Core verwenden und nicht .Net Framework. Das ist wichtig, man „verklickt“ sich hier leicht!

Im folgenden Bild habe ich euch dies nochmals extra markiert…




Diese Anwendung nennen wir nun **WpfPhoneBook.App** und wählen im nächsten Schritt .Net 5 als Zielplattform aus.

Wurde uns dieses Projekt zur Projektmappe hinzugefügt können wir sogleich das nächste Projekt erstellen.  Die View.

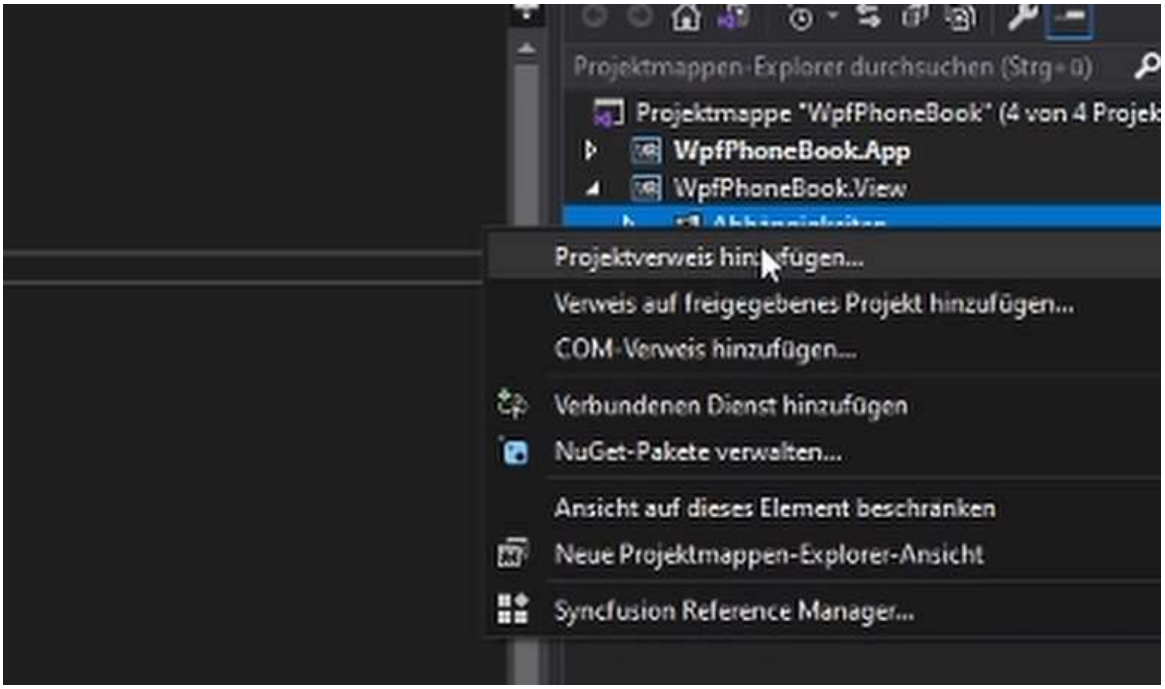
Diese ist von Typ „**Bibliothek für WPF-Benutzersteuerelementen**“. Auch wieder darauf zu achten das die Vorlage für .Net Core ist. Wir nennen diese „WPFPhoneBook.View“.

Als nächstes erstellen wir ein neues Projekt mit der Vorlage „**Klassenbibliothek**“. Dieses Projekt wird unser ViewModel. Wir nennen dieses also „WpfPhoneBook.ViewModel“.
**ACHTUNG**: Keine WPF Klassenbibliothek, eine normale Klassenbibliothek. Die WPF Klassenbibliothek bringt uns Abhängigkeiten mit die wir nicht haben möchten.

Zu guter Letzt benötigen wir noch ein Projekt für unser Model. Auch dies wird eine normale Klassenbibliothek für .Net 5. Ihr habt es erraten, wir nennen es „WpfPhoneBook.Model“.

In den Klassenbibliotheken können wir nun die erzeugten Class1.vb Dateien löschen, diese benötigen wir nicht. Im View-Projekt können wir das per Default erzeugte UserControl1.xaml löschen.

Nun müssen wir nur noch die Projektverweise unter den einzelnen Projekten herstellen…




Wie man Projektverweise herstellt solle man denke ich wissen wenn man sich an MVVM heranwagt, hier die Auflistung welches Projekt auf welches einen Verweis benötigt:

**Model** -> benötigt keinen Verweis auf ein Projekt
**ViewModel** -> Model
**View** -> ViewModel
**App** -> Model, ViewModel, View

Das war es dann schon mal fürs erste, im nächsten Video erstellen wir dann die ersten Infrastruktur-Files.

Im Video könnte ihr alles Schritt für Schritt sehen.

Hier geht's zum Video: https://youtu.be/rGPiDPEOp1E

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

## 6.1 Das Model erstellen

Erstellen wir nun unsere Models. Zuerst empfiehlt es sich eine Basisklasse für das Model zu erstellen wovon dann alle Models erben. Somit ist jede Property automatisch in jedem Model verfügbar. Dort können wir dann zum Beispiel die Property ID anlegen welche uns dann eine eindeutige ID des Datensatzes zurückgibt. Noch ein paar weitere Propertys und unsere Basisklasse sieht so aus.

```
1 Public MustInherit Class ModelBase
2
3     Public Property ID As Guid = Guid.NewGuid()
4     Public Property CreatedAt = DateTime.Now()
5
6     Public Property Deleted As Boolean
7     Public Property DeletedAt As DateTime?
8 End Class
```

Nun erstellen wir die Klasse „Contact“ und lassen diese von der eben erstellten Basisklasse erben. Fügen wir nun ein paar Propertys hinzu die ein „Contact“ haben soll.

```
1  Public Class Contact
2      Inherits ModelBase
3
4      Public Property Firstname As String
5      Public Property Lastname As String
6      Public Property ImagePath As String
7      Public Property Birthday As Date?
8      Public Property Note As String
9
10
11     Public Property ContactData As List(Of ContactData)
12 End Class
```

„ContactData“ beinhaltet eine Auflistung von „ContactData“. Somit kann ein „Contact“ mehrere Kontaktdaten (ContactData) haben. Also legen wir noch die Klasse „ContactData“ an.

```
1  Public Class ContactData
2      Inherits ModelBase
3
4      Public Property Data As String
5      Public Property ContactType As PersonContactType = PersonContactType.Phonenumber
6
7  End Class
8
9  Public Enum PersonContactType
10     Phonenumber = 0
11     Mail = 1
12     SocialMedia = 2
13     Fax = 3
14     Webvsite = 4
15 End Enum
```

„PersonContactType“ beschreibt den Typen von „ContactData“. Das kann eine Telefonnummer, eine Mailadresse oder was auch immer sein. In Data steht dann der entsprechende Inhalt, die Telefonnummer, Mail etc.

Somit wäre unser Datenmodel für unsere App fertig.

Im Video könnte ihr alles Schritt für Schritt sehen.

Hier geht’s zum Video: https://youtu.be/VqLLEAqmP78

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

## 6.2 Das Viewmodel - Der Core

Im letzten Kapitel haben wir unsere Models erstellt und damit quasi auch das Datenmodel unserer Anwendung.
Kommen wir nun zu den Viewmodels:
fangen wir wieder mit einer Basisklasse an, von welcher dann jedes Viewmodel erben sollte.
Als erstes benötigen wir INotifyPropertyChanged, welches wichtig für das Binding ist.
Es sorgt dafür, dass die View benachrichtigt wird, wenn sich im Viewmodel eine Eigenschaft geändert hat.
Hierfür implementieren wir INotifyPropertyChanged und setzen noch den benötigten Import für den Namespace.

Wenn wir nun die Schnittstelle implementieren, wird uns von VisualStudio ein PropertyChangedEventHandler erstellt.

```
Imports System.ComponentModel
Imports System.Runtime.CompilerServices

Public Class ViewModelBase
    Implements INotifyPropertyChanged

Public Event PropertyChanged As PropertyChangedEventHandler
Implements INotifyPropertyChanged.PropertyChanged

End Class
```

Da es nicht schön und viel Tipparbeit ist, das Event bei jeder Eigenschaft auszulösen, erstellen wir uns eine Methode, die uns das Ganze etwas vereinfacht.
OK, erstellen wir nun die Methode RaisePropertyChanged und übergeben den Propertynamen als String.
Innerhalb der Methode lösen wir nun das Event RaisePropertyChanged aus und übergeben als Sender uns selbst, als EventArg ein neues PropertyChangedEventArgs und übergeben diesem den Propertynamen.

```
Imports System.ComponentModel
Imports System.Runtime.CompilerServices

Public Class ViewModelBase
    Implements INotifyPropertyChanged

Public Event PropertyChanged As PropertyChangedEventHandler
Implements INotifyPropertyChanged.PropertyChanged

Public Overridable Sub RaisePropertyChanged(prop As String = "")
      RaiseEvent PropertyChanged(Me, New
PropertyChangedEventArgs(prop))
End Sub

End Class
```

Erstellen wir uns testweise ein Viewmodel und nennen es MainViewModel (werden wir später auch benötigen), lassen es von ViewModelBase erben und erstellen darin ein Property.
Name und Type sind erst mal nicht wichtig.
Nun können wir im Setter der Propertys RaisePropertyChanged() aufrufen und müssen den Propertynamen angeben.

Da es als String deklariert ist haben wir zwei Möglichkeiten: entweder, wir schreiben den Namen direkt als String „Test“ , oder wir nutzen den nameOf() Ausdruck, welcher uns den Namen einer Variable, eines Typs oder eines Members als Zeichenkette zurückgibt.

```
Private _test As String
Public Property Test() As String
      Get
            Return _test
      End Get
      Set(ByVal value As String)
            _test = value
            RaisePropertyChanged("Test")
            RaisePropertyChanged(NameOf(Test))
      End Set
End Property
```

Wir könnten das nun so lassen,aber wir wollen es etwas verbessern.
Als erstes wird unsere ViewModelBase Klasse als MustInherit deklariert.
Somit ist schon mal sichergestellt, dass diese Klasse als Basisklasse dient und wir keine Instanz davon erstellen können.

Als nächstes ändern wir den Übergabeparameter.
Seit .Net Framework 4.5 haben wir die Möglichkeit den <CallerMemberName> anzugeben.
<CallerMemberName> ermöglicht das Abrufen des Methoden oder Eigenschaftsnamens des Methodenaufrufers.
Somit können wir z.B. im Setter einer Property einfach RaisePropertyChanged() schreiben.
Der Compiler übergibt dann in unserem Fall den Eigenschaftsnamen des Aufrufers und das ist der Name unserer Property.
Nun müssen wir den vorherigen Übergabeparmeter als Optional angeben und ihm einen expliziten Standardwert zuweisen.
Den Standardwert setzen wir auf Nothing.

```
Imports System.ComponentModel
Imports System.Runtime.CompilerServices

Public MustInherit Class ViewModelBase
    Implements INotifyPropertyChanged

Public Event PropertyChanged As PropertyChangedEventHandler
Implements INotifyPropertyChanged.PropertyChanged

Public Overridable Sub RaisePropertyChanged(<CallerMemberName>
Optional prop As String = "")
RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(prop))
End Sub

End Class
```

Viele fragen jetzt noch ab, ob der übergebene Paramter auch wirklich was enthält und werfen dann das Event.
Das kann man sich sparen.
Wenn nichts (Nothing) oder String.Empty übergeben wird, werden alle Properties der Klasse aktualisiert.
Das ist manchmal ganz hilfreich, wenn man eine ganze View aktualiseren möchte.

So braucht man nicht extra eine Methode „RefreshAll“ schreiben, wo dann zehn oder mehr RaisePropertyChanged() mit dem jeweiligen Propertynamen stehen.
Einfach ein RaisePropertyChanged(nothing) oder RaisePropertyChanged(„“) aufrufen und alle Properties der Klasse werden aktualisiert.

Gut, das wäre bisher unsere Basisklasse.

Es sei nochmal gesagt das alles, was die Viewodels benötigen, hier rein gehört. Da liegt es an jedem selbst was er braucht und in die Basisklasse packt.

Nehmen wir an, wir laden Daten aus einer Datenbank, wo wir nicht sicher sind, wie lange der Vorgang dauert, weil es größere Mengen an Daten sind, die geladen werden sollen.
Man könnte da z.B. einen Waitingindicator einbauen, indem man in der Basisklasse eine Property IsBusy() implementiert.

```
Imports System.ComponentModel
Imports System.Runtime.CompilerServices

Public MustInherit Class ViewModelBase
    Implements INotifyPropertyChanged

    Public Event PropertyChanged As PropertyChangedEventHandler
Implements INotifyPropertyChanged.PropertyChanged

    Public Overridable Sub RaisePropertyChanged(<CallerMemberName>
Optional prop As String = "")
        If prop IsNot Nothing Then RaiseEvent PropertyChanged(Me, New
PropertyChangedEventArgs(prop))
    End Sub

    Private _isBusy As Boolean
    Public Property IsBusy() As Boolean
        Get
            Return _isBusy
        End Get
        Set(ByVal value As Boolean)
            _isBusy = value
            RaisePropertyChanged()
        End Set
    End Property


End Class
```

In jedem ViewModel, welches von der Basisklasse erbt, ist dies dann verfügbar und man kann z.b. folgendes schreiben.

```
Public Class MainViewModel
    Inherits ViewModelBase

    Private test As String
    Public Property Test() As String
        Get
            Return Test
        End Get
        Set(ByVal value As String)
            test = value
            RaisePropertyChanged("")
        End Set
    End Property
    Public Sub New()
        IsBusy = True
        'Do Something
        IsBusy = False

    End Sub
End Class
```

Somit wäre die Basisklasse erst mal fertig.

Nun stellen sich viele die Frage, was denn nun alles zum Kern gehört.
Im Endeffekt muss jeder selbst wissen, was er benötigt.

In unserem Beispiel nehmen wir mal einen ContactDataManager.
Wir müssen die Daten ja irgendwie verwalten. Da wir noch nichts von der Festplatte laden wollen, oder anderweitig Serilalisieren wollen, halten wir die Klasse ganz einfach.

Wir erstellen uns im ViewModelProjekt einen neuen Ordner und nennen diesen DataManager. Darin erstellen wir und eine Klasse und nennen diese ContactDataManager.

In unserem Fall reicht es uns wenn wir uns ein Property vom Typ List<T> erstellen und es AllContacts nennen.

Im Konstruktor initialisieren wir nun diese Liste und fügen ihr ein oder zwei Contacts hinzu.

```
Public Class ContactsDatamanager
    Public Sub New()
        AllContacts = New List(Of Model.Contact)
        AllContacts.Add(New Model.Contact() With {.Firstname =
"Sascha", .Lastname = "Patschka", .Birthday = New Date(1983, 9, 12)})
        AllContacts.Add(New Model.Contact() With {.Firstname =
"Susi", .Lastname = "Sorglos", .Birthday = New Date(1988, 5, 2)})
    End Sub

    Public Property AllContacts As List(Of Model.Contact)


End Class
```

Im Grunde ist das schon unsere ganze Infrastruktur die wir benötigen.

Beim nächsten mal reden wir dann darüber, wie man z.b Messageboxen oder Dialoge öffnet.

Alles was mit der View zu tun hat oder UI ist und im Viewmodel gesteuert werden muss, muss abstrahiert werden, denn das Viewmodel soll für sich alleine stehen und soll nichts von der View wissen.

Wenn wir beispielsweise versuchen im Konstruktor eine Messagebox wie gewohnt via Messagebox.Show() aufzurufen, bekommen wir von VisualStudio nichts angeboten, was in die Klammern soll.
Was uns VisualStudio allerdings anbietet ist ein Import.
Entweder auf PresentationFramework oder WinForms.

Letzteres fällt schon mal ganz raus. =)

Das PresentationFramework wäre eine Option, sie würde funktionieren und wäre nicht gänzlich falsch. Damit haben wir später aber dennoch ein Problem.

Das MVVM Pattern sieht vor das Layer strickt getrennt sind, was auch jetzt noch so wäre. Das Problem bekommen wir bei den Unittests, dazu kommt noch ein eigenes Kapitel. Durch die Messagebox wäre das Viewmodel nicht mehr testbar.

Unittests laufen voll automatisiert ab, probiert ob der Code läuft und ob am Ende das rauskommt was auch rauskommen soll. Das würde bedeuten, wir müssten jedes-mal die Messagebox anklicken und wüssten im Zweifel nicht, was wir anklicken sollen, je nach dem welcher Test gerade läuft. Aus diesem Grund müssen Dinge wie Messageboxen, Dialoge etc abstrahiert werden.

Dazu aber dann wirklich mehr im eigenen Kapitel.

Im Video könnte ihr alles Schritt für Schritt sehen.

Hier geht's zum Video: https://youtu.be/LJdFTCsxCE4

Hier zum Thread im VB-Paradise-Forum:
https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

## 6.3 Die RelayCommand Klasse

Bevor wir nun zu den Services kommen, womit wir dann letztendlich die Messageboxen, Dialoge etc steuern, möchte ich noch einmal auf die RelayCommand Klasse eingehen.

Dazu erstellen wir uns eine Klasse im Viewmodelprojekt, benennen diese RelayCommand und fügen folgenden Code ein.

```
Imports System.Windows.Input


''' <summary>
''' Diese Klasse Implementiert das ICommand Interface, so muss man nicht in jeder Klasse eines ViewModel alles selbst implementieren.
''' Einfach eine Command wie folgt Instanzieren:
''' MyCommand = New RelayCommand(AddressOf MyCommand_Execute, AddressOf MyCommand_CanExecute)
''' </summary>
Public Class RelayCommand : Implements ICommand



#Region " Fields                          "
    ReadOnly _execute As Action(Of Object)
```

```
    ReadOnly _canExecute As Predicate(Of Object)
#End Region


#Region " Constructors"
    ''' <summary>
    ''' Erstellt einen neuen Command welcher NUR Executed werden kann.
    ''' </summary>
    ''' <param name="execute">The execution logic.</param>
    ''' <remarks></remarks>
    Public Sub New(execute As Action(Of Object))
        Me.New(execute, Nothing)
    End Sub


    ''' <summary>
    ''' Erstellt einen neuen Command welcher sowohl die Execute als auch die
CanExecute Logik beinhaltet.
    ''' </summary>
    ''' <param name="execute">Die Logik für Execute.</param>
    ''' <param name="canExecute">Die Logik für CanExecute.</param>
    ''' <remarks></remarks>
    Public Sub New(execute As Action(Of Object), canExecute As Predicate(Of
Object))
        If execute Is Nothing Then
            Throw New ArgumentNullException("execute")
        End If

        _execute = execute
        _canExecute = canExecute
    End Sub
#End Region


#Region " ICommand Members              "
```

```
    ''' <summary>
    ''' Setzt die CanExecute-Methode des ICommand-Interfaces auf True oder
False
    ''' </summary>
    ''' <param name="parameter"></param>
    ''' <returns>Gibt zurück ob die Aktion ausgeführt werden kann oder
nicht</returns>
    ''' <remarks>
    ''' Benutzt DebuggerStepThrough from System.Diagnostics
    ''' Der Debugger überspringt diese Prozedur also, es sei den es wird
explizit ein Haltepunkt gesetzt.
    ''' </remarks>
    <DebuggerStepThrough>
    Public Function CanExecute(parameter As Object) As Boolean Implements
ICommand.CanExecute
        Return _canExecute Is Nothing OrElse _canExecute(parameter)
    End Function


    ''' <summary>
    ''' Event welches geworfen wird wenn die Propertie CanExecuteChanged sich
ändert.
    ''' </summary>
    ''' <remarks></remarks>
    Public Custom Event CanExecuteChanged As EventHandler Implements
ICommand.CanExecuteChanged
        AddHandler(value As EventHandler)
            If _canExecute IsNot Nothing Then
                AddHandler CommandManager.RequerySuggested, value
            End If
        End AddHandler
        RemoveHandler(value As EventHandler)
            If _canExecute IsNot Nothing Then
                RemoveHandler CommandManager.RequerySuggested, value
            End If
```

```
        End RemoveHandler
        RaiseEvent(sender As Object, e As EventArgs)
        End RaiseEvent
    End Event

    ''' <summary>
    ''' Führt die Prozedur Execute des ICommand.Execute aus
    ''' </summary>
    ''' <param name="parameter"></param>
    ''' <remarks></remarks>
    Public Sub Execute(parameter As Object) Implements ICommand.Execute
        _execute(parameter)
    End Sub
#End Region
End Class
```

Ich werde an dieser stelle nicht weiter auf den Code eingehen, was die Klasse macht und welche Aufgabe sie hat haben wir schon in einem vorherigen Kapitel besprochen.

Schaut dazu bitte im Kapitel 2.1..8.7 nach.

Nachdem wir den Code eingefügt haben wird uns der CommandManager rot unterstrichen.

VisualStudio schlägt uns den Import auf System.Windows.Input vor.

Hier muss man beachten dass Klassenbibliotheken die in .NET5 .NET6 oder .NET Core Projekten erstellt werden Cross-Platformfähig sind. Da System.Windos.Input aus PresentationCore nur unter Windows zur verfügung steht wäre die .dll nicht mehr Cross-

Platformfähig. Sollen die Viewmodels noch in anderen Anwendungen benutzt werden muss man sich um eine andere Implementierung kümmern.

Der Import wird nun aber noch nicht funktionieren.

Schauen wir uns dazu die Projektdatei an.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>

    <RootNamespace>WpfPhoneBook.ViewModel</RootNamespace>

    <TargetFramework>net5.0</TargetFramework>

  </PropertyGroup>

  <ItemGroup>

    <ProjectReference Include="..\WphPhoneBook.Model\
WpfPhoneBook.Model.vbproj" />

  </ItemGroup>

</Project>
```

Wie wir sehen ist dass TargetFramework .NET5.0

Bedeutet es hat nichts mit Windows zu tun und kennt daher auch die Windows spezifischen Namespaces nicht.

Hier müssen wir nun angeben das es sich um eine Windows App handelt, zusätzlich noch das wir WPF nutzen.

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>

    <RootNamespace>WpfPhoneBook.ViewModel</RootNamespace>

    <TargetFramework>net5.0-windows</TargetFramework>

    <UseWPF>true</UseWPF>

  </PropertyGroup>

  <ItemGroup>

    <ProjectReference Include="..\WphPhoneBook.Model\
WpfPhoneBook.Model.vbproj" />

  </ItemGroup>

</Project>
```

Gehen wir nun in die RelayCommandKlasse und speichern diese ab erkennt der Compiler das wir System.Windows.Input zur Verfügung haben, damit dann auch den CommandManager.

Hier geht's zum Video: https://youtu.be/HmiRei4khsg

Hier zum Thread im VB-Paradise-Forum:

https://www.vb-paradise.de/index.php/Thread/124617-Tutorialreihe-WPF-lernen

Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/