

TUTORIALREIHE WPF LERNEN

WPF verständlich und anhand von Praxisbeispielen erklärt

Sascha-Heinz Patschka

Inhaltsverzeichnis (inkrementell)

1. *Vorstellung*
 - 1.1. *Einleitung*
 - 1.2. *Aufbau dieser Tutorialreihe*
2. *Grundlagen der WPF*
 - 2.1. *Einführung in die WPF*
 - 2.1.1. *Die wichtigsten Controls und deren Verwendung*
 - 2.1.1.1. *Die wichtigsten Controls und ihr Verhalten*
 - 2.1.1.2. *Styles, Templates und Trigger*
 - 2.1.1.3. *Controls eine neue Optik verpassen*
 - 2.1.2. *XAML Namespaces*
 - 2.1.2.1. *Kurze Theorie*
 - 2.1.2.2. *Erstellung eigener Namespaces*
 - 2.1.3. *Resourcen*
 - 2.1.3.1. *Was sind Resourcen, was bringen sie mir*
 - 2.1.3.2. *Unterschied StaticResource und DynamicResource*
 - 2.1.4. *Binding und das Bindingsystem*
 - 2.1.4.1. *Was ist DataBinding? Das Konzept dahinter + Videocast*
 - 2.1.4.2. *Binding anhand einfacher Beispiele und Klassen*
 - 2.1.4.3. *DesignTime-Support für Binding*
 - 2.1.4.4. *Binding über Converter*
 - 2.1.4.5. *Binding über DataTemplates*
 - 2.1.4.6. *Binding an Collections. Warum ICollectionViewSource? Collections Filter, Sortieren, Gruppieren ohne viel Aufwand*
 - 2.1.4.7. *Validierung von Benutzereingaben*
 - 2.1.4.8. *Rücksicht nehmen auf die aktuelle Culture*
 - 2.1.5. *DependencyProperties*
 - 2.1.5.1. *Was sind Dependency Properties und wie unterscheiden sie sich von normalen Properties*
 - 2.1.5.2. *Eigene Dependency Properties implementieren*

- 2.1.6. *Markuperweiterungen*
 - 2.1.6.1. *Kurze Theorie*
 - 2.1.6.2. *Beispiele Anhand von Ressourcen und Styles*
- 2.1.7. *Attached Properties*
 - 2.1.7.1. *Kurze Theorie (wozu Attached Properties)*
 - 2.1.7.2. *Beispiele Anhand vom Grid und dem DockPanel*
 - 2.1.7.3. *Eigene Attached Properties erstellen ;-)*
- 2.1.8. *Attached Events*
 - 2.1.8.1. *Wir gehen gleich in die Praxis, gibt nicht viel zu sagen*
- 2.1.9. *Inputs und Command*
 - 2.1.9.1. *Die Input-API*
 - 2.1.9.2. *Tastatur und Mausklassen*
 - 2.1.9.3. *Eventrouting (Direct, Bubbling, Tunneling)*
 - 2.1.9.4. *Keyboard, Mouse und Textinput*
 - 2.1.9.5. *Touch und Multitouch (wird ja immer wichtiger)*
 - 2.1.9.6. *Focus (Der Unterschied zwischen Keyboardfocus und Logicalfocus)*
 - 2.1.9.7. *Commands (Integrierte und eigene)*
 - 2.1.9.8. *RelayCommand Klasse*
 - 2.1.9.9. *CommandBinding und Command-Parameter*
- ~~3. *Eine Telefonbuch Applikation unter WPF (ohne Binding)*~~
 - ~~3.1. *Hauptfenster erstellen und Funktionen festlegen*~~
 - ~~3.2. *Ein Primitives Telefonbuch rein mit CodeBehind ala WinForms*~~
 - ~~3.3. *Fazit*~~
- 4. *Eine Telefonbuch Applikation unter WPF (mit Binding der CodeBehind)*
 - 4.1. *Das Model erstellen / was soll die App können*
 - 4.2. *Die erste View erstellen und Binden*
 - 4.3. *Die erste View besser strukturieren*
 - 4.4. *Filtermöglichkeit einbinden (oder doch mehr?)*
 - 4.5. *Einträge bearbeiten und speichern*
- 5. *Das MVVM Pattern*
 - 5.1. *Was ist das MVVM Pattern?*
 - 5.2. *Wann MVVM und wann nicht?*
 - 5.3. *Welchen Mehrwert kann ich aus dem Pattern gewinnen?*
 - 5.4. *MVVM und CodeBehind – verboten?*
 - 5.5. *Model – View – ViewModel – Wars das?*
 - 5.6. *Erstellen einer korrekten MVVM Projektmappe in Visual Studio*
- 6. *Unser Telefonbuch in MVVM*
 - 6.1. *Projekt anlegen und Struktur besprechen*
 - 6.2. *Das Model erstellen*
 - 6.3. *ViewModel - Der Core – Was benötigen wir alles ehe wir anfangen mit unserem Programm*

- 6.4. *Wie MessageBox, Dialoge, Mousecursor oder TaskbarInfo steuern wenn ich die View nicht kenne*
- 6.5. *Jetzt anfangen? Ne? Warum?*
- 6.6. *Das MainViewModel erstellen*
- 6.7.
- 6.8. *Fazit zum MVVM Pattern*
- 7. *Lokalisierung und Globalisierung*
 - 7.1. *Lokalisierung nur mit Boardmitteln*
 - 7.2. *Lokalisierung mit schwung (unter Zuhilfenahme von zwei NuGet-Paketen)*
 - 7.3. *Globalisierung (Datum, Wahrung, usw.)*
 - 7.4. *Lokalisieren von Werten aus Fremssystemen (DB, XML usw.)*
 - 7.5. *Gute Hilfsprogramme und Helferlein*
- 8. *UnitTest und IntegrationTests*
 - 8.1. *Wozu UnitTests?*
 - 8.2. *Wie schreibe ich Tests (Grundlagen)*
 - 8.3. *Testen unseres ViewModels moglich?*
 - 8.4. ...
 - 8.5. ...
 - 8.6. *Fazit*
- 9. *Businesslogic*
 - 9.1. *Warum besser nicht im ViewModel*
 - 9.2. *Vor und Nachteile einer Businesslogik*
 - 9.3. *Neubau unseres Telefonbuchs mit eigener Businesslogik*
 - 9.4. *Fazit*
- 10. *Repository (DataAccessLayer)*
 - 10.1. *Noch einen Schritt weiter? Wozu?*
 - 10.2. *Besprechen und Aufsetzen eines Repository`s*
 - 10.3. *Wir bauen unser Telefonbuch abermals neu ;-(*
 - 10.4. *Fazit*

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.

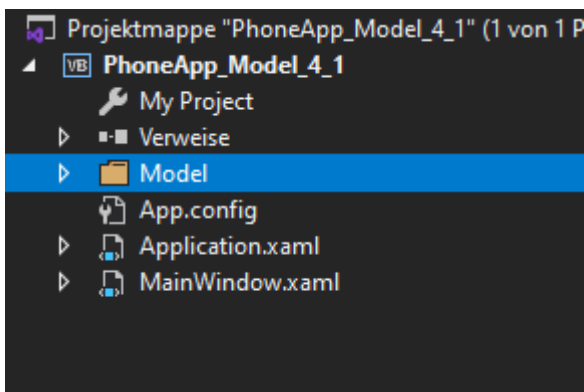
Es wird eine Telefonbuch App.

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 unterscheiden 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 Konstruktoren zu sehen, wird entweder eine neue Instanz eines `Contact` erstellt falls beim Instanzieren 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 meist 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 Leerzeichen 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 unterscheiden. 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.Ponenumber}}}))
            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(NumberOf(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.

```

<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>

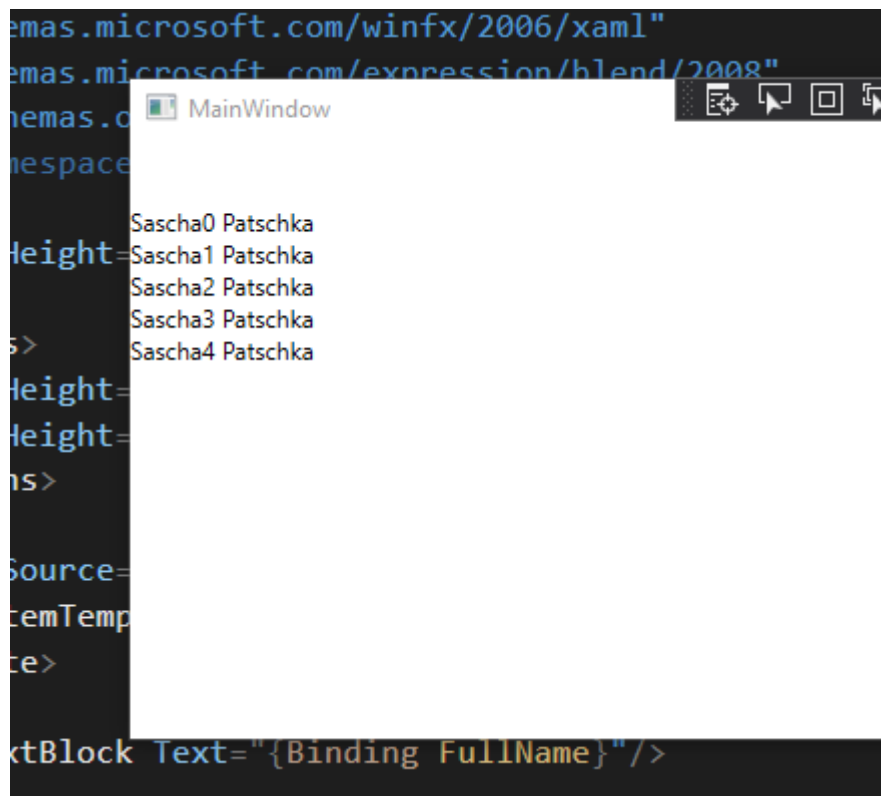
          <TextBlock Text="{Binding FullName}"/>

        </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 Ressourcen 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 Ordnung 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
PropertyChangeEventArgs(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.

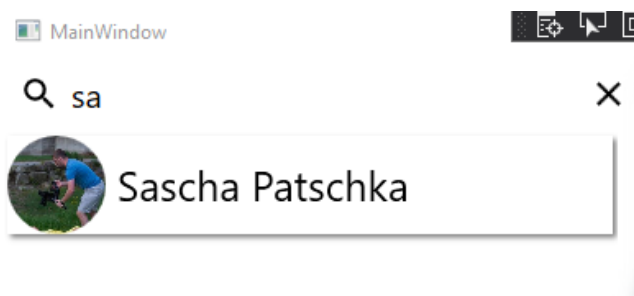
Nun fehlt uns nur noch die Methode für den eigentlichen Filter zu erstellen.

```
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 `ContentControl` 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 `MouseDown` 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 *GroupStyle* 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 Gruppieren.

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)

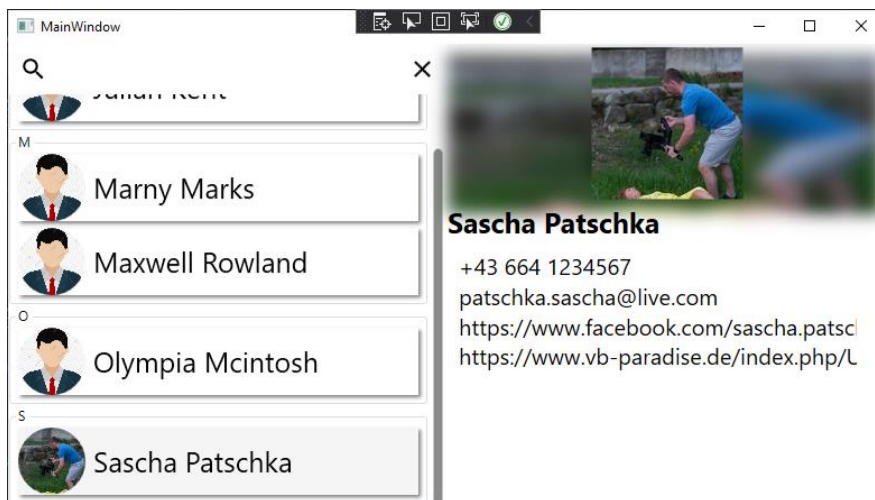
    Dim dp As New ContactsDataProvider()
    dp.GetAllContacts(True).ToList.ForEach((Sub(x) AllContacts.Add(New ContactViewModel(x))))

    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 Szenario 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 weiß die WPF dass 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 Instanzieren hier ein neues [Window](#) und setzen den [Content](#). In diesem Fall unser soeben erstelltes [UserControl](#). Den DataContext 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 Synchronisiert 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 allerdings 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 weiß 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

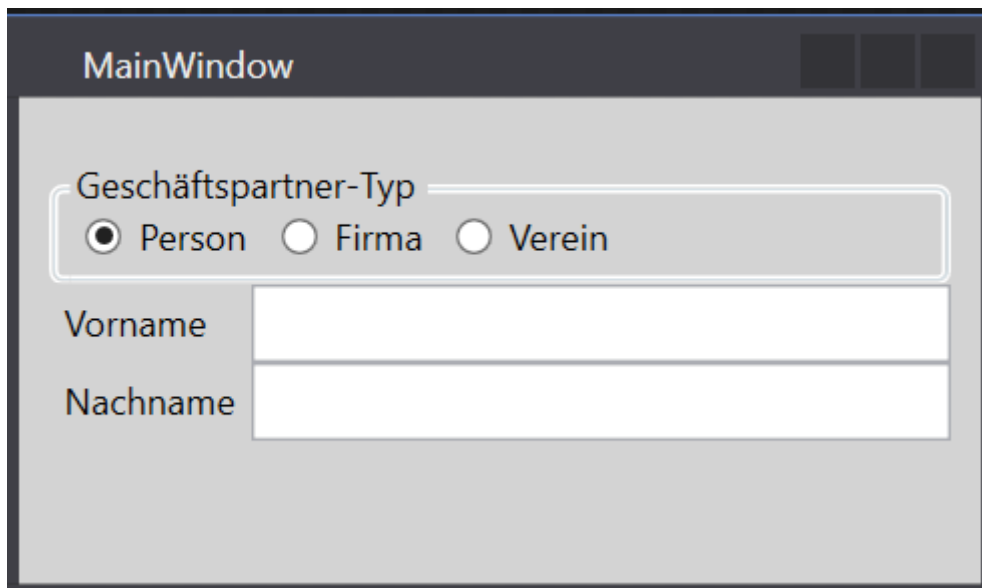
MVVM und CodeBehind – Verboten?

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:



The image shows a screenshot of a WPF window titled "MainWindow". The window has a dark title bar with standard Windows window controls (minimize, maximize, close) on the right. The main content area is light gray and contains a form. At the top of the form is a label "Geschäftspartner-Typ" followed by three radio buttons: "Person" (which is selected), "Firma", and "Verein". Below this are two text input fields, one labeled "Vorname" and one labeled "Nachname".

```

<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 Sonderzeichen 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 forbittenPasswords 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 forbittenPasswords.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 minimumPasswordLength As Integer =
CInt(settings.GetSettingValue("MinimumPasswordLength").Value)
    If newPassword.Length < minimumPasswordLength Then
        errorMessage = $"Das neue Passwort muss mindestens
{minimumPasswordLength} 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,
Nachname, Wörterbuchvergleich usw.)

    '....

    '....
    Return errorMessage
End Function

```

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](https://www.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

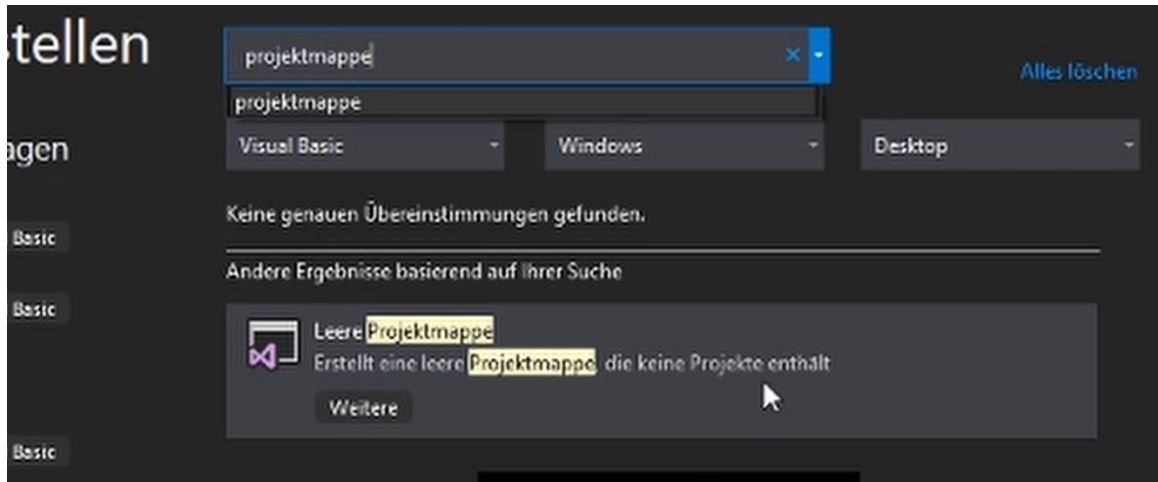
Wir erstellen nun eine Projektmappe unter Visual Studio welche „MVVM Gerech“ sein soll. Das bedeutet das alle Layer (Model – View – ViewModel) in getrennte Assemblys verteilt werden.

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 wiederverwendbarkeit, 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.

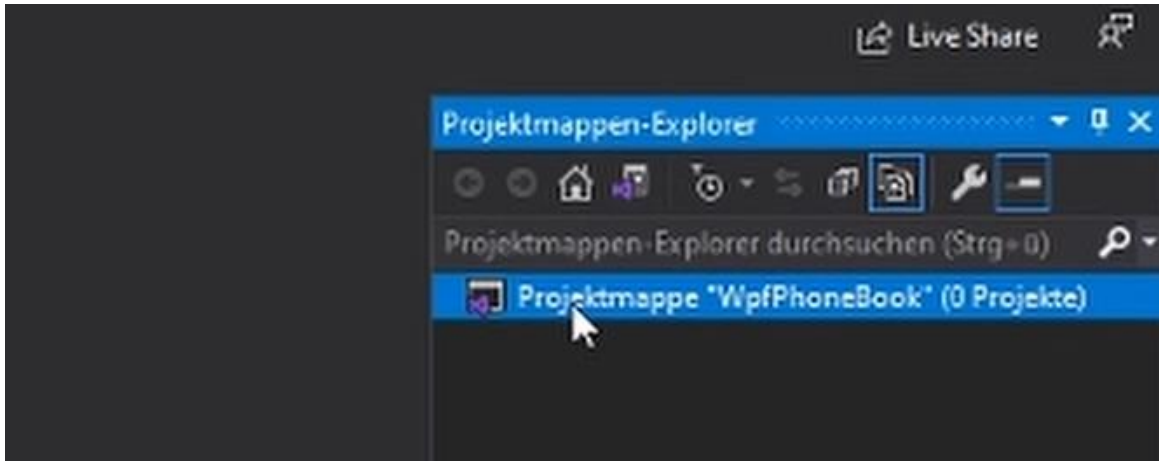
Erstmal erstelle ich immer eine leere Projektmappe welcher ich den Namen des Projekts gebe, in diesem Fall ist dies „WpfPhoneBook“.



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

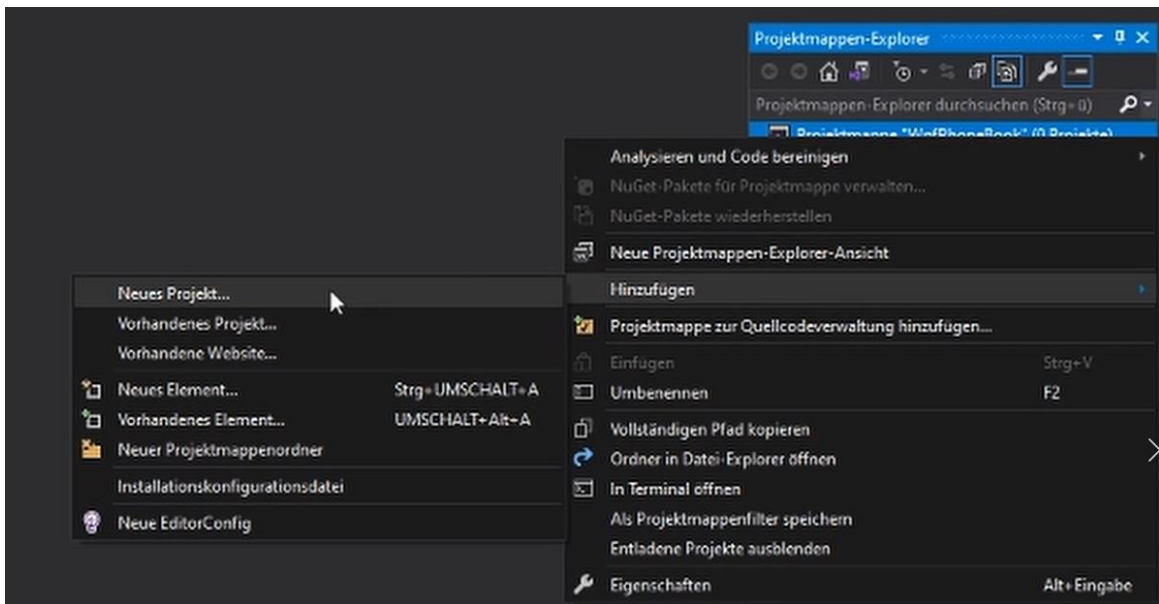


Benennen diese Projektmappe nun „WpfPhoneBook“, wählen einen Speicherort und bestätigen die Angaben.



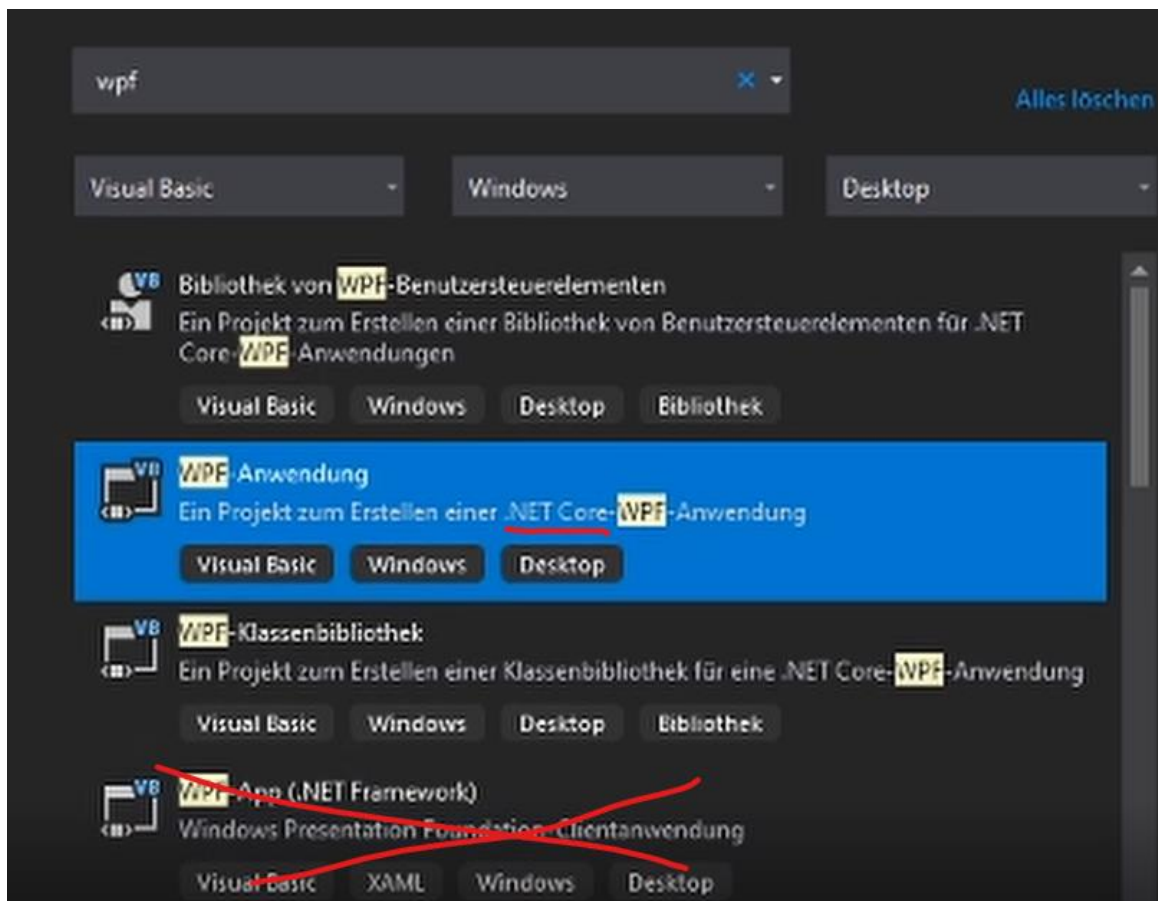
Daraufhin erhalten wir eine leere Projektmappe in welche sich keine Dateien befinden. Diese dient nur dafür verschiedene Typen von Projekten zu beherbergen.

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 „verclickt“ 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“.

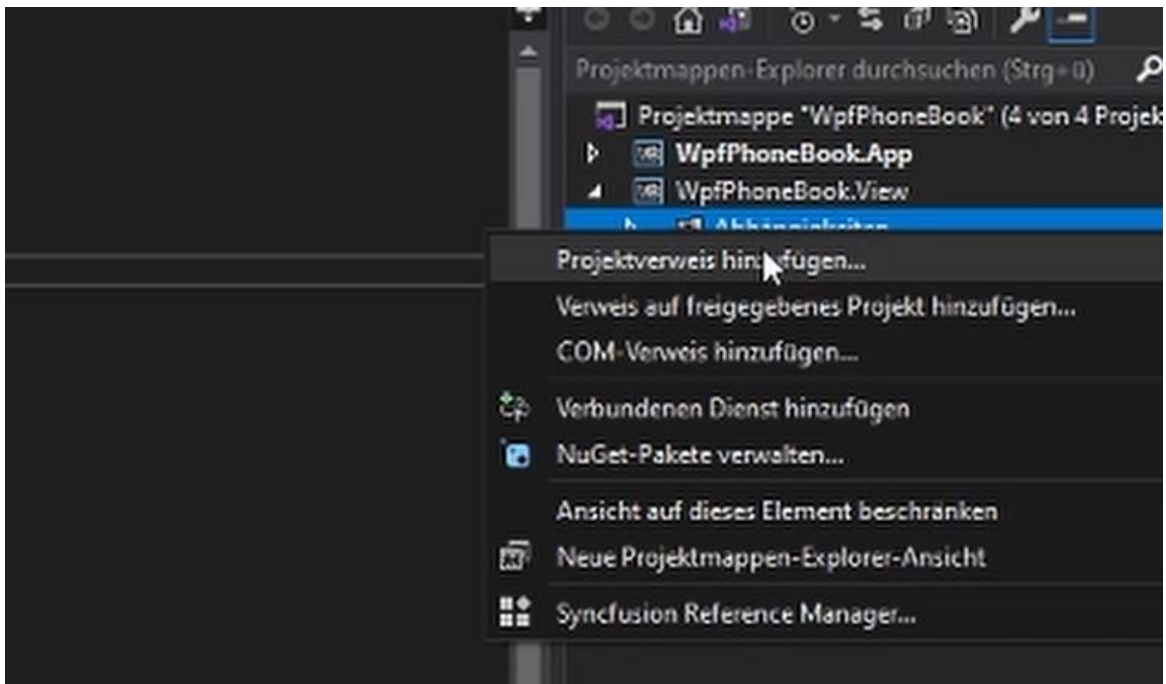
Als nächstes erstellen wir ein neues Projekt mit der Vorlage „**Klassenbibliothek**“. Dieses Projekt wird unser ViewModel. Wir nennen dieses also „WpfPhoneBook.ViewModel“.

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 sollte 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     Website = 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 Datenmodell 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 Datenmodell 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 EventArgs 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 Parameter 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 aktualisieren 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 Serialisieren wollen, halten wir die Klasse ganz einfach.

Wir erstellen uns im ViewModelProjekt einen neuen Ordner und nennen diesen DataManager. Darin erstellen wir 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 strikt 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-Plattformfähig sind. Da System.Windows.Input aus PresentationCore nur unter Windows zur Verfügung steht wäre die .dll nicht mehr Cross-Plattformfähig. 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/>

6.4 Services

Diesmal geht es darum wie wir [Messageboxen](#), [Dialoge](#) etc. aus den *Viewmodels* heraus steuern können. Also alles was die View betrifft, im MVVM Pattern kennen die *Viewmodels* die Views ja nicht.

Beispielsweise kann man in einen Viewmodel nicht einfach eine [MessageBox](#) aufrufen. Die Intellisense bietet es auch gar nicht an. Wenn wir es ausschreiben würden, wird uns ein *Import* von [System.Windows](#) angeboten. Damit würde es dann funktionieren aber wir wollen die Layer ja trennen, und die Testbarkeit wäre damit auch nicht mehr gegeben.

Wie gehen wir nun vor? Wenn wir eine [MessageBox](#) aufrufen wollen muss das von einem Ort aus passieren der die [MessageBox](#) kennt. Bedeutet wir müssen Abstrahieren. Abstrahierung funktioniert eigentlich immer mit Hilfe von Interfaces.

Ich nenne alles was ich in die Applikation hineinreiche gerne *Service*.

Fangen wir mit etwas einfachem an und nehmen zu Beginn eine Inputbox.

Erstellen wir uns im Viewmodel Projekt einen neuen Ordner und benennen diesen [Services](#).

Innerhalb dieses Ordners erstellen wir ein Interface und benennen dieses [IInputBoxService](#).

Hier empfiehlt es sich einen eigenen Namespace zu erstellen, so hat man alles was mit Services zu tun hat beisammen.

```
Namespace Services
Public Interface IInputBoxService
    End Interface
End Namespace
```

Um zu sehen, was eine Inputbox überhaupt kann schauen wir uns mal die [MSDN Seite](#) dazu an.

Hier sehen wir nun das die Inputbox einen `prompt` und einen `title` als Parameter hat und einen `string` zurück gibt. Optional kann man auch noch die `Position` angeben, einen vordefinierten Text anzeigen lassen etc. Dies benötigen wir erst mal nicht.

Wir wollen die Inputbox aufrufen, ihr 2 Parameter übergeben und sie soll einen `string` zurück geben.

```
Namespace Services
    Public Interface IInputBoxService
        Function Show(prompt As String, title As String) As String
    End Interface
End Namespace
```

Somit ist unser `Interface` bereits fertig. Mehr benötigen wir hier nicht.

Nun müssen wir das Interface auch mit Leben füllen. Dies passiert im *.App Projekt* denn dort haben wir vollen Zugriff auf `Messageboxen`, `Dialoge` etc.

Hierzu legen wir auch im *.App Projekt* einen neuen Ordner an und benennen diesen **Services**.

Darin erstellen wir eine Klasse, benennen diese `InputBoxService`, Implementieren das Interface und importieren den benötigten Namespace. Jetzt noch die Schnittstelle importieren und wir haben folgendes Gerüst.

```
Imports WpfPhoneBook.ViewModel.Services
Public Class InputBoxService
    Implements IInputBoxService
        Public Function Show(prompt As String, title As String) As String Implements IInputBoxService.Show
            Throw New NotImplementedException()
        End Function
End Class
```

Wir wollen aber den `string` der `InputBox` zurückgeben. Also returnen wir die `InputBox` und übergeben die beiden Parameter.

```
Imports WpfPhoneBook.ViewModel.Services
Public Class InputBoxService
    Implements IInputBoxService
    Public Function Show(prompt As String, title As String) As String Implements IInputBoxService.Show
        Return InputBox(prompt, Title:=title)
    End Function
End Class
```

Das wäre auch schon die ganze Implementierung. Mehr benötigen wir an dieser Stelle nicht.

Jetzt haben wir die Implementierung unseres Services, müssen diese aber noch „anmelden“.

Wir können die Services aus den Viewmodels heraus nicht nutzen denn das Viewmodel Projekt hat ja keinen Verweis auf das *.App Projekt*.

Wie lösen wir das nun?

Wir bauen uns einen Container. In diesen werden **Instanzen** von Services hineingereicht. Diese Instanzen können wir dann im Viewmodel aufrufen, ohne das wir etwas davon wissen müssen.

Wir kennen im Viewmodel ja nur das Interface.

Erstellen wir im ViewModel Projekt innerhalb des Ordners Services eine Klasse, benennen diese `ServiceContainer` und fügen folgenden Code ein.

Namespace Services

```
Public Class ServiceContainer
    Public Shared ReadOnly Instance As New ServiceContainer()

    Shared Sub New()
        ServiceMap = New Dictionary(Of Type, Object)()
        ServiceMapLock = New Object()
    End Sub

    Public Sub AddService(Of TServiceContract As Class)(implementation As TServiceContract)
        SyncLock ServiceMapLock
            ServiceMap(GetType(TServiceContract)) = implementation
        End SyncLock
    End Sub

    Public Shared Function GetService(Of TServiceContract As Class)() As TServiceContract
        Dim service As Object = Nothing
        SyncLock ServiceMapLock
            ServiceMap.TryGetValue(GetType(TServiceContract), service)
        End SyncLock
        Return TryCast(service, TServiceContract)
    End Function

    Shared ReadOnly ServiceMap As Dictionary(Of Type, Object)
    Shared ReadOnly ServiceMapLock As Object
End Class
End Namespace
```

Für den Code ist wichtig zu wissen das es `Dictionary(of Type, Object)()` gibt.

Man kann hier Objekte hinzufügen, gibt den Typ an und das Object ist quasi die Instanz davon, die man dem Container hinzufügen möchte.

Dafür gibt es die Methode `AddService()`, mit `GetService()` kann man einen gewissen `Type` abrufen.

Somit wäre auch dieser Teil fertig.

Gehen wir nun in das *.App Projekt*, öffnen die `Application.vb` und generieren uns mal das Startup Ereignis.

```
Class Application
```

```
    Private Sub Application_Startup(sender As Object, e As StartupEventArgs) Handles Me.Startup
```

```
    End Sub
```

```
    ' Application-level events, such as Startup, Exit, and DispatcherUnhandledException
```

```
    ' can be handled in this file.
```

```
End Class
```

Zum Start der App wollen wir unsere Services anmelden. Dazu benötigen wir den zuvor erstellten `ServiceContainer` und können dann folgendes schreiben.

```
Imports WpfPhoneBook.ViewModel
```

```
Imports WpfPhoneBook.ViewModel.Services
```

```
Class Application
```

```
    Private Sub Application_Startup(sender As Object, e As StartupEventArgs) Handles Me.Startup
```

```
        ServiceContainer.Instance.AddService(Of IInputBoxService)(New InputBoxService())
```

```
    End Sub
```

```
    ' Application-level events, such as Startup, Exit, and DispatcherUnhandledException
```

```
    ' can be handled in this file.
```

```
End Class
```

Wir haben den `ServiceContainer`, fügen dessen Instanz mit `AddService()` einen Service hinzu, geben den Type an, in unserem Fall `IInputBoxService`, und übergeben mit `New InputBoxService()` die Instanz.

Somit ist unser Service fertig und wir können ihn ausprobieren.

Dazu gehen wir in das `MainViewModel` und schreiben im Konstruktor folgenden Code:

```
Imports WpfPhoneBook.ViewModel.Services
```

```
Public Class MainViewModel
```

```
    Inherits ViewModelBase
```

```
    Public Sub New()
```

```
        Dim ib = ServiceContainer.GetService(Of IInputBoxService)
```

```
        Debug.WriteLine(ib.Show("Hallo Welt", "Test"))
```

```
    End Sub
```

```
End Class
```

Wir holen uns den [ServiceContainer](#), mit [GetService](#) den Service und geben an welchen Typ der Service haben soll. Wir rufen die Inputbox auf und lassen uns den zurückgegebenen [string](#) in der Ausgabe anzeigen.

Nun müssen wir nur noch das [MainViewModel](#) instanziiieren. Am einfachsten machen wir das im [Applikation_Startup](#) Ereignis.

```
Dim test As New MainViewModel
```

Damit ist alles fertig und mit [F5](#) können wir die Applikation starten.

Die Inputbox erscheint mit unserem Titel und Text. Schreiben wir etwas in die Eingabezeile und bestätigen mit Ok, erscheint der eingegebene Text in der Ausgabe.

Erstellen wir noch einen weiteren Service, einen den eigentlich fast jeder benötigt, die [MessageBox](#).

Wir erstellen also im Projekt *ViewModel* innerhalb des Ordners *Services* ein weiteres Interface und benennen dies [IMessageboxService](#).

Schauen wir wieder auf der MSDN Seite nach wie eine [MessageBox](#) aufgebaut ist.

Wir wissen ja was eine [MessageBox](#) an Parameter hat. [Title](#), [Message](#), [Icon](#) und [Buttons](#).

Aber wie sind die Buttons und Icons implementiert?

Suchen wir uns auf der MSDN Seite mal den Konstruktor raus der alle 4 Parameter hat und klicken dort auf „[MessageBoxButtons](#)“ um zur Implementierung zu gelangen. Jetzt müssen wir leider feststellen das die Buttons vom Type „[System.Windows.Forms.MessageBoxButtons](#)“ sind. Selbes beim Icon und [MessageBoxResult](#). Das würde bedeuten das wir uns die View ins Viewmodel holen. Aber genau das wollen wir nicht.

Wie umgehen wir das?

Wir müssen wieder Abstrahieren.

Am einfachsten geht das wenn wir uns Testweise in der Codebehind des [MainWindows](#) eine [MessageBox](#) erstellen.

```
Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles Me.Loaded
    MessageBox.Show("test", "Test", MessageBoxButton.OK,
    MessageBoxImage.Warning)
End Sub
```

Wenn wir jetzt den Cursor in „[MessageBoxButton](#)“ setzen und **F12** drücken bekommen wir die Definition angezeigt und können sehen wie es im .Net Framework implementiert ist.

```
Namespace System.Windows

    Public Enum MessageBoxButton
        OK = 0
        OKCancel = 1
        YesNoCancel = 3
        YesNo = 4
    End Enum
End Namespace
```

Wenn wir nun eine Kopie des Enumerators erstellen haben wir keinerlei Probleme bei der Übergabe.

Bedeutet wir kopieren den Enumerator und fügen ihn in unser Interface ein. Selbiges machen wir auch mit dem Icon und dem [MessageBoxResult](#). Definition ansehen und ins Interface kopieren.


```
Namespace Services
  Public Interface IMessageboxService
  End Interface
  Public Enum MessageBoxButton
    OK = 0
    OKCancel = 1
    YesNoCancel = 3
    YesNo = 4
  End Enum

  Public Enum MessageBoxImage
    None = 0
    [Error] = 16
    Question = 32
    Warning = 48
    Information = 64
  End Enum
  Public Enum MessageBoxResult
    None = 0
    OK = 1
    Cancel = 2
    Yes = 6
    No = 7
  End Enum
End Namespace
```

Nun können wir anfangen und den Konstruktor erstellen.

```

Namespace Services
    Public Interface IMessageboxService
        Function Show(title As String, message As String, button As
MessageBoxButton, icon As MessageBoxImage) As MessageBoxResult
    End Interface
    Public Enum MessageBoxButton
        OK = 0
        OKCancel = 1
        YesNoCancel = 3
        YesNo = 4
    End Enum
    Public Enum MessageBoxImage
        None = 0
        [Error] = 16
        Question = 32
        Warning = 48
        Information = 64
    End Enum
    Public Enum MessageBoxResult
        None = 0
        OK = 1
        Cancel = 2
        Yes = 6
        No = 7
    End Enum
End Namespace

```

Die `MessageBox` im CodeBehind kann nun gelöscht werden, diese benötigen wir nicht mehr.

Kommen wir jetzt zur Implementierung des Interfaces.

Im *.App Projekt* innerhalb des Ordners `Services` erstellen wir eine Klasse, benennen diese `MessageboxService` und implementieren das `IMessageboxService` Interface und lassen die Schnittstelle implementieren.

Nun haben wir nichts weiter zu tun als die `Messagebox` aufzurufen und die Parameter zu übergeben.

```
Imports WpfPhoneBook.ViewModel.Services
```

```
Public Class MessageBoxService  
    Implements IMessageboxService  
    Public Function Show(title As String, message As String, button As  
MessageBoxButton, icon As MessageBoxImage) As MessageBoxResult Implements  
IMessageboxService.Show  
        Return MessageBox.Show(message, title, button, icon)  
    End Function  
End Class
```

Damit wäre dann auch die Implementierung soweit fertig.

Der Service muss nun noch bei der App angemeldet werden.

Das machen wir wieder in der [Application.vb](#).

```
ServiceContainer.Instance.AddService(Of IMessageboxService)(New MessageBoxService())
```

Erstellen wir uns noch ein [UserControl](#) im *View Projekt* welches den Button beinhaltet.
Den [DataContext](#) des [UserControls](#) setzen wir auf das *MainViewModel* und Binden den
Button an ein [Command](#).

```

<UserControl x:Class="uc1TestMessagebox"
    xmlns=http://schemas.microsoft.com/winfx/2006/xaml/presentation
    xmlns:x=http://schemas.microsoft.com/winfx/2006/xaml
    xmlns:mc="http://schemas.openxmlformats.org/markup
compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:WpfPhoneBook.View" xmlns:viewmodel="clr
namespace:WpfPhoneBook.ViewModel;assembly=WpfPhoneBook.ViewModel"
    mc:Ignorable="d"
    d:DesignHeight="450" d:DesignWidth="800">
    <UserControl.DataContext>
        <viewmodel:MainViewModel/>
    </UserControl.DataContext>
    <StackPanel Margin="30">
        <Button Content="Show Messagebox over Service" Command="{Binding
ShowBoxCommand}"/>
    </StackPanel>
</UserControl>

```

Jetzt benötigen wir noch das Command im *MainViewModel*.

Den Aufruf der Inputbox im *MainViewModel* können wir auskommentieren.

```

Imports System.Windows.Input
Imports WpfPhoneBook.ViewModel.Services

Public Class MainViewModel
    Inherits ViewModelBase

    Public Sub New()
        _showBoxCommand = New RelayCommand(AddressOf ShowBoxCommand_Execute)
        'Dim ib = ServiceContainer.GetService(Of IInputBoxService)
        'Debug.WriteLine(ib.Show("Hallo Welt", "Test"))
    End Sub

    Private _showBoxCommand As ICommand
    Public ReadOnly Property ShowBoxCommand() As ICommand
        Get
            Return _showBoxCommand
        End Get
    End Property

    Private Sub ShowBoxCommand_Execute(obj As Object)
        ServiceContainer.GetService(Of IMessageboxService).Show("Test", "Das ist
ein Test", MessageBoxButton.YesNo, MessageBoxImage.Warning)
    End Sub
End Class

```

Wir holen uns wie im vorherigen Beispiel über den `ServiceContainer` den Service und geben den Typ an und können direkt danach `Show()` aufrufen. Dies ist jetzt die Variante in einer Zeile. Bei der Inputbox hatten wir es in 2 Zeilen. Jeder wie er es mag.

Da unser `UserControl` bisher noch nicht zugeordnet ist, setzen wir das nun in unser `MainWindow`.

```

<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:view="clr-namespace:WpfPhoneBook.View;assembly=WpfPhoneBook.View"
    xmlns:local="clr-namespace:WpfPhoneBook.App"
    mc:Ignorable="d"
    Title="MainWindow" Height="450" Width="800">
    <Grid>
        <view:uclTestMessagebox/>
    </Grid>
</Window>

```

Testen wir nun die App mit Klick auf `F5` und betätigen den Button erscheint unsere `Messagebox`.

Alles so wie wir es möchten.

Allerdings gibt es noch einen kleinen Haken. Wir haben in den Projekten **OptionStrict** noch nicht auf **ON** gestellt. :(

Das holen wir jetzt nach und schauen mal was passiert.

Also, alle Projekte auf **OptionStrict On** stellen.

Hier bekommen wir jetzt einen Fehler angezeigt der sagt das die Typen der **MessageBoxButtons**, **MessageBoxIcons** und des **MessageBoxResults** nicht übereinstimmen.

Das ist auch richtig, wir haben uns ja eine Kopie der Enumerator erstellt. Diese sind vom Typ **ViewModel.Services.MessageBoxButton** und nicht vom Typ **Sytem.Windows.MessageBoxButton**.

Da wir eine Kopie der Enumerator erstellt haben sind die Werte identisch und wir haben es einfach zu konvertieren.

```
Imports WpfPhoneBook.ViewModel.Services
```

```
Public Class MessageBoxService
```

```
    Implements IMessageboxService
```

```
    Public Function Show(title As String, message As String, button As  
MessageBoxButton, icon As MessageBoxImage) As MessageBoxResult Implements  
IMessageboxService.Show
```

```
        Return DirectCast(MessageBox.Show(message, title, DirectCast(button,  
Windows.MessageBoxButton), DirectCast(icon, Windows.MessageBoxImage)),  
MessageBoxResult)
```

```
    End Function
```

```
End Class
```

Nochmal mit **F5** starten und schauen ob alles funktioniert.

Damit ist das Service dann auch erfolgreich implementiert. :)

Gehen wir noch das Handling der Rückgabewertes durch, wenn wir **Ja** oder **Nein** abfragen möchten.

Dazu gehen wir in die *Execute* Methode des **Commands**.

```
Private Sub ShowBoxCommand_Execute(obj As Object)
    If ServiceContainer.GetService(Of IMessageboxService).Show("Test", "Das
ist ein Test", MessageBoxButton.YesNo, MessageBoxImage.Warning) =
MessageBoxResult.Yes Then
        Debug.WriteLine("ER HAT JA GEKLICKT!!!!")
    End If
End Sub
```

Das wars schon. Wir können die `MessageBox` nun wie gewohnt nutzen, müssen es nur vorher aus dem Service herausholen.

Hier geht's zum Video: <https://youtu.be/4G-cTdUahFM>

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/>

