TUTORIALREIHE WPF LERNEN

WPF verständlich und anhand von Praxisbeispielen erklärt

Sascha-Heinz Patschka

Inhaltsverzeichnis (inkrementell)

1.0

Vorstellung

Mein Name ist Patschka Sascha-Heinz, ich bin 1983 geboren und arbeite als EDV Techniker. Beruflich habe ich fast nichts mit der Programmierung zu tun und komme sohin nur privat dazu mich sowohl weiterzubilden als auch mehr Übung zu bekommen.
Da ich fast von Anfang an unter WPF programmiere und unter WinForms wirklich nur ca. 2-3 Monate gearbeitet habe gab es für mich von Anfang an nur die Richtung zur WPF. Die WPF ist ein sehr leistungsstarkes Framework welches einem nicht nur in Punkto Optik neue Möglichkeiten eröffnet.
Anfangs hatte ich keine Ahnung von Pattern wie dem MVVM oder anderen, da ich das meiste einfach per "lerning by doing" gelernt habe. Erst nach einigen Jahren aktiver Programmierung unter WPF kam ich zu dem Pattern MVVM. Erstmals totales Neuland mit vielen verschiedenen Ansätzen und Anfangs schwer zu durchschauen, dachte ich mir nicht das dieses Pattern mich irgendwann dazu bringen könnte eine Aussage wie "wenn möglich verwende ich nur noch MVVM in der WPF" zu tätigen, doch seit einiger Zeit ist dies immer öfter der Fall. Ich kann sehr gut nachvollziehen wie frustrierend es sein kann mit der WPF zu arbeiten wenn man bereits längere Zeit mit z.B. WinForms gearbeitet hat. Es kann(!) sehr frustrierend sein wenn man nicht auf Anhieb weiterkommt und im Netz finden sich sowohl was die WPF Ansicht und deren Verwendung angeht viele verschiedene Ansätze also auch was das MVVM angeht. Das kann sehr frustrierend sein. Einige davon mehr oder weniger gut und manche leider auch sehr schlecht und gar nicht skalierbar. Ich selbst habe bereits sicher 20 verschiedene Ansätze der Umsetzung eines mehr oder weniger korrekten MVVM Patterns gesehen. Weiteres über MVVM in einem späteren Kapitel.

1.1

Einleitung

Ich werde absichtlich so wenig wie nur möglich mit Fremdwörtern oder kompliziertem Code um mich werfen.
Es soll in dieser Tutorialreihe darum gehen den Code zu verstehen. Auch Anfänger sollten den Code lesen und nachbauen können. Evtl. wird auch Code auskommentiert werden und darunter eine andere Möglichkeit geboten wie z.B. eine Schleife gegen eine Lambda Expression vereinfacht werden kann, einfach damit auch Personen welche noch nicht mit Lambda gearbeitet haben verstehen was hier passiert.
Einige Dinge werden in den Folgekapiteln sicher einfacher gehen oder besser gelöst werden können, hierfür wird dann ein Diskusionsthread zur Verfügung stehen. Auch werde ich nur die wichtigsten Zeilen kommentieren damit bei Anfängern der Lerneffekt nicht ausbleibt.

Ich werde in VisualStudio 2017 Update 3 schreiben und die .Net Sprache VB.NET verwenden.
Falls Ihr Fragen zu diesem Tutorial, den Code oder über mich habt freue ich mich über ein Mail von euch. Auch für Kritik bin ich natürlich immer offen. Mails bitte an patschka.sascha@live.com oder eine PM im Forum www.vb-paradise.de an „NoFear23m“.

1

Ich setze in dieser Tutorialreihe Kenntnisse in der objektorientierten Programmierung voraus und gehe davon aus das die Grundkenntnisse und Syntax von VB.Net soweit bekannt sind.

Und nun viel Spaß mit meinen Tutorials.

1.2

Aufbau dieser Tutorialreihe

Es wird ca. 1-mal pro Woche ein Beitrag mit mindestens einem Kapitel auf www.vb-paradise.de online gestellt. Es kann vorkommen das auch mal 2 oder mehr Kapitel behandelt werden. Je nachdem wie ich dazu komme und Zeit habe. Falls es vorkommen sollte dass ich mal eine Woche auslasse entschuldige ich mich bereits im Voraus dafür, bitte habt Verständnis das ich mal in Urlaub fahre oder beruflich etwas mehr Stress habe.
Diese Tutorialreihe wird als "Hybrid" aufgebaut. Teile werden als normale Beiträge in reinem Text bzw. mit Bildern erstellt, andere Teile aber auch als Videocast.
Es wird außerdem für jedes Kapitel ein ZIP File online gestellt welches dieses Inhaltsverzeichnis und die Kapitel bis zum aktuellen Zeitpunkt enthält. Außerdem mit in dem ZIP File wenn vorhanden die VisualStudio Solution abwärtskompatibel bis Visual Studio 2015, sowie Links zu den Videos sofern vorhanden.
Sollte ein Beitrag rein als Text ohne Video erstellt worden sein wird ein PDF mit in der ZIP enthalten sein damit jeder auch offline in Ruhe alles lesen kann.
Sämtliche Verweise in den Solutions werden nur als NuGet Verweise in das jeweilige Projekt eingebunden um sicherzustellen das die Solution nach dem Download auch bei jedem läuft da NuGet automatisch nachgeladen wird wenn nicht vorhanden. Weitere Infos könnt Ihr hier nachlesen.

Warum mit Videos? Man könnte jetzt sagen das ist totaler Quatsch und aus einem Video kann ich nichts rauskopieren und ich kann schwer später gezielt zu einem bestimmten Code springen um mir diesen nochmals anzusehen. Aber genau das sollte auch vermieden werden. Ich bin kein Freund von Copy&Paste. Nur wenn ich den Code tippe kann ich versuchen ihn zu lernen und zu verstehen. Auch bin ich der Meinung dass über ein Video viel mehr über die Funktionalität von der IntelliSense vermittelt werden kann. Außerdem kann ich in einem Video schöner gewisse Tastenkombinationen und Tricks vermitteln welche einem das tägliche Leben leichter machen oder einem viel Tipparbeit ersparen wie z.B. bei CodeSnippets. Das sind alles Gründe warum ich persönlich ein Video einem normalen Text/Bild Beitrag vorziehe.

Bitte entschuldigt wenn ich in meinem Video evtl. mal in meinen österreichischen Dialekt falle. Ich werde mich bemühen so gut wie möglich in einem verständlichen Hochdeutsch zu sprechen.

2.0

GRUNDLAGEN DER WPF

Wir kommen zu den Grundlagen. Die WPF bietet eine eigene "Designersprache". XAML (Extensible Application Markup Language); ist eine Markupsprache und ist ähnlich im Aufbau wie XML. Der Großteil der Benutzeroberfläche wird in der WPF mit XAML erstellt.
Was ist XAML? Im Grunde vereinfacht XAML das Erstellen einer UI einer .NET Anwendung.
Es können sichtbare UI-Elemente im deklarativen XAML-Markup erstellt und anschließend die UI-definition mithilfe von Code-Behind, die über partielle Klassendefinitionen an das Markup geknüpft sind, von der Laufzeitlogik trennen.

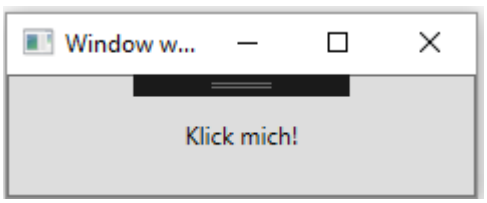
Die Darstellung passiert als Text über XML Dateien welche die Erweiterung .xaml aufweisen. Es kann mit jeder Codierung gearbeitet werden, typisch ist jedoch die Codierung als UTF-8.

Ich drifte aber zu weit ab, hier ein Beispiel für einen XAML - Code:

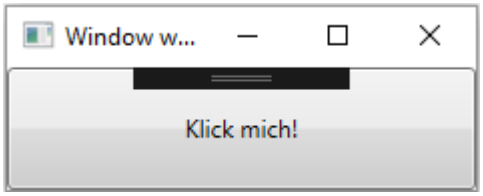
```
<Window xml:Class="MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window with Button"
   Width="250" Height="100">
<!-- Button hinzufügen -->
  <Button Name="button">Klick mich!</Button>
</Window>
```

Hier werden ein Fenster und eine Schaltfläche mithilfe der Elemente Window und Button definiert. Jedes Element wird mit Attributen konfiguriert und korrespondiert mit den jeweiligen Properties des Elements. Zeile zwei und drei beinhalten die per Default eingetragenen XAML Namespaces, dazu kommen wir auch später mal, da möchte ich im Moment nicht so weit vorgreifen. Ein Window hat ein Property Title und kann hier mit dem Attribut Title geändert werden. Geben wir also für den Button beim Attribut 'Name' den Wert 'button' an wird dieser Wert in das Property 'Name' geschrieben. In diesem Fall handelt es sich um ein Dependency Property aber dazu kommen wir mal in einem der Videos zu sprechen.

So würde dieser Code im Programm aussehen:



Unter Windows 7 so:



Im Hintergrund passiert nichts anderes als das die WPF die Objekte anhand Ihrer Attribute und deren Werten erstellt. Erstellen wir dieses Fenster mal im Code.

```
Dim MainWindow As New Window
MainWindow.Title = "Window with Button"
MainWindow.Width = 250
MainWindow.Height = 100
Dim myButton As New Button
myButton.Name = "button"
myButton.Content = "Klick mich!"
AddHandler myButton.Click, AddressOf Button_Click
MainWindow.Content = myButton
```

Wenn wir nun den XAML mit dem Code vergleichen fällt uns ziemlich schnell auf wie die WPF das macht.
In einem <Button/> wird z.B. `Dim myButton As New Button` erstellt. Jedes Attribut steht für ein Property der jeweiligen Klasse.
z.B. wird `myButton.Name = "button"` in XAML zu `Name="button"`

Wenn man diese Info hat wird man die XAML Syntax gleich viel schneller verstehen.

CODE BEHIND

Ich lese immer wieder in Foren dass kein Code Behind verwendet werden soll und dass man unter WPF nur MVVM verwenden soll, alles andere wäre Quatsch. Doch das ist nicht korrekt, auch in Code von Microsoft und diversen großen Herstellern wie DevExpress, welcher einer der größten Komponentenhersteller im .Net Bereich ist, wird immer wieder über Code Behind gearbeitet. Zweifels ohne ist die WPF auf Binding und gewisse Pattern ausgerichtet wodurch man gewisse Vorteile erlangt wenn man diese verwendet. Dennoch ist es so dass ich für eine kleinere Anwendung kein MVVM empfehlen würde. Näheres in einem späteren Kapitel.

Ähnlich wie bei WinForms kann ich nun einen Handler für den Button_Click erzeugen und in diesem meinen Code schreiben.

Wir schreiben das Attribut 'Click' in den Button XAML Code und drücken zweimal Tab.
Nun wird folgende Codezeile als Button vorhanden sein:

```
<Button Name="button" Click="button_Click_1">Klick mich!</Button>
```

Visual Studio hat uns nun den Code in der Code Behind des Windows erstellt.
Mit einem Rechtsklick auf den Code des Click Attributes können wir nun mit "Gehe zu Definition" direkt zu diesem Code springen.

Hier steht nun folgendes:

```
Private Sub button_Click_1(sender As Object, e As RoutedEventArgs)

End Sub
```

Und dies kennen wir nun ja wieder aus WinForms.
Was anders ist, ist der RoutedEventArg, dies erkläre ich allerdings im Kapitel RoutedEvents, da möchte ich jetzt noch nicht vorgreifen.

Ein weiteres Beispiel mit Attributen:

```
<Button Background="Blue" Foreground="Red" Content="This is a button"/>
```

Es wird die Schriftfarbe des Controls auf die Farbe "Red" gesetzt und der Hintergrund des Controls auf "Blue".

Allerdings gibt es auch Eigenschaften eines Objekts welche nicht über die Attributsyntax gesetzt werden da diese z.B. zu komplex sind. Auf diese kann dann über die Eigenschaftenelementsyntax zugegriffen werden.
Oft ist es aber auch Geschmacksache oder Übersichtlichkeit für welche Art man sich entscheidet.
Hier ein Beispiel für das umgestaltet des oben stehenden Codes unter Verwendung der Eigenschaftenelementsyntax:

```
<Button>
     <Button.Background>
          <SolidColorBrush Color="Blue"/>
     </Button.Background>
     <Button.Foreground>
          <SolidColorBrush Color="Red"/>
     </Button.Foreground>
     <Button.Content>
          This is a button
     </Button.Content>
</Button>
```

Dies wäre nun derselbe Button nur das die Properties über die Eigenschaftenelementsyntax gesetzt wurden.
Jetzt könnte man sich denken: "Wozu gibt es denn die Eigenschaftenelementsyntax wenn ich das mit der Attributsyntax ja auch machen kann?"
Hierfür gibt es mehrere Gründe. Einer wäre z.B. dass die Background Eigenschaft eines Buttons ja eigentlich einen Brush erwartet. Warum kann man dann `Background="Blue"` anwenden?
Die WPF stellt intern diverse TypeConverter bereit, so kann sie den String "Red" in einen SolidColorBrush mit der Color "Red" umwandeln. Das gleiche gilt für die Eigenschaft Content des Buttons welche von Typ „Object“ ist. Hier macht die WPF automatisch einen String daraus.

Aber warum soll ich die lange Variante schreiben wenn ich die kurze doch auch schreiben könnte?
In diesem Fall (SolidColorBrush) geht das noch alles über die Attributsyntax da die WPF den String ja in einen SolidColorBrush wandelt. Wenn man nun einen Farbverlauf als Hintergrund

verwenden möchte muss man aber schon auf die Eigenschaftenelementsyntax zurückgreifen.
z.B.:

```
<Button>
     <Button.Background>
          <LinearGradientBrush>
               <LinearGradientBrush.GradientStops>
                    <GradientStop Offset="0.0" Color="Red" />
                    <GradientStop Offset="1.0" Color="Blue" />
               </LinearGradientBrush.GradientStops>
          </LinearGradientBrush>
     </Button.Background>
     <Button.Foreground>
          <SolidColorBrush Color="Red"/>
     </Button.Foreground>
     <Button.Content>
          This is a button
     </Button.Content>
</Button>
```

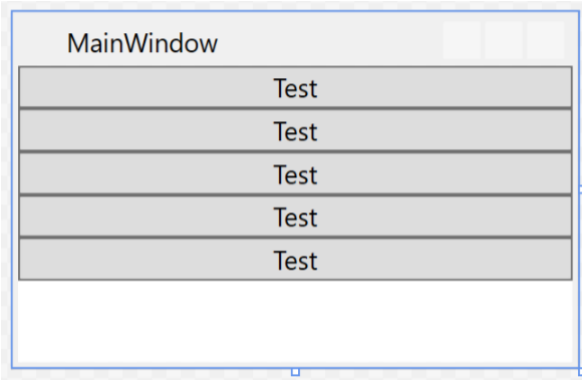
Hier wird beim ersten Button für die Hintergrundfarbe ein Farbverlauf gewählt.
Dieser Farbverlauf geht von oben nach unten von Rot nach Blau. Und zwar gleichmäßig.
Über das Offset könnte man hier Einfluss auf die "Geschwindigkeit" des Verlaufs nehmen.

Gehen wir gleich zu den XAML Inhaltseigenschaften.
Diese sind auch ganz interessant. In der WPF gibt es viele Controls welche ein Property Content oder Child enthalten.
Oft sind diese Properties vom Typ Object. Da ein Object übergeben werden kann, kann dies im Grunde alles Mögliche sein. Nehmen wir wieder als Beispiel den guten alten Button.
Die folgenden Buttons sehen alle völlig gleich aus obwohl man dies auf den ersten Blick vielleicht nicht vermuten mag.

```
        <StackPanel>
            <Button>Test</Button>
            <Button>
                <Button.Content>
                    Test
                </Button.Content>
            </Button>
            <Button Content="Test"/>
            <Button>
                <TextBlock Text="Test"/>
            </Button>
            <Button>
```

```
            <Button.Content>
                <TextBlock>
                    <TextBlock.Text>
                        Test
                    </TextBlock.Text>
                </TextBlock>
            </Button.Content>
        </Button>
    </StackPanel>
```

Hier ein Screenshot:



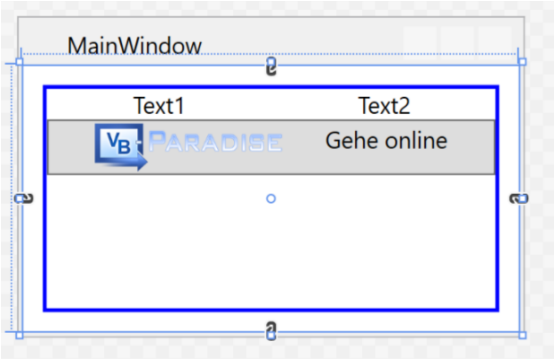
Und da kommen wir gleich zum nächsten, der Auflistungssyntax:
Es gibt Container welche mehr als ein Control als Inhalt bekommen können. z.B., Grid, UniformGrid, alle Arten von Panels usw.

Folgendes Beispiel fügt einem Border ein StackPanel hinzu welches ein UniformGrid und einen Button enthält.
Das UniformGrid bekommt zwei Spalten welchen jeweils ein TexBlock hinzugefügt wird.
Dem Button wird wieder ein StackPanel zugewiesen worin sich ein Image und ein Textblock horizontal aufgelistet befinden.

```
<Border BorderThickness="2" BorderBrush="Blue" Margin="10">
    <StackPanel>
        <UniformGrid Columns="2">
            <TextBlock HorizontalAlignment="Center">Text1</TextBlock>
            <TextBlock HorizontalAlignment="Center"
                       Grid.Column="1">Text2</TextBlock>
        </UniformGrid>
        <Button>
            <StackPanel Orientation="Horizontal">
                <Image Width="100"
  Source="http://www.vb-paradise.de/wcf/images/wbbLogo_vbp.png"/>
                <TextBlock Text="Gehe online" Margin="20,0,0,0"/>
            </StackPanel>
        </Button>
    </StackPanel>
```

```
        </Border>
```

Hier wieder ein Screenshot:



Ich denke das war jetzt erstmal genug Theorie, ich finde das man mit *"lerning by doing"* einfacher das gelernte behält. Sicher ist es gut wenn man weiß wie die Syntax aufgebaut ist, man muss aber auch damit umgehen können. Die wichtigsten Grundlagen der Syntax habe ich ja aufgezeigt, ich würde sagen wir legen einfach mal los. Beim "basteln" der ersten Anwendung werden sicher viele Fragen bereits beantwortet. Ich werde auch versuchen meine Schritte immer zu kommentieren und euch hin und wieder verschiedene Wege zu zeigen um ans Ziel zu kommen.

Weitere Grundlagen und Infos findet Ihr hier: https://msdn.microsoft.com/de-de/library/ms746927(v=vs.100).aspx
Gut in verschiedene Kategorien unterteilt und mit vielen Beispielen. Habe diese Pages damals mehrfach gelesen.

Ich werde in den nächsten Kapiteln eine Applikation mit reinem Code Behind erstellen und absichtlich auch aufzeigen das man auch ohne Binding in der WPF zurechtkommt, allerdings ist der Komfort nicht gegeben welchen die WPF eigentlich bietet. Es soll jetzt niemanden Animieren kein Binding zu verwenden, ich möchte nur aufzeigen das auch dies möglich ist und mich langsam an das Binding herantastet. So denke ich, kann man besser umdenken und verstehen wie das Binding funktioniert wenn man beide Wege kennt und parallelen ziehen kann.

2.1

EINFÜHRUNG IN DIE WPF

In diesem Kapitel werden wir ein paar Beispiele durchgehen. Das Hauptaugenmerk wird hier auf XAML gelegt. Wir werden die wichtigsten Controls kennenlernen, wie wir diese Anwenden und auch untereinander kombinieren können um somit ein völlig neues Aussehen und zum Teil neues Verhalten über Trigger und Animations in das Control bekommen. Außerdem werden wir lernen wie wir ein Fenster so gestallten das dieses völlig dynamisch auf Größenänderungen reagiert.
Dabei werden die Grundlagen von DataBinding in der WPF durchgehen, erst anhand von Bindings innerhalb des Views wie Beispielsweise wie ich ein Property eines Controls auf ein

Property eines anderen Controls Binden kann wo wir auch Converter kennenlernen werden aber auch mit Binding an eine selbst geschrieben Klasse bis zum DesignTime-Support.

Dieses Kapitel wird bereits einige Videos enthalten da in einem Video einfach besser Dinge wie IntelliSense zur Geltung kommen. Bitte verzeiht mit nochmals wenn ich anfangs vielleicht nicht so geübt rüberkomme.

2.1.1

DIE WICHTIGSTEN CONTROLS UND DEREN VERWENDUNG

2.1.1.1

Wir erstellen ein WPF Projekt und schreiben XAML, wobei wir aber aus gutem Grund auf Drag&Drop in den Designer verzichten werden, warum erkläre ich im Video.
Wir lernen die wichtigsten Controls kennen und spielen damit dass sich diese an die Fenstergröße anpassen.

- *Videolink: https://www.youtube.com/watch?v=jSGgv9v8osA*

Styles, Templates, Trigger
Wir lernen Styles und Templates kennen und sehen uns an was es damit auf sich hat. Auch Trigger schneiden wir an bevor wir auch Styles zur Wiederverwendung auslagern.

2.1.1.2

Styles

Oft soll die Darstellung von Elementen desselben Typs innerhalb einer UI identisch sein. Wie im Web das CSS gibt es in der WPF hierfür Styles. Die Wiederverwendung von Stilen (Styles) erleichtert das Entwickeln und die Wartung eines UI.

Hier ein Beispiel für einen Style:

```
        <Style TargetType="{x:Type Button}">
            <Setter Property="VerticalContentAlignment" Value="Top" />
            <Setter Property="HorizontalContentAlignment" Value="Right" />
            <Setter Property="FontWeight" Value="Bold" />
            <Setter Property="Margin" Value="0,0,0,5" />
            <Setter Property="Background" Value="Aqua"/>
        </Style>
```

Styles können in XAML mit einem Key versehen oder über den Typ definiert werden. Gibt man einen Key an so kann man bei jedem Steuerelement über das „Style“ Attribut den Style als StaticResource bzw. DynamicResource angegeben.

```
<Window.Resources>
        <Style x:Key="myButtonStyle" TargetType="{x:Type Button}">
            <Setter Property="VerticalContentAlignment" Value="Top" />
            <Setter Property="HorizontalContentAlignment" Value="Right" />
            <Setter Property="FontWeight" Value="Bold" />
            <Setter Property="Margin" Value="0,0,0,5" />
            <Setter Property="Background" Value="Aqua"/>
        </Style>
    </Window.Resources>
    <Grid>
        <Button Content="Testbutton" Style="{StaticResource myButtonStyle}"/>
    </Grid>
```

Styles welchen ein „TargetType“ angegeben wird greifen auf jedes Steuerelement dieses Typs unterhalb der Hierarchie. Gibt man in den Window-Resourcen einen Style mit dem TargetType „Button“ an, greift dieser Style auf jeden Button innerhalb dieses Fensters. Auch wenn sich das Steuerelement in einem UserControl befindet welches sich im Fenster befindet greift das Style auf Buttons innerhalb des UserControls. Stichwort: Vererbung. Allerdings können jederzeit einzelne Setter eines Styles überschrieben werden. Wenn ein Style die Hintergrundfarbe von Buttons auf BLAU festlegt sind alle Buttons blau. Möchte ich das für einen Button explizit ändern ohne auf das Style für alle anderen Buttons verzichten zu müssen kann ich bei diesem Button einfach mit `Background="Green"` diesen einen Setter des Style überschreiben, die anderen Setter bleiben allerdings uneingeschränkt vorhanden.

Auch für Styles habe ich wieder ein Video für euch.

- *Videolink: https://youtu.be/VnH8wEMpf-c*

2.1.1.3

Templates

Bei Template müssen wir zunächst mal zwischen ControlTemplates, DataTemplates, ItemsPanelTemplates und HierarchialDataTemplates unterscheiden. Es ist erstmal wichtig zu wissen welche Art von Template man gerade benötigt um die aktuelle Aufgabenstellung meistern zu können. Nur wenn man weiß was für ein Template man gerade benötigt wird man über die Suchmaschine seiner Wahl auch korrekte Ergebnisse bekommen und erspart sich erstmal die Suche nach dem richtigen Begriff.

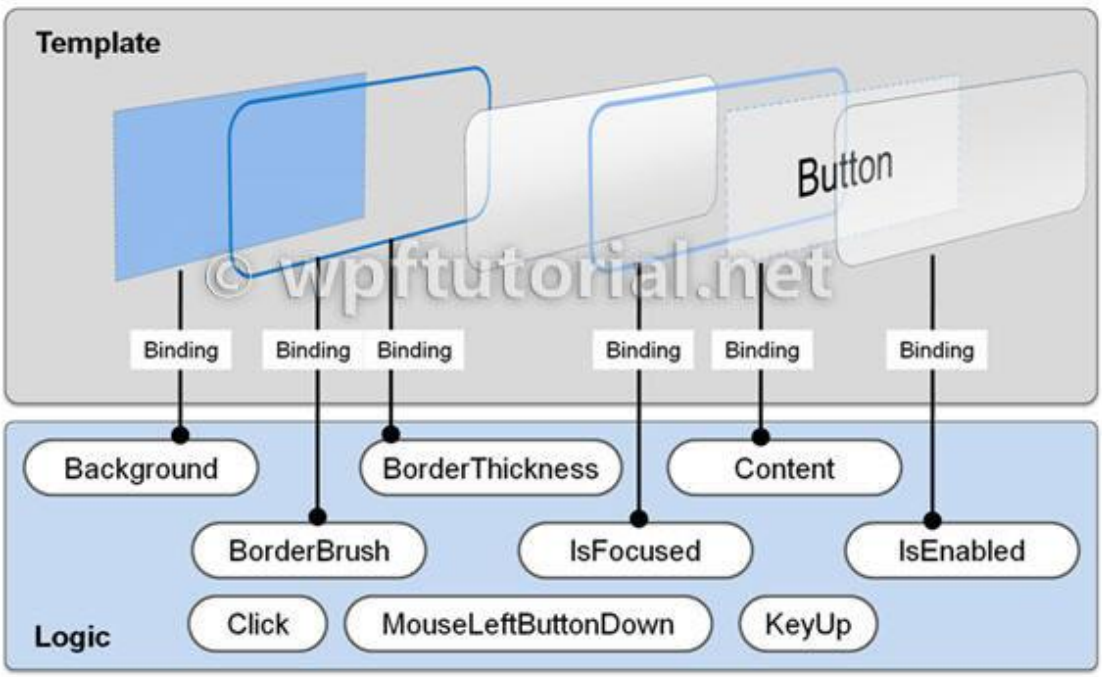
Fangen wir mit den ControlTemplates an.

Generell sind Steuerelemente in der WPF mit einer gewissen Logik versehen welche States, Styles, Events, Properties und ein Template beinhalten welche das Aussehen des Steuerelements wie z.b. einen Button definieren/steuern.
Die Verbindung zwischen dieser Logik und dem Template passiert über Binding.

Jedes Steuerelement besitzt ein Standard – Template welches für jede Windows-Version mitgeliefert wird. Unter Windows 7 sieht ein Button Beispielsweise anders aus als unter Windows 8/10. Dieses Template ist verpackt in einem Style, dem *DefaultStyleKey* und kann überschrieben werden. Diesen Style besitzt jedes UI Steuerelement.

Ein Template ist definiert über ein Dependency Property mit dem Namen *Template*.
Durch setzen dieses Property kann das Aussehen eines Steuerelementes völlig neu übernommen werden.

Hier eine sehr gute Grafik von wpftutorials.net:



Diese Grafik erklärt sehr gut wie ein solches einfaches Template aufgebaut ist und wie es auf Eigenschaftenänderung reagiert. Siehe Properties IsFocused oder IsEnabled.

Sehen wir uns eine einfache Überschreibung eines Templates anhand eines Buttons an.

```
<Style x:Key="MyButtonStyle" TargetType="Button">
            <Setter Property="Template">
                <Setter.Value>
                    <ControlTemplate TargetType="{x:Type Button}">
                        <Grid>
                            <Path Data="M 0,0 A 100,100 90 0 0 100,100 L 100,100
100,0" Fill="{TemplateBinding Background}"
                                        Stroke="{TemplateBinding BorderBrush}"/>
                            <ContentPresenter HorizontalAlignment="Center"
                                              VerticalAlignment="Center"/>
                        </Grid>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
        </Style>
```

Hier wird ein ganz einfacher Button erstellt. Dieser besitzt weder Hover noch andere Styleelemente welche einen Button auszeichnen. Beispielsweise möchten wir ja wenn wir den Button klicken, dass dieser richtig hineingedrückt wird. Wenn ein Button „Disabled“ ist soll er ausgegraut sein. All diese Funktionalitäten Fehlen hier.

Im folgenden Video sprechen wir nun darüber wie wir dies bewerkstelligen.
Außerdem zeige ich euch wie ihr das Default-Template eines jeden Controls erfahren und anpassen könnt. Dies ist insofern Praktisch, wenn Ihr ein anderes Verhalten eines Buttons bewerkstelligen wollt, ohne ein neues Control erstellen zu müssen.

Nun zu den DataTemplates

DataTemplates helfen und dabei gewisse Daten oder Klassen so darzustellen wie wir das möchten. Beispielsweise Binden wir eine Property welche Autos beinhaltet an eine ListBox. *Autos* enthält viele Instanzen von der Klasse *Auto*. Die ListBox versucht nun mit den Daten etwas anzufangen und sucht nach einem DataTemplate. Erst bei sich selbst, dann in den Resourcen von darüber liegenden Controls, bis in der Hierarchie nichts mehr vorhanden ist, dann sucht sie in der Application.xaml nach einem DataTemplate.

Nehmen wird eine einfache Klasse,- sagen wir mal diese soll ein Auto enthalten.
Also erstellen wir uns eine Klasse „Auto“.

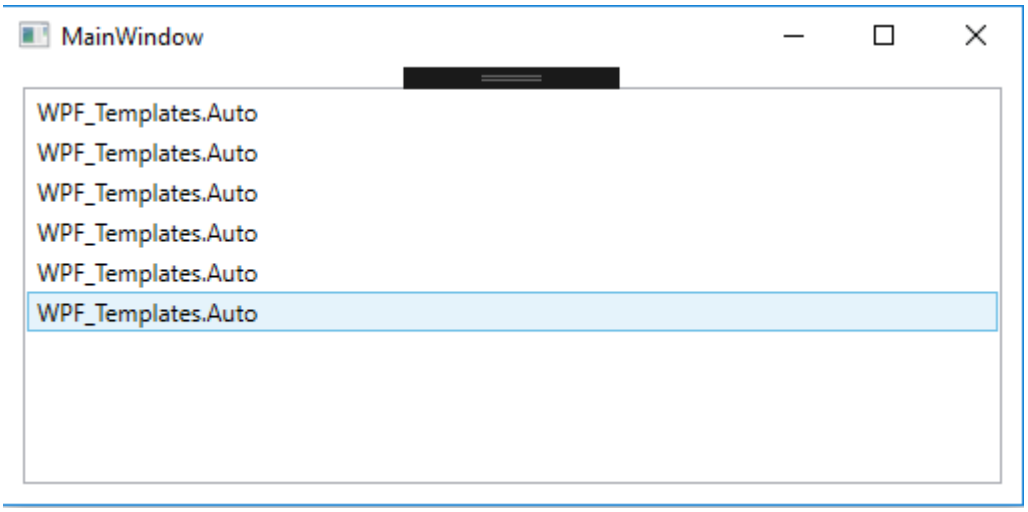
```
Public Class Auto

    Public Sub New()
    End Sub

    Public Sub New(marke As String, modell As String, ps As Integer)
        Me.Marke = marke : Me.Modell = modell : Me.PS = ps
    End Sub

    Public Sub New(marke As String, modell As String, ps As Integer, logo As String)
        Me.Marke = marke : Me.Modell = modell : Me.PS = ps : Me.Logo = logo
    End Sub


    Public Property Marke As String
    Public Property Modell As String
    Public Property PS As Integer
     Public Property Logo As String

End Class
```

In der CodeBehind des MainWindow sorgen wird jetzt einfach dafür das eine ObservableCollection (Auflistung) mit verschiedenen Autos befüllt wird.

```
Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles
Me.Loaded
        AutoListe = New ObservableCollection(Of Auto)
       With AutoListe
            .Add(New Auto("Audi", "R8 V10+", 610, "BrandImages/Audi.png"))
            .Add(New Auto("VW", "Arteon TDI 4 Motion", 240, "BrandImages/Vw.png"))
            .Add(New Auto("Seat", "Leon ST Cupra TSI DSG", 300,
"BrandImages/SEAT.png"))
            .Add(New Auto("Skoda", "Octavia RS", 230, "BrandImages/Skoda.png"))
            .Add(New Auto("Lamborghini", "Aventador LP 700", 700,
"BrandImages/Lambo.png"))
            .Add(New Auto("Bentley", "Continental Supersports", 630,
"BrandImages/Bentley.png"))
        End With

        Me.DataContext = Me
    End Sub


    Public Property AutoListe As ObservableCollection(Of Auto)
```

Im View (MainWindows.xaml) müssen wir nun nur noch eine ListBox erstellen und diese auf das Property „AutoListe“ binden. Fertig.
Aber was sehen wir nun genau?

Hier eine ListBox gebunden an *Autos* ohne DataTemplate.



Nicht so schön. Wieder schlägt der TypeConverter der WPF zu. Die WPF wirft keinen Fehler sondern versucht die Daten welche ihr übergeben werden einfach irgendwie zu Rendern. Was liegt näher als .ToString aufzurufen. Und genau das macht sie.

Wir könnten nun die ToString-Methode in der Klasse „Auto“ überschreiben um einen „verständlichen“ Text in die ListBox zu bekommen.

```
Public Overrides Function ToString() As String
      Return $"Marke: {Marke }, Modell: {Modell } Leistung: {PS }"
  End Function
```

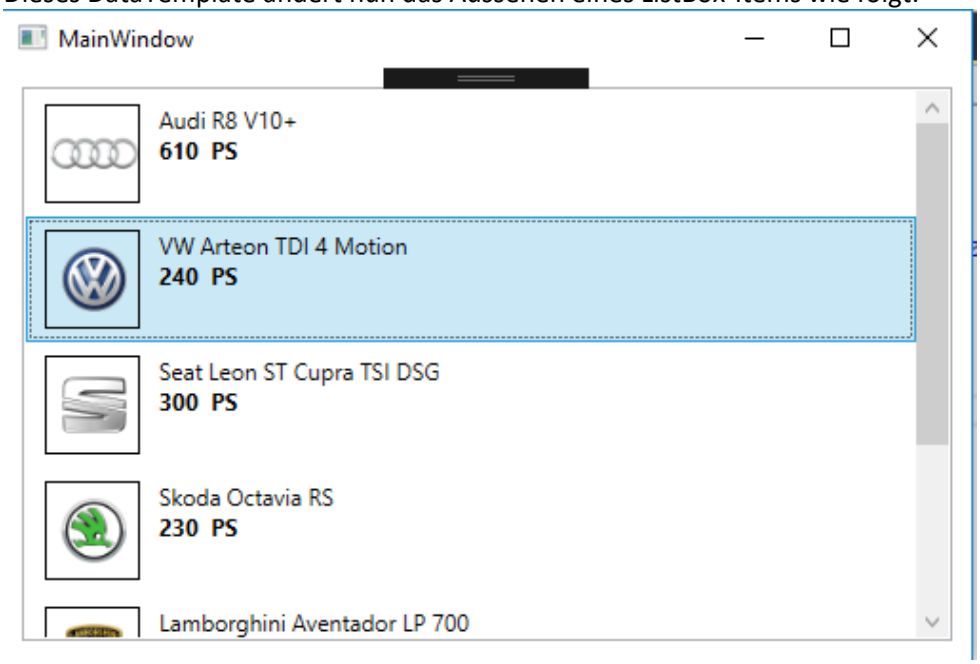
Was uns folgendes Ergebnis Liefert:

OK, schon um einiges besser, aber lange noch nicht zufriedenstellend. Wir befinden uns in der WPF, das geht ja mal sicher viel besser und schöner oder?
Richtig. Mittels DataTemplate kann man nun bestimmen wie das Template für ein ListItem aussehen soll. Template = Vorlage. Also erstellen wir eine Vorlage und geben der WPF die Info für was (Die Klasse Auto) dieses Template verwendet werden soll.

Hier ein DataTemplate direkt in das ItemTemplate der ListBox:

```
<ListBox ItemsSource="{Binding AutoListe}" Margin="10">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Grid>
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="60"/>
                            <ColumnDefinition Width="*"/>
                        </Grid.ColumnDefinitions>
                        <Border Margin="5" BorderBrush="Black" BorderThickness="1">
                            <Image Source="{Binding Logo}" Stretch="Uniform"
Width="50" Height="50" RenderOptions.BitmapScalingMode="HighQuality" />
                        </Border>
                        <StackPanel Grid.Column="1" Margin="5">
                            <StackPanel Orientation="Horizontal">
                                <TextBlock Text="{Binding Path=Marke}" />
                                <TextBlock Text="{Binding Path=Modell}"
Padding="3,0,0,0"/>
                            </StackPanel>
                            <TextBlock FontWeight="Bold" >
                                <Run Text="{Binding PS,FallbackValue=0}"/>
                                <Run Text=" PS"/>
                            </TextBlock>
                        </StackPanel>
                    </Grid>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
```

Dieses DataTemplate ändert nun das Aussehen eines ListBox-Items wie folgt:



Mehr im Video weiter unten, aber jetzt gehen wir mal weiter zu dem ItemsPanelTemplates.

ItemsPanelTemplates

Diese Art von Templates erlaubt es uns das Layout, wie Items eines ItemControls wie z.b. einer ListBox zu bestimmen. Jedes ItemsControl besitzt von Haus aus ein „Default Panel“.

Die ListBox z.b. besitzt von Haus aus ein VirtualizingStackPanel als PanelTemplate.
Dieses ist im Grunde ein normales StackPanel welches ein Zusatzfeature besitzt, welches ermöglicht das nicht alle Elemente sofort gerendert werden sondern erst dann wenn diese z.b. durch Scrollen auftauchen was Resourcen spart und stark auffällt wenn man mehrere tausend Items in der Liste hat.

Um dieses Template nun zu überschreiben, erstellen wir einfach ein ItemsPanelTemplate und suchen uns ein Panel-Control aus welches uns besser passt.

```
<ListBox ItemsSource="{Binding AutoListe}">
      <ListBox.ItemsPanel>
          <ItemsPanelTemplate>
              <UniformGrid Columns="4"/>
          </ItemsPanelTemplate>
      </ListBox.ItemsPanel>
</ListBox>
```

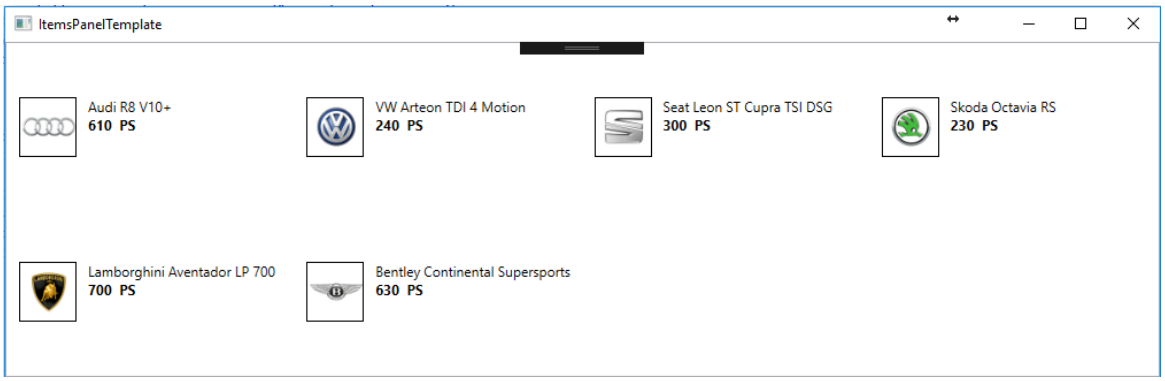
In diesem Fall nehmen wir jetzt mal ein UniformGrid mit vier Spalten. Dieses UniformGrid wird nun die Items auf vier Spalten aufteilen. Das fünfte Element würde dann in eine neue Zeile verschoben werden.

Mit dem Codeschnipsel von oben würde nun folgendes generiert werden:

Wir können sehen dass wir vier Spalten und zwei Zeilen haben. Wieder hat die WPF unsere .ToString Methode in der Klasse Auto aufgerufen. Aber das wir DataTemplates oben bereits durchgenommen haben ist es für uns ja jetzt ein leichtes ein solches DataTemplate abermals zu implementieren. Wir können gleich unser DataTemplate von oben auch in dieser ListBox verwenden, mit dem Unterschied das wir das soeben erstellte ItemsPanelTemplate auch mit in der ListBox behalten:

```
<ListBox ItemsSource="{Binding AutoListe}">
            <ListBox.ItemsPanel>
                <ItemsPanelTemplate>
                    <UniformGrid Columns="4"/>
                </ItemsPanelTemplate>
            </ListBox.ItemsPanel>
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Grid>
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="60"/>
                            <ColumnDefinition Width="*"/>
                        </Grid.ColumnDefinitions>
                        <Border Margin="5" BorderBrush="Black" BorderThickness="1">
                            <Image Source="{Binding Logo}" Stretch="Uniform"
Width="50" Height="50" RenderOptions.BitmapScalingMode="HighQuality" />
                        </Border>
                        <StackPanel Grid.Column="1" Margin="5">
                            <StackPanel Orientation="Horizontal">
                                <TextBlock Text="{Binding Path=Marke}" />
                                <TextBlock Text="{Binding Path=Modell}"
Padding="3,0,0,0"/>
                            </StackPanel>
                            <TextBlock FontWeight="Bold" >
                                <Run Text="{Binding PS,FallbackValue=0}"/>
                                <Run Text=" PS"/>
                            </TextBlock>
                        </StackPanel>
                    </Grid>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
```

Hierdurch erhalten wir wieder die Optik unseres Items aber in der Anordnung wie wir es im ItemsPanelTemplate vorgegeben hatten.

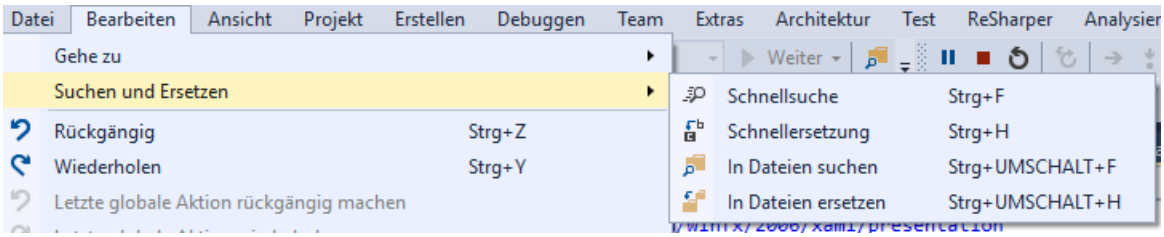


Wir sehen, alle Elemente werden so gerendert wie wir das haben wollten und in dem Schema wie wir dies vorgegeben haben. So eröffnen sich tolle Möglichkeiten für eine neue Anwendungsoptik und eine bessere und flexiblere Benutzerführung.
Langsam dürfte wiedermal ein kleiner WOW-Effekt einsetzen was nun alles möglich ist mit ein paar Zeilen XAML ;-)

Last but not least, **HierarchialDataTemplates**

Mit HierarchialDataTemplate wie der Name vermuten lässt kann ich bestimmen wie Daten welche in Form einer Hierarchie vorliegen darstellen.
Tja, wo liegt nun der Einsatz von einem solchen Template? Z.b. bei der Verwendung eines TreeViews oder eines Menus.

Wir alle kennen ein Menu:

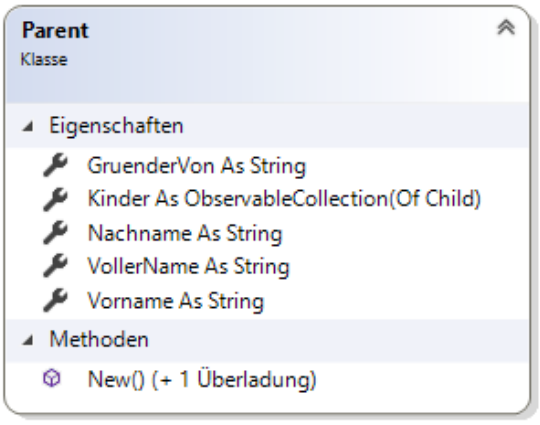


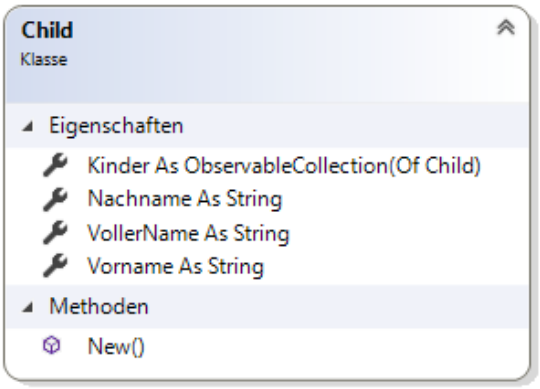
Erstmal haben wir die erste Ebene (Root) mit „Datei“, „Bearbeiten“, Ansicht“, usw.
Danach haben wir eine Unbestimmte Anzahl an Ebenen welche wieder eine unbestimmte Anzahl an weiteren Kind-Ebenen haben kann.

Auch für solch einen Fall hat die WPF vorgesorgt, und ermöglicht uns das Binding.
Damit wir allerdings auch bestimmen können wie die Darstellung passiert gibt es eben die HierarchialDataTemplates. Besser darstellen lässt sich solch ein unterfangen mit einem TreeView.

Als Fallbeispiel nehmen wir mal eine Art Stammbaum.
Wie soll es anders sein nehme ich den Stammbaum der Familie Porsche/Piech:

Erstmal erstelle ich zwei Klassen. Zum einen die Klasse „Parent“ und zum anderen die Klasse „Child“. Ich denke die Namen sprechen für sich.





Jedes Parent kann X Childs (Kinder) haben. Jedoch kann jedes ChildElement wieder X Childs (Kinder) haben, usw.

Füllen wir diese Klassen mit ein paar Daten von hier und konzentrieren wir uns wieder auf den XAML.

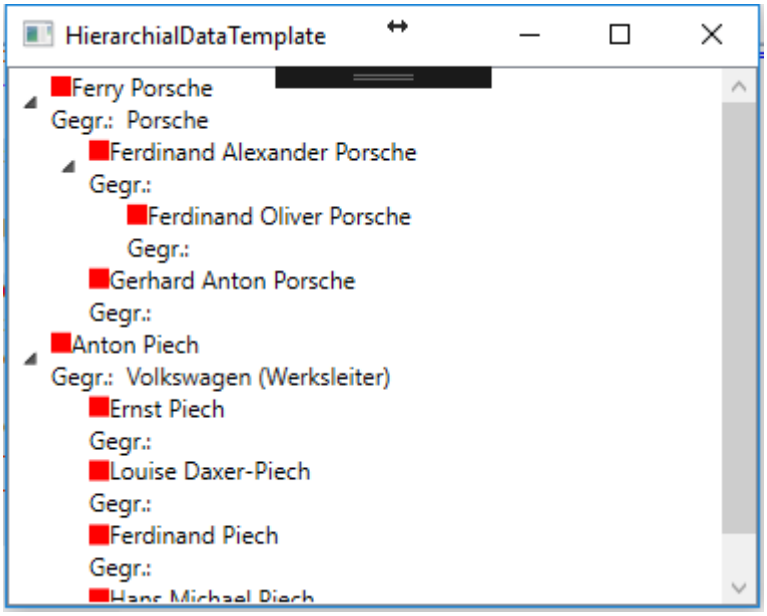
Wir erstellen ein TreeView und geben ein HierarchialDataTemplate an.

```
<TreeView DataContext="{Binding}" ItemsSource="{Binding HierarchieDaten}">
            <TreeView.ItemTemplate>
                <HierarchicalDataTemplate ItemsSource="{Binding Kinder}">
                    <StackPanel>
                        <StackPanel Orientation="Horizontal">
                        <Rectangle Width="10" Height="10" Fill="Red"/>
                        <TextBlock Text="{Binding VollerName}"/>
                        </StackPanel>
                            <TextBlock>
                            <Run Text="Gegr.: "/>
                            <Run Text="{Binding GruenderVon}"/>
                        </TextBlock>

                    </StackPanel>
                </HierarchicalDataTemplate>
            </TreeView.ItemTemplate>
        </TreeView>
```

Das sieht ja gar nicht so kompliziert aus. Das TreeView selbst haben wir an ein Property mit dem Namen „HierachieDaten“ gebunden. Dieses befindet sich in der CodeBehind und ist vom Typ Parent.
Wie wir im Diagramm sehen konnten hat dieses eine Property Kinder. Dieses kann viele Child halten und Child wiederum auch viele Childs.

Beim Erstellen des HierarchialDataTemplates geben wird für dieses dann als ItemsSource das Property an welches innerhalb der Elternklasse die Childs hält. Und der Rest ist einfach nur XAML welches bestimmt wie ein TreeViewItem dann für dieses Template aussehen soll.

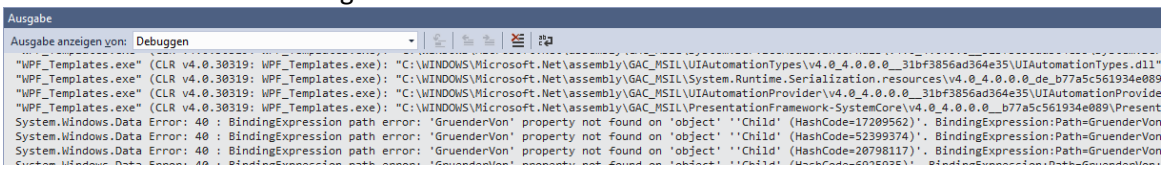
Folgende Ansicht wird in diesem Fall generiert:



Soweit sieht das ja schon ganz gut aus. Wie wir aber sehen können hat nicht jedes Kind der Familie wieder etwas oder eine Marke gegründet. Da wir aber im Template hinterlegt haben das

ein Item so gerendert werden soll wird uns der Text „Gegr.:“ angezeigt, da die Klasse Child allerdings kein solches Property besitzt kann die WPF gar nicht darauf Binden.

Dies macht sich auch im Ausgabefenster bemerkbar:



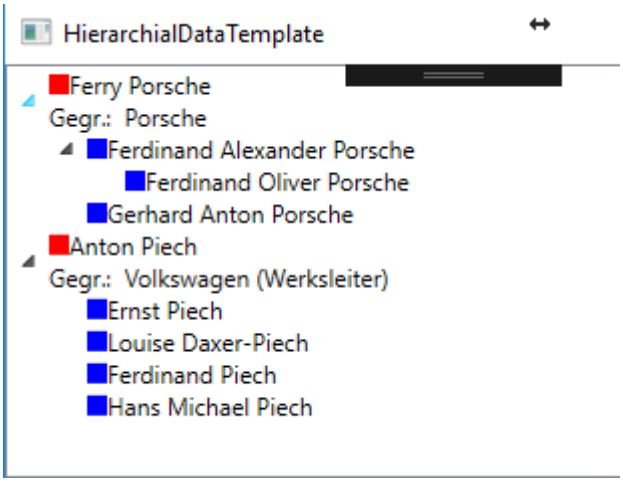
OK, wir wissen also das wir nicht am Holzweg sind, aber es hier noch Luft nach oben gibt, es muss ja möglich sein für jede Art von Element eine eigene Optik zu definieren.
Naja, vorher hatten wir gerade noch die DataTemplates gelernt, das wäre doch eine Möglichkeit gleich das gelernte anhand dieses Beispiels zu versuchen.

Wir müssen also ein wenig umdenken. Wir möchten mehrere HierarchialDataTemplates definieren, werden jedoch scheitern wenn wir versuchen noch ein Template darunter zu erstellen. Wir versuchen es in den Ressourcen:

```
  <TreeView DataContext="{Binding}" ItemsSource="{Binding HierarchieDaten}">
            <TreeView.Resources>
                <HierarchicalDataTemplate ItemsSource="{Binding Kinder}"
DataType="{x:Type local:Parent}">
                    <StackPanel>
                        <StackPanel Orientation="Horizontal">
                            <Rectangle Width="10" Height="10" Fill="Red"/>
                            <TextBlock Text="{Binding VollerName}"/>
                        </StackPanel>
                        <TextBlock>
                            <Run Text="Gegr.: "/>
                            <Run Text="{Binding GruenderVon}"/>
                        </TextBlock>
                    </StackPanel>
                </HierarchicalDataTemplate>
                <HierarchicalDataTemplate ItemsSource="{Binding Kinder}"
DataType="{x:Type local:Child}">
                    <StackPanel Orientation="Horizontal">
                        <Rectangle Width="10" Height="10" Fill="Blue"/>
                        <TextBlock Text="{Binding VollerName}"/>
                    </StackPanel>
                </HierarchicalDataTemplate>
            </TreeView.Resources>
        </TreeView>
```

Wenn wir uns dieses TreeView ansehen werden wir die Stirn runzeln.
Zwei Templates? JA, das erste Template dient zur Anzeige eines Parent-Objekts. Zu erkennen an der Angabe des DataType: `DataType="{x:Type local:Parent}"`
Das zweite Template dient zur Anzeige des Child-Objekts: `DataType="{x:Type local:Child}"`
Die WPF entscheidet nun anhand des Übergebenen Typs jedes Knotens wie dieser gerendert werden soll.

Hier das Ergebnis:



Man kann schön erkennen das beim Child-Knoten nicht nur das Quadrat in einer anderen Farbe erstrahlt sondern auch das hier der TextBlock mit „Gegr.:“ fehlt, da wir diesen TextBlock ins Child-Template nicht mit eingebaut haben.
Eine schöne und saubere Sache.

In folgendem Video gehe ich etwas näher darauf ein und zeige euch verschiedene Templates anhand von einigen Beispielen:

- *Videolink: https://www.youtube.com/watch?v=QpZmfQz0rek*

Trigger

Auch bei Triggern müssen wir wieder unterscheiden da es hier mehrere Arten von Triggern gibt. Folgende Trigger gibt es:

- Trigger
- DataTrigger
- MultiTrigger
- MultiDataTrigger
- EventTrigger

Trigger werden überwiegend in Styles oder ControlTemplates verwendet. Er Triggert ein Property auf ein anderes Property dieses Controls. Beispiel: Ein Trigger soll die Hintergrundfarbe des Buttons auf ROT setzen wenn das Property IsMouseOver True ist.

```
<Style x:Key="ButtonStyle" TargetType="{x:Type Button}">
    <Style.Triggers>
        <Trigger Property="IsMouseOver" Value="True">
            <Setter Property="Background" Value="Red" />
        </Trigger>
    </Style.Triggers>
</Style>
```

Ein DataTrigger wird bei Datenbindung eingesetzt und wird überwiegend in DataTemplates verwendet. Beispielsweise kann ein DataTrigger verwendet werden um die Hintergrundfarbe eines Controls in ROT zu ändern wenn das Property Alert der Klasse True ist. DataTrigger können allerdings auch in ControlTemplates nützlich sein um einen roten Rahmen um ein Controls zu machen wenn in der Klasse *Kontoinfo* das Property *Kontostand* weniger als 0 Beträgt. Hierfür kann ein IValueConverter bestimmten das bei negativen Werten der Converter True zurückgibt und sohin der Rahmen Rot gezeichnet wird.

```
<DataTrigger Binding="{Binding Alert}" Value="True">
      <Setter Property="Background" Value="Red"></Setter>
</DataTrigger>
```

MultiTrigger und MultiDataTrigger sind im Grunde dasselbe wie die beiden Triggerarten oben nur das mehrere Bedingungen angegeben werden können. Ein MultiDataTrigger würde dann nur greifen wenn alle Bedingungen erfüllt sind.

```
<DataTemplate.Triggers>
    <MultiDataTrigger>
        <MultiDataTrigger.Conditions>
            <Condition Binding="{Binding Path=Picture}" Value="{x:Null}" />
            <Condition Binding="{Binding Path=Title}" Value="Waterfall" />
        </MultiDataTrigger.Conditions>
        <MultiDataTrigger.Setters>
           <Setter TargetName="viewImage" Property="Source"
Value="/Images/noImage.png"/>
           <Setter TargetName="viewImage" Property="Opacity" Value="0.5" />
           <Setter TargetName="viewText" Property="Background" Value="Brown" />
        </MultiDataTrigger.Setters>
    </MultiDataTrigger>
</DataTemplate.Triggers>
```

EventTrigger schließlich sind Trigger welche auf ein Event reagieren können.
Beispiel: Der Hintergrund eines Buttons soll sich ändern wenn sich die Maus über den Button fährt. EventTrigger werden Beispielsweise oft in Storyboards verwendet um z.b. eine Animation zu starten.

```
<Border Name="border1" Width="100" Height="30"
                BorderBrush="Black" BorderThickness="1">
            <Border.Background>
                <SolidColorBrush x:Name="MyBorder" Color="LightBlue" />
            </Border.Background>
            <Border.Triggers>
                <EventTrigger RoutedEvent="Mouse.MouseEnter">
                    <BeginStoryboard>
                        <Storyboard>
                            <ColorAnimation Duration="0:0:1"
                                            Storyboard.TargetName="MyBorder"
                                            Storyboard.TargetProperty="Color"
To="Gray" />
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger>
            </Border.Triggers>
        </Border>
```

Da Animationen wiederum ein oder mehrere Kapiteln füllen würde, ich darauf aber nicht näher eingehen werde da ich der Meinung bis das diese eigentlich relativ selten gebraucht werden, hier ein Link wo Ihr alle Wissenswertes über Animationen gut erklärt bekommt. WPF Basic Animations
Wer sich hier durchklickt kommt bald zu den Easing Functions, zu den Key Frame Animations und schließlich zu den PathAnimations.

In folgendem Video zeige ich euch nun wie wir mit Triggern umgehen und wie wir das Verhalten von Controls beeinflussen können ohne viele, viele Codezeilen schreiben zu müssen. Selbst wenn wir Converter schreiben bestehen diese meist nur aus 1-5 Zeilen Code.
Eine saubere und übersichtliche Sache:

- *Videolink: https://youtu.be/iZVziwGqrXs*

2.1.2

XAML Namespaces

Kurze Theorie

In diesem Thema widmen wir uns den XAML-Namespaces.
Wir klären was es mit den Namespaces auf sich hat und wie wir diese Verwenden und eigene Namespaces definieren.

Was ist ein XAML-Namespace?

Wie in XML-Namespaces ist ein XAML-Namespace eine Erweiterung dieses Konzepts.
Eine etwas technische Beschreibung des Begriffs gibt es auf MSDN unter folgenden Link:
https://msdn.microsoft.com/de-de/library/ms747086(v=vs.110).aspx

Ich möchte es allerdings etwas verständlicher beschreiben. Im Grunde kann man sich einen Namespaceimport vorstellen wie in VB einen "Import" im Code.

Eine ganz gute Beschreibung findet man auch im vb-paradise Forum vom User „ErfinderDesRades“ unter folgendem Link:

http://www.vb-paradise.de/index.php/Thread/117451-Grundlagen-Xaml-Syntax/?postID=1022811#post1022811

Im Grunde kann jeder CLR Namespace auch in XAML eingebunden werden. Beispiel:

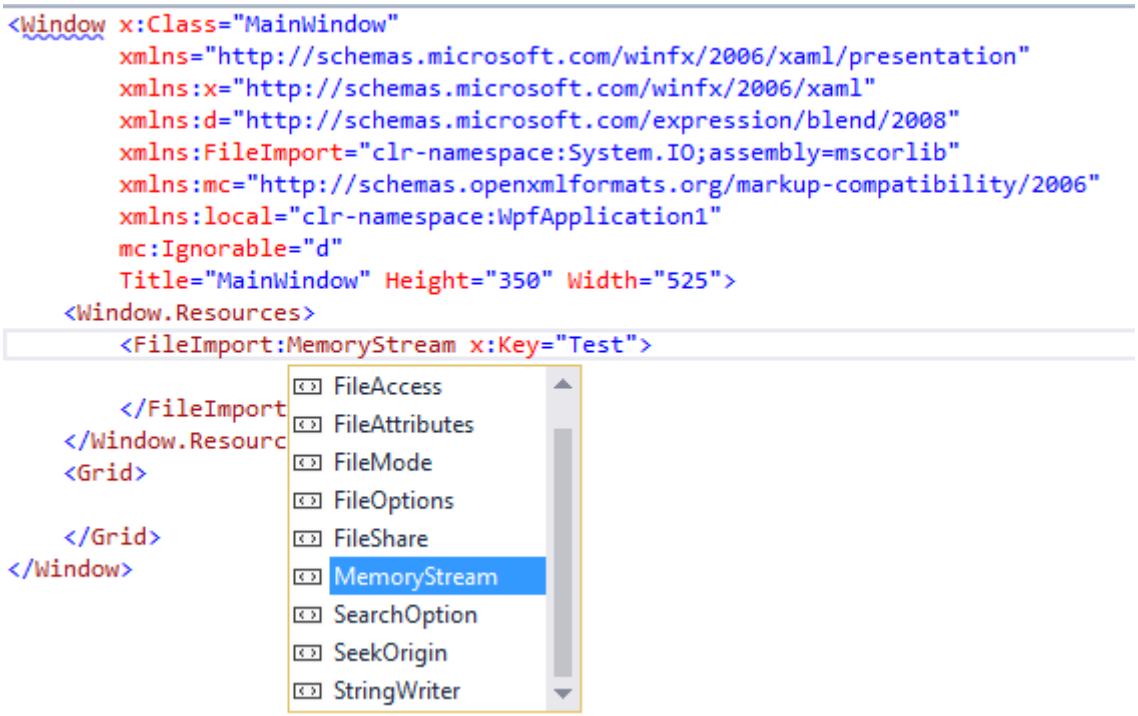
```
Imports FileImport = System.IO

Module Module1

    Sub Main()
        If FileImport.File.Exists(path) Then
            'Do something
        End If
    End Sub

End Module
```

Solch ein Import im Code könnte wie folgt in XAML aussehen:

```
xmlns:FileImport="clr-namespace:System.IO;assembly=mscorlib"
```

Der Sinn eines Imports des System.IO Namespace sei jetzt mal dahingestellt, damit soll nur gezeigt werden das wirklich jeder beliebige Namespace eingebunden werden könnte wie

folgender Screenshot zeigt:



In einem Window sehen wir auch immer einen Namespace (im obigen Beispiel in der zweiten Zeile) welcher kein Präfix hat.

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

Dieser Namespace wird als „Default“ gesehen. Es muss also kein Präfix vorangestellt werden. Beispielsweise befinden sich alle WPF Controls in diesem Namespace wie z.b. ein Button weshalb man in XAML einfach <Button/> schreiben kann um einen Button zu erstellen.

Jetzt fällt uns an diesem Beispiel aber auch auf das wir Namespaces über den CLR Namen eingebunden haben und andere wiederum mit einer URL. Aber was hat die URL zu bedeuten? Dazu später mehr.

Erstellung eigener Namespaces

Wir erstellen uns im Projekt eine Klasse mit dem Namen „FahrzeugeVm“ sowie eine Klasse Fahrzeug. Die FahrzeugeVm-Klasse enthält ein Property mit dem Namen „Fahrzeuge“ vom Typ ObservableCollection(Of Fahrzeug). Fahrzeuge kann also viele Instanzen von Fahrzeug enthalten. Beide Klassen befinden sich im Namespace ViewModel unseres Projekts.
Ich setzt jetzt mal voraus das jeder Namespaces in VB.Net kennt und weis was es mit Namespaces auf sich hat und wie diese funktionieren. Ansonsten kann dies hier in den Microsoft Docs nachgelesen werden.

Zur Vereinfachung habe ich hier beide Klassen in eine Datei geschrieben.

```
Imports System.Collections.ObjectModel

Namespace ViewModel
    Public Class FahrzeugeVm
        Public Sub New()
            Fahrzeuge = New ObservableCollection(Of Fahrzeug) From {
                New Fahrzeug() With {.Marke = "Volkswagen", .Modell = "Passat", .LeistungPs = 128},
                New Fahrzeug() With {.Marke = "Seat", .Modell = "Ibiza", .LeistungPs = 89},
                New Fahrzeug() With {.Marke = "Audi", .Modell = "A3", .LeistungPs = 150}}
        End Sub

        Public Property Fahrzeuge As ObservableCollection(Of Fahrzeug)
    End Class

    Public Class Fahrzeug
        Public Property Marke As String
        Public Property Modell As String
        Public Property LeistungPs As Integer

        Public Overrides Function ToString() As String
            Return $"{Marke} {Modell} hat {LeistungPs} PS"
        End Function
    End Class
End Namespace
```

In einem Window möchten wir die Fahrzeuge in einer ListBox darstellen damit wir alle Fahrzeuge sehen können. Wir erstellen uns in unserem Fenster eine ListBox.

```
<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:_2_1_2_2_XamlNamespaces"
        mc:Ignorable="d"
        Title="MainWindow" Height="150" Width="200">
    <Grid>
        <ListBox SelectedIndex="0"/>
    </Grid>
</Window>
```

Als nächstes möchten wir für unser Window einen Datencontext (DataContext) festlegen damit wir die ListBox darauf binden und sohin die Fahrzeuge anzeigen lassen können.

Wir fügen also erstmal unseren Namespace ViewModel in dem XAML Namespaces an und vergeben ein Präfix. Ich entscheide mich jetzt mal für den Präfix „vm“.

```
xmlns:vm="clr-namespace:_2_1_2_2_XamlNamespaces.ViewModel"
```

Dann sieht unser XAML Code folgendermaßen aus:

```
<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:_2_1_2_2_XamlNamespaces"
        xmlns:vm="clr-namespace:_2_1_2_2_XamlNamespaces.ViewModel"
        mc:Ignorable="d" Title="MainWindow" Height="150" Width="200">
    <Grid>
        <ListBox SelectedIndex="0"/>
    </Grid>
</Window>
```

Auch hier haben wir IntelliSense wenn wir zwischendurch mal Kompiliert haben.
Einfach: xmlns:vm= eingeben und es erscheint eine Liste von verfügbaren Namespaces.
Hier muss der Namespace auch nicht mühselig rausgesucht werden. Einfach beginnen „viewmodel“ einzugeben und schon werden die Ergebnisse gefiltert. Siehe Video zu diesem Kapitel.

- ***Praxistip:*** *Habt ihr mal einen Namespace oder ein Property nicht in der IntelliSense kann dies daran liegen dass zwischenzeitlich nicht kompiliert wurde. Hin und wieder mal das komplette Projekt zu kompilieren (im Menu Erstellen -> Projektmappe erstellen oder mit der Tastenkombination „STRG + SHIFT + B“ kann oft „wunder“ wirken.*

Jetzt haben wir unseren eigenen Namespace in XAML eingebunden und können nun unseren DataContext für unser Window setzen, denn unsere FahrzeugeVm-Klasse möchten wir nun als Grundlage für unser Fenster verwenden.

Das machen wir indem wir den DataContext unseres Fensters setzen mit:

```
<Window.DataContext>
    <vm:FahrzeugeVm/>
</Window.DataContext>
```

Ab jetzt hat das Fenster und durch die Vererbung alle dem Fenster untergeordneten Objekte unsere FahrzeugeVm Klasse als Datenkontext und können somit auf dessen Eigenschaften zugreifen was in unserem Fall ja „nur“ die Eigenschaft „Fahrzeuge“ ist.

Unser Window sieht nun wie folgt aus:

```
<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:_2_1_2_2_XamlNamespaces"
        xmlns:vm="clr-namespace:_2_1_2_2_XamlNamespaces.ViewModel"
        mc:Ignorable="d" Title="MainWindow" Height="150" Width="200">
    <Window.DataContext>
        <vm:FahrzeugeVm/>
    </Window.DataContext>
    <Grid>
        <ListBox SelectedIndex="0"/>
    </Grid>
</Window>
```

Wenn wir nun unsere ListBox an unser Property in der FahrzeugeVm Klasse binden möchten müssen wir nur das ItemsSource Property der ListBox an das Property Fahrzeuge in unserem DatenContext Binden.

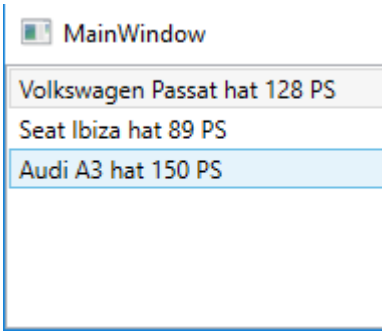
```
    <Grid>
        <ListBox ItemsSource="{Binding Fahrzeuge}" SelectedIndex="0"/>
    </Grid>
```

Es wird euch sicher aufgefallen sein das wir hier kein IntelliSense hatten als wird Das Binding eingegeben haben. Das liegt daran das wir zwar den DatenContext für die Laufzeit eingegeben haben aber keinen für die DesignTime (Entwicklungszeit). Dies kommt zwar noch genauer in einem späteren Kapitel aber ich zeige es hier schon mal schnell vor:

Innerhalb des Window-Knoten geben wir wie folgt den DesignTime-DataContext an:

```
d:DataContext="{d:DesignInstance IsDesignTimeCreatable=True, Type={x:Type
vm:FahrzeugeVm}}"
```

Wie man sieht machen wir auch hier wieder Gebrauch von unserem Namespace.
Jetzt haben wir auch zum Binden die IntelliSense zur Verfügung. Starten wir nun unser Programm sehen wir die Einträge in der ListBox:



Vorhin in diesem Kapitel haben wir ja bereits gesehen das es auch Namespaces gibt welche mit einer URI statt mit dem CLR Namespace angegeben werden, dies kann mehrere Vorteile haben. Der wohl größte ist das man hierbei mehrere Namespaces zusammenführen kann.

Beispiel: Der per Default ohne Präfix in jedem Window vorhandene Namespace enthält alle Controls welche es für die WPF von Microsoft Seite gibt. Aber ihr könnt euch vorstellen dass sich nicht alle Controls in ein und demselben CLR Namespace befinden. Damit man nicht X Namespaces in jedem Window immer und immer wieder einbinden muss gibt es diese Möglichkeit.

Beim Namespace http://schemas.microsoft.com/winfx/2006/xaml/presentation beispielsweise sind unter anderem z.b. die Namespaces System.Windows und System.Windows.Controls eingebunden.

Geben wir die URL in den Browser ein bekommen wir entweder eine 404 Antwort (Die Seite kann nicht gefunden werden) oder wir bekommen einfach einen Text das die Seite nicht verfügbar ist.

Aber wie weis die WPF was man mit der URL meint, und viel wichtiger, woher kommt die URL? Dies wird klarer, wenn wir selbst versuchen einen solchen Namespace zu definieren.

Im folgenden Video werden wir das soeben gelesene nochmals durchgehen und auch Namespaces mit URI definieren und einsetzen.

https://www.youtube.com/watch?v=Rm7u4VP9vQA

2.1.3

Resourcen

Was sind Resourcen und was bringen sie mir

Grundsätzlich gibt es in einer Anwendung zwei Arten von Resourcen. Die XAML Resourcen und die Anwendungsresourcen. Eine Resource ist ein Objekt welches an unterschiedlichen Stellen einer Anwendung erneut verwendet werden kann.

Die Anwendungsresourcen kennt Ihr sicher aus WinForms auch, dies sind nicht ausführbare Datendateien die eine Anwendung zur Laufzeit benötigt wie z.B. ein Icon oder Bild.
Ein Beispiel für XAML Resourcen sind Brushes oder Styles. Diese können an einer Stelle definiert werden und stehen anschließend zur Verfügung.

In XAML besitzt jedes Framework-Element (FrameworkElement oder FrameworkContentElement) eine Resources-Eigenschaft. Alle Elemente welche von FrameworkElement erben besitzen somit auch die Resources Eigenschaft.

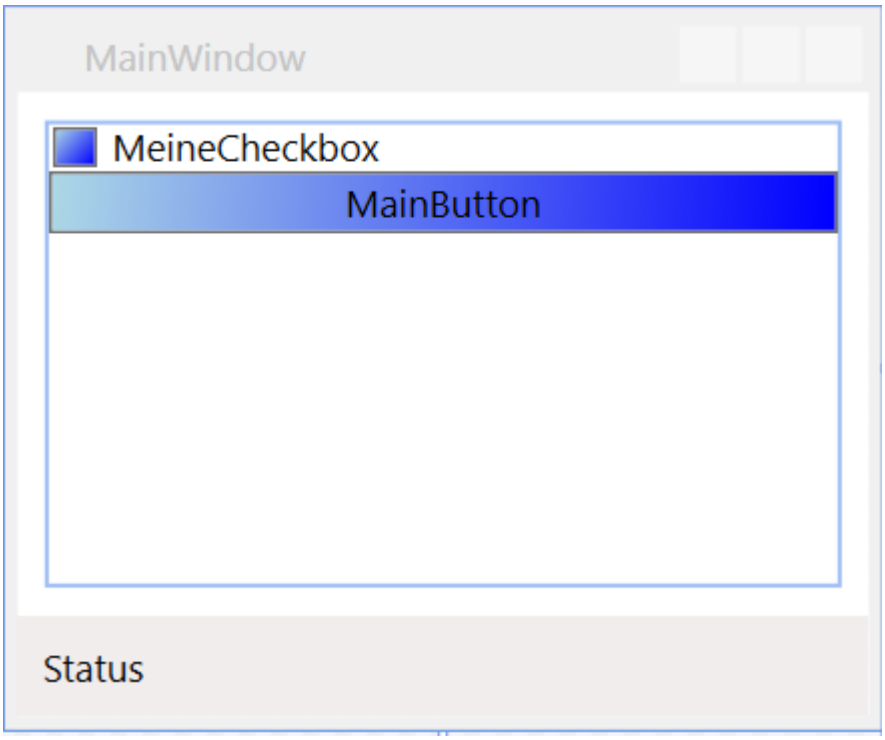
Hier ist interessant zu wissen das Resourcen "vererbt" werden. Vererben ist zwar nicht ganz das richtige Wort jedoch kann man sich darunter schon mal ungefähr etwas vorstellen.

Angenommen wir haben ein Window.
In diesem Window befindet sich ein DockPanel mit zwei Controls. Einer Statusleiste unten und einem StackPanel oben.
Definieren wir Resourcen innerhalb des StackPanels kann das StackPanel sowie auch alle darunter liegenden Elemente auf diese Resourcen zugreifen. Sowohl die Statusbar als auch das DockPanel kann nicht darauf zugreifen da sich die Resourcen ja unterhalb dieses Elements befinden.

Folgender XAML Code setzt diesen Test um:

```
<DockPanel LastChildFill="True">
            <StatusBar DockPanel.Dock="Bottom">
                <Label Content="Status"/>
            </StatusBar>
            <StackPanel Margin="10">
                <StackPanel.Resources>
                    <LinearGradientBrush x:Key="MyTestBackgroundBrush">
                        <GradientStop Color="LightBlue"/>
                        <GradientStop Color="Blue" Offset="1"/>
                    </LinearGradientBrush>
                </StackPanel.Resources>
                <CheckBox Content="MeineCheckbox" Background="{StaticResource
MyTestBackgroundBrush}"/>
                <Button Content="MainButton" Background="{StaticResource
MyTestBackgroundBrush}"/>
            </StackPanel>
        </DockPanel>
```

Folgendes Window wird im Designer angezeigt:



Versuchen wir nun Beispielsweise dem Label in der Statusbar diese Resource zuzuweisen zeigt hier bereits der Designer einen Fehler dass die gesuchte Resource nicht gefunden werden kann.

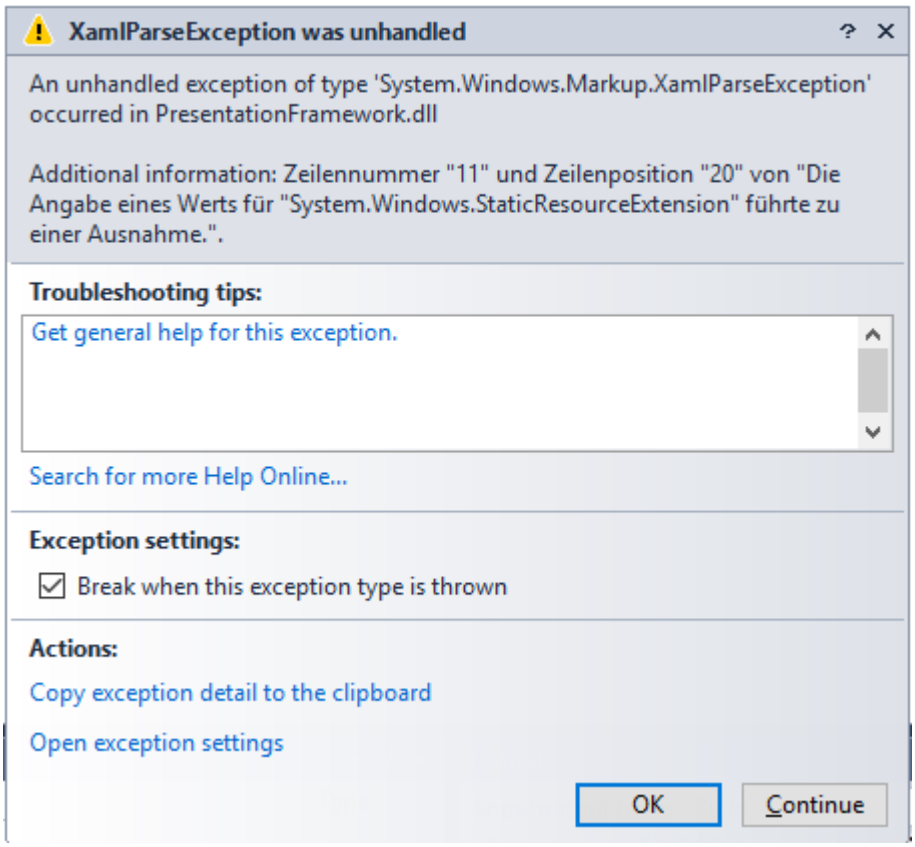
```
Panel LastChildFill="True">
StatusBar DockPanel.Dock="Bottom">
   <Label Content="Status" Background="{StaticResource MyTestBackgroundBrush}"/>
/StatusBar>
StackPanel Margin="10">
   <StackPanel.Resources>
```

The resource "" could not be resolved.

Das lässt sich allerdings trotz angezeigtem Fehler kompilieren und starten. Warum?

Es kann ja gut sein das diese Resource zur Laufzeit zur Verfügung steht; Zum einen können Resourcen auch im Code erstellt werden zum anderen können XAML Files auch zur Laufzeit dynamisch nachgeladen werden oder es werden Assemblys geladen welche Resourcen enthalten.

Wird dies allerdings nicht gemacht quittiert uns das der Debugger mit einer schönen 'XamlParseException'.



Hier auch die Info:
Angabe eines Werts für "System.Windows.StaticResourceExtension" führte zu einer Ausnahme."

Selbst wenn wir nun auf "Continue" klicken wird das Debugging beendet.

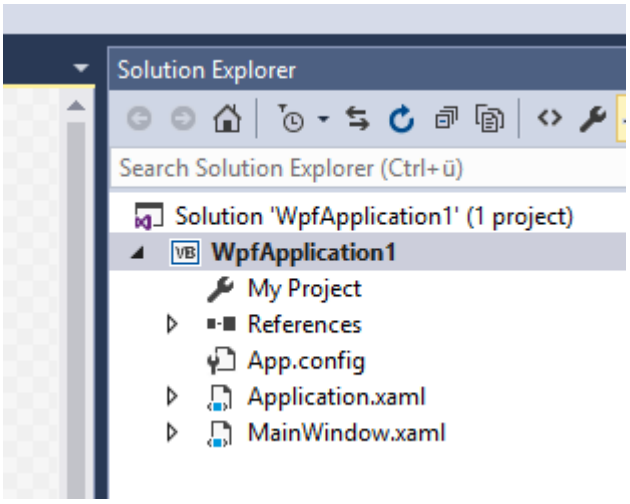
Also wo könnten wir nun die Resource definieren um diese sowohl in den Controls des StackPanels als auch in den Controls der Statusbar zur Verfügung zu haben?

Sowohl die Statusbar als auch das StackPanel befinden sich im DockPanel. Also ist das Dockpanel der "tiefste" Punkt an welchem diese Resource definiert werden kann damit all die genannten Controls auch auf diese Resource zugreifen können.

Ist eine Resource in den Resourcen des Window definiert gilt diese folglich für alle Elemente welches sich in diesem Window befinden.

Spätestens hier fragen sich schon einige wo den der "höchste" Punkt für Resourcen wäre. Den immer wieder erneut alles im Window zu definieren ist ja nicht unbedingt das Gelbe vom Ei. Der höchste Punkt an sich sind die verwalteten Resourcen des Betriebssystems. Denn in MS Windows gibt es seit Windows XP auch verschiedene Themes.

Für unser Programm wäre es allerdings die Application.xaml welche sich im Hauptknoten unserer Anwendung befindet.



Dieses File beinhaltet im Grunde nur ein leeres ResourceDictionary. Alle Resources, Styles und DataTemplates welche hier definiert werden sind von der ganzen Anwendung zugänglich.
Wir haben ja bereits gelernt das wenn beispielsweise für einen Style kein Key sondern lediglich ein TargetType angegeben wird dieser für alle Elemente dieses Typs gelten.
Haben wir z.b. einen Style ohne Angabe eines Keys welcher im Setter das Margin eines Buttons auf "15" festgelegt gilt dieses Margin für alle Elemente darunter.

Ist dieses Style in der Application.xaml angegeben gilt dieses für alle Button des Projekts.

Aber Moment mal!

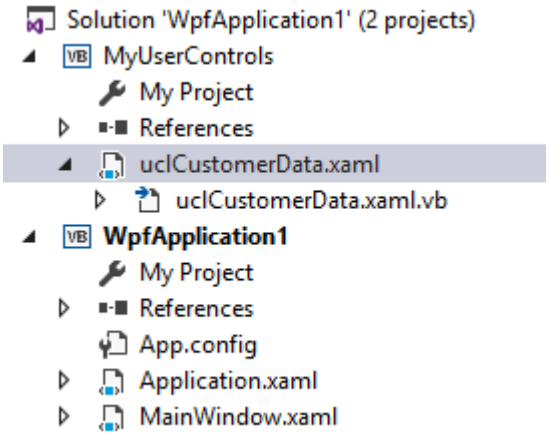
"Für dieses Projekt" würde doch bedeuten das, wenn ich Beispielsweise Controls oder UserControls in eine DLL auslagere diese das Margin nicht übernehmen richtig?

Ja, teilweise. Erstelle ich in einer Assembly ein Control mit einem Button bekommt dieser Button erstmal diesen Style nicht zugeordnet. Packe ich dieses UserControl allerdings dann innerhalb meiner App (in welcher das Margin definiert ist) in ein Window oder wiederum in ein UserControl bekommt der Button sehr wohl das Margin da dieser sich nun innerhalb der Anwendung befindet wo im Style ein Margin für alle Buttons definiert ist.

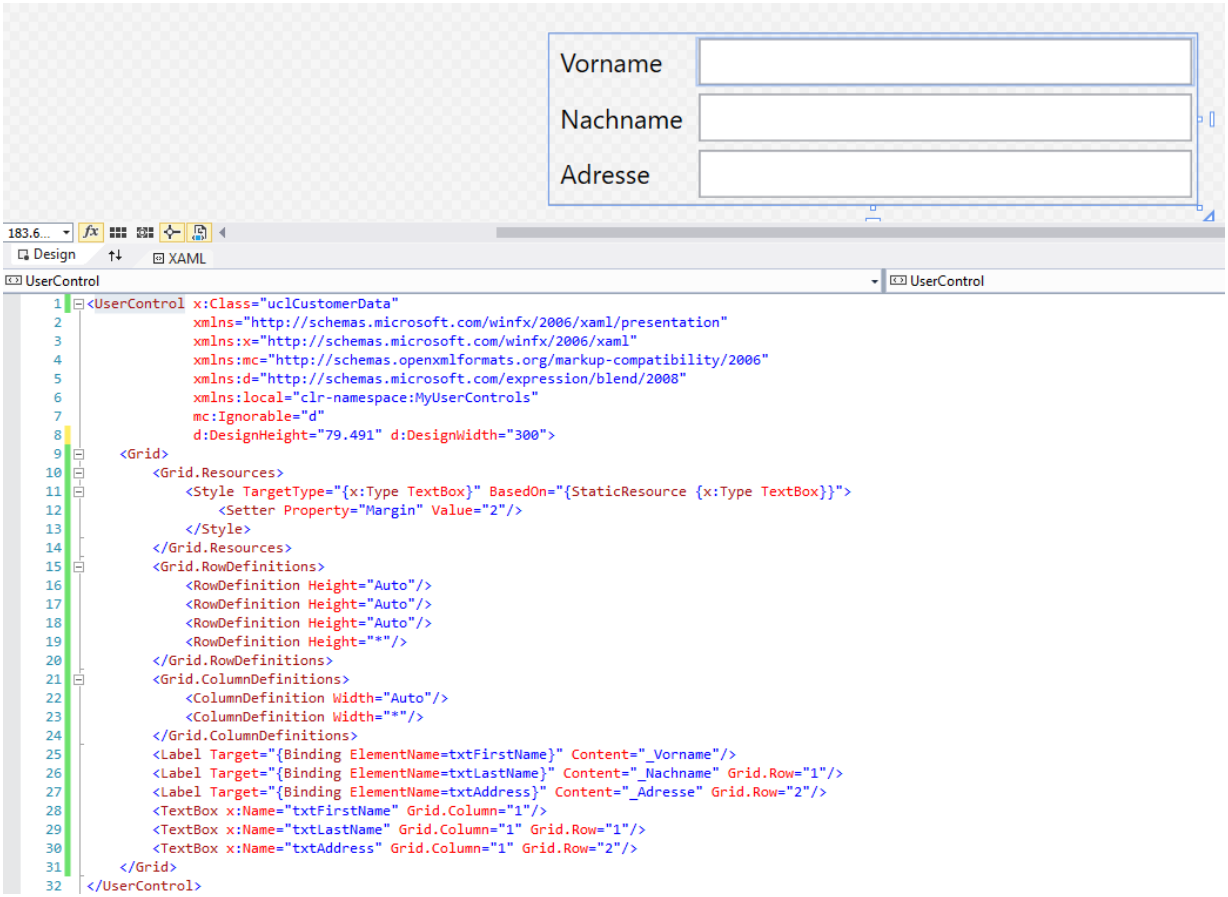
Folgende Vorgehensweise zeigt das Verhalten wie ich finde ganz gut.

Wir haben ein UserControl erstellt welches wir gerne immer wieder verwenden möchten um Kundendaten anzuzeigen.
Da wir es in mehreren Projekten verwenden möchte erstellen wir und ein neues Projekt vom Typ "WPF Benutzersteuerelementbibliothek" am besten innerhalb dieser Projektmappe. Hier erstellen wir uns nun ein UserControlmit dem Namen "uclCustomerData".

Hier die Projektmappe wie diese nun aussieht:



Und hier unser UserControl:



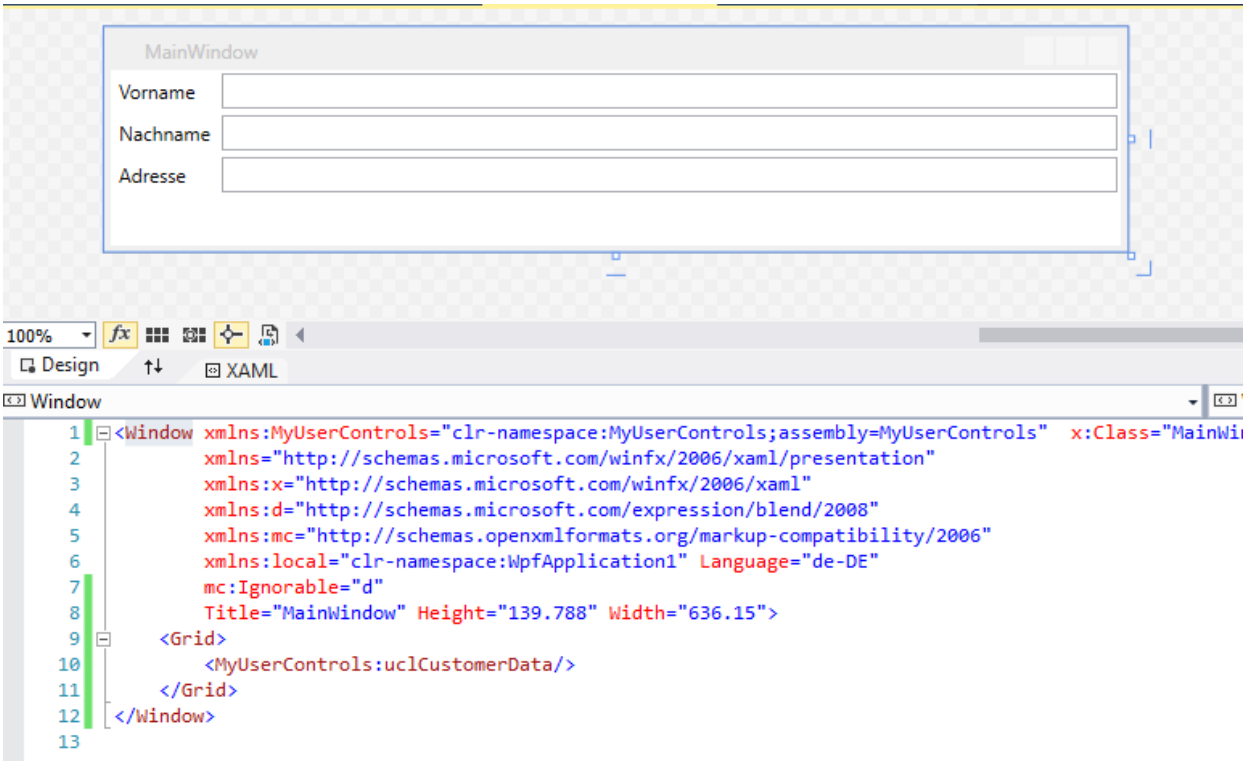

```
<UserControl x:Class="uclCustomerData"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:MyUserControls"
             mc:Ignorable="d"
             d:DesignHeight="79.491" d:DesignWidth="300">
    <Grid>
        <Grid.Resources>
            <Style TargetType="{x:Type TextBox}" BasedOn="{StaticResource {x:Type TextBox}}">
                <Setter Property="Margin" Value="2"/>
            </Style>
        </Grid.Resources>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <Label Target="{Binding ElementName=txtFirstName}" Content="_Vorname"/>
        <Label Target="{Binding ElementName=txtLastName}" Content="_Nachname" Grid.Row="1"/>
        <Label Target="{Binding ElementName=txtAddress}" Content="_Adresse" Grid.Row="2"/>
        <TextBox x:Name="txtFirstName" Grid.Column="1"/>
        <TextBox x:Name="txtLastName" Grid.Column="1" Grid.Row="1"/>
        <TextBox x:Name="txtAddress" Grid.Column="1" Grid.Row="2"/>
    </Grid>
</UserControl>
```

Einigen wird nun sicher auffallen das hier kein Binding auf das Text-Property der TextBoxen erfolgt ist. Die Eingegebenen Daten können also nie irgendwohin übertragen werden. Da es hier in diesem Kapitel allerdings rein um Resourcen geht wollte ich darauf jetzt verzichten.

Wir sehen in den Grid-Resourcen einen Style welcher auf einer Resource basiert. Und zwar erbt dieser Style von dem für eine TextBox definiertem Style.
Wir nehmen also, falls eine Ebene vor unserem UserControl einen Style für eine TextBox definiert hat, diesen her und ändern Ihn ab. In unserem Fall überschreiben wir den Setter für die Eigenschaft "Margin".

Hier greifen wir also bereits auf Resourcen zu. Um das UserControl nun in unserem Hauptprojekt verwenden zu können benötigen wir vom Hauptprojekt einen Verweis auf das Projekt mit unserem UserControl. Diesen fügen wir hinzu.

Nun können wir das UserControl in unserem "MainWindow" der Hauptapplikation einfügen:

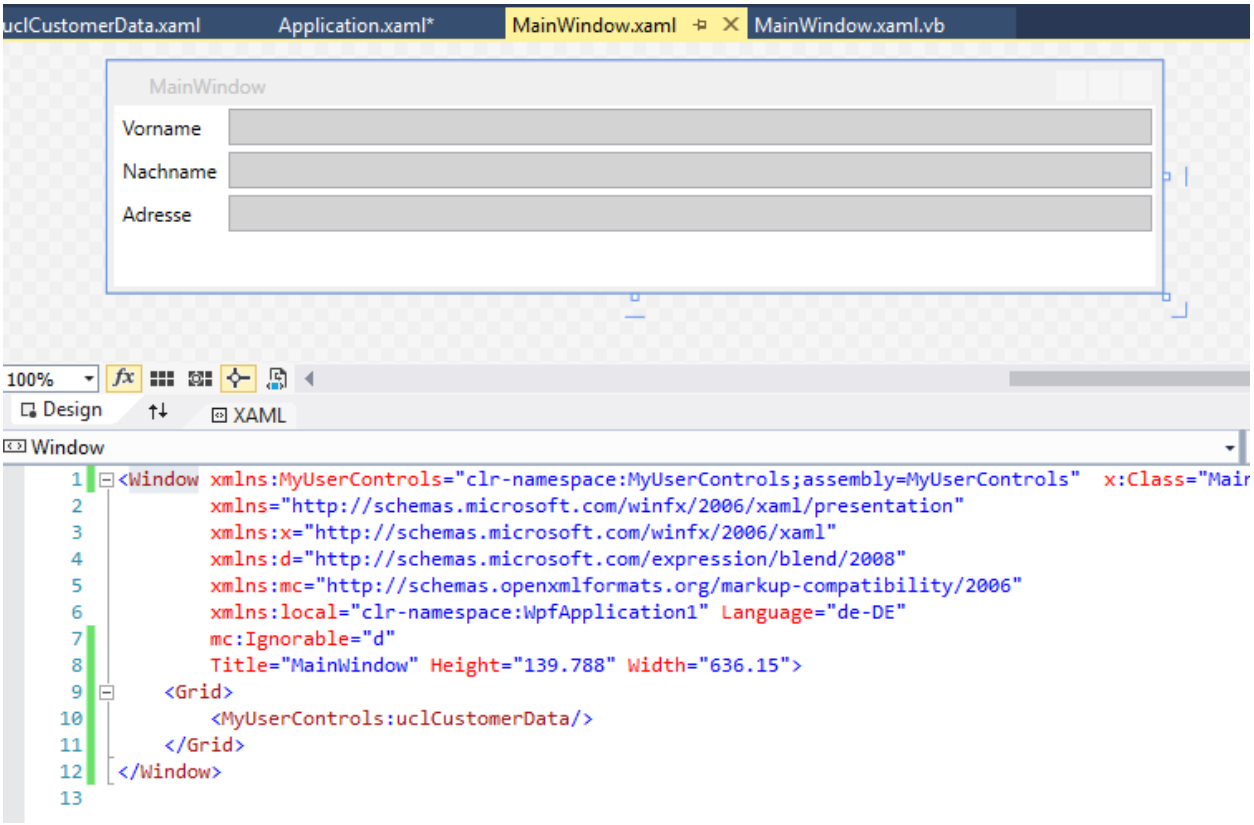



```
1  <Window xmlns:MyUserControls="clr-namespace:MyUserControls;assembly=MyUserControls"  x:Class="MainWi
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
5          xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
6          xmlns:local="clr-namespace:WpfApplication1" Language="de-DE"
7          mc:Ignorable="d"
8          Title="MainWindow" Height="139.788" Width="636.15">
9      <Grid>
10         <MyUserControls:uclCustomerData/>
11     </Grid>
12 </Window>
13
```

Wir möchten allerdings, dass in unserer Anwendung alle TextBoxen mit einem grauem Hintergrund versehen werden.
Also definieren wir in unseren Application.xaml-Resourcen das dies der Fall sein soll.

```
ustomerData.xaml   Application.xaml*   MainWindow.xaml   MainWin
Application.Resources
 1  <Application x:Class="Application"
 2       xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presen
 3       xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
 4       xmlns:local="clr-namespace:WpfApplication1"
 5       StartupUri="MainWindow.xaml">
 6       <Application.Resources>
 7
 8           <Style TargetType="{x:Type TextBox}">
 9               <Setter Property="Background" Value="LightGray"/>
10           </Style>
11
12       </Application.Resources>
13  </Application>
14
```

Sehen wir uns nun unser MainWindow nochmals an sehen wir das diese Änderung nicht nur auf eine TextBox auswirkt welche direkt in unserer Anwendung definiert worden wäre sondern auch auf eine TextBox welche sich in einem UserControl befindet welches in einer völlig anderen Bibliothek definiert wurde.



Was wenn wir aber ein UserControl erstellen wo wir dieses Verhalten NICHT haben möchten?

Dann sagen wir der WPF einfach nicht das es den Style von den TextBox-Resourcen erben soll und nehmen das BasedOn raus:

```
<UserControl x:Class="uclCustomerData"
             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:MyUserControls"
             mc:Ignorable="d"
             d:DesignHeight="79.491" d:DesignWidth="300">
    <Grid>
        <Grid.Resources>
            <!--<Style TargetType="{x:Type TextBox}" BasedOn="{StaticResource {x:Type TextBox}}">
                <Setter Property="Margin" Value="2"/>
            </Style>-->
            <Style TargetType="{x:Type TextBox}">
                <Setter Property="Margin" Value="2"/>
            </Style>
        </Grid.Resources>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
```

Nun erbt es den Style nicht mehr!

Mir ist klar dass ich jetzt wieder ein wenig in den Styles Bereich gekommen bin, nur ist es wichtig zu verstehen das im Grunde auch ein Style eine Resource ist und wie die Vererbung funktioniert.

Kommen wir nun wieder zurück zum wesentlichen.

Unterschied zwischen StaticResource und DynamicResource

In XAML gibt es zwei Arten eine Resource zuzuweisen. Als StaticResource oder als DynamicResource.

Die Unterschiede zeige ich hier mal anhand einer Tabelle:

Statische Resource	Dynamische Resource
Syntax: <object property="{StaticResource key}" .../> <object> <object.property> <StaticResource ResourceKey="key" .../> </object.property> </object>	**Syntax:** <object property="{DynamicResource key}" .../> <object> <object.property> <DynamicResource ResourceKey="key" .../> </object.property> </object>
Der Wert wird beim kompilieren zugewiesen und zur Laufzeit fest einkompiliert.	Der Wert wird zur Laufzeit in dem Moment geladen wenn er benötigt wird.
Werteänderungen zur Laufzeit werden ignoriert.	Der aktuelle Wert wird beim laden des Objekts zugewiesen. Hat sich der Wert vorher geändert wird der geänderte Wert zugewiesen.
Für die gesamte Lebensdauer der Applikation wird immer der selbe Wert verwendet	
Optimal wenn sich ein Wert nicht ändert oder nicht geändert werden soll.	Optimal wenn sich der Wert über Code Behind zur Laufzeit ändern lassen soll.

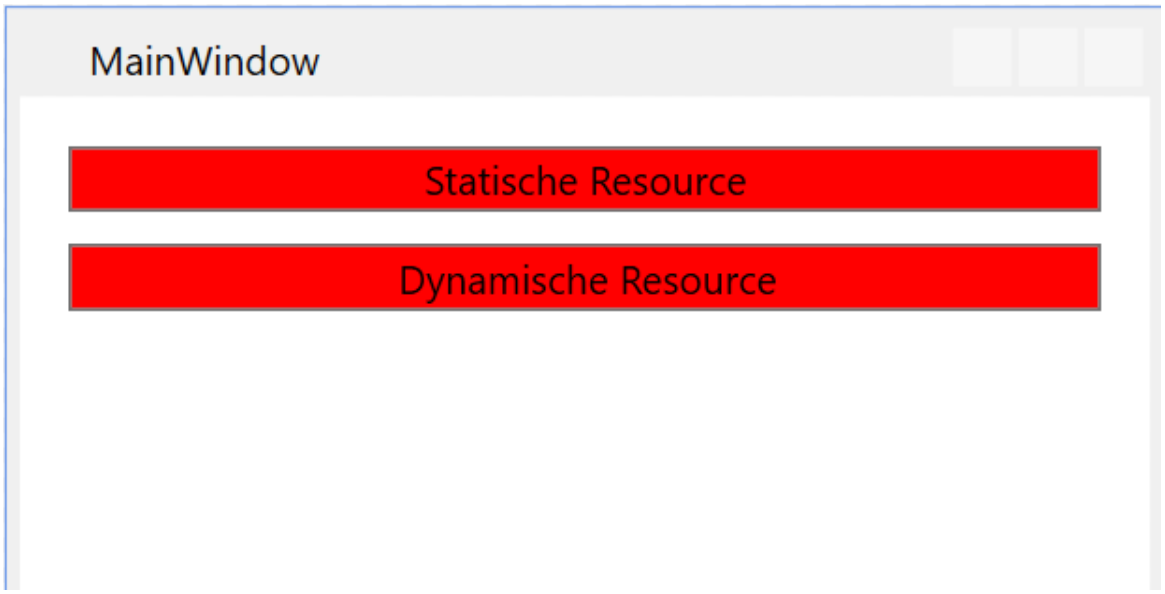
Sehen wir uns das ganze mal anhand eines simplen Beispiels an.
Wir definieren in einer Window-Resource eine Color und einen SolidColorBrush.

```
<Window.Resources>
        <Color x:Key="myRedBackgroundBrush">Red</Color>
        <SolidColorBrush x:Key="myBackgroundBrush" Color="{StaticResource
myRedBackgroundBrush}"/>
    </Window.Resources>
```

Damit man nun den großen Unterschied zwischen einer Statischen und einer Dynamischen Resource sieht erstelle ich zwei Buttons in einem StackPanel.

```
<StackPanel Name="mystackPanel" Margin="10">

    <Button Content="Statische Resource"
            Background="{StaticResource myBackgroundBrush}"
            Margin="5"/>
    <Button Content="Dynamische Resource"
            Background="{DynamicResource myBackgroundBrush}"
            Margin="5"/>

</StackPanel>
```

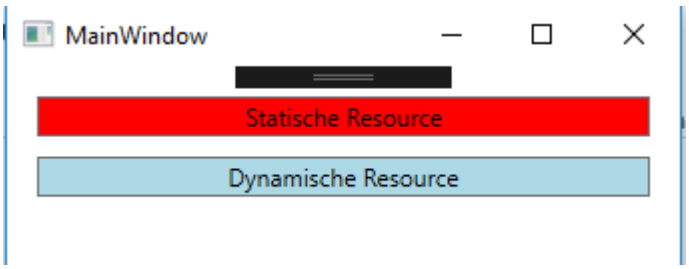
Demnach sehen beide Buttons - bis auf den Content – gleich aus.



Wo der Unterschied liegt erkennen wir wenn wir nun mit folgendem CodeBehind die Color des SolidColorBrush mit dem Namen *„myBackgroundBrush“* auf LightBlue ändern.

```
Class MainWindow
    Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles Me.Loaded
        mystackPanel.Resources("myBackgroundBrush") = Brushes.LightBlue
    End Sub

End Class
```

Folgendes Ergebnis erscheint uns zur Laufzeit:



Ich habe mich absichtlich dafür entschieden die Resource über die Resourcen des StackPanels zu ändern und zu zeigen dass man auf die Resourcen auch über ein untergeordnetes Objekt Zugriff bekommt. Die Resource „myBackgroudBrush“ wurde in den Window-Resourcen definiert, trotzdem greife ich über das StackPanel darauf zu und ändere diese.

Diesmal gibt es zum Abschluss eines Kapitels mal kein Video. Ich denke hier konnte ich alles Nötige ohne Video vermitteln. Probiert es aus und spielt mit Resourcen herum. Erstellt auch gerne mal ein UserControl in einem anderen Projekt und seht ob es sich evtl. verändert wenn Ihr es in ein anderes Projekt mit Projektweiten Resourcen einbindet.

2.1.4

Binding und das Bindingsystem

Ein wenig Theorie muss sein

Die WPF-Datenbindung bietet für Anwendungen eine einfache und konsistente Möglichkeit, Daten darzustellen und mit ihnen zu interagieren. Die Datenbindung ermöglicht im Grunde ein einfaches interagieren mit Daten. Elemente können aus einer Vielzahl von Datenquellen in Form von Common Language Runtime-Objekten (CLR-Objekten) und XML-Daten gebunden werden.

Die Datenbindung in der WPF bietet gegenüber herkömmlichen viele Vorteile. Beispielsweise die klare Trennung zwischen Geschäftslogik und UI sowie die flexible Darstellung.

2.1.4.1

Was ist DataBinding – Das Konzept dahinter

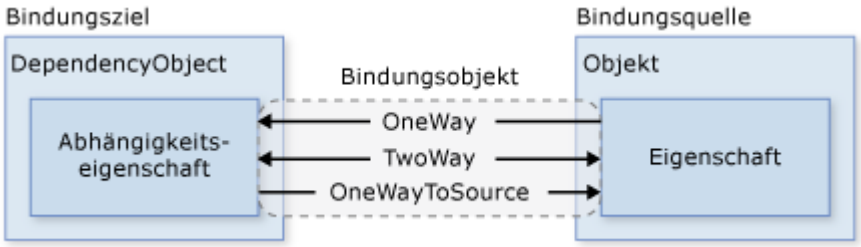
Durch das Binden wird eine Verbindung zwischen der UI und der Geschäftslogik hergestellt. Durch ein Benachrichtigungssystem werden Daten zwischen einer Logik (z.b. einer Klasse) und dem UI (z.b. eine TextBox) hin und her synchronisiert. Beispielsweise ist das Text-Property einer TextBox auf ein String-Property einer Klasse gebunden. Tippt der Benutzer einen Text in die TextBox wird der eingegebene Text automatisch in das Property der Klasse gereicht und umgekehrt. Wie und wann synchronisiert werden soll unterstützt das Bindingsystem durch mehrere Eigenschaften welche gesetzt werden können.

Hier ein Modell für ein Binding (Quelle: Microsoft Docs):

Quelle: https://docs.microsoft.com/de-de/dotnet/framework/wpf/data/data-binding-overview

In der Abbildung dargestellt, die Brücke zwischen Bindungsziel und Bindungsquelle.
Ein Bindungsziel wäre Beispielsweise eine TextBox (DependencyObject) und die Abhängigkeitseigenschaft wäre somit z.b. das Text-Property der TextBox.

Die Richtung des Datenflusses:



Quelle: https://docs.microsoft.com/de-de/dotnet/framework/wpf/data/data-binding-overview

Wie bereits erwähnt kann die Richtung in welche die Daten synchronisiert werden sollen beeinflusst werden. Wie in der Abbildung durch die Pfeile Dargestellt gibt es drei Arten der Synchronisierung. OneWay, TwoWay und OneWayToSource. Tatsächlich ist die Abbildung allerdings ein wenig unvollständig da man in der Intellisense bei der Angabe des *Modes* noch zwei weitere Vorschläge angezeigt bekommt. OneTime und Default.

- **OneWay**

  Ein Binding mit dem Mode OneWay für dazu das Daten von der gebundenen Klasse an das Property des Controls übergeben werden sobald dieses geändert wird, jedoch nicht zurück. Ändert der User den Wert (z.b. den Text in einer TextBox) wird die Änderung des Textes nicht an die gebundene Klasse übergeben. Der Wert des Properties innerhalb der Klasse bleibt also unverändert.

- **TwoWay**

  Dies führt dazu das in beide Richtungen synchronisiert wird. Wird in der gebundenen Klasse der Wert des Properties geändert wird die Änderung an die UI, also an das Property des Controls übertragen. Ändert der Benutzer den Wert wie z.b. den Text in einer TextBox wird der neue Text in die gebundene Klasse übertragen.

- **OneWayToSource**

  Kann als Umkehrung des OneWay Bindings gesehen werden. Nun wird nur noch eine Änderung wie der Text in einer TextBox an die gebundene Klasse übergeben, Änderungen innerhalb der Klasse (wenn z.b. eine Prozedur innerhalb der Klasse das Property ändert) werden allerdings nicht an das Text-Property der TextBox übertragen.

- **OneTime**

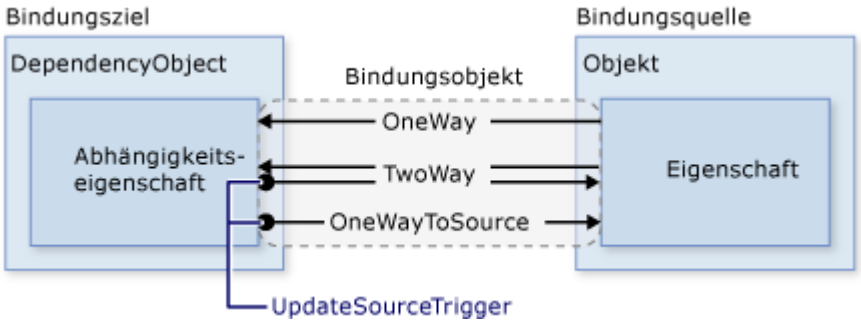
  Hier wird ein OneWay Binding erstellt welches allerdings nur beim initialisieren des UI synchronisiert wird. Änderungen werden später nicht mehr aktualisiert. Im Grunde eine Momentaufnahme.

- **Default**

  Das ist der per Standard eingestellte Wert. DependencyObjekte verfügen in der Regel über ein DependencyProperty auf welches gebunden wird. Diese DependencyProperties besitzen Metadaten in welchen Dinge wie der Standartwert und der Default-Binding-Mode festgelegt sind. Ich habe bisher noch keine gute Dokumentation gefunden aus welcher hervorgeht bei welchem Property eines Controls welcher Bindingmodus der Default ist. OneWay oder TwoWay weshalb ich immer empfehle den Modus mit anzugeben sollte man sich unsicher sein.

Woher weis die WPF nun das sich ein Wert in der Klasse geändert hat oder umgekehrt?

Hierfür gibt es im Bindingsystem die Eigenschaft UpdatesourceTrigger. Der Trigger bestimmt wann eine Änderungsbenachrichtigung vom UI zur gebundenen Klasse erfolgt. Wie dieser Trigger von der Klasse aus erfolgt erfahren wir später.



Quelle: https://docs.microsoft.com/de-de/dotnet/framework/wpf/data/data-binding-overview

Es gibt 3 Werte für die Eigenschaft UpdateSourceTrigger.
LostFocus, *PropertyChanged* und *Explizit*.

Für die meisten Controls und die meisten Eigenschaften dieser ist der Standardwert dieser Eigenschaft *PropertyChanged*, ein gutes Beispiel wo dies nicht der Fall ist, ist das *Text* Property der TextBox. Hier ist der Standardwert *LostFocus*. Dies hat einfach Performancegründe. Wo es beim *Checked*-Property einer CheckBox völlig in Ordnung ist bei jeder Änderung (aktiviert oder nicht aktiviert) eine Synchronisation anzustoßen ist dies beim *Text*-Property der TextBox nicht von Vorteil ja bei jedem Tastendruck immer synchronisiert werden würde was in den meisten Fällen unnötig ist. *LostFocus* bewirkt das Beispielsweise erst beim Verlassen der TextBox der Wert in die Klasse geschrieben werden würde.

In die andere Richtung, also von der Klasse zum UI wird dies im Code gesteuert, hierfür muss das Interface *INotifyPropertyChanged* implementiert werden. Im Setter des jeweiligen Properties muss das Event *PropertieChanged* geworfen werden wobei diesem *PropertyChangedEventArgs* übergeben werden - welche einen String mit dem Namen des Propertys erwarten - übergeben werden. Aber hierzu kommen wir später noch genauer.

Aber wie wird nun gebunden?

Da die WPF was das Binding betrifft überaus flexibel ist kann auf verschiedene Arten gebunden werden sowie auf verschiedene Datentypen, es gibt die Möglichkeit der Datenkonvertierung und Strandardkonvertierungen als auch die Datenvalidierung. All dies werde ich in den nächsten Videos erläutern da dies wie ich finde wieder mit praxisnahen Beispielen besser vermittelt werden kann.

Der Forumsbeitrag zu diesem Kapitel befindet sich hier. Ein Video oder eine Projektmappe gibt es zu diesem Kapitel keine.

2.1.4.2

Binding anhand einfacher Beispiele

Wie schon erwähnt erkläre ich euch Binding anhand eines Videos da man hier viel besser rüberbringen kann um was es geht und mit Praxisbeispielen arbeiten kann.

Wir werden an Eigenschaften anderer Steuerelemente Binden und dann an eine eigene Klasse. Zuerst Binden wir Steuerelemente in einem Window an die eigene Code Behind bevor wir an eine selbst geschriebene Klasse binden, und auch hierfür gibt es wieder mehrere Möglichkeiten. Zum Schluss zeige ich euch auch noch wie man direkt an My.Settings binden kann.

Das Video findet Ihr hier: https://youtu.be/SmYgc6wg-Aw

2.1.4.3

Designtime-Support und Intellisense für Binding

Binding ist in der WPF das wohl wichtigste Feature und sollte wirklich aus dem FF beherrscht werden da man sich sonst ständig wegen irgendeinem kleinen Problemchen ärgern muss.

Beim Binding ist zudem noch darauf zu achten das hier auf die Groß-Kleinschreibung Rücksicht genommen wird, Binding ist also Case Sensitive.

Heute sind wir es mittlerweile gewohnt von einer Entwicklungsumgebung wie VisualStudio nicht nur Feedback darüber zu bekommen das wir uns gerade vertippt haben oder wir eine Funktion falsch verwenden sondern das wir so gut wie überall auch Intellisense nutzen dürfen. Wir bekommen immer alles Mögliche vorgeschlagen und drücken anschließend nur noch Tab und ersparen uns zum einen Tipparbeit und zum anderen machen wir so viel weniger Fehler.

Tja, aber wie sieht das nun mit dem Binding aus. Der Designer zeigt uns unser Fenster oder UserControl an und wir können über Binding auf ein Property binden aber wenn wir wissen möchten ob das Binding funktioniert, wir uns nicht vertippt haben oder ob wir überhaupt in der richtigen Ebene unterwegs sind müssen wir unsere App kompilieren und starten.
Das dies sehr mühsam werden kann muss ich wohl niemanden sagen.

Stellt euch vor ihr habt einen View in den tiefen eures Programms versteckt wie z.b. die Einstellungen, habt vielleicht noch einen Login in eurem Programm usw.
Ihr müsst also eurer Programm jedes Mal starten, einloggen und in die Einstellungen navigieren nur damit Ihr wisst ob das was Ihr gemacht habt funktioniert. Das ist weder angenehm noch

Zeitgerecht. Mal ganz zu schweigen das es Zeit raubt welche Ihr verwenden könnt um aktiv zu entwickeln.

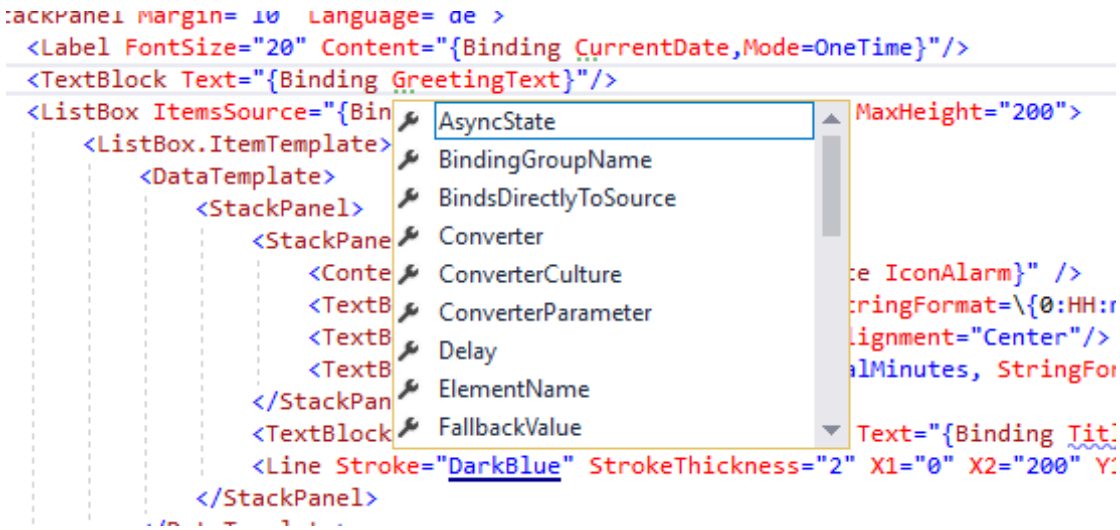
Der DesignTime-Mode macht es euch möglich hier um einiges produktiver zu sein. Nicht nur das Ihr plötzlich Intellisense zu Verfügung habt, ihr könnt auch Beispieldaten laden lassen um Anhand „echter“ Daten das verhalten eurer Steuerelemente oder des ganzen Views zu sehen und dementsprechend abzuändern.

Nehmen wir wieder unsere DayInfo Beispielklasse aus dem letzten Kapitel her und erstellen einen View für diese wie wir ihn im letzten Kapitel auch bereits gesehen hatten.

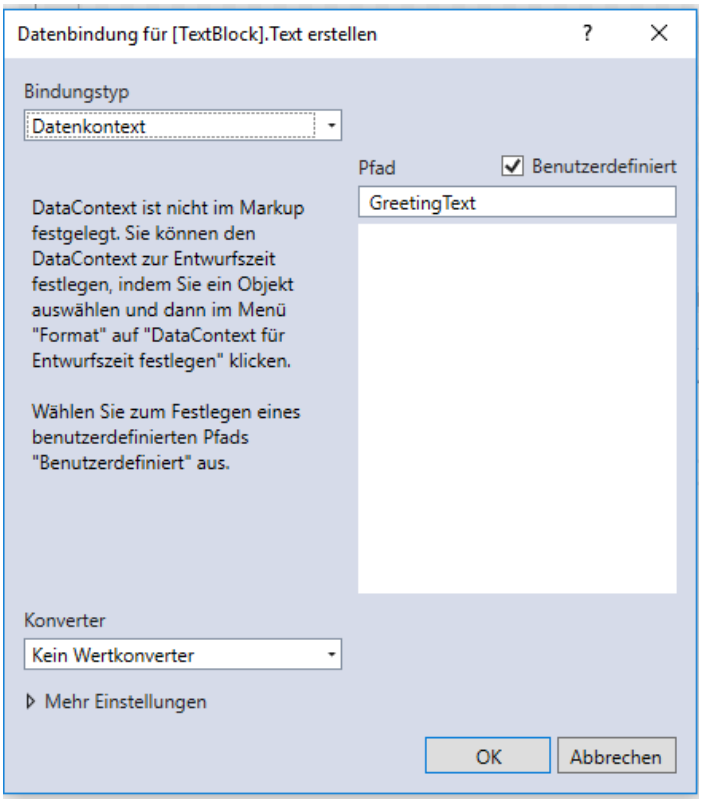
Nehmen wir nun Beispielsweise mal die TextBox welche wir auf den Begrüßungstext GreetingsText binden möchten:

```
<TextBlock Text="{Binding GreetingText}"/>
```

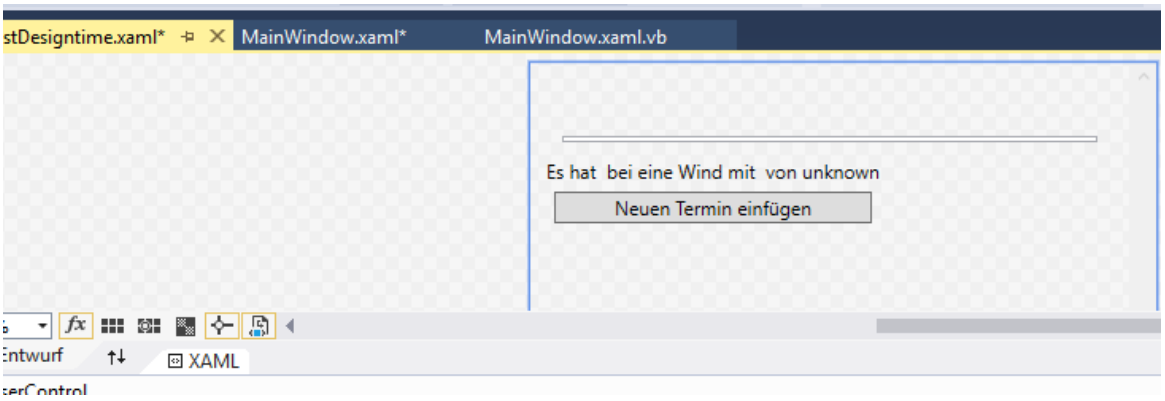
Auch über den sogenannten „Binding-Picker“ über das Eigenschaftenfenster des jeweiligen Controls haben wir nur die Möglichkeit manuell ein Binding zu setzen aber der Designer weis nicht welche Properties die Klasse hat. Wie auch, er weis nicht nicht mal von welcher Klasse wir überhaupt reden.



Wir haben keine Intellisense und zur Verfügung



Der Designer kann auch die Controls nicht mit Beispieldaten füllen:



Der Designer weis nichts von unserer Klasse

Um dem Abhilfe zu schaffen kann man dem Designer bekanntgeben an welchen Typ wir zur Designtime binden möchten damit dieser weis was wir möchten und welche Eigenschaften unsere Klasse im Endeffekt besitzt.

Beim Anlegen eines Windows oder eines UserControls befinden sich per Default immer 5 importierte Namespaces in Form von XAML in unserem Objekt.

```
<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:_2_1_4_3_DesignTimeSupport"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

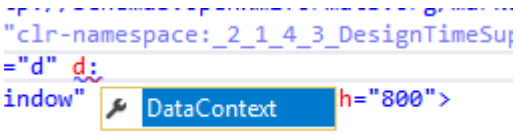
Hier ein normales Fenster mit einem Default-Namespace und vier benannten.

Wir werden nun gleich den mit einem „d“ benannten Namespace

```
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

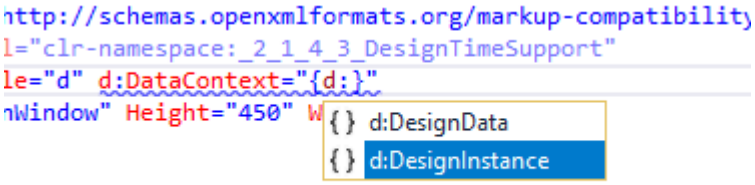
sowie noch den mit „x“ benannten Namespace verwenden und müssen uns noch einen eigenen hinzuholen da wir den Namespace in welchem unsere DayInfo-Klasse liegt auch noch benötigen um Zugriff darauf zu haben.

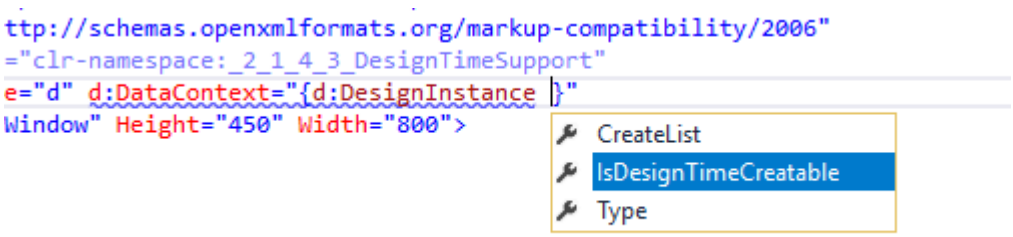
Innerhalb des „d“ Namespaces haben wir Zugriff auf ein Property mit dem Namen „DataContext“:

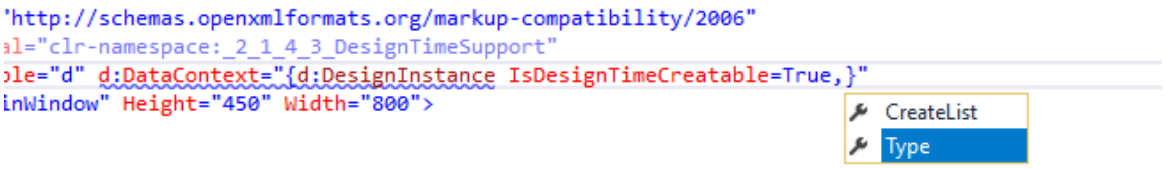


Dieser Erwartet eine DesignInstance welcher wir wiederum einen Typ übergeben müssen.
Ich versuche dies Anhand von Screenshots darzustellen, werde aber hierzu auch noch ein Video erstellen.

Djfsl djfsl

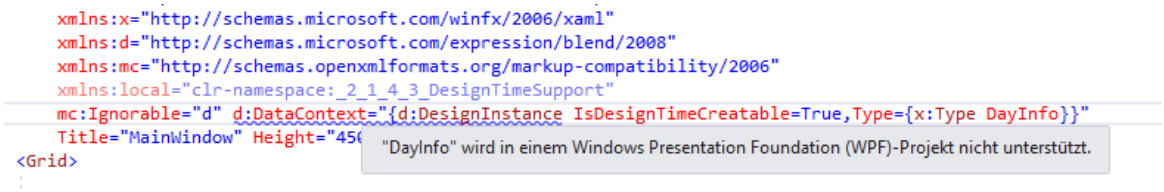






Sind wir fertig mit unserer Eingabe meckert die IDE allerdings das „DayInfo“ in einem WPF Projekt nicht unterstützt wird. Schade, WPF kann kein DayInfo. Wie jetzt?

OK, die Fehlermeldung kann etwas verwirrend sein, es liegt einfach daran das im Default-Namespace die Klasse „DayInfo“ nicht vorhanden ist und die WPF DayInfo somit nicht finden kann. Wir müssen also unseren Namesapce in welchem die DayInfo Klasse liegt in den View holen. Dabei unterstützt uns die IDE und wir können dies mit STRK + . tun.



Nun haben wir unseren Namespace importiert und die IDE meckert auch nicht mehr:

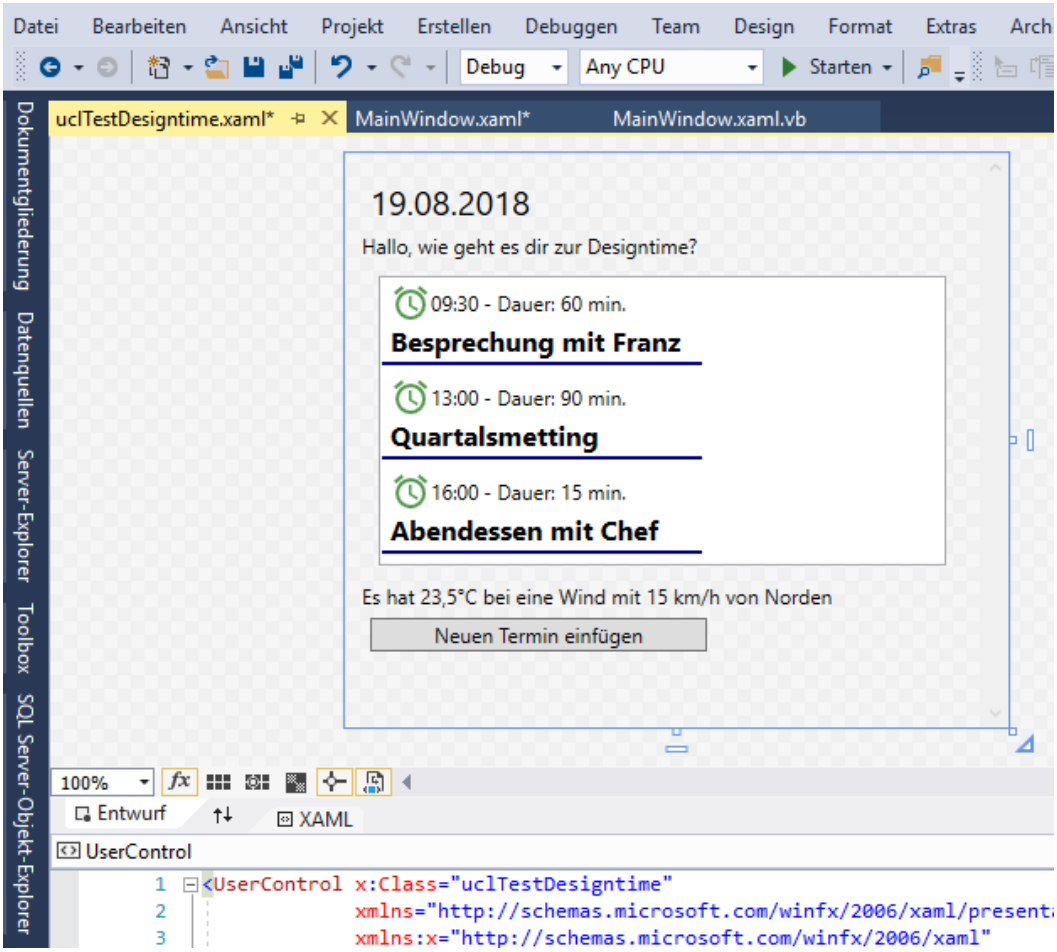
```
<Window x:Class="MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:_2_1_4_3_DesignTimeSupport"
        xmlns:testClasses="clr-namespace:_2_1_4_3_DesignTimeSupport.TestClasses"
        mc:Ignorable="d" d:DataContext="{d:DesignInstance IsDesignTimeCreatable=True,Type={x:Type testClasses:DayInfo}}"
        Title="MainWindow" Height="450" Width="800">
    <Grid>

    </Grid>
</Window>
```

Genau das machen wir nun mit unserem UserControl welches nun wie folgt (gekürzte Ansicht) aussieht:

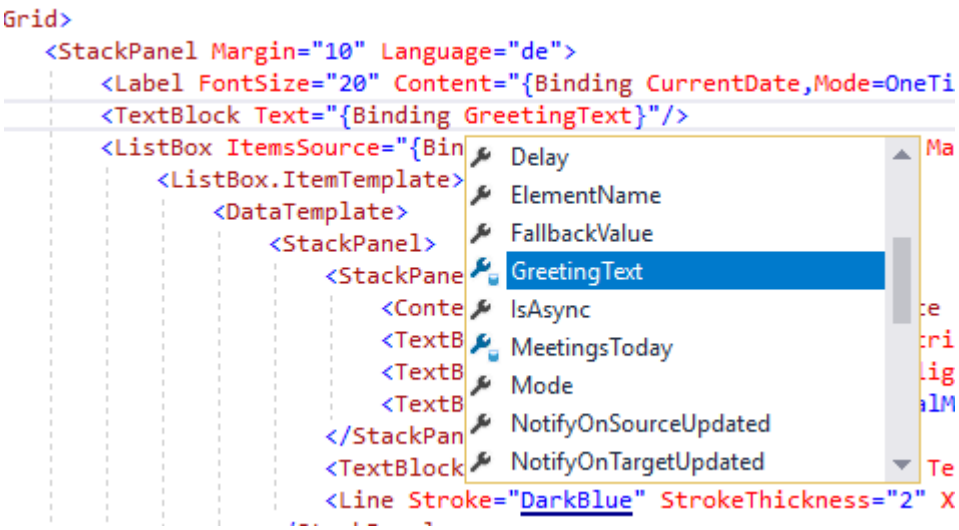
```
<UserControl x:Class="uclTestDesigntime"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             xmlns:local="clr-namespace:_2_1_4_3_DesignTimeSupport"
             xmlns:testClasses="clr-namespace:_2_1_4_3_DesignTimeSupport.TestClasses"
             mc:Ignorable="d" d:DataContext="{d:DesignInstance
IsDesignTimeCreatable=True,Type={x:Type testClasses:DayInfo}}"
             d:DesignHeight="341" d:DesignWidth="394">
    <Grid>
        <Grid.Resources>
            <Viewbox …
            </Viewbox>
        </Grid.Resources>
        <ScrollViewer>
            <Grid>
….
            </Grid>
        </ScrollViewer>
    </Grid>
</UserControl>
```

Und siehe da, nun zeigt der Designer auch gleich viel mehr an und ich sehe wie dieser View wohl

zur Laufzeit ausehen wird:



Auch die Intellisense ist nun vorhanden:



So ist ein Arbeiten mit Binding schon mal viel angenehmer.
Aber von wo kommen diese Beispieldaten? Muss ich diese immer selbst implementieren und hier immer Beispieldaten angeben oder kann ich mir die Arbeit sparen auch?

Fangen wir damit an was der Designer macht wenn wir einen Designtime DatenContext angeben.

Dadurch das wir beim instanzieren des DesignTime-Datencontexts angegeben haben das die Instanz von DayInfo zu Designtime generiert werden kann ruft der Designer den paramaeterlosen Konstruktor dieser Klasse auf.

Achtung: Wenn echte Beispieldaten verwendet werden sollen MUSS die Klasse einen parameterlosen Konstruktor aufweisen da dies sonst der Designer mit einer entsprechenden Fehlermeldung quittiert.

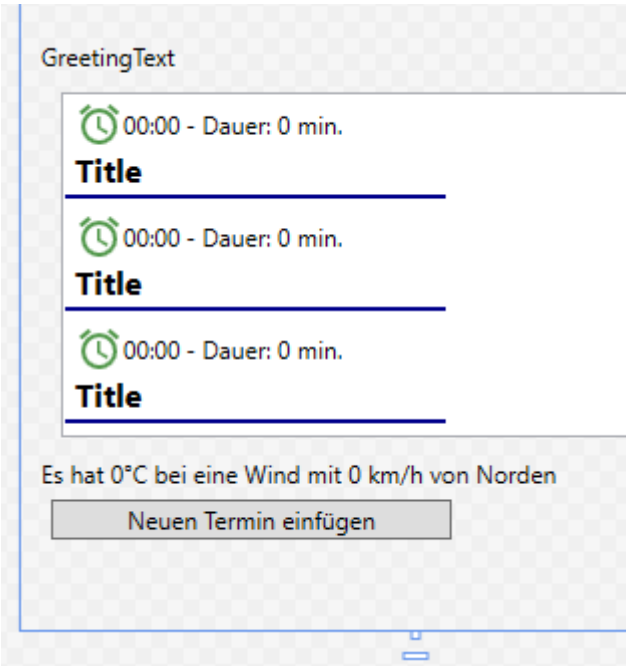
Folgenden Code habe ich im parameterlosen Konstruktor der Klasse:

```
   Public Sub New()
            If DesignerProperties.GetIsInDesignMode(New DependencyObject) Then
                GreetingText = "Hallo, wie geht es dir zur Designtime?"
                CurrentDate = DateTime.Today()

                MeetingsToday = New ObservableCollection(Of Meeting) From {
                    New Meeting("Besprechung mit Franz", New Date(CurrentDate.Year, CurrentDate.Month, CurrentDate.Day, 9, 30, 0),
TimeSpan.FromMinutes(60)),
                    New Meeting("Quartalsmetting", New Date(CurrentDate.Year, CurrentDate.Month, CurrentDate.Day, 13, 0, 0),
TimeSpan.FromMinutes(90)),
                    New Meeting("Abendessen mit Chef", New Date(CurrentDate.Year, CurrentDate.Month, CurrentDate.Day, 16, 0, 0),
TimeSpan.FromMinutes(15))
                    }

                TodayWeather = New WeatherInfo() With {.CurrentTemp = 23.5, .WindDirection = WindDirection.North, .Windspeed = 15}
            Else

                GreetingText = "Hallo, wie geht es dir an diesem schönen Tag?"
                CurrentDate = DateTime.Today()

                MeetingsToday = New ObservableCollection(Of Meeting) From {
                    New Meeting("Besprechung mit Franz", New Date(CurrentDate.Year, CurrentDate.Month, CurrentDate.Day, 9, 30, 0),
TimeSpan.FromMinutes(60)),
                    New Meeting("Quartalsmetting", New Date(CurrentDate.Year, CurrentDate.Month, CurrentDate.Day, 13, 0, 0),
TimeSpan.FromMinutes(90)),
                    New Meeting("Abendessen mit Chef", New Date(CurrentDate.Year, CurrentDate.Month, CurrentDate.Day, 16, 0, 0),
TimeSpan.FromMinutes(15))
                    }

                TodayWeather = New WeatherInfo() With {.CurrentTemp = 23.5, .WindDirection = WindDirection.North, .Windspeed = 15}
            End If

        End Sub
```

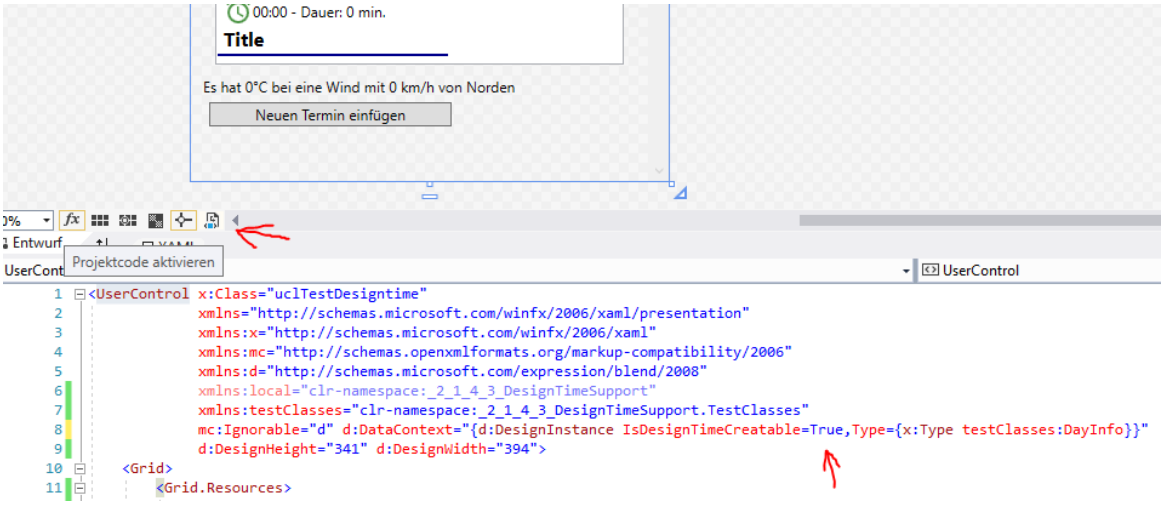
Wir unterscheiden also zwischen Designtime und Runtime. OK, aber muss ich jetzt für jede Klasse extra Code schreiben wo Beispieldaten generiert werden nur damit ich Intellisense habe?

Nein, nur wenn ich mit Beispieldaten testen möchte muss ich dies tun, gebe ich Beispielsweise beim Instanzieren des Designtime-DataContext an das die Klasse nicht zur Designzeit erstellbar ist wird der Code aus dem Konstruktor ignoriert und es wird von der IDE selbst etwas generiert.

Nämlich wird der Name der Eigenschaft als Wert geschrieben bzw. werden bei Auflistungen drei Beispieleinträge – jeweils mit dem Standartwert der Eigenschaften geschrieben.



Das selbe passiert im übrigen auch wenn der Wert beim Designer die Option „Projektcode deaktivieren“ aktiv ist wie folgende Abbildung zeigt:



So kann man selbst entscheiden ob man die Beispieldaten generiert oder ob man mit der automatischen Generierung zufrieden ist.

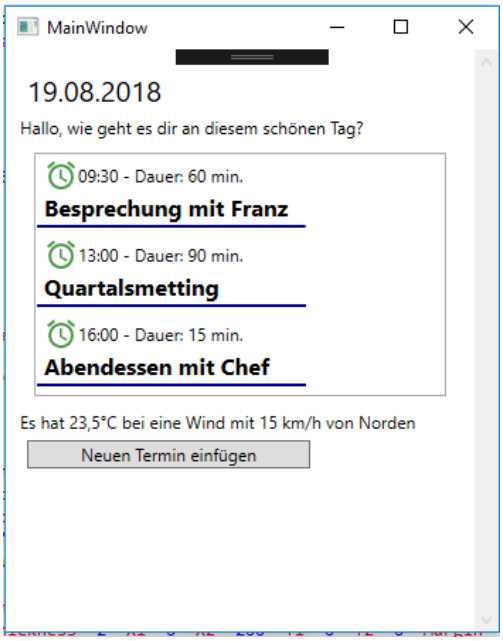
Ich habe das UserControl nun in ein Window gepackt und starte das Programm nun.

Doch was ist los, zur Designzeit haben wir die Daten drinnen, es sollte doch alles funktionieren. Warum sehen wir nun keine Daten? Das liegt daran das wir den DesignTime-Datenkontext angegeben haben und der Designer nun weis was Sache ist, aber wir haben ja bislang kein

Databinding zur Laufzeit angegeben also erstelle ich folgenden Code in der CodeBehind des MainWindow:

```
Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles
Me.Loaded
        Me.DataContext = New DayInfo
    End Sub
```

Nun sieht unser UserControl zur Laufzeit genauso aus wie zur Designzeit.



In folgendem Video gehe ich Schritt für Schritt mit euch nochmals durch diese Vorgänge und erkläre dabei auf was es ankommt und wie Ihr euch hier viele Nerven sparen könnt.

Fazit: In den meissten Büchern, wie auch in dem Buch welches ich hier bei mir liegen habe wird auf über 1200 Seiten reiner WPF kein Wort von einem DesignTime-Datenkontext erwähnt. Ich finde das extrem schade, gerade mit dieser Option spare ich mir nicht nur extrem viel Zeit und die Zusammenarbeit mit z.b. einem Designer wird damit viel einfacher wenn dieser Beispieldaten serviert bekommt. Instellisense rundet das ganze nochmals ab.

Schade das hier von vielen Seiten nicht darauf eingegangen wird.

Viel spass mit dem Video dazu: https://youtu.be/PmTCEpHSGGY

Das Visual Studio Projekt und den Thread findet Ihr wie immer im Vb-Paradise Forum.

2.1.4.4

Binding über Converter

Es wird vorkommen das man Daten an die View Binden möchte welche so nicht zu Binden sind. Beispielsweise haben Sie eine Eigenschaft in Ihrer zugrundeliegenden Klasse vom Typ Boolean vorliegen, möchten allerdings das im View abhängig von deren Wert ein Element ein oder ausgeblendet wird. Das ist so ohne weiteres nicht möglich. In der WPF ist das zuständige Property welches diesen Zustand ändert das Property Visible welches vom Typ Visibility ist.

Der Typ Visibilty ist eine Enumeration mit den Werten: *Collapsed*, *Hidden* und *Visible*.
Visible blendet das Element ein. Zwischen *Collapsed* und *Hidden* gibt es allerdings einen Unterschied welchen es unter WinForms nicht gab.

Collapsed zeigt das Element nicht an **und reserviert auch keinen Platz**.
Hidden zeigt das Element nicht an, **reserviert allerdings den Platz**.

Wir können also nicht einfach so darauf Binden. Jetzt kommen viele auf die Idee das ganze mittels Trigger (DataTrigger) zu lösen und einfach wenn per Binding ein True herauskommt das Control.Visible auf *Visible* zu setzen und bei False auf *Collapsed*.

Auch eine möglichkeit ist das ich ein Property in der zugrunde liegenden Klasse schaffe welches mit diese Umwandlung macht.

Von beidem ist abzuraten. Warum?
In beiden Fällen wiederholt man seinen Code immer und immer wieder wenn ein solches Problem auftaucht. Egal ob in XAML oder in der Klasse als VB oder C# Code.

Möchte ich meine Logik mal ändern muss ich das an jeder Stelle tun und darf keine vergessen. Weiters werden wir später, wenn wir uns mit MVVM beschäftigen sehen das ein Property mit dem Datentyp Visibility nichts im Code zu suchen hat. Um sollche Dinge soll sich alleine die View kümmern.

Gut, aber wie machen wir das jetzt? Wir schreiben einen Converter. Converter sind in der Regel sehr einfach gestrickte kleine Klassen welche so wenig Logik wie möglich beinhalten sollten.

Eine solche Konverterklasse muss das Interface IValueConverter implementieren!

Wir erstellen uns eine Klasse mit dem Namen BooleanToVisibilityConverter (per Standardkonvention sollte der Name einer Konverterklasse immer beschreiben von welchem Typ in welchem diese umwandelt, gefolgt von „Converter“) und Implementieren die Schnittstelle IValueConverter. Ich empfehle wie immer einen eigenen Namespace zu erstellen. Nun sieht unsere Klasse wie folgt aus.

```
Imports System.Globalization

Namespace Converter
    Public Class BooleanToVisibilityConverter
        Implements IValueConverter

        Public Function Convert(value As Object, targetType As Type, parameter As
Object, culture As CultureInfo) As Object Implements IValueConverter.Convert

            Throw New NotImplementedException()
        End Function

        Public Function ConvertBack(value As Object, targetType As Type, parameter As
Object, culture As CultureInfo) As Object Implements IValueConverter.ConvertBack

            Throw New NotImplementedException()
        End Function
    End Class
End Namespace
```

Die Funktion Convert wird von der WPF aufgerufen wenn von der Klasse in Richtung View der Wert konvertiert werden soll.
Wird der Konverter im XAML eingebunden ruft die WPF diese Funktion auf um zu versuchen den Wert zu Konvertieren. Im Parameter *value* ist also der Wert enthalten welcher von der Klasse kommt, also der Wert des Property's auf welches gebunden wurde. In unserem Fall ist dies ein Wert vom Typ Boolean. Wichtig ist hier immer ob der Wert welcher hier hereingereicht wird nicht vielleicht *Nothing* ist.
Anschließend müssen wir den Wert von einem Boolean in einen Wert vom Typ Visibility umwandeln.

```
Public Function Convert(value As Object, targetType As Type, parameter As Object,
culture As CultureInfo) As Object Implements IValueConverter.Convert
            If value Is Nothing Then Return Visibility.Hidden
            If CBool(value) Then
                Return Visibility.Visible
            Else
                Return Visibility.Hidden
            End If
 End Function
```

Ist der Wert in *value* Nothing geben wir den Wert *Hidden* des Typs Visibility zurück.
Ist der Wert in *value* True geben wir den Wert *Visible* des Typs Visibility zurück.
Ist der Wert in *value* False geben wir den Wert *Hidden* des Typs Visibility zurück.

Soweit, war das gar nicht schwer. Was ist aber mit den anderen Parametern der Funktion?
TargetType gibt uns den Typ zurück welchen die WPF von uns erwartet. In unserem Fall wird dies ein Visibility sein.
Parameter kann in XAML definiert werden und wird uns hier hereingereicht. Parameter ist kein Dependency Property und kann somit nicht gebunden werden, dazu kommen wir aber noch.

CultureInfo enthält die Informationen über die aktuellen Sprach und lokalisierungeinstellungen welche im Projekt oder im View festgelegt wurden.

Die Funktion Convertback ist für die entgegengesetzte Richtung zuständig. In unserem Fall würden wir diese Methode nicht benötigen weil es unwarscheinlich sein wird das wir für Visibility ein *TwoWay Binding* haben werden, möglich ist allerdings auch das. Wir werden die Methode also für diese Beispiel auch ausprogrammieren.

Wird im View der Wert gesetzt. Beispielsweise über ein Dropdown welches an den Enumerator gebunden ist und wir hier umstellen, will die WPF (wenn in TwoWay gebunden wurde) den Wert wieder vom View zurück in die Klasse also in das Property der Klasse schreiben.
Die WPF bemüht also abermals den Converter weil Sie weis – was in die eine Richtung erledigt werden muss gilt auch für die andere. Hier bekommen wir in *value* nun aber einen Wert vom Typ Visibility hereingereicht und müssen diesen nun zurück in einen Wert vom Typ Boolean konvertieren.

Wie wir das genau machen obligt uns, wir werden uns für dieses Beispiel nun dafür entscheiden was wir Hidden und Collapsed zu einem *False* wandeln und Visible zu einem *True*.

```
Public Function ConvertBack(value As Object, targetType As Type, parameter As Object,
culture As CultureInfo) As Object Implements IValueConverter.ConvertBack
            If value Is Nothing Then Return False
            Select Case DirectCast(value, Visibility)
                Case Visibility.Collapsed, Visibility.Hidden
                    Return False
                Case Else
                    Return True
            End Select
End Function
```

Wie binden wir nun einen solchen Konverter im XAML ein. Wir wechseln zu unserem XAML Code und Binden den Namespace der *Converter* in unserem XAML ein und vergeben einen Key.

z.b.:

```
<UserControl.Resources>
            <Converter:BooleanToVisibilityConverter
x:Key="BooleanToVisibilityConverter"/>
    </UserControl.Resources>
```

Nun können wir beim Binden diesen als Statische Resource eingebundenen Konverter verwenden:

```
<Label Content="Test" Visibility="{Binding MyBoolPropertyInClass,
Converter={StaticResource BooleanToVisibilityConverter}}"/>
```

Oben hatte ich bereits die Converterparameter angesprochen, über setzen des Paramters können wir zusätzliche Informationen an den Converter übergeben. Welche Informationen

bleibt im Grunde uns überlassen, nur ist zu beachten das hier kein Binding möglich ist da Converterparameter kein DependecyProperty ist.

```
<Label Content="Test" Visibility="{Binding MyBoolPropertyInClass,
Converter={StaticResource BooleanToVisibilityConverter},ConverterParameter=reverse}"/>
```

Beispielsweise kann es sein das wir diesen Konverter auch umdrehen möchten.
Möchten wir also auch den Fall das wir umgekehrt ragiert möchten (True = Hidden und False = Visible) abdecken, könnten wir hier als Parameter z.b. „reverse“ mit übergeben und der Konverter soll uns das Verhalten umkehren. So müssen wir keinen zweiten Konverter schreiben, sondern können uns einfach den einen Konverter erweitern.

```
Public Function Convert(value As Object, targetType As Type, parameter As Object,
culture As CultureInfo) As Object Implements IValueConverter.Convert
            If value Is Nothing Then Return Visibility.Hidden
            If CBool(value) Then
                If parameter?.ToString().ToLower() = "reverse" Then Return
Visibility.Hidden
                Return Visibility.Visible
            Else
                If parameter?.ToString().ToLower() = "reverse" Then Return
Visibility.Visible
                Return Visibility.Hidden
            End If
        End Function

        Public Function ConvertBack(value As Object, targetType As Type, parameter As
Object, culture As CultureInfo) As Object Implements IValueConverter.ConvertBack
            If value Is Nothing Then Return False
            Select Case DirectCast(value, Visibility)
                Case Visibility.Collapsed, Visibility.Hidden
                    If parameter?.ToString().ToLower() = "reverse" Then Return True
                    Return False
                Case Else
                    If parameter?.ToString().ToLower() = "reverse" Then Return False
                    Return True
            End Select
        End Function
```

Mehr zu Konvertern und mehrere Praxisbeispiele gibt es wieder in meinem Video und für alle die das Video aufmerksam und zu Ende sehen gibt es diesmal noch eine kleinen aber sehr nüztlichen Extratip!!

Hier geht’s zum Video: https://youtu.be/4M44oazjdww

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/

2.1.4.5

Binding über DataTemplates

Wie der Titel dieses Kapitels vermuten lässt geht es um DataTemplates. Wieder!

Ja, dem aufmerksamen leser wird auffallen das es bereits in *Kapitel 2.1.1.2 - Styles,Templates und Trigger* um Templates ging und das wir hier bereits einiges gelernt haben was diese Vorlagen (Templates auf Deutsch = Vorlagen) betrifft. Da ich allerdings der Meinung bin das DataTemplates eine besondere Aufmerksamkeit geschänkt werden sollte greife ich dieses Thema nun nochmals auf um das Wissen nochmals zu festigen und hier auch ein weiteres Beispiel für den Einsatz von DataTemplates zeigen zu können. Gerade wenn es dann später mal um MVVM oder andere Pattern geht sind DataTemplates ein wichtiges Thema und können einem auch viel Arbeit ersparen. Die WPF ist hier so leistungsstark das plötzlich Dinge möglich werden welche vorher kaum vorstellbar gewesen wären.

Aber fangen wir am besten an. Da ich davon ausgehen darf das die Funktion von DataTemplates bereits bekannt ist gehe ich gleich über zu einem Praxisbeispiel.

Wir stellen uns eine Anwendung vor in welcher wir Fahrzeuge verwalten möchten.
Erstmal überlegen wir und was für Eigenschaften Fahrzeuge haben können um zu wissen wie wir welche Daten in unserer Anwendung anzeigen möchten. Was gibt es denn für Fahrzeugtypen?

Es gibt Autos, Flugzeuge, Hubschrauber, Schiffe usw. Die Frage ist nun wie wir diese unter „einen Hut bekommen“ wollen. Klar, eine Basisklasse muss her. Welche Eigenschaften haben all diese Fahrzeugtypen gemein? Einige wären *Marke*, *Modell*, *Gewicht* und vielleicht die *Höchstgeschwindigkeit*. Bestimmt hätten alle Fahrzeugtypen mehr gemeinsam, ich erstelle für dieses Beispiel mal diese 4 Eigenschaften in einer Basisklasse TransportBase.

```
Namespace Classes
    Public MustInherit Class TransportBase
        Public Property Brand As String
        Public Property Model As String
        Public Property Weight As Double
        Public Property TopSpeed As Double
    End Class
End Namespace
```

Jetzt haben wir eine Basisklasse welche als ***MustInherit*** (Abstract in C#) deklariert ist was bedeutet das diese Klasse lediglich als Basisklasse verwendet werden darf und nicht direkt erstellt werden kann.

Als nächstes geht es darum mit einem Fahrzeugtyp anzufangen. Wir nehmen uns mal das Auto her und erstellen eine Klasse Car welche von TransportBase erbt.

```
Namespace Classes
   Public Class Car
       Inherits TransportBase

   End Class
End Namespace
```

Durch die ableitung von TransportBase besitzt die Klasse Car nun automatisch die Eigenschaften von TransportBase ohne das wir etwas tun müssen.

Hauchen wir unserer Klasse etwas mehr Leben ein und verpassen wir der Klasse Car noch zwei Eigenschaften. Die Anzahl der Zilinder des Motors eines Autos und die Sonderausstattung vielleicht noch. Damit wir die Sonderausstattung nicht einfach nur als *String* haben erstellen wir uns noch eine Klasse CarEquipment und haben nun folgendes:

```
Namespace Classes
    Public Class Car
        Inherits TransportBase

        Public Property Cylinders As Integer
        Public Property Equipment As List(Of CarEquipment)
    End Class

    Public Class CarEquipment
        Public Property Caption As String
        Public Property Description As String
    End Class
End Namespace
```

Nun besitzt ein Auto noch die Eigenschaft *Cylinders* und *Equipment* wobei Equipment viele Ausstattungen mit Bezeichnung und einer Beschreibung dazu beinhalten kann.

Das selbe können wir nun auch mit anderen Fahrzeugtypen machen wie einem Flugzeug. Nur das bei einem Flugzeug andere Daten wichtig sind. Anders als bei einem Fahrzeug interessiert uns vielleicht die Ausstattung nicht sondern wieviele Passagiere befördert werden können oder die maximale Reiseflughöhe. Und statt der Anzahl der Cylinder ist die Anzahl der Turbinen vielleicht interessant. Bei einem Schiff wäre es vermutlich der maximale Tiefgang der uns interessiert.

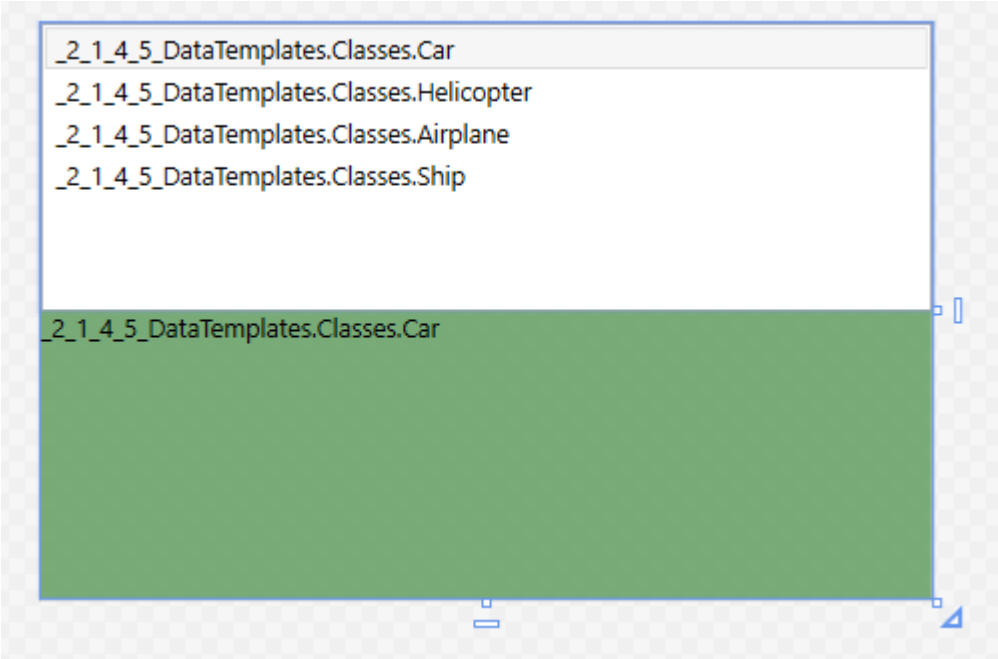
Aber jetzt wird es interessant. Wir möchten das alle Fahrzeugtypen mit Ihren Eigenschaften sowohl in einer Listbox dargestellt werden als auch wenn ein Eintrag selektiert wird, dieser unterhalb der Listbox in groß angezeigt wird. Und zwar mit all den Eigenschaften welche der jeweilige Fahrzeugtyp besitzt. Aber wie? Einige kommen dann auf die Idee (wir erinnern uns an Trigger) alle Eigenschaften welche es geben kann zu erstellen. Also für jede Eigenschaft welche ein Fahrzeug besitzen kann einen TextBlock welcher diese Eigenschaft anzeigt und dann per DataTrigger den TextBlock ein oder ausgeblendet wird. Extrem umständlich, sehr schlecht wartbar und unübersichtlich hoch 5. Da wir in der WPF DataTemplates zur Verfügung haben ist dies viel einfacher.

Erstmal erstelle ich ein UserControl mit einer ListBox und binde dieses auf eine Klasse in welcher ich eine reihe von Fahrzeugen lade. Die Klasse besitzt zwei Eigenschaften AllTransport von Typ ObservableCollection(Of TransportBase) und SelectedTransport vom Typ TransportBase.

```
<Grid>
     <Grid.RowDefinitions>
          <RowDefinition Height="1*"/>
          <RowDefinition Height="1*"/>
     </Grid.RowDefinitions>
     <ListBox ItemsSource="{Binding AllTransports}"
              HorizontalContentAlignment="Stretch"
              SelectedItem="{Binding SelectedTransport}"/>
     <Border Grid.Row="1" Background="#C6559655">
         <ContentControl Content="{Binding SelectedTransport}">

         </ContentControl>
     </Border>
 </Grid>
```

Unter der ListBox haben wir noch ein ContentControl welches auf SelectedTransport gebunden ist um das aktuell selektierte Fahrzeug in groß darstellen zu können.

Das sieht im Moment noch eher unspektakulär aus:



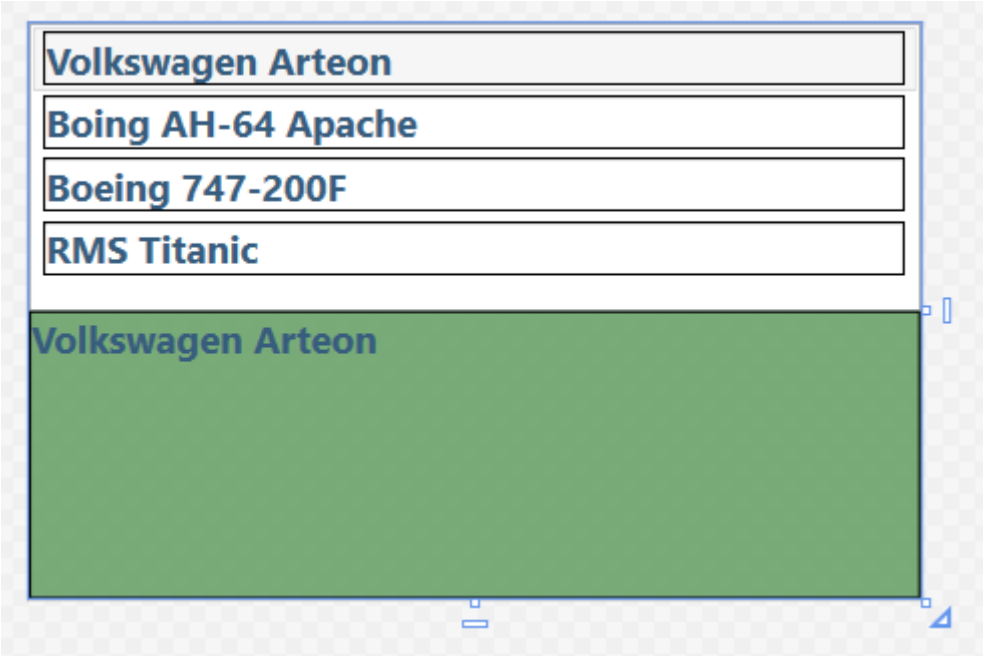
Die WPF versucht den Typ korrekt zu Rendern und ruft die Methode ToString() des jeweiligen Objekts auf was dazu führt das der Klassentyp den Objekts angezeigt wird.

Ich weis was Ihr nun denkt. Jetzt muss ich mir für jeden Typ ein DataTemplate erstellen damit es korrekt angezeigt wird. Aber was ist wenn ich das gar nicht möchte? Angenommen ich möchte in der ListBox nur Daten anzeigen welche in der Basisklasse vorhanden sind wie Marke und Modell, hier müsste ich gar nicht 4 verschiedene Templates erstellen. Wenn die WPF für den jeweiligen Typ ein Template sucht, aber nicht fündig wird, **sucht sie sogar für den Basistyp weiter**.

Beispiel: Für den Eintrag vom Typ Airplane (an Platz drei in der Liste) sucht die WPF nach einem DataTemplate für den Typ Airplane, wird sie nicht fündig sucht sie nach einem DataTemplate vom Typ TransportBase, würde sie hier auch nicht fündig werden sucht sie nach einem DataTemplate vom Typ Obbjekt und hier wird sie fündig da dieses von MS implementiert ist und wir in dem oben stehenden Screenshot sehen.

Möchten wir also nicht für jeden Fahrzeugtypen ein Template erstellen können wir auch ein Allgemeines erstellen. Damit wir dieses allerdings nicht nur innerhalb der Listbox zur verfügung haben machen wir das in den Resourcen des UserControls, so profitiert auch das ContentControl davon:

```
<UserControl x:Class="uclShowTransports"
             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:_2_1_4_5_DataTemplates"
             xmlns:classes="clr-namespace:_2_1_4_5_DataTemplates.Classes"
             mc:Ignorable="d d:DesignHeight="273.623" d:DesignWidth="424.364">
    <UserControl.Resources>
        <DataTemplate DataType="{x:Type classes:TransportBase}">
            <Border BorderBrush="Black" BorderThickness="1">
                <TextBlock FontWeight="Bold"
                           FontSize="18"
                           Foreground="#375D7E">
                    <Run Text="{Binding Brand}"/>
                    <Run Text="{Binding Model}"/>
                </TextBlock>
            </Border>
        </DataTemplate>
    </UserControl.Resources>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="1*"/>
            <RowDefinition Height="1*"/>
        </Grid.RowDefinitions>
        <ListBox ItemsSource="{Binding AllTransports}"
                 HorizontalContentAlignment="Stretch"
                 SelectedItem="{Binding SelectedTransport}"/>
        <Border Grid.Row="1" Background="#C6559655">
            <ContentControl Content="{Binding SelectedTransport}">

            </ContentControl>
        </Border>
    </Grid>
</UserControl>
```



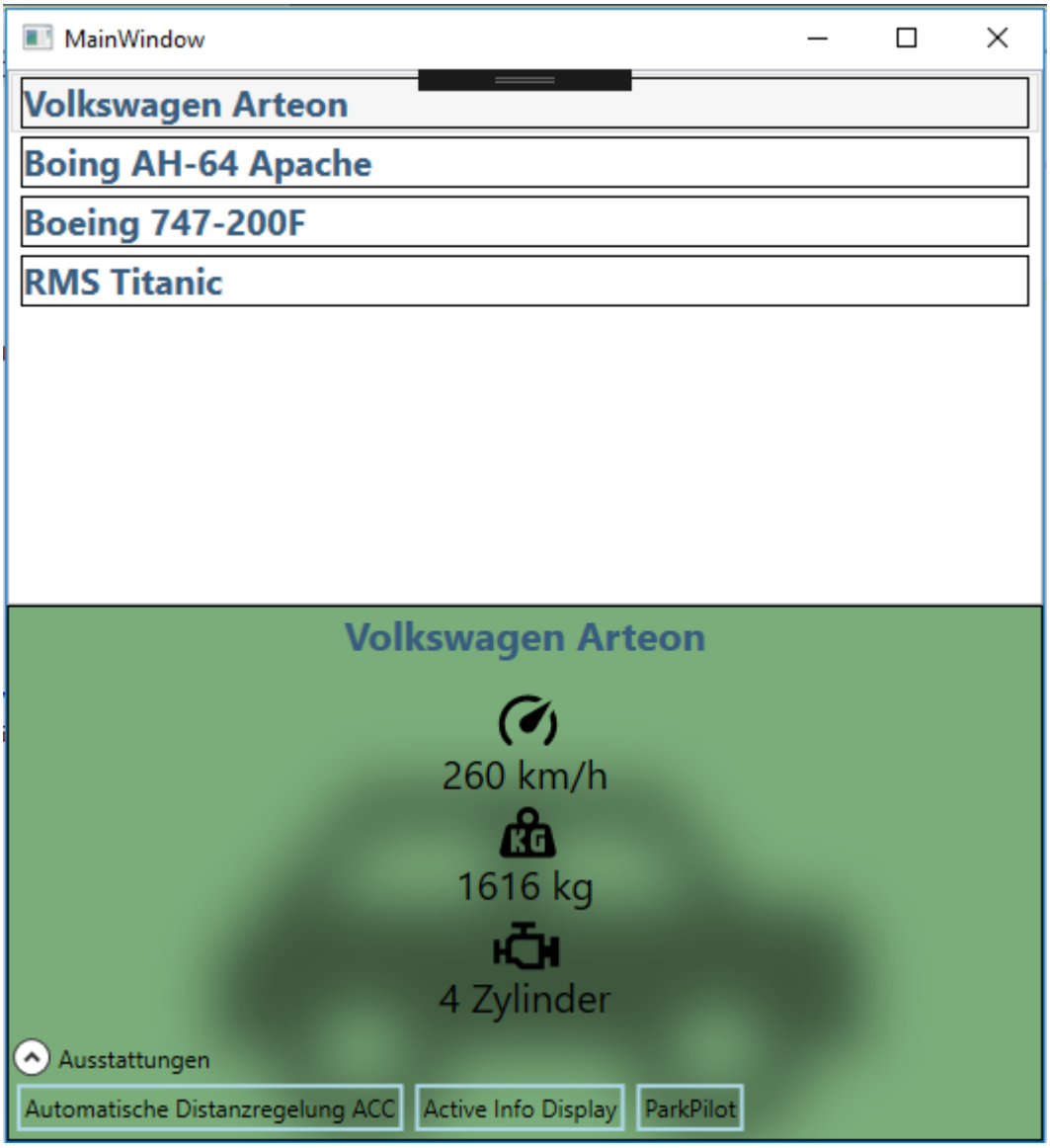
Das Ergebniss dieser kleinen Umbauarbeit innerhalb der UserControl-Resourcen bewirkt bereits einiges! Nun haben wir *Marke* und *Model* sowohl in der ListBox als auch darunter inkl. einer Umrandung in der Detailansicht stehen.

Das ist aber noch nicht zufriedenstellend. Nun möchten wir das ein Fahrzeug in der Detailansicht anders dargestellt wird als in der ListBox. Hierfür müssen wir nur Beispielsweise in den ListBox-Resourcen ein DataTemplate definieren. Dies führt dazu das dieses DataTemplate das Template in den UserControl-Resourcen überschreibt da dieses weiter unten im Element-Tree definiert ist.

Wir gehen aber noch einen Schritt weiter, wir möchten auch gesondert darauf eingehen was für ein Fahrzeugtyp gerade angezeigt wird, also in der ListBox selektiert wurde. Hierfür müssen wir aber nun für jeden Fahrzeugtyp ein Template definieren.

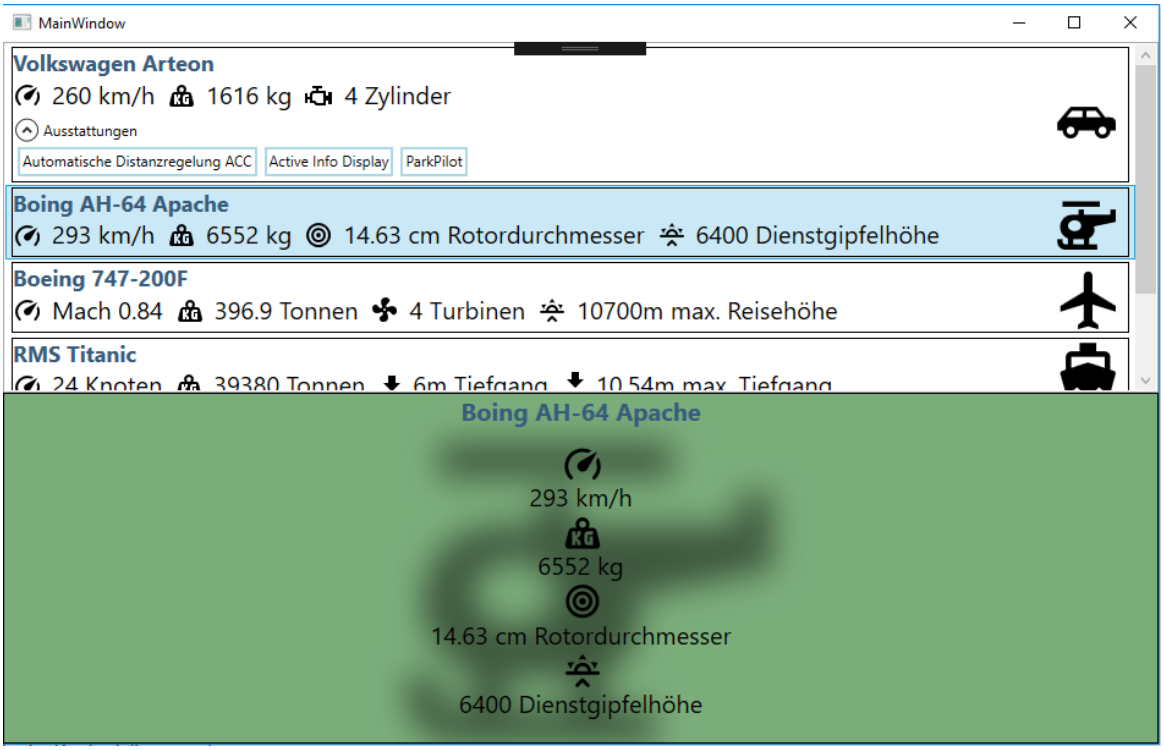
```
<ContentControl Content="{Binding SelectedTransport}">
    <ContentControl.Resources>
        <DataTemplate DataType="{x:Type classes:Car}">
            <local:uclCar/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type classes:Helicopter}">
            <local:uclHeli/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type classes:Airplane}">
            <local:uclAirplane/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type classes:Ship}">
            <local:uclShip/>
        </DataTemplate>
    </ContentControl.Resources>
</ContentControl>
```

Das Ergebniss kann sich nun bereits sehen lassen, wir haben eine Detailansicht wie folgendes Bild zeigt inkl. schönen Symbolen und einem Fahrzeugbild im Hintergrund und einem Expander welcher mit den Austattungen eines Fahrzeugs befüllt ist. Auch für die anderen Fahrzeugtypen habe ich Vorlagen erstellt. Diese sind alle in einzelne UserControls unterteilt damit hier der XAML nicht zu lange wird.



Einen Schritt weiter machen wir noch, wir entscheiden uns dafür das wir auch für die ListBox-Einträge für jeden Datentyp eine eigene Vorlage haben.

Ich habe hierfür fast den selben Code wie für die Detailansicht verwendet nur das die Spezifikationen Horizontal dargestellt werden. Unser „allgemeines“ Template kann trotzdem im XAML-Code bleiben wo es ist. So ist sichergestellt das wenn ein neuer Fahrzeugtyp hinzukommen würde dieser trotzdem angezeigt werden könnte, zwar nur mit *Marke* und *Modell* aber es würde etwas angezeigt werden. Das kann man also wie eine Art „Versicherung“ sehen.



Ihr merkt schon, ohne das Ihr jetzt komplizierten Code schreiben oder diverse Umwege gehen müsst habt ihr übersichtlich die Anzeige von Fahrzeugen in XAML definiert. Hier der komplette XAML des UserControls:

```
<UserControl x:Class="uclShowTransports"
             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:_2_1_4_5_DataTemplates"
             xmlns:classes="clr-namespace:_2_1_4_5_DataTemplates.Classes"
             mc:Ignorable="d" d:DataContext="{d:DesignInstance
IsDesignTimeCreatable=True,Type=local:AllTransports}"
             d:DesignHeight="273.623" d:DesignWidth="424.364">
    <UserControl.Resources>
        <DataTemplate DataType="{x:Type classes:TransportBase}">
            <Border BorderBrush="Black" BorderThickness="1">
                <TextBlock FontWeight="Bold" FontSize="18" Foreground="#375D7E">
                    <Run Text="{Binding Brand}"/>
                    <Run Text="{Binding Model}"/>
                </TextBlock>
            </Border>
        </DataTemplate>
        <DataTemplate DataType="{x:Type classes:Car}">
            <local:uclCarListItem/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type classes:Helicopter}">
            <local:uclHeliListItem/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type classes:Airplane}">
            <local:uclAirplaneListItem/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type classes:Ship}">
            <local:uclShipListItem/>
        </DataTemplate>
    </UserControl.Resources>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="1*"/>
            <RowDefinition Height="1*"/>
        </Grid.RowDefinitions>
        <ListBox ItemsSource="{Binding AllTransports}"
                 HorizontalContentAlignment="Stretch"
                 SelectedItem="{Binding SelectedTransport}"/>
        <Border Grid.Row="1" Background="#C6559655">
            <ContentControl Content="{Binding SelectedTransport}">
                <ContentControl.Resources>
                    <DataTemplate DataType="{x:Type classes:Car}">
                        <local:uclCar/>
                    </DataTemplate>
                    <DataTemplate DataType="{x:Type classes:Helicopter}">
                        <local:uclHeli/>
                    </DataTemplate>
                    <DataTemplate DataType="{x:Type classes:Airplane}">
                        <local:uclAirplane/>
                    </DataTemplate>
                    <DataTemplate DataType="{x:Type classes:Ship}">
                        <local:uclShip/>
                    </DataTemplate>
                </ContentControl.Resources>
            </ContentControl>
        </Border>
    </Grid>
</UserControl>
```

Natürlich erkläre ich dies alles auch wieder interaktiv in einem Video wo ihr besser sehen könnt wie einfach und übersichtlich man mit DataTemplates arbeiten kann. Spätestens jetzt sollte man die stärken der WPF erkannt haben würde ich meinen.

Die Solution und alle Kapitel und Downloads sind im Tutorialthread enthalten, viel spaß mit dem Video.

Hier geht's zum Video: https://youtu.be/nl1pxoTuOMo

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/

2.1.4.6

Binding an Collection

Wie Ihr in den vorigen Kapitel bereits gesehen habt kann man direkt an eine Collection binden, dabei ist es im Grunde völlig gleich ob diese eine List(Of T) , eine ObservableCollection(Of T) oder gar nur ein *Array* ist. Was hier aber irgendwie fehlt sind Funktionen wie *Filtern*, *Sortieren* oder auch *Gruppieren*. Aber auch die Informationen welches Item einer Auflistung nun gerade Beispielsweise in einer ListBox selektiert ist fehlt uns.

Letzteres Feature kann man noch herstellen indem man eine Eigenschaft in der Klasse definiert und die SelectedItem Eigenschaft der ListBox auf diese bindet. Es geht aber einfacher.

Die CollectionView in der WPF ist eine Instanz einer Klasse welche das Interface ICollectionView implementiert. Im folgenden sehen wir uns mal diese ICollectionView an um zu verstehen was diese so besonders macht.

Wie unter Microsoft Docs schön zu sehen ist implementiert ICollectionView wiederum zwei Schnittstellen IEnumerable und INotifyCollectionChanged durch die bereits schon einiges an Funktionalität geboten wird. Zusätzlich verfügt die Schnittstelle noch über einige Eigenschaften und Methoden welche wir uns zu nutze machen. Hier die wichtigsten welche uns im laufe dieses Kapitels interessieren werden.

Eigenschaften

CurrentItem, GroupDescriptions, Filter, Groups, SortDescriptions, SourceCollection

Methoden

MoveCurrent….() und Refresh()

Wir sehen, hier wird einiges geboten. OK, aber wie erhalten wir nun eine CollectionView?

Wir rufen die Statische Methode GetDefaultView der Klasse CollectionViewSource auf. Diese gibt uns einen DefaultView zurück. Existiert noch kein DefaultView wird mit GetDefaultView eine erzeugt. Welcher Typ eines ICollectionView von GetDefaultView genau zurückgegeben wird hängt davon ab welcher CollectionTyp der Methode übergeben wird. Für eine IList Collection ist es z.b. eine ListCollectionView.

Aber ich komme wieder zu sehr in die Theorie, wir möchten dies anhand von Beispielen lernen, das macht mehr spass und prägt sich meiner Meinung nach besser ein.

Es gibt zwei möglichkeiten eine CollectionView zu erstellen. Entweder definieren wir diese in XAML oder per Code. Definieren wir die CollectionViewSource in XAML können wir hier direkt angeben wie wir Beispielsweise sortieren oder Gruppieren möchten, beide Methoden haben ihre Vorteile, beide aber auch ihre Nachteile.

Machen wir dies mal anhand eines Beispiels und fangen damit an das wir die CollectionViewSource innerhalb unseres XAML Codes definieren:

Wir erstellen ein neues WPF Projekt und erstellen uns eine neue Klasse „Person“ welche wie in der WPF üblich INotifyPropertyChanged implementiert.

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

Public Class Person
    Implements INotifyPropertyChanged
    Public Sub New()
    End Sub
    Public Sub New(firstname As String, lastname As String, Optional gender As
PersonGender = PersonGender.Male)
        Me.FirstName = firstname : Me.LastName = lastname :  Me.Gender = gender
    End Sub

    Private _firstName As String
    Public Property FirstName() As String
        Get
            Return _firstName
        End Get
        Set(ByVal value As String)
            _firstName = value
            RaisePropertyChanged()
            RaisePropertyChanged(NameOf(FullName))
        End Set
    End Property

    Private _lastName As String

    Public Property LastName() As String
        Get
            Return _lastName
        End Get
        Set(ByVal value As String)
            _lastName = value
            RaisePropertyChanged()
            RaisePropertyChanged(NameOf(FullName))
        End Set
    End Property

    Public ReadOnly Property FullName As String
        Get
            Return $"{FirstName} {LastName}"
        End Get
    End Property

    Private _gender As PersonGender = PersonGender.Male

    Public Property Gender() As PersonGender
        Get
            Return _gender
        End Get
        Set(ByVal value As PersonGender)
            _gender = value
            RaisePropertyChanged()
        End Set
    End Property


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

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

Innerhalb der selben Datei können wir nun den Enumerator auch erstellen:

```
Public Enum PersonGender
    Male = 0
    Female = 1
End Enum
```

Wir öffnen nun die *MainWindow.vb* und implementieren hier ebenfalls die INotifyPropertyChanged Schnittstelle sowie eine ObservableCollection(Of Person) um eine Liste von Personen halten zu können:

```
Imports System.Collections.ObjectModel
Imports System.ComponentModel
Imports System.Runtime.CompilerServices

Class MainWindow
    Implements INotifyPropertyChanged

    Private _allPersons As ObservableCollection(Of Person)

    Public Property AllPersons() As ObservableCollection(Of Person)
        Get
            Return _allPersons
        End Get
        Set(ByVal value As ObservableCollection(Of Person))
            _allPersons = value
            RaisePropertyChanged()
        End Set
    End Property

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

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

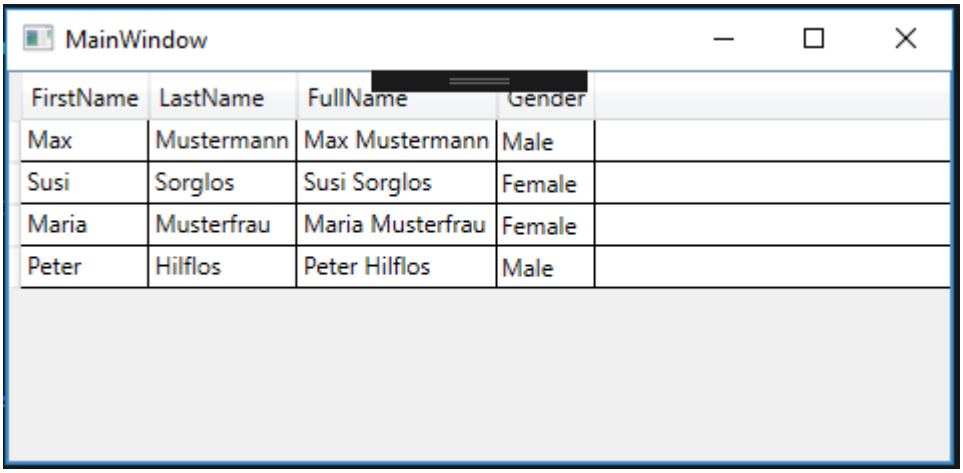
Jetzt müssen wir noch einige Personen zum Testen einfügen. Hierfür bietet sich das Loaded-Ereigniss des Window an.

```
Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles
Me.Loaded
        AllPersons = New ObservableCollection(Of Person)
        AllPersons.Add(New Person("Max", "Mustermann"))
        AllPersons.Add(New Person("Susi", "Sorglos", PersonGender.Female))
        AllPersons.Add(New Person("Maria", "Musterfrau", PersonGender.Female))
        AllPersons.Add(New Person("Peter", "Hilflos"))

        Me.DataContext = Me
    End Sub
```

Nun können wir bereits mit unserem View beginnen und probehalber einfach mal ein DataGrid auf die AllPersons-Eigenschaft binden um zu sehen ob wir hier bereits Daten sehen.

```
<Grid>
    <DataGrid ItemsSource="{Binding AllPersons}" CanUserAddRows="False">

    </DataGrid>
</Grid>
```

In der Tat sehen wir unsere Namen welche wir im Loaded Ereigniss der Auflistung hinzufügen. Warum also eine CollectionView erstellen? Oft will man die Einträge der Auflistung statisch oder dynamisch Sortieren, Gruppieren oder Filtern. Ausserdem bietet die *CollectionView* Eigenschaften wie *CurrentItem* oder *CurrentIndex* welche es uns ermöglichen im Code oder per Binding abzufragen welche Person ausgewählt wurde. Genau genommen wird im Hintergrund durch die WPF automatisch eine CollectionView erstellt welche wir nun nicht zur Gesicht bekommen.



Wir definieren in den Resourcen des Fensters eine *CollectionViewSource* mit einer *SortDescription* um eine Sortierung der Einträge vorzugeben. Um *SortDescription* zur Verfügung zu bekommen müssen wir allerdings den *ComponentModel*-Namespace importieren:

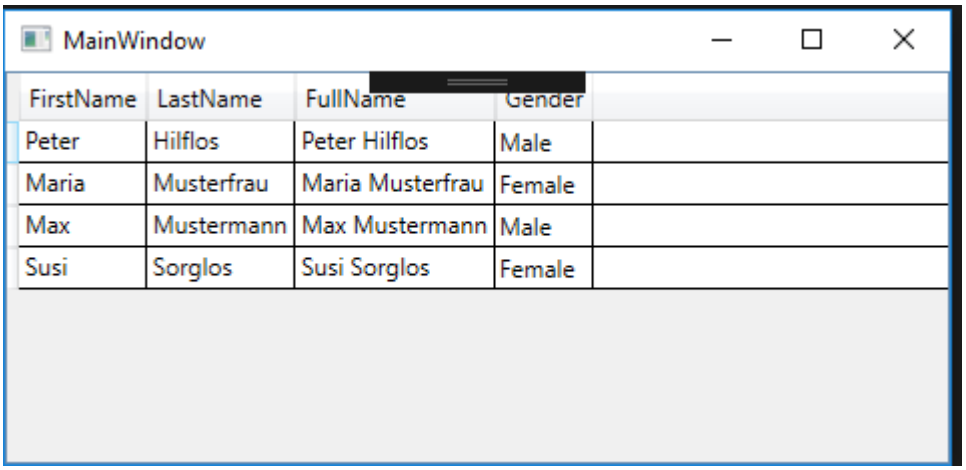
```
xmlns:componentModel="clr-namespace:System.ComponentModel;assembly=WindowsBase"

<Window.Resources>
        <CollectionViewSource Source="{Binding AllPersons}" x:Key="personsView">
            <CollectionViewSource.SortDescriptions>
                <componentModel:SortDescription PropertyName="LastName"/>
            </CollectionViewSource.SortDescriptions>
        </CollectionViewSource>
    </Window.Resources>
```

Allerdings ändert sich nun noch nichts. Wir haben nun eine *CollectionViewSource* erstellt diese allerdings noch nicht an das DataGrid gebunden. Aber wie binden wir nun diese – als Resource definierte *CollectionViewSource* an ein DataGrid?

Indem wir beim Binding an die statische Resource verweisen:

```
<Grid>
   <DataGrid ItemsSource="{Binding Source={StaticResource personsView}}"
CanUserAddRows="False">

   </DataGrid>
</Grid>
```

OK, die CollectionViewsource hat nun als Quelle unsere ObservableCollection der CodeBehind das MainWindow und das DataGrid hat als Quelle für die Items diese CollectionViewSource.
In der *CollectionViewSource* haben wir eine *SortDescription* für die Eigenschaft „LastName“ angegeben damit diese dafür sorgt das alle Einträge nach dem Nachnamen der Person sortiert werden. Probieren wir dies aus indem wir unsere App starten:

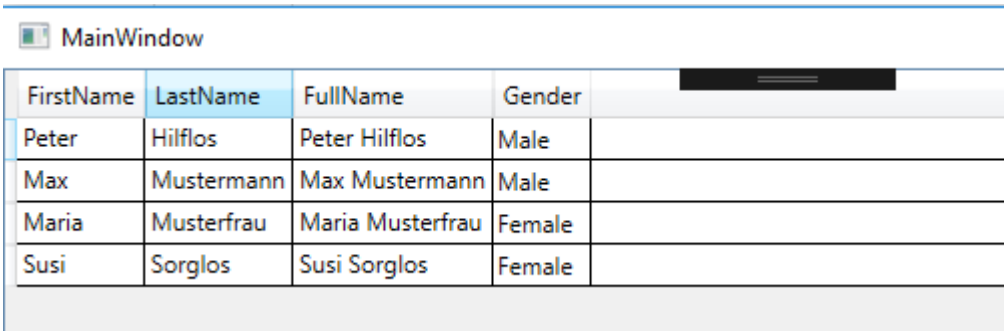


Wir sehen bereits den Unterschied, die Einträge sind nun tatsächlich sortiert.
Gehen wir weiter und versuchen nun die Einträge nach dem Geschlecht zu Gruppieren.

Hierfür müssen wir nur eine „GroupDescription“ zu unserer *CollectionViewSource* hinzufügen und die Eigenschaft nach welcher Gruppiert werden soll angeben. In unserem Fall die Eigenschaft „Gender“ unserer Person Klasse.

```
<CollectionViewSource Source="{Binding AllPersons}" x:Key="personsView">
       <CollectionViewSource.GroupDescriptions>
           <PropertyGroupDescription PropertyName="Gender" />
       </CollectionViewSource.GroupDescriptions>
       <CollectionViewSource.SortDescriptions>
           <componentModel:SortDescription PropertyName="LastName"/>
       </CollectionViewSource.SortDescriptions>
 </CollectionViewSource>
```

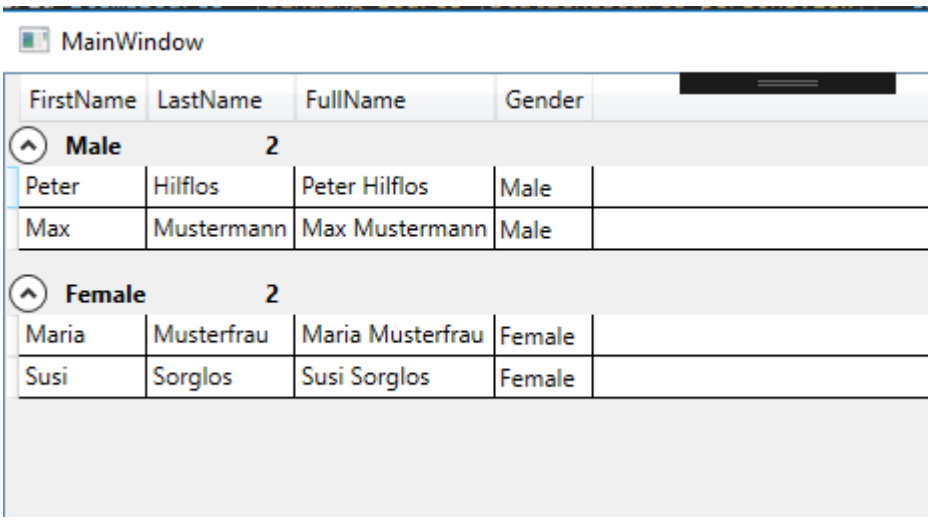
Starten wir nun unsere App nochmals wird uns auffallen das nun zwar Gruppiert wurde, aber irgendwie passt die sortierung nicht mehr. Stimmt nicht ganz, es wurde schon nach den Nachnamen sortiert allerdings innerhalb der Gruppierung.



Die Gruppierung sieht jetzt nicht unbedingt sehr toll aus und zeigt nicht wirklich gut an das nun innerhalb dieser Liste Gruppiert wurde. Das liegt daran das MS keinen *GroupStyle* definiert hat, also machen wir dies schnell mal damit wir die Gruppierung besser erkennen können:

```
<Grid>
   <DataGrid ItemsSource="{Binding Source={StaticResource personsView}}"
CanUserAddRows="False">
       <DataGrid.GroupStyle>
           <GroupStyle>
               <GroupStyle.ContainerStyle>
                   <Style TargetType="{x:Type GroupItem}">
                       <Setter Property="Margin" Value="0,0,0,5"/>
                       <Setter Property="Template">
                           <Setter.Value>
                               <ControlTemplate TargetType="{x:Type GroupItem}">
                                   <Expander IsExpanded="True"
                                             BorderThickness="1,1,1,5">
                                       <Expander.Header>
                                           <DockPanel>
                                               <TextBlock FontWeight="Bold"
                                                          Text="{Binding Path=Name}"
                                                          Margin="5,0,0,0"
                                                          Width="100"/>
                                               <TextBlock FontWeight="Bold"
                                                    Text="{Binding Path=ItemCount}"/>
                                           </DockPanel>
                                       </Expander.Header>
                                       <Expander.Content>
                                           <ItemsPresenter />
                                       </Expander.Content>
                                   </Expander>
                               </ControlTemplate>
                           </Setter.Value>
                       </Setter>
                   </Style>
               </GroupStyle.ContainerStyle>
           </GroupStyle>
       </DataGrid.GroupStyle>
   </DataGrid>
</Grid>
```

Mit diesem Style sehen wir jetzt schön wie Gruppiert wird, wobei innerhalb der Gruppierung auch die Sortierung nach dem Nachnamen korrekt ausgeführt wird.



Wie weiter oben in diesem Kapitel angesprochen hat diese Art der erstellung (innerhalb des XAML) auch Nachteile. Ich muss einen „Umweg“ gehen um das ganze dynamisch zu machen.

Will ich nun per Code-Behind die Sortierung ändern oder eine neue Sortierung hinzufügen, Filtern oder die Gruppierung ändern kann ich dies nicht ohne weiteres machen da die *CollectionViewSource* inkl. ihren Eigenschaften in XAML festgelegt ist.
Wir müssen uns also die *CollectionViewSource* in die CodeBehind hereinholen.
Dies können wir über FindResource machen.

Eine Resource bekommt man indem man die Methode FindResource der Window Klasse aufruft und den Key der Resource als Parameter übergibt. Der Key unserer Resource ist „personsView“.

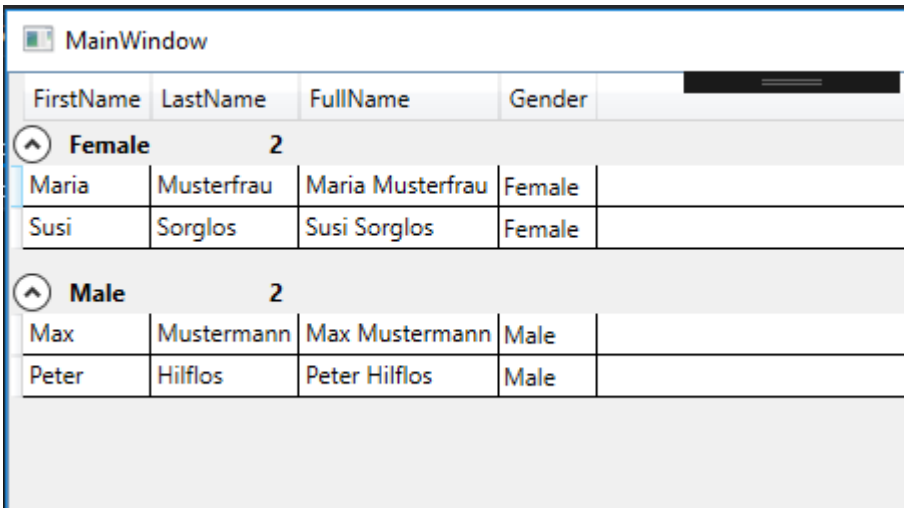
```
Dim colView As CollectionViewSource = CType(Me.FindResource("personsView"),
CollectionViewSource)
```

Die Rückgabe von FindResource muss man Casten da FindResource immer ein Objekt von Typ Object zurückgibt.

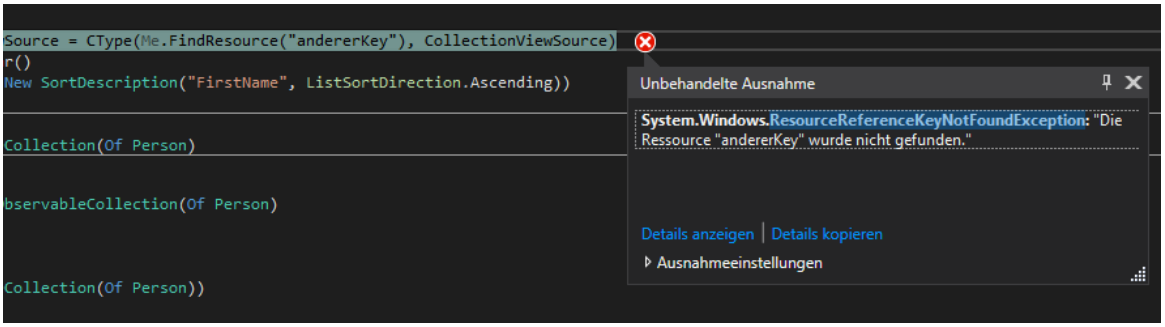
Nun können wir im Code die SortDescription Auflistung zurücksetzen und eine neue SortDescription hinzufügen um Beispielsweise nach dem Vornamen der Personen sortieren zu können.

```
Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles
Me.Loaded
        AllPersons = New ObservableCollection(Of Person)
        AllPersons.Add(New Person("Max", "Mustermann"))
        AllPersons.Add(New Person("Susi", "Sorglos", PersonGender.Female))
        AllPersons.Add(New Person("Maria", "Musterfrau", PersonGender.Female))
        AllPersons.Add(New Person("Peter", "Hilflos"))

        Me.DataContext = Me

        Dim colView As CollectionViewSource = CType(Me.FindResource("personsView"),
CollectionViewSource)
        colView.SortDescriptions.Clear()
        colView.SortDescriptions.Add(New SortDescription("FirstName",
ListSortDirection.Ascending))
    End Sub
```

Wie der folgende Screenshot zeigt wurde nun nach Vornamen sortiert. Nun kann man auch schön sehen das uns die WPF auch die Sortierung der Gruppierung abgeändert hat.



Ich persönlich finde es nicht sehr flexibel die *CollectionViewSource* in XAML zu definieren, ein weiteres Problem ist das wenn man die *CollectionViewSource* aus dem XAML entfernt oder den Key abändert der Compiler dies nicht mitbekommt. Das Programm kompiliert ganz normal und zu Laufzeit bekommt man eine ResourceReferenceKeyNotFoundException um die Ohren geworfen.



Also kommen wir zur zweiten Methode welche ich oben bereits angesprochen hatte. Die erstellung und manipulation einer *CollectionViewSource* per Code.

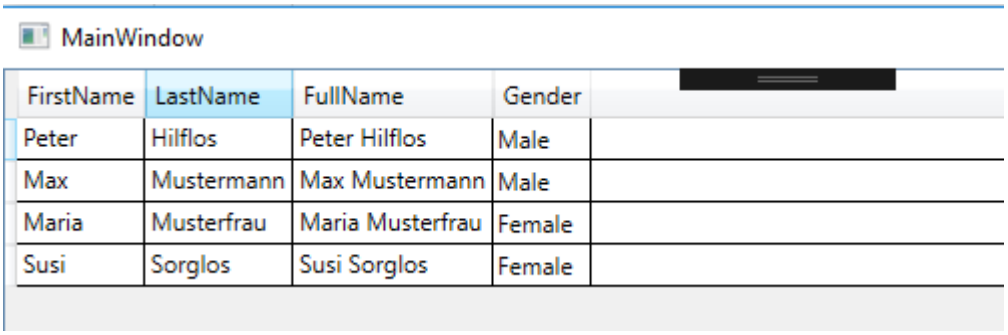
Hier werde ich auch auf ein paar Funktionen der CollectionViewSource auf welche ich bisher nicht eingegangen bin zu sprechen kommen, ihre Verwendung ist allerdings die gleiche für beide Arten der erstellung.

Hierfür erstelle ich abermals ein neues Projekt und füge dort meine *Person* Klasse hinzu. Auch unseren XAML übernehme ich weitestgehend damit ich den *GroupStyle* nicht neu schreiben muss nur das mein Binding der ItemsSource-Eigenschaft des DataGrid nun wieder lediglich auf AllPersons verweist da es unsere Resource nun ja nicht gibt.

Innerhalb der CodeBehind unseres *MainWindows* implementieren wir wieder die INotifyPropertyChanged Schnittstelle und erstellen unsere ObservableCollection(Of Person). In *Window_Loaded* füllen wie abermals die Collection mit Daten.

```
Class MainWindow
    Implements INotifyPropertyChanged
    Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles
Me.Loaded
        AllPersons = New ObservableCollection(Of Person)
        AllPersons.Add(New Person("Max", "Mustermann"))
        AllPersons.Add(New Person("Susi", "Sorglos", PersonGender.Female))
        AllPersons.Add(New Person("Maria", "Musterfrau", PersonGender.Female))
        AllPersons.Add(New Person("Peter", "Hilflos"))

        Me.DataContext = Me
    End Sub

    Private _allPersons As ObservableCollection(Of Person)
    Public Property AllPersons() As ObservableCollection(Of Person)
        Get
            Return _allPersons
        End Get
        Set(ByVal value As ObservableCollection(Of Person))
            _allPersons = value
            RaisePropertyChanged()
        End Set
    End Property

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

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

Nun haben wir wieder unseren Ausgangspunkt wie wir Ihn bereits hatten:



Um nun eine CollectionViewSource zu erstellen und darauf zu Binden erstellen wir uns in der MainWindow.vb ein Property „AllPersonsView“ von Typ ICollectionView.

```
Private _allPersonsView As ICollectionView
    Public Property AllPersonsView() As ICollectionView
        Get
            Return _allPersonsView
        End Get
        Set(ByVal value As ICollectionView)
            _allPersonsView = value
            RaisePropertyChanged()
        End Set
    End Property
```

Im MainWindow_Loaded werden wir nun unser Property instanzieren und zwar nutzen wir hierfür wie schon weiter oben angesprochen die statischen Methode GetDefaultView.

```
AllPersonsView = CollectionViewSource.GetDefaultView(AllPersons)
```

Woraufhin wir nun unser Binding korrigieren können. Statt auf AllPersons können wir nun auf AllPersonsView binden.

Soweit haben wir nun direkt an eine *CollectionView* gebunden welche wir im Code erstellt haben. *Sortieren*, *Gruppieren* und *Filtern* werden wir nun in die Hand nehmen.
Im vorigen Beispiel haben wir bereits gesehen wie wir dies bewerkstelligen können. Der einzige unterschied ist das wir nun direkt auf das Property zugreifen können und uns die *CollectionView* nicht aus den Resourcen holen müssen.

```
Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles
Me.Loaded
        AllPersons = New ObservableCollection(Of Person)
        AllPersons.Add(New Person("Max", "Mustermann"))
        AllPersons.Add(New Person("Susi", "Sorglos", PersonGender.Female))
        AllPersons.Add(New Person("Maria", "Musterfrau", PersonGender.Female))
        AllPersons.Add(New Person("Peter", "Hilflos"))

        AllPersonsView = CollectionViewSource.GetDefaultView(AllPersons)
        AllPersonsView.SortDescriptions.Add(New SortDescription("LastName",
ListSortDirection.Ascending))
        AllPersonsView.GroupDescriptions.Add(New PropertyGroupDescription("Gender"))

        Me.DataContext = Me
    End Sub
```

Gehen wir nun über zum Filtern. Auch zum Filtern bietet die *CollectionView* einen Mechanismus welcher im Grunde sehr leicht handzuhaben ist. Hierfür besitzt sie eine Eigenschaft mit dem Namen *„Filter“* welcher ein Predicate(Of Object) erwartet.

```
AllPersonsView.Filter = AddressOf AllPersonsView_Filter
```

Und die Prozedur:

```
Private Function AllPersonsView_Filter(obj As Object) As Boolean

End Function
```

Jetzt müssen wir uns die Frage stellen was wir Filtern möchten. Im Normalfall hätten wir bestimmt eine Textbox im View in welche der User etwas eintippen kann woraufhin gefiltert werden soll. Das werden wir auch machen. Der Benutzer tippt etwas in die TextBox woraufhin direkt gefiltert werden soll. Hierfür benötigen wir ein Property in *MainWindow.vb* welches unseren Text der Textbox hält und welches wir die TextBox binden.

```
Private _filterText As String
    Public Property FilterText() As String
        Get
            Return _filterText
        End Get
        Set(ByVal value As String)
            _filterText = value
            RaisePropertyChanged()
        End Set
    End Property
```

Unser View ändern wir so ab das wir im oberen Teil eine *TextBox* haben welche auf diese Eigenschaft gebunden ist wobei wir beim Binding die UpdateSourceTrigger Eigenschaft auf PropertyChanged festlegen damit nach jedem Tastenanschlag gefiltert wird.

```
<Window x:Class="MainWindow"
        ...
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <DockPanel>
        <StackPanel Orientation="Horizontal" DockPanel.Dock="Top" Margin="5">
            <Label Content="Filter:"/>
            <TextBox Text="{Binding FilterText,UpdateSourceTrigger=PropertyChanged}"
Width="100" VerticalContentAlignment="Center"/>
        </StackPanel>
        <DataGrid ItemsSource="{Binding AllPersonsView}" CanUserAddRows="False">
           ...
        </DataGrid>
    </DockPanel>
</Window>
```

Immer wenn nun etwas in die TextBox eingegeben wird, wird der Setter des Property durchlaufen was auch ein guter Zeitpunkt ist um das Filtern anzustoßen. Um dies zu tun müssen wie lediglich die Methode Refresh() der *CollectionView* aufrufen, alles andere erledigt die *CollectionView* bzw. unser Delegat (welcher noch leer ist) für uns.

```
Private _filterText As String
    Public Property FilterText() As String
        Get
            Return _filterText
        End Get
        Set(ByVal value As String)
            _filterText = value
            AllPersonsView.Refresh()
            RaisePropertyChanged()
        End Set
    End Property
```

Also müssen wir nun nur noch unseren Delegaten mit leben füllen.
Was passiert nun in unserer Filter-Methode. Die *CollectionView* ruft diese für **JEDES** Element auf und Fragt uns quasi ob das Element gefiltert werden soll oder nicht!

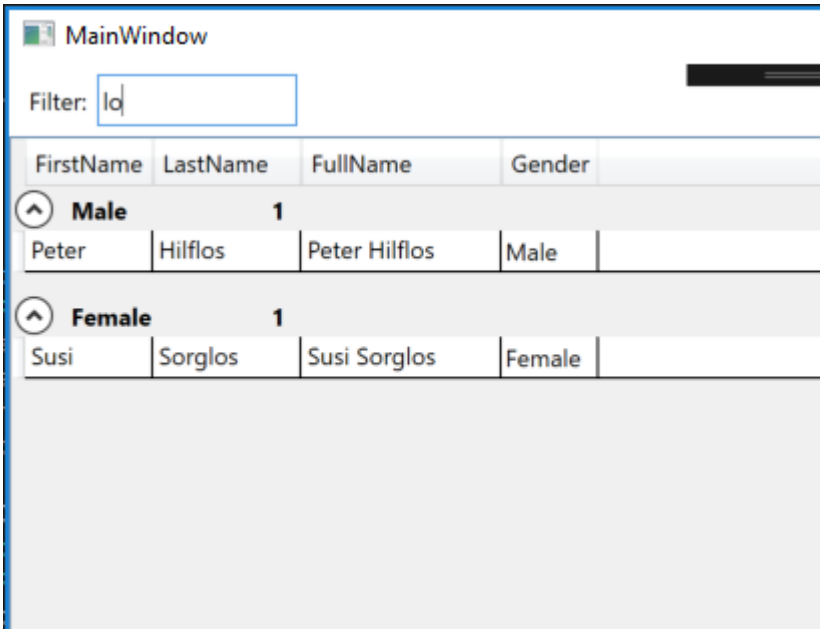
Man würde vermuten das in einer Methode „Filter“ danach „gefragt“ wird ob man ein Element Filtern möchte, dies ist allerdings nicht der Fall. Jedes Element für welches man True zurück gibt wir NICHT gefiltert. Also ist dieser Filter eher wie eine Suche zu verstehen als ein Filter.
Wir möchten das unser Filter (oder auch Suche) mit dem Namen funktioniert und zwar auch mit Teilen davon.

```
Private Function AllPersonsView_Filter(obj As Object) As Boolean
    If String.IsNullOrEmpty(FilterText) Then Return True
    Dim currentPerson As Person = CType(obj, Person)
    Return currentPerson.FullName.ToLower.Contains(FilterText.ToLower)
End Function
```

Zuerst prüfen wir ob unser FilterText-Property nicht Nothing oder leer ist. Ist dies der Fall geben wir alle Elemente direkt zur anzeige frei.

Wir Casten das hereingereichte Objekt in ein *Person-Objekt* und prüfen ob der *FilterText* im *FullName* der Person enthalten ist wobei wir hier die Groß – Kleinschreibung außer acht lassen.

Folgender Screenshot zeigt das Ergebnis:



Der Filter funktioniert und arbeitet wie gewollt. Wenn Ihr das Beispiel ausprobiert setzt mal einen Haltepunkt auf das *AllPersons* Property und seht dort mal nach welche Einträge dort vorhanden sind wenn der Filter aktiv ist. Ihr werdet sehen das dort IMMER alle Elemente verbleiben. Es wird lediglich in der CollectionView gefiltert, die zugrundeliegende Collection bleibt wie sie ist und bleibt auch von der Sortierung und der Gruppierung unberührt. Was sehr von Vorteil für die weiterbearbeitung sein kann. Fast immer ist es so das es für den Programmierer unerheblich ist wie im View die Daten gerade angezeigt werden, wichtiger ist diesem meist das die Daten so bleiben wie sie sind damit sich Beispielsweise der Index eines Elements nicht ändert nur weil der Benutzer gerade die Sortierung geändert hat. Für den Programmierer bleibt „seine“ Liste immer wie sie ist was ich sehr schön finde.

Hier geht’s zum Video vom Teil 1: https://youtu.be/BNt52DxpAPw

Hier geht’s zum Video vom Teil 2: https://youtu.be/hMwbpJ7f8Tw

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/

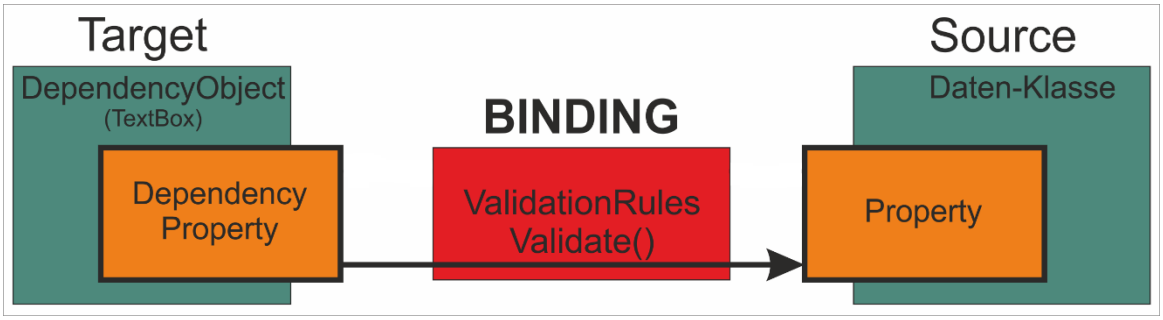
2.1.4.7

Validieren von Benutzereingaben

Eine oft unterschätzte aber meiner Meinung nach sehr wichtige Aufgabe eines Programms ist die Validierung von Benutzereingaben auf ihre Richtigkeit. Hat der Benutzer einen gültigen Wert angegeben? Ich sehe es immer wieder Programme welche Benutzereingaben schlichtweg nicht prüfen bzw. nur beim Speichern prüfen. Soll bedeuten, der Benutzer kann eingeben was er möchte und beim Speichern erhält dieser eine Meldung das die Eingaben nicht korrekt wären. Oft sogar ohne jegliche Information wo nun die Falscheingabe vorliegt oder wo genau der Fehler liegt. Ich sehe immer wieder Meldungen wie „Bitte korrigieren Sie die Eingaben“. Das ist natürlich keine gute Benutzerführung. Die WPF bietet hierfür ValidationRules.

Um die vom Benutzer eingegebenen Daten zu validieren, besitzt das *Binding*-Objekt eine *ValidationRules*-Property. Alle dort hinterlegten Regeln greifen allerdings nur wenn ein Wert von der *Target*-Property zur *Source*-Property geschrieben wird. Also im *TwoWay*-Mode bzw. im *OneWayToSource*-Mode eines Bindings.

Folgende Abbildung zeigt das Modell der Validierung:



Wie gut zu sehen greift die Validierung genau im Bindingmechanismus ein. Werden Daten nicht zurückgeschrieben wird auch nicht Validiert. Das bedeutet allerdings auch das eine Validierung bei einem Binding mit dem UpdateSourceTrigger=LostFocus erst ausgeführt wird wenn die TextBox den Focus verliert.

Zur *ValidationRules* Property werden Instanzen vom Typ *ValidationRule* hinzugefügt. Die ValidationRule Klasse selbst ist Abstrakt und enthalt die abstrakte Validate Methode.

Die Validate Methode kann von Subklassen überschrieben werden. Die WPF bringt von Haus aus zwei Subklassen. Die ExceptionValidationRule und die DataErrorValidationRule, aber natürlich können auch eigene ValidationRules erstellt werden was wir im Zuge dieses Kapiteln auch machen werden.

Betrachten wir uns erst eine BuildIn *ValidationRule* anhand eines Beispiels um danach eine eigene *ValidationRule*-Klasse zu erstellen. Zu guter Letzt werden wir uns die DataErrorValidationRule ansehen und was es hiermit genau auf sich hat.

Sehen wir uns anhand eine Beispiels die Funktionsweise von *ValidationRules* an indem wir ein neues Projekt erstellen und den Preis eines Fahrzeugs eingeben. Üblicherweise kann bei einem Preis für ein Auto kein Negativer Preis eingegeben werden.

```
<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:_2_1_4_7_Validierung"
        mc:Ignorable="d"
        Title="MainWindow" Height="127.34" Width="386.383" Language="de">
    <Window.DataContext>
        <local:Car Price="2379.99"/>
    </Window.DataContext>
    <Grid Margin="5">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <StackPanel Orientation="Horizontal">
            <Label Content="_Preis"/>
            <TextBox x:Name="txtPrice"
                     Width="60"
                     Text="{Binding Price,UpdateSourceTrigger=PropertyChanged}"
                     VerticalContentAlignment="Center"/>
        </StackPanel>
        <StackPanel Orientation="Horizontal" Grid.Row="1">
            <TextBlock Text="Aktueller Preis des Fahrzeugs: "/>
            <TextBlock Text="{Binding Price, StringFormat=\{0:C\}}"/>
        </StackPanel>
    </Grid>
</Window>
```

In diesem Feld kann nun ein Negative Wert eingegeben werden, dies gilt es nun zu verhindern da dies nicht möglich sein soll.

Wir fügen also unserem Binding der TextBox eine ExceptionValidationRule hinzu.

```
<TextBox x:Name="txtPrice"
                     Width="60"
                     VerticalContentAlignment="Center">
     <Binding Path="Price" Mode="TwoWay" UpdateSourceTrigger="PropertyChanged">
          <Binding.ValidationRules>
                 <ExceptionValidationRule/>
          </Binding.ValidationRules>
     </Binding>
</TextBox>
```

Aber noch immer tut sich nichts. Das liegt daran das keine Exception geworfen wird. Der Datentyp Decimal lässt ja negative Werte zu. Wir müssen also eine Exception werfen falls ein Negativer Wert eingegeben wurde. Gehen wir hierfür in die *Car*-Klasse und sorgen dafür dass eine *Exception* geworfen wird, gesetzt den Fall das ein Negativer Wert übertragen wird.

```
Private _price As Decimal
    Public Property Price() As Decimal
        Get
            Return _price
        End Get
        Set(ByVal value As Decimal)
            If value < 0 Then
                Throw New ArgumentOutOfRangeException("Der Preis darf nicht negativ
sein!")
            End If
            _price = value
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("Price"))
        End Set
    End Property
```

Geben wir nun einen Negativen Wert in die TextBox ein indem wir einfach ein – vor den Preis setzen läuft der Debugger in genau diese Exception, wird drücken F5 damit das Programm weiterläuft und sehen das wir nun eine rot umrandete TextBox zu Gesicht bekommen, sowie das der Preis nicht übertragen wurde.

Außerdem sehen wir in der Ausgabe von VisualStudio das es hier einen DataError gab, inkl. unseres Textes.

```
System.Windows.Data Error: 8 : Cannot save value from target back to source.
BindingExpression:Path=Price; DataItem='Car' (HashCode=58328727); target element is
'TextBox' (Name='txtPrice'); target property is 'Text' (type 'String')
ArgumentOutOfRangeException:'System.ArgumentOutOfRangeException: Das angegebene
Argument liegt außerhalb des gültigen Wertebereichs.
Parametername: Der Preis darf nicht negativ sein!
```

Als nächsten werden wir eine eigene ValidationRule erstellen und diese einbinden.
Um eine *ValidationRule* zu erstellen erzeugen wir einfach eine Klasse welche von *ValidationRule* erbt und die Validate-Methode überschreibt. Üblicherweise folgt der Klassenname einer ValidationRule der Konvention *<Validatorname>ValidationRule*.

Wir können hier nun auf alle Eventualitäten prüfen und dementsprechend ein *ValidationResult* zurückgeben.

```
Public Class CarPriceValidationRule
    Inherits ValidationRule

    Public Overrides Function Validate(value As Object, cultureInfo As CultureInfo) As
ValidationResult
        Dim val As Decimal
        If Decimal.TryParse(value.ToString, val) Then
            If val < 0 Then Return New ValidationResult(False, "Der Preis darf nicht
kleiner als 0 sein!")
            Return ValidationResult.ValidResult
        Else
            Return New ValidationResult(False, "Ungültiger Wert!")
        End If
    End Function
End Class
```

HINWEIS:

Bei der Verwendung einer eigenen ValidationRule muss der Setter der Car-Klasse nicht verändert werden so dass dieser eine Exception wirft. Das ist gerade dann gut wenn man evtl. keinen Zugriff auf die Datenklasse hat.

Das Ergebnis im View sieht genau gleich aus. Die TextBox wird rot umrandet und zeigt uns somit eine Falscheingabe an. Der einzige Unterschied hier ist im Moment das der Debugger die Ausführung des Programms nicht anhält, da keine Exception geworfen wurde.

Zu guter Letzt kommen wir nun zur Validierung mit der DataErrorValidationRule.

Die dritte Möglichkeit basiert auf dem seit .Net 1.0 existierenden Interface IDataErrorInfo welches zwei Property enthält.

Wir implementieren also in unsere Car-Klasse das IDataErrorInfo-Interface wodurch uns zwei Eigenschaften erzeugt werden:

```
Public Class Car
    Implements INotifyPropertyChanged
    Implements IDataErrorInfo

    Private _price As Decimal
    Public Property Price() As Decimal
        Get
            Return _price
        End Get
        Set(ByVal value As Decimal)
            _price = value
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("Price"))
        End Set
    End Property

    Default Public ReadOnly Property Item(columnName As String) As String Implements
IDataErrorInfo.Item
        Get
            Throw New NotImplementedException()
        End Get
    End Property

    Public ReadOnly Property [Error] As String Implements IDataErrorInfo.Error
        Get
            Throw New NotImplementedException()
        End Get
    End Property

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

Von der WPF wird lediglich die *Item*-Property genutzt. Die Error-Property kann also ruhig *Nothing* zurückgeben.

Die Item-Property wird jedes Mal von der WPF aufgerufen wenn sich ein Wert in einem Feld ändert welches eine *DataErrorValidationRule* im Binding implementiert oder innerhalb des Binding die Eigenschaft *ValidatesOnDataError* auf True gesetzt wurde und übergibt als Parameter (columnName) den Namen des Binding, was in unserem Fall im Moment nur „Price“ sein kann.

In diesem Fall können wir nun wieder die wesentlich kürzere Attributsyntax verwenden um das Binding zu setzen:

```
<TextBox x:Name="txtPrice"
                     Width="60"
                     VerticalContentAlignment="Center"
                     Text="{Binding
Price,UpdateSourceTrigger=PropertyChanged,ValidatesOnDataErrors=True}">
            </TextBox>
```

Allerdings hat die Validierung über IDataErrorInfo auch einen Nachteil. Der Wert wird erst validiert nachdem dieser gesetzt wurde, in unserem Fall wird also der negative Wert in unserer Car-Klasse persistiert, nun müssen wir auf jeden Fall das Speichern verhindern.

Mir persönlich ist dennoch diese Methode die liebste da ich hier sehr flexibel bin und die kurze Schreibweise vom Binding über die Attributsyntax bevorzuge.

Aber wofür schrieben wir die ganze Zeit eine Errormeldung wenn die WPF diese überhaupt nicht anzeigt? Wir sehen lediglich einen roten Rahmen der anzeigt das es irgendeinen Fehler gibt. Damit der rote Rahmen angezeigt wird nutzt die *TextBox* AttachedProperties der *Validation*-Klasse. Die folgenden AttachedProperties stehen hier zur Verfügung: *Errors*, *HasError* und *ErrorTemplate*.

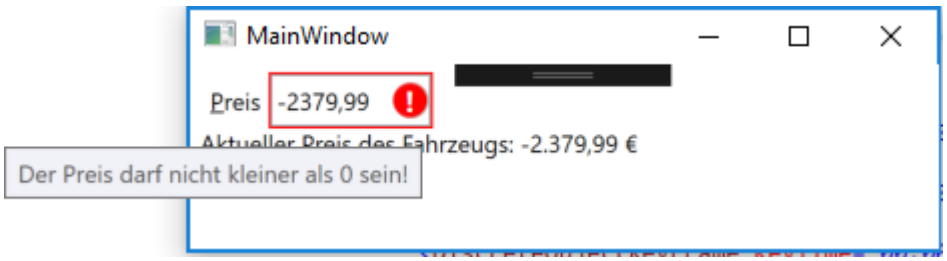
Wir werden nun einen Style für die ErrorTemplate-Eigenschaft definieren um das ErrorTemplate etwas informativer zu gestalten und dem User einige mehr Infos mit auf dem Weg zu geben. Hier ist eure Fantasie gefragt, wichtig ist in einem Style für ein ErrorTemplate das dieses ein AdornerElementPlaceholder Objekt enthält. Dieses gilt als Platzhalter für die TextBox.

Wir legen uns also einen Style für eine TextBox in den Window-Resourcen an und überschreiben den Style für die Eigenschaft *Validation.ErrorTemplate* welcher auch gleich der Error-Text in einem Tooltip über `AdornedElement.(Validation.Errors)[0].ErrorContent` anzeigen kann.

```
<Style TargetType="TextBox">
            <Setter Property="Validation.ErrorTemplate">
                <Setter.Value>
                    <ControlTemplate>
                        <DockPanel LastChildFill="True">
                            <Ellipse DockPanel.Dock="Right"
                                 ToolTip="{Binding ElementName=myTextbox,
Path=AdornedElement.(Validation.Errors)[0].ErrorContent}"
                                   Margin="-20,0,2,0" Height="18" StrokeThickness="1"
Fill="Red" >
                                <Ellipse.Stroke>
                                    <LinearGradientBrush EndPoint="1,0.5"
StartPoint="0,0.5">
                                        <GradientStop Color="#FFFA0404" Offset="0"/>
                                        <GradientStop Color="#FFC9C7C7" Offset="1"/>
                                    </LinearGradientBrush>
                                </Ellipse.Stroke>
                                <Ellipse.Triggers>
                                    <EventTrigger
RoutedEvent="FrameworkElement.Loaded">
                                        <BeginStoryboard Storyboard="{StaticResource
FlashErrorIcon}"/>
                                    </EventTrigger>
                                </Ellipse.Triggers>
                            </Ellipse>
                            <TextBlock DockPanel.Dock="Right"
                                ToolTip="{Binding ElementName=myControl,

Path=AdornedElement.(Validation.Errors)[0].ErrorContent}"
                                Foreground="White"
                                FontSize="11pt"
                                Margin="-14,0,0,0" FontWeight="Bold"
VerticalAlignment="Center">!
                                <TextBlock.Triggers>
                                    <EventTrigger
RoutedEvent="FrameworkElement.Loaded">
                                        <BeginStoryboard Storyboard="{StaticResource
FlashErrorIcon}"/>
                                    </EventTrigger>
                                </TextBlock.Triggers>
                            </TextBlock>
                            <Border BorderBrush="Red" BorderThickness="1">
                                <AdornedElementPlaceholder Name="myControl"/>
                            </Border>
                        </DockPanel>
                    </ControlTemplate>
                </Setter.Value>
            </Setter>
            <Style.Triggers>
                <Trigger Property="Validation.HasError" Value="true">
                    <Setter Property="ToolTip"
                        Value="{Binding RelativeSource={x:Static RelativeSource.Self},
                        Path=(Validation.Errors)[0].ErrorContent}"/>
                </Trigger>
            </Style.Triggers>
        </Style>
```

Damit das UI entwas intreressanter wird habe ich noch eine Animation erstellt welche die Ellipse alle 200ms blinken lässt.

```
<Storyboard x:Key="FlashErrorIcon">
            <ObjectAnimationUsingKeyFrames BeginTime="00:00:00"

Storyboard.TargetProperty="(UIElement.Visibility)">
                <DiscreteObjectKeyFrame KeyTime="00:00:00" Value="{x:Static
Visibility.Hidden}"/>
                <DiscreteObjectKeyFrame KeyTime="00:00:00.2000000" Value="{x:Static
Visibility.Visible}"/>
                <DiscreteObjectKeyFrame KeyTime="00:00:00.4000000" Value="{x:Static
Visibility.Hidden}"/>
                <DiscreteObjectKeyFrame KeyTime="00:00:00.6000000" Value="{x:Static
Visibility.Visible}"/>
                <DiscreteObjectKeyFrame KeyTime="00:00:00.8000000" Value="{x:Static
Visibility.Hidden}"/>
                <DiscreteObjectKeyFrame KeyTime="00:00:01" Value="{x:Static
Visibility.Visible}"/>
            </ObjectAnimationUsingKeyFrames>
        </Storyboard>
```



Jetzt bleibt noch eine oft gestellte Frage über. Wie kann man ALLE aktuellen Validierungsfehler anzeigen lassen? Oft möchte man dem Benutzer irgendwo im UI alle Fehler anzeigen.
Gibt es mehrere Felder welche Validierungsfehler haben könnten wäre es interessant dem User etwas in der Art anzeigen zu lassen:

- *Der Preis darf nicht kleiner als 0 sein*
- *Die Marke ist ein Pflichtfeld*
- *Die Farbe darf keine Sonderzeichen enthalten*

Für solche Anzeigen habe ich bereits sehr viele Lösungen gesehen und in jedem Buch findet man glaube ich weitere. Jede für sich hat Ihre Vorteile, jede auch ihre Nachteile.

Oft kommt man auf die Nachteile erst wenn man sich später mit MVVM beschäftigt und somit keine Möglichkeit mehr hat auf das Window zuzugreifen da das Window die Eigenschaft BindingGroup besitzt und man über diese auch Fehler auslesen kann, gesetzt den Fall man hat vorher BindingGroups definiert. Die meisten Beispiele in Büchern stützen sich auf diese BindingGroups wobei ich dies nicht unterstützen möchte da wir später abermals umlernen müssten. Außerdem hat man bei den meisten Lösungen nicht die Möglichkeit innerhalb der

Klasse die Fehler abzufragen da fast alle Lösungen rein auf der View-Ebene stattfinden. Da ich eher jemand bin der gerne vom Code aus über die View Bescheid weis finde ich diese Lösungen eher suboptimal. Ich stelle hier eine Lösung vor welche vielleicht am Anfang etwas umständlicher erscheint aber bei genauerem Hinsehen werdet Ihr erkennen das dies nicht der Fall.

Erstmal passen wir unsere Car-Klasse an und fügen noch die Eigenschaften *Marke* und *Modell* hinzu damit wir mehrere Eigenschaften haben welche wir auf Ihre Gültigkeit prüfen können.

Wir fügen folgende Eigenschaften ein:

```
Private _marke As String
    Public Property Marke() As String
        Get
            Return _marke
        End Get
        Set(ByVal value As String)
            _marke = value
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("Marke"))
        End Set
    End Property

    Private _modell As String
    Public Property Modell() As String
        Get
            Return _modell
        End Get
        Set(ByVal value As String)
            _modell = value
            RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs("Modell"))
        End Set
    End Property
```

Auch die View wird angepasst um diese neuen Eigenschaften anzeigen zu können, weiter fügen wir ein *ItemsControl* hinzu um später die Fehler anzeigen zu können:

```
<Window x:Class="MainWindow"
       ...
        Title="MainWindow" Height="202.755" Width="385.147" Language="de">
    <Window.Resources>
        <Storyboard x:Key="FlashErrorIcon">
            ...
        </Storyboard>
        <Style TargetType="TextBox">
              ...
        </Style>
    </Window.Resources>
    <Window.DataContext>
        <local:Car Price="-2379.99" Marke="Volkswagen" Modell="Golf"/>
    </Window.DataContext>
    <Grid Margin="5">
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <StackPanel Orientation="Horizontal">
            <Label Content="M_arke"/>
            <TextBox x:Name="txtMarke"
                     Width="80"
                     VerticalContentAlignment="Center"
                     Text="{Binding
Marke,UpdateSourceTrigger=PropertyChanged,ValidatesOnDataErrors=True}">
            </TextBox>
        </StackPanel>
        <StackPanel Orientation="Horizontal" Grid.Row="1">
            <Label Content="_Modell"/>
            <TextBox x:Name="txtModell"
                     Width="80"
                     VerticalContentAlignment="Center"
                     Text="{Binding
Modell,UpdateSourceTrigger=PropertyChanged,ValidatesOnDataErrors=True}">
            </TextBox>
        </StackPanel>
        <StackPanel Orientation="Horizontal" Grid.Row="2">
            <Label Content="_Preis"/>
            <TextBox x:Name="txtPrice"
                     Width="80"
                     VerticalContentAlignment="Center"
                     Text="{Binding
Price,UpdateSourceTrigger=PropertyChanged,ValidatesOnDataErrors=True}">
            </TextBox>
        </StackPanel>
        <ItemsControl Grid.Row="3" Foreground="DarkRed"/>


    </Grid>
</Window>
```

Damit haben wir unser View mal angepasst.

Fügen wir nun eine Funktion in unsere Car-Klasse ein welche uns eine List(Of ValidationResult) zurückgibt:

```
Public Function ValidationErrors() As List(Of ValidationResult)
    Dim valRet As New List(Of ValidationResult)

    Return valRet
End Function
```

Diese ist noch ohne „Leben“ was wir allerdings gleich ändern werden, vorher allerdings passen wir den Code von unserem Item-Property an welches wir weiter oben vom IDataErrorInfo Interface implementiert hatten an.

```
Default Public ReadOnly Property Item(columnName As String) As String Implements
IDataErrorInfo.Item
    Get
        Dim valRes As ValidationResult = ValidationErrors.Where(Function(v)
v.MemberNames.Contains(columnName) = True).FirstOrDefault

        If valRes Is Nothing Then Return Nothing

        Return valRes.ErrorMessage
    End Get
End Property
```

Vorher hatten wir unseren Code zum Validieren der Felder innerhalb dieses Getters. Dies überlassen wir ab sofort der neu hinzugefügten Funktion „*ValidationErrors*“ und geben hier lediglich das erste Vorkommen eines Fehlers des gesuchten Feldes zurück.
Nun prüfen wir in „ValidationErrors“ alle Eigenschaften auf Richtigkeit:

```
Public Function ValidationErrors() As List(Of ValidationResult)
    Dim valRet As New List(Of ValidationResult)

    If String.IsNullOrEmpty(Marke) Or String.IsNullOrEmpty(Modell) Then
        Dim fieldList As New List(Of String)
        If String.IsNullOrEmpty(Marke) Then fieldList.Add(NameOf(Marke))
        If String.IsNullOrEmpty(Modell) Then fieldList.Add(NameOf(Modell))
        valRet.Add(New ValidationResult("Die Pflichtfelder müssen alle ausgefüllt
werden!", fieldList))
    Else
        If Marke.Length < 3 Then valRet.Add(New ValidationResult($"'{NameOf(Marke)}'
muss aus mindestens 2 Zeichen bestehen!", New List(Of String) From {NameOf(Marke)}))
        If Modell.Length < 3 Then valRet.Add(New ValidationResult($"'{NameOf(Modell)}'
muss aus mindestens 3 Zeichen bestehen!", New List(Of String) From {NameOf(Modell)}))
    End If

    If Price < 0 Then valRet.Add(New ValidationResult($"'{NameOf(Price)}' darf nicht
kleiner als 0 sein!", New List(Of String) From {NameOf(Price)}))
    Return valRet
End Function
```

Ich denke der Code erklärt sich von selbst. Alle Eigenschaften werden auf Ihre Gültigkeit überprüft, wobei für *Marke* und *Modell* die minimale Textlänge geprüft wird und für *Price* wie schon gehabt auf Negative Werte geprüft wird.

Da wir auf die Funktion vom View aus nicht direkt binden können fügen wir jetzt noch ein ReadOnly Property ein welches uns die Fehler (List(Of ValidationError)) als String zurückgibt indem wie aus ValidationErrors rein die ErrorMessage selektieren.

```
Public ReadOnly Property ErrorsList As List(Of String)
    Get
        Return ValidationErrors.Select(Function(e) e.ErrorMessage).ToList()
    End Get
End Property
```

Damit die WPF die ErrorList Eigenschaft immer neu abfragt sollte sich etwas ändern, werden wir diese darüber benachrichtigen indem wie das PropertyChanged-Event werfen.
Hierfür fügen wir noch eine Zeile im Item-Property ein so dass wir nun folgenden Code im Getter haben:

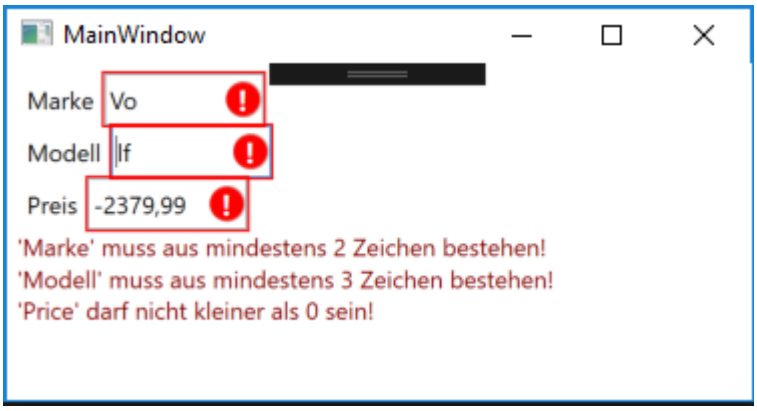
```
Dim valRes As ValidationResult = ValidationErrors.Where(Function(v)
v.MemberNames.Contains(columnName) = True).FirstOrDefault
RaiseEvent PropertyChanged(Me, New PropertyChangedEventArgs(NameOf(ErrorsList)))

If valRes Is Nothing Then Return Nothing

Return valRes.ErrorMessage
```

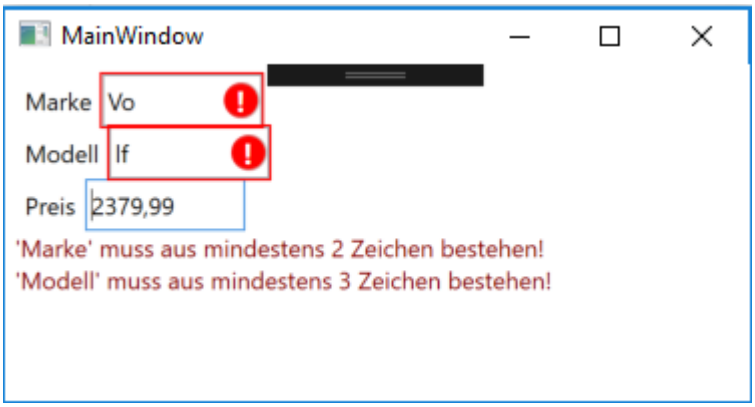
Damit haben wir alles erledigt und können im View darauf Binden:

```
<ItemsControl Grid.Row="3" ItemsSource="{Binding ErrorsList}" Foreground="DarkRed"/>
```

Das Ergebnis ist genau das was wir wollten. In eine Anwendung könnten wir nun Beispielsweise einen Speichern-Button deaktivieren wenn ValidationErrors.Any() True zurückgeben würde.

Ein speichern wäre dann erst gar nicht möglich solange noch Fehler vorhanden sind.
Ein weiterer Vorteil an dieser Methode gegenüber der Validierung rein im View ist das ich hier im Code auf andere Dinge eingehen kann wie Beispielsweise ob es das Fahrzeug bereits in der Datenbank gibt oder ob eine Fahrgestellnummer gültig ist. Das kann die View nicht da hierfür Logik notwendig wäre. Auch diese Fehler könnte ich anschließend anzeigen lassen.





Ich hoffe das ich euch damit auf den Geschmack gebracht habe so dass Ihr in Zukunft Wert auf eine gute Validierung von Daten legt. Das ist meiner Meinung nach ein sehr wichtiger Aspekt in der Softwareentwicklung und unterstützt den Benutzer ungemein bei diversen Eingaben.

Hier geht's zum Video: https://youtu.be/4ZB_uTmZGK8

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/

2.1.4.8

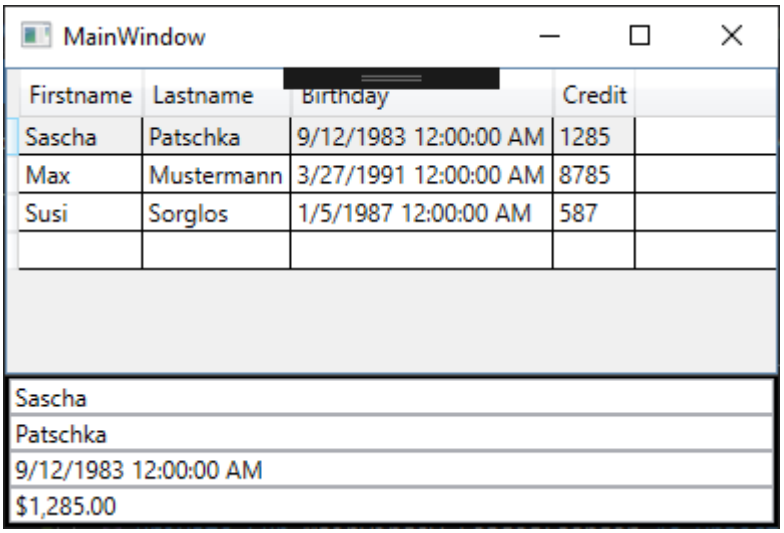
Rücksicht nehmen auf die aktuelle Culture

Die WPF biete diverse sinnvolle Mechanismen um z.b. den Text in einer Textbox so darzustellen wie dies der aktuelle Anwendungsfall erfordert. Das StringFormat beim Binding ist hier ein sehr gutes Beispiel welches wir bereits unter anderem in Kapitel 2.1.4.7 verwendet hatten.

Aber wie sieht es eigentlich aus mit der Kultur? Kann ich das selbst steuern? Was ist der Default wert bei einer WPF Anwendung? Wie kann ich steuern ob wie ein Datum angezeigt wird oder welches Währungssymbol hinter einem Betrag steht? Das sind Fragen auf welche wir heute eingehen werden.

Erstmal sein gesagt das die WPF per Default von en-US als Kultureinstellung ausgeht. Das bedeutet dass in allen Controls (DataGrid, TextBox, Label usw.) das amerikanische Format verwendet wird. Um dies zu demonstrieren habe ich mal ein Window erstellt welches dies demonstriert.

```
<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"
        mc:Ignorable="d"
        Title="MainWindow" Height="450" Width="800">
    <DockPanel LastChildFill="True">
        <Border BorderThickness="2" BorderBrush="Black" DockPanel.Dock="Bottom">
            <StackPanel DataContext="{Binding SelectedPerson}">
                <TextBox Text="{Binding Firstname}"/>
                <TextBox Text="{Binding Lastname}"/>
                <TextBox Text="{Binding Birthday}"/>
                <TextBox Text="{Binding Credit, StringFormat=\{0:C\}}"/>
            </StackPanel>
        </Border>
        <DataGrid ItemsSource="{Binding Persons}" SelectedItem="{Binding
SelectedPerson}"/>
    </DockPanel>
</Window>
```



Wie man gut erkennen kann wird sowohl für den Geburtstag als auch für das Guthaben der Personen die amerikanische Kultur verwendet. Bis hin zum Dollar-Zeichen. Das kann auch zu Problemen führen. Wenn wir und die TextBox für das Guthaben unten genauer ansehen sehen wir das hier das „Tausender-Trennzeichen“ ein Komma ist und das Komma ein Punkt.

Viele gehen nun her und verwenden schlicht statt dem StringFormat `\{0:C\` ein eigenes StringFormat und denken dass damit die Sache erledigt ist. Weit gefehlt.
Gibt der User nun einen Betrag wie 1493,67 ein wird die WPF in diesem Fall nämlich 149367 als Wert zurückspeichern. Fatal! Man müsste nämlich 1494.67 eingeben.

Also ist es wichtig dass wir die richtige Kultur (Culture) verwenden. Ob man nun die Culture des Windows Benutzers mittels CultureInfo.CurrentCulture verwendet oder ob man es den Benutzer aussuchen lässt (Beispielsweise über Einstellungen) bleibt einem hier selbst überlassen.

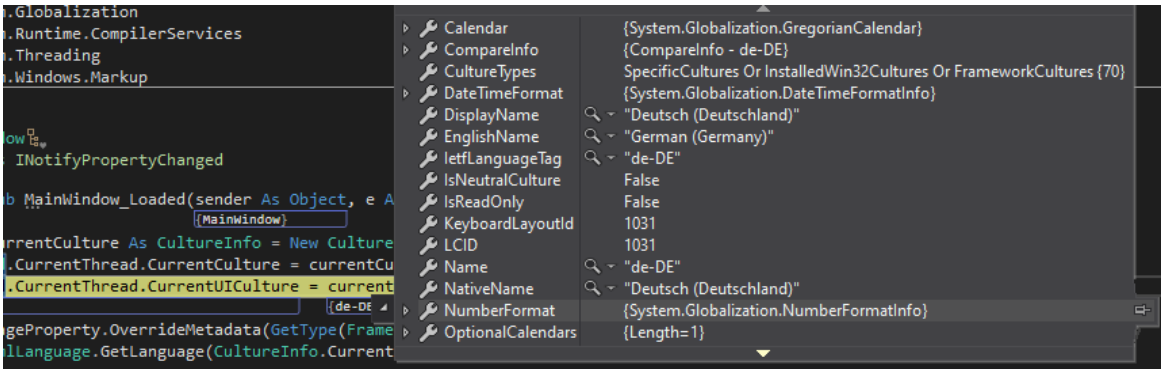
Am besten setzt man die Kultur in der Application_Startup Methode, kann allerdings natürlich auch sonst wo gesetzt werden wie Beispielsweise im Window_Loaded des Startfensters.

```
Dim currentCulture As CultureInfo = CultureInfo.CurrentCulture
Thread.CurrentThread.CurrentCulture = currentCulture
Thread.CurrentThread.CurrentUICulture = currentCulture
LanguageProperty.OverrideMetadata(GetType(FrameworkElement), New
FrameworkPropertyMetadata(
            XmlLanguage.GetLanguage(CultureInfo.CurrentCulture.IetfLanguageTag)))
```

Möchte man nicht die Systemkultur verwenden dann kann die erste Zeile wie folgt ersetzt werden:

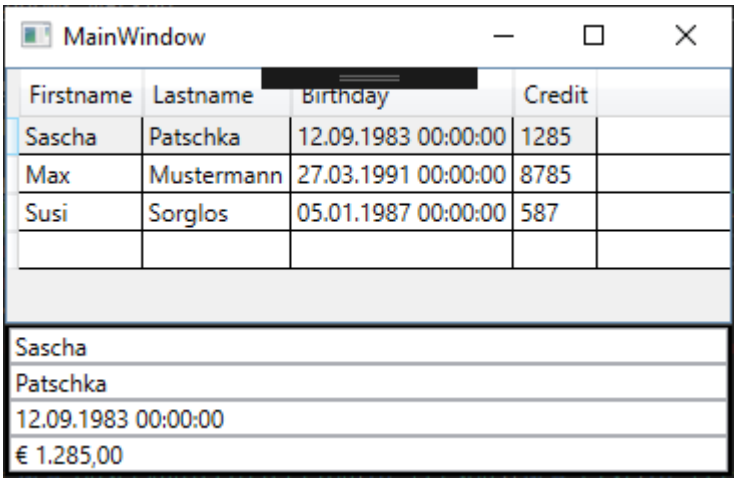
```
Dim currentCulture As CultureInfo =  New CultureInfo("de-DE")
```

So könnte in den Programmeinstellungen der String für die Kultur hinterlegt werden.
Hier ein kleiner Einblick was in der Culture so hinterlegt ist:



Wie wir sehen geht es bei der Kultur nicht einfach nur um die Sprache, sondern auch um Tastaturlayouts, Kalender, Nummernformate usw.! Es gibt auch Sprachen welche man von rechts nach links liest wie z.b. Arabisch.

OK, sehen wir uns mal an wie dies aussieht wenn wir die aktuelle Systemsprache auf das oben bereits gezeigt Fenster anwenden indem wir im Window_Loaded Event die Culture setzen.



Das sieht schon besser aus. Wir haben sowohl das Deutsche Datumsformat als auch das Euro-Symbol und das korrekte Trennzeichen. Geben wird nun einen Betrag mit einem Komma wie 1657,47 ein wird der Wert nun auch korrekt gespeichert.

Als nächstes sehen wir uns an wie dies mit Convertern aussieht. Converter haben wir ja bereits in Kapitel 2.1.4.4 kennengelernt. Nun sehen wir uns an wie wir innerhalb eines Converters auf die aktuelle Kultur eingehen.

Hierfür erstellen wir einen Konverter mit dem Namen „TestConverter“. Dieser konvertiert im Moment nichts. Er soll uns nur dazu dienen zu sehen welche Culture wir in einem Converter hineingereicht bekommen.

```
Public Class TestConverter
    Implements IValueConverter

    Public Function Convert(value As Object, targetType As Type, parameter As Object,
culture As CultureInfo) As Object Implements IValueConverter.Convert
        Debug.WriteLine("Aktuelles Währungssymbol: " &
culture.NumberFormat.CurrencySymbol)
        Return value
    End Function

    Public Function ConvertBack(value As Object, targetType As Type, parameter As
Object, culture As CultureInfo) As Object Implements IValueConverter.ConvertBack
        Throw New NotImplementedException()
    End Function
End Class
```

Wir lassen uns hier schlicht das aktuelle Währungssymbol in die Konsole ausgeben um anschließend einfach den value so wie dieser ist wieder zurückzugeben. Der Konverter gibt folgenden Text in die Konsole aus:

Aktuelles Währungssymbol: €

Wir sehen also dass hier auch die Systemkultur mit übergeben wird und könnten nun darauf reagieren um Beispielsweise Umwandlungen zu vollziehen.

Fazit: Die WPF nimmt per Default immer en-US als Culture heran. Dies muss vom Entwickler überschrieben werden da sonst immer und überall das amerikanische Format verwendet wird. Dies ist wie wir gesehen haben auch sehr wichtig da es ansonsten zu sehr unangenehmen Nebeneffekten kommen kann welche oft gerne mal länger unentdeckt bleiben.

Hier geht’s zum Video: https://youtu.be/UxshOJqC6B0

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/

2.1.5

Dependency Properties

Jeder der mit der WPF Arbeitet hört früher oder später von Dependency Properties. Sie sind Voraussetzung für Styles, Trigger oder Animationen. Jeder verwendet Dependency Properties da jedes Control diverse Eigenschaften besitzt welche als Dependency Properties implementiert sind. Bei den BuildIn Controls sind dies fast alle Properties.

2.1.5.1

Was sind Dependency Properties und wie unterscheiden sie sich von normalen Properties

Generell wird bei Dependency Properties zwischen zwei Arten unterschieden:

- *Dependency Properties, die durch klassische .Net Properties gekapselt werden. Diese werden auf einfach Dependency Properties genannt.*
- *Dependency Properties, welche durch zwei statische Methoden gekapselt werden. Dabei wird die Dependency Property auf Objekte anderer Klassen gesetzt. Diese werden auch als **Attached Properties** genannt.*

Wir werden klären wann und warum eine Dependency Property implementiert werden sollte bevor wir dann eigene Dependency Properties erstellen und Anwenden. Ich werde auch auf die Metadaten, die Validierung und Logik dahinter eingehen.

DependencyObject und DependencyProperty

Die Klasse DependencyProperty definiert im Grunde nur den Schlüssel zu einem Wert einer Dependency Property, später dazu mehr.

DependencyObject definiert zum Setzen oder Abfragen einer Dependency Property die Methoden GetValue und SetValue.

Eine DependencyProperty-Instanz stellt im Grunde lediglich den Schlüssel zum eigentlichen Wert dar. Vereinfacht gesehen speichert eine **DependencyObject**-Instanz die Werte von DependencyProperties intern in einer Art IDictionary-Instanz unter dem entsprechenden Schlüssel ab.

Da die Klasse DependencyObject für das Speichern und Verwalten von DependencyProperties zuständig ist können auch nur Objekte welche von DependencyObject ableiten Werte einer Dependency Property speichern.

Der Wert einer DependencyProperty ist von vielen Faktoren abhängig, daher der Name *Dependency Property* (Dependency = Abhängigkeit).
Der Wert kann durch ein DataBinding oder eine Animation geändert werden. Die Dependency Property kann einen Default-Wert haben und sogar durch ein im Logical Tree (der Elementebaum in der WPF) höher liegendem Element oder weiteren *Metadaten* beeinflusst werden welche Beispielsweise auch einen Layoutprozess auslösen können.

All diese Dinge müssen bedacht und auch Priorisiert werden. Z.b. muss der Wert einer Animation immer Vorrang vor einem lokal gesetzten Wert haben. Diese Logik übernimmt uns die *Property Engine*. Die Engine ist im Grunde etwas Code innerhalb von DependencyObject welcher uns diese Arbeit abnimmt wodurch einiges in der WPF gerade in Verbindung mit Binding „wie von Geisterhand“ wirkt.

Wie bereits angesprochen sind fast alle Properties von Elementen der WPF als Dependency Properties implementiert.

Auch in eigenen Klassen können wir Dependency Properties implementieren, der größte Vorteil ist das Properties welche als Dependency Properties implementiert sind bereits über einen internen Benachrichtigungsmechanismus verfügen und somit kein INotifyPropertieChanged implementiert werden und das Event OnPropertieChanged geworfen werden muss.
Dadurch ist eine Dependency Property ideal für DataBinding geeignet.

Neben diesem Vorteil gibt es noch andere Vorteile wie Default-Werte, oder Metadaten welche eben nur mit Dependency Properties zur Verfügung stehen.

Nachstehend eine Tabelle in welche ich diese einfach mal als Services bezeichne:

Service	Beschreibung

Animationen	Die WPF besitzt eine sehr gute Unterstützung von Animationen welche nur von Dependency Properties verwendet werden können.
Metadaten	Eine Dependency Property besitzt MetaDaten über welche sehr viel gesteuert werden kann wie Beispielsweise der Default-Wert.
Expressions	Expressions ermöglichen es das der Wert eines Dependency Property zur Laufzeit dynamisch ermittelt werden kann. Z.b. sind DataBinding und DynamicResources über Expressions implementiert.
Data Binding	DataBinding ist über Expressions implementiert, dabei muss die Ziel-Property ein DependencyProperty sein.
Styles	Styles haben wir bereits kennengelernt. In einem Style lassen sich allerdings nur Properties setzen welche als Dependency Property implementiert sind.
Vererbung	Der Wert einer Dependency Property kann über den Logical Tree vererbt werden was sich über die Metadaten definieren lässt.

2.1.5.2

Wir erstellen und Rendern eine „TextEllipse“ mittels Dependency Properties

Um die Funktionsweise und das Zusammenspiel von Dependency Properties in der Praxis kennenzulernen möchte ich ein Simples Beispiel erstellen welches eine Ellipse rendern soll in welcher sich in der Mitte ein Text befindet. Dies könnte z.b. so aussehen:

Um auch Binding und die Manipulation von Dependency Properties kennenzulernen soll man im XAML für diese Ellipse die Eigenschaft Diameter angeben können, welche den Durchmesser beschreibt. Weiters soll man einen Text innerhalb des XAML angeben können.

Was aber wenn der Text länger wird als der Durchmesser der Ellipse zulässt? In diesem Fall soll der gesetzte Wert des Durchmessers überschrieben werden damit sich die Ellipse automatisch dementsprechend vergrößert. Unabhängig davon was zuvor im XAML als Wert für den Durchmesser angegeben wurde.

Wir legen uns eine neue Klasse mit dem Namen ***TextEllipse*** an.
Diese Klasse erbt von *FrameworkElement* (da wir das Element auch Rendern möchten).

```
Public Class TextElipse
    Inherits FrameworkElement


End Class
```

Hinweis:
-> FrameworkElement erbt von UIElement
-> UIElement erbt von Visual
-> Visual erbt von DependencyObject

Das ist der Grund warum wir hier auch Dependency Properties verwenden können.

Wir wissen also dass wir zwei *DependecyProperty* benötigen.
Eines für den Text und eines für Diameter.

Wir beginnen mit dem Property Text:

```
Public Property Text As String
        Get
            CType(GetValue(TextProperty), String)
        End Get

        Set(ByVal value As String)
            SetValue(TextProperty, value)
        End Set
    End Property

    Public Shared ReadOnly TextProperty As DependencyProperty =
                           DependencyProperty.Register("Text",
                           GetType(String), GetType(TextEllipse))
```

Sehen wir uns dieses Konstrukt mal genauer an. Das eigentliche Property sieht wie ein normales .Net Property mit einem *Getter* und einem *Setter* aus. Ist es auch. Normalerweise haben wir beim einem Property mit einem *Getter* und *Setter* ein sogenanntes *BackingField* welches einen Wert in einer privaten Variable speichert welcher im *Getter* abgerufen und übergeben wird (Return Anweisung) und im *Setter* neu gesetzt wird. Das ist hier nicht der Fall. Wie im vorigen Kapitel bereits angesprochen wird der Wert eines Dependency Property immer mit den Methoden GetValue und SetValue aus der Klasse *DependencyObject* abgerufen bzw. gesetzt.

Das eigentliche Dependency Property befindet sich als *Feld* darunter und wird per Konvention immer mit dem Namen des Property + „Property“ benamst wie im obigen Beispiel gut zu sehen ist, und ist vom Typ DependencyProperty. Dieses ist außerdem immer Shared ReadOnly und muss Public sein. Dies ist quasi der bereits erwähnte „Schlüssel“ für den eigentlichen Wert.

Dieses Feld wird initialisiert indem die Statische Methode Register der Klasse DependencyProperty aufgerufen wird.

Diese Methode verlangt im ersten Konstruktor zumindest den Namen der .Net Eigenschaft (hier „Text“), den Typ der Eigenschaft (hier vom Typ String) und den Typ des Objekts welches die Eigenschaft besitzt. (Hier ist das die Klasse „TextEllipse“).

Tipp: Visual Studio erstellt automatisch ein Grundgerüst eines Dependency Property. Hierfür besitzt Visual Studio ein Snippet welche mit dem Keyword „wpfdp“ aufgerufen werden kann. Tippen Sie im VisualBasic Editor „wpfdp“ ein und drücken 2x TAB wird ein Grundgerüst eingefügt in welchem nur noch die Platzhalter ausgefüllt werden müssen. Näheres seht ihr im Video zu diesem Kapitel.

Es gibt aber weitere Überladungen der Register Methode. Hier die Definitionen:

```
Register(name As String, propertyType As Type, ownerType As Type) As
DependencyProperty

Register(name As String, propertyType As Type, ownerType As Type, typeMetadata As
PropertyMetadata) As DependencyProperty

Register(name As String, propertyType As Type, ownerType As Type, typeMetadata As
PropertyMetadata, validateValueCallback As ValidateValueCallback) As
DependencyProperty
```

Die zweite Überladung nimmt auch sogenannte Property-Metadaten an. Metadaten bestimmen Dinge wie Beispielsweise den Standardwert. Ihnen kann auch ein Delegate für den PropertyChangedCallback übergeben werden. Diese Methode wird dann aufgerufen wenn sich der Wert eines Dependency Property ändert.
Machen wir dies mal Anhand des Beispiels mit dem Text-Property.

```
Public Shared ReadOnly TextProperty As DependencyProperty =
                           DependencyProperty.Register("Text",
                           GetType(String), GetType(TextElipse),
                           New PropertyMetadata("",  AddressOf OnTextChanged))

Private Shared Sub OnTextChanged(ByVal d As DependencyObject, ByVal e As DependencyP-
ropertyChangedEventArgs)

    End Sub
```

Wir übergeben hier an die zweite Überladung von Register eine neue Instanz von PropertyMetadata welche in der vierten Überladung als ersten Parameter einen Default-Value erwartet welchen wir auf einen Leerstring festlegen. Als zweiten Parameter erwartet die

Überladung einen PropertyChangedCallback.
Die Methode welche wir hier angeben wird immer aufgerufen wenn sich der Wert des Property ändert.

Jetzt fragt sich der ein oder andere warum man denn nicht einfach die Änderung des Wertes im Setter des Property abfängt. Aus dem Grund das dies von „außen“ umgangen werden könnte da man den Wert nicht nur über das Property selbst sondern auch mit der Methode SetValue aus der Klasse *DependencyObject* überschreiben könnte. Dies würde den Setter „umgehen“.

Nun Fehlt noch das zweite Dependency Property:

```
Public Property Diameter As Double
        Get
            Return CDbl(GetValue(DiameterProperty))
        End Get

        Set(ByVal value As Double)
            SetValue(DiameterProperty, value)
        End Set
    End Property

    Public Shared ReadOnly DiameterProperty As DependencyProperty =
                           DependencyProperty.Register("Diameter",
                           GetType(Double), GetType(TextElipse),
                           New PropertyMetadata(Double.Parse("40"))
```

Um nun das Objekt zu Rendern überschreiben wir die Methode OnRender der Basisklasse:

```
Protected Overrides Sub OnRender(drawingContext As DrawingContext)
        MyBase.OnRender(drawingContext)

        drawingContext.DrawEllipse(New SolidColorBrush(Colors.Black), New Pen(New
SolidColorBrush(Colors.Yellow), 2), New Point(Diameter / 2, Diameter / 2), Diameter /
2, Diameter / 2)
        Dim formatted = GetFormattedText()
        drawingContext.DrawText(formatted, New Point((Diameter - formatted.Width) / 2,
(Diameter - formatted.Height) / 2))

    End Sub

    Public Function GetFormattedText() As FormattedText
        Return New FormattedText(Text, CultureInfo.InvariantCulture,
FlowDirection.LeftToRight, New Typeface("Arial"), 20, New
SolidColorBrush(Colors.Green), 1.25)
    End Function
```

Der Code inkl. der Hilfsmethode GetFormattedText ist im Moment nicht wichtig, wichtig ist das ihr wisst das man OnRender überschreiben muss um selbst ein *FrameworkElement* Rendern zu können.

In unserem Projekt binden wir nun unsere TextEllipse in unserem MainWindow.xaml.
Haben wir das Projekt nun einmahl erfolgreich kompiliert können wir unsere TextEllipse ganz

normal im XAML Editor verwenden und haben sogar intellisense für unsere selbst erstellen DependecyProperties.

```
<local:TextElipse x:Name="MyElipse" Grid.Column="0" Text="Hier steht Text"
Diameter="150"/>
```

Starten wir das Projekt nun sehen wir genau das was wir erwarten würden. Aber auch bereits zur Designzeit haben wir das Ergebnis gesehen.

Das Ergebnis sieht genauso aus wie erwartet. Fügen wir nun noch einen *Slider* und eine *TextBox* hinzu um die Größe und den Text der Ellipse ändern zu können.

```
<Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="*"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>
        <local:TextElipse x:Name="MyElipse" Grid.Column="0" Text="Hier steht Text"
Diameter="150"/>
        <StackPanel Grid.Column="1">
            <Slider Minimum="1" Maximum="500" Value="{Binding
ElementName=MyElipse,Path=Diameter,Mode=TwoWay}"/>
            <TextBox Text="{Binding
       ElementName=MyElipse,Path=Text,Mode=TwoWay,UpdateSourceTrigger=PropertyChanged}/
       >
            </StackPanel>
    </Grid>
```

Starten wir nun erneut das Projekt werden wir merken das wir in die TextBox schreiben können was wir wollen oder des Slider hin und her bewegen können – es wird sich nichts an unserer Ellipse ändern. Weder verändert sich die Größe noch der Text. Aber warum?
Das liegt daran das wir unser „Control“ selbst rendern. Wir müssen der WPF nun auch noch bekanntgeben dass sie doch bitte neu Rendern soll wenn wir ein Property ändern.

Hierfür übergeben wir statt PropertyMetadata diesmal FrameworkPropertyMetadata als letzten Parameter der RegisterMethode. Die Klasse FrameworkPropertyMetadata besitzt im

Gegensatz zu PropertyMetadata die interne Eigenschaft Flags welche in insgesamt vier Überladungen mit übergeben werden kann.

```
Public Property Diameter As Double
        Get
            Return CDbl(GetValue(DiameterProperty))
        End Get

        Set(ByVal value As Double)
            SetValue(DiameterProperty, value)
        End Set
    End Property

    Public Shared ReadOnly DiameterProperty As DependencyProperty =
                           DependencyProperty.Register("Diameter",
                           GetType(Double), GetType(TextElipse),
                           New FrameworkPropertyMetadata(Double.Parse("40"),
FrameworkPropertyMetadataOptions.AffectsRender))



    Public Property Text As String
        Get
            Return CType(GetValue(TextProperty), String)
        End Get

        Set(ByVal value As String)
            SetValue(TextProperty, value)
        End Set
    End Property

    Public Shared ReadOnly TextProperty As DependencyProperty =
                           DependencyProperty.Register("Text",
                           GetType(String), GetType(TextElipse),
                           New FrameworkPropertyMetadata("",
FrameworkPropertyMetadataOptions.AffectsRender, AddressOf OnTextChanged))
```

Wir teilen somit der WPF mit das neu „gerendert“ werden muss wenn sich der Wert dieses Property`s ändert. Die WPF wird in diesem Fall bei jeder Wertänderung die OnRender Methode aufrufen.

Als kleine „Fleisaufgabe“ werden wir noch sicherstellen das der eingegebene Text immer in die Ellipse passt indem wir die Textgröße berechnen und den Durchmesser der Ellipse in dem Fall das der Text breite sein sollte als der Durchmesser der Ellipse, diese vergrößern.

```
Private Shared Sub OnTextChanged(ByVal d As DependencyObject, ByVal e As
DependencyPropertyChangedEventArgs)
        Dim te As TextElipse = DirectCast(d, TextElipse)
        If te.GetFormattedText().Width >
Double.Parse(CType(te.GetValue(DiameterProperty), String)) Then
            te.SetValue(DiameterProperty, te.GetFormattedText().Width + 5)
        End If
    End Sub
```

Wir haben also nun gelernt das man Dependency Properties immer mit GetValue abfragen und mittels SetValue neu setzten soll. Dependecy Properties können über Ihre Metadaten einiges an Logik mitbekommen und können direkt gebunden werden. Im nächsten Kapitel werden wir ein Dependency Property in ein UserControl einbauen um hier von „außen“ das Verhalten dieses Control steuern zu können.

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

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/

2.1.5.3

Eigene DependencyProperties in einem UserControl implementieren

In diesem Kapitel kommen wir dazu dass wir ein Dependency Property in ein UserControl implementieren. Dies ist ein Scenario welches in der Praxis schon um einiges öfter vorkommt.

In diesem Beispiel gehen wir davon aus das wir ein UserControl haben welches uns Personendaten darstellen soll. Das Control soll aber so flexibel wie möglich sein um es möglichst wiederverwenden zu können.

Weiters sollen wir die Möglichkeit erhalten das wir die TextBoxen in diesem UserControl auf ReadOnly setzen. Und zwar von außen und via Binding. Klingt jetzt etwas kompliziert aber ihr werdet sehen dass es das gar nicht ist.

Wir erstellen ein neuen UserControl und nennen dieses ShowOrEditPersonControl und gestalten es mit zwei Labels und zwei TextBoxen.

Sieht unspektakulär aus, ist es auch. Zusätzlich, damit es ein wenig klarer wird ob die TextBox nun ReadOnly ist oder nicht habe ich einen Style hinzugefügt welcher bewirkt das der Background einer TextBox Grau wird wenn diese ReadOnly ist.

```
<UserControl x:Class="ShowOrEditPersonControl"
             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:DependencyProperties_Demo"
             mc:Ignorable="d"
             d:DesignHeight="72.541" d:DesignWidth="460.656">
    <UserControl.Resources>
        <Style TargetType="{x:Type TextBox}">
            <Setter Property="Background" Value="White"/>
            <Setter Property="Margin" Value="3"/>
            <Style.Triggers>
                <Trigger Property="IsReadOnly" Value="True">
                    <Setter Property="Background" Value="LightGray"/>
                </Trigger>
            </Style.Triggers>
        </Style>
    </UserControl.Resources>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto"/>
            <RowDefinition Height="Auto"/>
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="Auto"/>
            <ColumnDefinition Width="*"/>
        </Grid.ColumnDefinitions>

        <Label Content="Vorname"/>
        <Label Content="Nachname" Grid.Row="1"/>

        <TextBox Text="Sascha" Grid.Column="1"/>
        <TextBox Text="Patschka" Grid.Column="1" Grid.Row="1"/>

    </Grid>
</UserControl>
```

Im Moment haben wir den Vornamen sowie den Nachnamen noch fix gesetzt. Wie möchten aber dass dieser gebunden wird. Außen z.b. im MainWindow können wir dieses UserControl nun problemlos wie jedes andere UserControl im XAML verwenden.

```
<local:ShowOrEditPersonControl/>
```

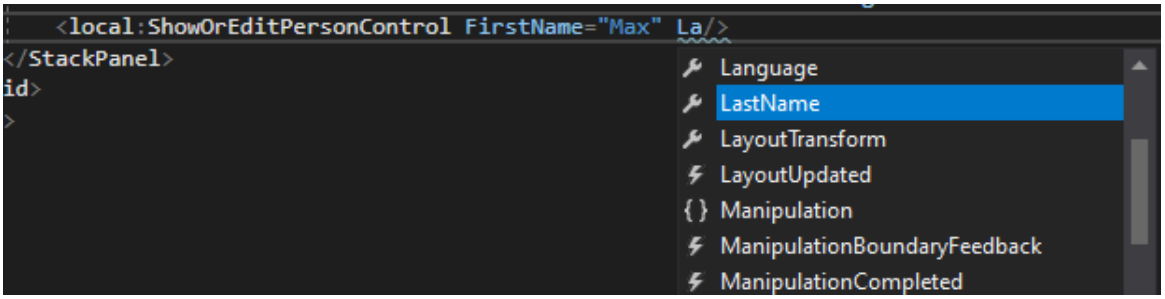
Um nun direkt vom MainWindow aus den Text für den Vornamen und den Nachnamen zu setzen, benötigen wie zwei *DependencyProperties* in der CodeBehind des UserControls. Eines für den Vornamen und eines für den Nachnamen.

Unsere *ShowOrEditPersonControl.vb* sieht nun wie folgt aus:

```
Public Class ShowOrEditPersonControl

    Public Property FirstName As String
        Get
            Return CType(GetValue(FirstNameProperty), String)
        End Get

        Set(ByVal value As String)
            SetValue(FirstNameProperty, value)
        End Set
    End Property

    Public Shared ReadOnly FirstNameProperty As DependencyProperty =
                           DependencyProperty.Register("FirstName",
                           GetType(String), GetType(ShowOrEditPersonControl),
                           New PropertyMetadata(Nothing))

    Public Property LastName As String
        Get
            Return CType(GetValue(LastNameProperty), String)
        End Get

        Set(ByVal value As String)
            SetValue(LastNameProperty, value)
        End Set
    End Property

    Public Shared ReadOnly LastNameProperty As DependencyProperty =
                           DependencyProperty.Register("LastName",
                           GetType(String), GetType(ShowOrEditPersonControl),
                           New PropertyMetadata(Nothing))

End Class
```

Zwei völlig normale aus dem „Default-Gerüst“ bestehende Dependency Property`s.
Mehr benötigt es auch schon nicht mehr.

Wir kompilieren das Projekt und sehen uns unser UserControl mal vom MainWindow aus an.
Wir merken dass wir nun bereits die Properties zur Verfügung haben:



Die Abbildung zeigt dass wir nun bereits Intellisenseunterstützung haben. Wir können die Werte bereits angeben und könnten nun auch darauf Binden. Setzen wir mal den Vornamen auf „Max“ und den Nachnamen auf „Mustermann“.

```
<local:ShowOrEditPersonControl FirstName="Max" LastName="Mustermann"/>
```

Aber wie kommen die Daten welche ich hier angebe oder Binde in die TextBox? Berechtigte Frage, denn wer versucht hier Werte anzugeben wird schnell merken das man dies zwar machen kann, es sich aber an der TextBox nichts ändert. Das liegt daran das wir die TextBoxen selbst auch noch Binden müssen.

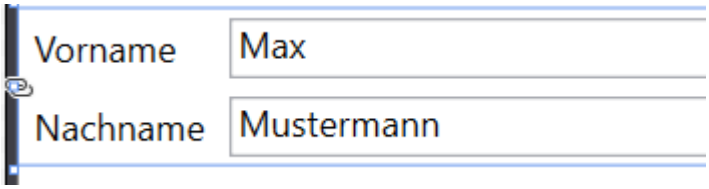
Hierfür gibt es mehrere Wege, ich gehe auf einen ein bei welchem ich der Meinung bin das es der „sicherste“ ist.

Wir ändern also der XAML im ShowOrEditPersonControl wie folgt:

```
<TextBox Text="{Binding FirstName, RelativeSource={RelativeSource FindAncestor,
AncestorType={x:Type local:ShowOrEditPersonControl}}}" Grid.Column="1"/>

<TextBox Text="{Binding LastName, RelativeSource={RelativeSource FindAncestor,
AncestorType={x:Type local:ShowOrEditPersonControl}}}"
                 Grid.Column="1" Grid.Row="1"/>
```

Das Binding in diesem Codeblock sucht im ElementTree der WPF nach oben, das nächste Element vom Typ „*ShowOrEditPersonControl*“. Wird es fündig verwendet es in diesem das Property „*LastName* bzw. „*FirstName*“.

Werfen wir nun (nach einem nochmaligem Buildvorgang) nochmals einen Blick in das MainWindow sehen wir bereits ein Bild welches wir schon eher erwarten würden.



Toll, das funktioniert und wir können von außen nun Binden und steuern was in den TextBoxen für Inhalt stehen soll.

Da war ja noch was? Genau, wir wollten das wir steuern können ob sich das UserControl im Editiermodus befindet oder nicht. Fügen wir uns erstmal im MainWindow eine Checkbox ein welche der Editiermodus ein bzw. ausschalten soll.

```
<StackPanel>
    <CheckBox x:Name="chkEditMode" Content="Im Editiermodus?" Margin="10"/>
    <local:ShowOrEditPersonControl FirstName="Max" LastName="Mustermann"/>
</StackPanel>
```

Nun benötigen wir ein weiteres Dependency Property innerhalb unseres UserControls:

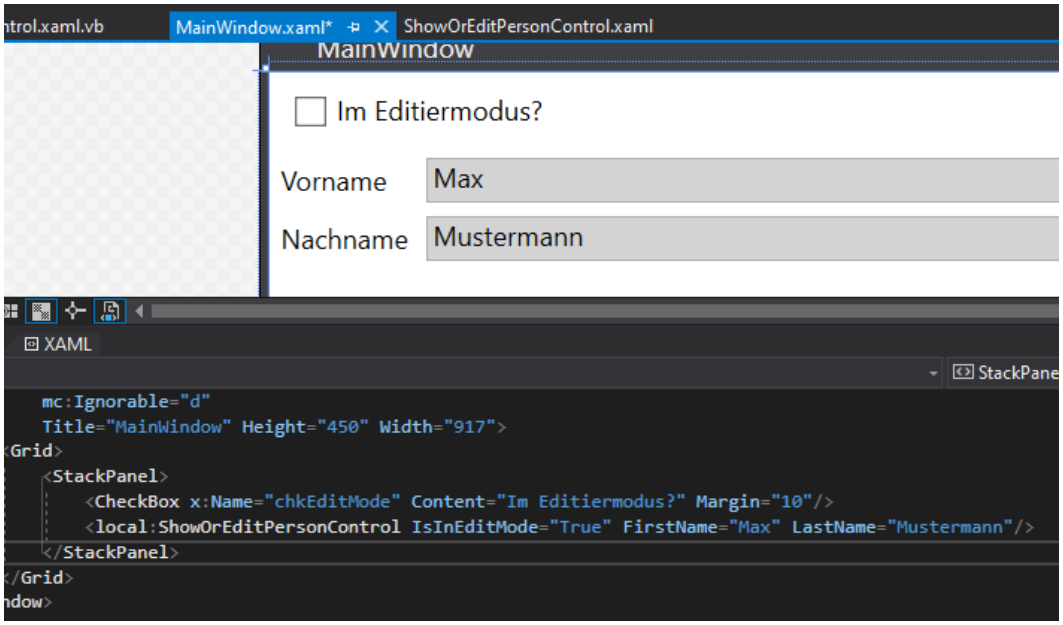
```
Public Property IsInEditMode As Boolean
        Get
            Return CBool(GetValue(IsInEditModeProperty))
        End Get

        Set(ByVal value As Boolean)
            SetValue(IsInEditModeProperty, value)
        End Set
    End Property

    Public Shared ReadOnly IsInEditModeProperty As DependencyProperty =
                           DependencyProperty.Register("IsInEditMode",
                           GetType(Boolean), GetType(ShowOrEditPersonControl),
                           New PropertyMetadata(False))
```

Diesem Property übergeben wir über die Metadaten False als Default-Wert.

Die Bindung auf die TextBoxen kann nun genauso wie beim Text-Property erfolgen oder – da wir einen Style definiert haben gleich in diesem damit wir uns nicht wiederholen müssen.

```
<Style TargetType="{x:Type TextBox}">
     <Setter Property="Background" Value="White"/>
     <Setter Property="Margin" Value="3"/>
     <Setter Property="IsReadOnly" Value="{Binding IsInEditMode,
RelativeSource={RelativeSource FindAncestor, AncestorType={x:Type
local:ShowOrEditPersonControl}}}"/>
     <Style.Triggers>
         <Trigger Property="IsReadOnly" Value="True">
             <Setter Property="Background" Value="LightGray"/>
         </Trigger>
     </Style.Triggers>
</Style>
```

Wenn wir dies nun ausprobieren indem wir im *MainWindow* wieder das Property setzen werden wir sehen dass sich unser UserControl nicht ganz so verhält wie wir das wollen. Nämlich genau anders rum.



Wir haben im **Kapitel 2.1.1.2** einiges über Trigger gelöst und könnten hiermit das verhalten umkehren. Die schönere Lösung wäre allerdings ein Converter wie in **Kapitel 2.1.4.4** gelernt. Schon aus dem Grund da dieser wiederverwendbar ist, wobei ich Trigger immer wieder neu runter tippen muss.

Wir legen uns also einen Converter an und nennen die Klasse BooleanReverseConverter.

```
Public Class BooleanReverseConverter
    Implements IValueConverter

    Public Function Convert(value As Object, targetType As Type, parameter As Object,
culture As CultureInfo) As Object Implements IValueConverter.Convert
        Return Not Boolean.Parse(value.ToString())
    End Function

    Public Function ConvertBack(value As Object, targetType As Type, parameter As
Object, culture As CultureInfo) As Object Implements IValueConverter.ConvertBack
        Return Not Boolean.Parse(value.ToString())
    End Function
End Class
```

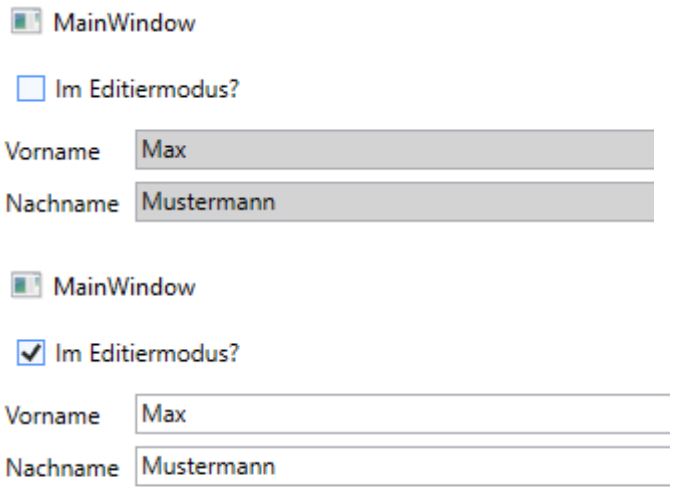
Nun müssen wir diesen Converter im unseren XAML einbinden. Am einfachsten geht dies über das Eigenschaftenfenster, aber auch direkt im Code indem wir den Converter als *Resource* hinzufügen und einen Key vergeben. Diesen Key verwenden wir anschließend im Binding:

```
<UserControl.Resources>
    <local:BooleanReverseConverter x:Key="BooleanReverseConverter"/>
    <Style TargetType="{x:Type TextBox}">
        <Setter Property="Background" Value="White"/>
        <Setter Property="Margin" Value="3"/>
        <Setter Property="IsReadOnly" Value="{Binding IsInEditMode,
            Converter={StaticResource BooleanReverseConverter},
            RelativeSource={RelativeSource FindAncestor,
            AncestorType={x:Type local:ShowOrEditPersonControl}}}"/>
        <Style.Triggers>
            <Trigger Property="IsReadOnly" Value="True">
                <Setter Property="Background" Value="LightGray"/>
            </Trigger>
        </Style.Triggers>
    </Style>
</UserControl.Resources>
```

Nach einem Buildvorgang zurück im *MainWindow* sehen wir das nun alles so funktioniert wie erwünscht. Nun können wir noch die Checkbox auf unser DependencyProperty binden:

```
<StackPanel>
    <CheckBox x:Name="chkEditMode" Content="Im Editiermodus?" Margin="10"/>
    <local:ShowOrEditPersonControl IsInEditMode="{Binding
ElementName=chkEditMode,Path=IsChecked}" FirstName="Max" LastName="Mustermann"/>
</StackPanel>
```

Hier nun zwei Abbildungen die zeigen wie unser Binding zur Laufzeit funktioniert:



Fazit: Dependency Properties sind eine feine Sache wenn man diese Versteht und weis was im Hintergrund abläuft. Auf Dependency Properties kann gebunden werden, können einen Standardwert haben und eine gewisse Logik insich implementiert haben.

Hier geht's zum Video: https://youtu.be/30Lf64dscS0

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/

2.1.6

Markuperweiterungen (Markup Extensions)

Markup Extensions nutzen wir jeden Tag in der WPF. Bekannte Markup Extensions sind Beispielsweise {StaticResource}, {TemplateBinding} oder {Binding}.

Wir verwenden diese ständig und sind ein fester Bestandteil der WPF, aber wie sieht es aus mit eigenen Extensions? Können wir eigene Extensions schreiben und was für einen Mehrwert bringen uns diese Extensions? Dies will ich in diesem Kapitel erläutern.

2.1.6.1

Markuperweiterungen - Theorie

Ich habe ja bereits einige Markup Extensions erwähnt. Aber es gibt noch einige weitere im XAML Namespace. x:Name, x:Static, x:Null sind einige welche wir immer wieder verwenden. Ich will auch nicht zu sehr ins Detail der Theorie gehen da dieses Thema nicht allzu oft Verwendung finden wird. Ich finde allerdings das die Markup Extension eine nette Möglichkeit ist wieder etwas Code, welcher immer und immer wieder benötigt wird auszulagern.

Eigene Markup Extension können wir implementieren indem wir entweder von der Basisklasse MarkupExtension erben oder das Interface IMarkupExtension in eine Klasse implementieren.

Die Basisklasse *MarkupExtension* besitzt eine Methode ProvideValue(System.IServiceProvider) As Object) welche als *MustOverride* deklariert ist. Diese Methode gilt es zu überschreiben und muss ein Ergebnis zurückliefern.

2.1.6.2

Markuperweiterungen - Beispiel

Als ersten möchten wir anhand eines einfachen Beispiels einfach nur ein „Hallo Welt“ zurückgeben, nur damit wir sehen wie die Implementierung einer Markup Extension von statten geht und was dabei evtl. zu beachten ist.

Unsere Markup Extension soll den Namen *HelloWorld* bekommen. Die Namenskonvention erwartet von uns dass die Klasse somit den Namen „*HelloWorldExtension*“ bekommt. Diese Namenskonvention ist keine Pflicht, wird aber empfohlen. Wir legen also eine neue Klasse mit diesem Namen an und erben von MarkupExtension. Tun wir dies generiert uns Visual Studio automatisch die Methode ProvideValue da diese in der Basisklasse ja mit *MustOverride* deklariert ist. Außerdem empfehle ich immer solche Extensions genauso wie z.b. Konverter in einen eigenen Namespace zu packen:

```
Namespace MyWpfExtensions
    Public Class HelloWorldExtension
        Inherits Markup.MarkupExtension


        Public Overrides Function ProvideValue(serviceProvider As IServiceProvider) As
Object
            Throw New NotImplementedException()
        End Function
    End Class
End Namespace
```

Dies ist also ein Grundkonstrukt einer Markup Extension. In unserem Beispiel möchten wir einfach das unsere Extension schlicht ein „*Hallo Welt*“ zurückgibt. Also machen wir dies:

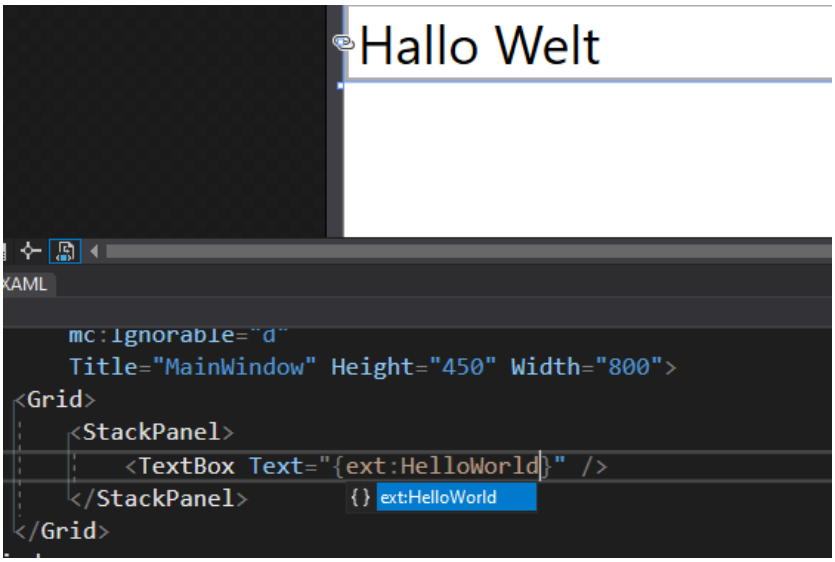
```
Namespace MyWpfExtensions
    Public Class HelloWorldExtension
        Inherits Markup.MarkupExtension


        Public Overrides Function ProvideValue(serviceProvider As IServiceProvider) As
Object
            Return "Hallo Welt"
        End Function
    End Class
End Namespace
```

Jetzt geht es darum diese Extension in unseren XAML einzubinden. Hierfür muss das Projekt erstmal kompiliert werden damit auch der Designer etwas von unserer neuen Klasse mitbekommt. Anschließend können wir erstmal den Namespace importieren:

```
xmlns:ext="clr-namespace:_2_1_6_1_MarkupExtensions.MyWpfExtensions"
```

Info: Mein Projekt hat den Namen und somit den Stammnamespace „2_1_6_1_MarkupExtensions“. Dem folgt einfach unser Namespace für die Extensions.

Nun können wir unsere Extension auch schon in unserem XAML nutzen:



OK, zugegeben. So etwas hat nun im Moment wenig nutzen.
Wie sieht es mit Parametern aus? Ja, machen wir ein Beispiel welches einfach zwei Ziffern Multiplizieren soll.

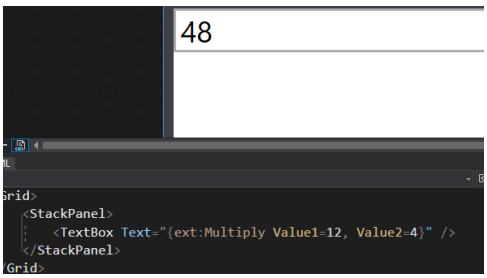
Wieder legen wir eine neue Klasse an welche von *MarkupExtension* erbt und nun aber zusätzlich zwei Eigenschaften besitzen soll. Hierfür implementieren wir neben dem paramterlosem Konstruktor (welcher Pflicht ist) einen weiteren welcher zwei Ziffern entgegennehmen kann:

```
Imports System.Windows.Markup

Namespace MyWpfExtensions
    Public Class MultiplyExtension
        Inherits MarkupExtension

        Public Sub New()
        End Sub

        Public Sub New(value1 As String, value2 As String)
            Me.Value1 = value1 : Me.Value2 = value2
        End Sub

        <ConstructorArgument("value1")>
        Public Property Value1 As Object
        <ConstructorArgument("value2")>
        Public Property Value2 As Object

        Public Overrides Function ProvideValue(serviceProvider As IServiceProvider) As
Object
            Throw New NotImplementedException()
        End Function
    End Class
End Namespace
```

Die Annotation der Eigenschaften ist übrigens nicht Pflicht wird aber empfohlen.
Nun haben wir einen zweiten Konstruktor um müssen nun die Werte nur noch Multiplizieren:

```
Public Overrides Function ProvideValue(serviceProvider As IServiceProvider) As Object
     If Value1 Is Nothing Then Value1 = 0
     If Value2 Is Nothing Then Value2 = 0

     Return Double.Parse(Value1.ToString) * Double.Parse(Value2.ToString)
End Function
```

Wenn wir nun unsere Extension in unserer TextBox anwenden sehen wir auch dass diese nun korrekt zwei Ziffern addieren kann.



Bis jetzt sind wir aber noch nicht Typsicher unterwegs, das soll sich nun aber ändern.
Die zu überschreibende Methode ProvideValue gibt uns ein IServiceProvider mit auf dem Weg.

Über diesen Serviceprovider kommen wir an die Information an welches Property das Ergebnis geht und somit auch welchen Datentyp dieses Property besitzt. Somit können wir auch versuchen eine Konvertierung durchzuführen.

Erstellen wir uns eine Klasse mit dem Namen MultiplyTypesaveExtension.
Unsere Konstruktoren und die Eigenschaften Value1 und Value2 ändern sich insofern das diese nun nur noch Double als Parameter erwarten. Die zu überschreibende Methode ProvideValue ändern wir wie folgt ab:

```
Public Overrides Function ProvideValue(serviceProvider As IServiceProvider) As Object
   If serviceProvider Is Nothing Then Return Nothing
   Dim valueTarget =
DirectCast(serviceProvider.GetService(GetType(IProvideValueTarget)),
IProvideValueTarget)

   Dim calcResult = Value1 * Value2

   If valueTarget.TargetProperty.GetType Is GetType(DependencyProperty) Then
       Dim dp = DirectCast(valueTarget.TargetProperty, DependencyProperty)
       Return Convert.ChangeType(calcResult, dp.PropertyType)
   Else
       Dim info = DirectCast(valueTarget.TargetProperty, PropertyInfo)
       Return Convert.ChangeType(calcResult, info.PropertyType)
    End If

End Function
```

Im ersten Schritt holen wir uns das IProvideValueTarget-Service. Dieses gibt uns Informationen über die Eigenschaft zurück welche wir benötigen. Nachdem wir das Ergebnis wie bereits gehabt errechnen müssen wir prüfen ob es sich bei der Eigenschaft um ein Dependency Property oder ein normales Property handelt. Den unterschied haben wir ja bereits in Kapitel 2.1.5 gelernt.

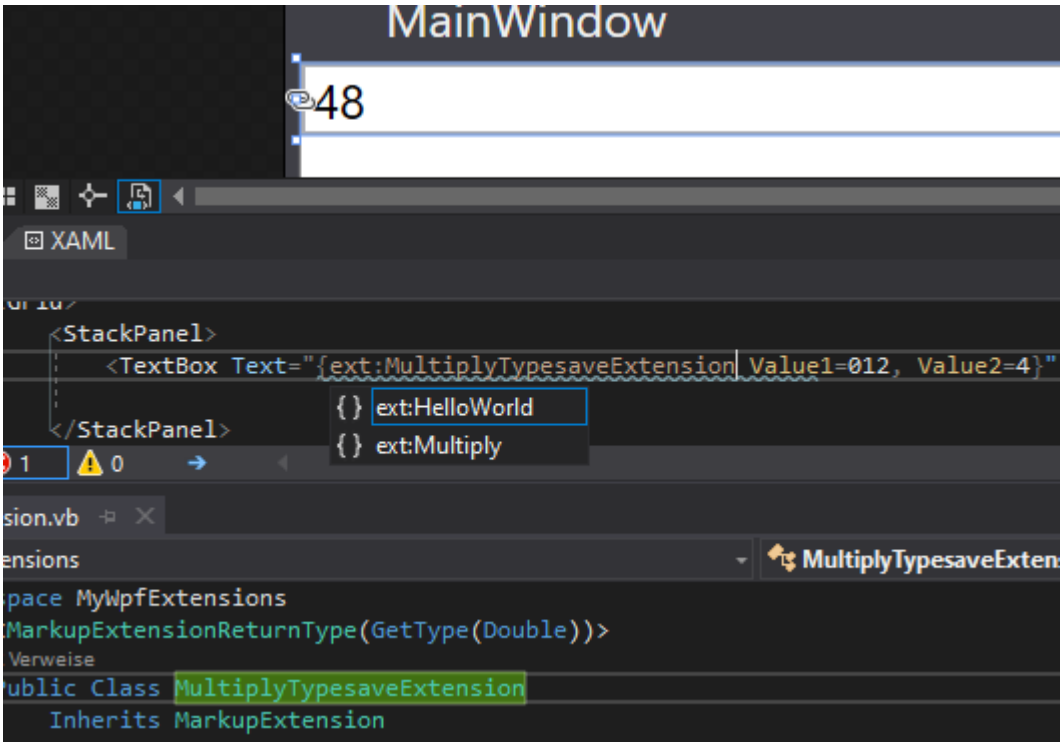
Bewaffnet mit unseren Informationen können wir nun in den richtigen Typ Konvertieren.

Zu guter letzt bringen wir auf Höhe der Klasse selbst noch eine Annotation an. Diese in nicht Plicht aber ich empfehle diese, warum zeige ich gleich noch.

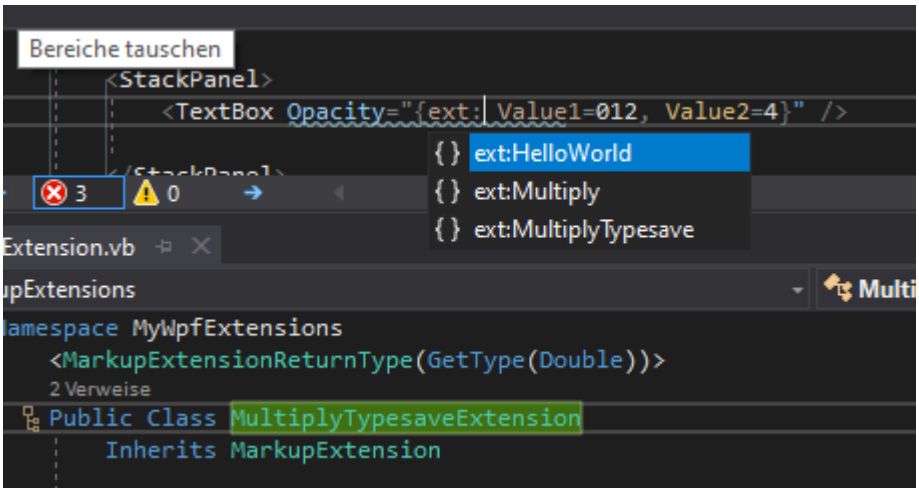
```
<MarkupExtensionReturnType(GetType(Object))>
    Public Class MultiplyTypesaveExtension

...

...
```

Ist er nun komplett verrückt? Da arbeiten wir nun Typsicher und dann eine Annotation mit Object als Datentyp? OK, das muss ich erklären. Diese Annotation gibt dem Designer der WPF die Möglichkeit zu wissen für was für einen Datentyp diese Extension gültig sein soll.

Machen wir ein Beispiel. Gebe ich in der Annotation als Datentyp Double an kann ich in einem TextBox.Text Property zwar die Extension nutzen, habe aber keine Intellisens.



Dafür aber Beispielsweise bei der Eigenschaft Opacity der TextBox, da diese von Typ Double ist.



Gebe ich in der Annotation aber Object an, habe ich die Intellisens überall zur Verfügung. Da wir die Konvertierung im Code der Extension korrekt durchführen ist dies auch kein Problem.

Ich hoffe das Kapitel hat euch nun einmal mehr von der Flexibilität der WPF überzeugt, ich bin mir sicher das bei euch im Kopf gerade ein paar Ideen für den Einsatz von MarkupExtensions herumschwirren. ;-)

Ein Typisches Beispiel wäre die Lokalisierung (Übersetzung in die aktuelle Sprache des Benutzers).

Hier geht's zum Video: https://youtu.be/9wrBFbswtGg

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/

2.1.7

Attached Properties

Jetzt wo wir über Dependency Properties Bescheid wissen kommen wir zu den Attached Properties. Diese sind eine andere Art von Dependency Properties. Wir benutzen Attached Properties im Grunde täglich während der Arbeit unter WPF. Viele Panels haben Attached Properties wie z.b. das DockPanel. Auch das Grid besitzt Attached Properties.

Haben wir ein Grid in welchen sich Steuerelemente befinden können wir bestimmen wo diese Steuerelemente befinden sollen (Spalte oder Zeile) indem wir Attached Properties nutzen.

```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition Height="1*"/>
        <RowDefinition Height="3*"/>
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*"/>
        <ColumnDefinition Width="*"/>
    </Grid.ColumnDefinitions>
    <Button Content="Test" Grid.Column="1" Grid.Row="0"/>
</Grid>
```

Grid.Column ist ein Attached Property welches für Dependency Object implementiert wurde. Das bedeutet das jedes Steuerelement der WPF (wir haben ja gelernt das in der WPF jedes Steuerelement indirekt von DependencyObject ableitet) dieses Attached Property nutzen kann.

So können wir auf einfache Art und Weise bestimmen wo genau der Button reingepackt wird.

Info:

Der oben stehenden XAML würde wie folgt in Code aussehen:

```
Dim btn = TestButton
btn.SetValue(Grid.ColumnProperty, 1)
btn.SetValue(Grid.RowProperty, 0)
'oder
Grid.SetColumn(btn, 1)
Grid.SetRow(btn, 0)
```

2.1.7.1

Attached Properties – Kurze Theorie

Dank den Attached Properties lassen sich Klassen elegant und einfach um beliebige Dependency Properties erweitern ohne dass dazu eine Subklasse erzeugt werden muss.

Um nun Beispielsweise einem Button ein weiteres Property zu spendieren müssen wir also nicht wie „früher“ eine Klasse erstellen welche von Button ableitet um ein weiteres Property und gegebenenfalls etwas Logik in diesen zu implementieren. Wir müssen uns also keinen neuen Button erstellen sondern können den vorhandenen verwenden so wie dieser ist.

In der deutschen MSDN wird ein Attached Property als *angefügte Eigenschaft* Bezeichnet.
Ich würde jedoch „*hinzugefügte Eigenschaft*“ sagen.

Um ein Attached Property zu erstellen muss es eine gewisse Implementierung vorweisen.
Ein Attached Property enthält wie auch ein Dependency Property ein öffentlich statisches Feld vom Typ Dependency Property. Damit XAML das Property als Attached Property erkennt müssen allerdings noch zwei Statische Methode Get und Set existieren.

TIPP: Visual Studio stellt auch für ein Attached Property ein CodeSnipped zur Verfügung.
Tippe hierfür „wpfdpa“ ein und drücke zweimal TAB.

2.1.7.2

Eigene Attached Properties

Kommen wir zur Praxis
So würde ein Attached Property aussehen:

```
Public Class MyFirstAttachedProperty
    Inherits DependencyObject


    Public Shared Function GetMyFirstProp(ByVal element As DependencyObject) As String
        If element Is Nothing Then
            Throw New ArgumentNullException("element")
        End If

        Return CType(element.GetValue(MyFirstPropProperty), String)
    End Function

    Public Shared Sub SetMyFirstProp(ByVal element As DependencyObject, ByVal value As
String)
        If element Is Nothing Then
            Throw New ArgumentNullException("element")
        End If

        element.SetValue(MyFirstPropProperty, value)
    End Sub

    Public Shared ReadOnly MyFirstPropProperty As _
                           DependencyProperty =
DependencyProperty.RegisterAttached("MyFirstProp",
                           GetType(String), GetType(MyFirstAttachedProperty),
                           New PropertyMetadata(Nothing))

End Class
```

Über die Signatur der Methoden Get und Set kann bestimmt werden für welche Controls ein Attached Property gültig sein soll. In diesem Fall wird bei beiden Methoden ein Element vom Typ Dependency Object erwartet, was bedeutet das dieses Attached Property auf jedes Objekt welches vom Typ DependencyObject ableitet angewandt werden kann. Dieses Argument kann auch angepasst werden um dies Beispielsweise nur für einen Button zu erlauben.

Erstellen wir mal ein einfaches Beispiel welches auf alle Controls angewandt werden kann:

Wir wollen zum Testen mal ein Attached Property erstellen welches eine Drehung unterstützt. Dies ist in der WPF zwar mit RenderTransform über XAML für jedes ObjeKt verfügbar aber es benötigt einige Zeilen Code. Um mit Attached Properties warm zu werden möchten wir dies mal einfacher gestalten. Nachstehend der XAML mit welchem man ein Element normalerweise drehen würde:

```
<Button Content="Test" RenderTransformOrigin="0.5,0.5">
     <Button.RenderTransform>
         <TransformGroup>
             <RotateTransform Angle="11"/>
         </TransformGroup>
     </Button.RenderTransform>
 </Button>
```

Dieser XAML Rendert einen Button um 11 Grad verdreht wobei der Mittelpunkt der Drehung sowohl horizontal als auch vertikal in der Mitte des Control ist.

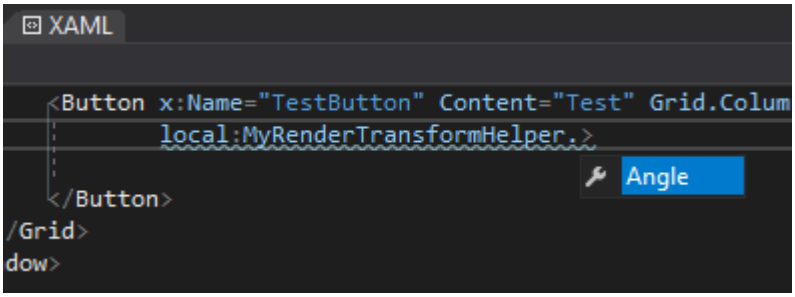
Um dies in Zukunft einfacher zu gestalten werden wir ein Attached Property hierfür erstellen welches wir Angle nennen werden.

Wir erstellen uns also mit Hilfe des CodeSnippets „wpfdpa“ ein Attached Property welches den Namen *Angle* besitzt und den Wertetyp Double besitzt.

Die Klasse habe ich in diesem Fall MyRenderTransformHelper benannt.

```
Public Class MyRenderTransformHelper
    Inherits DependencyObject


    Public Shared Function GetAngle(ByVal element As DependencyObject) As Double
        If element Is Nothing Then
            Throw New ArgumentNullException("element")
        End If

        Return CDbl(element.GetValue(AngleProperty))
    End Function

    Public Shared Sub SetAngle(ByVal element As DependencyObject, ByVal value As
Double)
        If element Is Nothing Then
            Throw New ArgumentNullException("element")
        End If

        element.SetValue(AngleProperty, value)
    End Sub

    Public Shared ReadOnly AngleProperty As _
                           DependencyProperty =
DependencyProperty.RegisterAttached("Angle",
                           GetType(Double), GetType(MyRenderTransformHelper),
                           New PropertyMetadata(Double.Parse("0"))

End Class
```

Kompilieren wir nun unser Projekt werden wir sehen das unser neues Property bereits für jedes Control zur Verfügung steht.



Aber noch haben wir keine Logik implementiert welche sicherstellen würde dass der Button auch wirklich gedreht wird. Bei den Dependency Properties haben wir bereits gelernt das wir auf Änderungen eines Property reagieren können indem wir einen Callback für das PropertyChangedCallback in den Metadaten des Properties setzen.

```
Public Shared ReadOnly AngleProperty As _
                            DependencyProperty =
DependencyProperty.RegisterAttached("Angle",
                            GetType(Double), GetType(MyRenderTransformHelper),
                            New PropertyMetadata(Double.Parse("0"), AddressOf
AngleProperty_Changed))
Private Shared Sub AngleProperty_Changed(ByVal d As DependencyObject, ByVal e As
DependencyPropertyChangedEventArgs)
    Dim elem As UIElement = TryCast(d, UIElement)
    If elem IsNot Nothing Then
        elem.RenderTransformOrigin = New Point(0.5, 0.5)
        elem.RenderTransform = New RotateTransform(CDbl(e.NewValue))
    End If

End Sub
```

Zurück im Designer – sobald wir kompiliert haben – sehen wir den nun gedrehten Button mit folgendem um einiges kürzeren Code:

```
<Button VerticalAlignment="Center"
        HorizontalAlignment="Center"
        Content="TestButton"
        local:MyRenderTransformHelper.Angle="15">
</Button>
```

Im nächsten Beispiel möchte ich zum einen zeigen wie wir ein Attached Property nur auf einen gewissen Control-Typ zulassen und zum anderen wie wir von einem Control aus ein anderes Control steuern. Ähnlich der Funktionalität des Grids.

Situation: Wir möchten von einem UserControl aus den Titel des Fensters in welchem sich dieses befindet setzen. Dies ist oft in MVVM Anwendungen von nöten.

Ich werde die Klasse WindowHelper nennen und nur auf UserControls zulassen.

```
Public Class WindowHelper
    Inherits DependencyObject

    Public Shared Function GetWindowTitle(obj As UserControl) As String
        Return DirectCast(obj.GetValue(WindowTitleProperty), String)
    End Function
    Public Shared Sub SetWindowTitle(obj As UserControl, value As String)
        obj.SetValue(WindowTitleProperty, value)
    End Sub
    Public Shared ReadOnly WindowTitleProperty As DependencyProperty =
        DependencyProperty.RegisterAttached("WindowTitle",
                                            GetType(String),
                                            GetType(WindowHelper),
                                            New UIPropertyMetadata("", AddressOf
WindowTitle_Changed))
    Private Shared Sub WindowTitle_Changed(ByVal d As DependencyObject, ByVal e As
DependencyPropertyChangedEventArgs)

End Class
```

Nun müssen wir nur noch die Methode WindowTitle_Changed mit Leben füllen.

Im Argument „d“ bekommen wir das Objekt herein welches das Property verwendet. In unserem Fall kann/darf dies nur ein UserControl sein da wir nur dieses zulassen was wir schön an den Argumenten der Methoden SetWindowTitle und GetWindowTitle sehen können. Hier haben wir die Möglichkeit ausschließlich auf UserControls beschränkt.

Das Argument „e“ stellt und unter anderem die Eigenschaft NewValue zur Verfügung, das kennen wir aber bereits aus dem Kapitel DependencyProperties.

Nach ein wenig überlegen kommen wir drauf das wir nach laden des UserControl den Fenstertitel überschreiben müssen. Folgender Code steht anschließend in der Methode:

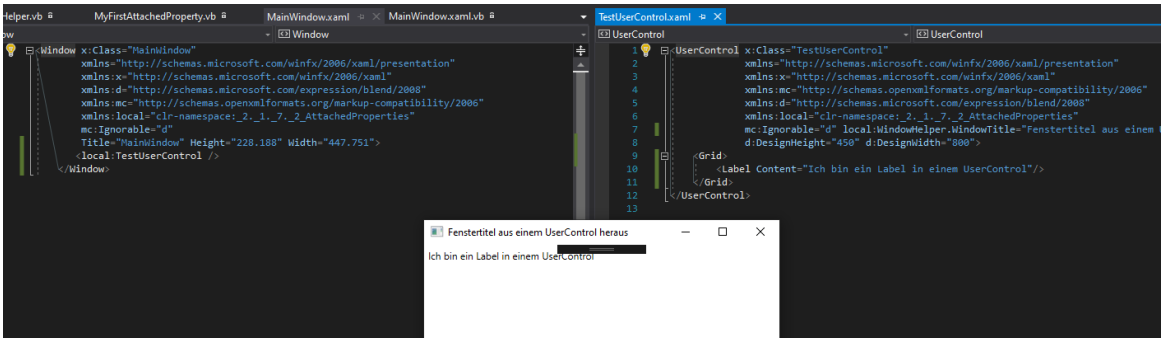
```
Private Shared Sub WindowTitle_Changed(ByVal d As DependencyObject, ByVal e As
DependencyPropertyChangedEventArgs)
    Dim ctl = TryCast(d, UserControl)
    AddHandler ctl.Loaded, Sub(sender, e2)
                               If Window.GetWindow(ctl) IsNot Nothing Then
                                   DirectCast(Window.GetWindow(ctl), Window).Title =
GetWindowTitle(ctl)
                               End If
                           End Sub
End Sub
```

Nun können wir ein UserControl erstellen und unser neues Attached Property probieren.

Erstellen wir ein UserControl mit dem Namen TestUserControl.

In dieses setzen wir zum Testen einfach nur ein Label mit ein wenig Text und setzen unser soeben erstelltes Attached Property.

Im MainWindow unseres Programms werden wir dieses UserControl mal einbinden und das Projekt starten…



Wir merken schon dass sich durch Attached Properties einige neue Möglichkeiten ergeben. Was mir persönlich gut an Attached Properties gefällt ist das man nicht unbedingt für jede kleine „Änderung“ an einem Control immer gleich ein neues Control erstellen muss sondern einfach ein wenig Logik „einpflanzt“. Ohne das Control selbst zu verändern. Der Vorteil dabei, ich kann die Klassen mit meinen Attached Properties in eine Klassenbibliothek legen und jederzeit in jedem Projekt einbinden und habe diese Funktionalitäten zur Verfügung.

Ich hoffe das hat euch gefallen. Nachstehend wieder ein Video zu diesem Kapitel, hinterlasst mir doch ein Kommentar was euch gefallen hat oder vielleicht auch was euch nicht gefallen hat.

Hier geht’s zum Video: https://youtu.be/AII2Cau0MQo

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/

2.1.8.1

Die Input API

Die Primäre API für die Eingabe ist in den Basiselementen UIElement, ContentElement, FrameworkElement und FrameworkContentElement enthalten. Einige davon haben wir ja bereits ausführlicher kennengelernt.

Alle diese Klassen bieten Funktionen für Eingabeereignisse die im Zusammenhang mit Tastatureingaben, Maustasten, dem Mausrad, sowie der Mausbewegung stehen.
Dieses System unterscheidet sich von dem System wie es in WinForms implementiert ist und ermöglicht eine andere Eingabearchitektur so das ein eigenes *EreignisRouting-Schema* möglich wird bei dem mehr als ein Element ein Ereignis behandeln kann. In der WPF ist sehr vielen Ereignissen ein Ereignispaar zugeordnet wie Beispielsweise das KeyDown-Ereignis und das PreviewKeyDown-Ereignis. Der Unterschied zwischen diesen Ereignissen liegt darin wie an das Zielelement weitergeleitet wird.

Vorschauereignisse (mit dem Präfix Preview) „Tunneln“ die Elementstruktur vom Stammelement nach unten wobei „Bubbling“-Ereignisse (ohne dem Präfix Preview) vom Zielelement an das Stammelement übergeben. Der Unterschied zwischen Bubbling und Tunneling wird aber in Kapitel 2.1.8.3 noch ausführlicher erläutert. Es ist auch wichtig dieses System zu verstehen da es hier gerne mal zu Verwirrung kommen kann wenn einem der Unterschied nicht klar ist.

Aber gehen wir vorher mal zu Tastatur und Mauseingaben…

2.1.8.2

Tastatur und Mausklassen

Zusätzlich zur Input API in den Basiselement-Klassen stellen die Keyboard-Klasse und die Mouse-Klasse weitere APIs für die Arbeit mit der Tastatur und Maus zur Verfügung.

Tastatur

Ein Beispiel hierfür wäre die GetKeyState der Keyboard Klasse.

```
If (Keyboard.GetKeyStates(Key.Return) And KeyStates.Down) > 0 Then
    Debug.WriteLine("Return is pressed!")
End If
```

Maus

Dasselbe gilt für die Mouse-Klasse. Im folgenden Beispiel wird ermittelt ob die linke Maustaste gedrückt wurde.

```
If Mouse.LeftButton = MouseButtonState.Pressed Then
     Debug.WriteLine("Left Mousebutton pressed!")
End If
```

Stifteingabe

Die WPF verfügt über integrierte Unterstützung für Stylus-Eingaben.
Stylus sind Stifte zur Eingabe an einem Tablet, gerade in letzter Zeit und nicht zuletzt durch die Surface Produkte von Microsoft erfreut sich diese Art der Eingabe einer neuen Beliebtheit. WPF Anwendungen können den Stift mithilfe der Mause-API als Maus behandeln aber auch als Stift-Gesten. Hierfür verfügt die WPF über Stylus-Api. All Stiftbezogenen APIs enthalten „Stylus“ in ihrem Namen. Wie Beispielsweise folgende Events: PreviewStylusButtonDown, StylusButtonDown, PreviewStylusInAirMove, StylusInAirMove.

Wie man gut erkennen kann gibt es auch hier die „Paare“ mit und ohne dem Präfix „Preview“.

Da der Stift aber auch als Maus fungieren kann, können bestehende Anwendungen welche nur die Mauseingabe unterstützen trotzdem bis zu einem gewissen Teil bedient werden, bestimmte einem Stift vorbehaltene Features können in diesem Fall allerdings nicht genutzt werden.

2.1.8.3

Eventrouting

Die ist meiner Meinung nach ein recht wichtiges Thema in der WPF um das Eventhandlich der WPF verstehen zu lernen.

Ich versuche es möglichst nicht zu Theoretisch zu erklären. In der WPF gibt es einen ElementTree. In diesem ElementTree sind die Elemente (Control) enthalten. Für Ereignisse aller Control innerhalb eines Elementrees gibt es mehrere Routingmechanissmen welche in den Metadaten des Events hinterlegt sind.

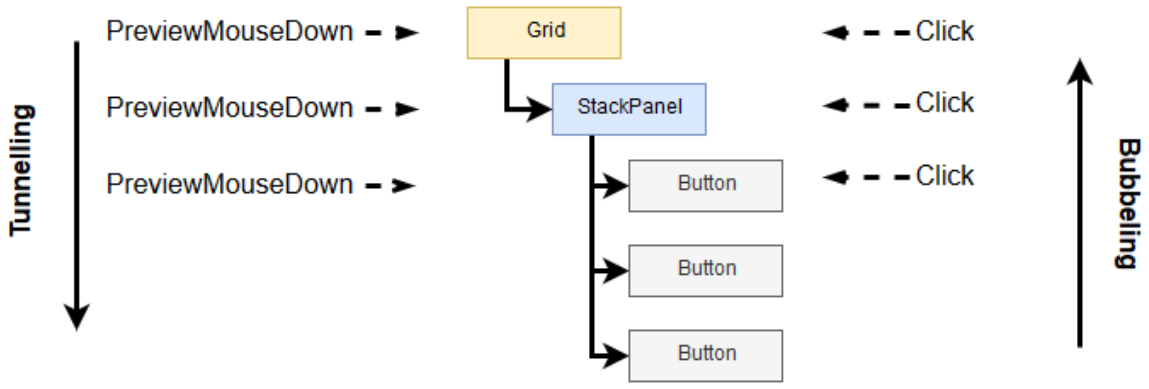
- *Direct*
- *Bubbling*
- *Tunneling*

In der WPF kann ein Ereignis nicht nur von einem einzelnen Element „bearbeitet“ werden.
Ich weis nicht ob „bearbeitet“ das richtige Wort ist, aber entscheide selbst.

Das Direkte Routing-Ereignis (Direct) wird direkt auf diesem Element ausgelöst und auch nur auf diesem. Wie man dies aus WinForms kennt.
Bubbling arbeitet sich die Elementstruktur nach oben. Es Blubbert also nach ob die Elemente durch, indem zuerst das Element benachrichtigt wird, von dem das Ereignis stammt, und dann das übergeordnete Element benachrichtigt, dann wieder eines höher usw.

Tunneling beginnt am Stamm der Elementstruktur (z.b. UserControl) und Arbeitet sich Element für Element nach unten bis es bei dem Element angelangt ist welches das Ereignis ursprünglich ausgelöst hatte.

Folgende Abbildung zeigt welche Ereignisse auf welchem Objekt ausgelöste werden wenn auf einen Button geklickt wird welches sich innerhalb eines Stackpanels befindet:



Per Konvention sind Events welche mit dem Präfix „Preview“ versehen sind Events welche das Tunneling als Routingsstrategie verwenden während Events ohne diesem Präfix immer ein Bubbling als Strategie verwenden.

Wenn also wie in der Abbildung zu sehen ist auf den Button geklickt wird zuerst am Button das Click-Event geworfen. Anschließend wird auf dem StackPanel das Click-Event geworfen und danach am Grid.

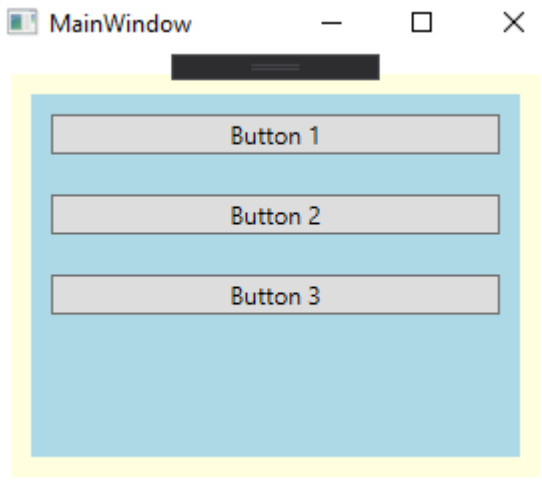
Schreiben wir eine kleine Demoanwendung welche dies besser veranschaulicht. Ihr werdet überrascht sein.

Wir erstellen eine neue WPF Anwendung und ersetzen im MainWindow den Code des per Default erstellten Grids gegen folgenden:

```
<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:BubblingTunnelingDemo"
        mc:Ignorable="d"
        Title="MainWindow" Height="260" Width="300">
    <Grid x:Name="MainGrid" Margin="10"  Background="LightYellow"
          PreviewMouseDown="PreviewMouseDown"
          PreviewMouseUp="PreviewMouseUp"
          MouseDown="MouseDown"
          MouseUp="MouseUp">
        <StackPanel x:Name="MainStackPanel" Margin="10" Background="LightBlue"
                    PreviewMouseDown="PreviewMouseDown"
                    PreviewMouseUp="PreviewMouseUp"
                    MouseDown="MouseDown"
                    MouseUp="MouseUp">
            <Button x:Name="Button1" Content="Button 1" Margin="10"
                    Click="ButtonClick"
                    PreviewMouseDown="PreviewMouseDown"
                    PreviewMouseUp="PreviewMouseUp"/>
            <Button x:Name="Button2" Content="Button 2" Margin="10"
                    Click="ButtonClick"
                    PreviewMouseDown="PreviewMouseDown"
                    PreviewMouseUp="PreviewMouseUp"/>
            <Button x:Name="Button3" Content="Button 3" Margin="10"
                    Click="ButtonClick"
                    PreviewMouseDown="PreviewMouseDown"
                    PreviewMouseUp="PreviewMouseUp"/>
        </StackPanel>
    </Grid>
</Window>
```

Vielleicht legen wir noch die Größe des Fensters auf:

```
Height="260" Width="300"
```



und fügen folgenden Code in die MainWindow.vb – CodeBehind Datei ein:

```
Private Sub ButtonClick(sender As Object, e As RoutedEventArgs)
        Debug.WriteLine($"{GetMethodName()} from Element with Name: '{DirectCast(sen-
der, FrameworkElement).Name}'")
    End Sub
    Private Shadows Sub PreviewMouseDown(sender As Object, e As MouseButtonEventArgs)
        Debug.WriteLine($"{GetMethodName()} from Element with Name: '{DirectCast(sen-
der, FrameworkElement).Name}'")
    End Sub
    Private Shadows Sub PreviewMouseUp(sender As Object, e As MouseButtonEventArgs)
        Debug.WriteLine($"{GetMethodName()} from Element with Name: '{DirectCast(sen-
der, FrameworkElement).Name}'")
    End Sub
    Private Shadows Sub MouseDown(sender As Object, e As MouseButtonEventArgs)
        Debug.WriteLine($"{GetMethodName()} from Element with Name: '{DirectCast(sen-
der, FrameworkElement).Name}'")
    End Sub
    Private Shadows Sub MouseUp(sender As Object, e As MouseButtonEventArgs)
        Debug.WriteLine($"{GetMethodName()} from Element with Name: '{DirectCast(sen-
der, FrameworkElement).Name}'")
    End Sub
    Private Function GetMethodName(<Runtime.CompilerServices.CallerMemberName>
    Optional memberName As String = Nothing) As String
        Return memberName
    End Function
```

Starten wir die Applikation und sehen wir uns das ganze mal genauer an und klicken auf den Button 1.

Hierdurch wird folgender Text in der Ausgabe generiert:

```
PreviewMouseDown from Element with Name: 'MainGrid'
PreviewMouseDown from Element with Name: 'MainStackPanel'
PreviewMouseDown from Element with Name: 'Button1'
PreviewMouseUp from Element with Name: 'MainGrid'
PreviewMouseUp from Element with Name: 'MainStackPanel'
PreviewMouseUp from Element with Name: 'Button1'
ButtonClick from Element with Name: 'Button1'
```

Wir sehen schön, dass hier auf jedem Element an welchem wir das Event mit dem Preview-Präfix abonniert haben das Tunneling greift und die Events von oben nach unten geworfen werden. Erst nachdem auch für den Button das PreviewMouseUp geworfen wurde wird das Click-Event für den Button geworfen. PreviewMouseDown und PreviewMouseUp Tunneln von Oben nach unten zum Zielelement.

Sehen wir uns nun an was passiert wenn wir einen Klick auf das StackPanel machen:

```
PreviewMouseDown from Element with Name: 'MainGrid'
PreviewMouseDown from Element with Name: 'MainStackPanel'
MouseDown from Element with Name: 'MainStackPanel'
MouseDown from Element with Name: 'MainGrid'
PreviewMouseUp from Element with Name: 'MainGrid'
PreviewMouseUp from Element with Name: 'MainStackPanel'
MouseUp from Element with Name: 'MainStackPanel'
MouseUp from Element with Name: 'MainGrid'
```

Nun sehen wir schön das die Bubbling-Events MouseUp und MouseDown in die Gegenrichtung ausgelöst werden. Nämlich zuerst am StackPanel und erst dann am Grid.

Um nun zu verhindern das Events weiter nach oben Bubbeln oder nach unten Tunneln kann das Handled Property des EventArgs auf True gesetzt werden. In diesem Fall gilt das Ereignis als behandelt und geht nicht mehr zum nächsten Element weiter.

2.1.8.4

Touch und Multitouch

Auch ein immer wichtiger werdendes Thema heute. Touch und Multitouch-Bildschirme werden immer günstiger und in vielen Geräten befinden sich bereits solche Bildschirme. Auch in der Industrie finden die Bildschirme immer mehr Beliebtheit da auch diese immer Robuster werden und die Bedienung schlichtweg intuitiver ist als mit einer Tastatur.

Seit .Net 4.0 und Windows 7 bieten die Klassen UIElement, UIElement3D und ContentElement Events an um die Funktionalität von Multitouch-Displays zu nutzen. Die Events werden geworfen wenn der Benutzer mit dem Finger ein WPF Element berührt, den Finger darin bewegt oder loslässt.

- *TouchDown*
  Tritt beim berühren des Bildschirms innerhalb eines Elements auf
- *TouchUp*
  Tritt auf wenn der Finger den Bildschirm wieder loslässt, wenn dieser vorher innerhalb eines Elements befand
- *TouchLeave*
  Tritt beim verlassen des Elements auf, also wenn der Finger aus dem Element herausfährt
- *TouchMove*
  Tritt auf wenn der Finger sich innerhalb des Elements bewegt

Alle Touch-Events sind als Bubbling-Events implementiert, Blubbern also den ElementTree nach oben und verwenden die TouchEventArgs.

Die Klasse TouchEventArgs besitzt unter anderem zwei interessante Eigenschaften bzw. Methoden. GetTouchPoint() gibt einen TouchPoint zurück welcher wiederum die Eigenschaft Position enthält und gibt die relative Position zum Berührungspunkt zurück.

Die TouchDevice Eigenschaft vom Typ TouchDevice enthält eine ID vom Typ Integer und enthält einen eindeutigen Wert für den Berührungspunkt. Die ID ist dabei abhängig vom Treiber und dem Betriebssystem. Da es bei einem MultiTouch-Display ja mehrere Berührungspunkte geben kann (auch mehr oder weniger gleichzeitig) ist die Auswertung der ID wichtig um Rückschlüsse auf die „Geste“ zu bekommen, welche der User gerade macht.

Aber neben den genannten „Low-Level-Touch“ Events bietet die Klasse UIElement auch die sehr interessanten Manipulation-Events.

Die Manipulation-Events werden zum *Skalieren*, *Rotieren* und *Verschieben* eines Elements genutzt. Damit die Events überhaupt geworfen werden muss die Eigenschaft IsManipulationEnabled des Elements auf True gesetzt werden da diese per Default deaktiviert ist. Das geht sowohl über XAML als auch über die CodeBehind.

Es gibt für die Manipulation einige Events: ManipulationStarting, ManimulationStarted, ManimulationDelta, ManipulationCompleted.

Da die MultiTouch Funktionalität hier an dieser Stelle und auch in einem WebCast Video recht schwer zu „zeigen“ ist verzichte ich an dieser Stelle darauf. Für alle die ein MultiTouch-Display zu Verfügung haben können sich mit folgendem Beispiel aus der MSDN ein wenig spielen:

https://docs.microsoft.com/en-us/dotnet/framework/wpf/advanced/walkthrough-creating-your-first-touch-application

2.1.8.5

Fokus

Wie jetzt ein eigenes Kapitel über den Fokus? In der Tat. Fokus ist nicht gleich Fokus.

Es gibt in der WPF zwei Hauptkonzepte. Den *Tastaturfokus* und den *logischen Fokus*. Es ist wichtig den Unterschied zwischen den beiden Konzepten zu kennen.

Die beiden Keyplayer sind die Klassen Keyboard und FocusManager. Die Klasse Keyboard betrifft den *Tastaturfokus* während der FocusManager den logischen Fokus in der Hand hat.

Tastaturfokus

Der Tastaturfokus bezieht sich immer auf das Element welches gerade Tastatureingaben empfängt. Es kann am gesamten Desktop nur ein Element geben welches Tastatureingaben empfängt. Innerhalb eine WPF Anwendung ist auf diesem Element die Eigenschaft IsKeyboardFocused in diesem Fall auf True festgelegt. Die statische Eigenschaft FocusedElement der Klasse Keyboard gibt immer das Element zurück welches gerade den Tastaturfokus besitzt.

Damit ein Element den Tastaturfokus erhalten kann muss die Eigenschaft Focusable auf True festgelegt sein. Dies ist bei Elementen wie der *TextBox* oder der *CheckBox* per Default auf True, bei Elementen wie Panels allerdings auf False, wodurch diese keinen Tastaturfokus erhalten können solange die Eigenschaft nicht auf True gesetzt wird.

Der Benutzer kann den Tastaturfokus auf ein Steuerelement festlegen indem er in dieses Klickt oder die Tabulatortaste benutzt. Aber der Focus kann auch über Code gesteuert werden.

Folgende Methode setzt nach dem laden des Fensters den KeyboardFocus auf den *Button* mit dem Namen firstButton.

```
Private Sub OnLoaded(ByVal sender As Object, ByVal e As RoutedEventArgs)
    Keyboard.Focus(firstButton)
End Sub
```

Mit der Eigenschaft IsKeyboardFocused eines Elements kann abgerufen werden ob dieses gerade den Tastaturfokus besitzt. Interessant ist auch die Eigenschaft IsKeyboardFocusWithin. Diese gibt einen Wert zurück welcher aussagt ob ein diesem Element untergeordnetes Element gerade den Focus besitzt.

Logischer Fokus

Der Logische Focus wird von der statischen Klasse FocusManager jeweils für einen gewissen Bereich verwaltet. Diese Bereiche nennt man auch Focusbereich (*Focus-Scope*) und wird definiert indem man das AttachedProperty FocusManager.IsFocusScope eines Elements auf True setzt.

Elemente wie *Window*, *Menu*, *Toolbar* oder *ContextMenü* haben diese Eigenschaft per Default auf True gesetzt und definieren damit ihren eigenen FocusScope, also ihren Fokus-Bereich.

Angenommen es gibt ein *Window* mit einem Menu welches ein *MenuItem* enthält und zusätzlich einer *TextBox* innerhalb des *Window*. Wechselt nun der Tastaturfokus von der *TextBox* auf das *MenuItem* weil der User mit TAB navigiert, verliert die *TextBox* den Tastaturfokus, behält aber den logischen Fokus innerhalb des Fokusbereichs des Windows. Das *MenuItem* erhält also den Tastaturfokus UND den logischen Fokus innerhalb des Fokusbereichs des *Menu*.

Geht nun der Tastaturfokus zurück zum *Window*, erhält das Element mit dem logischen Focus innerhalb dieses Fokusbereichs wieder den Tastaturfokus, was in diesem Fall die *TextBox* ist. Das *MenuItem* verliert also den Tastaturfokus aber behält aber innerhalb des Fokusbereichs des *Menu* den logischen Fokus.

Es können also mehrere Elemente im logischen Fokusbereich liegen aber nur ein Element innerhalb des Fokusbereichs den logischen Fokus haben.

Folgendes Beispiel ruft das fokussierte Element eines Fokusbereichs ab (hier im Fokusbereichs des *Menu*):

```
FocusManager.GetFocusedElement(meineMenuInstanz)
```

Falls kein Element in diesem Fokusbereich den logischen Fokus besitzt, gibt die Methode Nothing zurück.

Fazit:

Wichtig ist das wir den Unterschied zwischen dem logischen Fokus und dem Tastaturfokus kennen oder einfach wissen das es hier einen Unterschied gibt. Unser Hauptaugenmerk sollte immer auf dem Tastaturfokus liegen da uns der logische Fokus im Normalfall nicht so oft über den Weg läuft und wir uns nur bei komplexeren Szenarien beschäftigen müssen. Kommt es allerdings mal zu einem unerwünschten Verhalten was den Fokus betrifft wissen wir nun dass wir uns den Focus-Scope genauer ansehen müssen um der Sache auf den Grund zu gehen.

2.1.8.6

Commands

Commands sind ein sehr wichtiger Mechanismus in der WPF. Der Zeck von Command ist zum einen die Trennung zwischen dem Element welches den Command auslöst und der Logik welche hinter einem Command steht und zum anderen hat ein Command eine Eigenschaft welche bestimmt ob dieser ausgeführt werden kann. Dazu kommen wir aber noch.

Wenn wir einen EventHandler für einen Button-Klick setzen können wir innerhalb dieses EventHandler Logik unterbringen. Wir können denselben EventHandler allerdings nicht verwenden wenn wir die Logik zu einem anderen Ereignis als einem Klick ausführen möchten.

Weiters müssen wir uns selbst darum kümmern das ein Button „disabled“ wird wenn die Aktion im Moment nicht ausgeführt werden kann. Ein Beispiel hierfür wäre das Speichern eines Formulars wenn noch nicht alle Pflichtfelder ausgefüllt sind. In diesem Fall darf der User noch nicht auf Speichern klicken können. Wir müssen also immer prüfen ob alle Felder ausgefüllt sind und wenn dies der Fall ist den Button „enablen“. Dies kann uns alles ein Command abnehmen.

Außerdem können Command sehr einfach eine Tastenkombination zugeordnet werden und so Beispielsweise F5 für das Speichern zu verwenden.

Ein einfaches Beispiel

Die einfachste Möglichkeit einen Command zu verwenden ist es einen vordefinierten RoutedCommand zu verwenden. Microsoft war so nett uns eine Reihe von Commands zu definieren welche wir überall Verwenden können.
Der Command Paste ist einer dieser vordefinierten Commands und befindet sich in der statischen Klasse ApplicationCommands.

Erstellen wir uns ein neues Projekt und ersetzen das Grid des MainWindow gegen folgenden Code:

```
<StackPanel>
    <Menu>
        <MenuItem Command="ApplicationCommands.Paste" />
    </Menu>
    <TextBox />
</StackPanel>
```

Debuggen wir nun dieses Projekt sehen wir das das MenuItem „Einfügen“ benannt ist obwohl wir diesen Text nicht im XAML definiert haben. Weiters ist das MenuItem Enabled=False. Setzen wir nun den Keyboardfokus auf die TextBox wird sofort das MenuItem Enabled=True.

Haben wir im Moment einen Text in der Zwischenablage können wir diesen Text nun durch einen Klick auf das *MenuItem* einfach in die *Textbox* einfügen.

OK, das sieht jetzt alles ein wenig nach Magie und Hexenwerk aus, gebe ich zu.

Das Modell der RoutedCommands kann in vier Hauptkonzepte aufgeteilt werden.

- *Der Befehl (Command) – ist die Aktion die ausgeführt werden soll*
- *Die Befehlsquelle (Command-Source) – das Objekt welches den Befehl aufruft*
- *Das Befehlsziel (Command-Target) – das Objekt auf das der Befehl ausgeführt wird*
- *Die Befehlsbindung (Command-Binding) – das Objekt das den Befehl der Logik zuordnet*

Im vorherigen Beispiel ist Paste der Befehl, MenuItem ist die Befehlsquelle, die TextBox das Befehlsziel und die Befehlsbindung wird von der TextBox bereitgestellt.

Ein Command-Objekt implementiert die Schnittstelle ICommand. Die ICommand Schnittstelle hat zwei Methoden. Die Execute und die CanExecute Methode sowie ein Event CanExecuteChanged.

Wir können es uns bereits denken, Execute führt den Befehl aus und CanExecute bestimmt ob der Befehl ausgeführt werden kann.

Das Ereignis CanExecutChanged wird ausgeführt wenn der CommandManager der WPF eine Änderung einer Befehls-Quelle erkennt und CanExecute kann somit aktualisiert werden.

Nun ist es ja schon nicht mehr so viel Magie. Da einem RoutedCommand also die Quelle kennt (im obigem Beispiel das *MenuItem*) kann es dessen Header auch schreiben wodurch „Einfügen“ als Header eingesetzt wird. Allerdings nur wenn durch den Entwickler kein Header angegeben wurde. Dass das *MenuItem* automatisch *Enabled* oder *Disabled* wird, gibt Methode CanExecute zurück.

Input-Gestures

Imput-Gestures (Eingabegesten) können als Befehlsquelle verwendet werden. Es gibt z.b. KeyGesture und MouseGesture um zwei Eingabearten zu nennen.

Beispielsweise ist dem Command Paste die KeyGesture STRG+V zugeordnet.

Adaptieren wir das vorherige Beispiel um die Möglichkeit mit der Taste F3 den Einfügebefehl auszuführen.

Wir ersetzen den Code des StackPanels und fügen stattdessen folgenden Code ein:

```
<Window.InputBindings>
    <KeyBinding Key="F3"
          CommandTarget="{Binding ElementName=txtTest}"
          Command="ApplicationCommands.Paste" />
</Window.InputBindings>
<StackPanel>
    <Menu>
        <MenuItem Command="ApplicationCommands.Paste" />
    </Menu>
    <TextBox x:Name="txtTest" />
</StackPanel>
```

Da der Command nun nicht auf dem Ziel Definiert ist auf welchem es ausgeführt wird, müssen wir über die Eigenschaft CommandTarget das Ziel festlegen was die *TextBox* sein soll. Das ist auch der Grund warum die *TextBox* nun einen Namen bekommen hat.

Starten wir nun das Programm können wir Text aus der Zwischenablage mit F3 einfügen, unabhängig davon ob die *TextBox* den Fokus hat oder hatte, da wir das Ziel explizit angegeben haben.

Aber wir können einem vorhandenem Command auch eigene Logik implementieren. Hierfür können wir sogenannte CommandBindings im XAML definieren.

Versuchen wir dies Anhand des Commands Open:

```
<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open"
              Executed="OpenExecuted"
              CanExecute="OpenCanExecute"/>
</Window.CommandBindings>
```

Sowohl bei „OpenExecute“ als auch bei „OpenCanExecute“ können wir mit einem *Rechtsklick -> Gehe zu Definition* nun die EventHandler im CodeBehind erstellen lassen.

Folgende CodeBehind sollten wir nun vor uns finden:

```
Class MainWindow
    Private Sub OpenExecuted(sender As Object, e As ExecutedRoutedEventArgs)

    End Sub

    Private Sub OpenCanExecute(sender As Object, e As CanExecuteRoutedEventArgs)

    End Sub
End Class
```

In diese beiden Methoden können wir nun Logik platzieren welche beim ausführen dieses Command ausgeführt werden soll.

In OpenExecute möchten wir in diesem Beispiel einfach nur eine MessageBox zeigen in welcher wir uns anzeigen lassen möchten welcher Command auf welchem Objekt ausgeführt wurde um zu sehen was passiert.

```
Private Sub OpenExecuted(sender As Object, e As ExecutedRoutedEventArgs)
    Dim command = CType(e.Command, RoutedCommand).Name
    Dim targetobj = CType(sender, FrameworkElement).Name
    MessageBox.Show($"Der Command '{command}' auf dem Objekt '{targetobj}' wurde
ausgeführt.")
End Sub
```

Über die ExecutedRoutedEventArgs kommen wir unter anderem an den Command welcher ausgeführt wird und über den Parameter Sender können wir den Namen des Objekts abrufen.

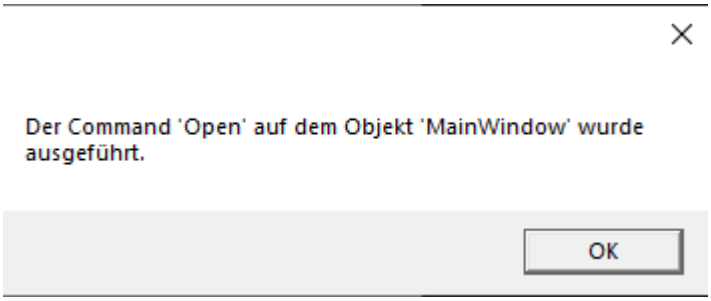
In der OpenCanExecute Methode setzen wir die CanExecute Eigenschaft des Parameters e vom Typ CanExecuteRoutedEventArg auf True um die Ausführung des Commands immer zu erlauben, da wir in diesem Fall keine Abhängigkeiten haben welche die Ausführung behindern nicht erlauben würde.

```
Private Sub OpenCanExecute(sender As Object, e As CanExecuteRoutedEventArgs)
     e.CanExecute = True
End Sub
```

Starten wir das Programm wird nichts passieren, wir haben nämlich weder eine InputBinding definiert noch einem *Button*, *MenuItem* oder einem anderen Steuerelement diesen Command zugewiesen. Das werden wir nachholen.

```
<Window.CommandBindings>
    <CommandBinding Command="ApplicationCommands.Open"
                    Executed="OpenExecuted"
                    CanExecute="OpenCanExecute"/>
</Window.CommandBindings>
<Window.InputBindings>
    <KeyBinding Key="F3"
          CommandTarget="{Binding ElementName=txtTest}"
          Command="ApplicationCommands.Paste" />
    <KeyBinding Modifiers="Ctrl"
                Key="O"
                Command="ApplicationCommands.Open"/>
</Window.InputBindings>
<StackPanel>
    <Menu>
        <MenuItem Command="ApplicationCommands.Paste" />
        <MenuItem Command="ApplicationCommands.Open" />
    </Menu>
    <TextBox x:Name="txtTest" />
</StackPanel>
```

Starten wir nun das Programm können wir sowohl über das *MenuItem* als auch über die Tastenkombination STRG+O das Command ausführen und erhalten die *MessageBox*:



Wir haben also gelernt was es mit *Commands*, *CommandBindings* und *InputBindings* auf sich hat. Am besten ist es man spielt ein wenig damit rum, und versucht ein wenig Logik in vorhandene Command zu bekommen. Die vorhandenen ApplicationCommands eigenen sich hervorragend um einen kleinen TextEditor zu basteln.

Eine Übersicht zu allen vordefinierten Command findet man hier:
https://docs.microsoft.com/de-de/dotnet/api/system.windows.input.applicationcommands?view=netframework-4.8

Eigene Commands erstellen

Um eigene Command zu erstellen gibt es zwei Möglichkeiten. Zum einen kann man eine Klasse erstellen welche die Schnittstelle ICommand implementiert oder, die gängigere Methode, die Erstellung eines RoutedCommand oder eines RoutedUiCommands.

Da wir die Schnittstelle im nächsten Kapitel „RelayCommands“ kennenlernen werden, gehe ich hier erstmal auf die letztere Möglichkeit ein.

Wir erstellen in diesem Beispiel einen RoutedCommand welcher bei seiner Ausführung einfach eine *MessageBox* zeigen soll. Um einen RoutedCommand zu erstellen gehen wir in die CodeBehind des *Window* und erstellen eine Statische Variable vom Typ RoutedCommand.
Ich erstelle hierfür ein neues Fenster mit dem Namen *CustomRoutedCommand*.

```
Public Shared ShowMessageCommand As RoutedCommand = New RoutedUICommand
```

Im XAML definieren wir nun ein CommandBinding für diesen Command. Da die Variable als Shared gekennzeichnet ist können wir im XAML über x:Static darauf zugreifen.
Wir kompilieren vorher kurz um im XAML Intellisense zu erhalten.

```
<Window.CommandBindings>
    <CommandBinding Command="{x:Static local:CustomRoutedCommand.ShowMessageCommand}"
                    Executed="ShowMessageExecuted"
                    CanExecute="ShowMessageCanExecute"/>
</Window.CommandBindings>
```

Nachdem wir uns die Handler generieren haben lassen, sieht unsere CodeBehind folgendermaßen aus:

```
Public Class CustomRoutedCommand

    Public Shared ShowMessageCommand As RoutedCommand

    Private Sub ShowMessageExecuted(sender As Object, e As ExecutedRoutedEventArgs)

    End Sub

    Private Sub ShowMessageCanExecute(sender As Object, e As
CanExecuteRoutedEventArgs)

    End Sub
End Class
```

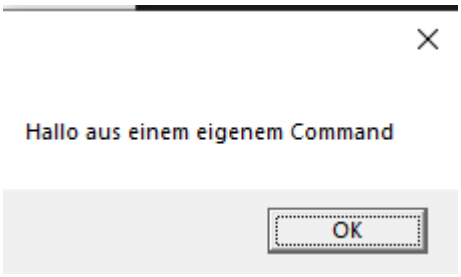
Gehen wir zurück zum XAML, dort möchten wir nun noch ein MenuItem erstellen welches diesen Command ausführen soll.

```
<StackPanel>
    <Menu>
        <MenuItem Header="Zeige Messagebox" Command="{x:Static
local:CustomRoutedCommand.ShowMessageCommand}"/>
    </Menu>
    <TextBox />
</StackPanel>
```

Zum Schluss noch den Code in die EventHandler eingetragen:

```
Public Class CustomRoutedCommand

    Public Shared ShowMessageCommand As RoutedCommand = New RoutedUICommand()

    Private Sub ShowMessageExecuted(sender As Object, e As ExecutedRoutedEventArgs)
        MessageBox.Show("Hallo aus einem eigenen Command")
    End Sub

    Private Sub ShowMessageCanExecute(sender As Object, e As
CanExecuteRoutedEventArgs)
        e.CanExecute = True
    End Sub
End Class
```

Starten wir nun das Programm bekommen wir die MessageBox nach einem Klick auf das MenuItem zu sehen.



Wir sehen also dass es nicht schwer ist eigene Commands zu implementieren. Auch Tastenkombinationen und das automatische „Labeln" des Objekts an welches der Command „gebunden" ist, ist möglich und durch die CanExecute Methode haben wir sogar Beispielsweise einen Button automatisch Disabled wenn der Command nicht ausgeführt werden kann.

Hier geht's zum Video: https://youtu.be/fY3lYug6u9w

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

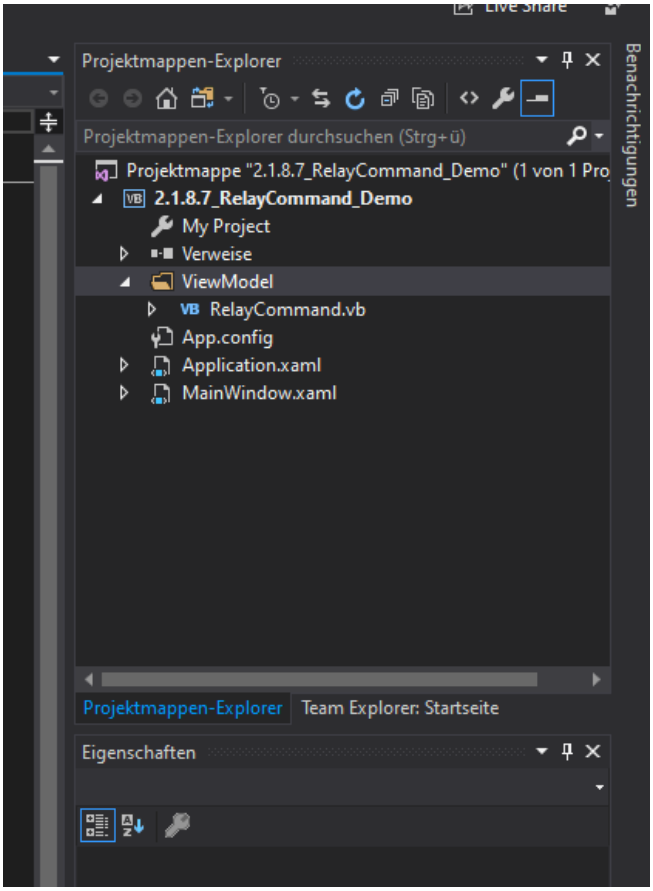
Hier zum SupportThread im Forum: https://www.vb-paradise.de/index.php/Thread/124641-Support-Anregungen-W%C3%BCnsche-zur-Tutorialreihe-WPF-lernen/

2.1.8.7

RelayCommand

Nun möchten wir uns das ganze etwas einfacher und mit voller Bindingunterstützung machen. Bisher mussten wir die Methoden der CodeBehind im XAML angeben wodurch die View nicht vom Code getrennt war. Aber genau das ist unter WPF ja immer unser Ziel. Wir müssen also eine Möglichkeit schaffen genau dies zu erreichen. Und da fällt einem nach kurzem Nachdenken ein das man doch am besten sowohl die Erstellung als auch das Verbinden von Command und den Methoden für CanExecute und Execute am besten im Code alleine machen sollte. So hat man die volle Kontrolle. Also probieren wir dies mal.

Wir haben ja gelernt dass jede Command-Klasse ICommand implementiert. OK, erstellen wir uns eine Klasse und nennen diese *RelayCommand*. Ich packe solche Klassen gerne in einen eigenen Namespace um Ordnung zu haben. Und da ich meine Ordnerstruktur der Solution immer meinen Namespaces anpasse erstelle ich somit einen Ordner. Für mich hat sich dieses Vorgehen als sehr Praktisch erwiesen.



Somit sieht die neue Klasse nun erstmal wie folgt bei mir aus:

```
Namespace ViewModel
    Public Class RelayCommand

    End Class
End Namespace
```

Nun müssen wie ICommand implementieren. Für die jenigen welche nicht wissen wie man eine Schnittstelle am einfachen implementiert hier ein paar Bilder. Wir schreiben also unter dem Klassennamen „Implements ICommand“ und Drücken ENTER.

Daraufhin bekommen wir von VisualStudio alle Methoden, Events und Eigenschaften direkt in unsere Klasse implementiert.

Nun sieht die Klasse wie folgt aus:

```
Namespace ViewModel
    Public Class RelayCommand
        Implements ICommand

        Public Event CanExecuteChanged As EventHandler Implements ICommand.CanExecuteChanged

        Public Sub Execute(parameter As Object) Implements ICommand.Execute
            Throw New NotImplementedException()
        End Sub

        Public Function CanExecute(parameter As Object) As Boolean Implements ICommand.CanExecute
            Throw New NotImplementedException()
        End Function
    End Class
End Namespace
```

Nun liegt es an uns die Klasse mit Code zu befüllen. Dafür klären wir mal was wir überhaupt brauchen. Bis jetzt haben wir nur erfahren das wir eine eigene Command-Klasse bauen möchten. Wieso wissen wir, aber wie genau nicht.

Der Nachteil an bestehenden Commandklassen wie der RoutedCommand-Klasse ist das man dieser keine Methode übergeben kann welche ausgeführt werden soll wenn der Command ausgeführt/angestoßen wird. Dies möchten wir in unserer Klasse besser machen. Mit Hilfe der Schlüsselworte Action oder Predicate sowie ein wenig Lambda (Hoch leben Lambda-Expressions) geht dies auch. Wir möchten also einen Konstruktor schaffen welchem wir ein Action(Of Object) übergeben können sowie ein Predicate(Of Object) für die CanExecute Methode. Hier wird ein Predicate benötigt weil die CanExecute Methode einen Rückgabewert (Boolean) besitzt welcher später zurückgibt ob der Command ausgeführt werden kann oder nicht.

Diese Objekte müssen wir uns auch speichern also benötigen wir zwei Felder. Diese können ReadOnly sein.

```
ReadOnly _execute As Action(Of Object)
ReadOnly _canExecute As Predicate(Of Object)
```

Und den Konstruktor:

```
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
```

Da „execute“ auf jeden Fall benötigt wird sollten wir Abfragen ob hier auch ein Objekt hereingereicht wird. Für „canexeute“ benötigen wir es nicht unbedingt da wir später auch gerne darauf verzichten können. Im Falle dass dieses Feld Nothing ist können wir gerne True zurückgeben.

Wir füllen erstmal unsere zwei Methoden aus welche beim Implementieren der ICommand-Schnittstelle generiert wurden.

In Execute führen wir im Grunde nur die übergebene Methode (Action) aus und in CanExecute prüfen wir ob unser Feld *_canExecute* nicht Nothing. Ist dies der Fall geben wir die Rückgabe der Methode als Ergebnis zurück.

So sieht unsere Klasse bis nun aus:

```
Namespace ViewModel
    Public Class RelayCommand
        Implements ICommand

        ReadOnly _execute As Action(Of Object)
        ReadOnly _canExecute As Predicate(Of Object)

        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

        Public Event CanExecuteChanged As EventHandler Implements ICommand.CanExecuteChanged

        Public Sub Execute(parameter As Object) Implements ICommand.Execute
            _execute(parameter)
        End Sub

        Public Function CanExecute(parameter As Object) As Boolean Implements ICommand.CanExecute
            Return _canExecute Is Nothing OrElse _canExecute(parameter)
        End Function
    End Class
End Namespace
```

Widmen wir uns nochmals dem CanExecute.
Wir haben im Kapitel ***Commands*** gelernt das es den CommandManager gibt, welcher dafür zuständig ist das die Oberfläche (z.b. ein Button) weis wenn sich am Status eines Commands etwas ändert. Bei jeder Aktion eines Users soll ja überprüft werden ob der Command ausgeführt werden darf. Der CommandManager ruft also immer und immer wieder sein Event RequerySuggested auf. Dieses Event abonnieren Commands um anschließend intern die Methode CanExecute abermals aufzurufen.

Dieses Event müssen auch wir in unserer Klasse abonnieren. Da uns die Schnittstelle ICommand auch dieses Event zur Verfügung gestellt hat müssen wir in dieses nun eingreifen. Dies geht unter VB.Net nur indem wir aus diesem Event ein **CustomEvent** machen.

Sobald wir ein CustomEvent daraus machen müssen wir AddHandler und RemoveHandler in dieses Event implementieren und anschließend das Event werfen (RaiseEvent).

```
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
```

Damit ist unsere RelayCommand-Klasse im Grunde fertig und wir können versuchen diese anhand eines kleinen Beispiels auszuprobieren.

Erstellen wir eine simple Klasse mit einer Eigenschaft „TestCommand“. Diese soll einfach nur eine Konsolenausgebe machen und soll immer ausführbar sein.

```
Public Class TestClass

    Public Sub New()

    End Sub

    Public Property TestCommand As ICommand

End Class
```

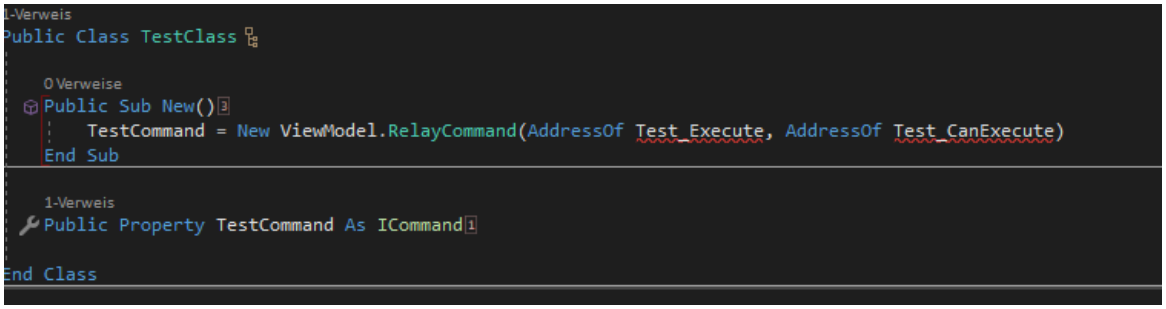
Vielleicht fragt Ihr euch warum ich als Typ für die Eigenschaft

ICommand angebe und nicht RelayCommand. Könnte ich auch, richtig. Das es allerdings ja sein könnte das ich mal mehrere verschiedene Commandklassen habe weil ich in eine Klasse evtl. zusätzliche Features implementieren möchte gebe ich den Typ an welchen alle diese Klassen gemeinsam haben, und das ist die Schnittstelle ICommand.

Im Konstruktor setze ich nun den Command auf eine neue Instanz unserer RelayCommand-Klasse und gebe über den ersten Konstruktor der Klasse die zwei Methodennamen an (welche noch nicht existieren):

```
Public Sub New()
    TestCommand = New ViewModel.RelayCommand(AddressOf Test_Execute, AddressOf Test_CanExecute)
End Sub
```

Da die beiden Methoden noch nicht existieren werden diese vom Compiler angemeckert. Siehe Screenshot:



Wenn wir nun den Cursor in eine der beiden Methodennamen setzen und STRG + . drücken bekommen wir die richtigen Vorschläge und uns die Methoden von VisualStudio erzeugen zu lassen.

Anschließend noch ein wenig einfachen Code in die Methoden und nun sieht unsere Klasse wir folgt aus:

```
Public Class TestClass

    Public Sub New()
        TestCommand = New ViewModel.RelayCommand(AddressOf Test_Execute, AddressOf Test_CanExecute)
    End Sub

    Public Property TestCommand As ICommand
    Private Function Test_CanExecute(ByVal obj As Object) As Boolean
        Return True
    End Function
    Private Sub Test_Execute(ByVal obj As Object)
        Debug.WriteLine("Der Command wurde ausgeführt!")
    End Sub

End Class
```

Erstellen wir uns schnell ein Window welches diese Klasse als DatenKontext erhält und drücken auf einen Button welcher auf diesen Command gebunden ist.

```
<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:_2._1._8._7_RelayCommand_Demo"
        mc:Ignorable="d"
        Title="MainWindow" Height="131.818" Width="201.515">
    <Window.DataContext>
        <local:TestClass/>
    </Window.DataContext>
    <StackPanel Margin="10">
        <Button Content="Klick mich"
                Command="{Binding TestCommand}"/>
    </StackPanel>
</Window>
```

Wenn wir dieses Projekt nun starten und auf den Button Klicken sehen wir folgendes in der Ausgabe von VisualStudio:

```
Der Command wurde ausgeführt!
```

Ein paar weitere Verbesserungen der Klasse und Tipps findet Ihr in dem Video zu diesem Kapitel. Es lohnt sich.

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

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/