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 das 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 "Deisgnersprache". 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.

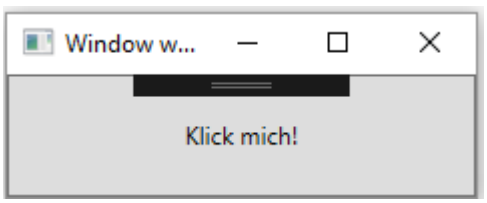
Die Darstellung passiert als Text über XML Dateien welche die Erweiterung .xaml aufweisen. Es kann mit jeder Codierung gearbeitet werden, typisch ist jedoch die Codierung als UTF-8.

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

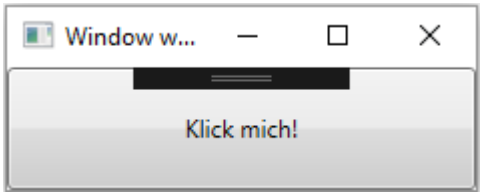
```
<Window xml:Class="MainWindow"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   Title="Window with Button"
   Width="250" Height="100">
<!-- Button hinzufügen -->
  <Button Name="button">Klick mich!</Button>
</Window>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

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 das kein Code Behind verwendet werden soll und das 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 umgestalten 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>
```

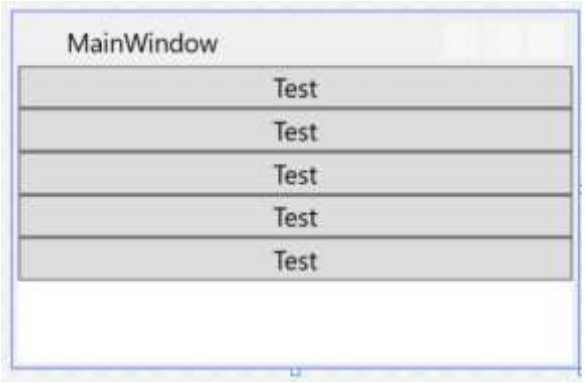
Hier wird beim ersten Button für die Hintergrundfarbe ein Farbverlauf gewählt.
Dieser Farbverlauf geht von oben nach unten von Rot nach Blau. Und zwar gleichmäßig.
Über das Offset könnte man hier Einfluss auf die "Geschwindigkeit" des Verlaufs nehmen.

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:



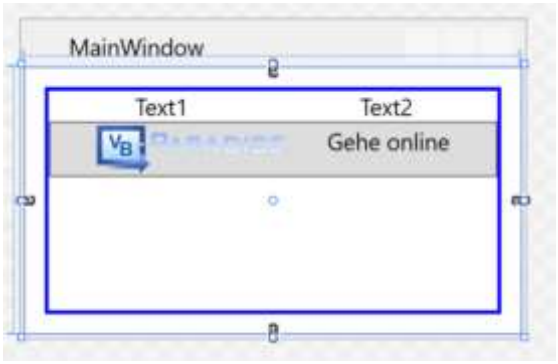
Und da kommen wir gleich zum nächsten, der Auflistungssyntax:
Es gibt Container welche mehr als ein Control als Inhalt bekommen können. z.B., Grid, UniformGrid, alle Arten von Panels usw.

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 herantasten. 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 HirachicalDataTemplates 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.

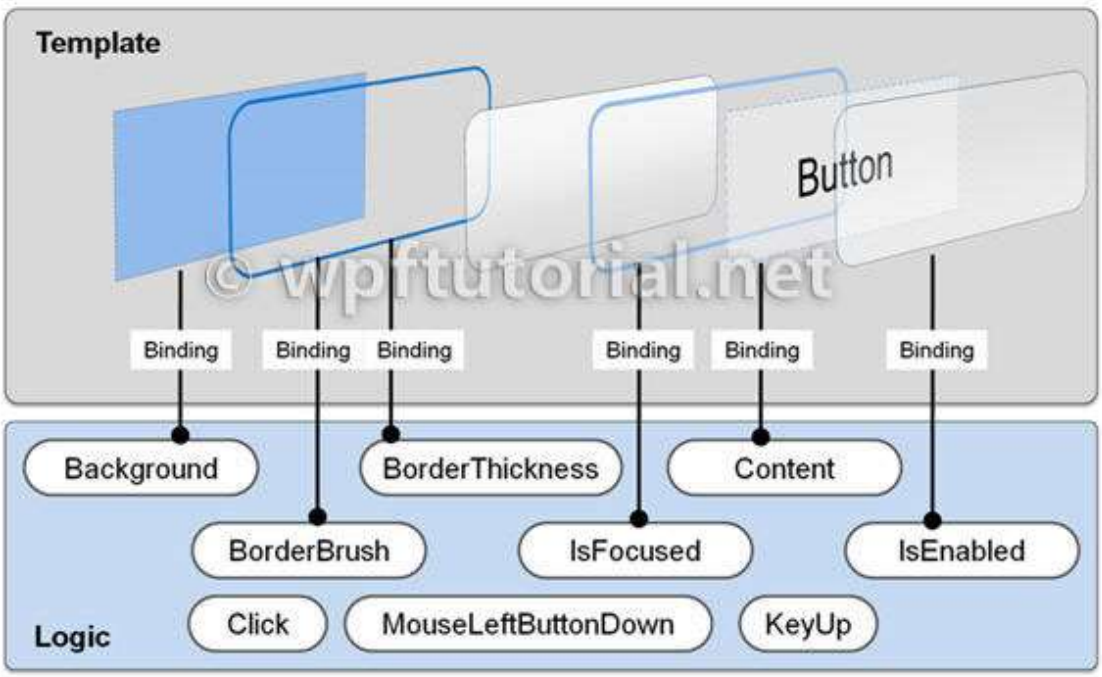
Fangen wir mit den ControlTemplates an.

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“.

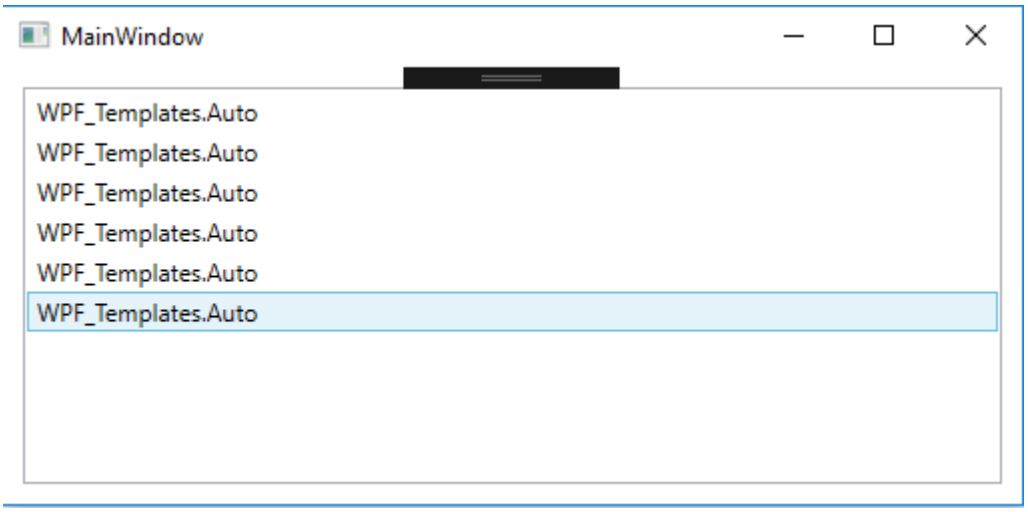
```
Public Class Auto

    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 Auto`s 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 (ist die WPF ne Frau?) 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
```

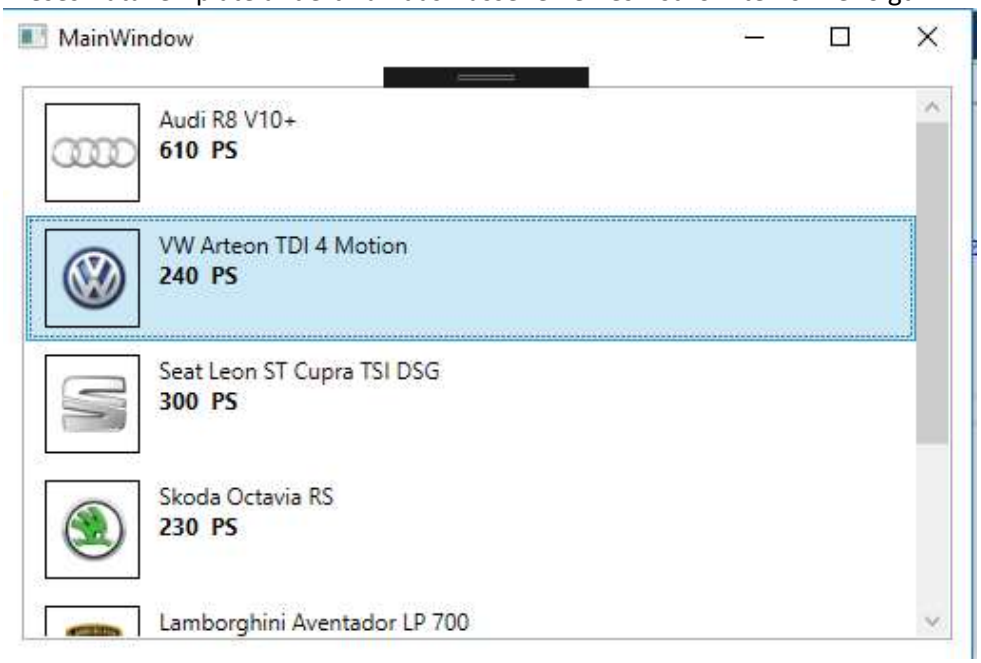
Was uns folgendes Ergebnis Liefert:

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. Jeder 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 das 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>
```

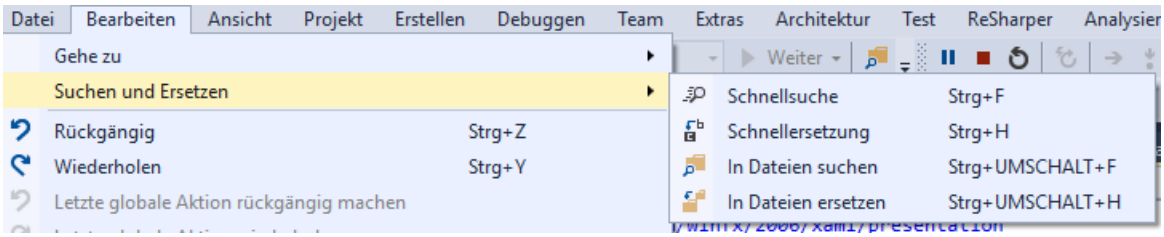
Hierdurch erhalten wir wieder die Optik unseres Items aber in der Anordnung wie wir es im ItemsPanelTemplate vorgegeben hatten.

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:

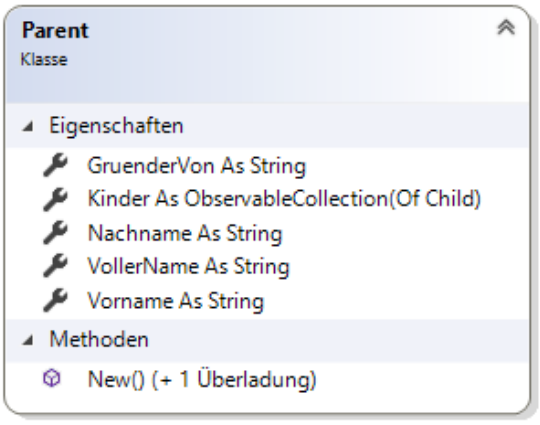


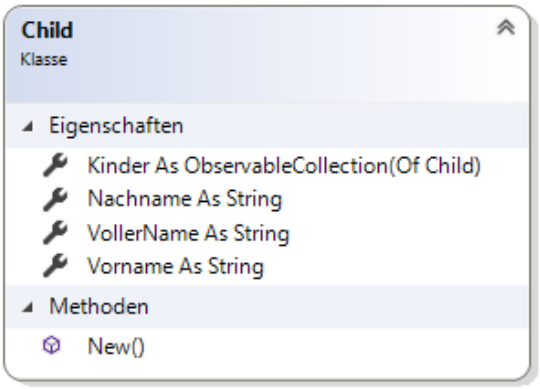
Erstmal haben wir die erste Ebene (Root) mit „Datei“, „Bearbeiten“, Ansicht“, usw.
Danach haben wir eine Unbestimmte Anzahl an Ebenen welche wieder eine unbestimmte Anzahl an weiteren Kind-Ebenen haben kann.

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.

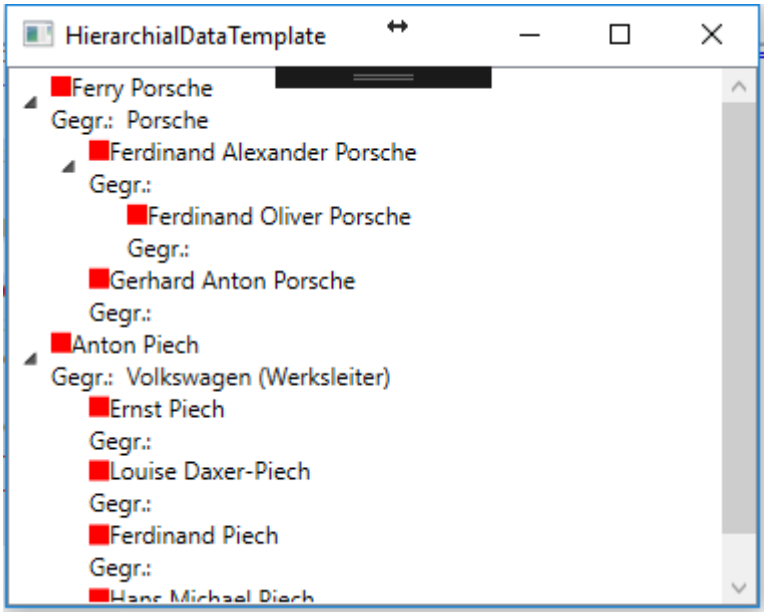
Wir erstellen ein TreeView und geben ein HierarchialDataTemplate an.

```
<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 ein 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.

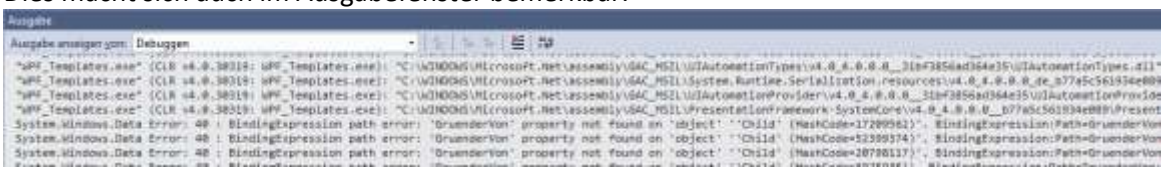
Folgende Ansicht wird in diesem Fall generiert:



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:



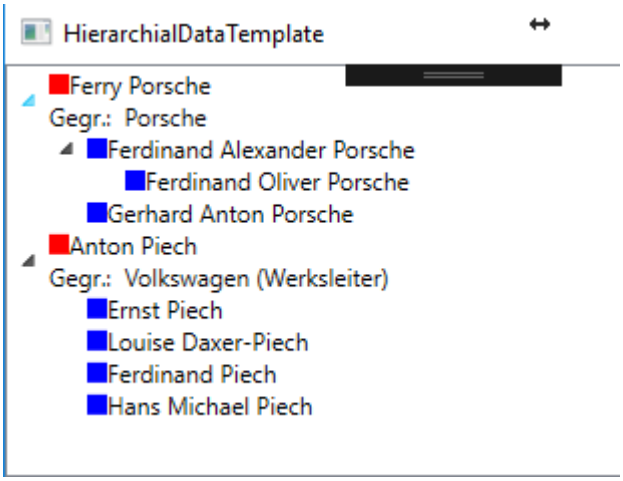
OK, wir wissen also das wir nicht am Holzweg sind, aber es hier noch Luft nach oben gibt, es muss ja möglich sein für jede Art von Element eine eigene Optik zu definieren.
Naja, vorher hatten wir gerade noch die DataTemplates gelernt, das wäre doch eine Möglichkeit gleich das gelernte anhand dieses Beispiels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

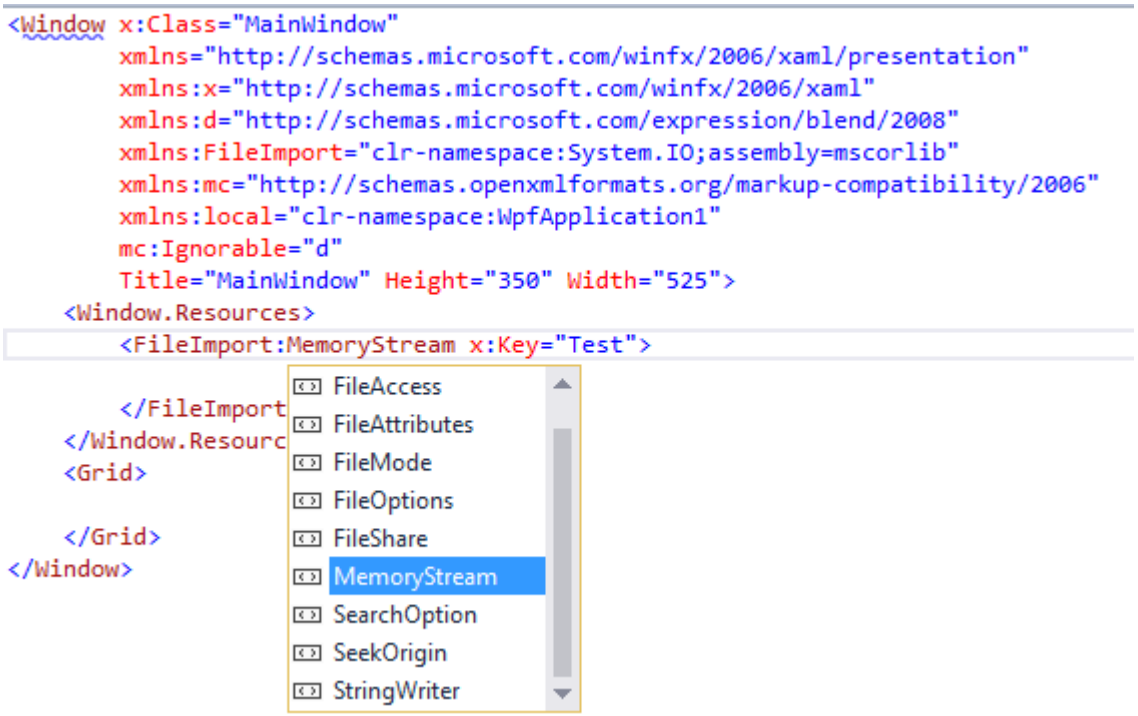
```
Imports FileImport = System.IO

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 keinen 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 einen 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.

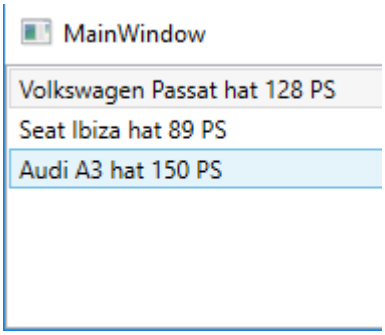
```
    <Grid>
        <ListBox ItemsSource="{Binding Fahrzeuge}" SelectedIndex="0"/>
    </Grid>
```

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 DesigntTime-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.

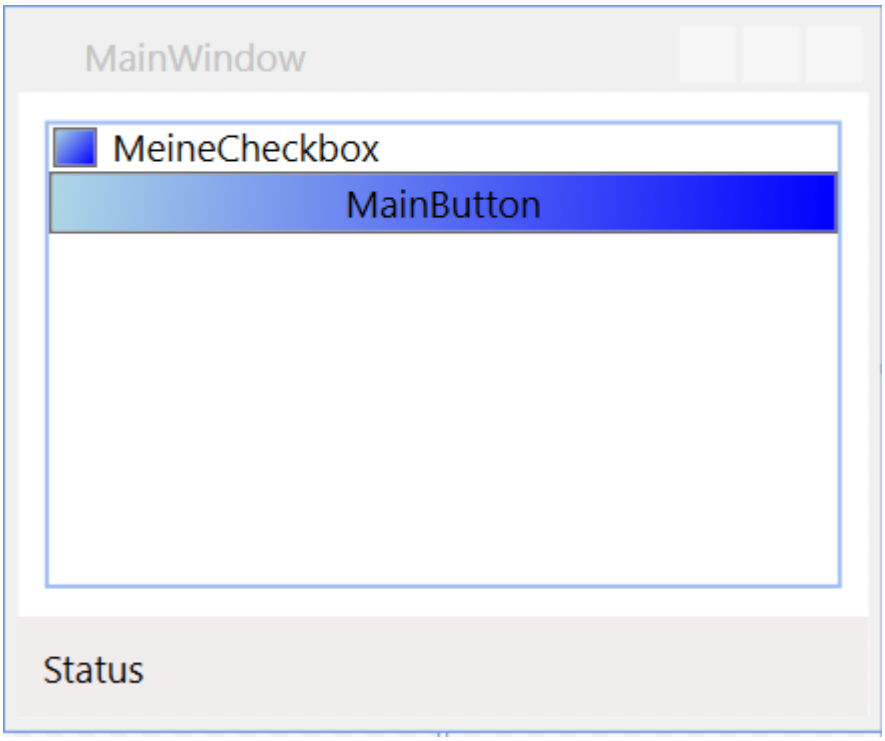
Hier ist interessant zu wissen das Resourcen "vererbt" werden. Vererben ist zwar nicht ganz das richtige Wort jedoch kann man sich darunter schon mal ungefähr etwas vorstellen.

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.

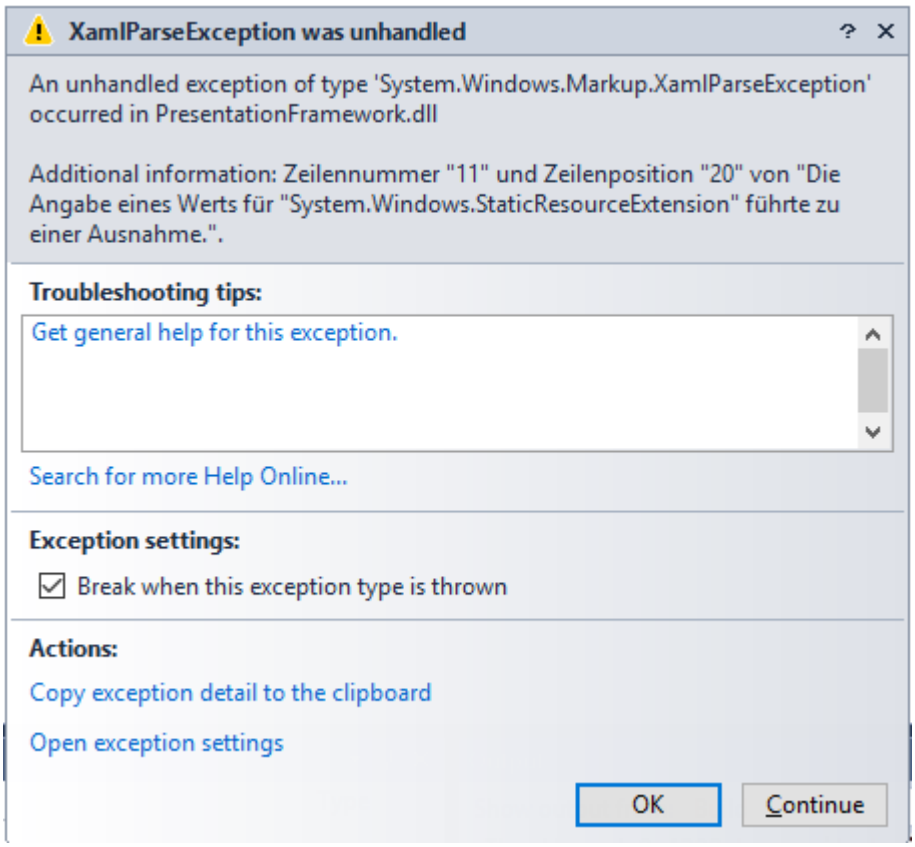
```
Panel LastChildFill="True">
StatusBar DockPanel.Dock="Bottom">
   <Label Content="Status" Background="{StaticResource MyTestBackgroundBrush}"/>
/StatusBar>
StackPanel Margin="10">
   <StackPanel.Resources>
```

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.

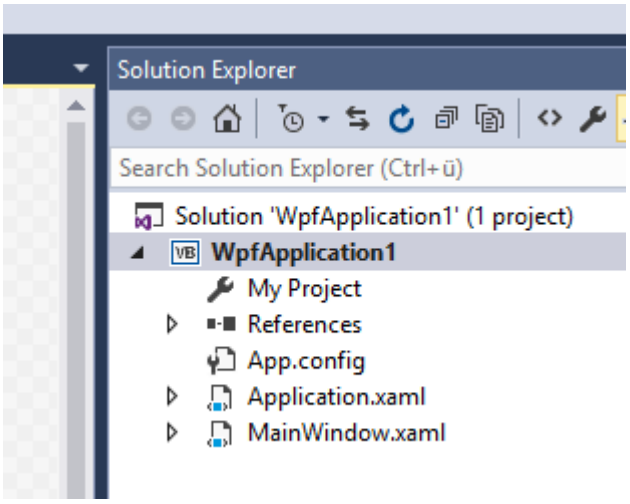
Also wo könnten wir nun die Resource definieren um diese sowohl in den Controls des Stackpanels als auch in den Controls der Statusbar zur Verfügung zu haben?

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!

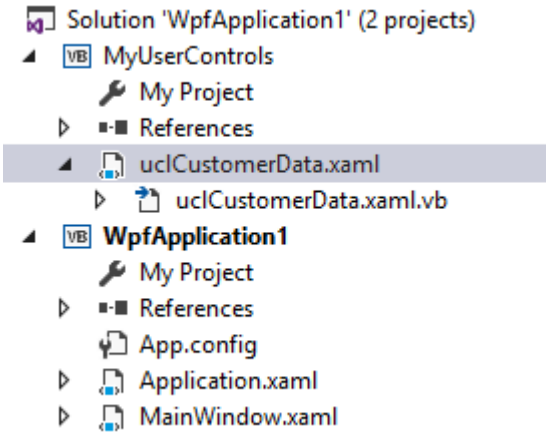
"Für dieses Projekt" würde doch bedeuten das, wenn ich Beispielsweise Controls oder Usercontrols in eine DLL auslagere diese das Margin nicht übernehmen richtig?

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.

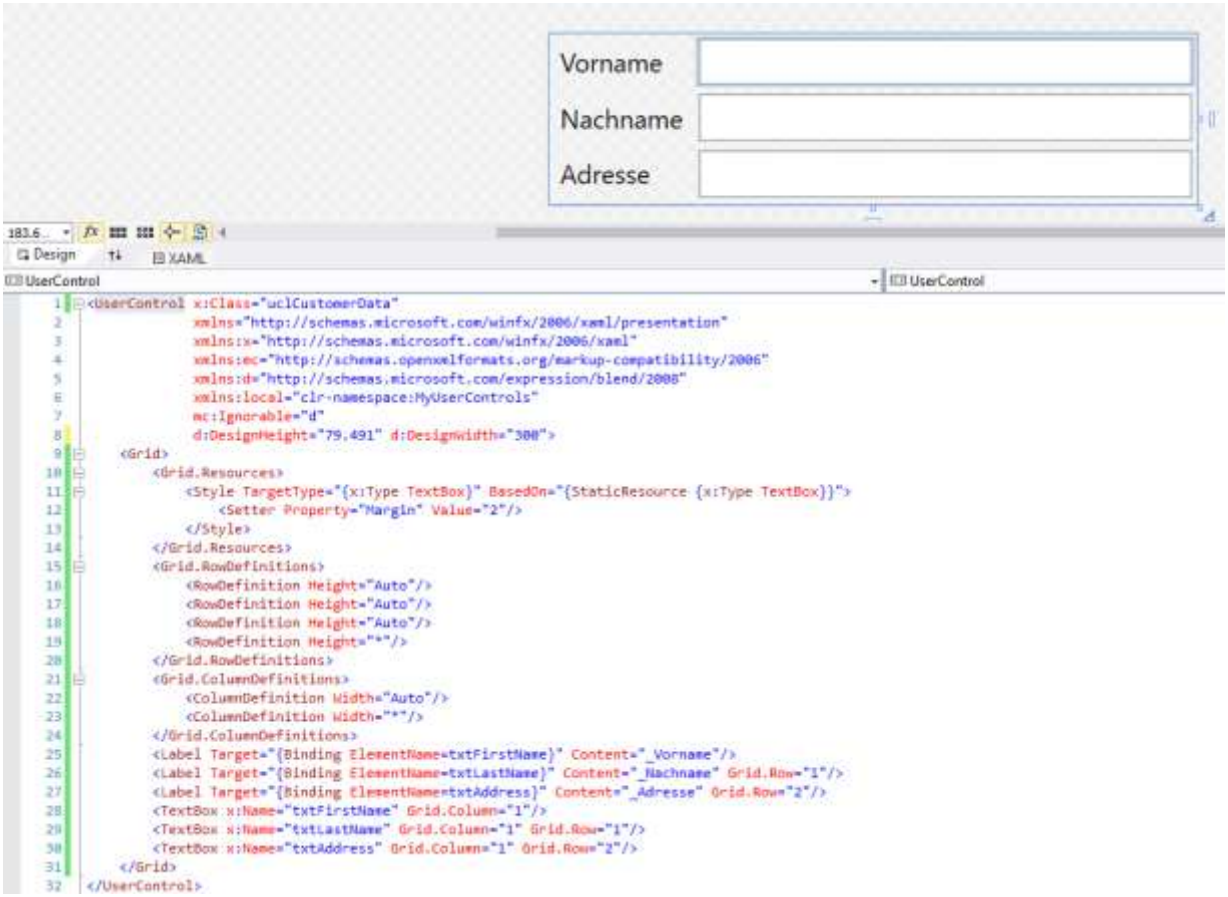
Folgende Vorgehensweise zeigt das Verhalten wie ich finde ganz gut.

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 UserControl mit dem Namen "uclCustomerData".

Hier die Projektmappe wie diese nun aussieht:



Und hier unser UserControl:

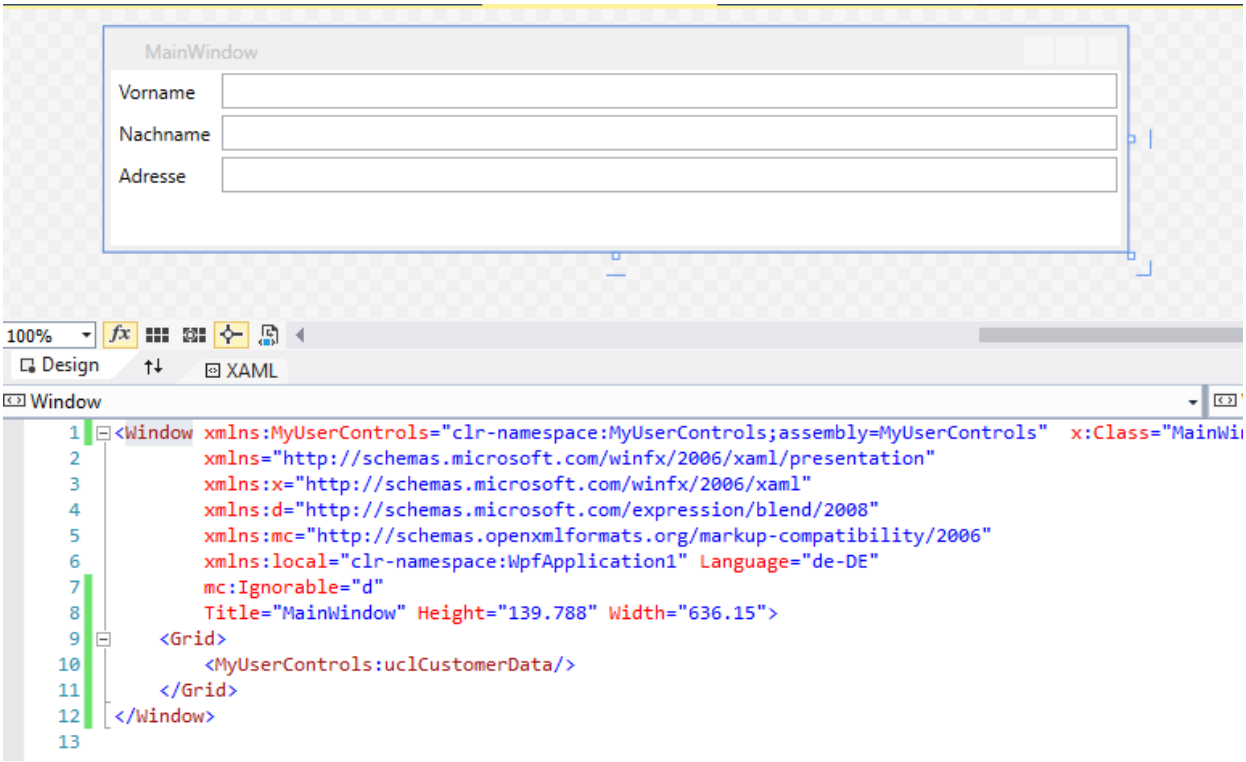


Einigen wird nun sicher auffallen das hier kein Binding auf das Text-Property der TextBoxen erfolgt ist. Die Eingegebenen Daten können also nie irgendwohin übertragen werden. Da es hier in diesem Kapitel allerdings rein um Resourcen geht wollte ich darauf jetzt verzichten.

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:

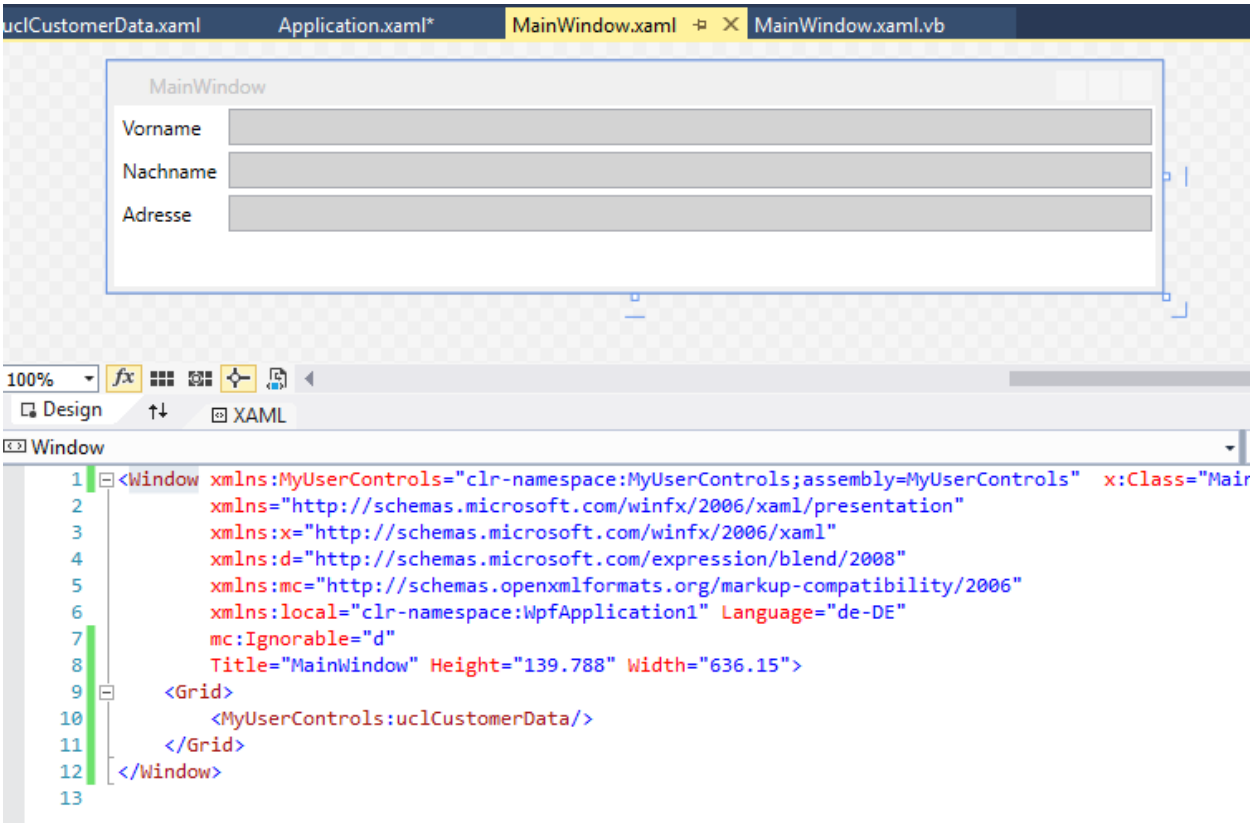



```
1  <Window xmlns:MyUserControls="clr-namespace:MyUserControls;assembly=MyUserControls"  x:Class="MainWi
2          xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
3          xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
4          xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
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.

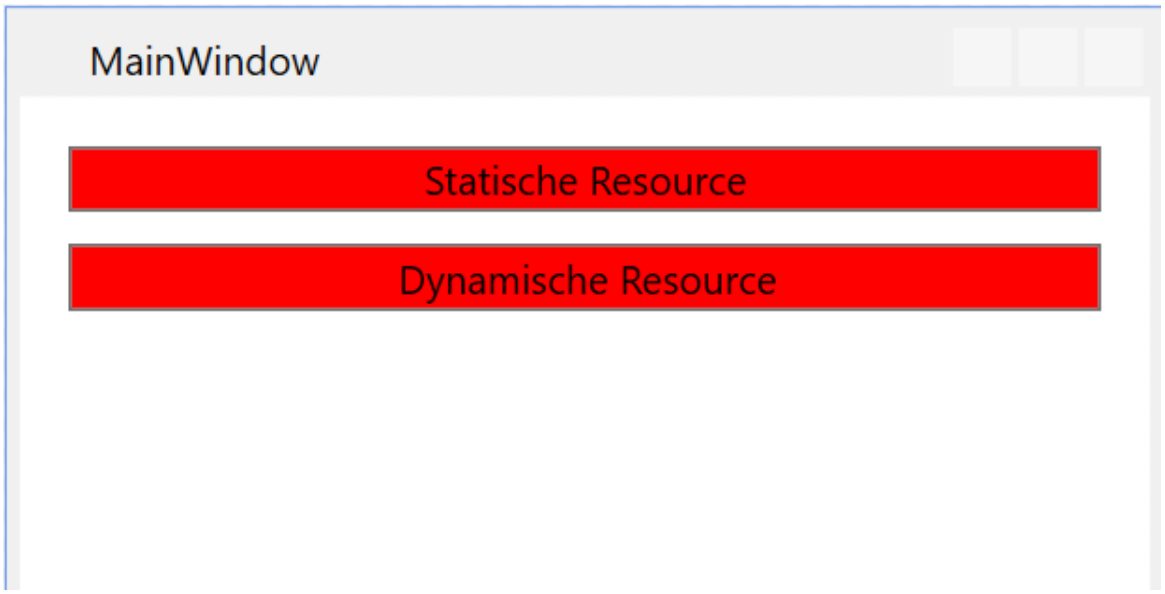
Sehen wir uns das ganze mal anhand eines simplen Beispiels an.
Wir definieren in einer Window-Resource eine Color und einen SolidColorBrush.

```
<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>
```

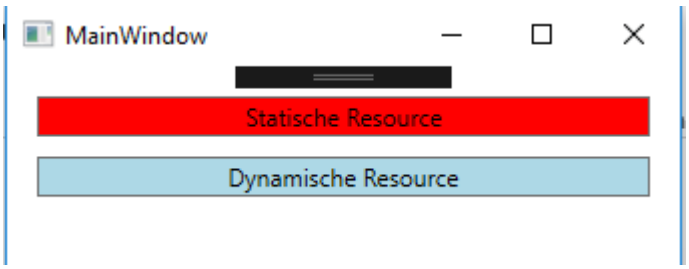
Demnach sehen beide Buttons - bis auf den Content – gleich aus.



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

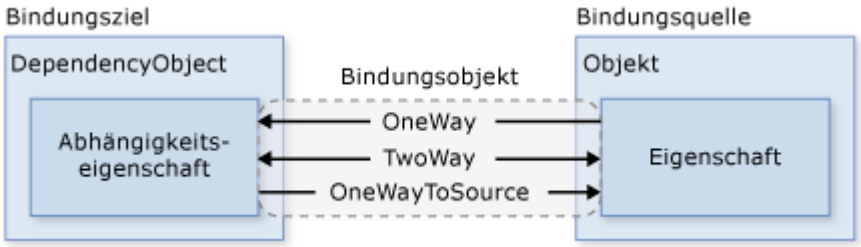
Durch das Binden wird eine Verbindung zwischen der UI und der Geschäftslogik hergestellt. Durch ein Benachrichtigungssystem werden Daten zwischen einer Logik (z.b. einer Klasse) und dem UI (z.b. eine TextBox) hin und her synchronisiert. Beispielsweise ist das Text-Property einer TextBox auf ein String-Property 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 führ 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**

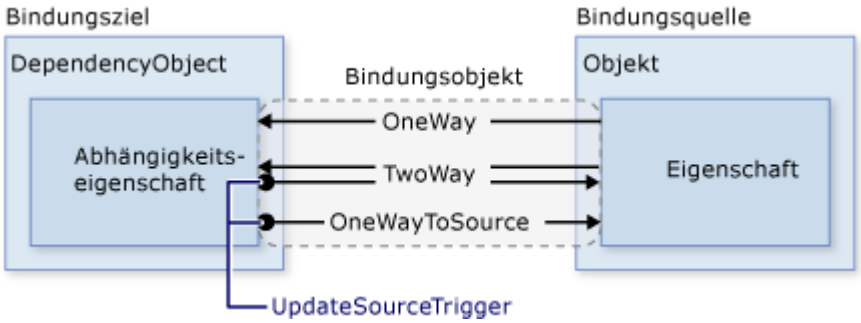
  Hier wird ein OneWay Binding erstellt welches allerdings nur beim initialisieren des UI synchronisiert wird. Änderungen werden später nicht mehr aktualisiert. Im Grunde eine Momentaufnahme.

- **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.

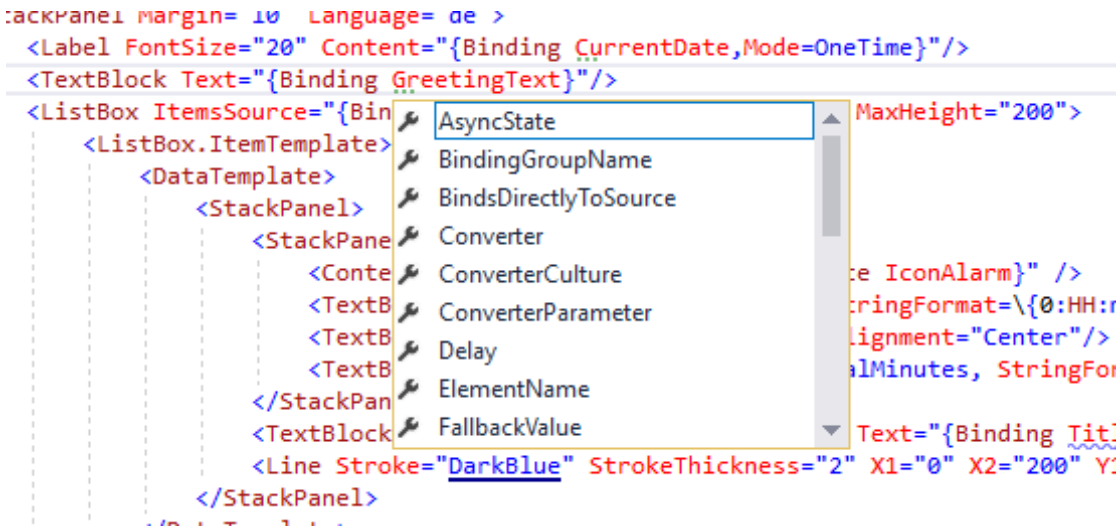
Der DesignTime-Mode macht es euch möglich hier um einiges produktiver zu sein. Nicht nur das Ihr plötzlich Intellisense zu Verfügung habt, ihr könnt auch Beispieldaten laden lassen um Anhand „echter“ Daten das verhalten eurer Steuerelemente oder des ganzen Views zu sehen und dementsprechend abzuändern.

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.

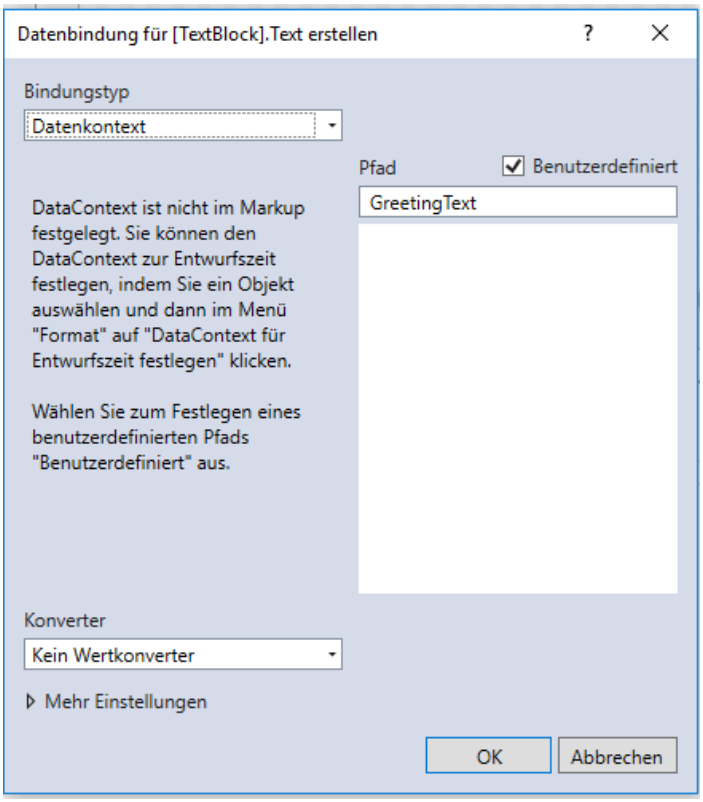
Nehmen wir nun Beispielsweise mal die TextBox welche wir auf den Begrüßungstext GreetingsText binden möchten:

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

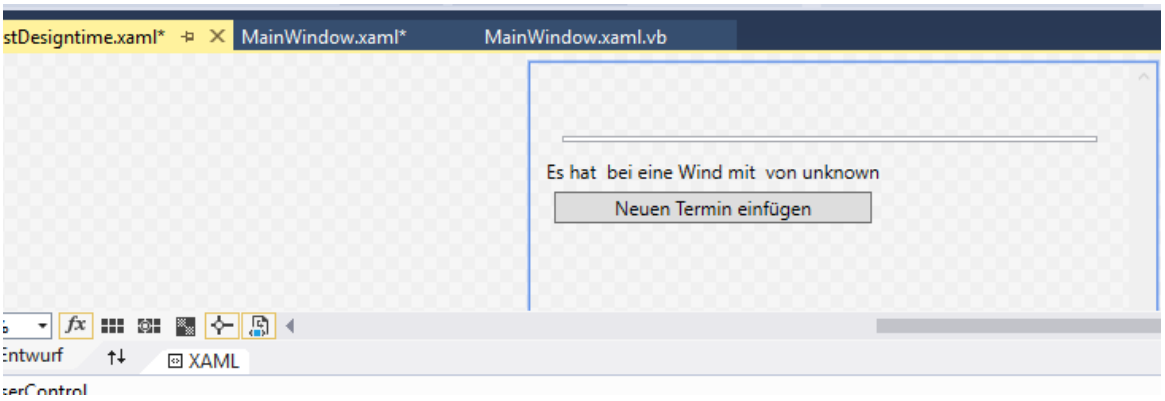
Auch über den sogenannten „Binding-Picker“ über das Eigenschaftenfenster des jeweiligen Controls haben wir nur die Möglichkeit manuell ein Binding zu setzen aber der Designer weis nicht welche Properties die Klasse hat. Wie auch, er weis nicht nicht mal von welcher Klasse wir überhaupt reden.



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>
```

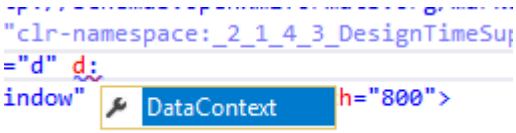
Hier ein normales Fenster mit einem Default-Namespace und vier benannten.

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

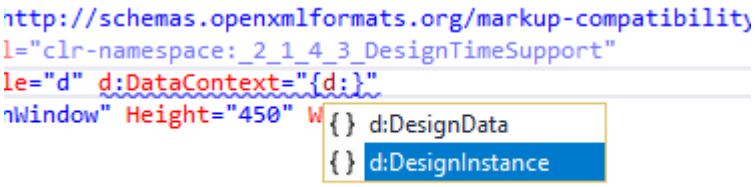
```
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
```

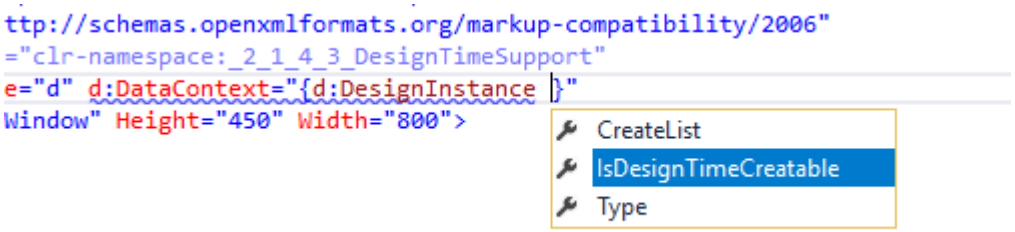
sowie noch den mit „x“ benannten Namespace verwenden und müssen uns noch einen eigenen hinzuholen da wir den Namespace in welchem unsere DayInfo-Klasse liegt auch noch benötigen um Zugriff darauf zu haben.

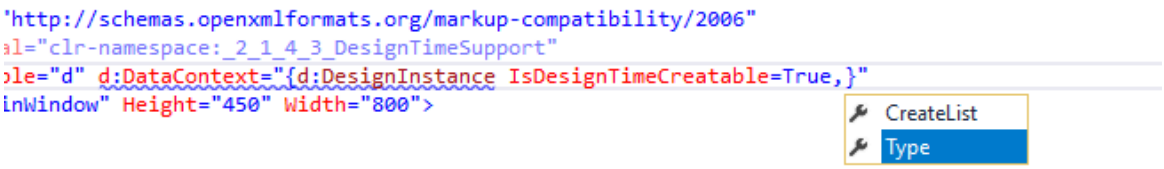
Innerhalb des „d“ Namespaces haben wir Zugriff auf ein Property mit dem Namen „DataContext“:



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.

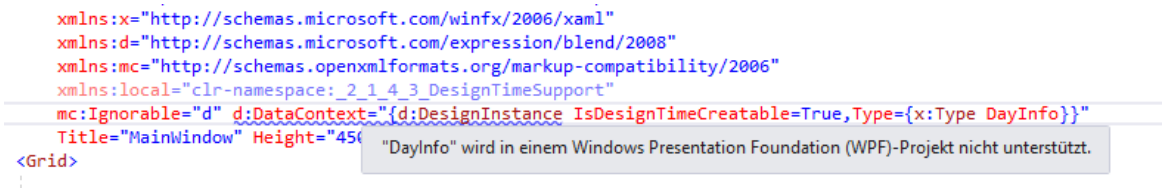






Sind wir fertig mit unserer Eingabe meckert die IDE allerdings das „DayInfo“ in einem WPF Projekt nicht unterstützt wird. Schade, WPF kann kein DayInfo. Wie jetzt?

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.

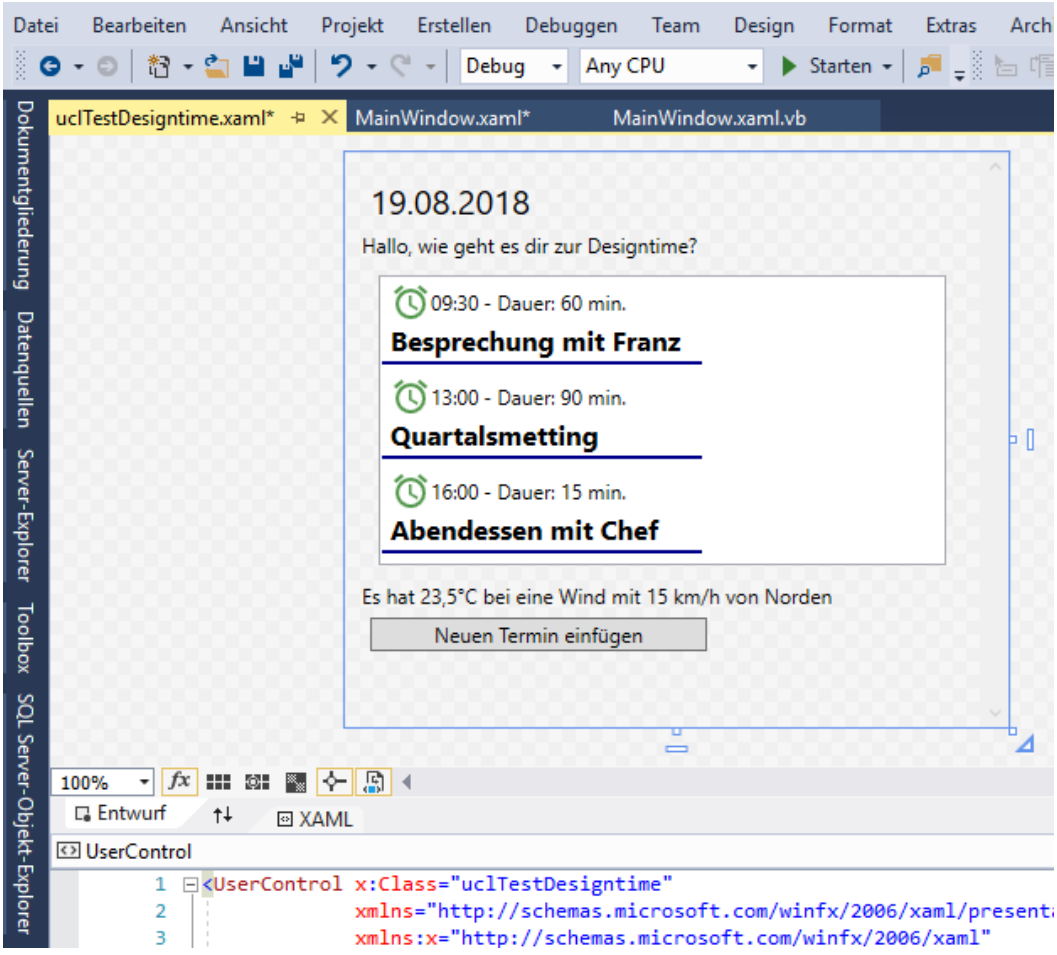


Nun haben wir unseren Namespace importiert und die IDE meckert auch nicht mehr:

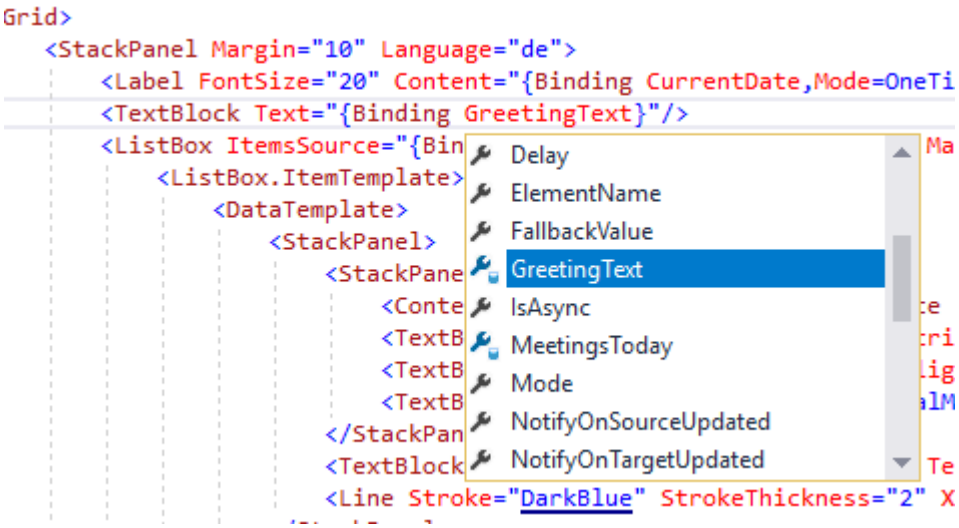
Genau das machen wir nun mit unserem UserControl welches nun wie folgt (gekürzte Ansicht) aussieht:

```
<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.

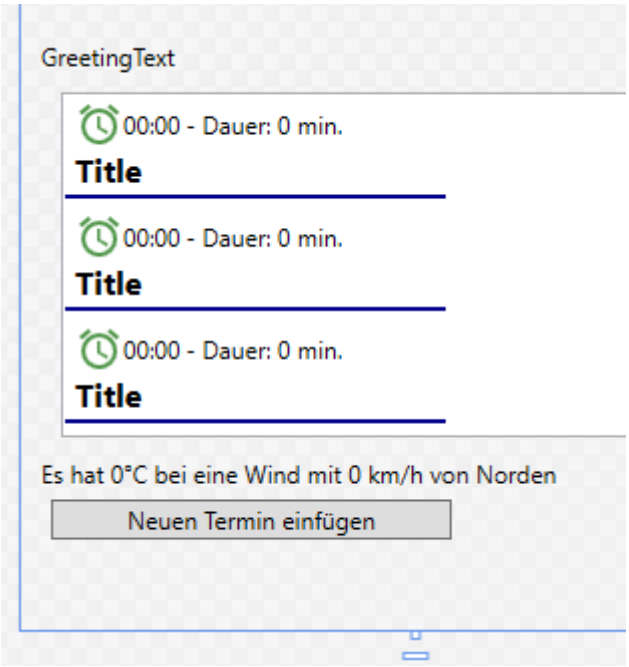
Achtung: Wenn echte Beispieldaten verwendet werden sollen MUSS die Klasse einen parameterlosen Konstruktor aufweisen da dies sonst der Designer mit einer entsprechenden Fehlermeldung quittiert.

Folgenden Code habe ich im parameterlosen Konstruktor der Klasse:

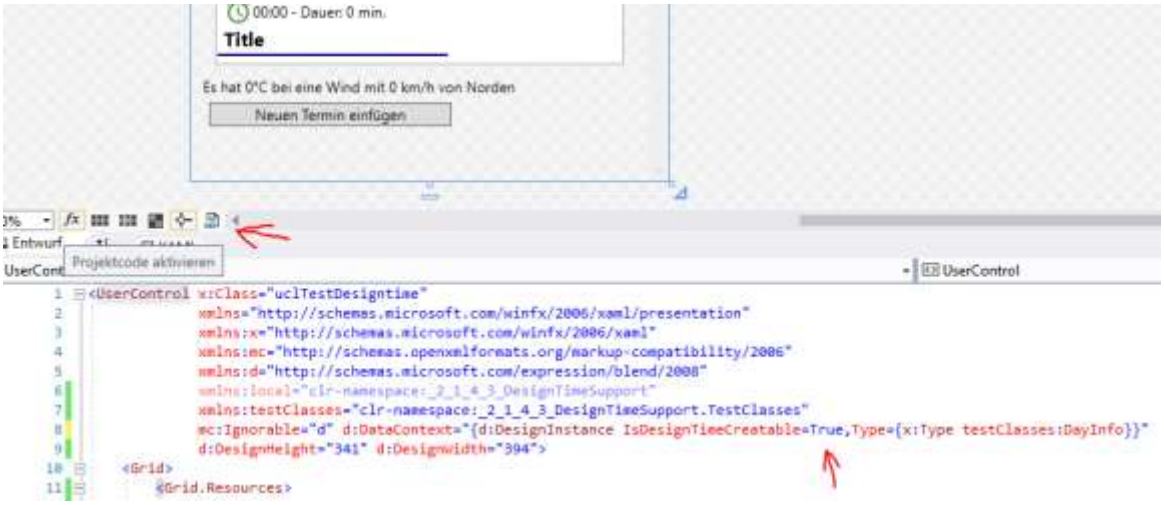
```
    Public Sub New()
            If DesignerProperties.GetIsInDesignMode(New DependencyObject) Then
                GreetingText = "Hallo, wie geht es dir zur Designtime?"
                CurrentDate = DateTime.Today()

                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:



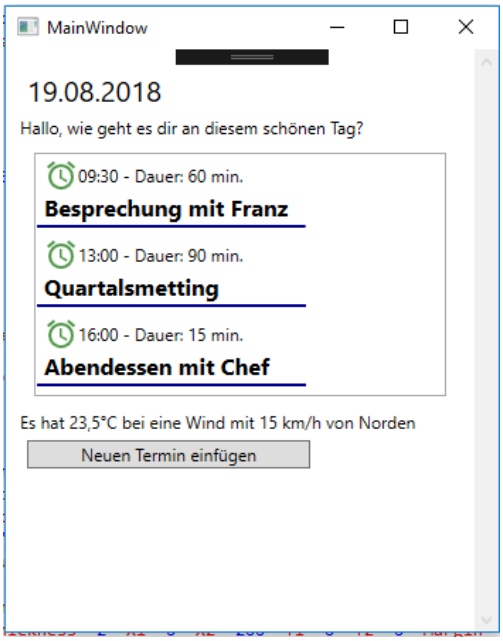
So kann man selbst entscheiden ob man die Beispieldaten generiert oder ob man mit der automatischen Generierung zufrieden ist.

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.