

TUTORIALREIHE WPF LERNEN

WPF verständlich und anhand von Praxisbeispielen erklärt

Sascha-Heinz Patschka

Inhaltsverzeichnis (inkrementell)

1. Vorstellung

1.1. Einleitung

1.2. Aufbau dieser Tutorialreihe

2. Grundlagen der WPF

2.1. Einführung in die WPF

2.1.1. Die wichtigsten Controls und deren Verwendung

2.1.1.1. Videocast: Die wichtigsten Controls und ihr Verhalten

2.1.1.2. Videocast: Styles, Templates und Trigger

2.1.1.3. Videocast: Controls eine neue Optik verpassen

2.1.2. XAML Namespaces

2.1.2.1. Kurze Theorie

2.1.2.2. Erstellung eigener Namespaces

2.1.3. Ressourcen

2.1.3.1. Was sind Ressourcen, was bringen sie mir

2.1.3.2. Unterschied StaticResource und DynamicResource

2.1.4. Binding und das Bindingsystem

2.1.4.1. Was ist DataBinding? Das Konzept dahinter + Videocast

2.1.4.2. Binding anhand einfacher Beispiele und Klassen

2.1.4.3. DesignTime-Support für Binding

2.1.4.4. Binding über Converter

2.1.4.5. Binding über DataTemplates

2.1.4.6. Binding an Collections. Warum ICollectionViewSource? Collections Filter, Sortieren, Gruppieren ohne viel Aufwand

2.1.4.7. Validierung von Benutzereingaben

2.1.4.8. Rücksicht nehmen auf die aktuelle Culture

2.1.5. DependencyProperties

- 2.1.5.1. *Was sind Dependency Properties und wie unterscheiden sie sich von normalen Properties*
 - 2.1.5.2. *Eigene Dependency Properties implementieren*
 - 2.1.6. *Markuperweiterungen*
 - 2.1.6.1. *Kurze Theorie*
 - 2.1.6.2. *Beispiele Anhand von Ressourcen und Styles*
 - 2.1.7. *Attached Properties*
 - 2.1.7.1. *Kurze Theorie (wozu Attached Properties)*
 - 2.1.7.2. *Beispiele Anhand vom Grid und dem DockPanel*
 - 2.1.7.3. *Eigene Attached Properties erstellen ;-)*
 - 2.1.8. *Attached Events*
 - 2.1.8.1. *Wir gehen gleich in die Praxis, gibt nicht viel zu sagen*
 - 2.1.9. *Inputs und Command*
 - 2.1.9.1. *Die Input-API*
 - 2.1.9.2. *Tastatur und Mausklassen*
 - 2.1.9.3. *Eventrouting (Direct, Bubbling, Tunneling)*
 - 2.1.9.4. *Keyboard, Mouse und Textinput*
 - 2.1.9.5. *Touch und Multitouch (wird ja immer wichtiger)*
 - 2.1.9.6. *Focus (Der Unterschied zwischen Keyboardfocus und Logicalfocus)*
 - 2.1.9.7. *Commands (Integrierte und eigene)*
 - 2.1.9.8. *RelayCommand Klasse*
 - 2.1.9.9. *CommandBinding und Command-Parameter*
- 3. *Eine Telefonbuch Applikation unter WPF (ohne Binding)*
 - 3.1. *Hauptfenster erstellen und Funktionen festlegen*
 - 3.2. *Ein Primitives Telefonbuch rein mit CodeBehind ala WinForms*
 - 3.3. *Fazit*
- 4. *Eine Telefonbuch Applikation unter WPF (mit Binding der CodeBehind)*
 - 4.1. *Umbau von Telefonbuch aus Kapitel 3.2 auf einfachstes Binding*
 - 4.2. *Fazit – Was wird besser, was schlechter*
- 5. *Das MVVM Pattern*
 - 5.1. *Was ist das MVVM Pattern?*
 - 5.2. *Wann MVVM und wann nicht?*
 - 5.3. *Welchen Mehrwert kann ich aus dem Pattern gewinnen?*
 - 5.4. *MVVM und CodeBehind – verboten?*
 - 5.5. *Model – View – ViewModel – Wars das?*
 - 5.6. *Erstellen einer korrekten MVVM Projektmappe in Visual Studio*
- 6. *Unser Telefonbuch in MVVM*
 - 6.1. *Projekt anlegen und Struktur besprechen*
 - 6.2. *Das Model erstellen*

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

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 "learning 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 Diskussionsthread 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“.

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

Und nun viel Spaß mit meinen Tutorials.

1.2

Aufbau dieser Tutorialreihe

Es wird ca. 1-mal pro Woche ein Beitrag mit mindestens einem Kapitel auf www.vb-paradise.de online gestellt. Es kann vorkommen das auch mal 2 oder mehr Kapitel behandelt werden. Je nachdem wie ich dazu komme und Zeit habe. Falls es vorkommen sollte dass ich mal eine Woche auslasse entschuldige ich mich bereits im Voraus dafür, bitte habt Verständnis das ich mal in Urlaub fahre oder beruflich etwas mehr Stress habe.

Diese Tutorialreihe wird als "Hybrid" aufgebaut. Teile werden als normale Beiträge in reinem Text bzw. mit Bildern erstellt, andere Teile aber auch als Videocast.

Es wird außerdem für jedes Kapitel ein ZIP File online gestellt welches dieses Inhaltsverzeichnis und die Kapitel bis zum aktuellen Zeitpunkt enthält. Außerdem mit in dem ZIP File wenn vorhanden die VisualStudio Solution abwärtskompatibel bis Visual Studio 2015, sowie Links zu den Videos sofern vorhanden.

Sollte ein Beitrag rein als Text ohne Video erstellt worden sein wird ein PDF mit in der ZIP enthalten sein damit jeder auch offline in Ruhe alles lesen kann.

Sämtliche Verweise in den Solutions werden nur als NuGet Verweise in das jeweilige Projekt eingebunden um sicherzustellen das die Solution nach dem Download auch bei jedem läuft da NuGet automatisch nachgeladen wird wenn nicht vorhanden. Weitere Infos könnt Ihr [hier](#) nachlesen.

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

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

2.0

GRUNDLAGEN DER WPF

Wir kommen zu den Grundlagen. Die WPF bietet eine eigene "Designersprache". XAML (Extensible Application Markup Language); ist eine Markupsprache und ist ähnlich im Aufbau wie XML. Der Großteil der Benutzeroberfläche wird in der WPF mit XAML erstellt.

Was ist XAML? Im Grunde vereinfacht XAML das Erstellen einer UI einer .NET Anwendung. Es können sichtbare UI-Elemente im deklarativen XAML-Markup erstellt und anschließend die UI-definition mithilfe von Code-Behind, die über partielle Klassendefinitionen an das Markup geknüpft sind, von der Laufzeitlogik trennen.

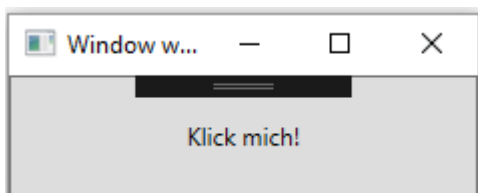
Die Darstellung passiert als Text über XML Dateien welche die Erweiterung `.xaml` aufweisen. Es kann mit jeder Codierung gearbeitet werden, typisch ist jedoch die Codierung als UTF-8.

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

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

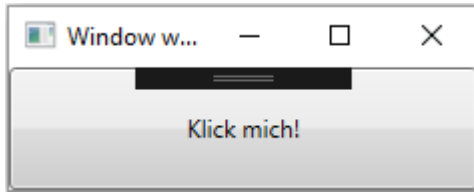
Hier werden ein Fenster und eine Schaltfläche mithilfe der Elemente Window und Button definiert. Jedes Element wird mit Attributen konfiguriert und korrespondiert mit den jeweiligen Properties des Elements. Zeile zwei und drei beinhalten die per Default eingetragenen XAML Namespaces, dazu kommen wir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

So würde dieser Code im Programm aussehen:



Unter Windows 7 so:

3



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

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

Wenn wir nun den XAML mit dem Code vergleichen fällt uns ziemlich schnell auf wie die WPF das macht.

In einem <Button/> wird z.B. `Dim myButton As New Button` erstellt. Jedes Attribut steht für ein Property der jeweiligen Klasse.

z.B. wird `myButton.Name = "button"` in XAML zu `Name="button"`

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

CODE BEHIND

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

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

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

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

Visual Studio hat uns nun den Code in der Code Behind des Windows erstellt.

Mit einem Rechtsklick auf den Code des Click Attributes können wir nun mit "Gehe zu Definition" direkt zu diesem Code springen.

Hier steht nun folgendes:

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

End Sub
```

Und dies kennen wir nun ja wieder aus WinForms.

Was anders ist, ist der RoutedEventArgs, dies erkläre ich allerdings im Kapitel RoutedEvents, da möchte ich jetzt noch nicht vorgreifen.

Ein weiteres Beispiel mit Attributen:

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

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

Allerdings gibt es auch Eigenschaften eines Objekts welche nicht über die Attributsyntax gesetzt werden da diese z.B. zu komplex sind. Auf diese kann dann über die Eigenschaftenelementsyntax zugegriffen werden.

Oft ist es aber auch Geschmacksache oder Übersichtlichkeit für welche Art man sich entscheidet. Hier ein Beispiel für das umgestaltet des oben stehenden Codes unter Verwendung der Eigenschaftenelementsyntax:

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

Dies wäre nun derselbe Button nur das die Properties über die Eigenschaftenelementsyntax gesetzt wurden.

Jetzt könnte man sich denken: "Wozu gibt es denn die Eigenschaftenelementsyntax wenn ich das mit der Attributsyntax ja auch machen kann?"

Hierfür gibt es mehrere Gründe. Einer wäre z.B. dass die Background Eigenschaft eines Buttons ja eigentlich einen [Brush](#) erwartet. Warum kann man dann `Background="Blue"` anwenden?

Die WPF stellt intern diverse [TypeConverter](#) bereit, so kann sie den String "Red" in einen SolidColorBrush mit der Color "Red" umwandeln. Das gleiche gilt für die Eigenschaft Content des Buttons welche von Typ „Object“ ist. Hier macht die WPF automatisch einen String daraus.

Aber warum soll ich die lange Variante schreiben wenn ich die kurze doch auch schreiben könnte?

In diesem Fall (SolidColorBrush) geht das noch alles über die Attributsyntax da die WPF den String ja in einen SolidColorBrush wandelt. Wenn man nun einen Farbverlauf als Hintergrund

verwenden möchte muss man aber schon auf die Eigenschaftensyntax 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.

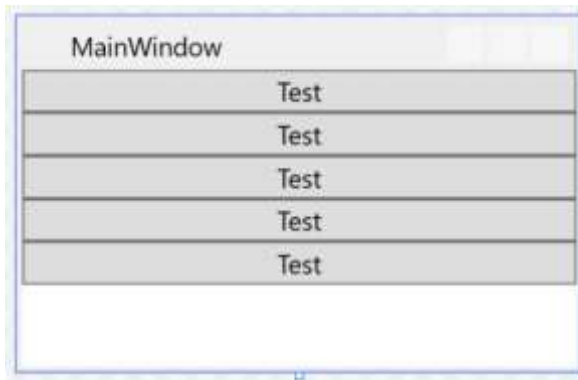
```
<StackPanel>
  <Button>Test</Button>
  <Button>
    <Button.Content>
      Test
    </Button.Content>
  </Button>
  <Button Content="Test"/>
  <Button>
    <TextBlock Text="Test"/>
  </Button>
</StackPanel>
```

```

    <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 TextBlock hinzugefügt wird.

Dem Button wird wieder ein StackPanel zugewiesen worin sich ein Image und ein Textblock horizontal aufgelistet befinden.

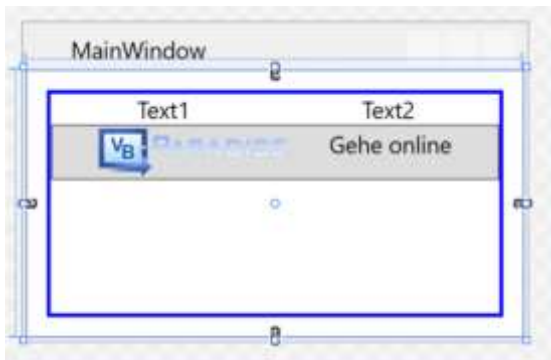
```

<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 "*learning 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](https://msdn.microsoft.com/de-de/library/ms746927(v=vs.100).aspx)

Gut in verschiedene Kategorien unterteilt und mit vielen Beispielen. Habe diese Pages damals mehrfach gelesen.

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

2.1

EINFÜHRUNG IN DIE WPF

In diesem Kapitel werden wir ein paar Beispiele durchgehen. Das Hauptaugenmerk wird hier auf XAML gelegt. Wir werden die wichtigsten Controls kennenlernen, wie wir diese Anwenden und auch untereinander kombinieren können um somit ein völlig neues Aussehen und zum Teil neues Verhalten über Trigger und Animations in das Control bekommen. Außerdem werden wir lernen wie wir ein Fenster so gestalten 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 geschriebene 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 angeben.

```

<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-Resources 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 [HierarchicalDataTemplates](#) 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.

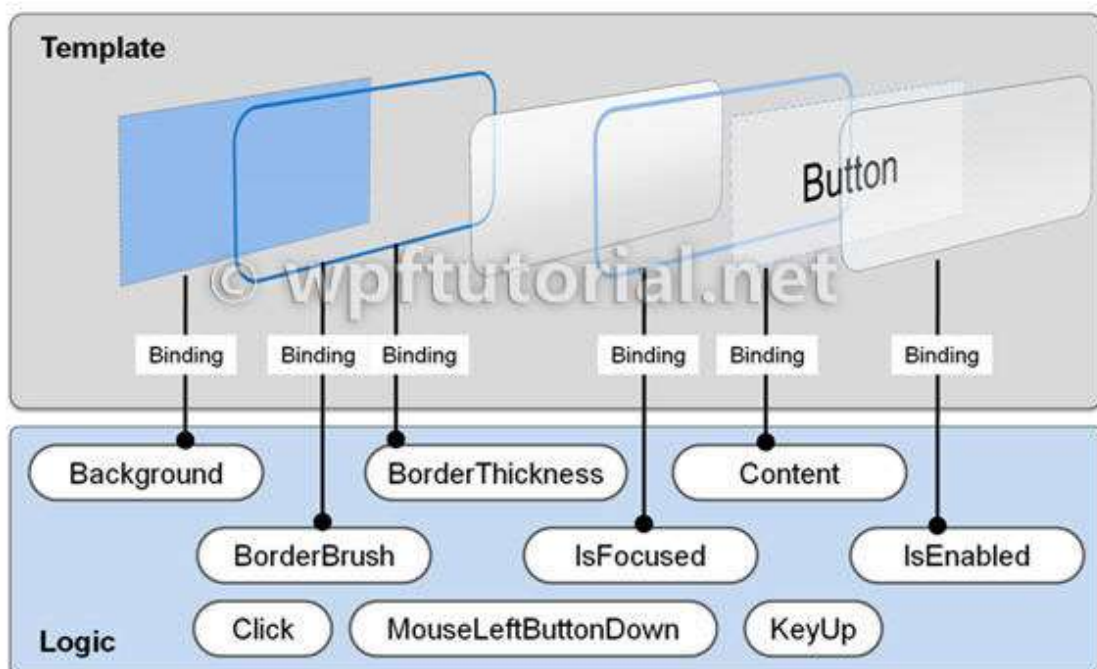
Die Verbindung zwischen dieser Logik und dem Template passiert über Binding.

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 erfahen 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 Ressourcen von darüber liegenden Controls, bis in der Hierarchie nichts mehr vorhanden ist, dann sucht sie in der `Application.xaml` nach einem `DataTemplate`.

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

```
Public Class Auto

    Public Sub New()
    End Sub

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

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

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

End Class
```

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

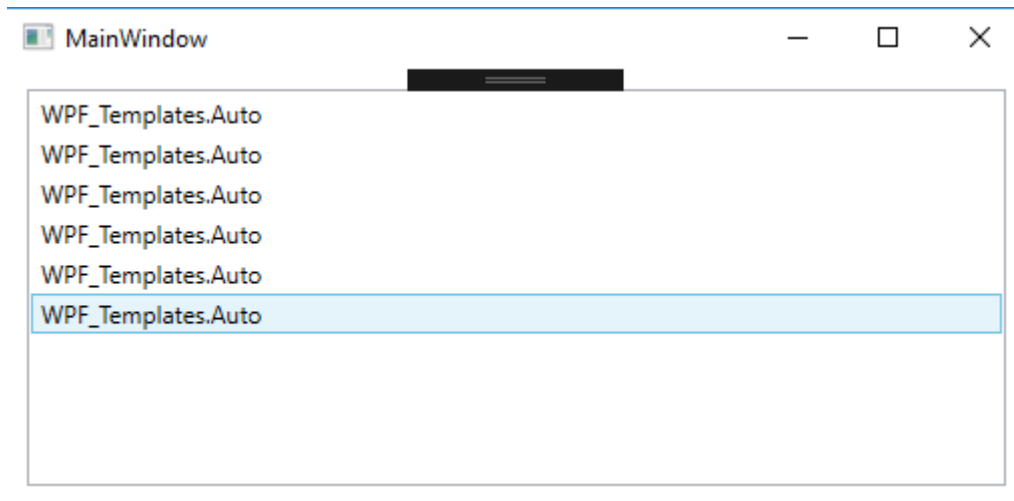
```
Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles
Me.Loaded
    AutoListe = New ObservableCollection(Of Auto)
    With AutoListe
        .Add(New Auto("Audi", "R8 V10+", 610, "BrandImages/Audi.png"))
        .Add(New Auto("VW", "Arteon TDI 4 Motion", 240, "BrandImages/Vw.png"))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

    Me.DataContext = Me
End Sub

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

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

Hier eine ListBox gebunden an *Autos* ohne *DataTemplate*.

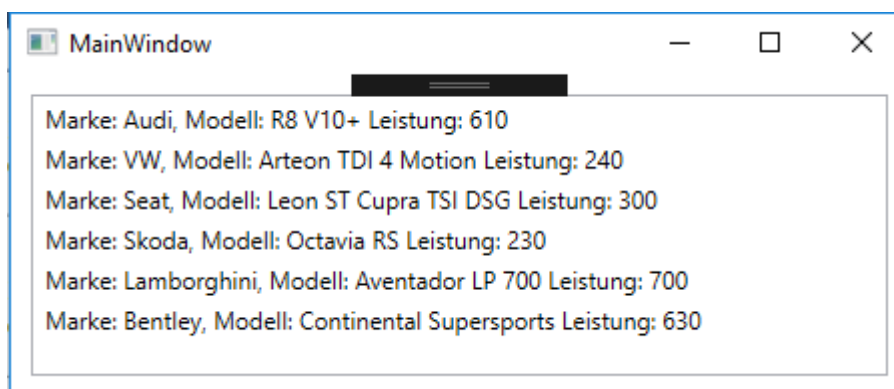


Nicht so schön. Wieder schlägt der [TypeConverter](#) der WPF zu. Die WPF wirft keinen Fehler sondern versucht die Daten welche ihr übergeben werden einfach irgendwie zu Rendern. Was liegt näher als `.ToString` aufzurufen. Und genau das macht sie.

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

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

Was uns folgendes Ergebnis Liefert:



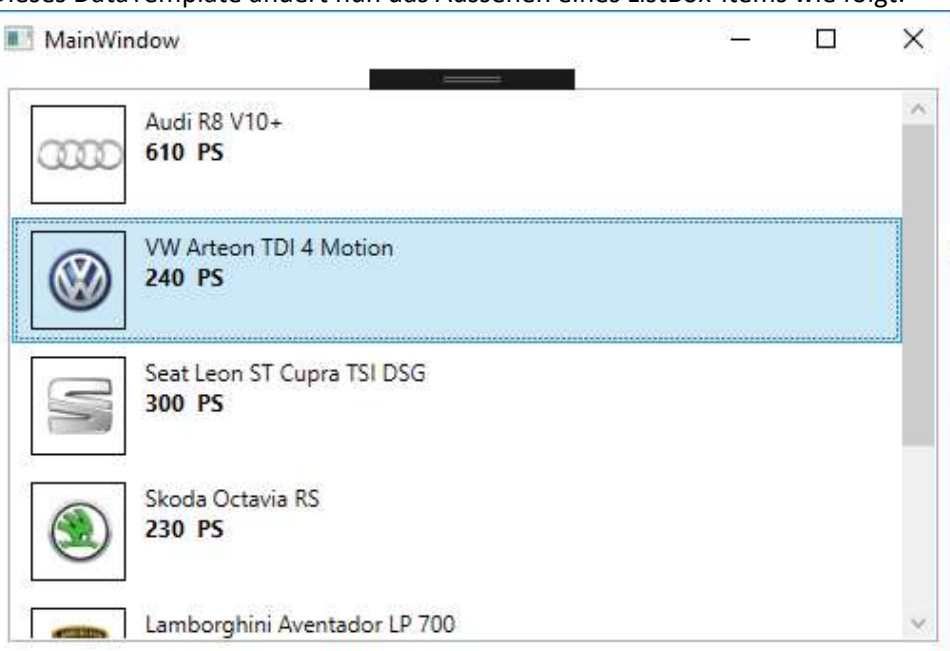
OK, schon um einiges besser, aber lange noch nicht zufriedenstellend. Wir befinden uns in der WPF, das geht ja mal sicher viel besser und schöner oder?

Richtig. Mittels DataTemplate kann man nun bestimmen wie das Template für ein ListItem aussehen soll. Template = Vorlage. Also erstellen wir eine Vorlage und geben der WPF die Info für was (Die Klasse Auto) dieses Template verwendet werden soll.

Hier ein DataTemplate direkt in das ItemTemplate der ListBox:

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

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



Mehr im Video weiter unten, aber jetzt gehen wir mal weiter zu dem [ItemsPanelTemplates](#).

ItemsPanelTemplates

Diese Art von Templates erlaubt es uns das Layout, wie Items eines ItemControls wie z.B. einer ListBox zu bestimmen. Jedes ItemControl 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 Ressourcen spart und stark auffällt wenn man mehrere tausend Items in der Liste hat.

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

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

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

Mit dem Codeschnipsel von oben würde nun folgendes generiert werden:



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

```

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

```

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



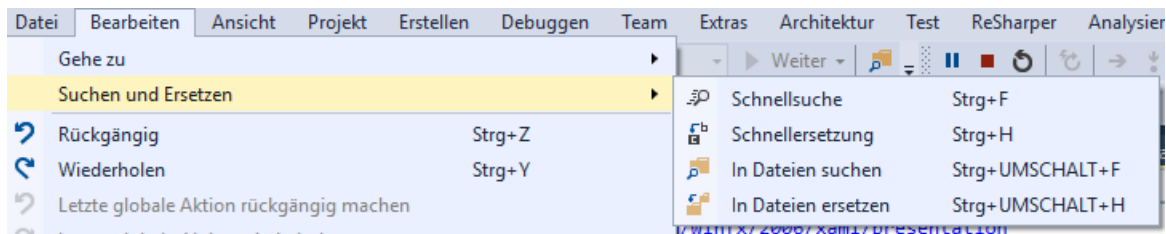
Wir sehen, alle Elemente werden so gerendert wie wir das haben wollten und in dem Schema wie wir dies vorgegeben haben. So eröffnen sich tolle Möglichkeiten für eine neue Anwendungsoptik und eine bessere und flexiblere Benutzerführung. Langsam dürfte wiederum ein kleiner WOW-Effekt einsetzen was nun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

Last but not least, **HierarchicalDataTemplates**

Mit **HierarchicalDataTemplate** 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 **TreeView** oder eines **Menus**.

Wir alle kennen ein Menu:



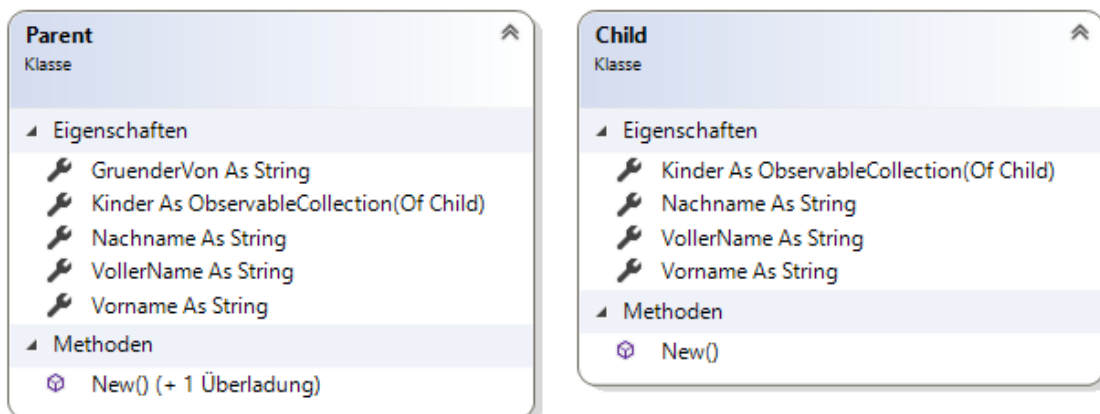
Erstmal haben wir die erste Ebene (Root) mit „Datei“, „Bearbeiten“, Ansicht“, usw. Danach haben wir eine Unbestimmte Anzahl an Ebenen welche wieder eine unbestimmte Anzahl an weiteren Kind-Ebenen haben kann.

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 **HierarchicalDataTemplates**. 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 `HierarchicalDataTemplate` an.

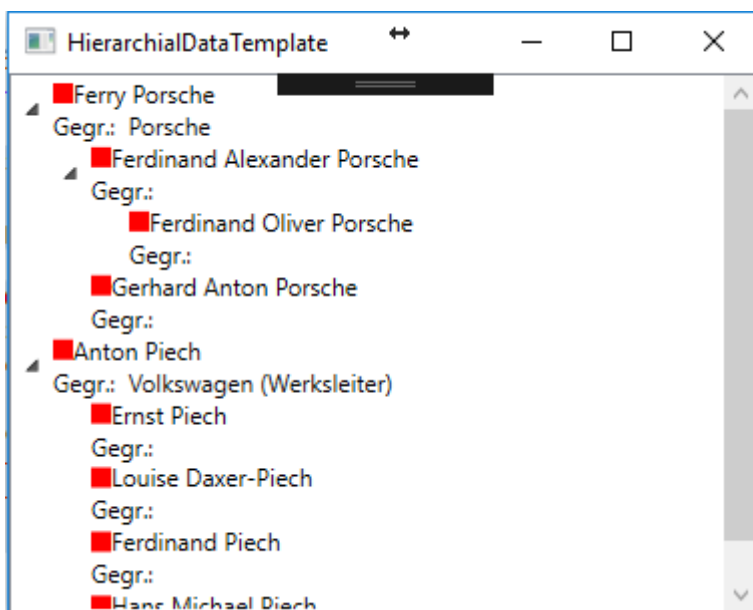
```
<TreeView DataContext="{Binding}" ItemsSource="{Binding HierarchieDaten}">
  <TreeView.ItemTemplate>
    <HierarchicalDataTemplate ItemsSource="{Binding Kinder}">
      <StackPanel>
        <StackPanel Orientation="Horizontal">
          <Rectangle Width="10" Height="10" Fill="Red"/>
          <TextBlock 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 „`HierarchieDaten`“ gebunden. Dieses befindet sich in der CodeBehind und ist vom Typ `Parent`.

Wie wir im Diagramm sehen konnten hat dieses eine Property `Kinder`. Dieses kann viele `Child` halten und `Child` wiederum auch viele `Childs`.

Beim Erstellen des `HierarchicalDataTemplate` geben wir 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.

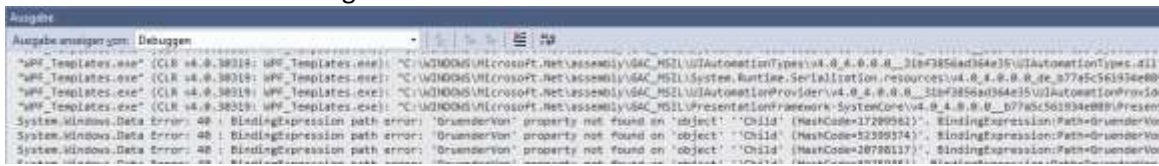
Folgende Ansicht wird in diesem Fall generiert:



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 `HierarchicalDataTemplates` 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>
```

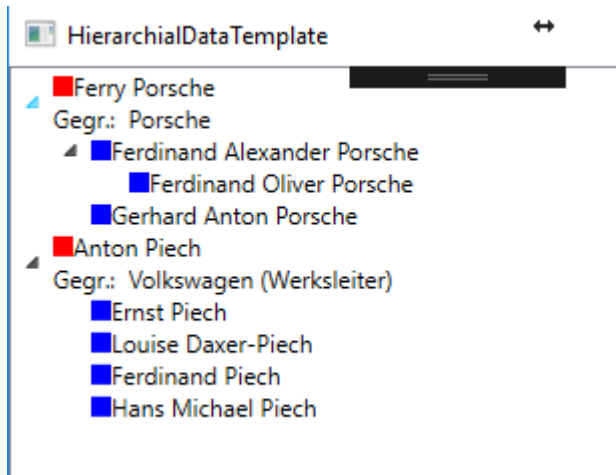
Wenn wir uns dieses TreeView ansehen werden wir die Stirn runzeln.

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 bin 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](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
```

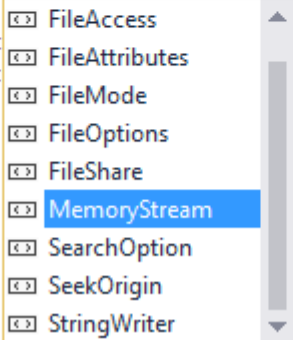
Solch ein Import im Code könnte wie folgt in XAML aussehen:

```
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:

```
<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:FileImport="clr-namespace:System.IO;assembly=mcorlib"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:WpfApplication1"
  mc:Ignorable="d"
  Title="MainWindow" Height="350" Width="525">
  <Window.Resources>
    <FileImport:MemoryStream x:Key="Test">
      </FileImport
    </Window.Resource
  <Grid>
    </Grid>
  </Grid>
</Window>
```



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 FahrzeugVm-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 `DataContext` Binden.

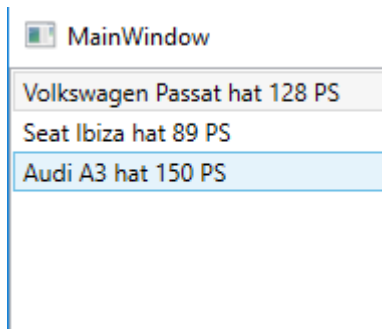
```
<Grid>
  <ListBox ItemsSource="{Binding Fahrzeuge}" SelectedIndex="0"/>
</Grid>
```

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 `DataContext` für die Laufzeit eingegeben haben aber keinen für die `DesignTime` (Entwicklungszeit). Dies kommt zwar noch genauer in einem späteren Kapitel aber ich zeige es hier schon mal schnell vor:

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

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

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



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

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

Beim Namespace <http://schemas.microsoft.com/winfx/2006/xaml/presentation> beispielsweise sind unter anderem z.b. die Namespaces [System.Windows](#) und [System.Windows.Controls](#) eingebunden.

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

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

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

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

2.1.3

Ressourcen

Was sind Ressourcen und was bringen sie mir

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

Die Anwendungsressourcen 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 Ressourcen 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 Ressourcen "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.

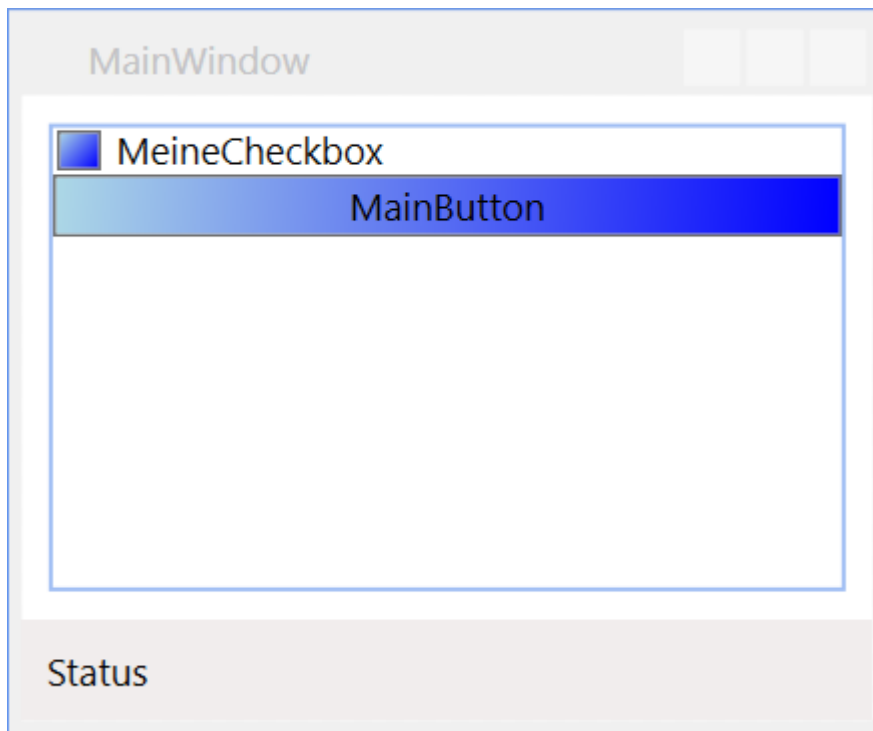
In diesem Window befindet sich ein DockPanel mit zwei Controls. Einer Statusleiste unten und einem StackPanel oben.

Definieren wir Ressourcen innerhalb des StackPanels kann das StackPanel sowie auch alle darunter liegenden Elemente auf diese Ressourcen zugreifen. Sowohl die Statusbar als auch das DockPanel kann nicht darauf zugreifen da sich die Ressourcen 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.

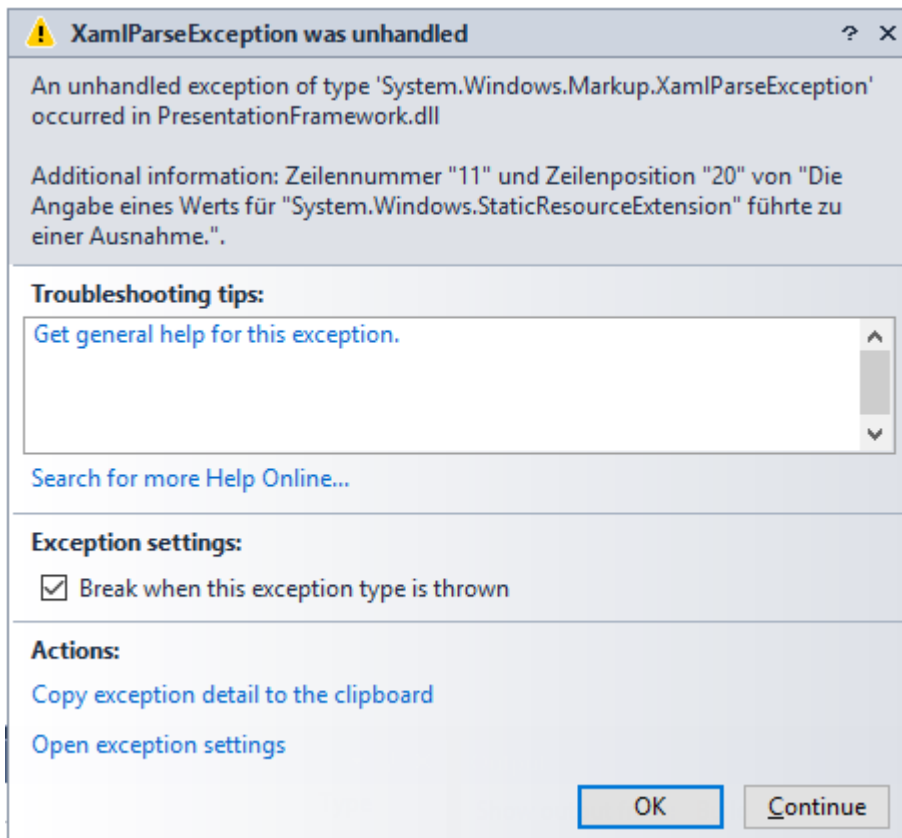
```
Panel LastChildFill="True">  
StatusBar DockPanel.Dock="Bottom">  
  <Label Content="Status" Background="{StaticResource MyTestBackgroundBrush}"/>  
/StatusBar>  
StackPanel Margin="10">  
  <StackPanel.Resources>
```

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 Ressourcen auch im Code erstellt werden zum anderen können XAML Files auch zur Laufzeit dynamisch nachgeladen werden oder es werden Assemblys geladen welche Ressourcen 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.

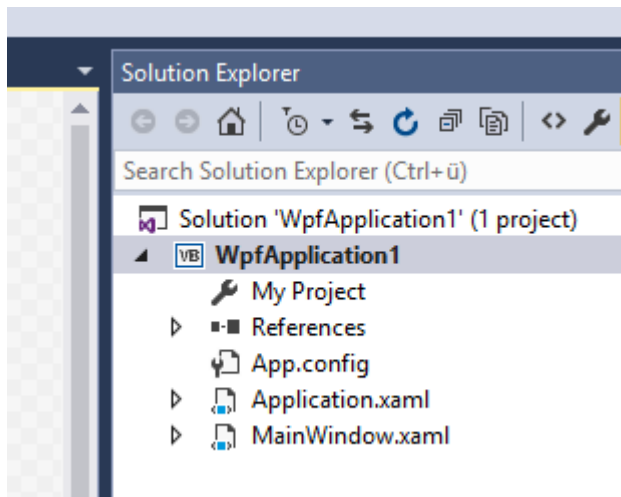
Also wo könnten wir nun die Resource definieren um diese sowohl in den Controls des StackPanels als auch in den Controls der Statusbar zur Verfügung zu haben?

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 Ressourcen 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 Ressourcen 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 Ressourcen 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?

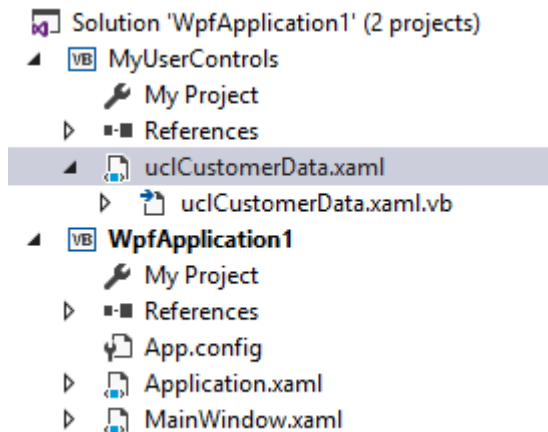
Ja, teilweise. Erstelle ich in einer Assembly ein Control mit einem Button bekommt dieser Button erstmal diesen Style nicht zugeordnet. Packe ich dieses UserControl allerdings dann innerhalb meiner App (in welcher das Margin definiert ist) in ein Window oder wiederum in ein UserControl bekommt der Button 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.

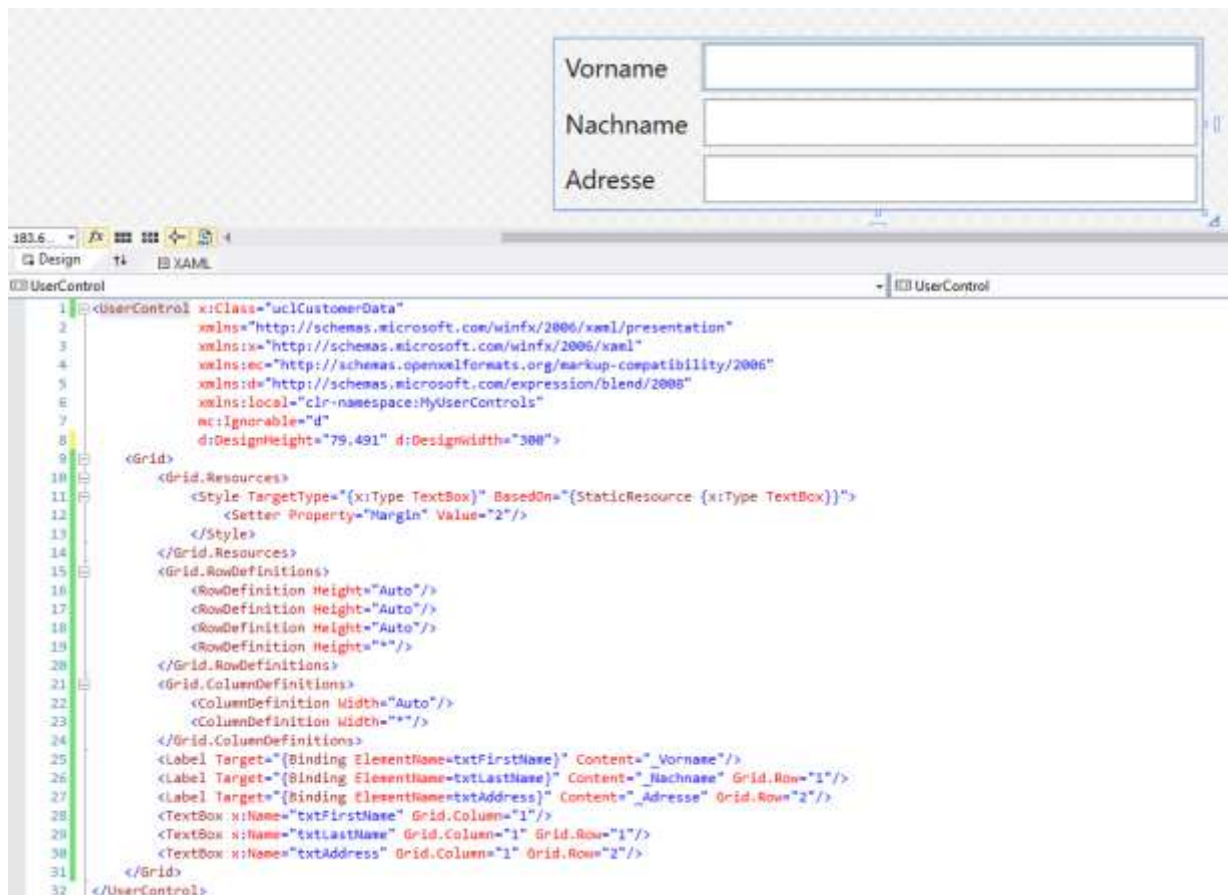
Wir haben ein UserControl erstellt welches wir gerne immer wieder verwenden möchten um Kundendaten anzuzeigen.

Da wir es in mehreren Projekten verwenden möchte erstellen wir 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:



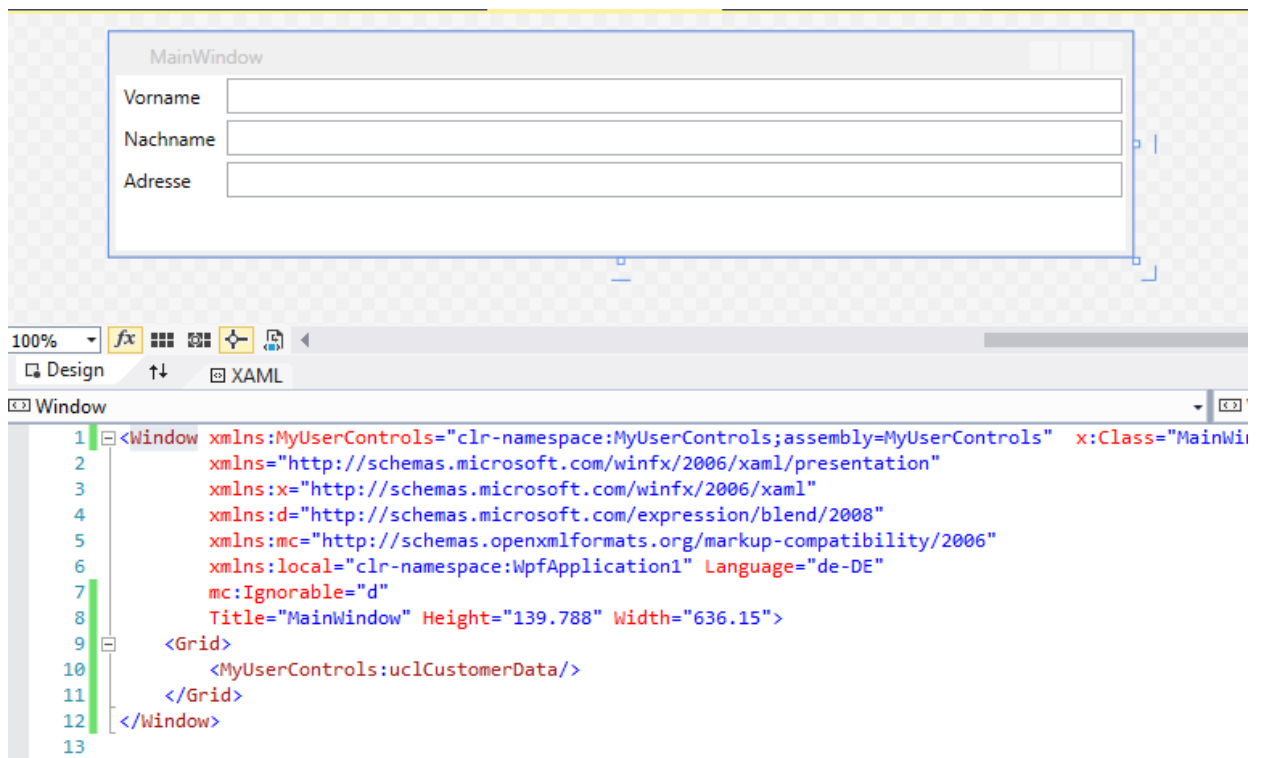
Einigen wird nun sicher auffallen das hier kein Binding auf das Text-Property der TextBoxen erfolgt ist. Die Eingeebenen Daten können also nie irgendwohin übertragen werden. Da es hier in diesem Kapitel allerdings rein um Ressourcen geht wollte ich darauf jetzt verzichten.

Wir sehen in den Grid-Ressourcen 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 Ressourcen 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:



Wir möchten allerdings, dass in unserer Anwendung alle TextBoxen mit einem grauem Hintergrund versehen werden.

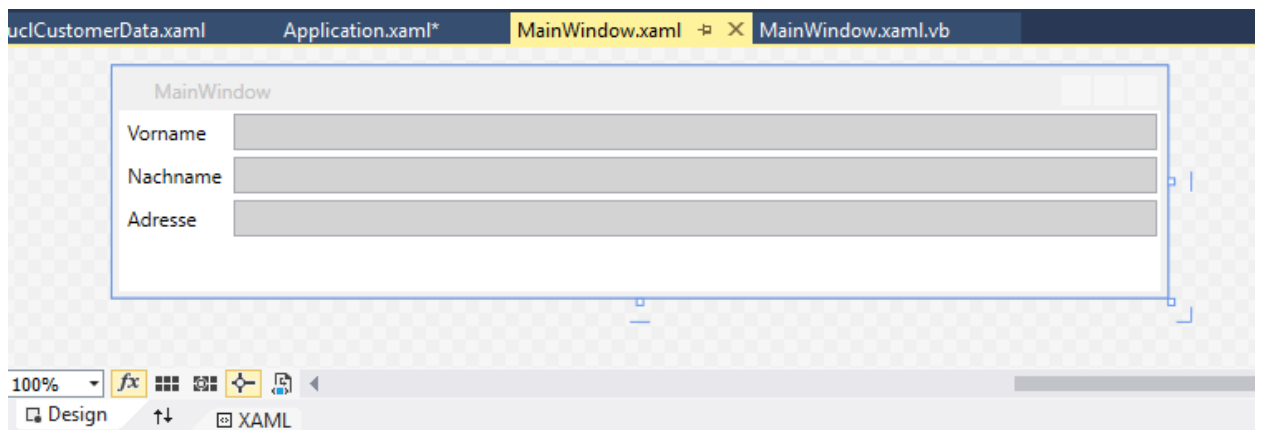
Also definieren wir in unseren [Application.xaml](#)-Ressourcen das dies der Fall sein soll.

```

CustomerData.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.



```

uclCustomerData.xaml  Application.xaml*  MainWindow.xaml  MainWindow.xaml.vb
MainWindow
Vorname
Nachname
Adresse
100%  fx  Design  XAML
Window
1  <Window xmlns:MyUserControls="clr-namespace:MyUserControls;assembly=MyUserControls" x:Class="Mair
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

```

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-Resources 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"/>
        </Grid.RowDefinitions>
    </Grid>
</UserControl>
```

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: <code><object property="{StaticResource key}" .../></code> <code><object></code> <code><object.property></code> <code><StaticResource ResourceKey="key" .../></code> <code></object.property></code> <code></object></code>	Syntax: <code><object property="{DynamicResource key}" .../></code> <code><object></code> <code><object.property></code> <code><DynamicResource ResourceKey="key" .../></code> <code></object.property></code> <code></object></code>
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>
```

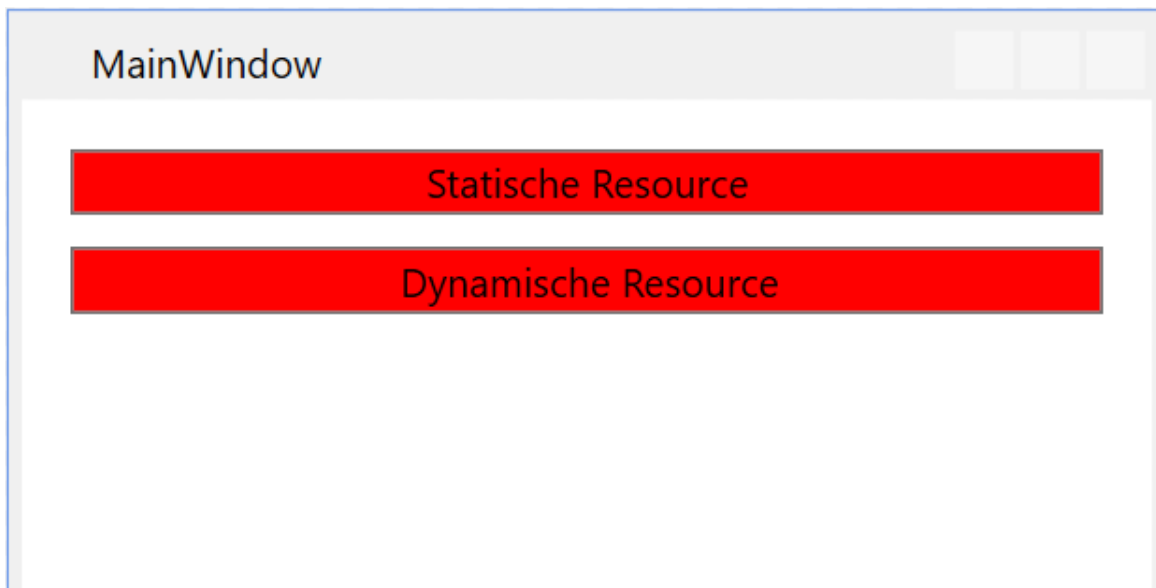
Damit man nun den großen Unterschied zwischen einer Statischen und einer Dynamischen Resource sieht erstelle ich zwei Buttons in einem StackPanel.

```

<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.



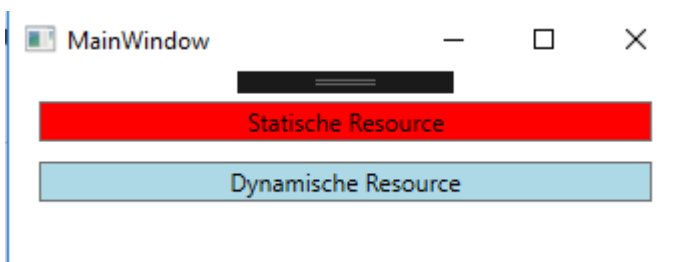
Wo der Unterschied liegt erkennen wir wenn wir nun mit folgendem CodeBehind die `Color` des `SolidColorBrush` mit dem Namen „`myBackgroundBrush`“ auf `LightBlue` ändern.

```

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 Ressourcen des [StackPanels](#) zu ändern und zu zeigen dass man auf die Ressourcen auch über ein untergeordnetes Objekt Zugriff bekommt. Die Resource „myBackgroudBrush“ wurde in den [Window](#)-Ressourcen 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 Ressourcen 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 Ressourcen einbindet.

2.1.4

Binding und das Bindingsystem

Ein wenig Theorie muss sein

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

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

2.1.4.1

Was ist DataBinding – Das Konzept dahinter

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

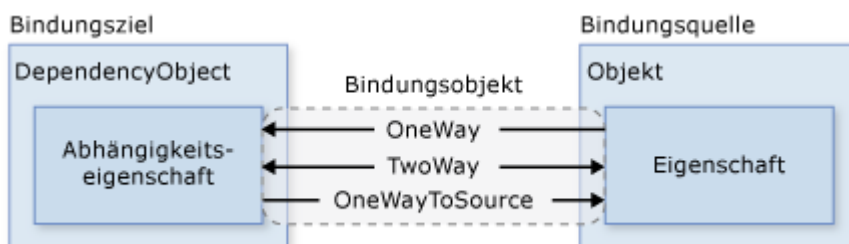
Hier ein Modell für ein Binding (Quelle: Microsoft Docs):



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

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

Die Richtung des Datenflusses:



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

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

- **OneWay**

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

- **TwoWay**

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

- **OneWayToSource**

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

- **OneTime**

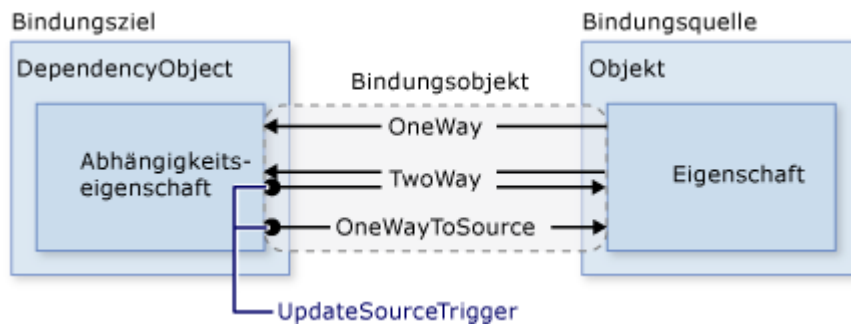
Hier wird ein OneWay Binding erstellt welches allerdings nur beim initialisieren des UI synchronisiert wird. Änderungen werden später nicht mehr aktualisiert. Im Grunde eine Momentaufnahme.

- **Default**

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

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

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



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

Es gibt 3 Werte für die Eigenschaft UpdateSourceTrigger.

LostFocus, *PropertyChanged* und *Explizit*.

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

In die andere Richtung, also von der Klasse zum UI wird dies im Code gesteuert, hierfür muss das Interface *INotifyPropertyChanged* implementiert werden. Im Setter des jeweiligen Properties muss das Event *PropertieChanged* geworfen werden wobei diesem *PropertyChangedEventArgs* übergeben werden - welche einen String mit dem Namen des Property 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 Standardkonvertierungen 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 überbringen 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](#) Beispielflasse aus dem letzten Kapitel her und erstellen einen View für diese wie wir ihn im letzten Kapitel auch bereits gesehen hatten.

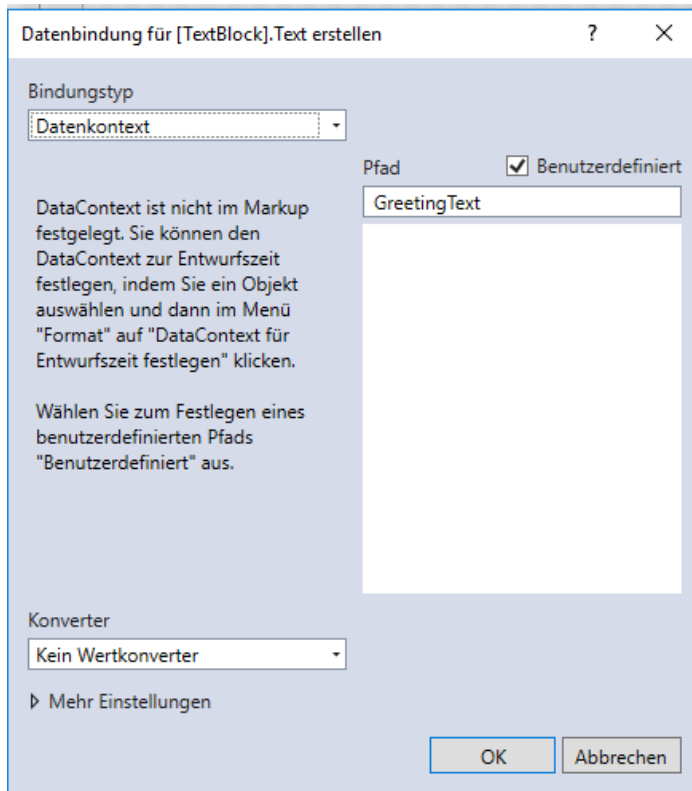
Nehmen wir nun Beispielsweise mal die TextBox welche wir auf den Begrüßungstext GreetingText binden möchten:

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

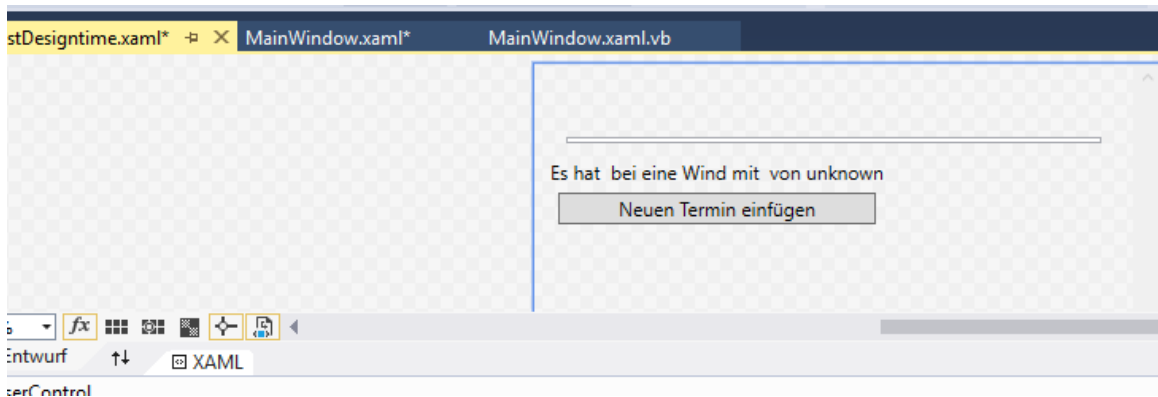
Auch über den sogenannten „Binding-Picker“ über das Eigenschaftenfenster des jeweiligen Controls haben wir nur die Möglichkeit manuell ein Binding zu setzen aber der Designer weiß nicht welche Properties die Klasse hat. Wie auch, er weiß nicht mal von welcher Klasse wir überhaupt reden.

```
StackPanel margin= 10 Language= de >
  <Label FontSize="20" Content="{Binding CurrentDate,Mode=OneTime}"/>
  <TextBlock Text="{Binding GreetingText}"/>
  <ListBox ItemsSource="{Binding ...}" MaxHeight="200">
    <ListBox.ItemTemplate>
      <DataTemplate>
        <StackPanel>
          <StackPanel>
            <ContentPresenter Content="{Binding IconAlarm}" />
            <TextBlock Text="{Binding ...}" StringFormat="{0:HH:mm:ss}"/>
            <TextBlock Text="{Binding ...}" Alignment="Center"/>
            <TextBlock Text="{Binding ...}" Text="Minutes, StringFormat=" />
          </StackPanel>
          <TextBlock Text="{Binding Title}"/>
          <Line Stroke="DarkBlue" StrokeThickness="2" X1="0" X2="200" Y1="0" Y2="0" />
        </StackPanel>
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
</StackPanel>
```

Wir haben keine Intellisense und zur Verfügung



Der Designer kann auch die Controls nicht mit Beispieldaten füllen:



Der Designer weiß nichts von unserer Klasse

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

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

```

<Window x:Class="MainWindow"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:local="clr-namespace:_2_1_4_3_DesignTimeSupport"
  mc:Ignorable="d"
  Title="MainWindow" Height="450" Width="800">
  <Grid>

  </Grid>
</Window>

```

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

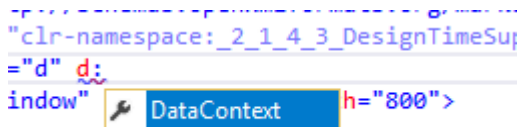
Wir werden nun gleich den mit einem „d“ benannten Namespace

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

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

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

```

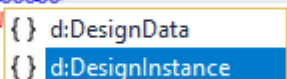
"clr-namespace:_2_1_4_3_DesignTimeSup
="d" d:
indow"  h="800">

```

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.

```

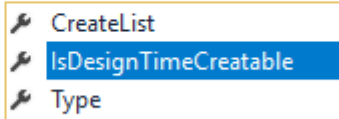
http://schemas.openxmlformats.org/markup-compatibility
l="clr-namespace:_2_1_4_3_DesignTimeSupport"
le="d" d:DataContext="{d:}"
nWindow" Height="450" W 

```

```

http://schemas.openxmlformats.org/markup-compatibility/2006"
="clr-namespace: 2_1_4_3_DesignTimeSupport"
e="d" d:DataContext="{d:DesignInstance
Window" Height="450" Width="800">

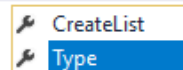
```



```

'http://schemas.openxmlformats.org/markup-compatibility/2006"
al="clr-namespace: 2_1_4_3_DesignTimeSupport"
ole="d" d:DataContext="{d:DesignInstance IsDesignTimeCreatable=True,}"
inWindow" Height="450" Width="800">

```



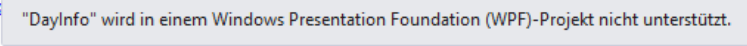
Sind wir fertig mit unserer Eingabe meckert die IDE allerdings das „DayInfo“ in einem WPF Projekt nicht unterstützt wird. Schade, WPF kann kein DayInfo. Wie jetzt?

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 Namespace in welchem die DayInfo Klasse liegt in den View holen. Dabei unterstützt uns die IDE und wir können dies mit **STRK + .** tun.

```

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" d:DataContext="{d:DesignInstance IsDesignTimeCreatable=True,Type={x:Type DayInfo}}"
Title="MainWindow" Height="450
<Grid>

```



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

```

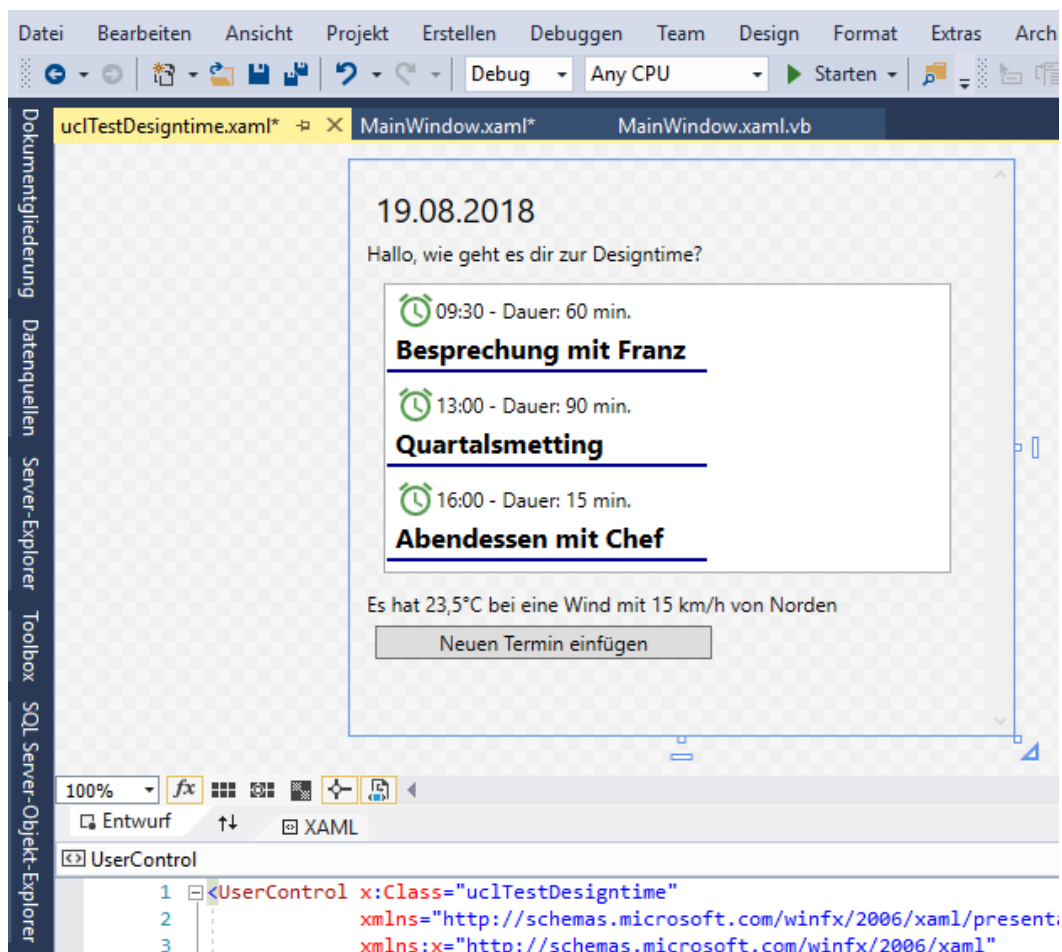
<Window x:Class="MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
xmlns:local="clr-namespace: 2_1_4_3_DesignTimeSupport"
xmlns:testClasses="clr-namespace: 2_1_4_3_DesignTimeSupport.TestClasses"
mc:Ignorable="d" d:DataContext="{d:DesignInstance IsDesignTimeCreatable=True,Type={x:Type testClasses:DayInfo}}"
Title="MainWindow" Height="450" Width="800">
<Grid>
</Grid>
</Window>

```

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

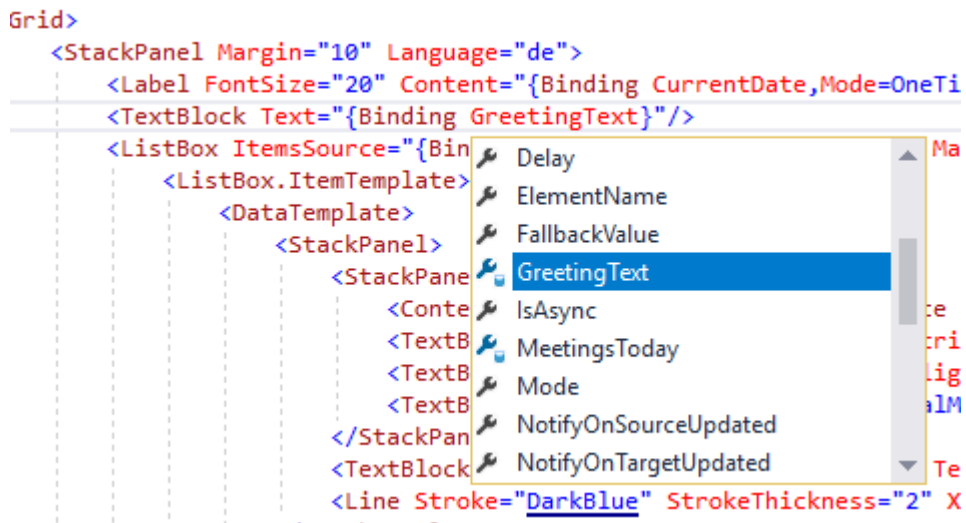
```
<UserControl x:Class="uclTestDesigntime"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"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

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



Auch die Intellisense ist nun vorhanden:

```
Grid>
  <StackPanel Margin="10" Language="de">
    <Label FontSize="20" Content="{Binding CurrentDate,Mode=OneTi
    <TextBlock Text="{Binding GreetingText}"/>
    <ListBox ItemsSource="{Bin
      <ListBox.ItemTemplate>
        <DataTemplate>
          <StackPanel>
            <StackPane
              <Conte
                <TextB
                <TextB
                <TextB
              </StackPan
            <TextBlock
          <Line Stroke="DarkBlue" StrokeThickness="2" X
```



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 DataContext 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 parameterlosen 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()

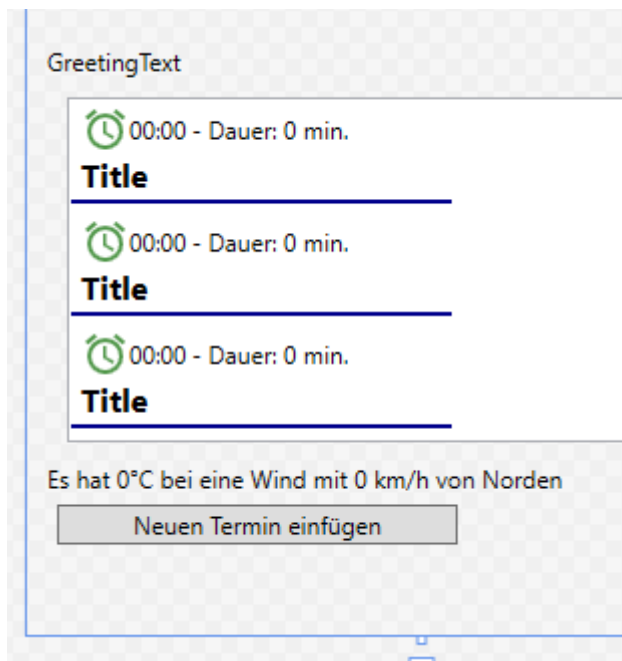
        MeetingsToday = New ObservableCollection(Of Meeting) From {
            New Meeting("Besprechung mit Franz", New Date(CurrentDate.Year, CurrentDate.Month, CurrentDate.Day, 9, 30, 0),
                TimeSpan.FromMinutes(60)),
            New Meeting("Quartalsmetting", New Date(CurrentDate.Year, CurrentDate.Month,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

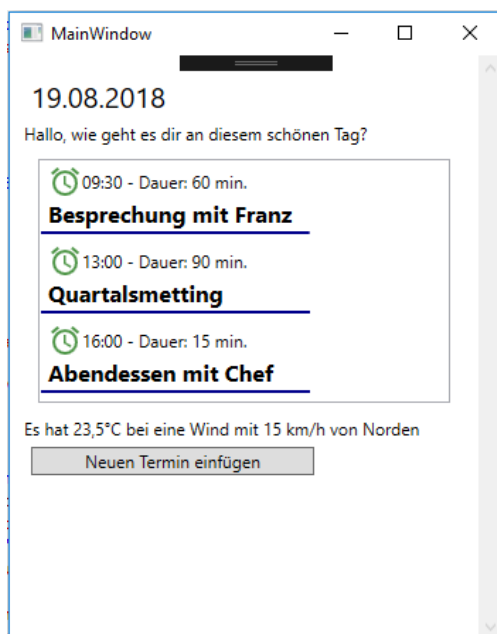
Doch was ist los, zur Designzeit haben wir die Daten drinnen, es sollte doch alles funktionieren. Warum sehen wir nun keine Daten? Das liegt daran das wir den DesignTime-Datenkontext angegeben haben und der Designer nun weiss was Sache ist, aber wir haben ja bislang kein Databinding zur Laufzeit angegeben 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 meiste Büchern, wie auch in dem Buch welches ich hier bei mir liegen habe wird auf über 1200 Seiten reiner WPF kein Wort von einem DesignTime-Datenkontext erwähnt. Ich finde das extrem schade, gerade mit dieser Option spare ich mir nicht nur extrem viel Zeit und die Zusammenarbeit mit z.B. einem Designer wird damit viel einfacher wenn dieser Beispieldaten serviert bekommt. Instellisense rundet das ganze nochmals ab.

Schade das hier von vielen Seiten nicht darauf eingegangen wird.

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

Das Visual Studio Projekt und den Thread findet Ihr wie immer im [Vb-Paradise Forum](#).

2.1.4.4

Binding über Converter

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

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

Collapsed zeigt das Element nicht an **und reserviert auch keinen Platz**.

Hidden zeigt das Element nicht an, **reserviert allerdings den Platz**.

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

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

Von beidem ist abzuraten. Warum?

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

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

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

Eine solche Konverterklasse muss das Interface `IValueConverter` implementieren!

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

```
Imports System.Globalization

Namespace Converter
    Public Class BooleanToVisibilityConverter
        Implements IValueConverter

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

            Throw New NotImplementedException()
        End Function

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

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

Die Funktion `Convert` wird von der WPF aufgerufen wenn von der Klasse in Richtung View der Wert konvertiert werden soll.

Wird der Konverter im XAML eingebunden ruft die WPF diese Funktion auf um zu versuchen den Wert zu Konvertieren. Im Parameter *value* ist also der Wert enthalten welcher von der Klasse kommt, also der Wert des Property's auf welches gebunden wurde. In unserem Fall ist dies ein Wert vom Typ `Boolean`. Wichtig ist hier immer ob der Wert welcher hier hereingereicht wird nicht vielleicht *Nothing* ist.

Anschließend müssen wir den Wert von einem `Boolean` in einen Wert vom Typ `Visibility` umwandeln.

```

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

```

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

Ist der Wert in *value* **True** geben wir den Wert *Visible* des Typs **Visibility** zurück.

Ist der Wert in *value* **False** geben wir den Wert *Hidden* des Typs **Visibility** zurück.

Soweit, war das gar nicht schwer. Was ist aber mit den anderen Parametern der Funktion?

TargetType gibt uns den Typ zurück welchen die WPF von uns erwartet. In unserem Fall wird dies ein **Visibility** sein.

Parameter kann in XAML definiert werden und wird uns hier hereingereicht. Parameter ist kein Dependency Property und kann somit nicht gebunden werden, dazu kommen wir aber noch.

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

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

Wird im View der Wert gesetzt. Beispielsweise über ein Dropdown welches an den Enumerator gebunden ist und wir hier umstellen, will die WPF (wenn in *TwoWay* gebunden wurde) den Wert wieder vom View zurück in die Klasse also in das Property der Klasse schreiben.

Die WPF bemüht also abermals den Converter weil Sie weiß – was in die eine Richtung erledigt werden muss gilt auch für die andere. Hier bekommen wir in *value* nun aber einen Wert vom Typ **Visibility** hereingereicht und müssen diesen nun zurück in einen Wert vom Typ **Boolean** konvertieren.

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

```

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

```

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

z.b.:

```

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

```

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

```

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

```

Oben hatte ich bereits die Converterparameter angesprochen, über setzen des Paramters können wir zusätzliche Informationen an den Converter übergeben. Welche Informationen bleibt im Grunde uns überlassen, nur ist zu beachten das hier kein Binding möglich ist da [Converterparameter](#) kein DependencyProperty ist.

```

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

```

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

```

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

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

```

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

Hier geht's zum Video: <https://youtu.be/4M44oazjdw>

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

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

2.1.4.5

Binding über DataTemplates

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

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

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

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

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

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

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

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

```
Namespace Classes
  Public Class Car
    Inherits TransportBase

  End Class
End Namespace
```

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

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

```
Namespace Classes
  Public Class Car
    Inherits TransportBase

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

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

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

Das selbe können wir nun auch mit anderen Fahrzeugtypen machen wie einem Flugzeug. Nur das bei einem Flugzeug andere Daten wichtig sind. Anders als bei einem Fahrzeug interessiert

uns vielleicht die Ausstattung nicht sondern wieviele Passagiere befördert werden können oder die maximale Reiseflughöhe. Und statt der Anzahl der Cylinder ist die Anzahl der Turbinen vielleicht interessant. Bei einem Schiff wäre es vermutlich der maximale Tiefgang der uns interessiert.

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

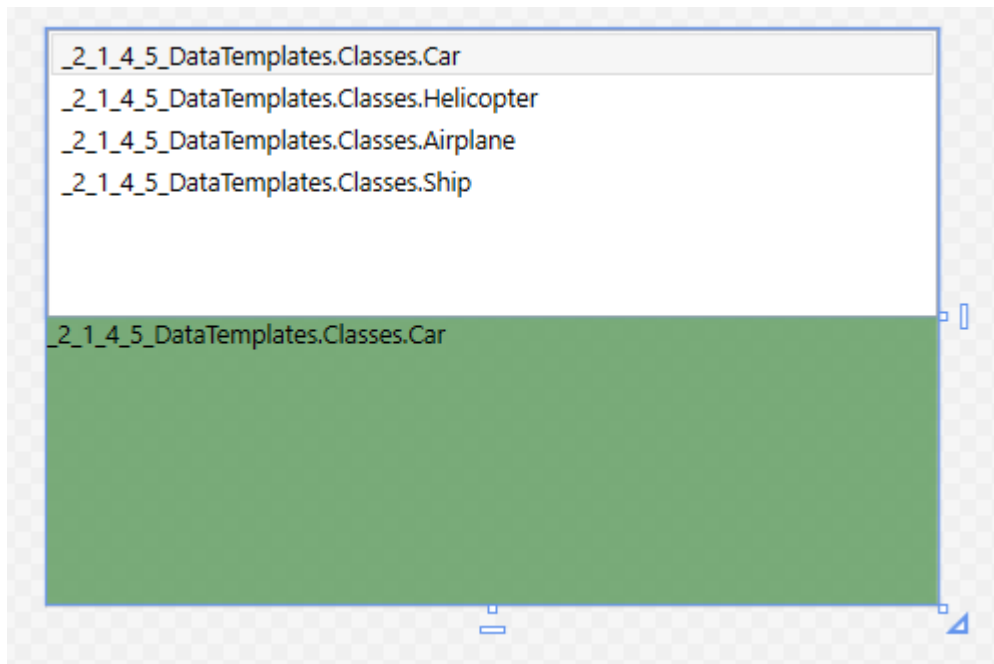
Erstmal erstelle ich ein UserControl mit einer ListBox und binde dieses auf eine Klasse in welcher ich eine reihe von Fahrzeugen lade. Die Klasse besitzt zwei Eigenschaften [AllTransport](#) von Typ [ObservableCollection\(Of TransportBase\)](#) und [SelectedTransport](#) vom Typ [TransportBase](#).

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

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

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

Das sieht im Moment noch eher unspektakulär aus:



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

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

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

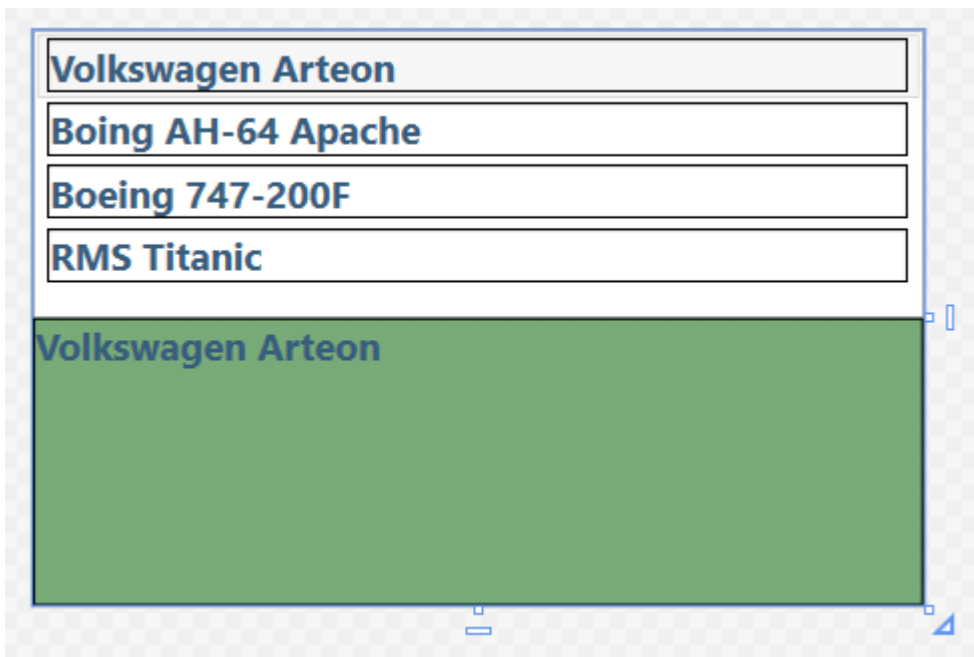
Möchten wir also nicht für jeden Fahrzeugtypen ein Template erstellen können wir auch ein Allgemeines erstellen. Damit wir dieses allerdings nicht nur innerhalb der ListBox zur verfügung haben machen wir das in den Ressourcen des UserControl, so profitiert auch das ContentControl davon:


```

<UserControl x:Class="uclShowTransports"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:local="clr-namespace:_2_1_4_5_DataTemplates"
  xmlns:classes="clr-namespace:_2_1_4_5_DataTemplates.Classes"
  mc:Ignorable="d d:DesignHeight="273.623" d:DesignWidth="424.364">
  <UserControl.Resources>
    <DataTemplate DataType="{x:Type classes:TransportBase}">
      <Border BorderBrush="Black" BorderThickness="1">
        <TextBlock FontWeight="Bold"
          FontSize="18"
          Foreground="#375D7E">
          <Run Text="{Binding Brand}"/>
          <Run Text="{Binding Model}"/>
        </TextBlock>
      </Border>
    </DataTemplate>
  </UserControl.Resources>
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="1*"/>
      <RowDefinition Height="1*"/>
    </Grid.RowDefinitions>
    <ListBox ItemsSource="{Binding AllTransports}"
      HorizontalContentAlignment="Stretch"
      SelectedItem="{Binding SelectedTransport}"/>
    <Border Grid.Row="1" Background="#C6559655">
      <ContentControl Content="{Binding SelectedTransport}">

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

```



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

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

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

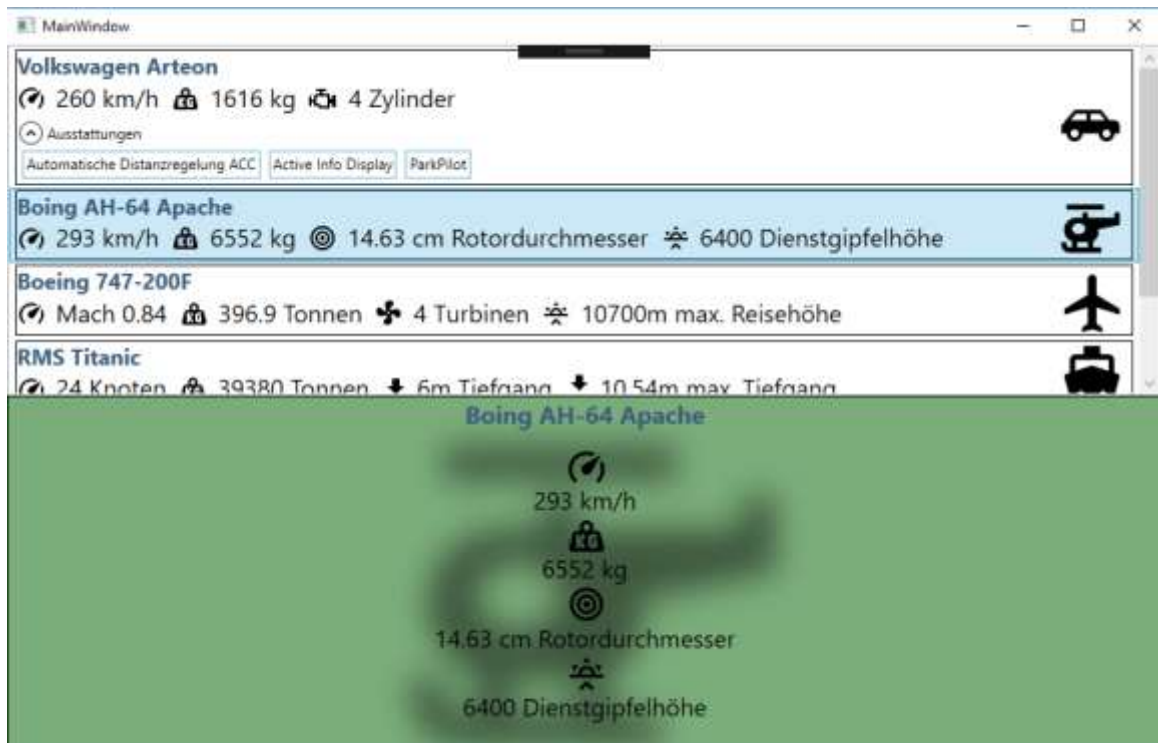
```
<ContentControl Content="{Binding SelectedTransport}">
  <ContentControl.Resources>
    <DataTemplate DataType="{x:Type classes:Car}">
      <local:uclCar/>
    </DataTemplate>
    <DataTemplate DataType="{x:Type classes:Helicopter}">
      <local:uclHeli/>
    </DataTemplate>
    <DataTemplate DataType="{x:Type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

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



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

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



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

```

<UserControl x:Class="uclShowTransports"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:local="clr-namespace:_2_1_4_5_DataTemplates"
    xmlns:classes="clr-namespace:_2_1_4_5_DataTemplates.Classes"
    mc:Ignorable="d" d:DataContext="{d:DesignInstance
IsDesignTimeCreatable=True,Type=local:AllTransports}"
    d:DesignHeight="273.623" d:DesignWidth="424.364">
    <UserControl.Resources>
        <DataTemplate DataType="{x:Type classes:TransportBase}">
            <Border BorderBrush="Black" BorderThickness="1">
                <TextBlock FontWeight="Bold" FontSize="18" Foreground="#375D7E">
                    <Run Text="{Binding Brand}"/>
                    <Run Text="{Binding Model}"/>
                </TextBlock>
            </Border>
        </DataTemplate>
        <DataTemplate DataType="{x:Type classes:Car}">
            <local:uclCarListItem/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type classes:Helicopter}">
            <local:uclHeliListItem/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type classes:Airplane}">
            <local:uclAirplaneListItem/>
        </DataTemplate>
        <DataTemplate DataType="{x:Type classes:Ship}">
            <local:uclShipListItem/>
        </DataTemplate>
    </UserControl.Resources>
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="1*"/>
            <RowDefinition Height="1*"/>
        </Grid.RowDefinitions>
        <ListBox ItemsSource="{Binding AllTransports}"
            HorizontalContentAlignment="Stretch"
            SelectedItem="{Binding SelectedTransport}"/>
        <Border Grid.Row="1" Background="#C6559655">
            <ContentControl Content="{Binding SelectedTransport}">
                <ContentControl.Resources>
                    <DataTemplate DataType="{x:Type classes:Car}">
                        <local:uclCar/>
                    </DataTemplate>
                    <DataTemplate DataType="{x:Type classes:Helicopter}">
                        <local:uclHeli/>
                    </DataTemplate>
                    <DataTemplate DataType="{x:Type classes:Airplane}">
                        <local:uclAirplane/>
                    </DataTemplate>
                    <DataTemplate DataType="{x:Type classes:Ship}">
                        <local:uclShip/>
                    </DataTemplate>
                </ContentControl.Resources>
            </ContentControl>
        </Border>
    </Grid>
</UserControl>

```

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

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

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

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

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