

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 (ausstehend)

2.1.2. XAML Namespaces

2.1.2.1. Kurze Theorie

2.1.2.2. Anwendung interessanter Namespaces

Eigene Namespaces erstellen und integrieren

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

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 das ich mal eine Woche auslasse entschuldige ich mich bereits im Voraus dafür, bitte habt Verständnis das ich mal in Urlaub fahre oder beruflich etwas mehr Stress habe.

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

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

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

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

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

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

2.0

GRUNDLAGEN DER WPF

Wir kommen zu den Grundlagen. Die WPF bietet eine eigene "Designersprache". XAML (Extensible Application Markup Language); ist eine Markupssprache 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.

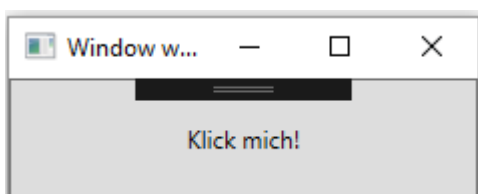
Die Darstellung passiert als Text über XML Dateien welche die Erweiterung `.xaml` aufweisen. Es kann mit jeder Codierung gearbeitet werden, typisch ist jedoch die Codierung als UTF-8.

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

```
<Window xmlns: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>
```

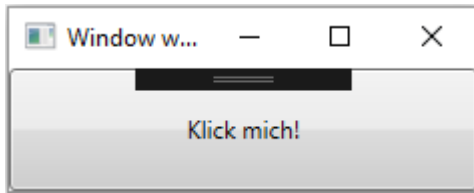
Hier werden ein Fenster und eine Schaltfläche mithilfe der Elemente `Window` und `Button` definiert. Jedes Element wird mit Attributen konfiguriert und korrespondiert mit den jeweiligen Properties des Elements. Zeile zwei und drei beinhalten die per Default eingetragenen XAML Namespaces, dazu kommen wir auch später mal, da möchte ich im Moment nicht so weit vorgehen. 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:



3

Unter Windows 7 so:



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

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

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

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

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

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

CODE BEHIND

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

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

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

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

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

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

Hier steht nun folgendes:

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

Und dies kennen wir nun ja wieder aus WinForms.

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

Ein weiteres Beispiel mit Attributen:

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

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

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

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

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

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

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

Hierfür gibt es mehrere Gründe. Einer wäre z.B. dass die Background Eigenschaft eines Buttons ja eigentlich einen [Brush](#) erwartet. Warum kann man dann `Background="Blue"` anwenden? Die WPF stellt intern diverse [TypeConverter](#) bereit, so kann sie den String "Red" in einen SolidColorBrush mit der Color "Red" umwandeln. Das gleiche gilt für die Eigenschaft Content des Buttons welche von Typ „Object“ ist. Hier macht die WPF automatisch einen String daraus.

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

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

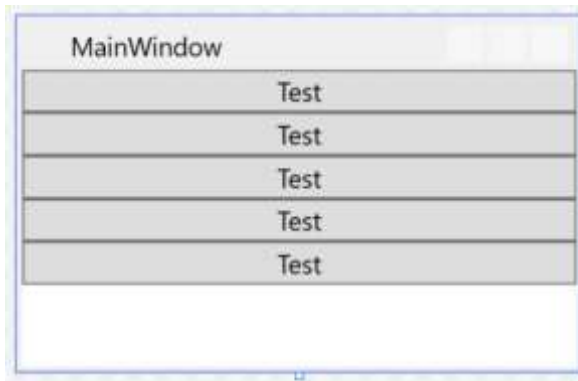
```
<StackPanel>
  <Button>Test</Button>
  <Button>
    <Button.Content>
      Test
    </Button.Content>
  </Button>
  <Button Content="Test"/>
  <Button>
```

```

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

```

Hier ein Screenshot:



Und da kommen wir gleich zum nächsten, der Auflistungssyntax:

Es gibt Container welche mehr als ein Control als Inhalt bekommen können. z.B., Grid, UniformGrid, alle Arten von Panels usw.

Folgendes Beispiel fügt einem Border ein StackPanel hinzu welches ein UniformGrid und einen Button enthält.

Das UniformGrid bekommt zwei Spalten welchen jeweils ein TextBlock hinzugefügt wird.

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

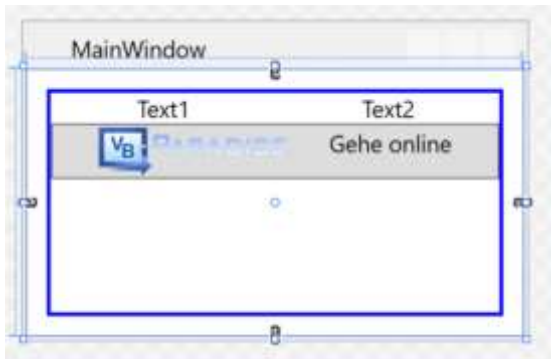
```

<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>

```

```
</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 herantasten. So denke ich, kann man besser umdenken und verstehen wie das Binding funktioniert wenn man beide Wege kennt und parallelen ziehen kann.

2.1

EINFÜHRUNG IN DIE WPF

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

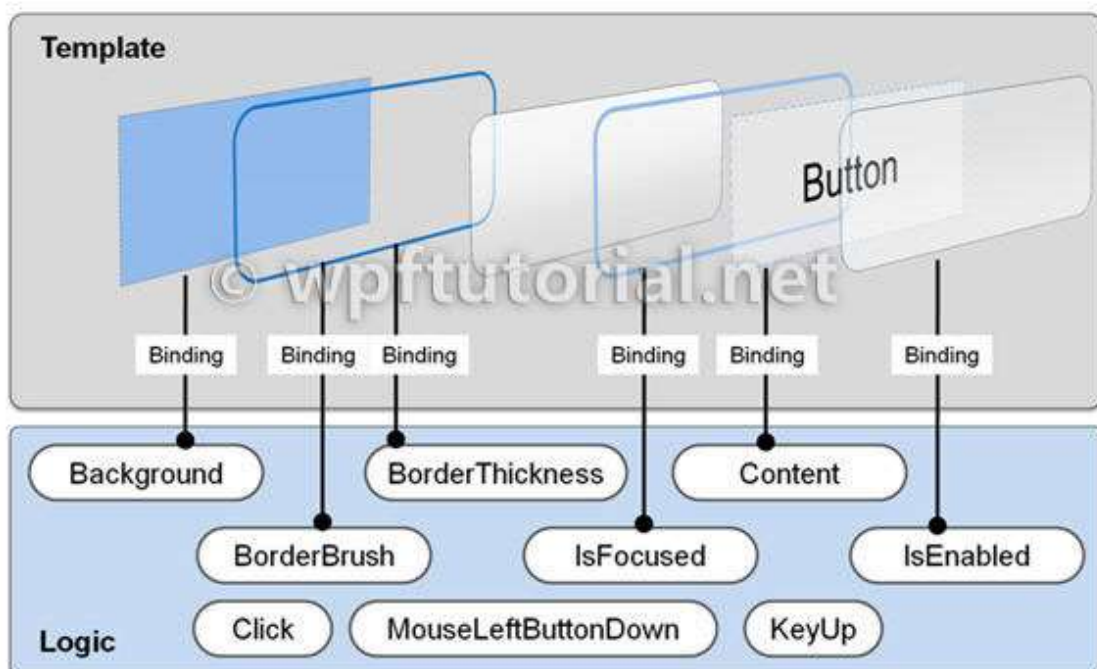
Die Verbindung zwischen dieser Logik und dem Template passiert über Binding.

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 Uri)
        Me.Marke = marke : Me.Modell = modell : Me.PS = ps : Me.Logo = logo
    End Sub

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

End Class
```

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

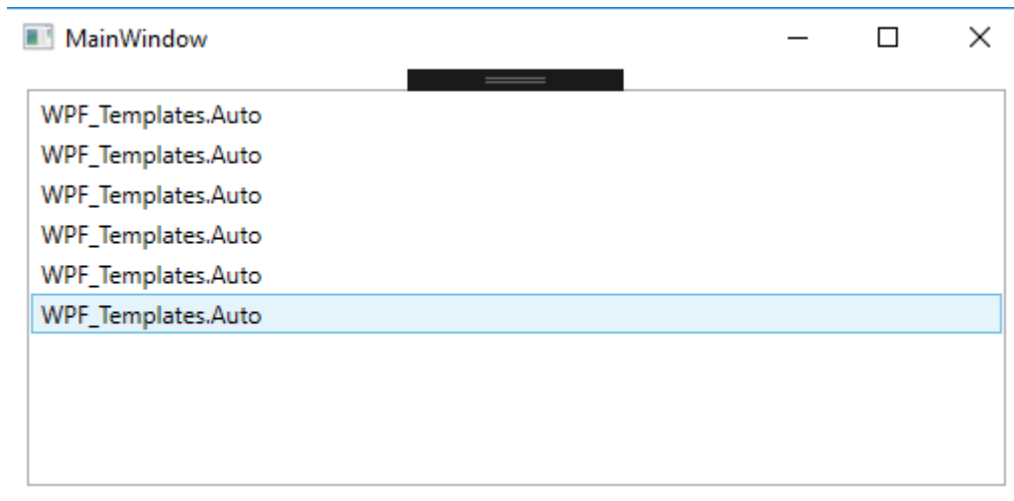
```
Private Sub MainWindow_Loaded(sender As Object, e As RoutedEventArgs) Handles Me.Loaded
    AutoListe = New ObservableCollection(Of Auto)
    With AutoListe
        .Add(New Auto("Audi", "R8 V10+", 610, "BrandImages/Audi.png"))
        .Add(New Auto("VW", "Arteon TDI 4 Motion", 240, "BrandImages/Vw.png"))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

    Me.DataContext = Me
End Sub

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

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

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

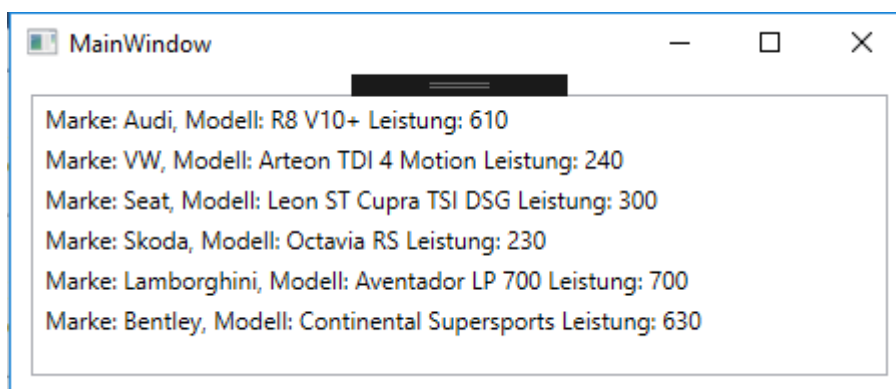


Nicht so schön. Wieder schlägt der [TypeConverter](#) der WPF zu. Die WPF wirft keinen Fehler sondern versucht die Daten welche ihr (ist die WPF ne Frau?) übergeben werden einfach irgendwie zu Rendern. Was liegt näher als `.ToString` aufzurufen. Und genau das macht sie.

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

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

Was uns folgendes Ergebnis Liefert:



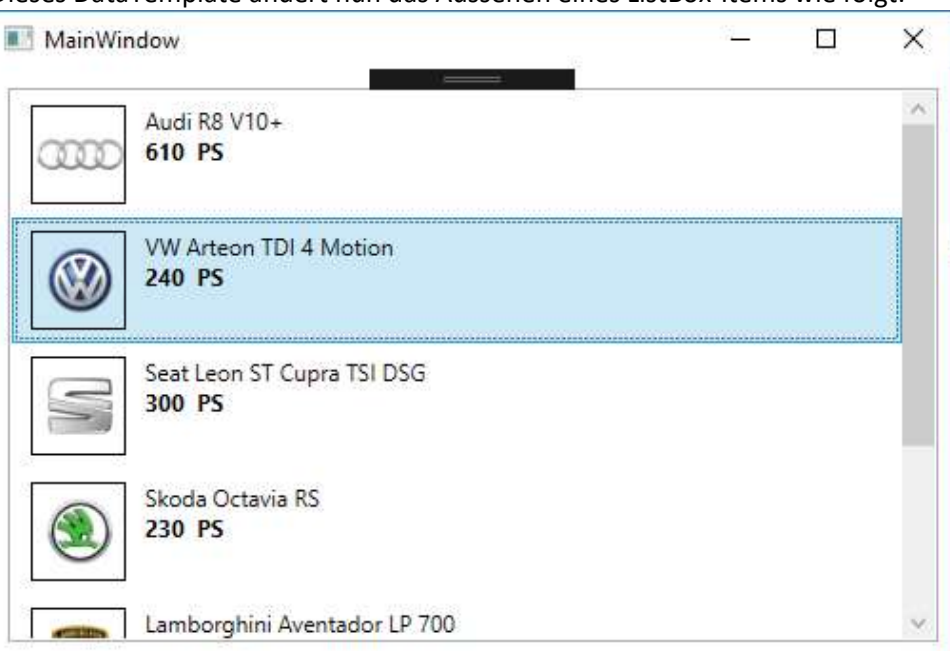
OK, schon um einiges besser, aber lange noch nicht zufriedenstellend. Wir befinden uns in der WPF, das geht ja mal sicher viel besser und schöner oder?

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

Hier ein DataTemplate direkt in das ItemTemplate der ListBox:

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

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



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

ItemsPanelTemplates

Diese Art von Templates erlaubt es uns das Layout, wie Items eines ItemControls wie z.B. einer ListBox zu bestimmen. Jeder 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:



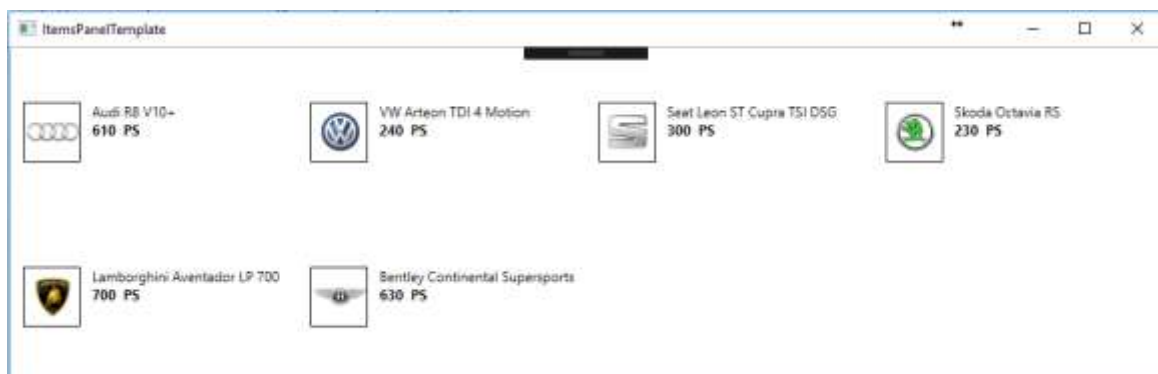
Wir können sehen das wir vier Spalten und zwei Zeilen haben. Wieder hat die WPF unsere ToString Methode in der Klasse Auto aufgerufen. Aber das wir DataTemplates oben bereits durchgenommen haben ist es für uns ja jetzt ein leichtes ein solches DataTemplate abermals zu implementieren. Wir können gleich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

```

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

```

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



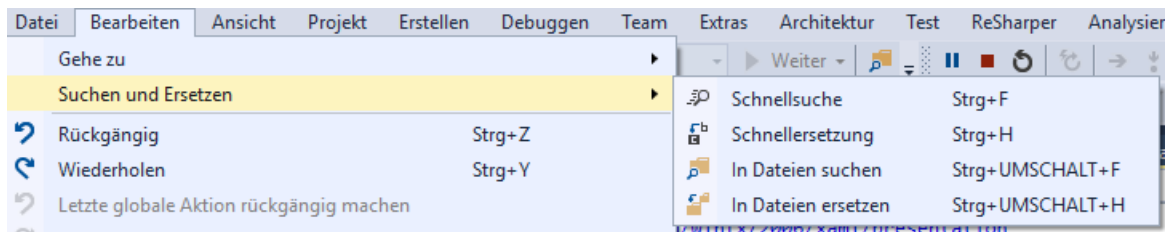
Wir sehen, alle Elemente werden so gerendert wie wir das haben wollten und in dem Schema wie wir dies vorgegeben haben. So eröffnen sich tolle Möglichkeiten für eine neue Anwendungsoptik und eine bessere und flexiblere Benutzerführung. Langsam dürfte wiederum ein kleiner WOW-Effekt einsetzen was nun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



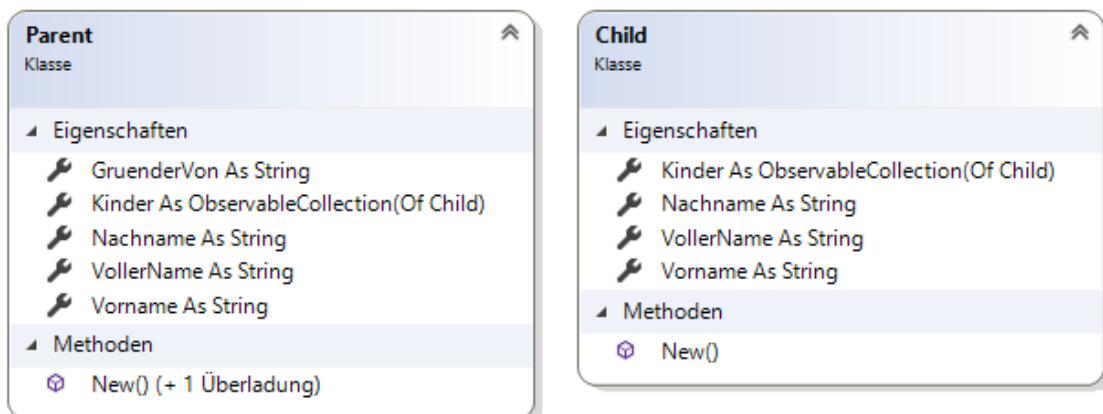
Erstmal haben wir die erste Ebene (Root) mit „Datei“, „Bearbeiten“, Ansicht“, usw. Danach haben wir eine Unbestimmte Anzahl an Ebenen welche wieder eine unbestimmte Anzahl an weiteren Kind-Ebenen haben kann.

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.

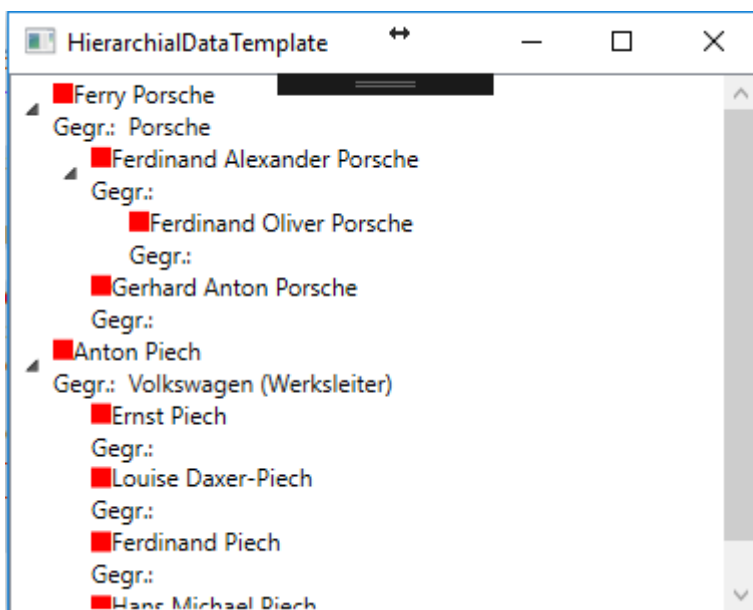
```
<TreeView DataContext="{Binding}" ItemsSource="{Binding HierarchieDaten}">
  <TreeView.ItemTemplate>
    <HierarchicalDataTemplate ItemsSource="{Binding Kinder}">
      <StackPanel>
        <StackPanel Orientation="Horizontal">
          <Rectangle Width="10" Height="10" Fill="Red"/>
          <TextBlock 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 ein 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.

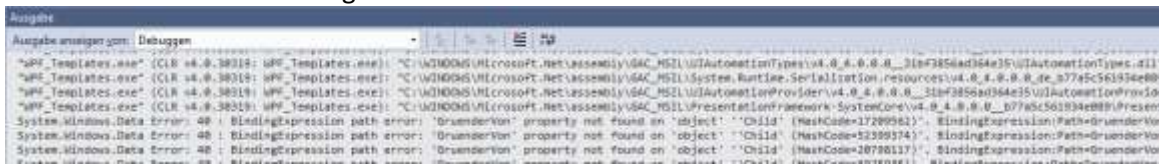
Folgende Ansicht wird in diesem Fall generiert:



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>
```

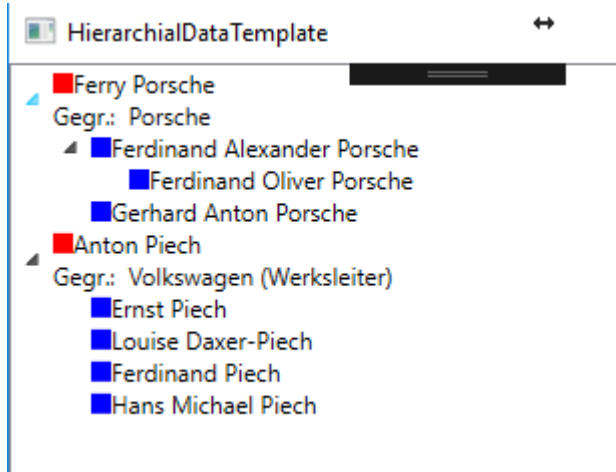
Wenn wir uns dieses TreeView ansehen werden wir die Stirn runzeln.

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>