

WPF Controls

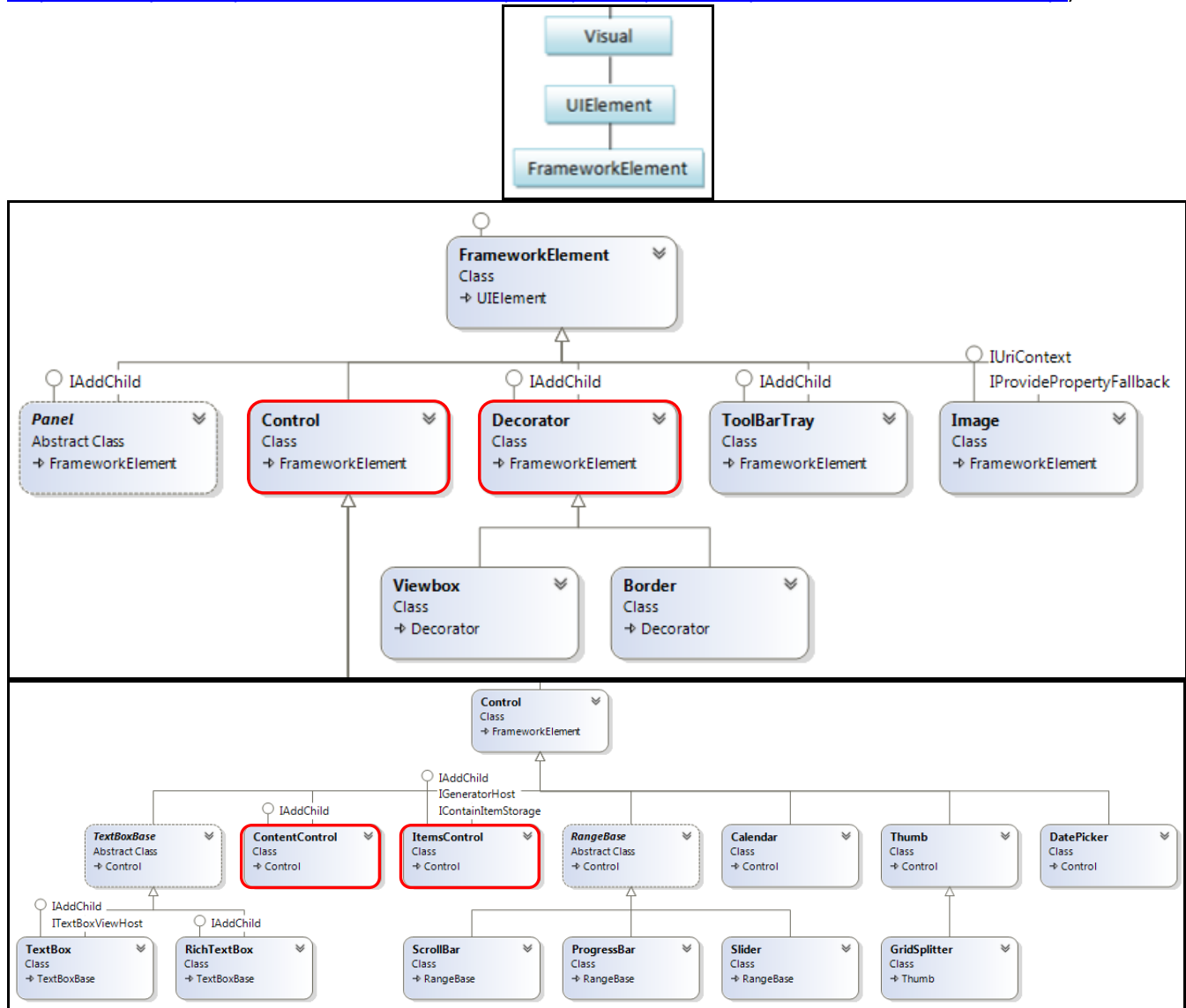
1 ÜBERBLICK	2
2 CODE PER CODE EINFÜGEN	3
3 CONTENTCONTROLS	4
3.1 CheckBox/RadioButton	5
3.2 HeaderedContentControl	5
3.2.1 GroupBox (RadioBox)	5
3.2.2 Expander	5
4 ITEMSCONTROLS	7
4.1 ListBox/ComboBox	7
4.1.1 DisplayMemberPath	8
4.2 Menü	8
4.3 ContextMenu	9
4.4 ToolBar	9
4.5 TabControl	10
5 DECORATOR	11
5.1 Border	11
5.2 ViewBox	11

1 Überblick

Hier sollen die wichtigsten Controls nur kurz besprochen werden. Die vielen Properties muss man einfach ausprobieren, die Namen sind aber meist selbsterklärend.

Die wichtigsten Controls sieht man in folgendem Klassen-Diagramm (siehe

<https://soumya.wordpress.com/2010/01/10/wpfsimplified-part-10-wpf-framework-class-hierarchy/>):



Viele Properties wie **Background**, **FontSize**, **BorderBrush**,... sind in der Klasse **Control** definiert und somit für alle Controls verfügbar.

Bei den meisten Controls erhält man bei **Doppelklick** einen Eventhandler für das **Default-Event**:

- Click für Button
- Checked für CheckBox
- ...

2 Code per Code einfügen

Zu Beginn soll gezeigt werden, wie man Controls mit C# in das Window bzw. ein Panel einfügen kann. Bekanntlich gilt ja:

- ein Tag im XAML entspricht einer Klasse in C#
- ein Attribut in XAML entspricht einer Property in C#

Als Beispiel soll ein Button in ein Stackpanel eingefügt werden.

```
<StackPanel x:Name="panButtons">
  <Button Content="First Button"
    Background="Blue" Width="200" HorizontalAlignment="Left"
    FontFamily="Arial" FontSize="14" Padding="1,3,2,4"
    Click="Button_Clicked" />
</StackPanel>
```

Diesen Button würde man per Code folgendermaßen in das Panel einfügen:

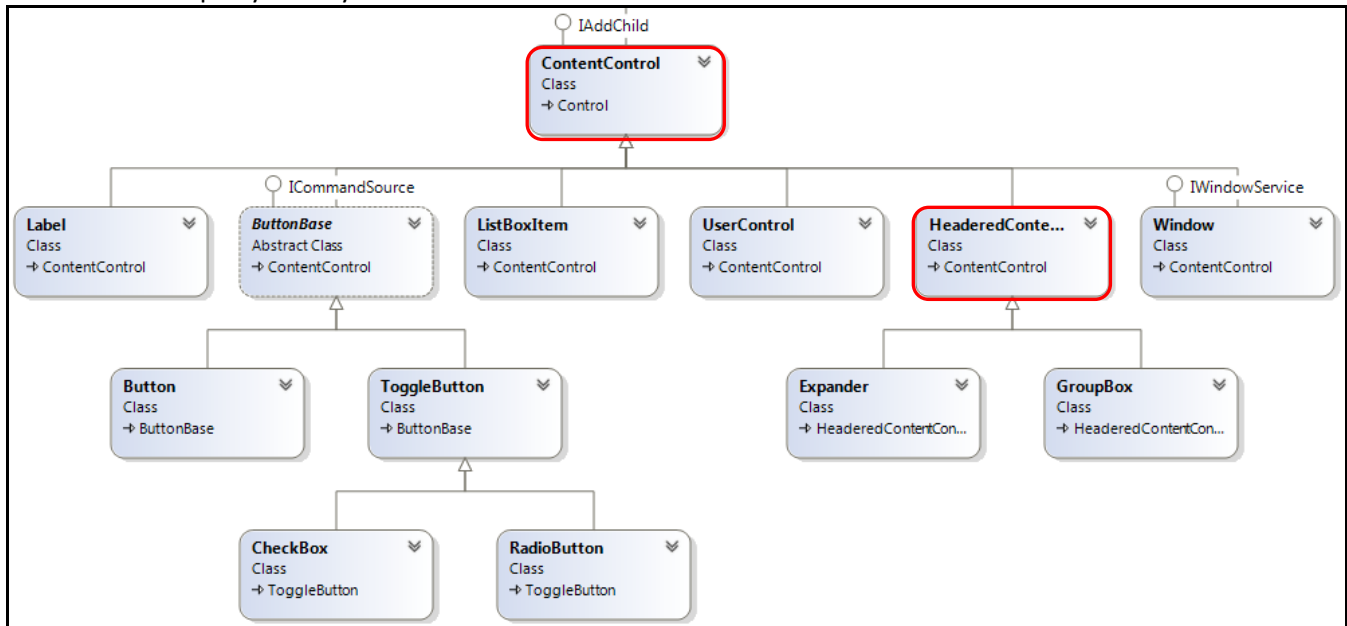
```
var button = new Button
{
    Content = "First Button",
    Background = new SolidColorBrush(Colors.Blue), //or: Brushes.Blue
    Width = 200,
    HorizontalAlignment = HorizontalAlignment.Left,
    FontFamily = new FontFamily("Arial"),
    FontSize = 14,
    Padding = new Thickness(1, 3, 2, 4)
};
button.Click += Button_Clicked;
panButtons.Children.Add(button);
```

Es fällt auf:

- die Properties sind natürlich nicht alle vom Typ String, sondern variieren je nach Property
- Events (hier konkret das Click-Event) müssen mit „+=“ zugewiesen werden

3 ContentControls

Wie der Name vermuten lässt, kann man **ContentControls** über die Property **Content** seinen Inhalt zuweisen. Dieser ist vom Typ **Object**. Will man einem Control als Content mehrere Objekte zuweisen, muss man der Content-Property ein Layout zuweisen.

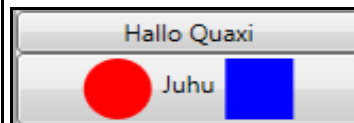


Aus der Klassenhierarchie geht hervor, dass man daher auch z.B. einem Button ein komplexes Layout zuweisen kann. Da der Content vom Typ **Object** ist, gibt es prinzipiell zwei Möglichkeiten:

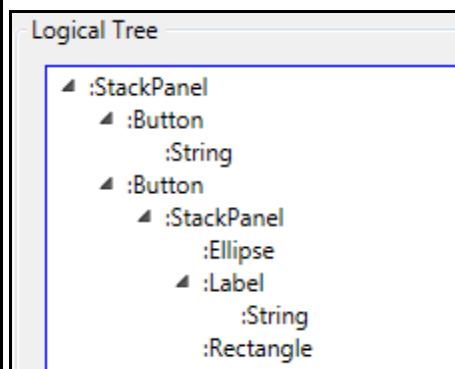
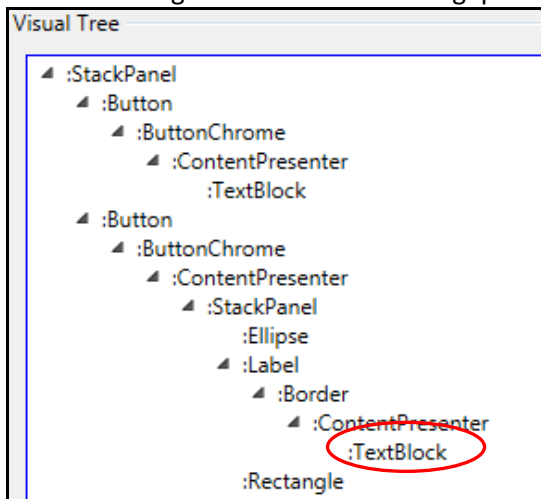
- Content ist ein **UIElement** → Element wird gezeichnet (genauer: es wird die **OnRender()**-Methode aufgerufen)
- Content ist **anderes Objekt** → Objekt wird in einen String umgewandelt, d.h. die **ToString()**-Methode aufgerufen, und in ein **<TextBlock>**-Element platziert

So kann man etwa Buttons auf folgende Weise definieren:

```
<Button>
  Hallo Quaxi
</Button>
<Button>
  <StackPanel Orientation="Horizontal">
    <Ellipse Fill="Red" Width="32" Height="32" />
    <Label>Juhu</Label>
    <Rectangle Fill="Blue" Width="32" Height="32" />
  </StackPanel>
</Button>
```



Dass der String in einen TextBlock eingepackt wird, sieht man dann im Visual Tree:



3.1 CheckBox/RadioButton

In WPF verhalten sich CheckBoxes und RadioButtons so wie gewohnt/erwartet:

- Property **IsChecked**
- Event **Checked**
- ...

Neu ist vielleicht, dass man über die Property **IsThreeState** die Werte auf **true**, **false**, **null** erweitern kann. Daher ist der Typ von **IsChecked** nicht **bool**, sondern **bool?**.

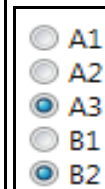
```
<CheckBox Name="chkThreeState" Content="ThreeState"
          IsThreeState="True"
          Click="chkThreeState_Clicked"/>
```

```
private void chkThreeState_Clicked(object sender, RoutedEventArgs e)
{
    var sChecked = chkThreeState.IsChecked?.ToString() ?? "null";
    Console.WriteLine(sChecked);
}
```



RadioButtons werden automatisch als zusammengehörig erkannt, wenn sie dasselbe direkte Elternelement im XAML haben. Haben sie das nicht, kann man sie über die **GroupName**-Property gruppieren

```
<StackPanel Grid.Column="1" Margin="10">
  <RadioButton Name="rdoA1" Content="A1"/>
  <RadioButton Name="rdoA2" Content="A2"/>
  <RadioButton Name="rdoA3" Content="A3" />
  <RadioButton Name="rdoB1" Content="B1" GroupName="GroupB"/>
  <RadioButton Name="rdoB2" Content="B2" GroupName="GroupB"/>
</StackPanel>
```

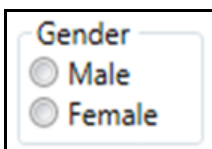


3.2 HeaderedContentControl

HeaderedContentControls sind - wie der Name sagt - ContentControls, die einen Header (also eine Überschrift) haben. Die beiden Vertreter sind **Expander** und **GroupBox**. Die einzelnen Elemente müssen dabei in einen Container gepackt werden, üblicherweise ein StackPanel.

3.2.1 GroupBox (RadioBox)

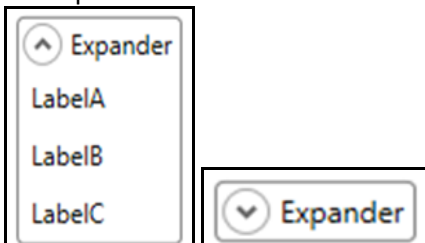
Die GroupBox wird hauptsächlich als RadioBox eingesetzt:



```
<GroupBox Name="grpGender" Header="Gender">
  <StackPanel>
    <RadioButton Name="rdoMale" Content="Male"/>
    <RadioButton Name="rdoFemale" Content="Female"/>
  </StackPanel>
</GroupBox>
```

3.2.2 Expander

Ein Expander funktioniert wie eine einklappbare GroupBox.

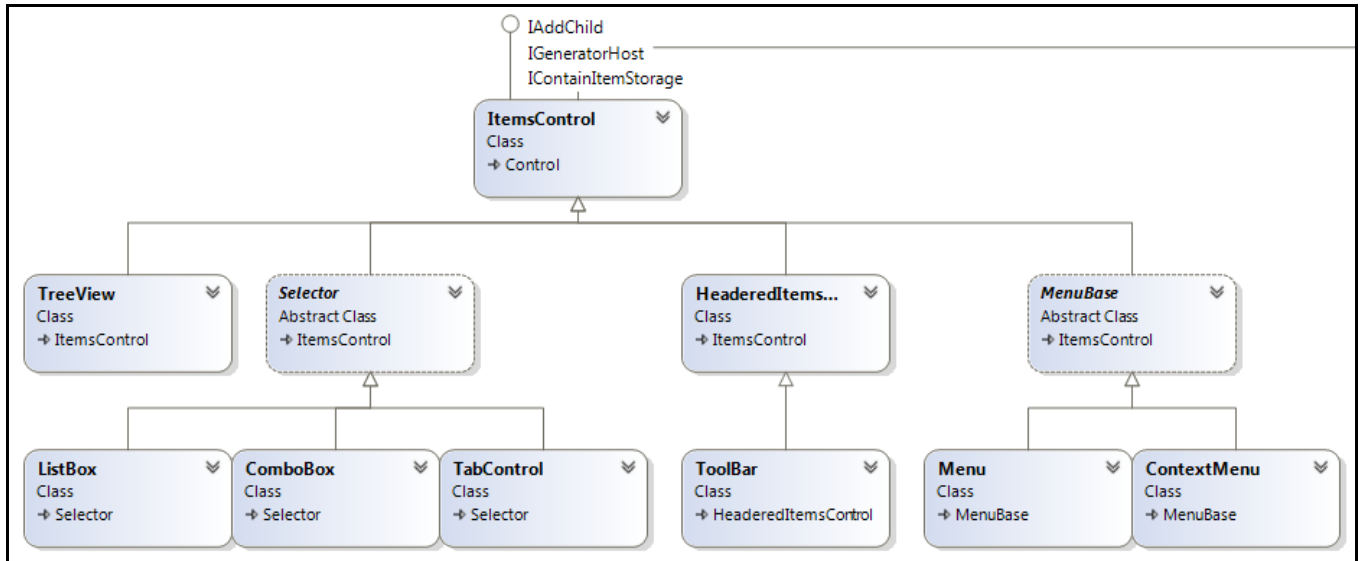


Allerdings ist ein Rahmen nicht automatisch vorhanden, dieser muss extra notiert werden:

```
<Border BorderThickness="1" CornerRadius="3" Margin="0,5,0,0">
  <Border.BorderBrush>
    <SolidColorBrush Color="Gray"/>
  </Border.BorderBrush>
  <Expander Name="expander" Header="Expander" IsExpanded="True">
    <StackPanel>
      <Label Content="LabelA" />
      <Label Content="LabelB" />
      <Label Content="LabelC" />
    </StackPanel>
  </Expander>
</Border>
```

4 ItemsControls

Controls, die von ItemsControl abgeleitet sind, haben eine Property **Items**, die mehrere Kindelemente aufnehmen kann.



Die einzelnen Kindelemente können direkt angegeben werden, weil die Klasse ItemsControl die **Items**-Property als **ContentProperty** definiert hat (wpf\src\Framework\System\Windows\Controls\ItemsControl.cs):

```

[DefaultEvent("OnItemsChanged"), DefaultProperty("Items")]
[ContentProperty("Items")]
[StyleTypedProperty(Property = "ItemContainerStyle", StyleTargetType = typeof(ItemsControl))]
[Localizability(LocalizationCategory.None, Readability = Readability.None)]
public class ItemsControl : Control, IAddChild, IGeneratorHost, IContainItemStorage
  
```

Die wichtigsten Vertreter sind:

- TreeView
- Selectors
 - ListBox
 - ComboBox
 - Tab
- Menu
- ToolBar
- StatusBar

Für fast alle ItemsControls gibt es ein entsprechendes Container-Element, z.B.

- **ComboBoxItem** für **ComboBox**
- **TreeViewItem** für **TreeView**
- ...

Den Inhalt kann man über die Property **Items** bzw. **ItemsSource** mit Daten befüllen. Über **ItemsSource** kann ein **IEnumerable** zugewiesen werden, jedoch darf die Items-Liste danach nicht mehr mit **Items.Add()** verändert werden.

4.1 ListBox/ComboBox

Kann man entweder im XAML oder per Code mit Werten befüllen:

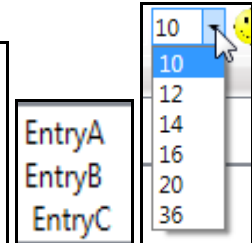
```

<ListBox Name="lstEntries">
  <ListBoxItem Content="EntryA"/>
  <ListBoxItem Content="EntryB"/>
  EntryC
</ListBox>
  
```

```

<ComboBox Name="cboFontName" Width="120" />
<ComboBox Name="cboFontSize" Width="40"/>
  
```

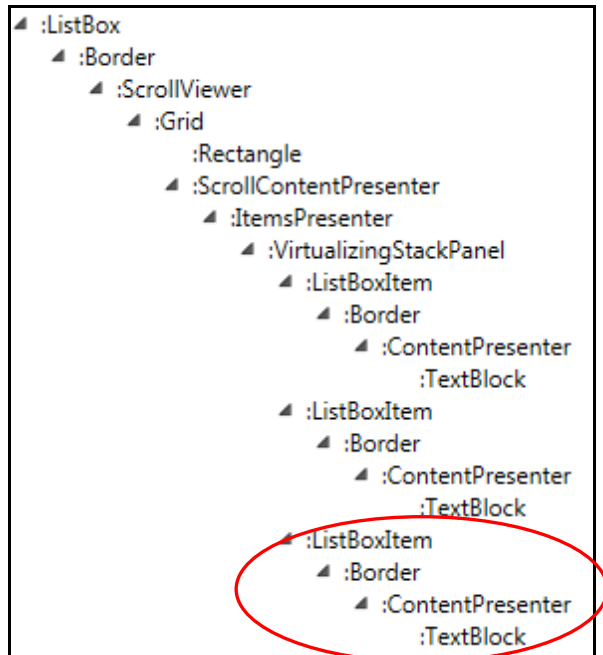
```
cboFontName.ItemsSource = Fonts.SystemFontFamilies.AsEnumerable();
foreach (var size in new double[] { 10,12,14,16,20,36})
{
    cboFontSize.Items.Add(size);
}
```



Befüllt man die ListBox wie oben gezeigt, muss man beim Iterieren auf die Kind-Typen aufpassen. Außerdem werden die Einträge leicht unterschiedlich angezeigt:

```
foreach (var item in lstEntries.Items) Console.WriteLine(item);
System.Windows.Controls.ListBoxItem: EntryA
System.Windows.Controls.ListBoxItem: EntryB
EntryC
```

Tatsächlich werden aber in beiden Fällen ListBoxItems erzeugt, wie man im Visual Tree sehen kann:



4.1.1 DisplayMemberPath

Oft hat man in den Items von ListBox/ComboBox komplexere Objekte hinzugefügt, möchte aber nur eine bestimmte Property davon anzeigen. Das kann man mit der Property **DisplayMemberPath** festlegen:

```
public class PersonWithId
{
    public int Id { get; set; }
    public string Name { get; set; }
}
```

```
<ListBox Name="lstPersons"
    DisplayMemberPath="Name"
    SelectionChanged="lstPersons_SelectionChanged" />
```

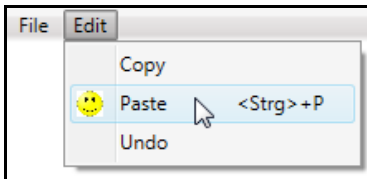
```
lstPersons.Items.Add(new PersonWithId { Id = 1, Name = "Hansi" });
lstPersons.Items.Add(new PersonWithId { Id = 2, Name = "Fitzi" });
lstPersons.Items.Add(new PersonWithId { Id = 3, Name = "Susi" });
lstPersons.Items.Add(new PersonWithId { Id = 4, Name = "Pepi" });
```

Wie wir später sehen werden, kann man die Anzeige mit Templates noch umfangreicher gestalten.

4.2 Menü

Menüs enthalten **MenuItem**-Elemente und können geschachtelt werden. Die Property **Header** ist dabei der Menütext. Ein Unterstrich gibt an, dass man damit mit der <Alt>-Taste und dem Buchstaben das Menü aufklappen bzw. den Befehl auslösen kann.

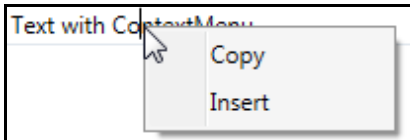
Links kann man ein Icon anzeigen, mit **InputGestureText** kann man rechts einen zusätzlichen String anzeigen.



```
<Menu DockPanel.Dock="Top">
  <Menu.Items>
    <MenuItem Header="_File">
      <MenuItem Header="_New" />
      <MenuItem Header="_Open" />
    </MenuItem>
    <MenuItem Header="_Edit">
      <MenuItem Header="_Copy" />
      <MenuItem Header="_Paste"
        Click="MenuItem_Click"
        InputGestureText="&lt;Strg&gt;+P">
        <MenuItem.Icon>
          <Image Source="smiley.jpg" Width="16" Height="16" />
        </MenuItem.Icon>
      </MenuItem>
      <MenuItem Header="_Undo" />
    </MenuItem>
  </Menu.Items>
</Menu>
```

4.3 ContextMenu

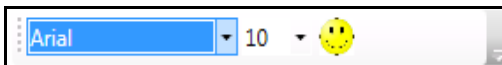
Derartige Menüs werden einem Control zugewiesen und poppen auf, wenn man mit der rechten Maustaste das Control anklickt. Es wird über ContextMenu zugewiesen, ansonsten verhält es sich genau wie ein normales Menu.



```
<TextBox Name="txtInput" Text="Text with ContextMenu">
  <TextBox.ContextMenu>
    <ContextMenu>
      <MenuItem Header="_Copy" Click="CopyTextClicked"/>
      <MenuItem Header="_Insert" />
    </ContextMenu>
  </TextBox.ContextMenu>
</TextBox>
```

4.4 ToolBar

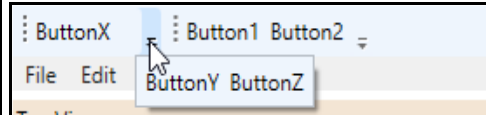
ToolBars werden üblicherweise bei einer App oben angezeigt und bestehen aus einer Anzahl von Schaltflächen (Button, ComboBox, CheckBox, ...). Ansonsten verhält es sich sehr ähnlich wie ein Menu.



```
<ToolBar DockPanel.Dock="Top">
  <ComboBox x:Name="cboFontName" Width="120"/></ComboBox>
  <ComboBox x:Name="cboFontSize" Width="40"/>
  <Button>
    <Image Source="smiley.jpg" Width="20" Height="20" />
  </Button>
</ToolBar>
```

ToolBars kann in einem sogenannten **ToolBarTray** gruppieren. Damit kann man dann die einzelnen ToolBars im Tray verschieben bzw. zusammenschieben (die Einträge werden dann bei Bedarf ausgeklappt).

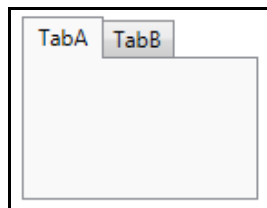
```
<ToolBarTray DockPanel.Dock="Top">
  <ToolBar>
    <Button Content="ButtonX"/>
    <Button Content="ButtonY"/>
    <Button Content="ButtonZ"/>
  </ToolBar>
  <ToolBar>
    <Button Content="Button1"/>
    <Button Content="Button2"/>
  </ToolBar>
</ToolBarTray>
```



Hinweis: einige Controls (z.B. Checkbox) sehen in der Toolbar anders aus als z.B. in einem StackPanel.

4.5 TabControl

Ein TabControl enthält Elemente vom Typ **TabItem** und wird wie erwartet dargestellt.



```
<TabControl Height="100">
  <TabItem Header="TabA"></TabItem>
  <TabItem Header="TabB"></TabItem>
</TabControl>
```

5 Decorator


Interessant ist vielleicht noch die Klasse **Decorator**. Diese hat nur eine Property, nämlich **Child**, das das Element enthält, das dekoriert werden soll. Wobei, wie vermutet, diese Property die **ContentProperty** ist und somit nicht angegeben werden muss.

```
[ContentProperty("Child")]  
public class Decorator : FrameworkElement, IAddChild
```

5.1 Border

Der wichtigste Vertreter der Decorator ist **Border**. Damit kann man Elemente „verzieren“.

```
<Border BorderBrush="Blue" BorderThickness="6" CornerRadius="6,1,3,1">  
    <TextBox Text="I am decorated"/>  
</Border>
```



5.2 ViewBox

Diese Klasse ist ebenfalls von Decorator abgeleitet und hat somit nur ein Kind-Element. ViewBox wird verwendet, um das (einzige) Kind-Element auf den verfügbaren Platz auszudehnen bzw. zu skalieren. Dazu verwendet man die Property **Stretch**, die angibt, ob bzw. wie das Kind-Element skaliert werden soll.

Stretch	Uniform
StretchDirecti...	None
Style	Fill
	Uniform
	UniformToFill