

A decorative graphic on the left side of the slide, consisting of a grid of squares in various shades of gray and blue, arranged in a pattern that tapers off to the right.

.NET Windows Presentation Foundation (WPF)

© J. Heinzlreiter
Version 2.3

Überblick

- Historischer Überblick
- Architektur
- XAML
- Aufbau von WPF-Programmen
- Klassen der WPF
- Layout-Klassen
- Ressourcen
- Routed Events und Kommandos
- Datenbindung
- MVVM
- Styles und Templates
- Animation und Grafik

Historische Entwicklung (im Windows-Bereich) (1)

■ Win32-Anwendungen

- Kennzeichen: Direkte Verwendung von Win32-APIs: *user32.dll*, *kernel32.dll*, *gdi32.dll*.
- Layout: C/C++-Quelltext, keine Werkzeugunterstützung.
- Repräsentation des Layouts: Quelltext.
- Methodik: Hauptereignisschleife, Funktionszeiger, Windows-Nachrichten.
- Grafikprogrammierung: GDI

■ Microsoft Foundation Classes (MFC)

- Kennzeichen: Dünne Schicht über Win32-APIs, Application-Framework.
- Layout: Quelltext, einfache Werkzeugunterstützung.
- Methodik: OO-Konzepte (Vererbung und dynamische Bindung), Makros (Message-Maps).
- Repräsentation des Layouts: Quelltext, Ressourcen.
- Grafikprogrammierung : GDI

Historische Entwicklung (im Windows-Bereich) (2)

- VB (≤ 6)
 - Kennzeichen: Gute Abstraktion der Win32-API, einfache Integration von COM-Komponenten, proprietäre Sprache.
 - Layout: grafisches Designer-Werkzeug, einfache Verwendbarkeit von ActiveX-Controls (anfangs VBX-Controls).
 - Repräsentation des Layouts: proprietäres (Text-)Format.
 - Methodik: Registrierung von Callback-Funktionen, Auslagerung der Geschäftslogik in COM-Komponenten.
 - Grafikprogrammierung: GDI
 - Vorzüge:
 - GUI-Entwicklung wesentlich vereinfacht (Hauptgrund für Popularität von VB6)
 - Unterstützung komponentenorientierter SW-Entwicklung.
 - Nachteile:
 - Proprietäre Programmiersprache für größere Anwendungen unbrauchbar.
 - Kein Framework.

Historische Entwicklung (im Windows-Bereich) (3)

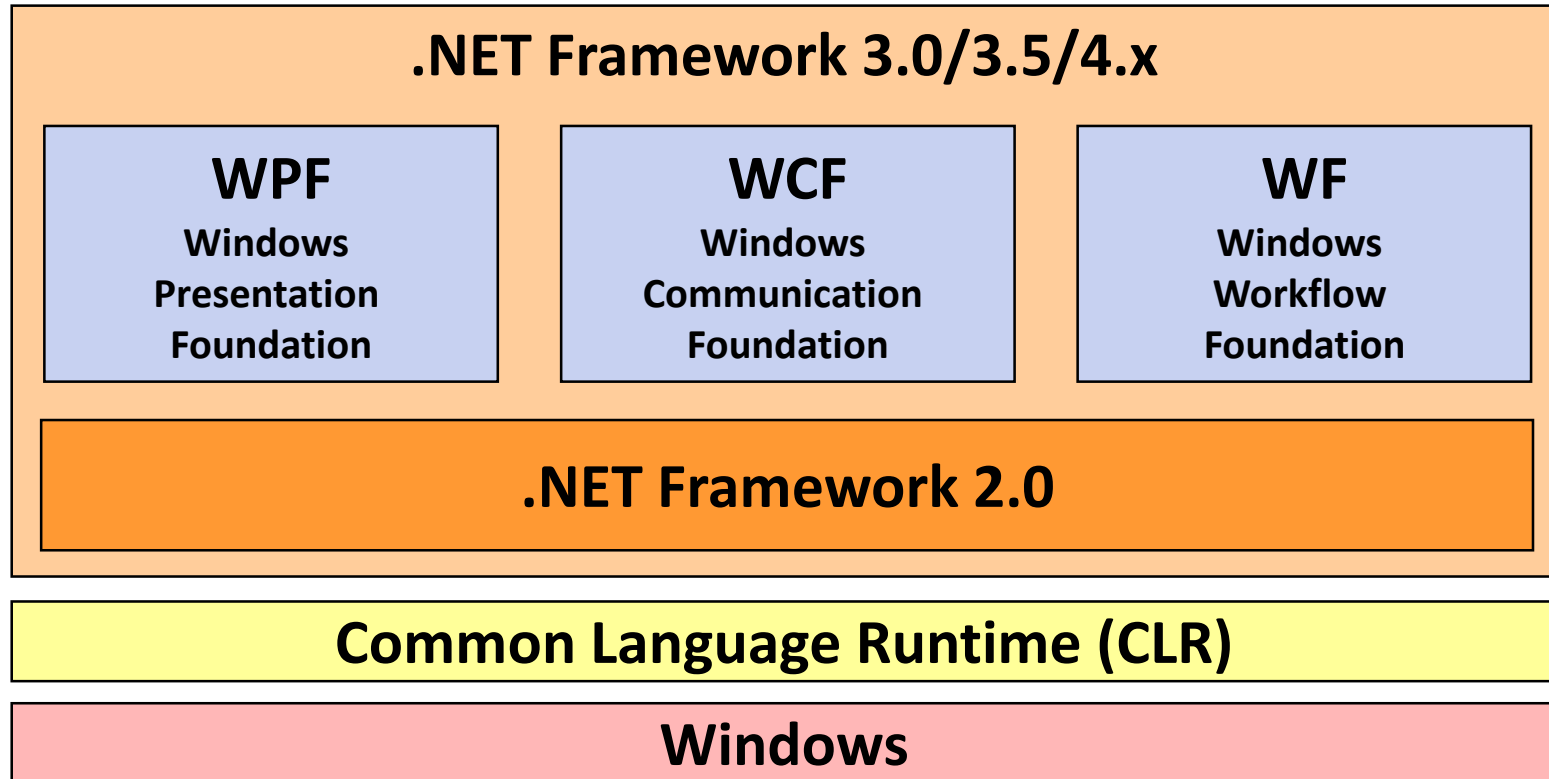
■ Windows Forms

- Layout: grafisches Design-Werkzeug, Möglichkeit zur Erweiterung bzw. Neuentwicklung von Steuerelementen.
- Repräsentation des Layouts: generierter Quelltext.
- Methodik: Ereignisbehandlung über Delegates, einfache Anbindung der Geschäftslogik.
- Grafikprogrammierung: GDI+
- Vorzüge:
 - Volle Integration in das .NET-Framework,
 - exzellentes Design-Werkzeug,
 - Entwicklung von vollwertigen Steuerelementen ist einfach, umfangreiches Angebot an Komponenten.
- Nachteile:
 - Unflexibles Layoutmanagement,
 - keine konsequente Trennung von Layout und Code,
 - Grafikfähigkeiten moderner PCs werden nicht genutzt.

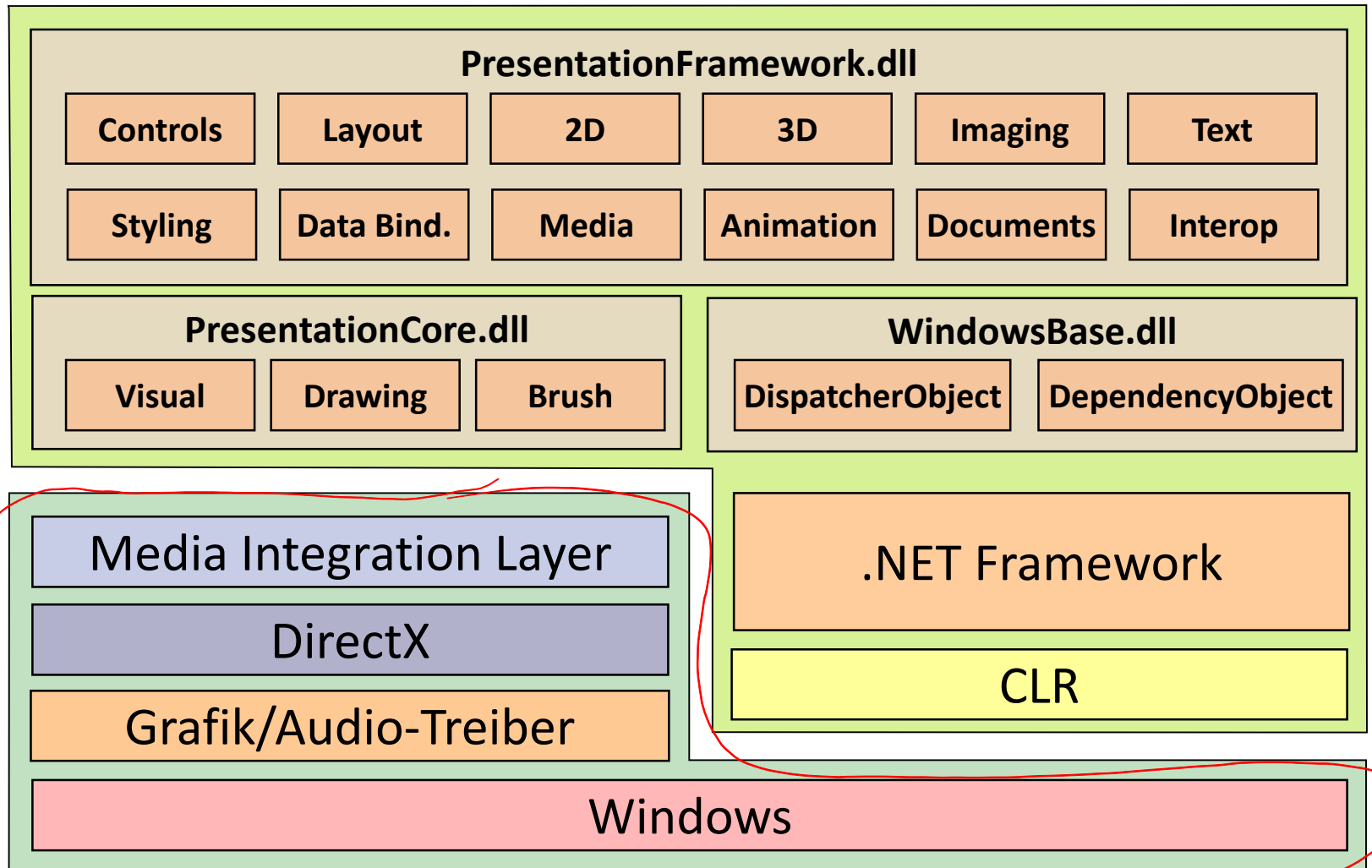


Architektur

.NET Framework



Architektur der WPF



Neuerungen in der WPF

- Deklarative Programmierung (XAML)
- Einheitliche API:
 - 2D-Grafik (ersetzt GDI, GDI+),
 - 3D-Grafik (deckt Teilbereiche von Direct3D bzw. OpenGL ab),
 - UI (ersetzt user32.dll bzw. Windows Forms),
 - Bild und Ton (DirectShow).
- Vektor-Grafik
- Neues Programmiermodell für Grafikanwendungen
- Starke Unterstützung von Text-Dokumenten
- Verhalten und Aussehen von Steuerelementen sind voneinander getrennt (Styles und Templates).
- Neues Konzept zur Datenbindung

Deklarative Programmierung

- Benutzeroberflächen können in *XAML* (eXtensible Application Markup Language) beschrieben werden.

```
<Window x:Class="XamlExperiments.SimpleDialog"
  xmlns="http://.../xaml/presentation"
  xmlns:x="http://.../xaml"
  Title="XamlExperiments" Height="107" Width="200">
  <Grid>
    <Label Height="25" HorizontalAlignment="Left"
      Margin="20,15,0,0" Name="label" VerticalAlignment="Top"
      Width="60">Name:</Label>
    <TextBox Height="25" Margin="80,15,10,0" Name="textBox"
      VerticalAlignment="Top" Padding="5,5,5,5">Hallo</TextBox>
    <Button Margin="0,0,0,10" Name="button" Height="23"
      VerticalAlignment="Bottom" HorizontalAlignment="Center"
      Width="80">Ok</Button>
  </Grid>
</Window>
```



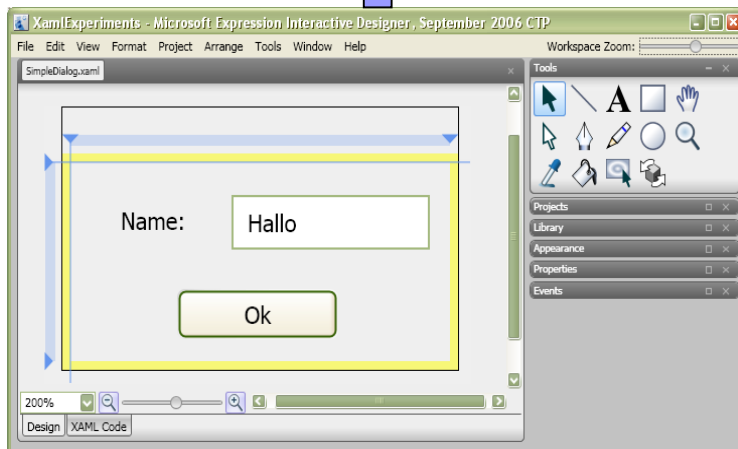
Trennung von Layout und Code

SampleDialog.xaml

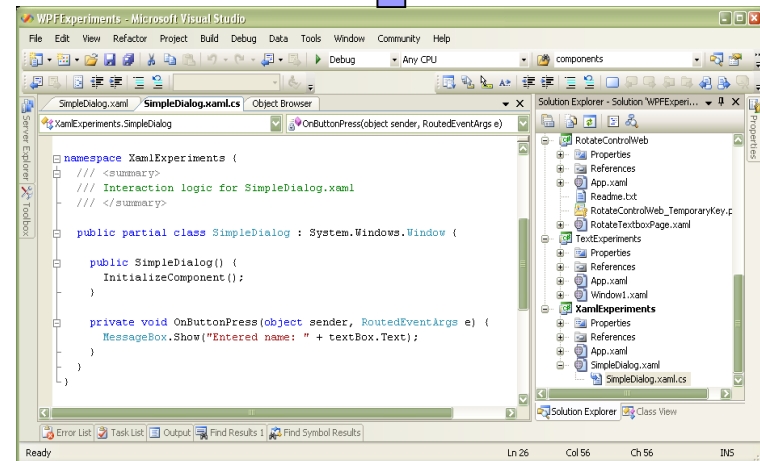
```
<Window x:Class="SimpleDialog"
  <Grid>
    <TextBox Name="textBox" ...>
      ...
    </TextBox>
    <Button Name="button" ...
      Click="OnButtonPressed">
    </Button>
  </Grid>
</Window>
```

SampleDialog.xaml.cs

```
partial class SimpleDialog : Window {
  public SimpleDialog() {
    InitializeComponent();
  }
  private void OnButtonPressed(
    object sender,
    RoutedEventArgs e) {
    textBox.Text = ...;
  }
}
```



XAML Designer (z.B MS Expression Blend)



Visual Studio

Das Übersetzungsmodell

```
<Window x:Class="SimpleDialog"
  <Grid>
    <TextBox Name="textBox" ...>
      ...
    </TextBox>
    <Button Name="button" ...
      Click="OnButtonPressed">
    </Button>
  </Grid>
</Window>
```

SampleDialog.xaml

```
partial class SimpleDialog : Window {
  public SimpleDialog() {
    InitializeComponent();
  }
  private void OnButtonPressed(
    object sender,
    RoutedEventArgs e) {
    textBox.Text = ...;
  }
}
```

SampleDialog.xaml.cs

```
partial class SimpleDialog : Window {
  internal Button button;
  internal Textbox textBox;
  public InitializeComponent() {
    Application.LoadComponent(this,
      new Uri("simpledialog.baml", ...));
  }
}
```

SampleDialog.g.cs

```
Simpdialog.baml: 0c 00 FF ...
.class public SimpleDialog extends Window {
  .field assembly class Button button;
  .field assembly class Textbox textBox;
  .method public void InitializeComponent() {}
  .method private void OnButtonPress(...) {}
}
```

SampleDialog.dll



XML Application Markup Language XAML

Was ist XAML?

- XAML ist eine XML-Sprache zur Beschreibung und Initialisierung von .NET-Objektgraphen.
 - WPF: Beschreibung von Benutzeroberflächen
 - WF: Beschreibung von Workflows
- Abbildung:
 - CLR-Namenräume → XML-Namenräume (mithilfe des Attributs *XmlnsDefinitionAttribute*).
 - Klassen → XML-Elemente
 - Properties → XML-Attribute
 - Registrierung von Ereignisbehandlungsmethoden → XML-Attribute.
 - Es existieren zahlreiche Konverter, die Zeichenketten (Werte von XML-Attributen) in die passenden CLR-Datentypen umwandeln.

Abbildung XAML → .NET-Konstrukte

.NET-Framework

```
[assembly:XamlnsDefinition(  
    "http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
    "System.Windows.Controls")]  
  
namespace System.Windows.Controls {  
    public class Button {  
        public object Content { get { ... } set { ... } }  
        public event RoutedEventHandler Click;  
    }  
}
```

Alle WPF-Namenräume sind auf "http://.../xaml/presentation" abgebildet.

XAML

```
<Button xmlns=  
    "http://schemas.microsoft.com/  
    winfx/2006/xaml/presentation"  
    Content="OK"  
    Click="button_Click"/>
```

C#-Code

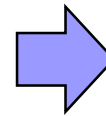
```
using Sytem.Windows.Controls;  
  
Button b = new Button();  
b.Content = "OK";  
b.Click += new  
    RoutedEventHandler(button_Click);
```

In beiden Fällen wird derselbe Objektgraph erzeugt.

Eigenschaftselemente (Property Elements)

- Viele Properties können komplexe Datentypen aufweisen, z. B. die Property *Content* von *Button*:

```
Button btn = new Button();  
ImageSource bitmap = new BitmapImage(new Uri(...));  
Image image = new Image { Source = bitmap,  
                           Height=50, Width=80 };  
btn.Content = image;
```



- Komplexe Datentypen sind nicht direkt auf XAML-Attribute abbildbar:

```
<Button Content="<Image .../>" /> <!-- funktioniert nicht! -->
```

- Eigenschaftselemente können komplexe Datentypen aufnehmen:

```
<Button>  
  <Button.Content>  
    <Image Source="smily.jpg" Height="50" Width="80" />  
  </Button.Content>  
</Button>
```


Kindelemente

- Benutzeroberflächen sind Bäume von Steuerelementen.
- In der WPF können Eigenschaften von Steuerelementen komplex strukturierte Werte zugewiesen werden (Objektbäume).
- Einfache Steuerelemente mit Kindelement:

```
<Button>  
  <Button.Content>  
    <Image Source="smily.jpg" />  
  </Button.Content>  
</Button>
```

```
<ToolTip>  
  <ToolTip.Content>  
    <StackPanel>...</StackPanel>  
  </ToolTip.Content>  
</ToolTip>
```

- Manchen Steuerelementen können auch Behälter mit Kindelementen zugewiesen werden:

```
<ListBox>  
  <ListBox.Items>  
    <ListBoxItem Content="Item 1"/>  
    <ListBoxItem Content="Item 2"/>  
  </ListBox.Items>  
</ListBox>
```

```
<ResourceDictionary>  
  <Image x:Key="myImage"  
    Source="pic.gif"/>  
  <SolidColorBrush  
    x:Key="myBrush"  
    Color="Red" />  
</ResourceDictionary >
```

Kindelemente – Content-Properties

- Viele Steuerelemente haben eine sogenannte *Content-Property*:

```
[ContentProperty("Content")]  
public class Button {  
    public object Content { ... }  
}
```

```
[ContentProperty("Items")]  
public class ListBox {  
    public ItemsCollection Items { ... }  
}
```

- Werte für Content-Properties können direkt als Kindelemente angegeben werden:

```
<Button>  
    <Image Source="smily.jpj" />  
</Button>
```



wird in Property *Content* gespeichert

```
<ListBox>  
    <ListBoxItem Content="Item 1"/>  
    <ListBoxItem Content="Item 2"/>  
</ListBox>
```



werden in Property *Items* gespeichert

Typkonverter

- In vielen Fällen können einfache Typen in XAML nur sehr schwerfällig beschrieben werden:

```
<Button.Background>  
  <SolidColorBrush>  
    <SolidColorBrush.Color>  
      <Color A="255" R="255" G="255" B="255"/>  
    </SolidColorBrush.Color>  
  </SolidColorBrush>  
</Button.Background>
```

- Typkonverter tragen zur Vereinfachung der XAML-Beschreibung bei:

```
<Button Background="White" />
```

- Typkonverter sind von *TypeConverter* abgeleitet und werden mit dem Attribut *TypeConverterAttribute* mit einem Typ oder einer Property verbunden:

```
[TypeConverter(typeof(BrushConverter))]  
public abstract class Brush : ... {  
  ...  
}
```

Markup Extensions

- Mit Markup-Extensions können Attributwerte flexibel definiert werden:

```
<Element SomeProperty = "{MyMarkupExtension Prop1=Value}">
```

- Die Markup-Extension fasst Parameter zusammen, die Property-Wert bestimmen.
- *ProvideValue* liefert den Wert, welcher der Property zugewiesen wird.

```
public MyMarkupExtension : MarkupExtension {  
    public override object ProvideValue(...);  
    public object Prop1 { ... }  
}
```

- Wichtigste Anwendungen:

- x:Static[Extension]
- Binding
- StaticResource[Extension]
- DynamicResource[Extension]

- Beispiel:

```
<Button Height="{x:Static Member = SystemParameters.IconHeight}"/>
```

```
<TextBox Text="{Binding Path=LastName}"/>
```

Einbindung von .NET-Klassen

- Mit XAML können Objekte beliebiger .NET-Klassen erzeugt werden.
- Die Initialisierung erfolgt über Attribute (→ Properties) bzw. Typkonverter.

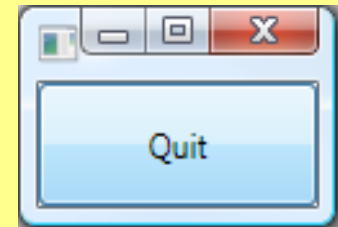
```
namespace PersonAdmin {  
    public class Person {  
        public string FirstName { get; set; }  
        public string LastName { get; set; }  
        public DateTime DateOfBirth { get; set; }  
    }  
}
```

```
<Window xmlns="http://.../xaml/presentation"  
        xmlns:x="http://.../xaml"  
        xmlns:pa="clr-namespace:PersonAdmin"  
        xmlns:sys="clr-namespace:System;assembly=mscorlib">  
    <Window.Resources>  
        <pa:Person x:Key="mayr" FirstName="Heinz" LastName="Mayr"  
                  DateOfBirth="1980-12-10" />  
        <sys:Double x:Key="dblVal">20.17</s:Double>  
    </Window.Resources>  
</Window>
```

Aufbau von WPF-Programmen

„Hello World“ mit der WPF (Code)

```
public class HelloWorld : Window {  
    private Button btn;  
    public HelloWorld() {  
        btn = new Button() { Content = "Quit" };  
        btn.Click += new RoutedEventHandler(OnClick);  
        this.AddChild(btn);  
  
        this.Title = "WPF Hello App";  
        this.Width = 120;  
        this.Height = 80;  
    }  
  
    void OnClick(object sender, RoutedEventArgs e) {  
        this.Close();  
    }  
  
    [STAThread]  
    static void Main(string[] args) {  
        Application app = new Application();  
        app.Run(new HelloWorld());  
    }  
}
```



Die Klassen *Application* und *Window*

- Das Singleton *Application* verwaltet die Fenster und die Hauptereignisschleife einer WPF-Anwendung.

```
Application app = new Application();
```

- Methode *Run*: Start der Hauptereignisschleife

```
Window win = new Window();  
app.Run(win);
```

$\hat{=}$

```
Window win = new Window();  
win.Show();  
app.Run();
```

- Methode *ShutDown*: Beenden der Hauptereignisschleife.
- Property *MainWindow*: Festlegung des Hauptfensters einer Anwendung.
- Property *ShutdownMode*:
OnLastWindowClose/OnMainWindowClose/OnExplicitShutdown
- Ereignis *Startup*: Hauptereignisschleife wurde gestartet.
- Ereignis *SessionEnding*: Benutzer loggt sich aus Windows aus.

„Hello World“ mit der WPF (XAML)

<Application

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

<Window

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

```
x:Class="HelloWindow"
```

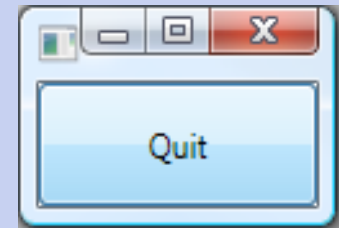
```
Title="WPF Hello App" Width="120" Height="80">
```

```
<Button Click="OnClick">
```

```
Quit
```

```
</Button>
```

```
</Window>
```



```
public partial class HelloWindow : Window {  
    private void OnClick(object sender, RoutedEventArgs e) {  
        this.Close();  
    }  
}
```

Das „Code-Behind“-Konzept

HelloWindow.xaml:

```
<Window xmlns=".../xaml/presentation" xmlns:x=".../xaml"
  x:Class="HelloWindow" >
  <Button Name="btnQuit" Click="OnClick">Quit</Button>
</Window>
```

HelloWindow.g.cs: dieser Teil der Klasse wird vom XAML-Compiler generiert

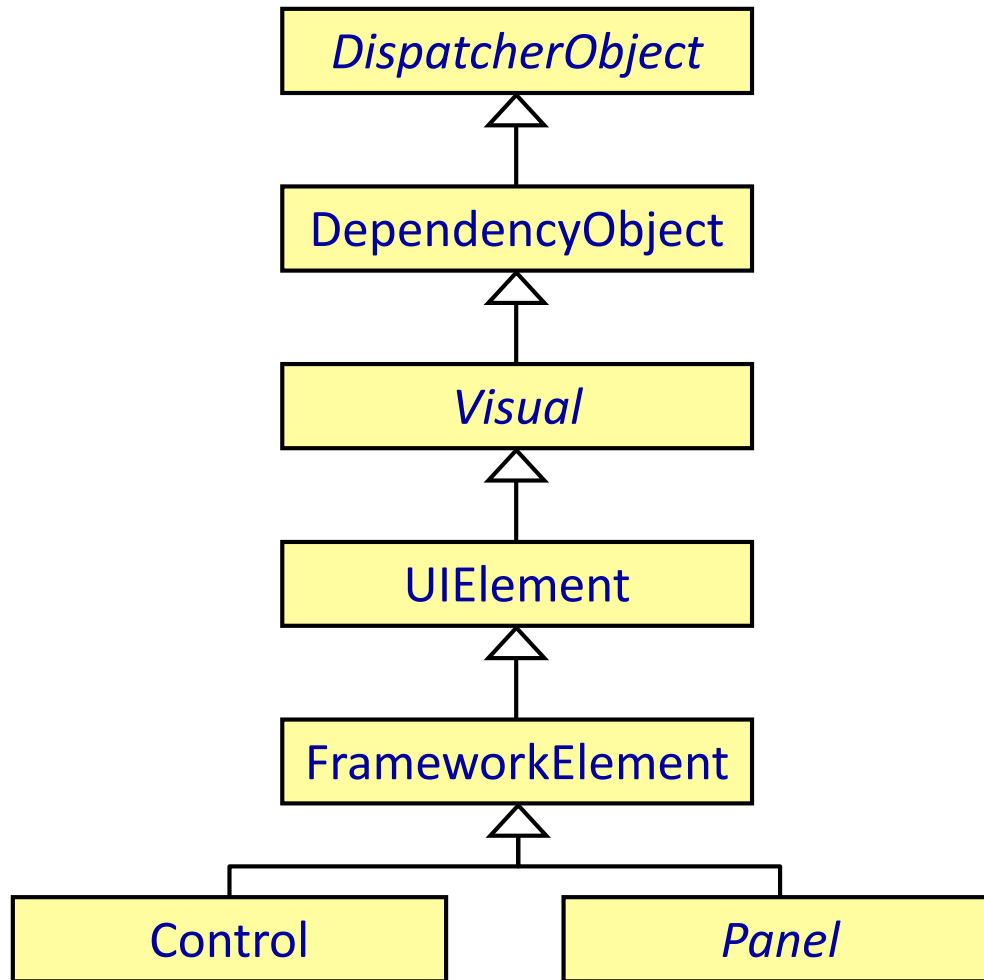
```
public partial class HelloWindow : Window {
    internal Button btnQuit;
    public void InitializeComponent() {
        Application.LoadComponent(this, new Uri("helloWindow.xaml", ...))
        btnQuit.Click += new RoutedEventHandler(this.OnClick); // vereinfacht
    }
}
```

HelloWindow.xaml.cs: dieser Teil der Klasse kann erweitert werden

```
public partial class HelloWindow : Window {
    public HelloWindow() { InitializeComponent(); }
    private void OnClick(object sender, RoutedEventArgs e) { ... }
}
```

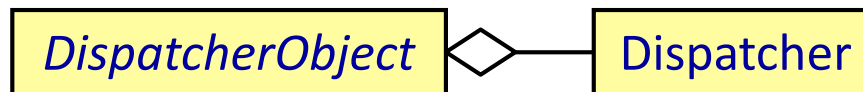
Die Klassen der WPF

Basisklassen der WPF



DispatcherObject

- Alle WPF-Klassen sind *nicht* Thread-sicher.
- Der mit jedem *DispatcherObject* assoziierte *Dispatcher* stellt jeden Methodenaufruf in die Ereigniswarteschlange, die vom UI-Thread abgearbeitet wird.



```
void MethodExecutedInSomeThread() {  
    // Variante 1: aktueller Thread blockiert bis Methode im UI-Thread terminiert  
    label.Dispatcher.Invoke(ChangeLabelText);  
    // Variante 2: aktueller Thread wird nicht blockiert  
    label.Dispatcher.InvokeAsync(ChangeLabelText);  
}  
void ChangeLabelText() { // wird im UI-Thread ausgeführt  
    label.Content = "new label text";  
}
```

await/async und die WPF

- await/async vereinfacht die asynchrone Programmierung enorm.
- Der Synchronisationskontext der WPF (Dispatcher-SynchronizationContext) sorgt dafür, dass nach Aufruf einer asynchronen Methode im UI-Thread, die Ausführung wieder im UI-Thread fortgesetzt wird.

```
void async Task<object> MethodExecutedInSomeThreadAsync() {  
    ...  
}  
void SomeEventHandler() { // wird im UI-Thread ausgeführt  
    var result = await MethodExecutedInSomeThreadAsync();  
    label.Content = result.ToString(); // wird im UI-Thread ausgeführt.  
}
```

→ hier darf auf keinen Fall blockiert werden

DependencyObject

- Ermöglicht die Definition von *Dependency Properties*.
- Einfache .NET-Properties verwalten nur einen Wert.
- *Dependency Properties* haben zusätzliche Eigenschaften:
 - Es kann ein Standardwert definiert werden.
 - Der Wert kann von Elternelementen im Steuerelementbaum geerbt werden.
 - Bei Wertänderungen werden Ereignisse gefeuert.
- *Dependency Properties* werden zur Realisierung zahlreicher WPF-Konzepte benötigt:
 - Styling
 - Datenbindung
 - Animationen
- Da nur von den Standardwerten abweichende Werte gespeichert werden, wird auch der Speicherplatzbedarf reduziert.

Implementierung von Dependency Properties

```
public class Control: FrameworkElement {
    // Deklaration der Dependency Property als Klassendatenkomponente
    public static readonly DependencyProperty FontSizeProperty;

    static Control() {
        // Festlegung der Eigenschaften der Property
        Control.FontSizeProperty = DependencyProperty.Register(
            "FontSize",           // Name
            typeof(double),      // Propertytyp
            typeof(Control),     // Besitzer
            new FrameworkPropertyMetadata(12.0,           // Standardwert
                FrameworkPropertyMetadataOptions.AffectsRender |
                FrameworkPropertyMetadataOptions.AffectsMeasure |
                FrameworkPropertyMetadataOptions.Inherits);

        ...
    }

    // Definition einer .NET-Property zur Verwaltung des Property-Werts
    public double FontSize {
        get { return (double)GetValue(Control.FontSizeProperty); }
        set { SetValue(Control.FontSizeProperty, value); }
    }
}
```

ändern, der dependency Properties

Anwendungsbeispiele für *Dependency Properties*

- **Beispiel 1:** Vererbung von Property-Werten

```
<Window FontSize="20">  
  <StackPanel>  
    <Button Content="My Button">  
      <Label Content="Some Label" />  
    </StackPanel>  
</Window>
```

- **Beispiel 2:** Styling von Steuerelementen

```
<Style TargetType="{x:Type Button}">  
  <Setter Property="Foreground" Value="Green"/>  
  <Style.Triggers>  
    <Trigger Property="IsMouseOver" Value="True">  
      <Setter Property="FontSize" Value="25" />  
    </Trigger>  
  </Style.Triggers>  
</Style>
```

- **Beispiel 3:** „Abhören“ von Änderungen des Property-Werts

```
DependencyPropertyDescriptor propDescr =  
    DependencyPropertyDescriptor.FromProperty(  
        UIElement.IsMouseOverProperty, typeof(UIElement));  
propDescr.AddValueChanged(button, (object source, EventArgs e) => ...);
```

Attached Properties

- *Attached Properties* sind spezielle *Dependency Properties*, die Objekten beliebiger Klassen zugeordnet werden können.
- Diese Klassen dienen lediglich als Datenbehälter, verwendet werden diese *Attached Properties* von anderen Klassen.
- Typische Anwendung: Layout-Klassen
 - Positionsparameter müssen bei den Kindelementen gespeichert werden.
 - In Kindelementen können nicht für alle möglichen Layout-Klassen Properties vorgesehen werden (keine Erweiterungsmöglichkeit).

- Verwendung in XAML

```
<DockPanel>  
  <Button Name="button" DockPanel.Dock="Top">My Button</Button>  
</DockPanel>
```

werden in den Kindelement gespeichert

- Verwendung im Code

```
DockPanel.SetDock(button, Dock.Bottom);
```

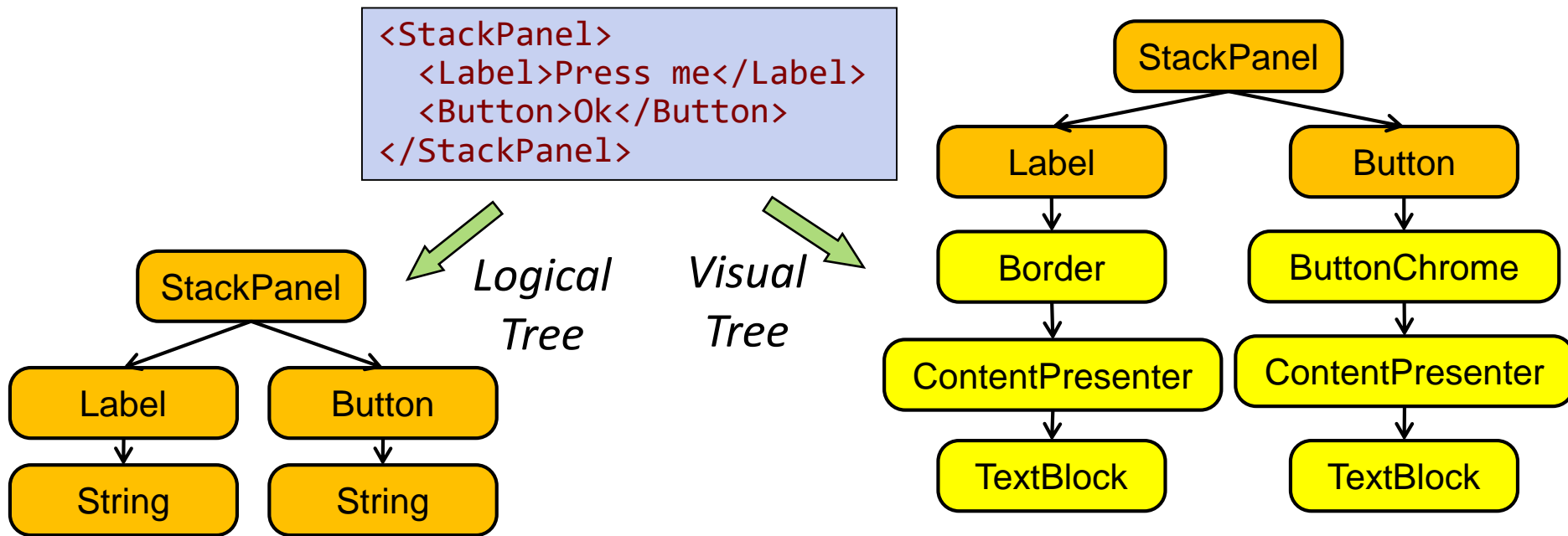
Implementierung von *Attached Properties*

```
class DockPanel : Panel {  
    public static readonly DependencyProperty DockProperty;  
  
    static DockPanel() {  
        DockProperty = DependencyProperty.RegisterAttached(  
            "Dock", typeof(Dock), typeof(DockPanel),  
            new FrameworkPropertyMetadata(...));  
    }  
  
    public static Dock GetDock(UIElement element) {  
        return (Dock)element.GetValue(DockPanel.DockProperty);  
    }  
  
    public static void SetDock(UIElement element, Dock dock) {  
        element.SetValue(DockPanel.DockProperty, dock);  
    }  
}
```

- Die Deklaration der Property erfolgt in *DockPanel*.
- Die Property-Werte werden aber in den Kindelementen gespeichert.

Visual (und Visual3D)

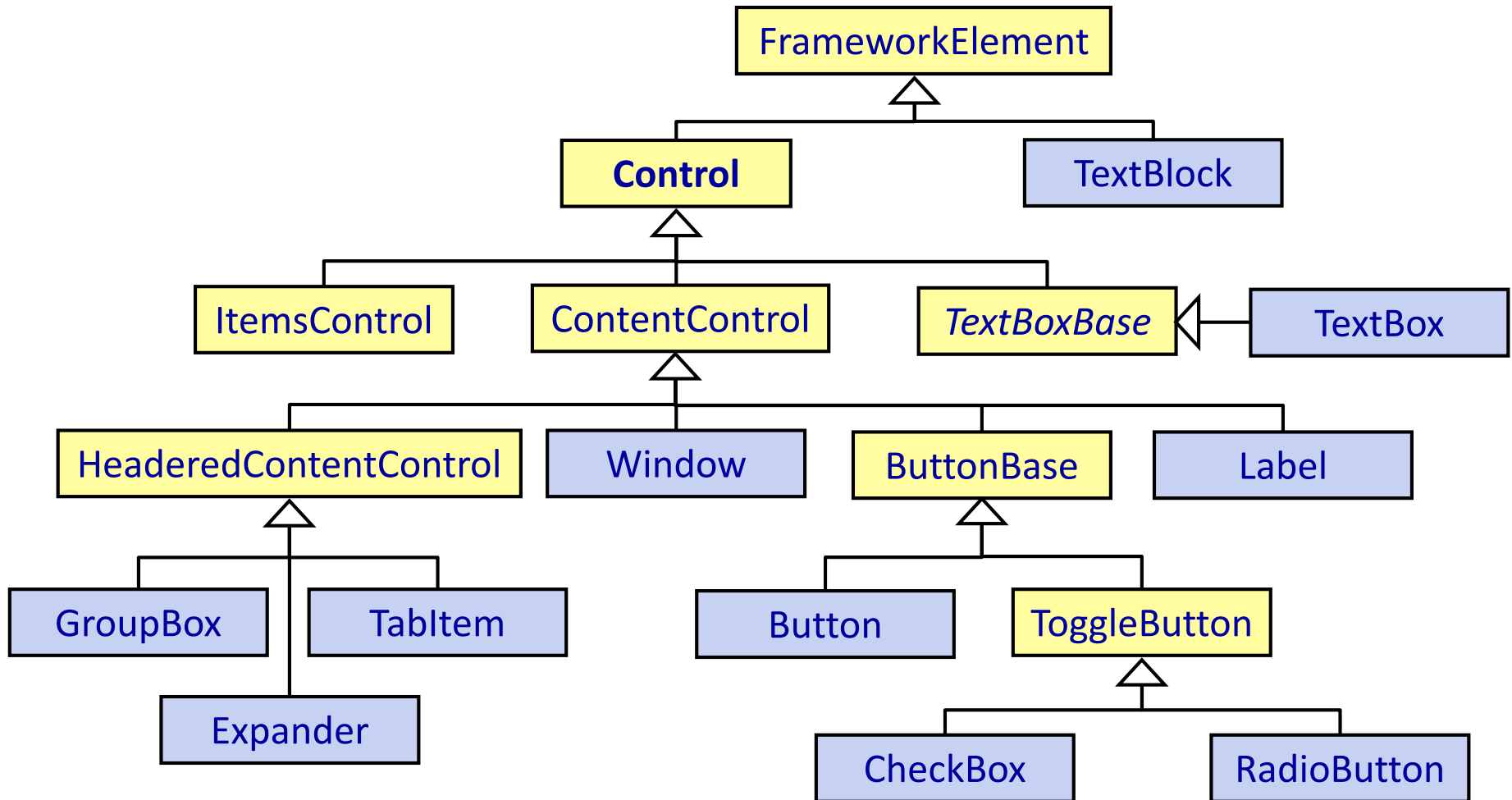
- *Visual* stellt Funktionalität zum Zeichnen von Steuerelementen zur Verfügung.
- Der *Visual Tree* wird durchlaufen und mit Hilfe von DirectX am Anzeigegerät dargestellt (im *Media Integration Layer*).



Andere Basisklassen der WPF

- *UIElement*:
 - Verarbeitung von Benutzereingaben (*Routed Events*)
 - Unterstützung für Layoutsystem (Positionierung, Größenbestimmung)
- *FrameworkElement*:
 - Datenbindung
 - Styling
 - Lokale Ressourcen
- **Control**
 - Basisklasse für alle Elemente, mit denen Benutzer interagieren kann.
 - Unterstützung von Styles und Control-Templates.
- **Panel**
 - (Abstrakte) Basisklasse für alle Layout-Manager.

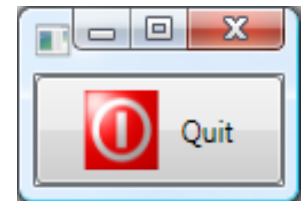
Die wichtigsten Steuerelemente



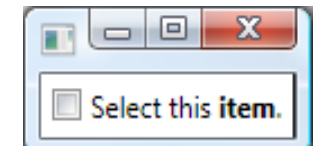
ContentControl

- Steuerelemente dieser Gruppe können *ein* beliebiges Kindelement enthalten (nicht nur eine Zeichenkette).
- Beispiele:

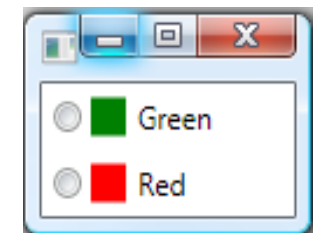
```
<Button Name="btnQuit">  
  <StackPanel Orientation="Horizontal" Margin="3">  
    <Image Source="quit.jpg"/>  
    <TextBlock Margin="10,0,0,0" Text="Quit"  
      VerticalAlignment="Center" />  
  </StackPanel>  
</Button>
```



```
<CheckBox>  
  <TextBlock>Select this <Bold>item</Bold>.</TextBlock>  
</CheckBox>
```



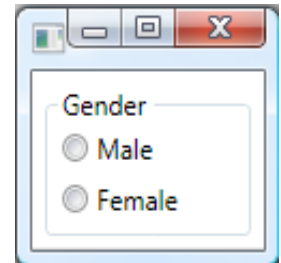
```
<StackPanel>  
  <RadioButton Margin="5">  
    <StackPanel Orientation="Horizontal">  
      <Rectangle Height="15" Width="15" Fill="Green"></Rectangle>  
      <TextBlock Padding="5,0,0,0">Green</TextBlock>  
    </StackPanel>  
  </RadioButton>  
  <RadioButton>...</RadioButton>  
</StackPanel>
```



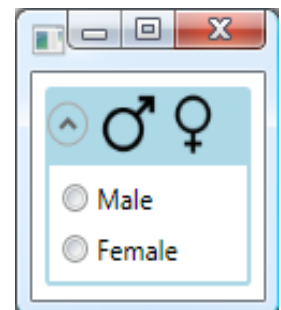
HeaderedContentControl

- Steuerelement besitzt neben dem Kindelement eine *Überschrift*.
- Beispiele:

```
<GroupBox Header="Gender" Margin="5">  
  <StackPanel>  
    <RadioButton Margin="3">Male</RadioButton>  
    <RadioButton Margin="3">Female</RadioButton>  
  </StackPanel>  
</GroupBox>
```



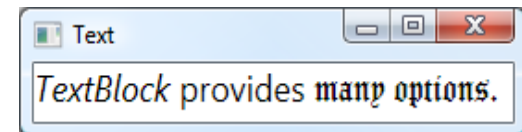
```
<Expander IsExpanded="True" Background="LightBlue" Margin="5">  
  <Expander.Header>  
    <StackPanel Orientation="Horizontal">  
      <Image Source="male.png" Height="30"/>  
      <Image Source="female.png" Height="30"/>  
    </StackPanel>  
  </Expander.Header>  
  <Border Margin="2" Background="White" Padding="3">  
    <StackPanel>  
      <RadioButton Margin="3">Male</RadioButton>  
      <RadioButton Margin="3">Female</RadioButton>  
    </StackPanel>  
  </Border>  
</Expander>
```



Steuerelemente zur Darstellung und Bearbeitung von Text

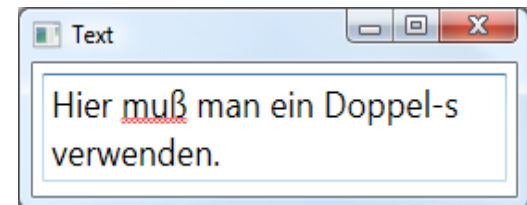
- *TextBlock*: Darstellung von Text in verschiedenen Fonts und mit diversen Hervorhebungsarten.

```
<TextBlock Margin="5" FontSize="18">  
  <Italic>TextBlock</Italic> provides  
  <Run FontFamily="Old English Text MT">many options.</Run>  
</TextBlock>
```



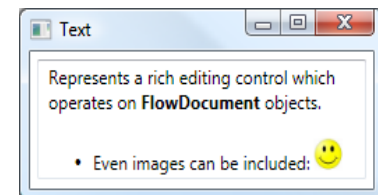
- *TextBox*: Erfassung und Bearbeitung von Text.

```
<TextBox Margin="5" FontSize="18"  
  xml:lang="de-AT" SpellCheck.IsEnabled="True"  
  SpellCheck.SpellingReform="Postreform"  
  AcceptsReturn="True" />
```

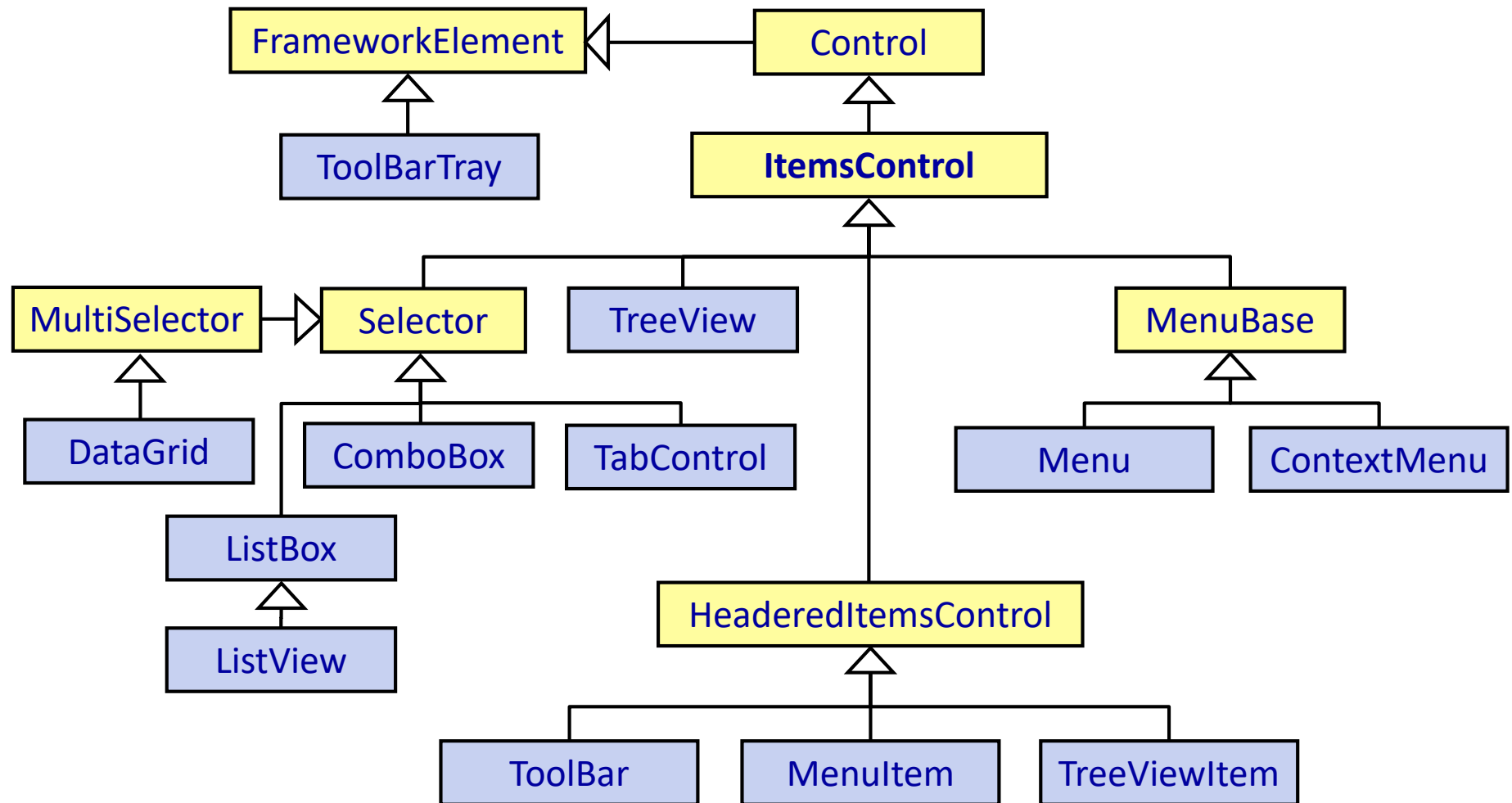


- *RichText*: Erfassung und Bearbeitung von *formatiertem Text*.

```
<RichTextBox Margin="5">  
  <FlowDocument>  
    <Paragraph>Represents ... <Bold>FlowDocument</Bold> ... </Paragraph>  
    <List>  
      <ListItem><Paragraph>Even ... <Image ... /></Paragraph></ListItem>  
    </List>  
  </FlowDocument>  
</RichTextBox>
```



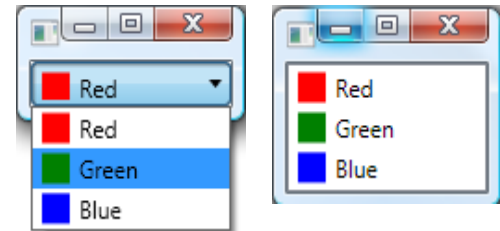
Steuerelemente mit mehreren Kindelementen (ItemsControl)



Listen mit Auswahlmöglichkeit (*Selector*)

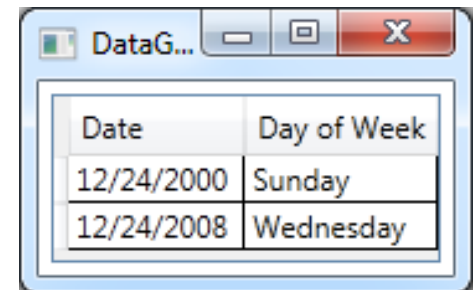
- *ComboBox/ListBox*: Liste mit beliebig vielen Kindelementen

```
<ComboBox>
  <StackPanel Margin="2" Orientation="Horizontal">
    <Rectangle Height="15" Width="15" Fill="Red" />
    <TextBlock Padding="5,0,2,0">Red</TextBlock>
  </StackPanel>
  <StackPanel>...</StackPanel>
  <StackPanel>...</StackPanel>
</ComboBox>
```



- *DataGrid*: Darstellung von Daten in tabellarischer Form

```
<DataGrid
  <DataGrid.Columns>
    <DataGridTextColumn Header="Date"
      Binding="{Binding Date}" />
    <DataGridTextColumn Header="Day of Week"
      Binding="{Binding DayOfWeek}" />
  </DataGrid.Columns>
  <sys:DateTime>2000-12-24</sys:DateTime>
  <sys:DateTime>2008-12-24</sys:DateTime>
</DataGrid>
```



Menüs

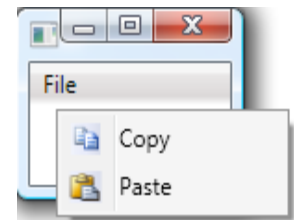
■ *Menüs in Menüleiste*

```
<Menu DockPanel.Dock="Top">
  <MenuItem Header="_File">
    <MenuItem>
      <MenuItem.Icon><Image Source="open.png" /></MenuItem.Icon>
      <MenuItem.Header>
        <StackPanel ...> ... <TextBox MinWidth="50"/></StackPanel>
      </MenuItem.Header>
    </MenuItem>
    <MenuItem Header="_Save"> ... </MenuItem>
  </MenuItem>
</Menu>
```



■ *Kontextmenü*

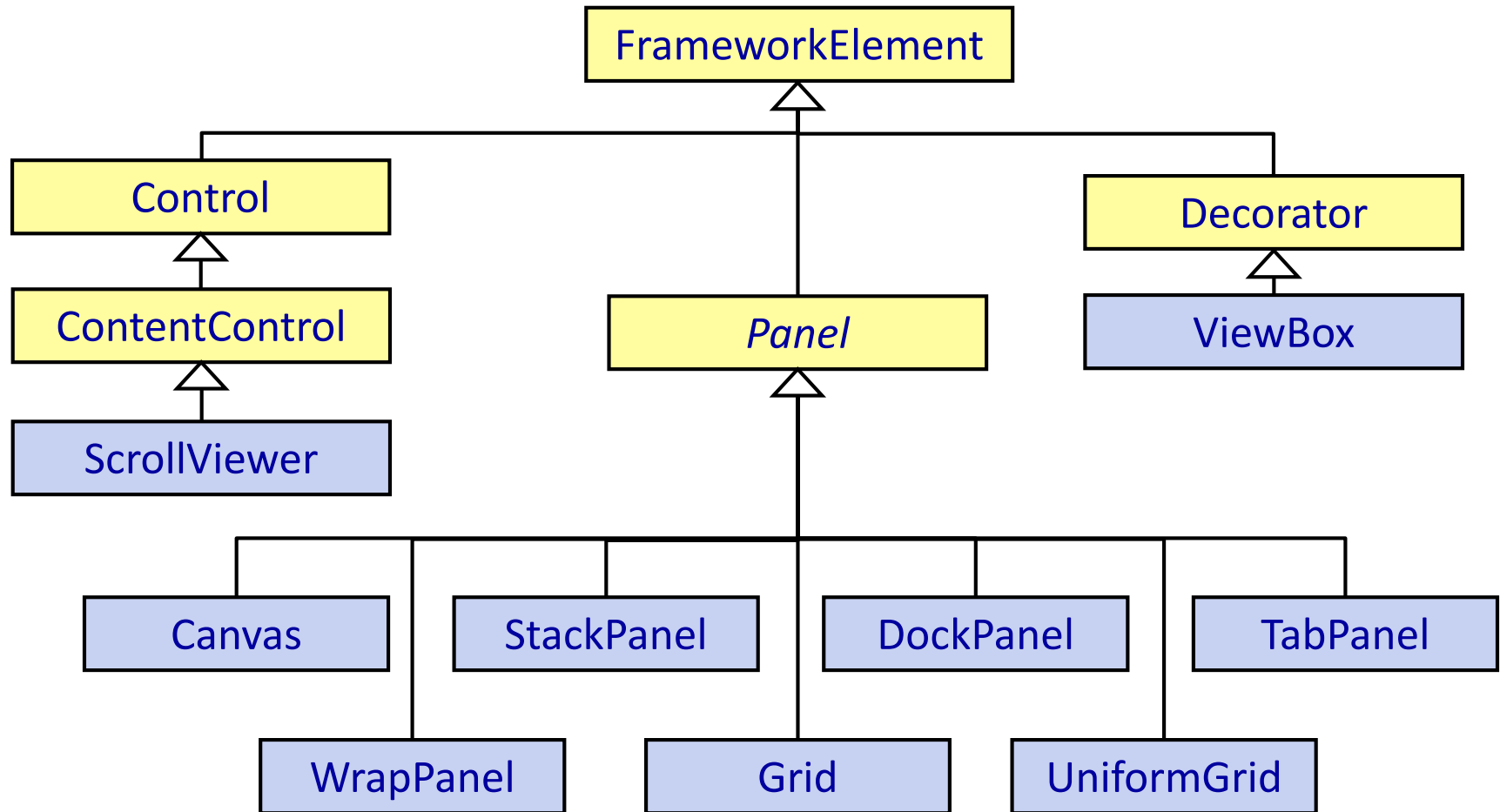
```
<Canvas Background="Transparent" Width="100" Height="40">
  <Canvas.ContextMenu>
    <ContextMenu>
      <MenuItem Header="_Copy">
        <MenuItem.Icon><Image Source="copy.png" /></MenuItem.Icon>
      </MenuItem>
      <MenuItem Header="_Paste"> ... </MenuItem>
    </ContextMenu>
  </Canvas.ContextMenu>
</Canvas>
```





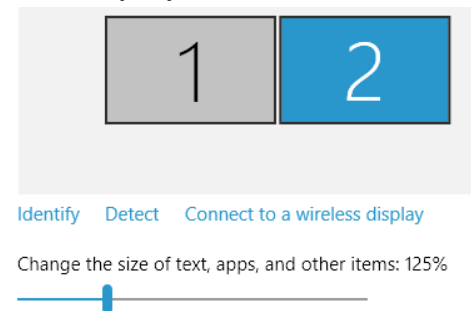
Layout-Klassen

Die Layout-Klassen der WPF



Layout-Klassen (Panels)

- Layout-Klassen ermitteln den Platzbedarf der Kindelemente (*measure*) und ordnen die Elemente im Behälter an (*arrange*).
- Jedem Kindelement wird eine Layout-Zelle zugeordnet. Die Positionierung innerhalb der Zelle wird durch *Layout-Properties* definiert, die den Kindelementen zugeordnet werden.
- Größenangaben erfolgen in *geräteunabhängigen Einheiten (logischen Einheiten)*.
 - 1 logische Einheit = 1/96 Zoll
 - Beispiel: `Width="96"` entspricht `Width="1in"` bzw. `Width="2.54cm"`.
 - Das Verhältnis logische Einheit zu physischer Einheit hängt von der physischen und der in Windows definierten Pixeldichte (*DPI Scaling*) ab.
 - Physische Pixeldichte = Windows-Pixeldichte = 96 DPI
→ 96 logische Einheiten = 96 physische Einheiten = 1 Zoll
 - Physische Pixeldichte = Windows-Pixeldichte = 192 DPI
→ 96 logische Einheiten = 192 physische Einheiten = 1 Zoll
 - Fonts werden ebenfalls in logischen Einheiten angegeben:
`FontSize="11"` entspricht `FontSize="8.25pt"` ($8.25\text{pt} = 8.25/72\text{in} = 11/96\text{in}$)



Layout-Properties (1)

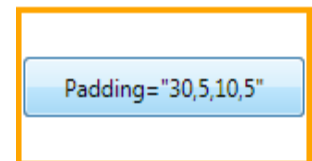
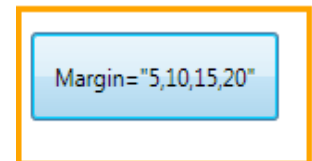
- **Width/Height:** Fixe Breite/Höhe in logischen Einheiten.
 - Mit dem Standardwert *Auto* überlässt man der Layout-Klasse die Festlegung der Größe des Elements (*Width="Auto"*).
 - Tatsächliche Größe kann mit *ActualWidth/ActualHeight* ermittelt werden.
- **MinWidth/MinHeight/MaxWidth/MaxHeight:** Minimale/Maximale Breite/Höhe.
- **HorizontalAlignment= ["Left"|"Right"|"Center"|"Stretch"]:** Horizontale Ausrichtung, falls Layout-Zelle breiter ist als das Kindelement.



- **VerticalAlignment= ["Top"|"Bottom"|"Center"|"Stretch"]:** Horizontale Ausrichtung, falls Layout-Zelle höher ist als das Kindelement.

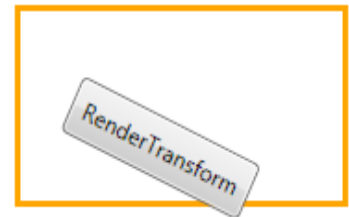


- **Margin="Left,Top,Right,Bottom":** Abstand des Kindelements zu den Rändern der Layout-Zelle.
- **Padding="Left,Top,Right,Bottom":** Abstand des Inhalts zum Rand des Kindelements.



Layout-Properties (2)

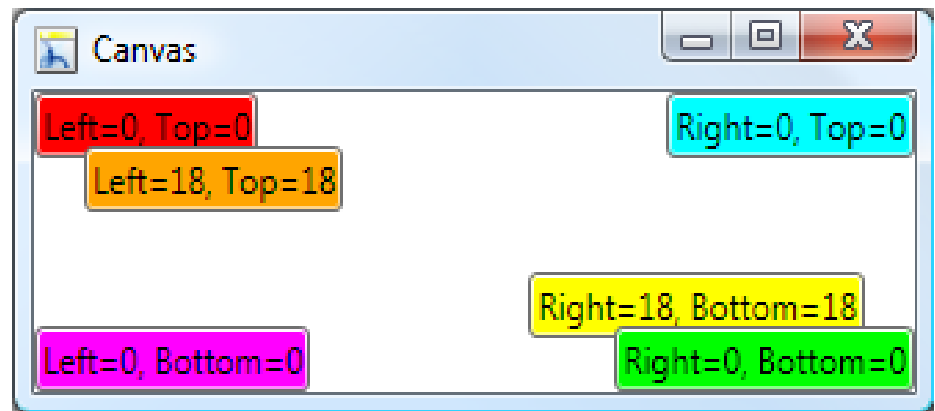
- *LayoutTransform*: Transformationsmatrix, die auf Kindelement angewandt wird (bewirkt Verschiebung, Skalierung, Rotation bzw. Verzerrung). Umschließendes Rechteck wird neu berechnet.
- *Rendertransform*: Wie *LayoutTransform*, die Größe des umschließenden Rechtecks wird aber nicht neu ermittelt.



Layout: *Canvas*

```
<Canvas Name="layoutRoot">
  <Button Background="Red" >...</Button>
  <Button Canvas.Left="18" Canvas.Top="18" Background="Orange" >...</Button>
  <Button Canvas.Right="18" Canvas.Bottom="18" Background="Yellow" >...</Button>
  <Button Canvas.Right="0" Canvas.Bottom="0" Background="Lime" >...</Button>
  <Button Canvas.Right="0" Canvas.Top="0" Background="Aqua" >...</Button>
  <Button Canvas.Left="0" Canvas.Bottom="0" Background="Magenta">...</Button>
</Canvas>
```

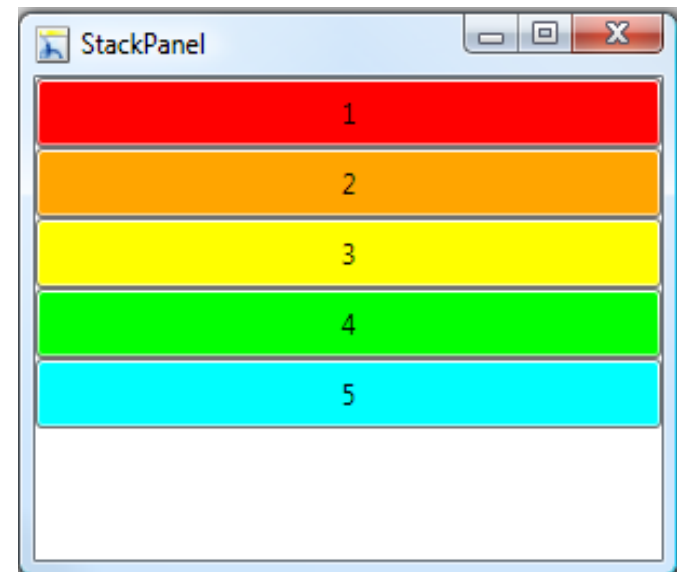
- Elemente können an maximal zwei angrenzenden Kanten angehängt werden.
- Wichtige Layout-Properties:
 - *Margin*: Nur für Seiten relevant, an die das Element angehängt wurde.



Layout: StackPanel

```
<StackPanel x:Name="layoutRoot" Orientation="Vertical">  
  <Button.Background="Red"...>1</Button>  
  <Button Background="Orange">2</Button>  
  <Button Background="Yellow">3</Button>  
  <Button Background="Lime" >4</Button>  
  <Button Background="Aqua" >5</Button>  
</StackPanel>
```

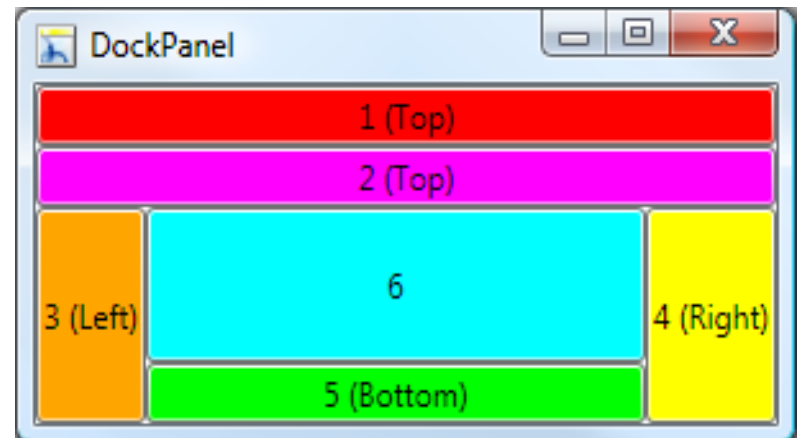
- Elemente werden übereinander (vertikal) oder nebeneinander (horizontal) angeordnet.
- Wichtige Layout-Properties:
 - *Margin*
 - *HorizontalAlignment* (falls Orientation="Vertical")
 - *VerticalAlignment* (falls Orientation="Horizontal")



Layout: DockPanel

```
<DockPanel>
  <Button DockPanel.Dock="Top"      Background="Red"    >1 (Top)</Button>
  <Button DockPanel.Dock="Top"      Background="Red"    >2 (Top)</Button>
  <Button DockPanel.Dock="Left"     Background="Orange" >3 (Left)</Button>
  <Button DockPanel.Dock="Right"    Background="Yellow" >4 (Right)</Button>
  <Button DockPanel.Dock="Bottom"   Background="Lime"   >5 (Bottom)</Button>
  <Button Background="Aqua" >6</Button>
</DockPanel>
```

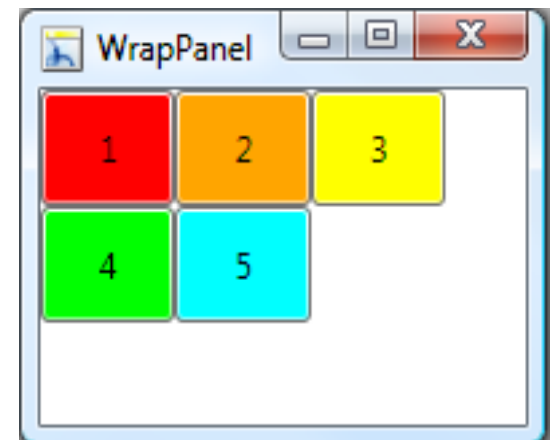
- Elemente werden entlang der Ränder des Behälters angeordnet.
- Wichtige Layout-Properties:
 - *Margin*
 - *HorizontalAlignment*: für oben, unten und im Zentrum angeordnete Elemente.
 - *VerticalAlignment*: für links, rechts und im Zentrum angeordnete Elemente.



Layout: WrapPanel

```
<WrapPanel Name="layoutRoot" Orientation="Horizontal">  
  <Button Background="Red"      >1</Button>  
  <Button Background="Orange"   >2</Button>  
  <Button Background="Yellow"   >3</Button>  
  <Button Background="Lime"     >4</Button>  
  <Button Background="Aqua"     >5</Button>  
</WrapPanel>
```

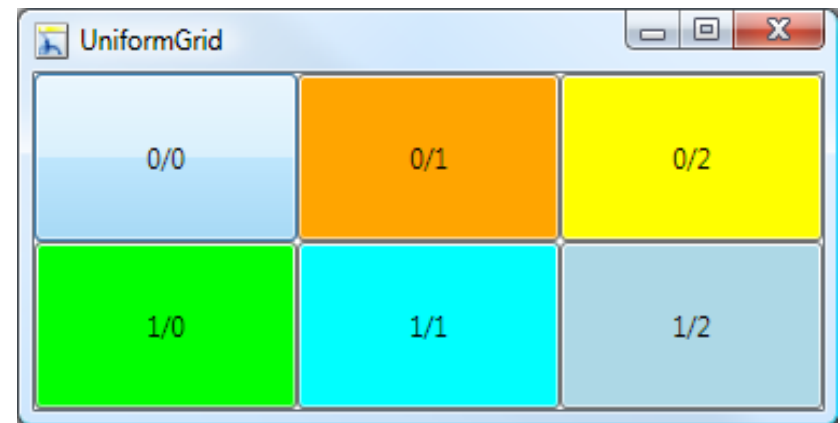
- Elemente werden von zeilenweise von rechts nach links oder spaltenweise von oben nach unten angeordnet.
- Wichtige Layout-Properties:
 - *Margin*
 - *HorizontalAlignment* (falls *Orientation*="Vertical"): Ausrichtung der Elemente innerhalb einer Zeile.
 - *VerticalAlignment* (falls *Orientation*="Horizontal"): Ausrichtung der Elemente innerhalb einer Spalte.



Layout: UniformGrid

```
<UniformGrid x:Name=layoutRoot" Rows="2" Columns="3">
  <Button Grid.Row="0" Grid.Column="0" Background="Red">0/0</Button>
  <Button Grid.Row="0" Grid.Column="1" Background="Orange">0/1</Button>
  <Button Grid.Row="0" Grid.Column="2" Background="Yellow">0/2</Button>
  <Button Grid.Row="1" Grid.Column="0" Background="Lime">1/0</Button>
  <Button Grid.Row="1" Grid.Column="1" Background="Aqua">1/1</Button>
  <Button Grid.Row="1" Grid.Column="2"
    Background="LightBlue">1/2</Button>
</UniformGrid>
```

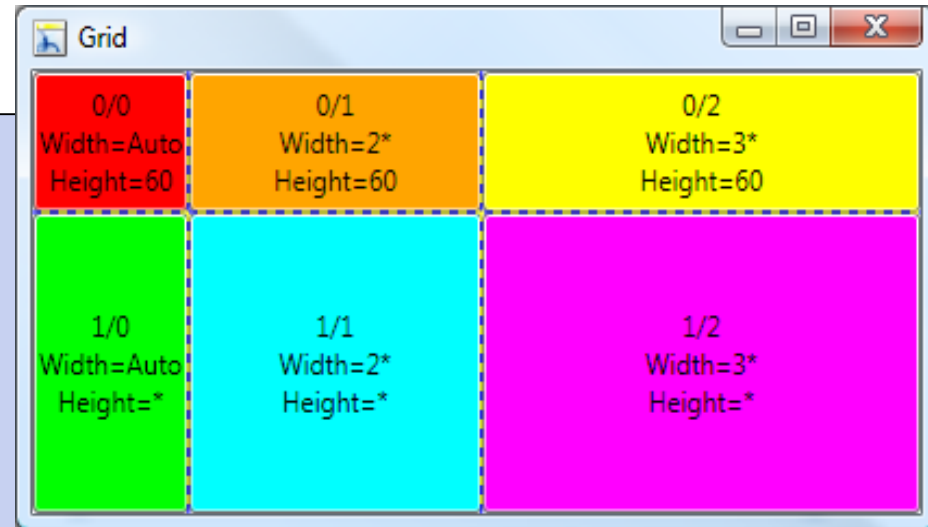
- Jedes Element bekommt eine gleich große Zelle zur Verfügung gestellt.
- Wichtige Layout-Properties:
 - *Margin*
 - *HorizontalAlignment*
 - *VerticalAlignment*



Layout: Grid (1)

```
<Grid x:Name= "layoutRoot">
  <Grid.RowDefinitions>
    <RowDefinition Height="60"/>
    <RowDefinition Height="*" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="Auto"/>
    <ColumnDefinition Width="2*" />
    <ColumnDefinition Width="3*" />
  </Grid.ColumnDefinitions>

  <Button Grid.Row="0" Grid.Column="0" Background="Red">...</Button>
  <Button Grid.Row="0" Grid.Column="1" Background="Orange">...</Button>
  <Button Grid.Row="0" Grid.Column="2" Background="Yellow">...</Button>
  <Button Grid.Row="1" Grid.Column="0" Background="Lime">...</Button>
  <Button Grid.Row="1" Grid.Column="1" Background="Aqua">...</Button>
  <Button Grid.Row="1" Grid.Column="2" Background="Magenta">...</Button>
</Grid>
```

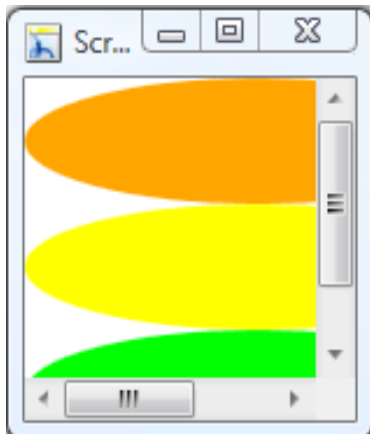


Layout: Grid (2)

- Der verfügbare Platz wird in Zeilen und Spalten zerlegt.
- Die Art der Zeile/Spalte bestimmt seine Höhe/Breite:
 - Zeile/Spalte hat eine fixe Höhe/Breite: `<RowDefinition Height="60">`
 - Zeile/Spalte bekommt so viel Platz, wie für das größte Kindelement zumindest erforderlich ist: `<RowDefinition Height="Auto">`
 - Der restliche Platz wird unter den verbleibenden Zeilen/Spalten aufgrund Ihrer *Gewichtung* verteilt: `<RowDefinition Height="2*">`
 - Der Standardwert ist "*".
- Mit dem Attribut *RowSpan/ColumnSpan* können Kindelemente auf mehrere Spalten/Zeilen verteilt werden.
- Layout-Properties:
 - *Margin*
 - *HorizontalAlignment*
 - *VerticalAlignment*

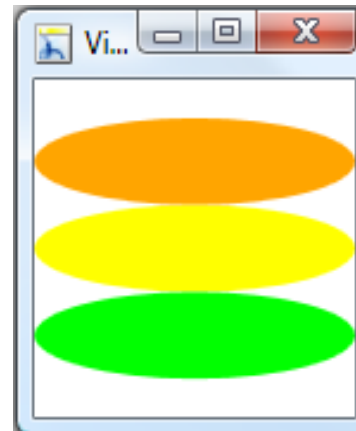
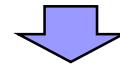
ScrollViewer und Viewbox

```
<ScrollViewer
  HorizontalScrollBarVisibility="Auto"
  VerticalScrollBarVisibility="Auto">
  <StackPanel>
    <Ellipse Width="200" Height="50"
      Fill="Orange" />
    <Ellipse Width="200" Height="50"
      Fill="Yellow" />
    ...
  </StackPanel>
</ScrollViewer>
```



Scrollbar, wenn
nicht genug Platz
für Kindelement
vorhanden ist.

```
<Viewbox StretchDirection="Both">
  <StackPanel>
    <Ellipse Width="200" Height="50"
      Fill="Orange" />
    <Ellipse Width="200" Height="50"
      Fill="Yellow" />
    <Ellipse Width="200" Height="50"
      Fill="Lime" />
  </StackPanel>
</Viewbox>
```



Kindelement wird
so weit verkleinert,
dass es im Fenster
Platz findet.



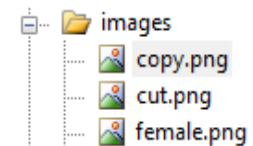
Ressourcen

Ressourcen

- WPF unterstützt
 - **Binäre Ressourcen** (auf Basis von .NET 2.0) und
 - **Logische Ressourcen** (neu in der WPF).
- Binäre Ressourcen
 - Können in Assembly eingebettet werden (*Resource*) oder
 - lose (*Content*) mit der Anwendung verbunden sein.
- Zugriff auf binäre Ressourcen:

```
<Image Source="images/copy.png" />
```

```
ImageSource src = new BitmapImage(  
    new Uri("images/copy.png", UriKind.Relative));  
Image img = new Image() { Source = src };
```



Advanced	
Build Action	Resource
Copy to Output D	Do not copy
Custom Tool	
Custom Tool Nam	
Misc	
File Name	copy.png
Full Path	U:\Lehre\2008\

Logische Ressourcen

- Häufig benötigte Objekte können an einer *zentralen Stelle definiert* und *mehrmals* in der *WPF-Anwendung verwendet* werden.
- Eine Ressource kann ein beliebiges .NET-Objekt sein, dem ein Schlüssel (*x:Key*) zugeordnet wird.
- **Definition von Ressourcen in XAML:**

```
<Window.Resources>  
  <SolidColorBrush x:Key="backBrush" Color="Orange" />  
</Window.Resources>
```

- **Definition von Ressourcen im Code:**

```
window.Resources.Add("backBrush", Brushes.Orange);
```

- Ressourcen können jedem Objekt, das mindestens vom Typ *FrameworkElement* ist, zugeordnet werden.

Zugriff auf logische Ressourcen

■ *StaticResource*

- Ressource wird beim **Laden des Fensters erzeugt** und nur **einmal** auf das Ziel **angewandt**.
- Ressource muss in XAML definiert werden, bevor sie verwendet wird.

```
<Button Background="{StaticResource backBrush}" ... />
```

```
button.Background = (Brush)button.FindResource("backBrush");
```

■ *DynamicResource*

- Ressource wird geladen, wenn **sie benötigt wird** (wenn das Objekt, das die Ressource referenziert, erzeugt wird).
- Element wird **aktualisiert, wenn sich Ressource ändert**.

```
<Button Background="{DynamicResource backBrush}" ... />
```

```
button.SetResourceReference(Window.BackgroundProperty, "backBrush");
```

Gültigkeitsbereiche von Ressourcen

- Ressourcen können für jedes Element im Elementbaum definiert werden.
- Anwendungsglobale Ressourcen werden dem Applikationsobjekt zugeordnet.
- Ressourcen werden vererbt, d. h. ein Element kann auch auf die Ressourcen der Elternelemente zugreifen.

```
<Application ...>
  <Application.Resources>
    <SolidColorBrush x:Key="brush1"
                      Color="Orange" />
  </Application.Resources>
</Application>
```

```
<Window...>
  <Window.Resources>
    <SolidColorBrush x:Key="brush2"
                      Color="Green" />
  </Window.Resources>
  <StackPanel>
    <StackPanel.Resources>
      <SolidColorBrush x:Key="brush2"
                        Color="Blue" />
    </StackPanel.Resources>
    <Button Background=
              "{StaticResource brush2}" />
    <Button Background=
              "{StaticResource brush1}" />
  </StackPanel>
</Window>
```

Resource Dictionaries

- Ressourcen können in eigene XAML-Dateien, so genannte *Resource Dictionaries*, ausgelagert werden.
- Auf diese Weise können Ressourcen zentral verwaltet und in mehreren Projekten verwendet werden.

```
<Application.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="ResDict1.xaml" />
      <ResourceDictionary Source="ResDict2.xaml" />
    </ResourceDictionary.MergedDictionaries>
  </ResourceDictionary>
</Application.Resources>
```

ResDict1.xaml

```
<ResourceDictionary xmlns="..."
  xmlns:x="...">
  <SolidColorBrush x:Key="brush1"
    Color="LightPink" />
</ResourceDictionary>
```

ResDict2.xaml

```
<ResourceDictionary xmlns="..."
  xmlns:x="...">
  <SolidColorBrush x:Key="brush2"
    Color="LightGreen" />
</ResourceDictionary>
```

Routed Events und Kommands

Behandlung von Ereignissen

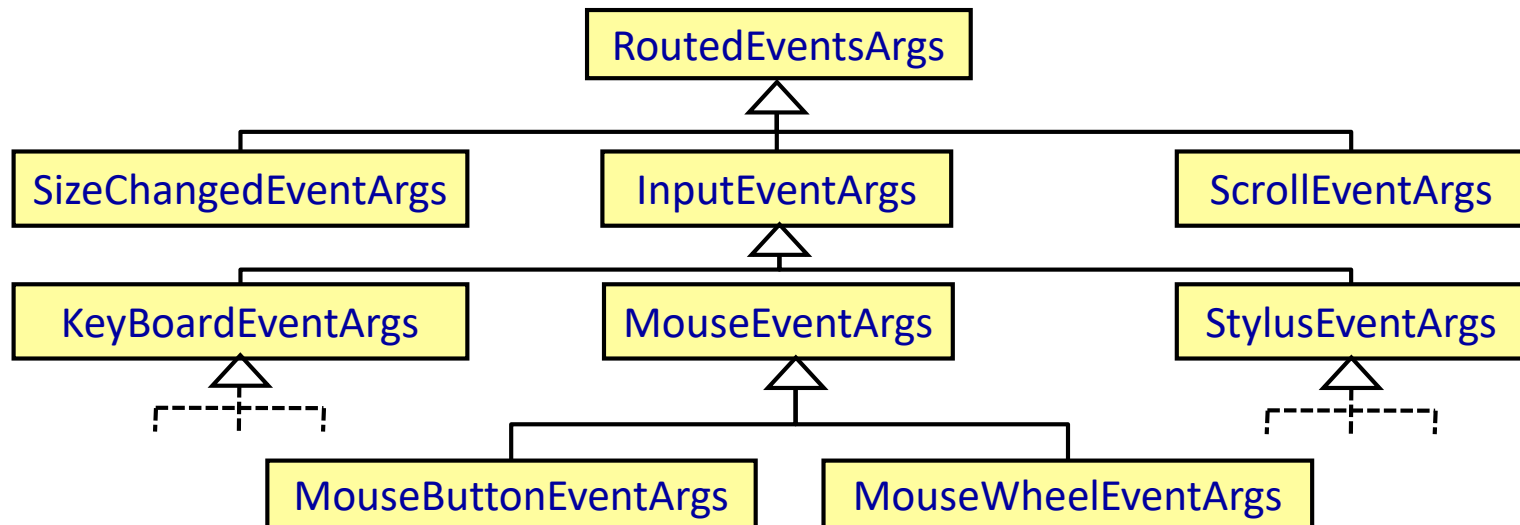
- Die Verbindung von Ereignissen mit Ereignisbehandlungsmethoden kann in XAML oder im Code erfolgen (auf Basis von .NET-Delegates).

```
<Button Name="button1" Click="ButtonClick" Content="Button 1" />
```

```
button1.Click += new RoutedEventHandler(ButtonClick);
```

```
private void ButtonClick(object sender, RoutedEventArgs e) { ... }
```

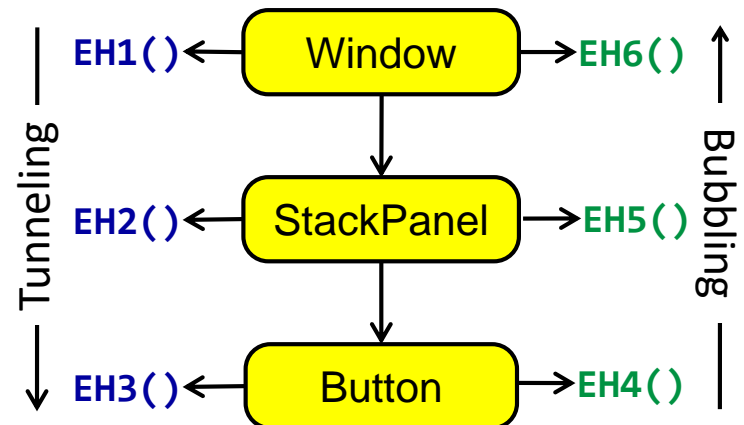
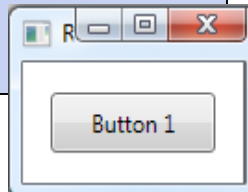
- Die Ereignisparameter werden in *RoutedEventArgs* verpackt:



Routed Events (1)

- *Routed Events* werden auch an die Eltern- bzw. Kindelemente des Ereignisauslösers weitergereicht.
- Jedem Ereignis ist eine *Routingstrategie* zugeordnet:
 - *Bubbling*: Beginnend mit dem Auslöser wird das Ereignis auch in allen Vorgängern gefeuert.
 - *Tunneling*: Feuern des Ereignisses in allen Elementen vom Wurzelement bis zum Auslöser.
 - *Direct*: Ereignis wird nur im auslösenden Element gefeuert.

```
<Window PreviewMouseDown="EH1"  
    MouseDown="EH6">  
  <StackPanel PreviewMouseDown="EH2"  
    MouseDown="EH5">  
    <Button Name="button1"  
      PreviewMouseDown="EH3"  
      MouseDown="EH4" />  
  </StackPanel>  
</Window>
```



Routed Events (2)

- Unterbrechen des Routings:

```
private void EventHandler(object sender, RoutedEventArgs e) {  
    ...  
    e.Handled = true; // Ab jetzt werden keine weiteren Handler aufgerufen.  
}
```

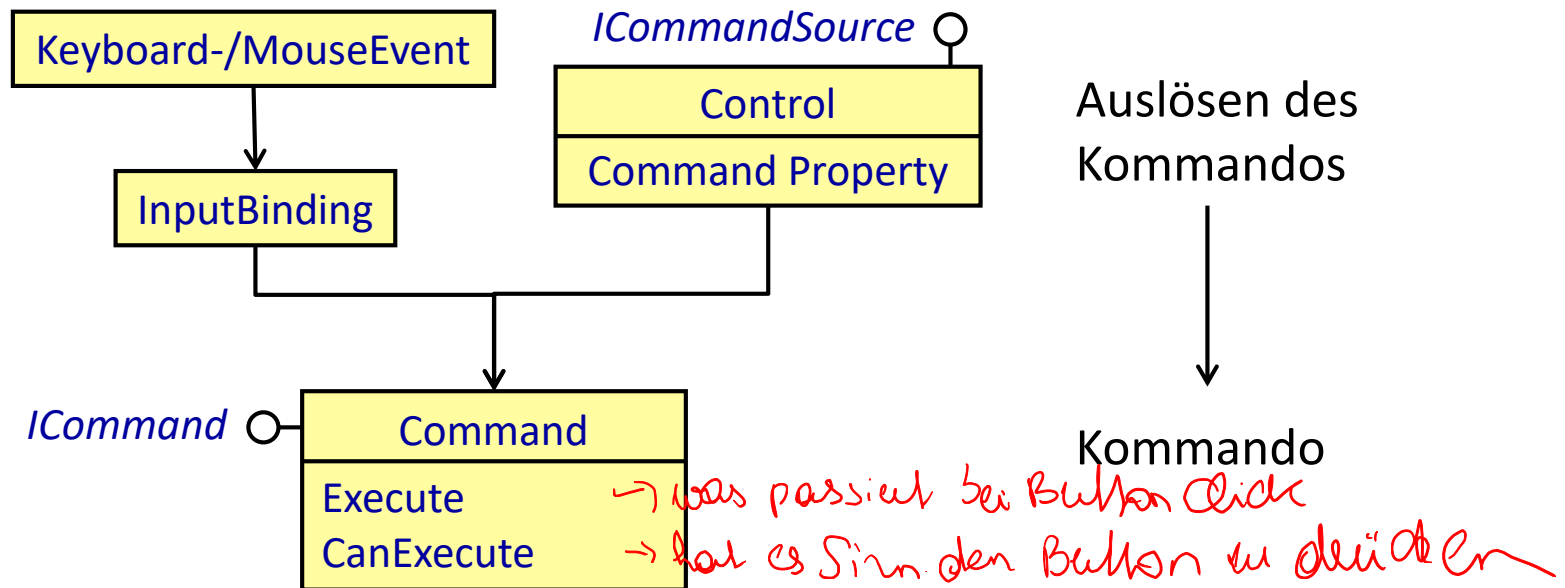
- Sender und Auslöser (*Source*) eines Ereignisses.

```
private void EventHandler(object sender, RoutedEventArgs e) {  
    object source = e.Source;  
    ...  
}
```

- *sender*: Objekt, das sich bei der Ereignisquelle registriert hat.
- *e.Source*: Objekt, welches das Ereignis ausgelöst hat.
- *e.OriginalSource*: Objekt im *Visual Tree*, welches das Ereignis ausgelöst hat.

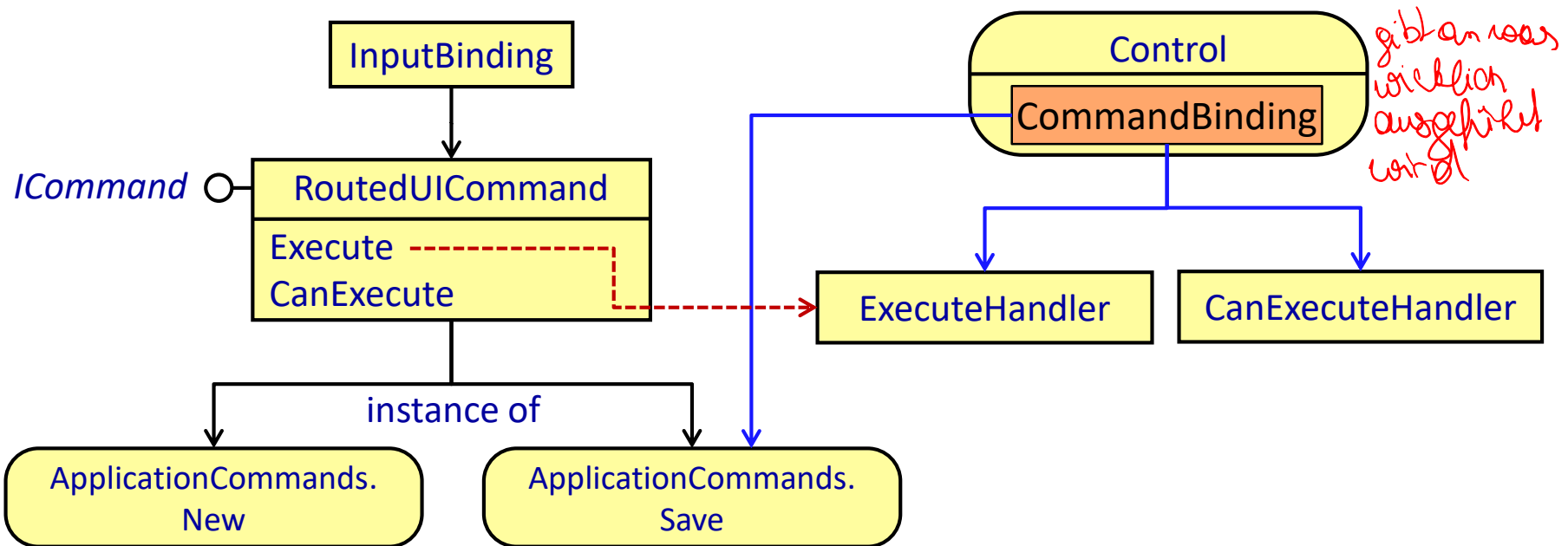
Kommandos

- Abstraktion von benutzergesteuerten Aktionen.
- Auslösen und Ausführen der Aktion wird voneinander getrennt.



Bindung von Kommandos

- Es gibt vordefinierte Kommandos vom Typ *RoutedUICommand* (*ApplicationCommands*.{*Save*, *New*, *Close*, *Copy*, ...})
- *RoutedUICommand* sucht im Steuerelementbaum nach *CommandBindings*.
 - Die damit verbundenen Methoden werden ausgeführt.



Kommandos – Beispiel

```
<Window.InputBindings>
  <KeyBinding Command="ApplicationCommands.Save"
              Key="S" Modifiers="Control" />
</Window.InputBindings>
<MenuItem Header="_Save" Command="ApplicationCommands.Save" />
<Button Content="Save" Command="ApplicationCommands.Save" />
<Window.CommandBindings>
  <CommandBinding Command="ApplicationCommands.Save"
                  Executed="SaveExecuteHandler"
                  CanExecute="SaveCanExecuteHandler" />
</Window.CommandBindings>
```

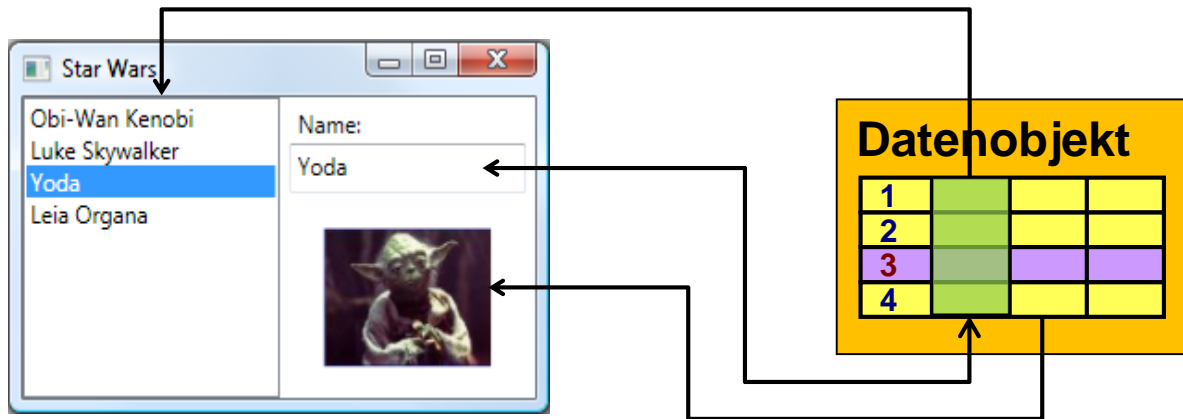
```
public void SaveExecuteHandler(object sender,
                               ExecutedRoutedEventArgs e) { ... }
public void SaveCanExecuteHandler(object sender,
                                   CanExecuteRoutedEventArgs e) {
    e.CanExecute = ...;
}
```

Datenbindung

.NET 1.6.

Problemstellung

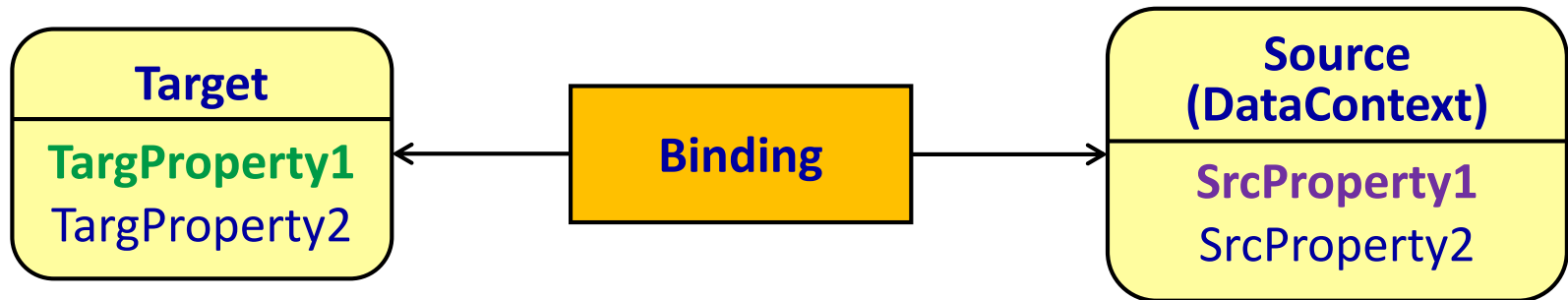
collbacks sind wahnt zauber!



- GUI-Elemente müssen häufig mit Datenobjekten synchronisiert werden.
- Ohne Framework-Unterstützung ist das relativ aufwändig:
 - GUI muss aktuelle Werte aus Datenobjekt auslesen.
 - Zustandsänderungen in der GUI müssen an Datenobjekt weitergeleitet werden: Registrierung von Ereignisbehandlungsmethoden, Konvertierung, Validierung, Speichern der neuen Werte.
 - Die Synchronisation der GUI-Elemente muss manuell erfolgen.
 - Änderungen am Datenobjekt müssen manuell an die GUI-Elemente weitergegeben werden.
- Bei vielen Frameworks muss die Definition der Beziehung zwischen GUI und Datenobjekt im Code erfolgen (nicht deklarativ).

Definition einer Bindung

- Mithilfe einer Bindung wird eine Property eines Quellobjekts (*Source*) mit einer Property eines Zielobjekts (*Target*) automatisch synchronisiert.



XAML

```
<Container DataContext="{StaticResource sourceObject}">  
  <Target TargProperty1="{Binding Path=SrcProperty1}" />  
</Container>
```

Quelle hierarchisch-weise Context

- Property *DataContext* wird vererbt → ermöglicht zentrale Definition
- Anforderungen an die beteiligten Properties:
 - Zielproperty muss eine *Dependency-Property* sein.
 - Quellproperty muss eine *Dependency-Property* sein oder die Quelle muss *INotifyPropertyChanged* implementieren, falls Wertänderungen der Quellproperty weitergeleitet werden sollen.

Die wichtigsten Bindungseigenschaften

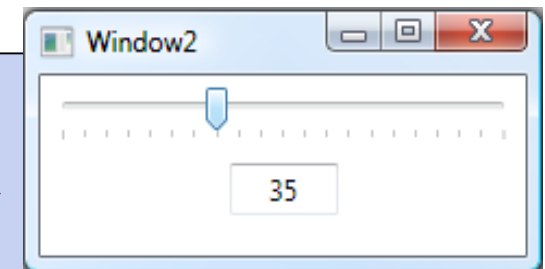
```
"{Binding Source=value, ElementName=value,  
    Path=value, Mode=value, Converter=value, ... }"
```

- **Source:** Referenz auf Datenquelle. Muss nur angegeben werden, falls *DataContext* nicht definiert wurde.
- **ElementName:** Name eines anderen GUI-Elements, das als Quelle fungiert.
- **Path:** Pfad zur Quellproperty, z.B. *Name*, *Name.Length*, *Errors[0].ErrorContent*
- **Mode:**
 - *ToWay*: Ziel ↔ Quelle
 - *OneWay*: Quelle → Ziel
 - *Default*: Bindungsmodus, der in den Metadaten der Dependency-Property festgelegt ist.
- **Converter:** Objekt, das zwischen Quell- und Zielproperty konvertiert.
- **UpdateSourceTrigger:** Welches Ereignis löst Aktualisierung der Zielproperty aus:
 - *LostFocus*, *PropertyChanged*, *Explicit*, *Default*.

Kopplung zweier GUI-Elemente



```
<StackPanel>
  <Slider Name="slider"
    Value="50" Minimum="0" Maximum="100" ... />
  <TextBox Name="textBox"
    Text="{Binding ElementName=slider,
      Path=Value,
      Mode=TwoWay,
      UpdateSourceTrigger=PropertyChanged}" ... />
</StackPanel>
```



Propagation von Eigenschaftsänderungen

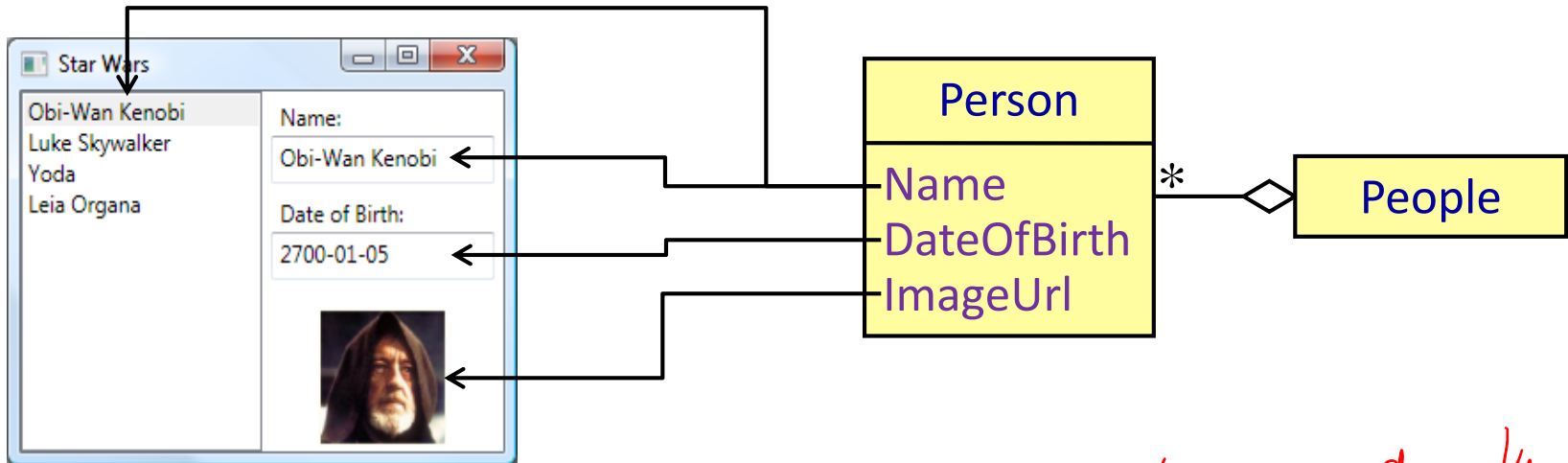
- Klassen, die *INotifyPropertyChanged* implementieren, verständigen interessierte Clients von Wertänderungen an ihren Properties.

```
public class Person : INotifyPropertyChanged {  
    private string name;  
    public event PropertyChangedEventHandler PropertyChanged;  
  
    public string Name {  
        get { return name; }  
        set {  
            oldName = name; name = value;  
            if (name != oldName)  
                PropertyChanged?.Invoke(this,  
                    new PropertyChangedEventArgs(nameOf(Name)));  
        }  
    }  
}
```

- Behälterklassen müssen zusätzlich *INotifyCollectionChanged* implementieren, damit auch das Hinzufügen und Löschen von Elementen berücksichtigt wird.
- *ObservableCollection* ist eine generische Implementierung dieses Interfaces:

```
public class People : ObservableCollection<Person> { ... }
```

Kopplung von GUI-Elementen an ein Datenobjekt



```

<Window.Resources>
  <src:People x:Key="people"/>
</Window.Resources>
<Grid DataContext="{StaticResource people}">
  <TextBox Text="{Binding Path=Name}" ... />
  <TextBox Text="{Binding Path=DateOfBirth}" ... />
  <ListBox ItemsSource="{Binding}" DisplayMemberPath="Name">
  <Image Source="{Binding Path=ImageUrl}" ... />
</Grid>

```

Handwritten notes:

- daten context wird nach unten vererbt* (data context is inherited downwards)
- Verbinden wir mit Object, also ohne ein Callback* (We connect with Object, so without a callback)
- nur der Name wird dargestellt* (only the name is displayed)

Data-Templates

- Mit Data-Templates kann man definieren, wie Einträge in Listen-Steuererelementen formatiert werden sollen.
- Bindungs-Ausdrücke stellen Platzhalter für Datenelemente dar.

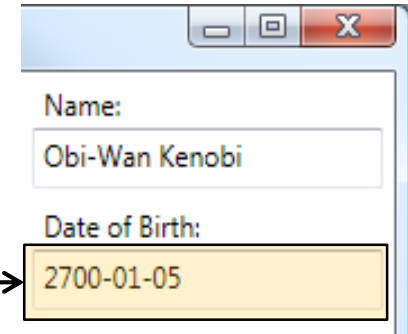
Star Wars	
Obi-Wan Kenobi	2700-01-05
Luke Skywalker	2730-12-30
Yoda	2000-03-15
Leia Organa	2730-12-30

```
<ListBox ItemsSource="{Binding}" ... >
  <ListBox.ItemTemplate> → wie wird ein Item dargestellt
    <DataTemplate>
      <StackPanel Orientation="Horizontal">
        <TextBlock Text="{Binding Path=Name}"
                   FontWeight="Bold" Width="100" />
        <TextBlock Text="{Binding Path=DateOfBirth, ...}" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Typkonvertierung

ValueConverter führen Typumwandlung zwischen Quell- und Ziel-Property durch.

```
<Window.Resources>
  <src:DateTimeFormatter x:Key="formatter"/>
</Window.Resources>
<Grid>
  <TextBox Text="{Binding Path=DateOfBirth,
    Converter={StaticResource formatter},
    ConverterParameter=yyyy-MM-dd}" />
</Grid>
```



Name:
Obi-Wan Kenobi

Date of Birth:
2700-01-05

```
public class DateTimeFormatter : IValueConverter {
    public object Convert(object value, Type targetType, object param, ...) {
        DateTime date = (DateTime)value;
        return date.ToString((string)param);
    }
    public object ConvertBack(object value, Type targetType, object param, ...) {
        DateTime dt;
        if (DateTime.TryParse(value.ToString(), out dt))
            return dt;
        return value;
    }
}
```

Validierung (1)

- Bei der Konvertierung zwischen der Quell- und Ziel-Property können Fehler auftreten, die behandelt werden müssen.
- Validierungsregeln sind für die Prüfung der Eingaben verantwortlich (eingebaute Regel: *ExceptionValidationRule*).
- Validierungsfehler können im Code behandelt werden.
- Steuerelemente können bei Validierungsfehlern mit einem eigenen Template gerendert werden (*Validation.ErrorTemplate*).

```
<TextBox Validation.Error="HandleDateTimeValidationError" ...>
  <TextBox.Text >
    <Binding Path="DateOfBirth" UpdateSourceTrigger="PropertyChanged"
      Converter="{StaticResource formatter}"
      ConverterParameter="yyy-MM-dd">
      <Binding.ValidationRules>
        <ExceptionValidationRule />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</TextBox>
```

Date of Birth:

2700-01-xx

Validierung (2)

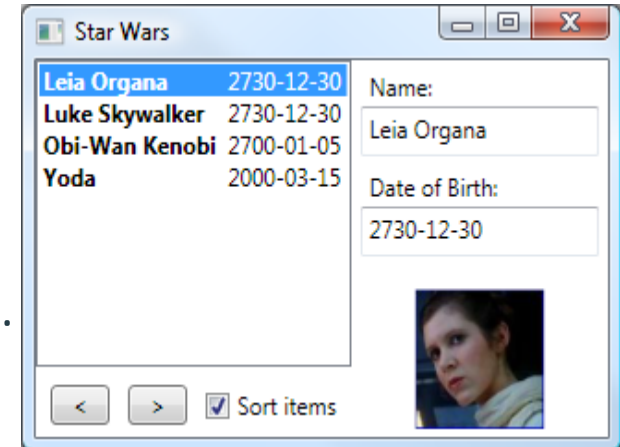
- Definition von benutzerdefinierten Validierungsregeln:

```
<TextBox Validation.Error="HandleDateTimeValidationError" ...>
  <TextBox.Text>
    <Binding Path="DateOfBirth" ...>
      <Binding.ValidationRules>
        <src:DateValidationRule />
      </Binding.ValidationRules>
    </Binding>
  </TextBox.Text>
</Text>
```

```
public class DateValidationRule : ValidationRule {
    public override ValidationResult Validate(object value, ...) {
        DateTime dt;
        if (! DateTime.TryParse((string)value, out dt))
            return new ValidationResult(false, "Invalid date format.");
        else
            return new ValidationResult(true, null);
    }
}
```

Sichten (Views)

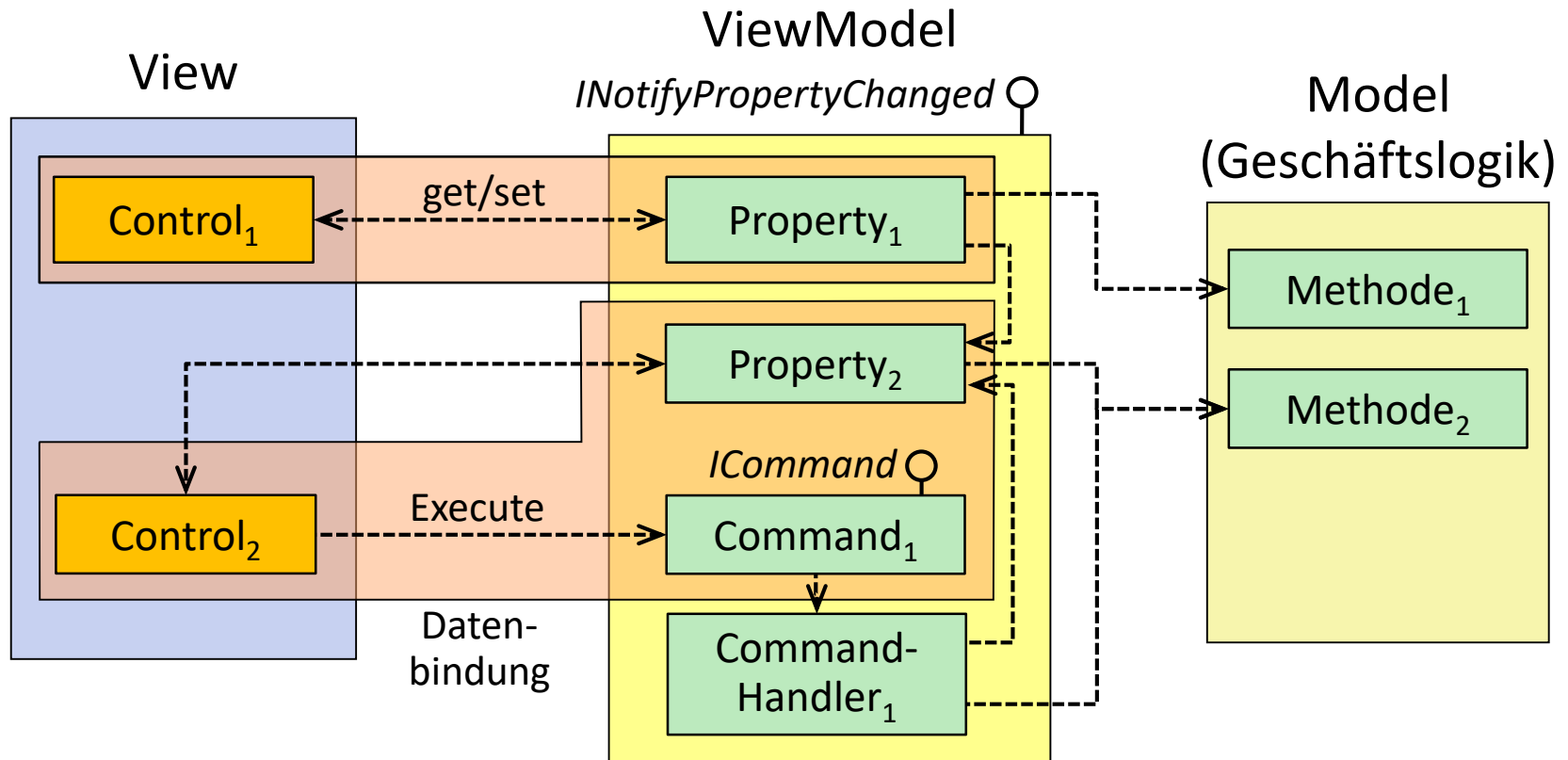
- Mit jedem Datenbehälter können mehrere Sichten verbunden werden.
- Sichten sind vollkommen entkoppelt von den Daten.
- In der Sicht sind Navigations-, Sortier-, Gruppierungs- und Filterparameter gespeichert.



```
private void chkSortItems_Click(object sender, RoutedEventArgs e) {  
    ICollectionView view =  
        CollectionViewSource.DefaultView(GetPeopleCollection());  
    view.SortDescriptions.Clear();  
    if (chkSortItems.IsChecked == true)  
        view.SortDescriptions.Add(new SortDescription("Name",  
                                                        ListSortDirection.Ascending));  
}
```

```
private void btnFwd_Click(object sender, RoutedEventArgs e) {  
    ICollectionView view =  
        CollectionViewSource.DefaultView(GetPeopleCollection());  
    view.MoveCurrentToNext();  
    if (view.IsCurrentAfterLast) view.MoveCurrentToLast();  
}
```

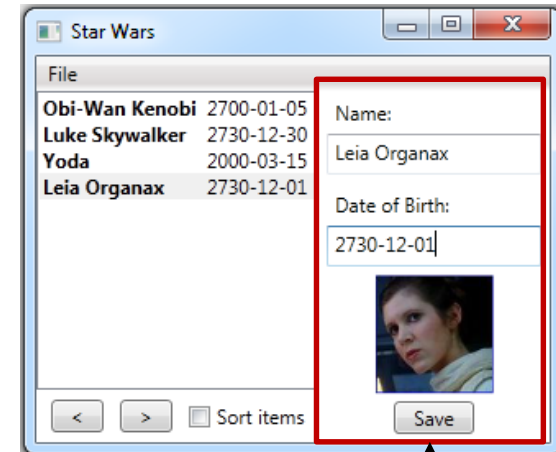
Model-View-ViewModel (MVVM)



- ViewModel bereitet Daten für die View auf.
- View und ViewModel sind nur über Datenbindung aneinander gekoppelt.
- ViewModel gibt Werteänderung durch *PropertyChange*-Events weiter.

MVVM – Beispiel

```
<Grid DataContext="{Binding Path=CurrentPerson}">
  <TextBox Text="{Binding Path=Name}" ... />
  <TextBox Text="{Binding Path=DateOfBirth}" ... />
  <Button Command="{Binding Path=SaveCommand}" ... />
  ...
</Grid>
```

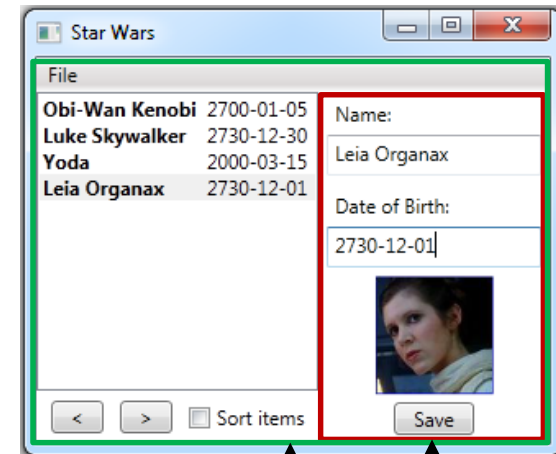


```
public class PersonVM :
    INotifyPropertyChanged {
    private ICommand saveCommand;
    private Person person;
    public PersonVM(Person p) { ... }
    public string Name {
        get { return person.Name; }
        set {
            person.Name = value;
            RaisePropertyChanged("Name");
        }
    }
}
```

```
public ICommand SaveCommand {
    get {
        if (saveCommand == null)
            saveCommand =
                new RelayCommand(
                    param => logic.Save(person));
        return saveCommand;
    }
}
```

MVVM – Navigation im View-Modell

```
<Window x:Class="Mvvm.MainWindow"
    DataContext="{DynamicResource peopleVM}" ... >
    <ListBox ItemsSource="{Binding Path=People}"
        SelectedItem="{Binding
            Path=CurrentPerson, Mode=TwoWay}" />
    <Grid DataContext="{Binding Path=CurrentPerson}">
        <TextBox Text="{Binding Path=Name}" ... />
        ...
    </Grid>
</Window>
```



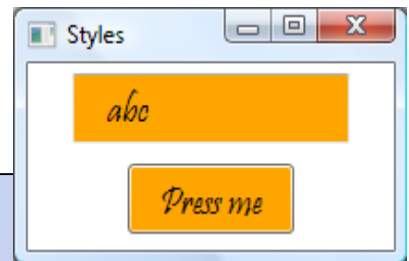
```
public class PeopleVM : INotifyPropertyChanged {
    private PersonVM currentPerson;
    private ObservableCollection<PersonVM> people;
    public PersonVM CurrentPerson { ... }
    public ObservableCollection<PersonVM> People { ... }
}
```



Styles und Templates

Styles

- *Styles* fassen eine Menge von Eigenschaften zu einer Einheit zusammen.
- Ein *Style* kann von mehreren Steuerelementen (unterschiedlichen Typs) verwendet werden.



```
<Window.Resources>
  <Style x:Key="controlStyle">
    <Setter Property="Control.Background" Value="Orange" />
    <Setter Property="Control.Padding" Value="15,5,15,0" />
    <Setter Property="Control.FontFamily" Value="Pristina" />
    <Setter Property="Control.FontSize" Value="20" />
    <Setter Property="Control.Margin" Value="5" />
  </Style>
</Window.Resources />

<StackPanel>
  <TextBox Style="{StaticResource controlStyle}" ... >abc</TextBox>
  <Button Style="{StaticResource controlStyle}" ... >Press me</Button>
</StackPanel>
```

Arten von Styles

- Styles können auf bestimmte Steuerelemente eingeschränkt werden → *typisierte Styles*:

```
<Style x:Key="controlStyle" TargetType="{x:Type Button}>  
  <Setter Property="Background" Value="Yellow" />  
  ...  
</Style>
```

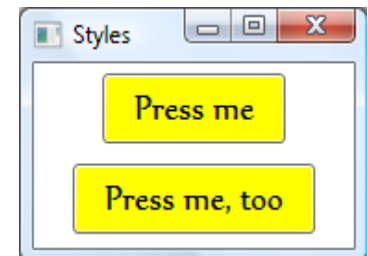
- *Benannte Styles* (x:Key="name") müssen explizit referenziert werden:

```
<Button Style="{StaticResource controlStyle}" ... >Press me</Button>
```

- *Implizite Styles* werden auf alle Instanzen eines Steuerelements angewandt:

```
<Style TargetType="{x:Type Button}">  
  <Setter Property="Background" Value="Yellow" />  
  ...  
</Style>
```

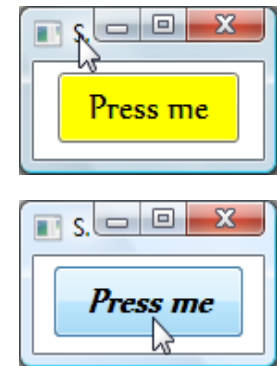
```
<StackPanel>  
  <Button ... >Press me</Button>  
  <Button ... >Press me, too</Button>  
</StackPanel>
```



Trigger (1)

- *Property-Trigger:*
 - Wertänderung einer Property löst Wertänderungen von anderen Properties aus.
 - Sobald die Bedingung nicht mehr erfüllt ist, werden Wertänderungen wieder zurückgenommen.

```
<Style TargetType="{x:Type Button}">
  <Setter Property="Background" Value="Yellow" />
  ...
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True">
      <Setter Property="FontStyle" Value="Italic" />
    </Trigger>
  </Style.Triggers>
</Style>
```



Trigger (2)

- *Ereignis-Trigger (EventTrigger)*
 - Tritt ein (Routed) Event ein, werden Aktionen (*TriggerAction*) ausgelöst.

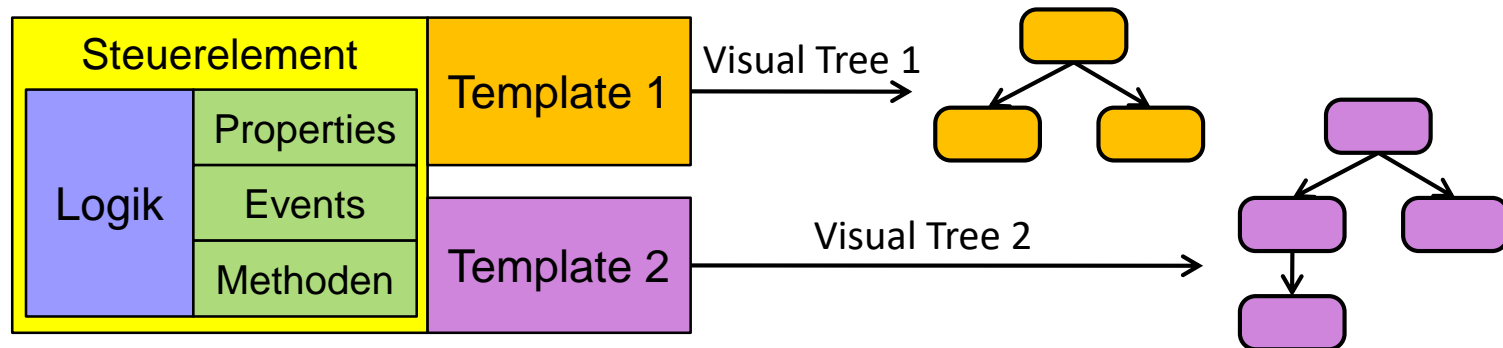
```
<Style TargetType="{x:Type Button}">
  <Style.Triggers>
    <EventTrigger RoutedEvent="Click">
      <SoundPlayerAction Source="sounds/pressed.wav" />
    </EventTrigger>
  </Style.Triggers>
</Style>
```

- *Trigger-Methoden (EventSetter)*
 - Tritt ein (Routed) Event ein, wird eine Ereignisbehandlungsmethode aufgerufen.

```
<Style TargetType="{x:Type Button}">
  <EventSetter Event="MouseEnter" Handler="ButtonEnter" />
  <EventSetter Event="MouseLeave" Handler="ButtonLeave" />
</Style>
```

Control-Templates

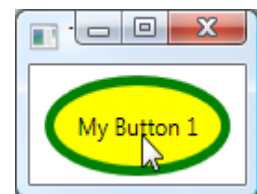
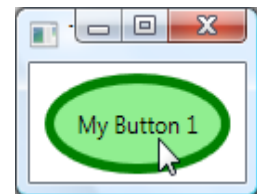
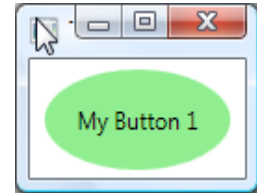
- Das Verhalten von WPF-Steuerelementen (Logik) ist unabhängig von ihrer visuellen Repräsentation.
- Mit Templates wird das **Erscheinungsbild eines Steuerelements festgelegt**, ohne auf **die Logik Einfluss zu nehmen**.



- WPF-Steuerelemente werden daher oft als **look-less** bezeichnet.
- Das Bindeglied zwischen Logik und Template stellen die Properties dar.

Beispiel: Alternatives Template für Button

```
<Window.Resources>
  <ControlTemplate x:Key="ovalBtnTempl" TargetType="{x:Type Button}">
    <Grid>
      <Ellipse Name="ellipse" Width="100" Height="50"
        StrokeThickness="5" Fill="LightGreen" />
      <ContentPresenter HorizontalAlignment="Center"
        VerticalAlignment="Center"/>
    </Grid>
    <ControlTemplate.Triggers>
      <Trigger Property="IsMouseOver" Value="true">
        <Setter TargetName="ellipse" Property="Stroke" Value="Green" />
      </Trigger>
      <Trigger Property="IsPressed" Value="true">
        <Setter TargetName="ellipse" Property="Fill" Value="Yellow" />
      </Trigger>
    </ControlTemplate.Triggers>
  </ControlTemplate>
</Window.Resources>
```



```
<Button Template="{StaticResource ovalBtnTempl}" Content="My Button 1" ... />
```

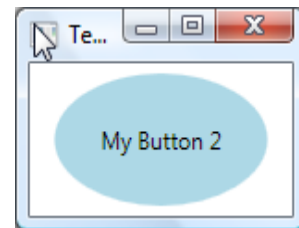
ContentPresenter ist ein Platzhalter für den Wert der Property *Content*.

TemplateBinding

Für `{TemplateBinding propertyName}` ist ein Platzhalter für den Wert der Property mit dem Namen `propertyName`.

```
<Window.Resources>
  <ControlTemplate x:Key="ovalBtnTempl" TargetType="{x:Type Button}">
    <Grid>
      <Ellipse Name="ellipse" Width="{TemplateBinding Width}"
        Height="{TemplateBinding Height}" StrokeThickness="5"
        Fill="{TemplateBinding Background}" />
      <Grid Margin="10">
        <ContentPresenter HorizontalAlignment="Center"
          VerticalAlignment="Center"
          Margin="{TemplateBinding Padding}"/>
      </Grid>
    </Grid>
  </ControlTemplate>
</Window.Resources>
```

```
<Button Template="{StaticResource ovalBtnTempl}"
  Padding="15" Background="LightBlue" Content="My Button 2" ... />
```

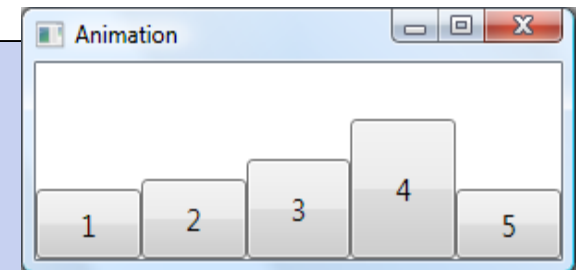


Animation und Grafik

Animation

- Viele Dependency-Properties können animiert werden: *int, double, Color, Point, Size, Vector, Thickness, Rotation, ...*
- Mit einem *Storyboard* kann festgelegt werden, wie sich ein Propertywert in der Zeit ändern soll.

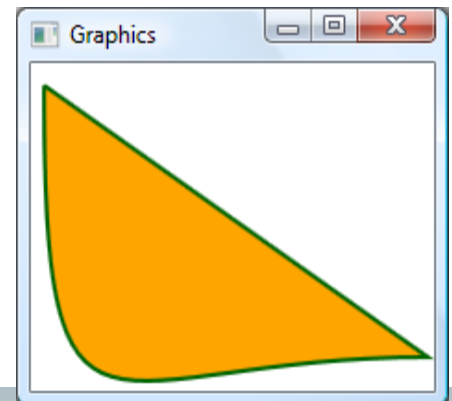
```
<Style TargetType="{x:Type Button}">
  <Setter Property="LayoutTransform">
    <Setter.Value><ScaleTransform /></Setter.Value>
  </Setter>
  <Style.Triggers>
    <EventTrigger RoutedEvent="Button.MouseEnter">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation
            Storyboard.TargetProperty="LayoutTransform.ScaleY"
            To="2" Duration="0:0:0.7" />
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.MouseLeave"> ... </EventTrigger>
  </Style.Triggers>
</Style>
```



2D-Grafik

- Grafikelemente werden wie andere Steuerelementen automatisch gerendert.
- Grafikelemente müssen nach dem Hinzufügen zum *Visual Tree* nicht neu gezeichnet werden.
- Die WPF unterstützt eine Reihe von Primitivelementen: *Rectangle*, *Ellipse*, *Line*, *Polygon*, *Polyline*, *Path*.
- Mit *Path* können durch Geraden- oder Bézier-Segmente begrenzte Figuren dargestellt werden.
- Grafikelemente können beliebig transformiert (skaliert, rotiert, ...) werden.
- Grafikelemente können mit booleschen Operationen kombiniert werden.

```
<Path Stroke="DarkGreen" StrokeThickness="3"  
      Fill="Orange"  
      Data="M 10,10  
           C 10,300 50,200 300,200 L 10,10" />
```



3D-Grafik

- WPF erlaubt die Visualisierung einfacher 3D-Modelle.
- Einfachheit ist wichtiger als Schnelligkeit.
- Für anspruchsvollere Anwendungen sollten OpenGL oder DirectX verwendet werden.
- Die Erstellung von 3D-Modellen wird in folgenden Bereichen unterstützt:
 - Beschreibung der Geometrie durch Gittermodelle mit Normalvektoren,
 - Unterstützung verschiedener Kameramodelle und Projektionen,
 - Möglichkeit der Definition von Materialeigenschaften und Texturen,
 - Unterstützung diverser Lichtmodelle,
 - Unterstützung für 3D-Transformationen.

Zusammenfassung

- Merkmale:
 - WPF unterstützt moderne Grafikhardware.
 - Einheitliche API für verschiedene Grafiksubsysteme.
 - Viele neue Konzepte zur Unterstützung der GUI-Programmierung.
 - Ermöglicht neue Form der Zusammenarbeit zwischen Designer und Entwickler.
- Mögliche Probleme:
 - Erhöhter Entwicklungsaufwand bei einfachen Anwendungen.
 - Leistungsfähige Grafikhardware ist erforderlich.
 - Mangelnde Unterstützung für Internationalisierung.