# Mobile and GUI Engineering
## WPF

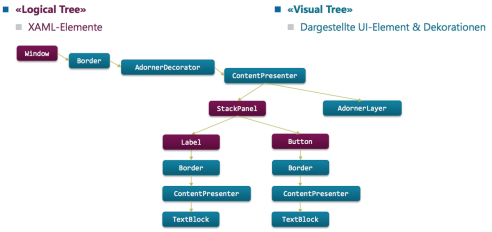
Oliviero Chiodo
Stefano Kals

Hochschule für Technik Rapperswil

Herbstsemester 2015

# Einführung

**Logical- und Visual-Tree** Der **Logical Tree** enstpricht der Struktur der XAML Elemente. Es beschreibt Beziehungen zwischen verschiedenen Elementen des UIs. Der **Visual Tree** entspricht der grafischen Repräsentation und beinhaltet alle dargestellten Elemente gemäss der Vorlage jedes Controls.




Der Logical-Tree entspricht der Strukutr der XAML-Elemente. Sie beschreibt die Beziehungen zwischen verschiedenen Elementen des UI. Es ist Zuständig für:

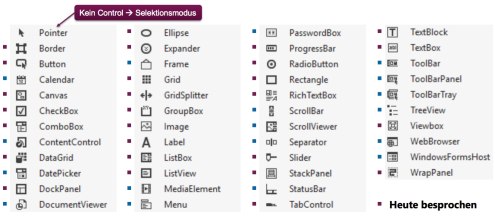
- Dependency Properties erben
- Dynamische Ressourcen-Referenzen auflösen
- Elementnamen für Datenbindung nachschlagen
- Routed Events weiterleiten

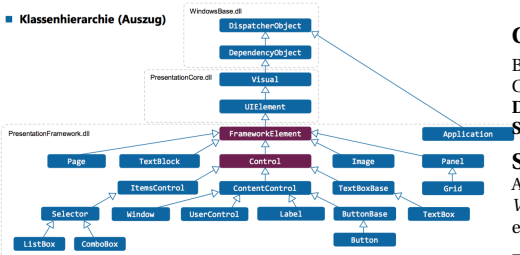
Der Visual-Tree entspricht der grafischen Repräsentation und beinhaltet alle dargestellten Elemente gemäss der Vorlage jedes Controls. Es ist Zuständig für:

- Visuelle Darstellung
- Vererben der Transparenzeinstellungen
- Vererben von Transformationen
- Vererben der IsEnabled-Property
- Hit-Testing

**Property Syntax** XAML verwendet die Property Syntax wohingegen HTML beispielsweise Attribute Syntax verwendet.

# Layout und Controls




**Grössenangaben** Zur Manipulation von Width und Height Attributen (`System.Windows.FrameworkElement`) kann man fixe Grössenangaben verwenden, diese werden als `double` Werte geschrieben. Wenn kein Kennzeichner angegeben ist (nur eine Zahl), wird automatisch *Device Independent Pixel* verwendet $\left(\frac{1}{96}''\right)$. Qualfizierte Grössenangaben sind:

- **px** Device Independent Pixels $\left(1\text{px} = \frac{1}{96}''\right)$
- **in** Inches (Zoll) $1\text{in} = 96\text{px}$
- **cm** $1\text{cm} = \frac{96}{2.54}\text{px}$
- **pt** Points $1\text{pt} = \frac{1}{72}'' = \frac{96}{72}\text{px}$

Zusätzlich kann man noch `MinWidth` und `MaxWidth` definieren. Bei Platzproblemen wird zuerst MinWidth, dann MaxWidth und dann Width evaluiert. Das Read-only Property `ActualWidth` kann verwendet werden um die Fenstergrösse während der Laufzeit abzufragen.

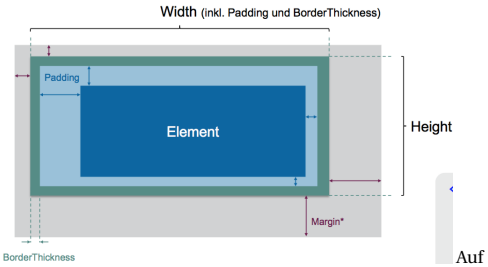
**Ausrichtung** HorizontalAlignment und VerticalAlignment Attribute manipulieren die Ausrichtung innerhalb des Containers. Es gibt auch HorizontalContentAlignment und VerticalContentAlignment für Inhaltsaurichtung beispielsweise für TextBox. Valide Werte sind: Left, Center, Right und Strecth für Horizontal bzw. Top, Center, Bottom und Stretch für Vertical. Strecth füllt jeweils die komplette Verfügbare Achse.

**Ränder & Rahmen** Um Rahmengrössen anzupassen gibt es Margin, Padding, BorderThickness und CornerRadius. Für die ersten 3 sind folgende Werte zulässig

- **l,t,r,b**
- **l,t** Left=Right, Top=Bottom
- **x** Auf allen Seiten gleich viel Abstand

Bei Corner Radius gibt es nur eine zulässige Definition:

- TopLeft, TopRight, BottomRight, BottomLeft




Auf jedes **UIElement** kann man noch `IsEnabled`, `SnapsToDevicePixels` (Rundet Pixelangaben auf physikalische Gerätepixelwerte) und `Visibility` (Collapsed, Hidden, Visible). Auf jedes **FrameworkElement** kann man `Name`, `Resource`, `Tag`, `Tooltip` und `UseLayoutRounding` anwenden. Auf jedes **Control** kann man `Background`, `BorderBrush`, `Foreground`, `FontFamily`, `FontSize`, `FontStrech`, `FontStyle` und `FontWeigt` anwenden.

**Brush** Mit Brushes kann man einfache oder komplexe Farbverläufe darstellen. Es gibt 6 Pinseltypen

- SolidColorBrush (Einfarbig)
- LinearGradientBrush (Einfacher Farbverlauf)
- RadialGradient Farbverlauf (Runder Farbverlauf, errinert an Kugel)
- ImageBrush (Bild innerhalb des Brushes)
- DrawingBrush (Spezielle Muster)
- VisualBrush (Komplexte Textdarstellung)

**Clipping** Mit `ClipToBounds` kann man definieren, ob Child Controls an den Rändern des Parent Controls abgeschnitten werden sollen. Mit `Clip` kann man definieren welche Form zum Zuschneiden eines Controls verwendet werden soll.

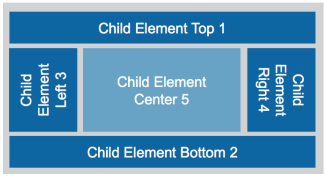
```
<Image.Clip>
    <EllipseGeometry
        RadiusX="100"
        RadiusY="75"
        Center="100, 75" />
</Image.Clip>
```

# Container Controls

Bei Container gibt es welche mit Layout und ohne Layout. Container mit Layout sind: **StackPanel**, **WrapPanel**, **DockPanel** und **Grid**. Container ohne Layout sind: **Canvas**, **ScrollViewer**, **Viewbox** und **Border**.

**StackPanel** Das StackPanel und WrapPanel haben ein Attribut `Orientation` welches entweder auf *Horizontal* oder *Vertical* gesetzt werden kann. Die Elemente werden entsprechend Angeordnet

**DockPanel** Das DockPanel ist ideal für eine Header, Footer Darstellung. Mit dem Attribut `DockPanel.Dock` kann man definieren, wo der Container sein wird. Beim DockPanel ist die Reihenfolge wichtig, denn der Verfügbare Platz wird in absteigender Reheinfolge verteilt

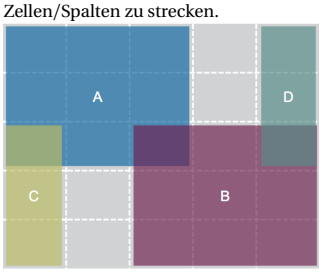



**Grid** Im Grid können Child Elemente einer Zelle darin zugeordnet werden. Die Zeilen und Spalten müssen explizit angegeben werden

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
        <ColumnDefinition></ColumnDefinition>
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
        <RowDefinition></RowDefinition>
    </Grid.RowDefinitions>
</Grid>
```

Listing 1: "3 x 3 Grid"

Auf den einzelnen Definitions kann man `Width` und `Height` definieren (sowie Min/Max etc.). Ebenfalls ist es mit `RowSpan` bzw. `ColumnSpan` möglich, Elemente über mehrere Zellen/Spalten zu strecken.




```
<Grid>
    <Grid.ColumnDefinitions>...</Grid.
      ↪ ColumnDefinitions>
    <Grid.RowDefinitions>...</Grid.RowDefinitions>
    <Button Content="A"
        Grid.Column="0" Grid.Row="0"
        Grid.RowSpan="3" Grid.ColumnSpan="3" />
    <Button Content="B"
        Grid.Column="2" Grid.Row="2"
        Grid.RowSpan="3" Grid.ColumnSpan="3" />
    <Button Content="C"
        Grid.Column="0" Grid.Row="2"
        Grid.RowSpan="3" Grid.ColumnSpan="1" />
    <Button Content="D"
        Grid.Column="4" Grid.Row="0"
        Grid.RowSpan="3" Grid.ColumnSpan="1" />
</Grid>
```

**Spezialfall** Bei einem 1 x 1 Grid werden die Elemente in der Zelle gestapelt und man kann mit Alignments und Margin ein flexibles Layout erstellen. Mit dem GridSplitter kann man die Spalten und Zellengössen anpassen. Mit der SharedSizeGroup können mehrere Spalten/Zeilen verknüpft werden um eine einheitliche Breite/Höhe festzulegen.

**Canvas** In einem Canvas Control haben alle Elemente eine absoulte Positionierung und es gibt keine Layout Logik. Die Position innerhalb des Canvas kann mittels Attached Properties festgelegt werden:

- `Canvas.Left`: Abstand vom linken Rand
- `Canvas.Top`: Abstand vom oberen Rand
- `Canvas.Right`: Abstand vom rechten Rand
- `Canvas.Bottom`: Abstand zum unteren Rand
- `Canvas.ZIndex`: Z-Position (Ebene) im Canvas

In Verbindung mit `Shapes` ist ein Canvas Control gut zum "programmierten Zeichnen" geeignet.

```
<Canvas>
    <Rectangle
        Canvas.Left="80" Canvas.Top="60"
        Width="128" Height="80"
        Fill="#006aa6" />
    <Ellipse
        Canvas.Left="260" Canvas.Top="160"
        Width="120" Height="120"
        Fill="#6E1C50" />
    <Path
        Canvas.Left="120" Canvas.Top="64"
        Width="260" Height="200"
        Stroke="DarkGray" Stretch="Fill"
        Data="M1,0 L0,1"/>
</Canvas>
```

**Shapes** Die Shapes Klasse beinhaltet `Ellipse`, `Polygon`, `Line`, `Polyline`, `Path` und `Rectangle`. Alle Shapes haben folgende Attribute:

- **Fill:** Brush für die Füllung
- **Stroke:** Pen für Rahmen/Pinselstrich
- **StrokeThickness:** Breite des Rahmens/Pinselstrichs
- **StrokeDasharray:** Muster für gestrichelte Linien

  **"2 1"** ist das Verhätniss zwischen Linie/Lücke immer 2:1

  **"2 1 4 1"** ist das Verhältniss zuerst 2:1, dann 4:1 dann 2:1 etc.
- **StrokeLineJoine:** Art des Übergangs bei Ecken

  **Miter:** Spitzer Rand

  **Bevel:** Abgeschnittener Rand

  **Round:** Abgerundeter Rand

**ViewBox** Skaliert, mittels Transformation, ein einzelnes Child Control um den Verfügbaren Platz auszunutzen. Das Attribut `Strech` kann verwendet werden um zu definieren wie das Control skaliert werden soll. Valid sind: `None` (keine Skalierung), `Uniform` (Vergrössern bis es ins Controll passt), `Fill` (Strecken, dass kompletter Platz verwendet wird), `UniformToFill` (Vergrössern und Strecken, dass Proportionen erhalten bleiben, aber ganzes Control ausgefüllt wird. Bei Übergrösse wird das Alignment berücksichtigt.

**Border** Die Border zeichnet lediglich ein Rahmen um ein Child Control. Es kann mit Panels kombiniert werden.

```
<Border Background="GhostWhite" CornerRadius="8,0,8,0"
        BorderBrush="#ddd" BorderThickness="1"
        Margin="10"
        Padding="10">
    <StackPanel Orientation="Vertical">
        <Button Margin="0,0,0,5" Content="Dock Sample
      ↪ 1" />
        <Button Margin="0,0,0,5" Content="Dock Sample
      ↪ 2" />
        <Button Margin="0,0,0,5" Content="Grid Sample
      ↪ 1" />
        <Button Margin="0,0,0,5" Content="Grid Sample
      ↪ 2" />
        <Button Margin="0,0,0,5" Content="Grid Sample
      ↪ 3" />
        <Button Margin="0,0,0,5" Content="Canvas
      ↪ Sample 3" />
    </StackPanel>
</Border>
```

## Input Controls

**Texteingabe** Für die Texteingabe gibt es die `TextBox` (Normalen Text Input, mit Option für mehrzeilige Eingabe), `RichTextBox` (Text Input mit Formatierung) und die `PasswordBox`.

**Buttons** Es gibt verschiedene Arten von Buttons: `Button`, `RepeatButton` (Löst Click-Event wiederholt aus, bis Button losgelassen wird), `ToggleButton` (hat zwei Zustände), `RadioButton`, `CheckBox`.

## List Control

**Listen** Es gibt verschiedene Listen in WPF. Die `ListBox` ist eine komplett sichtbare (scrollable) Liste mit Daten. Diese Einträge können auch Formatierungen beinhalten.

```
<ListBox SelectedIndex="1" SelectionMode="Multiple">
    <ListBoxItem>Erster Eintrag</ListBoxItem>
    <ListBoxItem>Zweiter Eintrag</ListBoxItem>
    <StackPanel Orientation="Horizontal">
        <Rectangle Width="10" Height="10" Fill="Blue"
      ↪ />
        <TextBlock Margin="10,0">Dritter Eintrag</
      ↪ TextBlock>
    </StackPanel>
</ListBox>
```

Die ComboBox ist eine normale DropDown Liste. Sie kann editierbar gemacht werden, mit `IsEditable` bzw. `IsReadOnly`.

```
<ComboBox SelectedIndex="0">
    <ComboBoxItem>Erster Eintrag</ComboBoxItem>
    <ComboBoxItem>Zweiter Eintrag</ComboBoxItem>
</ComboBox>
```

Eine `ListView` ist ein Read-Only DataGrid, eine generalisierte Anzeige einer Liste (Explorer Ansichten). Eine `TreeView` ist eine hierarchische Anzeige von Daten in einer Baumstruktur (Windows Explorer Pfad Baum).

## Beschriftungen

**TextBlock** Dies ist eine einfache Textanzeige, kein vollwertiges Control. Es kann auch als einfaches "Spacer" Element gebraucht werden.

**Label** Ein Label kann mittels Keyboard Shortcuts aufgerufen werden (vollwertiges Control). Es kann nicht nur Text anzeigen, sondern auch Layouts beinhalten und das Aussehen kann mittels Templates gestaltet werden.

**ToolTip** Das ToolTip kann kontextbezongene Infos bei einem Mouse-Over anzeigen, es kann ebenfalls ein beliebiges Layout beinhalten. Bei deaktivierten Controls muss die Property `ToolTipService.ShowOnDisabled` gesetzt werden, wenn ToolTip trotzdem erscheinen soll.

```
<Button.ToolTip>
    <StackPanel>
        <TextBlock Text="Submit Button"
            FontWeight="Bold" />
        <TextBlock Text="Submits ..." />
    </StackPanel>
</Button.ToolTip>
```

## Wertebereiche

**Progress Bar** wird verwendet um den Benutzer auf eine lange andauernde Operation hinzuweisen. Mit Container Controls kann dieser beliebig dekoriert werden.

```
<ProgressBar Minimum="0" Maximum="100" Value="75" />
```

**Slider** Der Slider gibt eine Auswahl aus einem vorgegeben Wertebereich zurück

```
<Slider Maximum="100" Value="75" TickFrequency="10"
      ↪ TickPlacement="BottomRight" />
```

## Organisation

**GroupBox** gruppiert visuell zusammengehörende Controls

```
<GroupBox Header="Mouse Handedness">
    <StackPanel>
        <RadioButton Content="Left-Handed" />
        <RadioButton Content="Right-Handed"
            IsChecked="True"/>
    </StackPanel>
</GroupBox>
```

**Expander** ermöglicht das Ein und Ausblenden von Infos bei Bedarf. Er wird oft für Zusatzinformationen gebraucht.

```
<Expander>
    <Expander.Header>More info</Expander.Header>
    <TextBlock TextWrapping="Wrap" Text="Drag ..." />
</Expander>
```

**TabControl** teilt das UI in mehrere Seiten auf. Jedes `TabItem` enthält ein eigenes Layout. Das Header Attribut kann mit beliebigem Layout gefüllt werden.

```
<TabControl>
    <TabItem Header="1 - Intro">
        <Grid Margin="10">
        ...
        </Grid>
    </TabItem>
    <TabItem Header="2 - Daten" />
    <TabItem Header="3 - Optionen" />
    <TabItem>
        <TabItem.Header>
            <StackPanel Orientation="Horizontal">
                <Grid>
                    <Ellipse Fill="Silver" Width="16"
                        Height="16" />
                    <Label Content="4" FontSize="10"
                        Foreground="White" />
                </Grid>
            <TextBlock Text="Review" />
            </StackPanel>
        </TabItem.Header>
    </TabItem>
</TabControl>
```

**Eigene Controls** Es gibt zwei Arten von Controls. Das `UserControl` welches als **Komposition** implementiert wird. Es ist eine Wiederverwendbare Zusammenstellung mehrer COntrols als Gruppe und besteht aus einem XAML und Code-Behind. Es kann nicht mit Styles/Templates umgehen. Das `CustomControl` ist eine Ableitung der zu verändernden Klasse. Es erweitert ein bestehendes Control um neue Funktionen. Es besteht aus einem Code-File und ggf. aus einem Standart-Style. Es kann mit Styles/Templates umgehen.

## Menüs

### Benutzerführung

Die höchste Klasse jeder XAML App ist die `System.Windows.Application`. Diese beinhaltet u.a. ein `Current` Property (Singelton) welches statischen Zugriff auf das Application Objekt bietet, das `MainWindow`, das Zugriff auf das Hauptfenster bietet und den `ShutdownMode` welche das Verhalten beim Programmende definiert.

**Current** Das Application Objekt ist als Singelton implentieret. Um es innerhalb der App zu verwenden, muss es oft gecastet werden.

```
public App MyApp => Application.Current as App;
```

**ShutdownMode** Der ShutdownMode definiert das Verhalten der App beim beenden, davon gibt es 3.

- `OnLastWindowClose`: Dies ist das Standardverhalten, die App beendet sich, sobald das letzte Fenster geschlossen wurde
- `OnMainWindowClose`: Die App wird beendet sobald das Hauptfenster geschlossen wird
- `OnExplicitShutdown`: Die App wird erst beendet, wenn die `Shutdown()` Methode aufgerufen wurde.

**StartupUri** Dies ist der Name der UI Ressource, die beim Start angezeigt werden soll, es ist normalerweise ein `Window`

**Windows** Das `Windows` Property ist eine Liste aller instanziierter Fenster (Window-Objekte) innerhalb der App.

**Application-Klasse Methoden** Die `LoadComponent` Methode lädt eine XAML-Ressource.

```
public static object LoadComponent(Uri uri)
```

Der Rückgabetyp muss in den korrekten Typ gecastet werden. Die `FindResouce` Methode, sucht eine Ressource mit dem angegeben Namen. Es durchsucht App-Ressourcen und System-Ressourcen. Der Rückgabetyp muss ebenfalls in den korrekten Typ gecastet werden.

```
public object FindResource(object resourceKey)
```

Die `Shutdown` Methode beendet die App und gibt optional einen mitgegebenen Exit Code ans System zurück.

```
public void Shutdown([int exitCode])
```

### Application-Klasse Events

- `Activated`: Die App wurde aktivitert (in den Vordergrund geholt)
- `Deactivated`: Die App wurde deaktiviert
- `DispatcherUnhandledException`: Eine nicht gefangene Exception ist aufgetreten
- `Exit`: Die App wird gleich beendet
- `FragmentNavitation`: Es wurde zu einem bestimmten `NavigationWindow/Frame` navigiert
- `LoadCompleted`: Aktuelles `NavigationWindow/Frame` wurde vollständig geladen
- `Navigated`: Aktuelles `NavigationWindow/Frame` wurde gefunden
- `Navigating`: Navigation zu einem `NavigationWindow/Frame` wurde angefordert
- `NavigationFailed`: Navigation zu einem `NavigationWindow/Frame` ist fehlgeschlagen
- `NavigationProgress`: Statusinformation zum Downloadstatus bei Internet-Ressourcen
- `NavigationStopped`: Ladevorgang des Inhalts der `NavigationWindow/Frame` wurde gestoppt
- `SessionEnding`: Windows Session wird beendet
- `Startup`: Die App startet gerade

**SplashScreen** Es gibt 2 Arten einen SplashScreen zu implementieren. Die automatische Variante ist, ein Bild in das Projekt zu kopieren und bei den FileProperties `BuildAction` auf SplashScreen zu setzten. Die kontrollierte Variante ist, in der `App_OnStartup` Methode den SplashScreen zu instanzieren und anzuzeigen.

```
private void App_OnStartup(object sender,
      ↪ StartupEventArgs e)
{
    var screen = new SplashScreen("media/splash.png");
    screen.Show(true);
}
```

Der Screen muss dann natürlich auch irgendwann wieder ausgeblendet werden. Alternativ kann man diesen auch nach einer gewissen Zeit ausblenden.

```
private void App_OnStartup(object sender,
      ↪ StartupEventArgs e)
{
    var screen = new SplashScreen("media/splash.png");
    screen.Show(true); // in den Folien steht false
    Thread.Sleep(2000);
    screen.Close(TimeSPan.FromMilliseconds(500));
}
```

### Window Klasse

Die `Window` Klasse beschreibt ein Fenster. Sie ist abgeleitet von `ContentControl` und erlaubt genau 1 Child Element (`LayoutContainer`).

- `Icon`: Dieses Property beinhaltet eine ICO-Datei, die als Fenster Icon verwendet wird (Ausführbaren Datei und Fensterttitel)
- `ShowInTaskBar`: Ein Property das angibt, ob das Fenster in der Taskleite angezeigt wird
- `Topmost`: Zeigt Fenster über allen andern Fenstern der Anwendung an
- `SizeToContent`: Legt fest, ob die Grösse eines Fensters automatisch an die Grösse des Inhalts angepasst wird.
  - `Manual`: Fenstergrösse anhand Width/Height Angabe(Standart)
  - `Height`: Höhe wird automatisch anhand des Inhalts festgelegt
  - `Width`: Breite wird automatisch anhand des Inhalts festgelegt
  - `WidthAndHeight`: Breite und Höhe werden automatisch anhand des Inhalts festgelegt
- `ResizeMode`: Gibt an, wie sich die Fenstergrösse ändern darf. Je nach Einstellung werden zusätzlich die Schaltflächen Minimieren/Maximieren im Fenstertitel angezeigt
  - `NoResize`: Fenstergrösse nicht änderbar
  - `CanMinimize`: Fenster kann minimiert und wiederhergestellt werden
  - `CanResize`: Fenstergrösse kann frei verändert werden (Standard)
  - `CanResizeWithGrip`: Wie `CanResize` aber mit zusätzlichem Resize Grip unten rechts im Fenster
- `WindowsStartupLocation`: Legt die Postition beim Start fest
  - `CenterOwner`: Fenster wird in Mitte des aufrufenden angezeigt
  - `CenterScreen`: Fenster wird in der Mitte des Bildschirm angezeigt
  - `Manual`: Position wird durch Left/Top Angabe bestimmt (Standart)

- `WindowState`: Beschreibt die Fensterzustände (Normal, Minimized, Maximized)
- `WindowStyle`: Gibt den Rahmentyp des Fensters an:
  `None`: Nur das Client Area ist sichtbar
  `SingleBorderWindow`: Fenster mit einfachem (dünnem) Rahmen
  `ThreeDBorderWindow`: Fenster mit 3D Rahmen
  `ToolWindow`: Verankertes Tool-Fenster

## Window-Klasse Methoden

- `Activate`: Fenster aktivieren
- `Close`: Fenster schliessen
- `DragMove`: Ermöglich das Verschieben des Fensters, falls die linke Maustaste gedrückt ist
- `GetWindow`: Statische Methode, ruft das Root-Window Objekt zum Control ab (DI)
- `Hide`: Macht das Fenster unsichtbar
- `Show`: Zeigt das Fenster an
- `ShowDialog`: Zeigt das Fenster an und blockiert, bis das Fenster geschlossen wird

## Window-Klasse Events

- `Activated`: Fenster wurde aktiviert
- `Closed`: Fenster wurde geschlossen
- `Closing`: Fenster wird gleich geschlossen
- `ContentRendered`: Fensterinhalt wurde gezeichnet
- `Deactivated`: Fenster wurde deaktiviert
- `LocationChanged`: Position des Fensters wurde geändert
- `StateChanged`: WindowState hat geändert

## Dialogfenster

Dialogfenster dienen zum Abruf von Daten und sind meist Modal (blockierend).
Ein Dialogfenster wird mit der Methode `ShowDialog` angezeigt. Im Dialogfenster kann die Property DialogResult gesetzt werden (boolean). Sobald dies gesetzt wurde wird das Dialogfenster geschlossen.

```
<Button Content="Cancel" IsCancel="true" />
<Button Name="OkButton" Content="OK" IsDefault="true"
    ↪ Click="OkButton_OnClick" />
```

Das `IsCancel` Property schliesst bei `true` das Dialogfenster automatisch (kein Event-Handling).

```
private void OkButton_OnClick(object sender,
    ↪ ReoutedEventargs e)
{
    SelectedCustomer = "MaxMuster";
    // trigger dialog close as side effect
    DialogResult = true;
}
```

## Fenster mit Spezialformen

Die Fensterrahmen können mittels Clipping verändert werden. Das setzt jedoch folgende Fenstereigenschaften voraus:

- `AllowsTransparency=true`
- `WindowStyle=none`
- `ResizeMode=NoResize`

Man kann die Form in `Window.Clip` mittels einem Geometry-Element festlegen

```
<!-- Rechteck mit abgerundeten Raendern -->
<Window.Clip>
    <RectangleGeometry RadiusX="8" RadiusY="8" Rect="
    ↪ 0,0,800,600" />
</Window.Clip>
```

## Menu

Die Menüs sind herkömmliche Windows-Menüs und werden üblicherweise am oberen Fensterrand angedockt. Mit dem Property `IsMainMenu=true` kann man das Menü standartmässig als Hauptmenü definieren.

**MenuItem** MenuItems sind beliebig verschachtelbar. Sie haben ein `Header` als anzuzeigenden Texts (Underscore = Accelerator-Key), `IsCheckable` definiert, ob der Eintrag ein Zustand ist (mit `IsChecked` kann Zustand gesetzt bzw. geprüft werden). Das `InputGestureText` Property definiert, definiert einen Text, der im MenuItem rechtsbündig angezeigt wird (Shortcuts wie Ctrl+V), um zu Funktionieren müssen diese noch behandelt werden, beispielsweise mit Commands.

```
<Menu DockPanel.Dock="Top">
    <MenuItem Header="_File">
        <MenuItem Header="_Recent">
            <MenuItem Header="Secret.txt" />
            <MenuItem Header="TopSecret.txt" />
        </MenuItem>
        <Separator />
        <MenuItem Header="E_xit" InputGestureText="
    ↪ CTRL + Q" />
    </MenuItem>
    <MenuItem Header="_Edit">
        <MenuItem Header="_Cut" InputGestureText="CTRL
    ↪ + X" />
        <MenuItem Header="C_opy" InputGestureText="
    ↪ CTRL + C" />
        <MenuItem Header="_Paste" InputGestureText="
    ↪ CTRL + V" />
        <Separator />
        <MenuItem Header="_Settings">
            <MenuItem Header="_Always On Top"
    ↪ IsChecked="True"></MenuItem>
            <MenuItem Header="_Modern UI" IsChecked="
    ↪ false"></MenuItem>
        </MenuItem>
    </MenuItem>
</Menu>
```

**Kontextmenus** Das `ContextMenu` stellt ein kontextspezifisches Menü zur verfügung. Es funktioniert wie das Menü und kann direkt mit der Property-Syntax für ein Control erstellt werden.

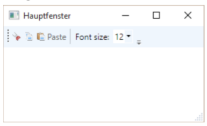
```
<Button.ContextMenu>
    <ContextMenu>
        <MenuItem Header="Hilfe" />
        <MenuItem Header="Fehler melden" />
    </ContextMenu>
</Button.ContextMenu>
```

**StatusBar** Die Statusbar dient zur Anzeige von zusätzlichen Informationen (meist am unteren Fensterrand). Es ist ein Container, ähnlich wie beim DockPanel, dabei füllt das letzte Element die ganze restliche Breite.

```
<StatusBar DockPanel.Dock="Bottom">
    <StatusBarItem Width="150">Press CTRL + Q to quit<
    ↪ /StatusBarItem>
    <Separator />
    <StatusBarItem>Ready</StatusBarItem>
</StatusBar>
```

Für Kontextbezogene Statusinformationen können `StatusBarItem` Elemente benannt werden und dann programmatisch darauf zugegriffen werden, oder man arbeitet mit Databinding.

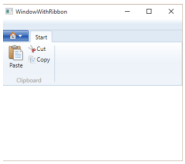
**Toolbar** kann entweder ein `ToolBarTray` sein, der als Container für Toolbars dient, oder ein `Toolbar` der als Container für Toolbar-Buttons und sonstige Toolbar-Controls dient. Die ToolBarTray ermöglicht das Verschieben und Umgruppieren der Toolbars und ist typischerweise angedockt.




```
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar>
        <Button Command="Cut" ToolTip="...">
            <Image Source="/media/clip_cut.png" Width=
    ↪ "12" Height="12" />
        </Button>
    <Button Command="Copy" ToolTip="Copy selection to
    ↪ Windows Clipboard.">
        <Image Source="/media/clip_copy.png" Width="12
    ↪ " Height="12" />
    </Button>
    <Button Command="Paste" ToolTip="Paste from
    ↪ Windows Clipboard.">
        <StackPanel Orientation="Horizontal">
            <Image Source="/media/clip_paste.png"
    ↪ Width="12" Height="12" />
            <TextBlock Margin="3,0,0,0">Paste</
    ↪ TextBlock>
        </StackPanel>
    </Button>
    <Separator />
    <Label>Font size:</Label>
    <ComboBox>
        <ComboBoxItem>10</ComboBoxItem>
        <ComboBoxItem IsSelected="True">12</
    ↪ ComboBoxItem>
        <ComboBoxItem>14</ComboBoxItem>
        <ComboBoxItem>16</ComboBoxItem>
    </ComboBox>
    </ToolBar>
</ToolBarTray>
```

**Ribbon** Das Ribbon ist seit Office 2007 bekannt und ist seither immer stärker verbreitet.

```
<Ribbon Name="Ribbon">
    <Ribbon.ApplicationMenu>
        <RibbonApplicationMenu SmallImageSource="media
    ↪ /home.png">
            <RibbonApplicationMenuItem Header="Hello
    ↪ _Ribbon"
                    Name="MenuItem1"
                    ImageSource="media/home.png"/>
        </RibbonApplicationMenu>
    </Ribbon.ApplicationMenu>
    <RibbonTab x:Name="HomeTab" Header="Start">
        <RibbonGroup x:Name="Clipboard" Header="
    ↪ Clipboard">
            <RibbonButton x:Name="Button1" Label="
    ↪ Paste"
                    LargeImageSource="media/clip_paste.png
    ↪ " />
            <RibbonButton x:Name="Button2" Label="Cut"
                    SmallImageSource="media/clip_cut.png"
    ↪ />
            <RibbonButton x:Name="Button3" Label="Copy
    ↪ "
                    SmallImageSource="media/clip_copy.png"
    ↪ />
        </RibbonGroup>
    </RibbonTab>
</Ribbon>
```



## Commands

bieten eine Alternative zum Event. Commands müssen nicht abgefangen und behandelt werden, sie werden durch die Control selbst aufgerufen. Es ist eine standartisierte Ausführung eines Befehls und ermöglicht die Wiederverwendung derselben Aktion. Eine Command-Methode muss `RoutedUICommand` als Rückgabetyp haben und statisch sein.

```
public static RoutedUICommand MyCutCommand = new
    ↪ RoutedUICommand(
    "Ausschneiden",
    "MyCut",
    typeOf(WindowWithToolbar)
);
```

Der `RoutedUICommand` nimmt auch noch einen 4. Paramter entgegen, in dem man `InputGestures` entweder einzeln oder als `InputGestureCollection` mitgeben kann.

```
<MenuItem Header="_Cut" InputGestureText="CTRL + X"
    ↪ Command="local:WindowWithToolbar.MyCutCommand
    ↪ " />
```

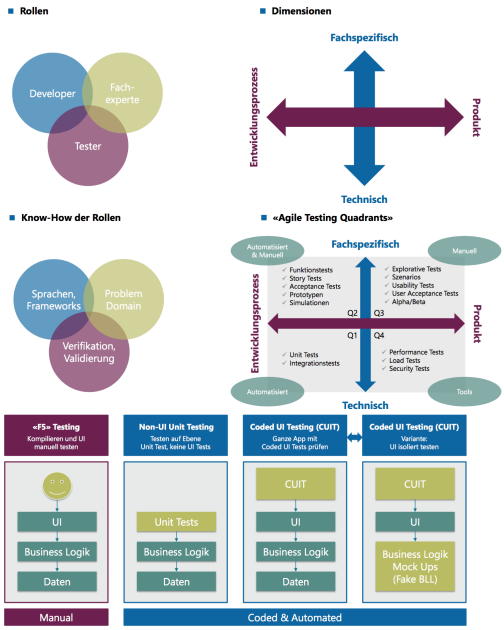
Commands können auch an Event-Handler gebunden werden und Shortcuts definiert werden.

```
<!-- Binding an Event -->
<Window.CommandBindings>
    <CommandBinding Command="local:WindowWithToolbar.
    ↪ MyCutCommand" Executed="MyCutCommand_Executed
    ↪ " />
</Window.CommandBindings>
<!-- Shortcut definieren -->
<Window.InputBindings>
    <KeyBinding Key="X" Modifiers="Control" Command="
    ↪ local:WindowWithToolbar.MyCutCommand" />
</Window.InputBindings>
```

Im Code-Behind kann dann wie gewohnt ein Handler definiert werden.

```
public void MyCutCommand_Executed(object sender,
    ↪ ExecutedRoutedEventArgs e){ ... }
```

## Automated UI Testing




Es gibt 4 Varianten UIs zu testen. Das **Non-UI Unit Testing** Test auf Code Ebene (normale Unit-Tests) und führt keine UI-Tests durch. Das **Coded UI Testing (CUIT)** testet die ganze App mit Coded UI Tests. Diese Variante kann man auch mit isolierten UI Komponenten durchführen, welche dann aber Fake Komponenten benötigen (die 4. Variante ist das F5-Testing, der Benutzer testet die App gleich selbst).

## TestStack

Der TestStack ist eine Sammlung von OpenSource Projekten um Tests in .NET Projekten zu automatisieren. Dazu gehört das **TestStack.White** Framework. Es ist ein CUIT-FW für WPF und basiert auf Microsofts UI Automation Framework.
Als Vorbereitung muss man eine Testklasse erstellen.

```
[TestClass]
public class MyUiTest
{
    [TestMethod]
    public void MyUiTestMethod(){ ... }
}
```

Danach muss man das Verzeichnis mit der WPF App Assembly festlegen. Dies kann dann als Read-Only Property in der Testklasse hinzugefügt werden.

```
// Directory in which tests are running
public string BaseDir => Path.GetDirectoryName(
    ↪ Assembly.GetExecutingAssembly().Location);
// System under test
public string SutPath => Path.Combine(BaseDir, ${
    ↪ nameof(MenusAndCommands)}.exe);
//$
```

Nun kann man die Tests entwickeln.
Zuerst neues Application Objekt aus dem WPF App Assembly erstellen:

```
var app = Application.Launch(SutPath);
```

Listing 2: "Neues Application Objekt aus dem WPF App Assembly erstellen"

Die Variable enthät ein UI-Automatisierungsobjekt des Typs `TestStack.Whie.Application`. Danach kann man die Fenster abrufen:

```
// Mit Fenstertitel abrufen
var window = app.GetWindow("Hauptfenster",
    ↪ InitializeOption.NoCache);
// Aus Liste der Fenster abrufen
var window = app.GetWindows.First();
// Anhand der ID(Name-Attribut) abrufen
var window = app.getWindow(SearchCriteria.
    ↪ ByAutomationId"(Win1"), InitializeOption.
    ↪ NoCache);
```

Die Variable `window` enthäht nun ein UI-Automatisierungsobjekt des Typs `TestStack.White.UIItems.WindoItems.Window`. Danach muss man die Controls abrufen:

```
// Anhand Beschriftung abrufen
var button = window.Get<Button>(SearchCriteria.ByText(
    ↪ "Speichern"));
// Anhand ID(Name-Attribut) abrufen
var button window.Get<Button>("SaveButton");
```

Die Variable `button` enthält nun ein UI-Automatisierungsobjekt des Typs `TestStack.White.UIItems.Button`.
Schlussendlich können Aktionen ausgeführt werden:

```
Assert.AreEqual("Speichern", button.Text);
button.Click();
Assert.AreEqual("Gespeichert!", button.Text);
```

```
var input = window.Get<TextBox>("NameInput");
Assert.IsTrue(string.IsNullOrEmpty(input.Text));
var newText = "You've been hacked!";
input.Text = newText;
Assert.AreEqual(newText, input.Text);
```

Schlussendlich muss die App noch mittels `app.Close()` beendet werden.
Mit `Desktop.CaptureScreenshot()` erhält man ein Bitmap Objekt mit einem Bild des aktuellen Bildschirms.

```
var screenshot = Desktop.CaptureScreenshot();
var path = System.IO.Path.Combine(BaseDir, "screenshot
    ↪ .png");
screenshot.Save(path, ImageFormant.Png);
```

# GUI Design

## WPF Ressourcen

Mit WPF Ressourcen kann man nützliche Objekte (Brushes, Styles, Templates etc.) zentral definieren und wiederverwenden.

### Resources

**Physikalische Ressourcen** In den File Properties kann man eine Datei als WPF-Ressource deklarieren welche dann in die Binärdatei einkompiliert werden. Zugegriffen kann dann mittels Resource-Key (relativer Pfadname).

**Resource** Jedes beliebige Objekt in XAML kann als Ressource definiert werden. Es bird mit einem Key-Attribut und dem x-Namespace benannt.

```
<Application.Resources>
    <SolidColorBrush x:Key ="MyButtonBackground" Color
      ↪ ="#EEEEEE" />
</Application.Resources>
```

Unter diesem Key sind Ressourcen dann auch ansprechbar

```
<Button Background="{StaticResource MyButtonBackground
    ↪ }" Content="Save" />
```

**ResourceDictionary** Ist ein Behälter um Ressourcen zu speichern. Es indexiert nach dem Ressourcen-Namen (`x:Key`) und kann in allen Elementen, welche von FrameworkElement ableiten, verwendet werden. Wenn auf eine Ressource zugegriffen werden möchte, wird *1.* Key im Element und allen Parent-Nodes gesucht (Logical Tree), *2.* dann wird der Key in `Application.Resources` gesucht, *3.* als letztes wird in System-Ressourcen gesucht.. Die Reheinfolge im XAML-Code ist wichtig.

**System Ressources** Auf Ressourcen in Namespace `System.Windows` kann man mittels statischer Properties zugegriffen werden. Dazu gehören: `SystemColors`, `SystemFonts` und `SystemParamters`.

```
StaticResource="{StaticResource[name]}"
```

Die Statische Bindung macht CompileTime Check und findet Fehler früh.

```
DynamicResource="{DynamicResource[name]}"
```

Anstelle von `[name]` steht der Key der Ressource In der Dynamische Bindung wird ein RuntimeCheck gemacht und lässt dynamisch erzeugte und geladene Ressourcen zu. Beim Static Binding wird bei Objektkonstruktion ausgewertet, beim Dynamic Binding einmal pro Zugriff.

**FindResource** Mit der Methode `FrameworkElement.FindResources` können Zugriffe auf XAML Resourcen gemacht werden.

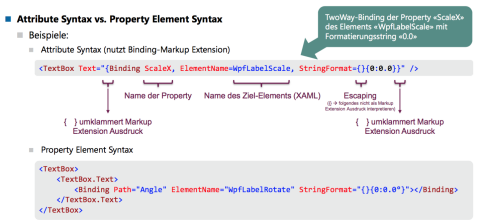
```
var okText = (string)FindResource("OkText");
var bgBrush = FindResource("DarkBrush") as Brush;
```

# Umgang mit Daten

Ein grosses Feature von WPF ist das DataBinding.

## Markup Extensions

- `{x:Type [Datentyp]}`: Liefert angegebene Klasse
- `{x:Static [Pfade]}`: Bindet eine Konstante, statische Property, Feld oder Enum
- `{x:Null}`: Null Wert
- `{StaticResource [Name]}`: Statische Bindung an Ressource
- `{DynamicResource [Name]}`: Dynamische Bindung an Ressource
- `{Binding ...}`: Data Binding Ausdruck
- `{RelativeSource ... }`: Setzt die Data Binding Source auf eine relevanten Bezug im Logical Tree
- `{TemplateBinding ...}`: Bindet Wert an Eigenschaft des mittels Template dargestellten Controls
- `{x:Reference ...}`: Abk. für `{Binding Elementname=...}`




## Überblick DataBinding

### Binding Base

- `Delay`: Verzögerung in Millisekunden zwischen DataBinding Operation
- `FallbackValue`: Standardwert, falls Data Binding nicht funktioniert
- `StringFormat`: Formatierungsangabe für die Umwandlung des Quellwertes in einen String
- `TargetNullValue`: Standardwert, falls Quellwert == null

### Binding

- `BindsDirectlyToSource`: Soll Angabe der Path-Property relativ zum aktuellen Datenprovider (`true`) oder relativ zum Datenkontext (`false`) ausgewertet werden (Nur selten sinnvoll)
- `Converter`: Converter der beim Binding benutzt werden soll
- `ConverterCulture`: Länderspezifische Einstellungen (`CultureInfo`), welche beim Konvertieren benutzt werden soll
- `ConverterParameter`: Zusätzlicher Parameterwert, der dem Converter übergeben werden soll
- `ElementName`: Name des XAML-Elements auf welches gebunden werden soll
- `Mode`: Richtung des DataBinding (OneTime, OneWay, TwoWay, OneWayToSource)
- `Path`: Pfad zur Datenquelle, Objektpfadsyntax
- `XPath`: XPath Ausdruck zum Zugriff auf eine XML Datenquelle
- `RelativeSource`: Setzt Datenquelle auf Objekt relative zum Ort des aktuellen Elements
- `Source`: Setzt Datenquelle
- `UpdateSourceTrigger`: Zeitpunkt zu welchem DataBinding getriggert wird

  Default: Nutzt festgelegte Trigger der Ziel Property

  Explicit: Nur beim expliziten Aufruf von `UpdateSource()`

  LostFocus: Fokusverlust des Elements

  PropertyChanged: Bei jeder änderung des Inhalts

### MultiBinding

- `Bindings`: Auflistung von Binding-Elementen
- `Converter`
- `ConverterCulture`
- `ConverterParamter`
- `Mode`: Richtung des DataBinding. TwoWay und OneWayToSource sind schwierig zu implementieren, da im MultivalueConverter aus 1 Wert, *n* Werte erzeugt werden müssen
- `UpdateSourceTrigger`
- `RelativeSource`
- `Source`

### DataContext, Source, RelativeSource

Die Datenquelle ist standartmässig nicht gesetzt, muss also explizit gesetzt werden.

**DataContext** Jedes Element, das von `FrameworkElement` ableitet, besitzt diese Property. Diese wird Standartmässig von DataBinding als Quelle genutzt und ist ebenfalls in Child-Controls gültig. Diese Property wird im Code-Behind, meist auf Ebene der Fenster, gesetzt.

**Source** Ermöglicht die Angabe der Datenquelle direkt im DataBinding Ausdruck. Dieses Property kann auf Ressourcen (Static/Dynamic) oder statische (Static) binden.

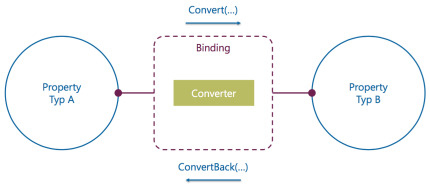
**RelativeSource** Ermöglicht die Angabe einer relativen Datenquelle im Visual Tree. Es gibt eine eigene Markup Extension dafür. Mit dem Property `Mode` gibt man den Suchmodus an. Modi dafür sind: `FindAncestor`(Sucht übergeordnetes Element des Typs), `PreviousData` (Bindet auf das vorhergehende Element, bsp: Delta Vergleiche), `Self` (Bindet auf das Element selbst), `TemplatedParent` (Bindet auf Element, für welches Control Template gilt, sinnvoll innerhalb Template). Das Property `AncestorLevel` Gibt die Vorgänger-Position an und der `AncestorType` ist der Typ des zu suchenden Vorgänger-Elements.

```
<Label Content="{Binding RelativeSource={
    ↪ RelativeSource FindAncestor, AncestorType=
    ↪ Window}, Path=Title}" />
```

**Path** Ist die Standart-Property eines Binding-Ausdrucks. Dieser kann deshalb weggelassen werden (`{Binding Firstname}` ist dasselbe wie `{Binding Path=Firstname}`). Für die Angabe der zu bindenden Property kann auch Objektsyntax verwendet werden (auch Array Syntax ist erlaubt).

## Converter

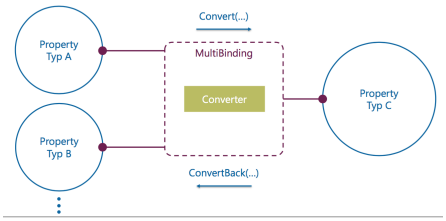
**IValueConverter** Ein Interface welches alle Converter implementieren müssen




```
public sealed class BooleanToVisibilityConverter:
      ↪ IValueConverter
{
    // value = bool oder nullable, targetType =
      ↪ Visibility
    public object Convert(object value, Type
      ↪ targetType, object parameter, CultureInfo
      ↪ culture)
    {
        return value is bool && (bool)value == true ?
      ↪ Visibility.Visible : Visibility.Collapsed;
    }
    // value = Visibility value, targetType = bool
    public object ConvertBack(object value, Type
      ↪ targetType, object parameter, CultureInfo
      ↪ culture)
    {
        return value as Visibility? == Visibility.
      ↪ Visible
    }
}
```

Listing 3: "Konvertiert bool oder Nullable in Visibility und zurück"

**IMultiValueConverter** Das Interface dass alle MultiConverter implementieren müssen.




```
public class RgbToColorConverter: IMultiValueConverter
{
    public object Convert(object[] values, Type
      ↪ targetType, object paramter, CultureInfo
      ↪ culture)
    {
        if (values == null)
            return DependencyProperties.Unsetvalue;
        if(values.Length != 3)
            throw new NotSupportedException("3 Values
      ↪ needed)";
        var r = (byte)System.Convert.ToInt32(values
      ↪ [0]);
        var g = (byte)System.Convert.ToInt32(values
      ↪ [1]);
        var b = (byte)System.Convert.ToInt32(values
      ↪ [2]);
        return Color.FromRgb(r,g,b);
    }
    public object[] ConvertBack(object value, Type[]
      ↪ targetType, object parameter, CultureInfo
      ↪ culture)
    {
        if(value == DependencyProperty.UnsetValue)
            return null;
        if(value is Color)
        {
            var color = (Color) value;
            var colors = new object[] {color.R, color.
      ↪ G, color.B};
            return colors;
        }
        return null;
    }
}
```

**Value Converter anwenden** Um einen Converter anwenden zu können wird eine Instanz benötigt. Instanziierung:

```
<Window.Resources>
    <local:RgbToColorConverter x:Key="
      ↪ MyRgbToColorConverter" />
    <local:BestContrastingColorConverter x:Key="
      ↪ MyBestContrastingColorConverter" />
</Window.Resources>
```

Listing 4: Instanzieren"

Danach kann man den Converter wie gewohnt nutzen:

```
<SolidColorBrush>
    <SolidColorBrush.Color>
        <MultiBinding Converter="{StaticResource
      ↪ MyRgbToColorConverter}">
            <Binding ElementName="ColorR" Path="Value"
      ↪ ></Binding>
            <Binding ElementName="ColorG" Path="Value"
      ↪ ></Binding>
            <Binding ElementName="ColorB" Path="Value"
      ↪ ></Binding>
        </MultiBinding>
    </SolidColorBrush.Color>
</SolidColorBrush>
<SolidColorBrush Color="{Binding Path=Content,
      ↪ ElementName=ColorLabel,
    Converter={StaticResource
      ↪ MyBestContrastingColorConverter}, Mode=OneWay
      ↪ }" />
```

Wenn man eigene Value Converters implementiert wird der XAML Code kürzer und man hat eine Entkopplung von Wert und dessen Darstellungseigenschaften. Aber es ist aufwändig und teilweise nicht trivial.

**Bindung auf eigene Objekte** Die sogenannten **DependencyProperties** ermöglichen ein Two-Way Binding und funktionieren mit Key-Value Dictionaries.Sie sind spezialisiert für die Vewendung in einem UI und somit nicht geeignet für Business Objects.

```
public int MyProperty
{
  get {return (int(GetValue(MyPropertyPropert); }
  set { SetValue(MyPropertyProperty, value); }
}
public static readonly DependencyProperty
      ↪ MyPropertyProperty =
    DependencyPropery.Register("MyProperty",
            typeof(int),
            typeoof(ownerclass),
            new PropertyMetadata(0));
```

Mit dem `INotifyPropertyChanged` Interface können Properties einer Klasse überwacht werden. Das Interface schreibt das Event `PropertyChanged` vor, das implementiert werden muss. Der Event Handler übermittelt den Namen der geänderten Property.

```
public class Person: INotifyPropertyChanged
{
    private string _firstName;
    public String FirstName
    {
        get {return _firstName; }
        set
        {
            if(value != _firstName)
            {
                _firstName = value;
                OnPropertyChanged(nameof(FirstName));
            }
        }
    }
    public event PropertyChangedeventHandler
      ↪ PropertyChanged;
    public void OnPropertyChanged(string name)
    {
        var handler = PropertyChanged;
        if (handler != null)
            handler(this, new PropertyChangedEventArgs
      ↪ (name));
    }
}
```

Dies kann auch mit einer Basisklasse gelöst werden, welche die Benachrichtigung implementiert.

```
public abstract class BindableBase:
      ↪ INotifyPropertyChanged
{
    public event PropertyChangedEventHandler
      ↪ PropertyChanged;
    protected bool SetProperty<T>(ref T field, T value
      ↪ , string name = null)
    {
        if(Equals(field,value))
            return false;
        field=value;
        OnPropertyChanged(name);
        return true;
    }
    protected void OnPropertyChanged(string name =
      ↪ null)
    {
        PropertyChanged?.Invoke(this, new
      ↪ PropertyChangedEventArgs(name));
    }
}
public class Person: BindableBase
{
    private string _firstName;
    public string FirstName;
    {
        get { return _firstName; }
        set { SetProperty(ref _firstName, value,
      ↪ nameof(FirstName)); }
    }
}
```

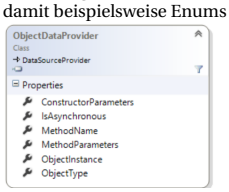
**Berechnete Properties** Möchte man Änderungen kommunizieren, sobald sich Quellwerte verändern, muss bei jedem Wechsel eine eigene Notification versendet werden.

```
protected bool SetProperty<T>(ref T storage, T value,
      ↪ string name, params string[] otherNames)
{
    if(Equals(storage, value))
        return false;
    storage = value;
    OnPropertyChanged(name);
    foreach(var n in otherNames)
        OnPropertyChanged(n);
    return true;
}
```

**PropertyChanged.Fody** Fody ist ein Framwork, welches sich in den Kompilationsprozess einhängt und automatisch Properties überwacht. Damit Fody weiss welche Properties er überwachen muss, muss man es mit dem Attribut `ImplementPropertyChanged` markieren.

**ObservableCollection** Diese Klasse implementiert das INotifyCollectionChanged und INotifyPropertyChanged Interface. Diese Collection meldet Änderungen an sich automatisch, somit ist Event Handling auf neue Listeninhalte, bestimmte Positionen oder die gesamte Liste möglich.

**ObjectDataProvider** Ermöglicht Verwendung einer beliebigen Datenquelle mit folgenden Zusatzmöglichkeiten. Es können Parameter an den Konstruktor übergeben werden (ConstructorParameters) oder es können Methoden inkl. Paramter gebunden werden (MethodParameters). Man kann damit beispielsweise Enums auf eine ComboBox binden.




```
<Window.Resources>
    <ObjectDataProvider x:Key="Alignments"
        MethodName="GetNames"
        ObjectType="{x:Type sys:Enum}">
    <ObjectDataProvider.MethodParameters>
        <x:Type TypeName="VerticalAlignment" />
    </ObjectDataProvider.MethodParameters>
    </ObjectDataProvider>
</Window.Resources>
<!— Verwendung —>
<ComboBox ItemsSource="{Binding Source={StaticResource
      ↪ Alignments}}" />
```

**DataBinding Debuggen** Man hat diverse Möglichkeiten die Abläufe hinter dem DataBinding sichtbar zu machen.

1. Direkt im Binding konfigurieren

```
<!— System.Diagnostics—Namespace – hinzufuegen
      ↪ —>
xmlns:diag="clr—namespace:System.Diagnostics;
      ↪ assembly=WindowsBase"
<!— Binding Ausdruck um Setzten des
      ↪ TraceLevels erweitern —>
<TextBlock Text="{Binding ElementName=stack,
      ↪ Path=InvalidPath,
      ↪ diag:PresentationTraceSources.
      ↪ TraceLevel=High}" />
```

2. Dummy Converter schreiben

```
public class DebugDummyConverter :
      ↪ IValueConverter
{
    public object Convert(object value, Type
      ↪ targetType, object parameter,
      ↪ CultureInfo culture)
    {
        return value;
    }
    public object ConvertBack(object value,
      ↪ Type targetType, object parameter,
      ↪ CultureInfo culture)
    {
        return value;
    }
}
```

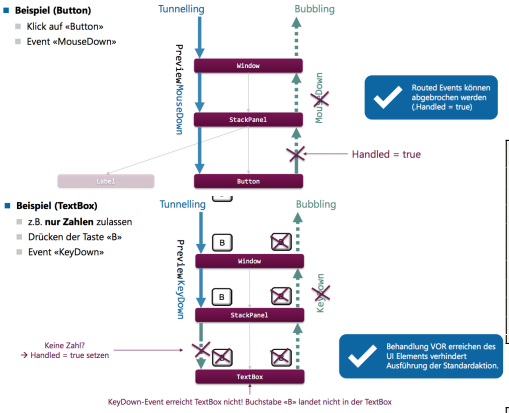
```
<Window.Resources> ... <
      ↪ local:DebugDummyConverter x:Key="
      ↪ MyDummy" /> ... </Window.Resources>
...
<TextBlock Text="{Binding ElementName=stack,
      ↪ Path=InvalidPath, Converter={
      ↪ StaticResource MyDummy}}" />
```

3. In VisualStudio DataBinding Debugging auf "All" setzten und im Output-Window nach `Syste.Windows.Data` suchen.

# Benutzerinteraktion

## Routed Events

Events wandern vom Window den Visual Tree bis zum auslösenden Element. Routed Events besitzen meist ein Preview und ein normales Event. Das Preview Event wird ausgeführt, bevor es das Element erreicht, das normale Event, nachdem das Element das Event behandelt hat.




Routed Events können mit `Handeled = true` unterbochen werden, sie werden dann nicht mehr weitergereicht. Man kann das Event direkt beim UI Element selbst behandeln.

```
<Button Name="SaveButton"
    PreviewMouseDown="SaveButton_OnPreviewMouseDown"
    MouseDown="SaveButton_OnMouseDown">Save</Button>
```

Oder beim Parent Element

```
<StackPanel PreviewMouseDown="
      ↪ StackPanel_OnPreviewMouseDown">
    ...
    <Button Name="SaveButton"
        PreviewMouseDown="
      ↪ SaveButton_OnPreviewMouseDown"
        MouseDown="SaveButton_OnMouseDown">Save</
      ↪ Button>
    ...
</StackPanel>
```

Eine andere Alternative wäre mit Attached Events.

```
<StackPanel Button.Click="StackPanel_OnClick">
    ...
    <Button Name="SaveButton" Click="
      ↪ SaveButton_OnClick">Save</Button>
    ...
</StackPanel>
```

Beliebte Mouse Events:

| | |
|---|---|
| MouseDown | MouseEnter* |
| MouseLeave* | MouseLeftButtonDown |
| MouseLeftButtonUp | MouseMove |
| MouseRightButtonDown | MouseRightButtonUp |
| MouseUp | MouseWheel |

| | | |
|---|---|---|
| Interessante Keyboard Events: | KeyDown<br>TextInput | KeyUp |
| Interessante Touch-Events: | TouchDown<br>TouchLeave*<br>TouchUp | TouchEnter*<br>TouchMove |

* haben kein Preview Event.

## RoutedEventArgs

RoutedEventArgs leiten von EventArgs ab.

- `Handled`: Boolean der aussagt, ob das Event behandelt wurde
- `OriginalSource`: Quell-Element (Hit-Testing) welches das Event ausgelöst hat
- `RoutedEvent`: Das RoutedEvent, welches mit diesem Objekt verknüpft ist
- `Source`: Element, welches das Event rapportiert hat

Die OriginalSource kann ein Kind-Element des Source-Elements sein, das per Hit-Testing als 1. Adressat des Events bestimmt wurde. Die Source ist das Element im Logical Tree, welches das Event rapportiert hat. Beispielsweise beim Subscribe auf MouseDown auf einem Button, wird das Event vom Button rapportiert (Source), wurde aber eigentlich vom Rahmen ausgelöst (OriginalSource).

## Ableitende Klasse von RoutedEventArgs

Es gibt für verschiedene Input-Geräte Ableitungen von RoutedEventArgs. Auswahl von `MouseEventArgs`:

- **LeftButton**: Zustand der linken Maustaste (Pressed/Released)
- **MiddleButton**
- **RightButton**
- **Delta**: Mausradbewegung (-120/120)

Auswahl von `KeyEventArgs`:

- **Key**: Taste (Enum)
- **IsDown**
- **IsUp**
- **IsRepeat**
- **SystemKey**: Die zusätzliche Taste, falls ALT-Taste gedrückt wurde. Key steht in `Key.System`

# Drag & Drop

Drag & Drop besteht aus 3 Phasen:

1. **Maustaste wird gedrückt:** Nach überschreiten einer Mindestdistanz wird die Aktion gestartet
2. **Während des Ziehens:** Steuerelemente unterhalb des Zeigers müssen melden, ob sie als Ziel infrage kommen
3. **Fallenlassen:** Steuerelement unterhalb des Mauszeigers muss eine Aktion mit dem bewegten Objekt durchführen

Beispiel:

```
public ObservableCollection<UserInfo> AvailableUsers {
      ↪ get; set; }
public ObservableCollection<UserInfo> SelectedUsers {
      ↪ get; set; }
public MainWindow()
{
    // Listen initialisieren und fuellen
    InitializeComponent();
    DataContext = this;
}
```

## Phase 1

Maustaste wird gedrückt

```
<ListBox Name="AvailableListBox" ItemsSource="{Binding
      ↪  AvailableUsers}"
    ItemTemplate="{StaticResource UserInfoTemplate}"
    PreviewMouseLeftButtonDown="AvailableListBox_
      ↪ OnPreviewMouseLeftButtonDown"
    PreviewMouseLeftButtonUp="AvailableListBox_
      ↪ OnPreviewMouseLeftButtonUp"
    MouseMove="AvailableListBox_OnMouseMove" />
```

```
private Point? dragStartPosition = null;
private void AvailableListBox_
      ↪ OnPreviewMouseLeftButtonUp(object sender,
      ↪ MouseButtonEventArgs e)
{
    dragStartPosition = null;
}
private void AvailableListBox_
      ↪ OnPreviewMouseLeftButtonDown(object sender,
      ↪ MouseButtonEventArgs e)
{
    dragStartPosition = e.GetPosition(this);
}
private bool IsMovementFarEnough(Point origPos, Point
      ↪ curPos)
{
    var minDistX = SystemParameters.
      ↪ MinimumVerticalDragDistance;
    var minDistY = SystemParameters.
      ↪ MinimumHorizontalDragDistance;
    return (Math.Abs(curPos.X - origPos.X) >= minDistX
      ↪  || Math.Abs(curPos.Y - origPos.Y) >=
      ↪ minDistY);
}
private void AvailableListBox_OnMouseMove(object
      ↪ sender, MouseEventArgs e)
{
    // gar nicht starten, wenn nicht (innerhalb Liste)
      ↪  geklickt
    if (dragStartPosition == null)
        return;
    // aktuelle Position holen und pruefen, ob die
      ↪ Maus genug
    // bewegt wurde, um die Bewegung als Drag zu
      ↪ interpretieren
    var position = e.GetPosition(this);
    if (!IsMovementFarEnough(dragStartPosition.Value,
      ↪ position))
        return;
    // Alles ok, drag kann starten
    dragStartPosition = null;
    StartDrag(AvailableListBox.SelectedItem as
      ↪ UserInfo);
}
private void StartDrag<T>(T obj)
{
    // Bereich definieren, auf dem der Drag-Vorgang
      ↪ gueltig ist
    var dragScope = this.Content as FrameworkElement;
    // Container fuer Nutzdaten (hier String)
    var dragData = new DataObject(typeof(T), obj);
    // Drag-Vorgang starten
    DragDrop.DoDragDrop(dragScope, dragData,
      ↪ DragDropEffects.Move);
}
```

Die `DragDrop.DoDragDrop` Methode blockiert die weitere Code-Ausführun bis die Operation beendet ist.

## Phase 2

Maus wird gezogen

```
<ListBox Name="SelectedListBox" ItemsSource="{Binding
      ↪ SelectedUsers}"
    ItemTemplate="{StaticResource UserInfoTemplate}"
    AllowDrop="True"
    DragOver="SelectedListBox_OnDragOver"
    Drop="SelectedListBox_OnDrop" />
```

```
private void SelectedListBox_OnDragOver(object sender,
      ↪  DragEventArgs e)
{
    // Objekt auspacken
    var data = e.Data.GetData(typeof(UserInfo)) as
      ↪ UserInfo;
    // falls Objekt verfuegbar, dann kopieren, sonst
      ↪ keine Operation
    e.Effects = data != null ? DragDropEffects.Copy :
      ↪ DragDropEffects.None;
}
```

## Phase 3

Objekt wird fallengelassen

```
<ListBox Name="SelectedListBox" ItemsSource="{Binding
      ↪ SelectedUsers}"
    ItemTemplate="{StaticResource UserInfoTemplate}"
    AllowDrop="True"
    DragOver="SelectedListBox_OnDragOver"
    Drop="SelectedListBox_OnDrop" />
```

```
private void SelectedListBox_OnDrop(object sender,
      ↪ DragEventArgs e)
{
    // Objekt auspacken
    var user = e.Data.GetData(typeof(UserInfo)) as
      ↪ UserInfo;
    // Dank Data Binding reicht es nun, das UserInfo
      ↪ Objekt
    // der ObservableCollection hinzuzufuegen:
    SelectedUsers.Add(user);
}
```

## Drag & Drop Events

- **DragEnter:** Tritt auf, wenn dieses Element während einer Drag-Operation als Drag Target fungieren würde (Hit-Testing)
- **DragLeave:** Tritt auf, wenn dieses Element während einer Drag-Operation verlassen wird
- **DragOver:** Tritt auf, wenn diesem Element eine Drag-Operation stattfindet
- **Drop:** Tritt auf, wenn das Objekt der Drag-Operation auf dieses Element fallengelassen wird
- **GiveFeedback:** Gibt der Quelle der Drag-Operation eine Chance, visuelles Feedback zu geben

# Hintergrund-Operationen

Lange Dauernde Operationen sollten nicht im UI-Thread ausgeführt werden, sonden in ein eigenen Thread ausgelagert werden, sodass das UI nicht einfriert. Um den Benutzer zu informieren, dass irgendwas "arbeitet" sollte man folgendermassen vorgehen:

1. Visuelles Feedback geben (Spinner, Overlay, Popup), wenn nötig
2. Starten eines Background-Threads als Reaktion auf ein Event, Kontrolle an UI Thread zurückgeben
3. Bei Thread-Ende aus dem Background-Thread den UI-Thread benachrichtigen

## Visuelles Feedback

Das visuelle Feedback umfasst auch, dass der Benutzer dieselbe Operation nicht aus versehen zweimal ausführt. Dies verhindert man, indem man beispielsweise den Button deaktiviert etc. Um dem Benutzer mittzuteilen, dass etwas läuft, kann man beispielsweise ein Spinner auf den Button legen (realisierbar durch Grid innerhalb des Buttons).

## Starten eines Background Threads

Dazu verwendet man am besten die Task Paralell Library.

```
Task.Run(() => {
    //Background operation
});
```

## UI-Thread benachrichtigen

Mit der TPL kann man einen Dispatcher verwenden, um Code im UI-Thread auszuführen.

```
Task.Run(() => {
    //Background operation
    Dispatcher.Invoke(() => {
        // UI interaktion
    });
});
```

Falls Dispatcher nicht aus Code-Behind aufgerufen wird (Dispatcher ist eine Property der UI-Elemente wie Window, muss man stattdessen `System.Windows.Threading.Dispatcher.CurrentDispatc` verwenden.