

# ***WPF***

## ***Einführung***

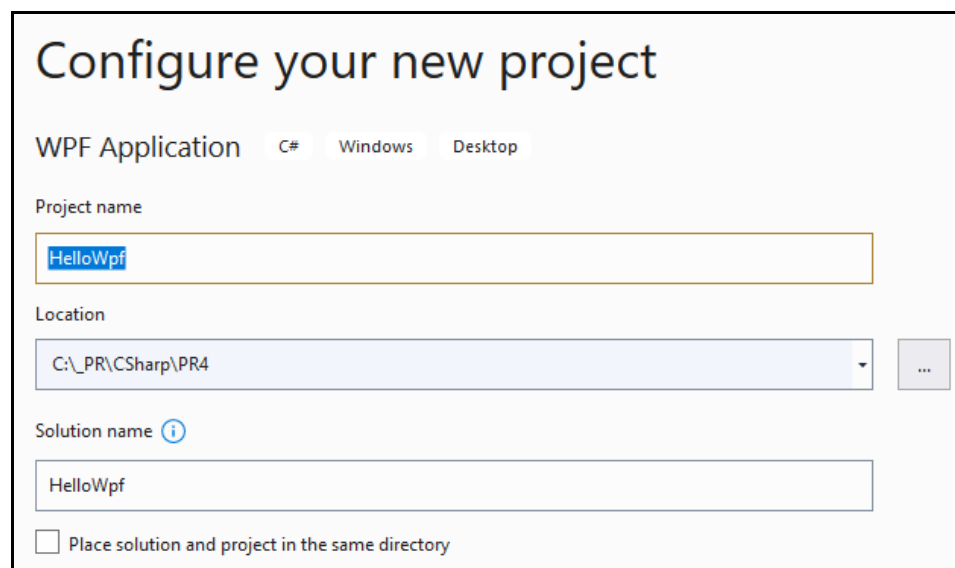
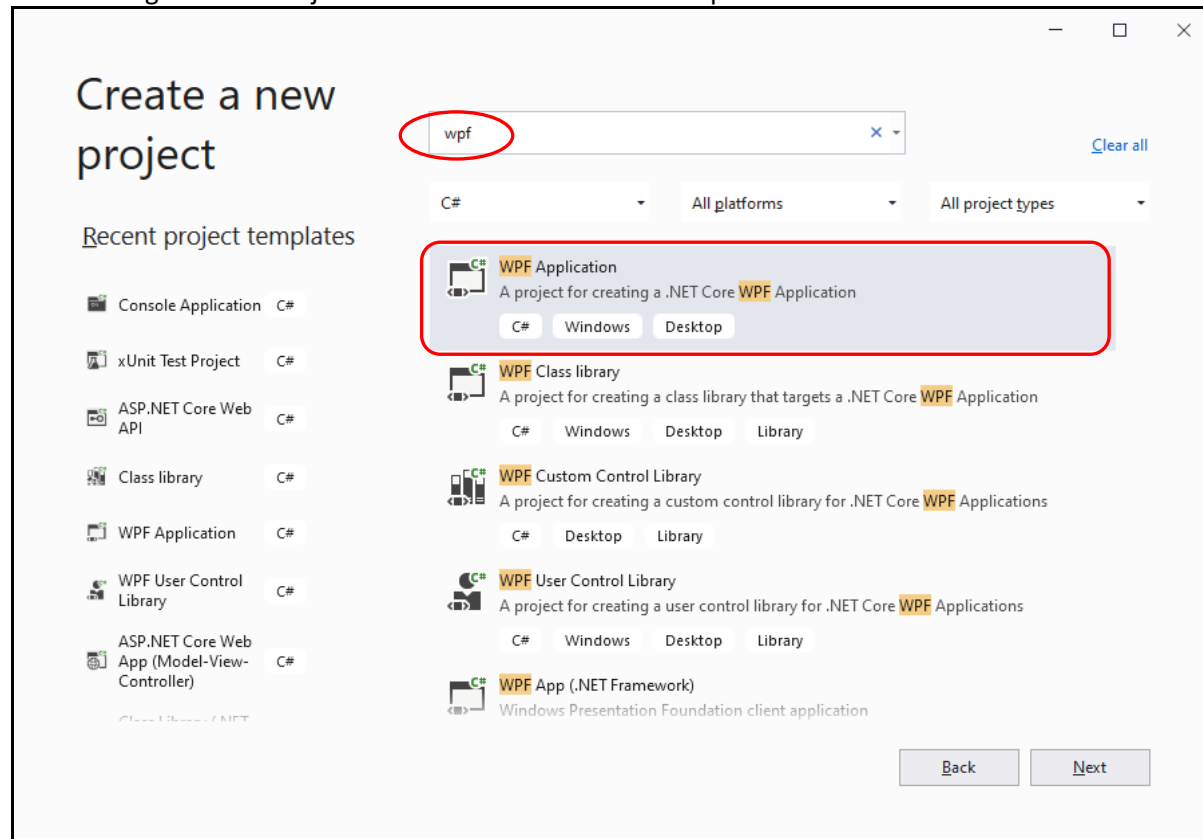
|   |    |
|---|----|
| 1 WPF                                   | 2  |
| 1.1 Projekt erstellen                   | 2  |
| 1.1.1 Anpassung für Console.Log         | 3  |
| 1.2 Projektstruktur                     | 3  |
| 1.3 WYSWYG                              | 4  |
| 1.4 MainWindow.xaml                     | 5  |
| 1.5 Events                              | 6  |
| 1.5.1 Generierter Code                  | 6  |
| 1.6 Loaded                              | 7  |
| 1.7 Properties der Controls             | 7  |
| 1.7.1 XAML - Attribut-Syntax            | 7  |
| 1.7.2 C#                                | 8  |
| 1.8 App.xaml                            | 8  |
| 1.8.1 Main                              | 9  |
| 1.8.2 Warum App?                        | 9  |
| 1.9 Logical vs. Visual Tree             | 10 |
| 2 ZUSATZINFOS                           | 11 |
| 2.1 Klassendiagramm                     | 11 |
| 2.2 Events der Klasse App               | 11 |
| 2.3 Live Visual Tree                    | 12 |
| 2.4 Treenavigation per C#               | 12 |
| 2.4.1 Navigieren/Suchen im Logical Tree | 12 |
| 2.4.2 Navigieren im Visual Tree         | 13 |

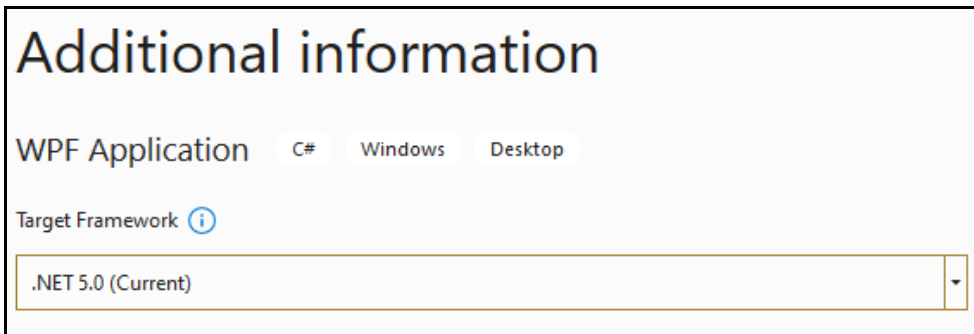
# 1 WPF

WPF (Windows Presentation Foundation) ist die Windows-Lösung für graphische Desktopanwendungen. Sie basiert auf einem XML-ähnlichen Code zur Gestaltung des UI und dahinterliegenden C#-Code für die Business Logic.

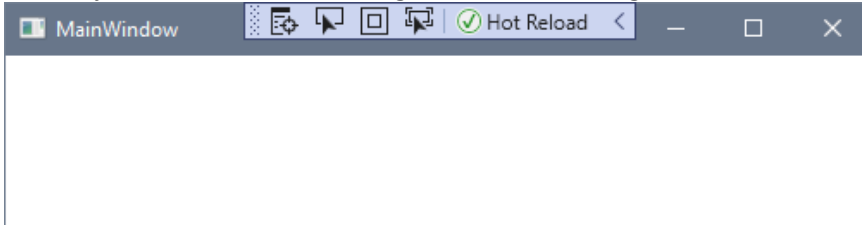
## 1.1 Projekt erstellen

Ein lauffähiges leeres Projekt bekommt man über den entsprechenden Wizard:





Das Projekt ist ohne Benutzereingriff sofort startfähig - startet man die App, sieht sie so aus:



### 1.1.1 Anpassung für Console.Log

Damit beim Start ein Konsolenfenster erscheint und darin dann etwaige Logs, muss man zwei Änderungen im csproj-File durchführen:

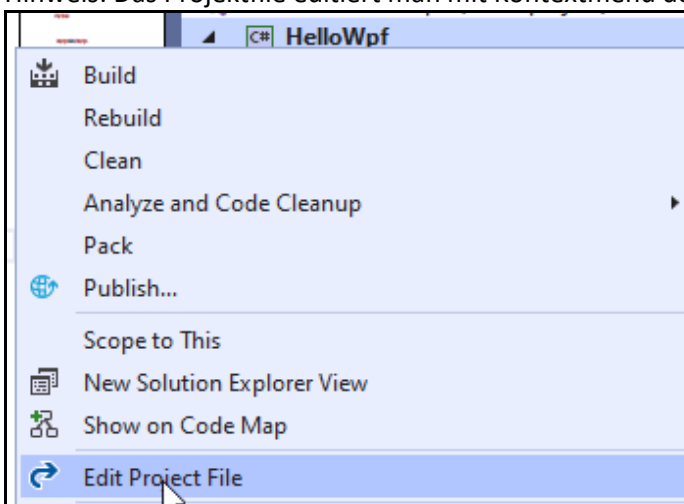
- **OutputType**: WinExe → Exe
- **DisableWinExeOutputInference** auf true setzen

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0-windows</TargetFramework>
    <UseWPF>true</UseWPF>
    <DisableWinExeOutputInference>true</DisableWinExeOutputInference>
  </PropertyGroup>

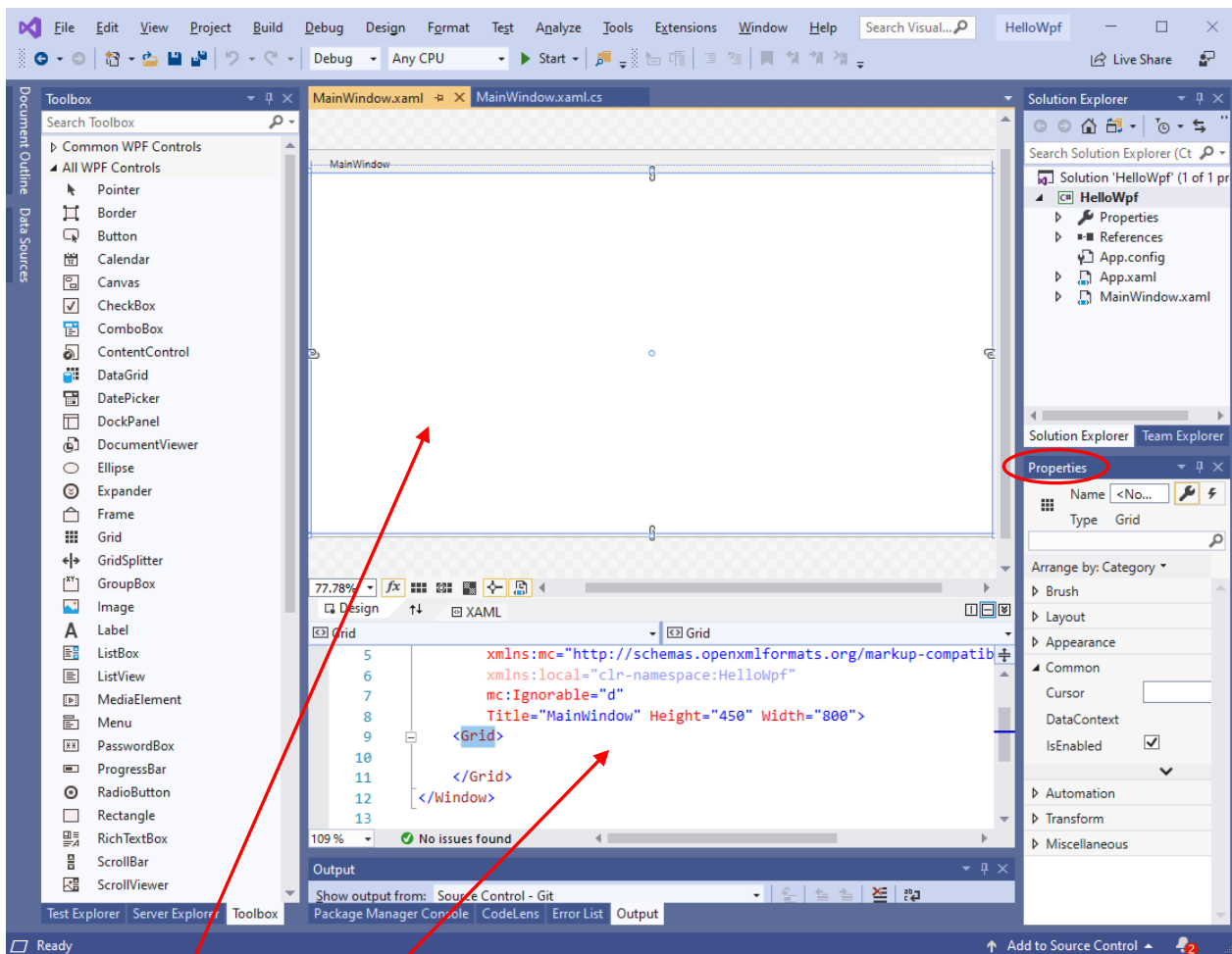
</Project>
```

Hinweis: Das Projektfile editiert man mit Kontextmenü des Projekts → Edit Projekt File

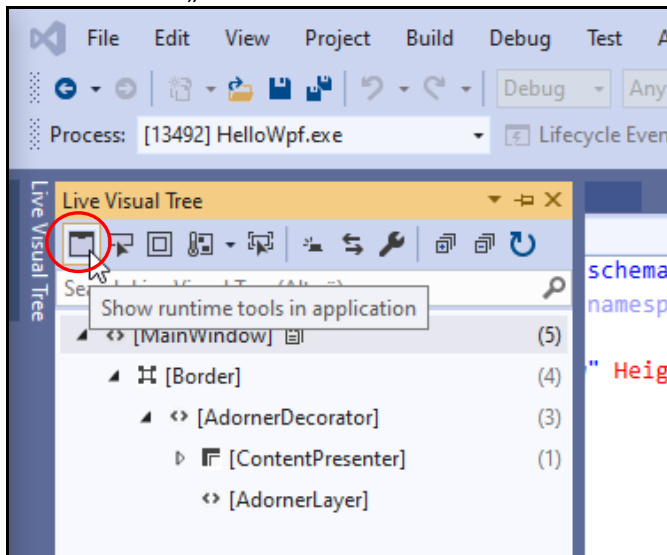


## 1.2 Projektstruktur

Dadurch erhält man folgendes Gerüst:

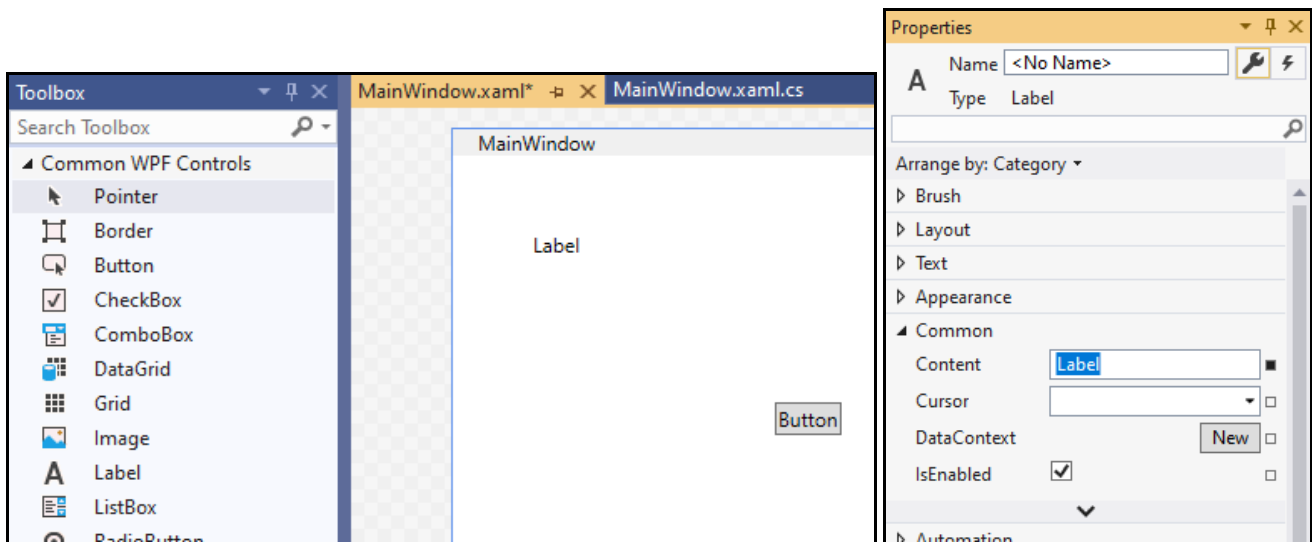


Die einzelnen Komponenten werden grafisch sowie als **XAML** dargestellt. Selektiert man eine Komponente (entweder grafisch oder im XAML), kann man im Properties-Fenster gewünschte Einstellungen vornehmen. Die graue Button-Leiste ist zum Debuggen sichtbar (siehe weiter unten). Falls man das nicht haben will, kann man das im Fenster „Live Visual Tree“ ausblenden:



### 1.3 WYSWYG

Der Editor funktioniert so wie erwartet und müsste grundsätzlich ohne weitere Erläuterung verwendbar sein. Einfach Elemente aus der Toolbox in das Fenster ziehen.



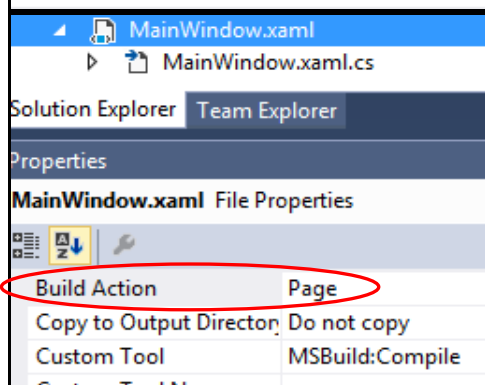
Die Eigenschaften können im Fenster „Properties“ rechts unten eingestellt werden. Die Änderungen sind sofort sichtbar.

## 1.4 MainWindow.xaml

Die Wurzel der XAML-Hierarchie muss ein Window-Element sein, dem mit **x:Class** jene Klasse zugewiesen wird, in der der C#-Code notiert ist. Die XAML-Datei ist zu Beginn praktisch leer und als Build Action ist Page eingetragen:

```
<Window x:Class="Hellowpf.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Hellowpf"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```



Von der Toolbox kann man die gewünschten Controls einfach in das Fenster ziehen. Die XAML-Datei ändert sich dabei automatisch mit. Die einzelnen Einstellungen kann man dann sowohl im Properties-Fenster als auch im XAML-File vornehmen, z.B.:

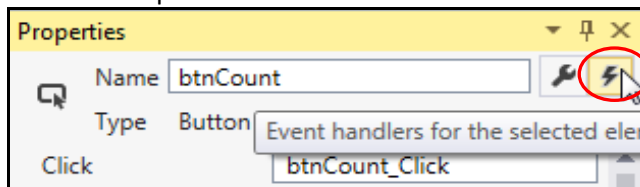
```
<Grid>
  <Button Name="btnCount" Content="Click to increase"
    HorizontalAlignment="Left" VerticalAlignment="Top"
    Margin="136,39,0,0" Width="107"/>
  <Label Name="lblCount" Content="Label"
    HorizontalAlignment="Left" VerticalAlignment="Top"
    Margin="68,36,0,0" />
</Grid>
```

Die Konfiguration über das Properties-Fenster ist selbsterklärend.

## 1.5 Events

Ein Klick-Event für den Button kann man folgendermaßen einstellen:

- Doppelklick auf das Control (weil Klick das Default-Event für Buttons ist)
- `Click="..."` im XAML als Attribut beim Button eintragen (das entsteht auch beim Doppelklick auf Button)
- über das Properties-Fenster einstellen. Dazu muss man rechts oben auf Events umschalten:



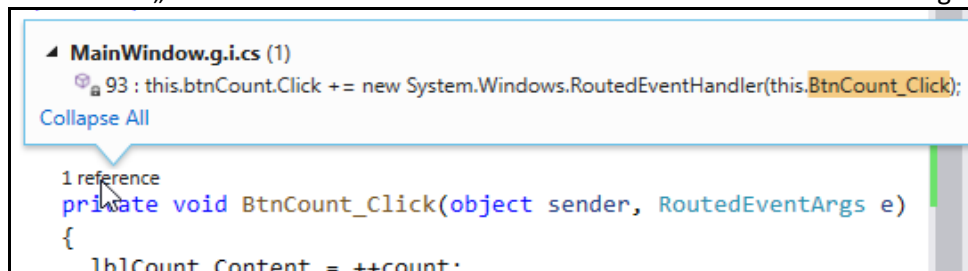
In der Codebehind-Datei **MainWindow.xaml.cs** wird dann der Methodenrumpf erzeugt. Wie man sieht, kann man darin dann für Controls die Properties verändern:

```
public partial class MainWindow : Window
{
    private int count = 0;
    0 references
    public MainWindow() ...
    1 reference
    private void BtnCount_Click(object sender, RoutedEventArgs e)
    {
        lblCount.Content = ++count;
    }
}
```

```
<Button Name="btnCount" Content="Click to increase"
  HorizontalAlignment="Left" VerticalAlignment="Top"
  Margin="136,39,0,0" Width="107"
  Click="BtnCount_Click"/>
```

### 1.5.1 Generierter Code

Das wird möglich, weil nämlich für jedes Control automatisch eine Variable mit dem bei **Name** angegebenen Namen angelegt wird. Dieser Code wird bei jedem Build in der Datei **MainWindow.g.cs** erzeugt (wie man auch am Hinweis „1 reference“ der Code Lens erkennen kann – aber erst nach erfolgtem Kompilieren):



```

public partial class MainWindow : System.Windows.Window, Syst

    #line 10 "..\..\MainWindow.xaml"
    [System.Diagnostics.CodeAnalysis.SuppressMessageAttribute
    internal System.Windows.Controls.Button btnCount;

    #line default

void System.Windows.Markup.IComponentConnector.Connect(int connectionId, object target)
{
    switch (connectionId)
    {
        case 1:
            this.btnCount = ((System.Windows.Controls.Button)(target));

            #line 13 "..\..\MainWindow.xaml"
            this.btnCount.Click += new System.Windows.RoutedEventHandler(this.BtnCount_Click);
            #line default

```

## 1.6 Loaded

Ein Window-Objekt kann ebenfalls Events registrieren. Das wichtigste dabei ist das **Loaded**-Event, das ausgelöst wird, wenn alle Controls aufgebaut wurden. Das wäre also die Stelle, in der man z.B. Listen mit Daten initialisieren kann.

```

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

private void Window_Loaded(object sender, RoutedEventArgs e)
{
    btnCount.Content = "Erhöhen";
}

```

Um zu verhindern, dass man im Konstruktor herumprogrammiert, kann man diesen auf einen Lambda-Ausdruck umschreiben:

```

public MainWindow() => InitializeComponent();

```

## 1.7 Properties der Controls

Ein Attribut in XAML wird auf eine Property oder ein Event der zugehörigen Klasse zugewiesen. Den Wert kann man dabei im XAML oder auch in C# setzen.

### 1.7.1 XAML - Attribut-Syntax

Attribute müssen laut XML-Syntax in Anführungszeichen gesetzt werden. Sie werden auf **gleichnamige Properties der Klasse** gesetzt. Dabei wird der Typ bei Bedarf automatisch von **String** auf **double**, **int**,... oder auch komplexere Typen (wie bei z.B. bei **Margin**) umgewandelt. Der anzuzeigende Text heißt für einen Button **Content**, für eine TextBox **Text**.

```

<TextBox Name="txtInput" HorizontalAlignment="Left" Margin="35,44,0,0" VerticalAlignment="Top" Width="120"
         Text="BlaBlaBla" />
<Button Name="btnUseIt" HorizontalAlignment="Left" Margin="179,42,0,0" VerticalAlignment="Top"
        Content="Click Me"
        Click="btnUseIt_Click" />
<Label Name="lblMessage" HorizontalAlignment="Center" Margin="0,41,0,0" VerticalAlignment="Top"
        Content="Label" />

```

## 1.7.2 C#

In C# kann man die Eigenschaften auch ganz einfach über die Properties verändern. Wichtig ist dabei, dass das Control mit der Property **Name** einen Namen erhält – mit diesem Namen wird automatisch eine Instanzvariable erzeugt.

```
public partial class MainWindow : System.Windows.Window, System.Windows.Markup.IComponentConnector {
    ///...
    #line 15 "..\..\..\MainWindows.Xaml"
    internal System.Windows.Controls.Label lblMessage;

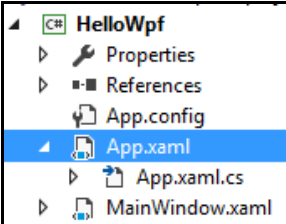
    ...
    0 references
    void System.Windows.Markup.IComponentConnector.Connect(int connectionId, object target) {
        switch (connectionId)
        {
            ///...
            case 3:
                this.lblMessage = ((System.Windows.Controls.Label)(target));
            return;
        }
        this._contentLoaded = true;
    }
}
```

Die Properties heißen jetzt genauso wie die Attribute in XAML und können einfach gesetzt werden:

```
private void btnUseIt_Click(object sender, RoutedEventArgs e)
{
    string text = txtInput.Text;
    lblMessage.Content = text;
}
```

## 1.8 App.xaml

Diese Komponente stellt den **Startpunkt** der App dar. Sie sieht so aus und muss normalerweise nicht verändert werden:



```
<Application x:Class="HelloWpf.App"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:local="clr-namespace:HelloWpf"
    StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

Wichtig ist dabei das Attribut **StartupUri**, das angibt, welches Fenster zu Beginn angezeigt werden soll.

Die zugehörige Codebehind-Datei (sie heißt **App.xaml.cs**) ist leer:

```
using ...

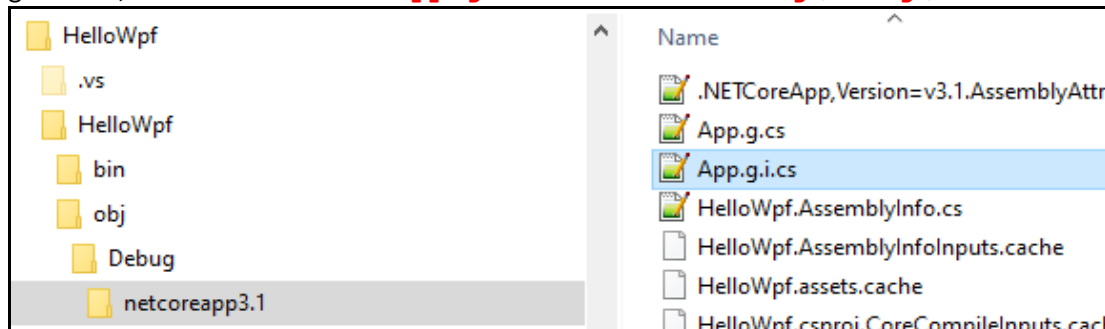
namespace HelloWpf
{
    /// <summary>
    /// Interaction logic for App.xaml
    /// </summary>
    3 references
    public partial class App : Application
    {
    }
}
```



### 1.8.1 Main

Wie man sieht, fehlt die Methode **Main()**, die ja für eine C#-Anwendung benötigt wird. Wo steht also diese Methode?

Wie vielleicht auffällt, ist App eine **partial class**. Jetzt wird beim Kompilieren Code zu dieser partial class generiert, und zwar in der Datei **App.g.i.cs** im Verzeichnis **obj\Debug\net5.0-windows**.



Diese Datei sieht so aus:

```
public partial class App : System.Windows.Application {

    /// <summary> InitializeComponent
    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuildTasks", "4.0.0.0")]
    public void InitializeComponent() {

        #line 5 "..\..\App.xaml"
        this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);
        #line default
        #line hidden
    }

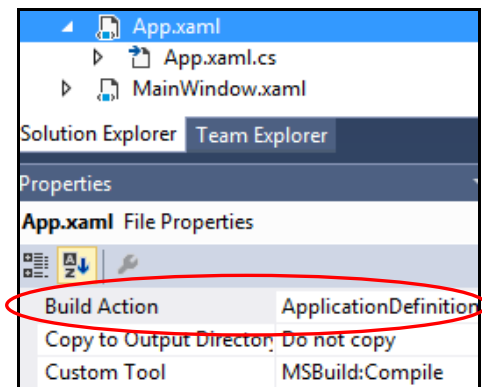
    /// <summary> Application Entry Point.
    [System.STAThreadAttribute()]
    [System.Diagnostics.DebuggerNonUserCodeAttribute()]
    [System.CodeDom.Compiler.GeneratedCodeAttribute("PresentationBuildTasks", "4.0.0.0")]
    public static void Main() {
        HelloWpf.App app = new HelloWpf.App();
        app.InitializeComponent();
        app.Run();
    }
}
```

In **Main()** wird ein App-Objekt erzeugt und mit diesem dann die **Run()**-Methode aufgerufen. Es hat keinen Sinn, hier etwas zu programmieren, weil sie bei jeder Kompilierung neu erstellt wird.

### 1.8.2 Warum App?

Man kann sich noch fragen, warum die Klasse App als Startpunkt erkannt wird, und nicht etwa eine andere Klasse.

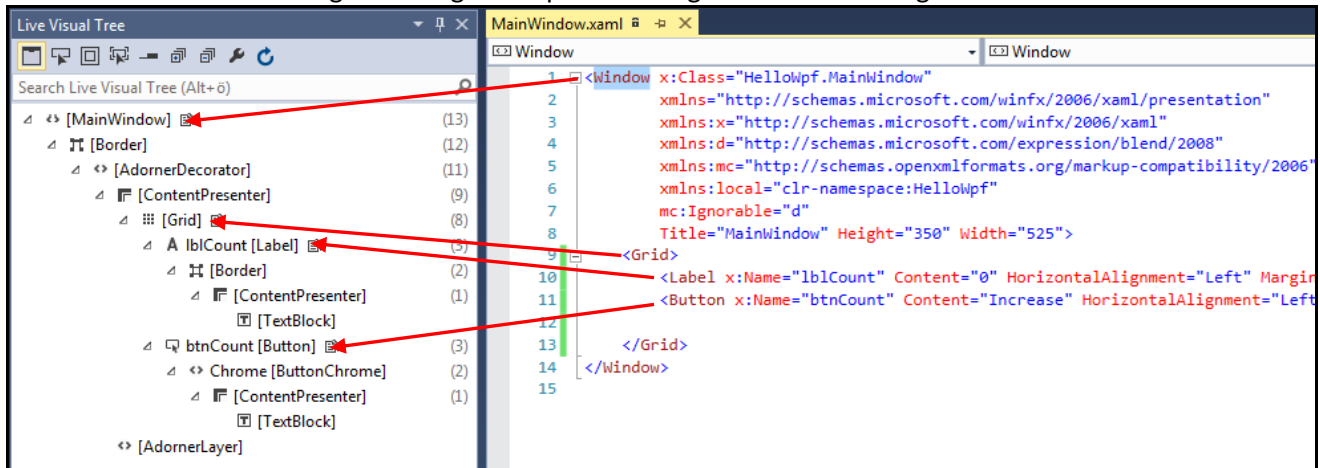
- Erstens muss die Klasse von **Application** abgeleitet sein.
- Zweitens muss das dem Compiler explizit mitgeteilt werden. Das erfolgt über die Angabe **Application Definition** für die **Build Action** (es darf nur eine Klasse diese Build Action haben).



## 1.9 Logical vs. Visual Tree

Als **Logical Tree** bezeichnet man jenen Tree, der direkt aus der **XAML-Struktur** entsteht. Er bildet die Grundstruktur des Fensters. In obigem Beispiel hat das Window-Element ein Kind (nämlich <Grid>), das wiederum zwei Kinder hat.

Der **Visual Tree** ist eine Obermenge des Logical Tree. Hier werden bei der **tatsächlichen Darstellung** des Fensters zusätzliche Elemente erzeugt. Für obiges Beispiel wird folgender Visual Tree gerendert:



Offensichtlich wird ein Node im Logical Tree auf bestimmte Weise dargestellt, also in einen Node im Visual Tree transformiert. Dies geschieht durch sogenannte **Templates**.

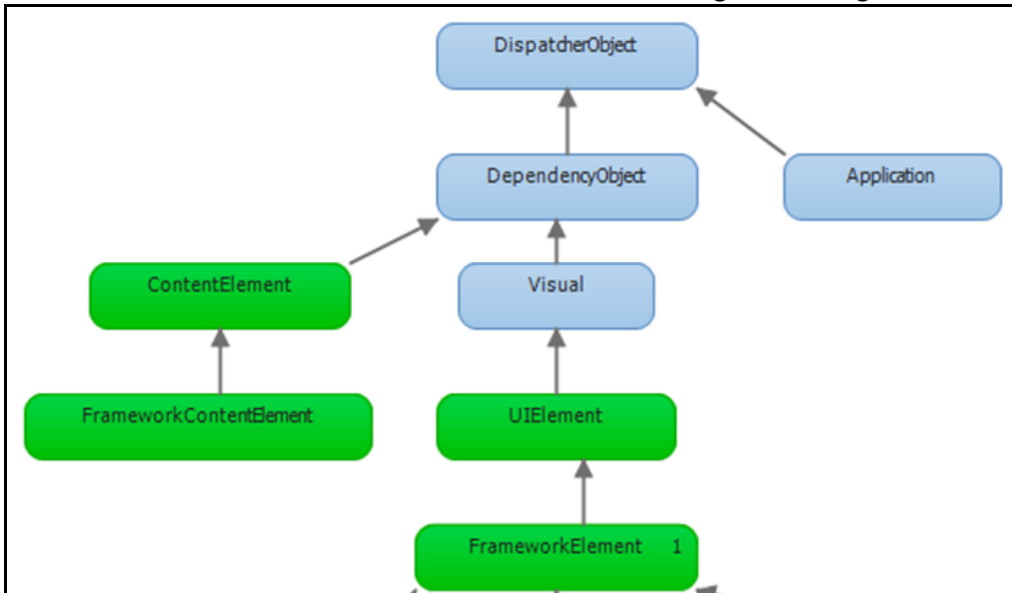
- Der Visual Tree wird beim Zuweisen von Events benötigt.
- Der Logical Tree ist zum Auffinden von Ressourcen, Styles und Templates wichtig.

Man kann in beiden Bäumen navigieren, ausgehend von einem **DependencyObject** (in den Beispielen unten depObj genannt).

## 2 Zusatzinfos

### 2.1 Klassendiagramm

WPF besteht aus vielen Klassen - die Basisklassen sind in folgendem Diagramm hierarchisch angeordnet:



Die jeweiligen Unterklassen werden in den folgenden Kapiteln besprochen.

### 2.2 Events der Klasse App

Beim Start werden einige Events ausgelöst, auf die man in der App-Klasse reagieren könnte, z.B. OnStartup:

```

public partial class App : Application
{
    override ons
}
  
```

OnSessionEnding(SessionEndingCancelEventArgs e)  
OnStartup(StartupEventArgs e)

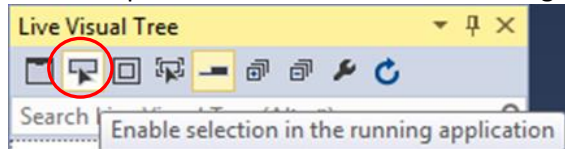
Hier hat man Zugriff auf die Commandline-Argumente (über die Property **Args** des Parameters **StartupEventArgs**) und könnte etwa auch zusätzliche Fenster starten:

```

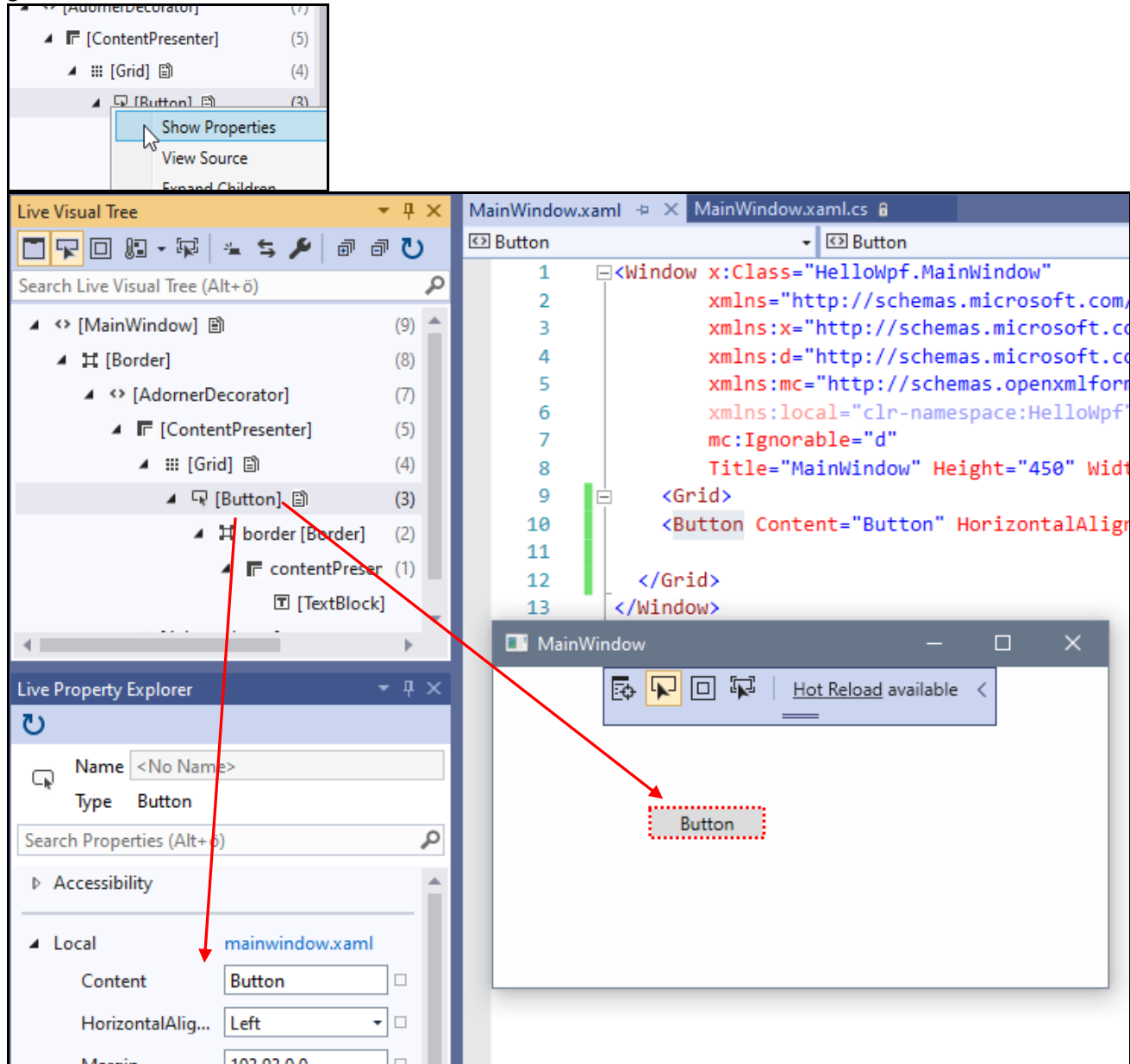
public partial class App : Application
{
    0 references
    protected override void OnStartup(StartupEventArgs e)
    {
        for (int i = 0; i < 3; i++)
        {
            var window = new MainWindow
            {
                Title = $"Window #{i}"
            };
            window.Show();
        }
    }
}
  
```

## 2.3 Live Visual Tree

Hilfreich ist auch der zweite Toggle-Button im Live Visual Tree, mit dem man im UI Elemente auswählen kann, deren Properties dann im Live Visual Tree angezeigt werden.



Im Live Property Explorer werden die Eigenschaften des ausgewählten Elements angezeigt und können dort auch geändert werden.



## 2.4 Treenavigation per C#

### 2.4.1 Navigieren/Suchen im Logical Tree

In der Klasse **LogicalTreeHelper** gibt es Methoden zum Navigieren in einem Logical Tree:

- Abwärts: **LogicalTreeHelper.Children(depObj)**
- Aufwärts: wie bei einer Baumstruktur gewohnt, gibt es für jeden Knoten nur einen Vorgänger. Diesen erhält man mit **LogicalTreeHelper.Parent(depObj)**

Sowie auch zum Suchen:

- Abwärts suchen: mit `LogicalTreeHelper.FindLogicalNode(nodeFrom, name2find)`  

```
var nodeName = txtFind.Text;  
var depObj = LogicalTreeHelper.FindLogicalNode(root, nodeName);
```
- Aufwärts suchen: `LogicalTreeHelper.FindName(nodeFrom, name2find)`

## 2.4.2 Navigieren im Visual Tree

Für das Navigieren im Visual Tree ist die Klasse `VisualTreeHelper` zuständig. Der Weg abwärts ist dabei leicht anders gelöst, indem man nur die Anzahl der Kinder eruieren kann und dann mit einem Index auf das jeweilige Kindelement zugreift:

```
for (int i = 0; i < VisualTreeHelper.GetChildrenCount(depObj); i++)  
{  
    TraverseVisualTree(VisualTreeHelper.GetChild(depObj, i), tvi);  
}
```

Die Methode aufwärts heißt aber ebenfalls `GetParent(depObj)`.