11 Wichtige WPF-Techniken

Nachdem wir im vorhergehenden Kapitel einige praktische Erfahrungen mit den WPF-Controls gesammelt haben, wollen wir uns jetzt mit den Besonderheiten der WPF-Programmierung im Vergleich zur Windows-Forms-/Win32-Programmierung beschäftigen.

Damit Sie die WPF-Philosophie verstehen, wollen wir Ihnen zu Beginn einige wichtige Eckpfeiler dieser Technologie erklären.

11.1 Eigenschaften

Da die Definition und Verwendung von Eigenschaften in WPF etwas anders organisiert ist als in den bekannten Windows-Forms-Anwendungen, sollten wir zunächst auf dieses Thema näher eingehen.

11.1.1 Abhängige Eigenschaften (Dependency Properties)

Mit den abhängigen (Dependency) und angehängten (Attached) Eigenschaften erweitert WPF das Spektrum der CLR-Eigenschaften. Abhängige Eigenschaften bieten gegenüber den „normalen“ Eigenschaften folgende erweiterte Möglichkeiten:

- eine interne Prüfung (Validierung),
- automatisches Aktualisieren von Werten,
- die Verwendung von Callback-Methoden zur Signalisierung von Wertänderungen
- sowie die Vorgabe von Defaultwerten.

Notwendig wurde diese Erweiterung des Eigenschaftensystems, um viele der WPF-Features (Animationen, Datenbindung, Styles etc.) zu realisieren. So werden beispielsweise Werte von Eigenschaften überwacht und Änderungen führen automatisch zu Änderungen an den abhängigen Objekten.

Abhängige Eigenschaften werden nicht als private Felder definiert, sondern als statische, öffentliche Instanzen der Klasse *System.Windows.DependencyProperty*, die über *get-* und *set-*Methoden angesprochen werden. Die Verwaltung der Eigenschaft wird vom WPF-Subsys-

tem übernommen (daher auch die *Register*-Methode, siehe folgendes Beispiel 11.1). Die der Eigenschaft übergeordnete Klasse stellt quasi nur noch eine Schnittstelle zu dieser Eigenschaft zur Verfügung.

Neben dem Wert können mit dem Defaultwert und der Callback-Methode auch weitere Metadaten zu einer Eigenschaft gespeichert werden.

Beispiel 11.1: Definition einer abhängigen Eigenschaft *Durchmesser* für ein Objekt *Kreis*

C#

```
using System.Windows;

namespace Dependency_Attached_Properties;
```

Wir definieren eine neue Klasse *Kreis* und leiten diese gleich von *DependencyObject* ab:

```
public class Kreis : DependencyObject
{
```

Hier die eigentliche Definition der abhängigen Eigenschaft (übergeben werden der Name, der Datentyp, das abhängige Objekt und optional die Metadaten, d. h. der Defaultwert und eine Callback-Methode):

```
    public static readonly DependencyProperty DurchmesserProperty =
         DependencyProperty.Register(nameof(Durchmesser), typeof(double),
         typeof(Kreis), new FrameworkPropertyMetadata(11.1,
         new PropertyChangedCallback(OnDurchmesserChanged)));
```

Wie Sie sehen, handelt es sich nicht mehr um ein privates Feld. Vielmehr wird die bisher übliche Kapselung aufgegeben, die Verwaltung des Zustands wird von WPF übernommen, die Instanz meldet seine „lokalen Speicher“ nur noch an (*Register*).

Die folgende Definition ist nur noch die allgemeine Schnittstelle nach außen, wie Sie es auch von den normalen Eigenschaften kennen:

```
    public double Durchmesser
    {
        get
        {
            return (double)GetValue(DurchmesserProperty);
        }
        set
        {
             SetValue(DurchmesserProperty, value);
        }
    }
```

Hier eine Callback-Methode, mit der eine Wertänderung überwacht werden kann:

```
    private static void OnDurchmesserChanged(DependencyObject obj,
            DependencyPropertyChangedEventArgs args)
    {
        MessageBox.Show((obj as Kreis)?.Durchmesser.ToString());
    }
}
```

11.1.2 Angehängte Eigenschaften (Attached Properties)

Mit den angehängten Eigenschaften (*Attached Properties*), einer speziellen Form der Dependency Properties, wird der Versuch unternommen, die Flut von WPF-Element-Eigenschaften etwas einzudämmen. Das Problem: Wird ein Element/Control in einem Container platziert, hängt beispielsweise dessen Position vom umgebenden Container (*Grid*, *Canvas*, *DockPanel* etc.) ab. Für jede Art von Container werden aber andere Eigenschaften zur Positionierung benötigt. Aus diesem Grund stellen die übergeordneten Elemente die zum Positionieren nötigen Eigenschaften zur Verfügung (*Canvas.Top, Canvas.Left, DockPanel.Dock*, *Grid.Column*, …), das eingelagerte Element nutzt diese lediglich in seinem Kontext. Der Vorteil: Nur wenn sich beispielsweise ein *Button* in einem *Canvas* befindet, werden auch die Eigenschaften *Canvas.Top, Canvas.Left* bereitgestellt und verwendet.

Beispiel 11.2: Positionieren einer Schaltfläche in einem *Canvas*-Control mit Attached Properties

XAML

```
<Canvas>
   <Button Canvas.Left="74" Canvas.Top="70" Height="45" Width="89">
      Beschriftung
   </Button>
 </Canvas>
```

11.2 Einsatz von Ressourcen

Bevor wird uns im Weiteren mit Styles etc. beschäftigen, müssen wir zunächst noch einen Blick auf die Ressourcenverwaltung in WPF-Anwendungen werfen, da diese die Voraussetzung für die Zuordnung darstellt.

11.2.1 Was sind eigentlich Ressourcen?

In WPF-Anwendungen zählen nicht nur Grafiken, Strings, Sprachinformationen oder beliebige binäre Informationen zu den Ressourcen, sondern auch die Beschreibung von Styles, Füllmustern oder sogar ganzer Controls.

Eine Ressource besteht immer aus einem eindeutigen Schlüssel (Key) und dem eigentlichen Content, der jederzeit austauschbar ist, da Bezüge auf die Ressource immer nur den Schlüssel verwenden.

11.2.2 Wo können Ressourcen gespeichert werden?

Eine Möglichkeit, Ressourcen in Ihrer Anwendung abzulegen, haben Sie sicher ganz unbewusst schon zur Kenntnis genommen. Die Rede ist von der Datei *App.xaml*, in der bereits ein *Resources*-Abschnitt vordefiniert ist:

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

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

Hierbei handelt es sich um anwendungsweit verfügbare Ressourcen, die allen Windows/Pages und deren Elementen zur Verfügung stehen.

Neben diesen Ressourcen können Sie auch jedem Element und jedem Window eigene Ressourcen zuordnen, zusätzlich sind auch sogenannte Systemressourcen (z. B. Systemfarben) verfügbar.

Bild 11.1 zeigt die mögliche Hierarchie von Ressourcen.

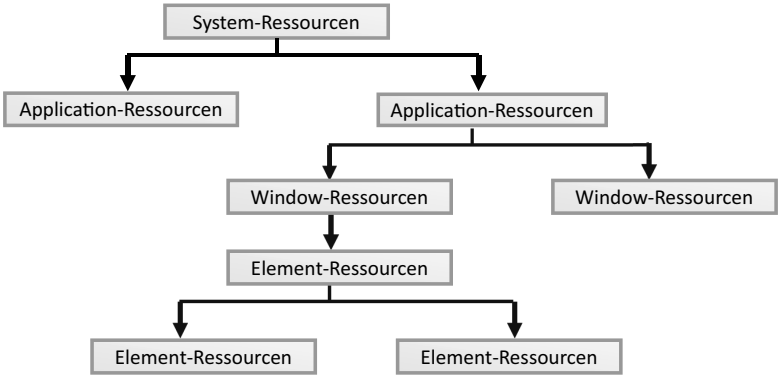

Bild 11.1 Hierarchie von Ressourcen

HINWEIS: Wird eine Ressource referenziert und damit auch gesucht, beginnt die Suche immer beim aktuellen Element. Wird die Ressource hier nicht gefunden, wird im übergeordneten Element (Container → Window → Application → System) gesucht. Damit ist auch klar, dass untergeordnete Elemente gleichnamige Ressourcen von übergeordneten Elementen überschreiben können. Innerhalb einer Hierarchieebene ist dies nicht möglich, da es sich in diesem Fall nicht um einen eindeutigen Schlüssel handelt.

Im XAML-Quellcode stellt sich die Definition von *Resources*-Abschnitten wie folgt dar:

Die Window-Ressourcen:

```
<Window>
  <Window.Resources>
    ...
  </Window.Resources>

  <StackPanel>
```

Die Ressourcen eines Containers:

```
    <StackPanel.Resources>
      ...
    </StackPanel.Resources>
...
    <Button>
```

Die Ressourcen eines Elements:

```
      <Button.Resources>
      ...
      </Button.Resources>
    </Button>
  </StackPanel>
</Window>
```

11.2.3 Wie definiere ich eine Ressource?

Haben Sie sich dafür entschieden, auf welcher Ebene Sie die Ressource definieren, können Sie zur Tat schreiten:

- Zunächst müssen Sie sich über die Art der Ressource im Klaren sein; diese bestimmt den Elementnamen (z. B. ein *ImageBrush*, mit dem ein Hintergrund festgelegt werden kann).
- Neben dieser Information müssen Sie sich noch für einen eindeutigen Schlüssel entscheiden, über den die Ressource angesprochen werden soll.
- Last, but not least, müssen Sie auch noch die eigentlichen Informationen zur Ressource (beim *ImageBrush* ist dies die *ImageSource*) festlegen.

Beispiel 11.3: Erzeugen und Verwenden einer Ressource auf Fensterebene

XAML

```
<Window x:Class="RessourcenSample.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:RessourcenSample"
        mc:Ignorable="d"
        Title="MainWindow" Height="250" Width="400">
```

```
  <Window.Resources>
    <ImageBrush x:Key="bck" ImageSource="back.jpg" />
  </Window.Resources>
```

Die Verwendung:

```
  <StackPanel Margin="0, 20, 0, 0">
    <Button>Hallo ohne Background</Button>
    <Button Background="{StaticResource bck}">Hallo mit Background</Button>
  </StackPanel>
</Window>
```

Ergebnis

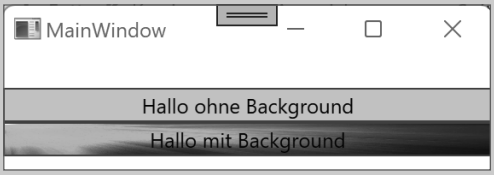

Bild 11.2
Button mit Ressource als Hintergrund

HINWEIS: Beim Einsatz von (statischen) Ressourcen ist es wichtig, dass diese **vor** der Verwendung definiert wurden, andernfalls kann die Ressource nicht gefunden werden. Zusätzlich müssen Sie auf die Schreibweise achten, hier wird die Groß-/Kleinschreibung berücksichtigt!

Alternativ können Sie bei der Verwendung der Ressource auch die folgende Syntax nutzen:

```
  <Button Height="50">
    <Button.Background>
      <StaticResource ResourceKey="bck" />
    </Button.Background>
    Hallo mit Background2
  </Button>
```

11.2.4 Statische und dynamische Ressourcen

Wie Sie im vorhergehenden Beispiel bereits gesehen haben, können Ressourcen statisch, d. h. unveränderlich, über den Schlüssel zugeordnet werden:

```
<StackPanel>
  <Button Background="{StaticResource bck}">Hallo mit Background</Button>
</StackPanel>
```

Voraussetzung ist das vorherige Definieren der Ressource. Spätere Änderungen an der Ressource werden nicht registriert und haben keine Auswirkung auf das verwendende Element.

Neben dieser Variante besteht auch die Möglichkeit, Ressourcen dynamisch festzulegen. In diesem Fall können Sie die Ressourcen zur Laufzeit zuordnen bzw. bereits vorhandene Ressourcen einfach austauschen.

Beispiel 11.4: Verwendung einer noch nicht definierten dynamischen Ressource.

XAML

Zunächst die Zuordnung von Eigenschaft und Ressource:

```
<Button Background="{DynamicResource bck2}"
        Click="Button_Click">Hallo</Button>
```

Da es die Ressource zu diesem Zeitpunkt noch nicht gibt, erscheint eine Warnung, die Sie aber ignorieren können.

C#

Mit dem Klick auf die Schaltfläche wollen wir eine Ressource zuordnen:

```
private void Button_Click(object sender, RoutedEventArgs e)
{
```

Neuen *ImageBrush* erzeugen:

```
    ImageBrush brush = new ImageBrush();
```

Eine Bitmap zuweisen:

```
    brush.ImageSource = new BitmapImage(new
            Uri("pack://application:,,,/Images/back2.jpg"));
```

Die Ressource erzeugen (Window-Ressource):

```
    Resources.Add("bck2", brush);
}
```

Ergebnis

Bild 11.3
Nach Klick hat auch der obere Button eine Ressource zugewiesen.

HINWEIS: Und warum verwenden wir nicht einfach immer dynamische Ressourcen? Da die Verwaltung dynamischer Ressourcen deutlich aufwendiger ist, würden hier unnötigerweise Systemressourcen verschwendet werden.

11.2.5 Wie werden Ressourcen adressiert?

Möchten Sie auf eingebundene Ressourcen (z. B. Grafiken) Ihrer Anwendung zugreifen, müssen Sie beispielsweise bei der Zuordnung von Grafiken per Code eine bestimmte Syntax einhalten, andernfalls wird die Ressource an der falschen Stelle gesucht und dann wohl auch nicht gefunden.

Die dazu erforderliche URI können Sie absolut, d. h. mit voller Pfadangabe, inklusive der aktuellen Assembly oder relativ zur aktuellen Assembly angeben.

Beispiel 11.5: Absolute Pfadangabe (Bild liegt in der Projekt-Root, Buildvorgang=*Resource*)

C#

```
Uri uri = new Uri("pack://application:,,,/Bild.jpg");
```

Beispiel 11.6: Absolute Pfadangabe (das Bild liegt im Unterverzeichnis *Images* des aktuellen Projekts, Buildvorgang=*Resource*)

C#

```
Uri uri = new Uri("pack://application:,,,/Images/Bild.jpg");
```

Beispiel 11.7: Relative Pfadangabe (das Bild liegt im gleichen Verzeichnis wie die Assembly)

C#

```
Uri uri = new Uri(@".\Back3.jpg", UriKind.Relative);
```

Beispiel 11.8: Relative Pfadangabe (das Bild liegt im relativen Unterverzeichnis *Images* der Assembly)

C#

```
Uri uri = new Uri(@".\Images\Back3.jpg", UriKind.Relative)
```

HINWEIS: Vielleicht ist Ihnen aufgefallen, dass in WPF noch immer viele *using*-Anweisungen zum Import von Namespaces gebraucht werden. Dies liegt daran, dass die implizite globale Verwendung in WPF-Projekten nicht standardmäßig aktiviert ist. Fügen Sie einfach *<ImplicitUsings>enable</ImplicitUsings>* in die Projektdatei ein.

Neben den gezeigten Möglichkeiten können Sie unter anderem auch auf eingebundene Assemblies zugreifen.

11.2.6 Systemressourcen einbinden

Neben den in Ihrer Anwendung definierten Ressourcen können Sie auch Systemressourcen verwenden. Die Einbindung erfolgt entweder statisch oder dynamisch. Im ersten Fall reagiert die Anwendung jedoch nicht auf aktuelle Änderungen an den Systemeinstellungen.

Beispiel 11.9: Einbinden von Systemressourcen

XAML

Anzeige der *VirtualScreenWidth* in einem *Label* (Ressourcenabfrage):

```
<Label Content="{StaticResource {x:Static
                SystemParameters.VirtualScreenWidthKey}}"/>
```

Anzeige, ob User-Interface-Effekte aktiviert sind (Wertabfrage):

```
<CheckBox IsChecked="{x:Static SystemParameters.UIEffects}"
                    Content="UIEffects" />
```

Ergebnis

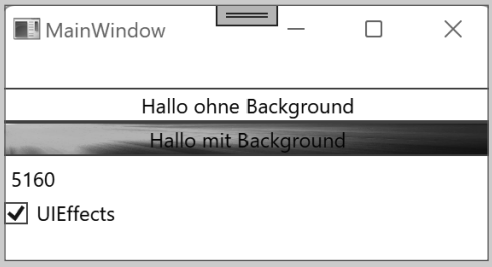

Bild 11.4
Anzeige von Systemressourcen

11.3 Das WPF-Ereignismodell

Nachdem wir in den bisherigen Beispielen recht sparsam mit der Verwendung von Eventhandlern umgegangen sind, wollen wir uns jetzt diesem Thema etwas intensiver widmen.

11.3.1 Einführung

Sicher haben Sie auch schon mehr oder weniger unbewusst von den Ereignissen in WPF Gebrauch gemacht. Nach dem Klick, z. B. auf eine Schaltfläche, wird der entsprechende Ereignishandler von Visual Studio bereitgestellt.

Beispiel 11.10: *Button* mit zugehöriger Ereignisbehandlung in C#.

XAML

```
<StackPanel>
    <Button Content="Klick mich!" Click="Button_Click"/>
</StackPanel>
```

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    MessageBox.Show("Hallo");
}
```

So weit, so gut – das kennen Sie sicher auch schon von der Programmierung mit Win32 oder Windows Forms. Doch was passiert, wenn Sie in WPF eine Schaltfläche aus einzelnen Elementen „zusammenbasteln“?

Beispiel 11.11: Button mit Text und Grafik

XAML

```
<Button Click="Button_Click">
  <StackPanel Orientation="Horizontal" Margin="10">
    <Image Source="Images/Flash.png" Width= "56" Height="46" />
    <TextBlock VerticalAlignment="Center">Klick mich!</TextBlock>
  </StackPanel>
</Button>
```

Ergebnis

Bild 11.5
Button mit eingebettetem Image und TextBlock

Rein intuitiv haben Sie sicher auch das *Click*-Ereignis dem Button zugeordnet. Doch warum sollte das eigentlich funktionieren? Was passiert, wenn der Nutzer auf das *Image* oder den *TextBlock* klickt, zusätzlich befindet sich „darunter“ ja noch das *StackPanel*.

Das Zauberwort heißt hier „Routed Events“, ein Verfahren, um auftretende Ereignisse in der Element-Hierarchie weiterzugeben.

11.3.2 Routed Events

WPF unterscheidet bei den Routed Events zwei verschiedene Varianten, die Sie als Programmierer auch auseinanderhalten sollten:

- **Tunneling Events:** Ausgehend vom Wurzelelement (*Window/Page*) werden die Ereignisse bis zum auslösenden Element weitergereicht. Diese Events werden **vor** den zugehörigen Bubbling Events ausgelöst.

- **Bubbling Events:** Ausgehend vom aktivierten Element werden die Ereignisse zum jeweils übergeordneten Element weitergereicht, d. h. im Endeffekt bis zum *Window* oder zur *Page*.

HINWEIS: Tunneling Events sind durch das einleitende „Preview…“ gekennzeichnet, Bubbling Events verzichten darauf.

Beispiel 11.12: Ablauf der Ereigniskette für einen Mausklick mit der linken Taste

XAML

```
<Window x:Class="EreignisseSample.MainWindow"
...
    Title="MainWindow" MouseLeftButtonDown="Window_MouseLeftButtonDown"
           PreviewMouseLeftButtonDown="Window_PreviewMouseLeftButtonDown">
  <StackPanel>
   <Button MouseLeftButtonDown="Button_MouseLeftButtonDown"
           PreviewMouseLeftButtonDown="Button_PreviewMouseLeftButtonDown">
     <StackPanel Orientation="Horizontal" Margin="10"
       MouseLeftButtonDown="StackPanel_MouseLeftButtonDown"
       PreviewMouseLeftButtonDown="StackPanel_PreviewMouseLeftButtonDown">
       <Image Source="Images/Flash.png" Width="56" Height="46"
          MouseLeftButtonDown="Image_MouseLeftButtonDown"
          PreviewMouseLeftButtonDown="Image_PreviewMouseLeftButtonDown" />
       <TextBlock VerticalAlignment="Center"
         MouseLeftButtonDown="TextBlock_MouseLeftButtonDown"
         PreviewMouseLeftButtonDown="TextBlock_PreviewMouseLeftButtonDown">
             Klick mich!
       </TextBlock>
     </StackPanel>
   </Button>
   <ListBox Name="listBox1" />
  </StackPanel>
</Window>
```

Ergebnis

Die Ereigniskette nach einem Klick auf das *Image:*

1. Window: *PreviewMouseLeftButtonDown*
2. Button: *PreviewMouseLeftButtonDown*
3. StackPanel: *PreviewMouseLeftButtonDown*
4. Image: *PreviewMouseLeftButtonDown*
5. Image: *MouseLeftButtonDown*
6. StackPanel: *MouseLeftButtonDown*

Wie Sie sehen, wird zunächst die komplette Tunneling-Ereigniskette durchlaufen, nachfolgend die Bubbeling-Events[1].

Im Normalfall werden Sie wohl nur Bubbling-Events verwenden. Tunneling-Events nutzen Sie beispielsweise, um Ereignisse bzw. deren Weiterleitung zu blockieren.

[1] Der Button löst ein *Click*-Ereignis aus. Die dazu nötige Logik verhindert das Auslösen entsprechender *MouseLeftButtonDown*-Ereignisse, deshalb ist hier die Ereigniskette zu Ende.

Beispiel 11.13: Das Weiterleiten der Ereignisse verhindern.

C#

Wir werten gleich das erste Ereignis aus (das Tunneling-Event beginnt beim Wurzelobjekt, d. h. dem Window):

```
private void Window_PreviewMouseLeftButtonDown(object sender,
             MouseButtonEventArgs e)
{
```

Ereignis behandelt

```
    e.Handled = true;
    listBox1.Items.Add(nameof(Window_PreviewMouseLeftButtonDown));
}
```

Beispiel 11.14: Das auslösende Element bestimmen

C#

Den Ursprung für ein Ereignis können Sie mit dem *...EventArgs.Source*-Parameter bestimmen:

```
private void Window_PreviewMouseLeftButtonDown(object sender,
             MouseButtonEventArgs e)
{
    MessageBox.Show(e.Source.ToString());
    listBox1.Items.Add(nameof(Window_PreviewMouseLeftButtonDown));
}
```

Ergebnis

Beim Klick auf das Image wird auch dieses als *Source* übermittelt.

Bild 11.6
Klick auf das Image

11.3.3 Direkte Events

Auch diese Form der Ereignisse ist nach wie vor präsent. Hierbei handelt es sich um die ganz normalen .NET-Ereignisse, wie Sie auch in Windows-Forms-Anwendungen auftreten.

Direkte Events werden eingesetzt, wenn die Verwendung von Bubbling oder Tunneling keinen Sinn macht, beispielsweise beim *MouseLeave*-Ereignis, das sehr objektbezogen ausgelöst wird.

Sie erkennen diese Ereignisse an einem fehlenden *Preview...*-Pendant.

11.4 Verwendung von Commands

Im Zusammenhang mit der Entwicklung von Anwendungen ist es Ihnen sicher auch schon passiert, dass Sie eine Menüfunktion zum x-ten Male neu programmiert haben. Das Gleiche trifft sicher ebenfalls auf die Toolbar zu. Das Prozedere ist doch immer gleich:

1. Methode mit der eigentlichen Logik erstellen
2. Menüpunkt erstellen
3. Menüpunkt Tastenkürzel zuweisen, Tastaturabfrage implementieren
4. Menüpunkt Ereignismethode zuweisen und Methode von 1. aufrufen
5. Toolbar-Button bereitstellen
6. Toolbar-Button Ereignismethode zuweisen und Methode von 1. aufrufen
7. Logik für das Sperren von 2. und 5. erstellen, wenn die Funktion nicht zur Verfügung steht

11.4.1 Einführung zu Commands

In WPF-Anwendungen können Sie diesen Aufwand wesentlich verringern, indem Sie entweder die von WPF vordefinierten Commands verwenden, oder indem Sie selbst eigene Commands implementieren.

Die neue Vorgehensweise (vordefinierte Commands, z. B. Einfügen in die Zwischenablage) im Vergleich zum bisherigen Vorgehen:

1. Entfällt, da für viele Controls bereits implementiert
2. **Menüpunkt erstellen und Command zuweisen**
3. Entfällt, da per Command automatisch zugewiesen
4. Entfällt, da per Command automatisch zugewiesen
5. **Toolbar-Button bereitstellen und Command zuweisen**
6. Entfällt, da per Command automatisch zugewiesen
7. Entfällt, da Command diese Logik bereitstellt

Das sieht doch schon wesentlich freundlicher aus als die mühsame und fehleranfällige erste Variante. Ähnlich einfach gestaltet sich die Wiederverwendung von selbst erstellten Kommandos, wenn Sie diese bereits einmal erstellt haben.

11.4.2 Verwendung vordefinierter Commands

Statt vieler Worte möchten wir Ihnen zunächst an einem Beispiel die Vorgehensweise bei der Verwendung von Commands demonstrieren.

Beispiel 11.15: Programmieren der Funktionen „Kopieren und Einfügen“ für ein Formular mit zwei TextBoxen

XAML

Der XAML-Code der Oberfläche:

```
<Window x:Class="CommandsSample.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:CommandsSample"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="400">
    <DockPanel>
```

Die Menüdefinition:

```
       <Menu DockPanel.Dock="Top" Height="22" Name="menu1">
           <MenuItem Header="_Bearbeiten">
```

Hier werden die beiden Menüpunkte „Kopieren“ und „Einfügen“ definiert:

```
               <MenuItem Command="ApplicationCommands.Copy"/>
               <MenuItem Command="ApplicationCommands.Paste"/>
```

Sie können auf die Angabe des Headers sowie der Tastenkürzel verzichten, diese stellt die obige *Command*-Definition automatisch zur Verfügung.

```
           </MenuItem>
       </Menu>
       <ToolBarTray DockPanel.Dock="Top">
           <ToolBar>
```

Hier findet sich bereits die Definition für die *ToolBar*-Buttons. Auch in diesem Fall genügt die Zuordnung der *Command*-Eigenschaft:

```
               <Button Width="30" Height="30" Command="ApplicationCommands.Copy">
                   <Image Source="Images/editcopy.png"/>
               </Button>
               <Button Width="30" Height="30" Command="ApplicationCommands.Paste">
                   <Image Source="Images/editpaste.png"/>
```

```
            </Button>
        </ToolBar>
    </ToolBarTray>
    <StackPanel>
```

Hier noch die beiden *TextBox*en, für die die Funktionen implementiert werden:

```
        <TextBox Name="txt1" Height="100">TextBox1 Bla Bla</TextBox>
        <TextBox Name="txt2" Height="100">TextBox2 Bla Bla</TextBox>
    </StackPanel>
</DockPanel>
</Window>
```

Ergebnis

Und wo bleibt der Code? Kurze Antwort: Ihr Programm ist an dieser Stelle bereits fertig. Nach dem Start können Sie sich von der Funktionstüchtigkeit überzeugen.

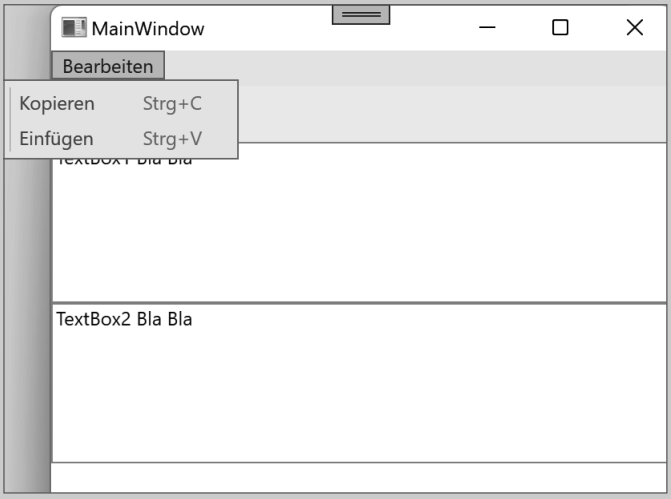

Bild 11.7 Fertige Menübefehle dank Commands

Beide Menüpunkte verfügen über eine Beschriftung und ein Tastenkürzel. Befindet sich Text in der Zwischenablage, ist das Menü *Einfügen* aktiviert, andernfalls ist der Menüpunkt gesperrt. Der Menüpunkt *Kopieren* ist nur aktiv, wenn Sie in einer der *TextBox*en Text markieren und die *TextBox* den Eingabefokus besitzt.

Nicht schlecht, wenn Sie dies mit der konventionellen Variante vergleichen, bei der Sie sicher wesentlich mehr Code produziert hätten.[2]

[2] Geben Sie trotzdem einen Wert an, überschreibt dies die vom Command bereitgestellten Werte.

11.4.3 Das Ziel des Commands

Doch gerade bei obigem Beispiel wird sicher bei manchem die Frage aufkommen, wohin denn eigentlich der Text eingefügt wird, haben wir doch zwei Textfelder. Die Frage ist sicher berechtigt, aber im obigen Beispiel noch recht einfach lösbar: Ziel des jeweils gewählten Commands ist standardmäßig immer das gerade aktive Control, d. h. die *TextBox*, die den Eingabefokus besitzt.

Schwieriger wird es, wenn die Zwischenablageinhalte gezielt in ein bestimmtes Control eingefügt werden sollen. In diesem Fall müssen Sie neben der *Command*- auch die *CommandTarget*-Eigenschaft definieren. Allerdings genügt in diesem Fall nicht die reine Angabe des Elementnamens, sondern Sie müssen die Binding-Syntax verwenden.

Beispiel 11.16: Zwischenablageinhalte sollen immer in *TextBox2* eingefügt werden

XAML

Wir fügen dem betreffenden Menüpunkt ein *CommandTarget* hinzu:

```
<MenuItem Header="_Bearbeiten">
  <MenuItem Command="ApplicationCommands.Copy"/>
  <MenuItem Command="ApplicationCommands.Paste"
            CommandTarget="{Binding ElementName=txt2}"/>
</MenuItem>
...
```

Ergebnis

Starten Sie jetzt das Programm, ist es egal, welches Control den Fokus besitzt. Drücken Sie die Tastenkombination **Strg**+**V** oder wählen Sie den entsprechenden Menüpunkt, wird der Text immer in die zweite *TextBox* eingefügt.

HINWEIS: Diese Vorgehensweise müssen Sie auch wählen, wenn Sie Commands von einzelnen Schaltflächen aus starten wollen. Da diese den Fokus erhalten können, würde nie klar sein, welches Ziel die Aktion haben soll.

Beispiel 11.17: „Freistehender" *Button* für das Einfügen des Zwischenablageinhalts

XAML

```
<Button Command="ApplicationCommands.Paste"
        CommandTarget="{Binding ElementName=txt2}">
    Paste (TextBox 2)
</Button>
```

11.4.4 Vordefinierte Commands

WPF bietet bereits „ab Werk“ eine ganze Reihe von Commands, die in der täglichen Programmierpraxis immer wieder anfallen. Diese Commands gliedern sich in die folgenden Gruppen:

- *ApplicationCommands* (z. B. *Copy*, *Cut*, *Paste*, *Help*, *New*, *Print*, *Save*, *Stop*, *Undo*)
- *ComponentCommands* (z. B. *ExtendSelectionLeft*, *MoveLeft*)
- *NavigationCommands* (z. B. *FirstPage*, *GotoPage*, *Refresh*, *Search*)
- *MediaCommands* (z. B. *Play*, *Pause*, *FastForward*, *IncreaseVolume*)
- *EditingCommands* (z. B. *Delete*, *MoveUpByLine*, *ToggleBold*)

Ob und, wenn ja, welche Commands implementiert sind, müssen Sie von Fall zu Fall ausprobieren. Sehr umfassend ist z. B. die Unterstützung bei *TextBox* und *RichTextBox* sowie beim *MediaElement* (siehe Onlinekapitel).

11.4.5 Commands an Ereignismethoden binden

Wie schon erwähnt, implementiert nicht jedes Control alle verfügbaren Commands. So steht zwar ein *ApplicationCommand.Open* (d. h. Beschriftung und Tastaturkürzel) zur Verfügung, im Normalfall passiert allerdings nichts, da kein Control eine entsprechende Logik implementiert hat, was bei derart komplexen Abläufen sicher auch nicht möglich ist.

Aus diesem Grund besteht die Möglichkeit, einem Command eine entsprechende Ereignisbehandlung per *CommandBinding* zuzuordnen. Zwei Ereignisse sind hier von zentraler Bedeutung:

- *Execute* (die eigentlich auszuführende Logik) und
- *CanExecute* (eine Abfrage, ob die Funktion überhaupt zur Verfügung steht)

Beispiel 11.18: Implementieren des *ApplicationCommand.Open*

XAML

Nutzen Sie für die Zuordnung der beiden Ereignismethoden am besten das Wurzelelement (*Window*):

```
<Window x:Class="CommandSample.MainWindow"
...
    Title="MainWindow" Height="350" Width="400">
    <Window.CommandBindings>
        <CommandBinding Command="ApplicationCommands.Open"
                        Executed="OpenCmdExecuted"
                        CanExecute="OpenCmdCanExecute"/>
    </Window.CommandBindings>
```

Vergessen Sie nicht, noch einen zusätzlichen Menüeintrag zu erstellen:

```
    <MenuItem Header="_Datei">
        <MenuItem Command="ApplicationCommands.Open"/>
    </MenuItem>
```

C#

Die beiden zugehörigen Ereignismethoden aus der Klassendefinition des Windows:

```
void OpenCmdExecuted(object target, ExecutedRoutedEventArgs e)
{
    MessageBox.Show("Was soll ich öffnen?");
    // hier steht die eigentliche Logik
}
```

Nur wenn die erste *TextBox* auch leer ist, ist das Öffnen neuer Dateien möglich:

```
void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
    if (string.IsNullOrEmpty(txt1.Text))
    {
       e.CanExecute = true;
    }
    else
    {
       e.CanExecute = false;
    }
}
```

Ergebnis

Ein Test zur Laufzeit zeigt, dass die entsprechende Schaltfläche nur freigegeben ist, wenn die *TextBox* leer ist.

Klicken Sie auf den obigen Menüpunkt oder verwenden Sie die Tastenkombination **Strg**+**O**, wird unsere MessageBox aus der Methode *OpenCmdExecuted* ausgeführt.

HINWEIS: Selbstverständlich können Sie *CommandBinding* auch per Code realisieren, wie das folgende Beispiel zeigt.

Beispiel 11.19: Ereignismethode per Code zuweisen

C#

```
public MainWindow()
{
    InitializeComponent();
    CommandBinding cmdOpen =
        new CommandBinding(ApplicationCommands.Open);
    cmdOpen.Executed += new ExecutedRoutedEventHandler(OpenCmdExecuted);
}
```

11.4.6 Wie kann ich ein Command per Code auslösen?

Neben der bereits beschriebenen Variante, Commands per Zuordnung im XAML-Code auszulösen, besteht auch die Möglichkeit, diese direkt mit der *Execute*-Methode aufzurufen.

Beispiel 11.20: Direkter Aufruf eines Commands in einer Ereignismethode

C#

```
private void Button_Click(object sender, RoutedEventArgs e)
{
    ApplicationCommands.Paste.Execute(null, txt2);
}
```

Im zweiten Parameter übergeben Sie das *CommandTarget*.

11.4.7 Command-Ausführung verhindern

Nicht immer und zu jeder Zeit können Kommandos einfach ausgeführt werden. Unter bestimmten Bedingungen steht eine Funktion nicht zur Verfügung, in diesem Fall sollen die Menüpunkte/Schaltflächen abgeblendet sein, um den Anwender nicht unnötig zu verwirren.

Die Lösungsmöglichkeit haben wir bereits beim Zuordnen von Ereignismethoden gezeigt. Im … *CanExecute*-Ereignis wird mit dem Parameter *e.CanExecute* entschieden, ob eine Aktion ausführbar ist oder nicht.

Beispiel 11.21: Command-Ausführung verhindern

C#

```
private void OpenCmdCanExecute(object sender, CanExecuteRoutedEventArgs e)
{
       e.CanExecute = false;
}
```

11.5 Das WPF-Style-System

Mit den WPF-Styles kommen wir jetzt zu einem Thema, das sicher von ganz zentraler Bedeutung für die oberflächenorientierten WPF-Anwendungen ist und auch einen der wesentlichsten Unterschiede zu den konventionellen Windows-Anwendungen darstellt.

11.5.1 Übersicht

Doch worum geht es eigentlich? Sicher haben Sie nach der Lektüre der beiden vorhergehenden Kapitel festgestellt, dass WPF-Controls mit Bergen von Eigenschaften ausgestattet sind, die es möglich machen, fast alle Aspekte der Darstellung zu beeinflussen.

Doch gerade für aufwendige Oberflächen ergeben sich einige Fragen:

- Wie kann ich mehr als einem Control ein spezifisches Aussehen zuweisen?
- Wie kann ich das Aussehen unter bestimmten Bedingungen ändern?
- Wie kann ich das grundsätzliche Aussehen eines Controls komplett ändern?
- Wie kann ich einfache Animationen realisieren?

Für alle diese Fragen bietet WPF eine Antwort:

- Verwendung von benannten oder Typ-Styles
- Verwendung von Triggern
- Verwendung von Templates
- Verwendung von StoryBoards

HINWEIS: Im Rahmen dieses Abschnitts werden wir die oben genannten Themen nur recht oberflächlich streifen, da dies eigentlich ein Hauptarbeitsgebiet für den Designer und nicht für den Programmierer ist. Außerdem ist Visual Studio für einige der obigen Aufgaben das falsche Tool. Für WPF-Designer gibt es das Tool *Blend for Visual Studio*, mit dem man komfortabel XAML-Dateien erstellen kann.

11.5.2 Benannte Styles

In den bisherigen Beispielen haben wir die Eigenschaften jedes Controls einzeln gesetzt, was bei größeren Ansammlungen recht schnell zum Geduldspiel ausarten kann. Denken Sie nur an unseren Taschenrechner aus dem WPF-Einführungskapitel, wo ca. zwanzig einzelne Tasten zu konfigurieren sind (Schrift, Randabstände, Farben etc.). Viel schöner wäre doch hier eine zentrale Vorschrift, wie ein derartiger Button auszusehen hat.

Genau diese Aufgabe übernimmt ein *benannter Style*. Dieser wird einmal definiert und kann dann per Key den einzelnen Elementen zugewiesen werden.

Beispiel 11.22: Alle Schaltflächen für den Taschenrechner sollen einen Randabstand von 2, eine fette Schrift und eine gelbe Hintergrundfarbe bekommen.

XAML

```
< <Window x:Class="StyleSample.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:StyleSample"
    mc:Ignorable="d"
    Title="MainWindow" Height="250" Width="400">
```

Jetzt wird auch klar, warum wir uns in diesem Kapitel bereits mit Ressourcen beschäftigt haben. Hoffentlich haben Sie diesen Abschnitt nicht gelangweilt überblättert!

```
   <Window.Resources>
```

Wir definieren einen neuen Style und legen dessen *Key* fest (über diesen können wir später auf den Style zugreifen bzw. auf diesen verweisen):

```
<Style x:Key="myBtnStyle">
```

Innerhalb des *Style*-Elements können Sie per *Setter* die gewünschten Eigenschaften beeinflussen:

```
    <Setter Property="Control.Margin" Value="2" />
    <Setter Property="Control.Background" Value="Yellow" />
    <Setter Property="Control.FontWeight" Value="UltraBold" />
```

Legen Sie jeweils *Property* und *Value* fest.

```
  </Style>
</Window.Resources>
```

Und jetzt kommt unser neuer Style zum Einsatz (setzen Sie diesen nur bei den Zifferntasten):

```
  <UniformGrid Name="uniformGrid1" Columns="5" Rows="4" Grid.Row="1">
    <Button Name="button1" Style="{StaticResource myBtnStyle}">7</Button>
    <Button Name="button2" Style="{StaticResource myBtnStyle}">8</Button>
    <Button Name="button3" Style="{StaticResource myBtnStyle}">9</Button>
...
```

Ergebnis

Schon im Designer dürfte sich jetzt etwas getan haben. Bild 11.8 zeigt die App während der Laufzeit.

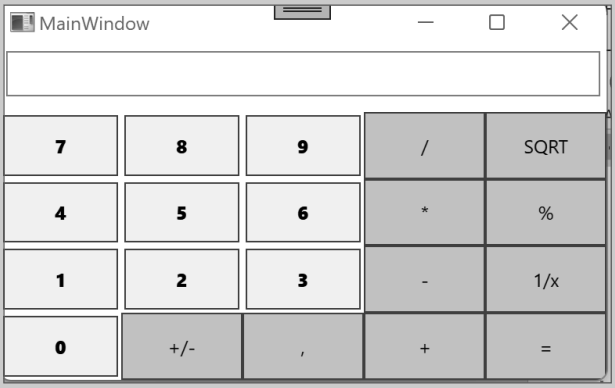

Bild 11.8 Button mit Zahlen mit einem einheitlichen Style

Was passiert eigentlich, wenn wir den Style einer *TextBox* zuweisen? Probieren Sie es ruhig aus, es kann nichts passieren. Vielleicht sind Sie überrascht, aber auch die *TextBox* wird nach dieser Aktion im gelben Outfit erscheinen und eine fette Schrift anzeigen.

Warum dies so ist? Ganz einfach, auch die *TextBox* verfügt über die aufgeführten Eigenschaften und übernimmt diese automatisch vom Style.

11.5.3 Typ-Styles

Ein „fauler“ Programmierer (sind wir das nicht alle?) wird immer einen einfacheren Weg suchen, und so ist es sicher mühsam, den Style jedem einzelnen Button explizit zuzuweisen, zumal die Syntax auch recht umfangreich ist. Aus diesem Grund gibt es noch eine zweite Art von Styles, die nicht über einen Key, sondern indirekt über den Klassennamen zugeordnet werden.

Der Vorteil: Alle WPF-Elemente, die Instanzen dieser Klasse sind, übernehmen automatisch diesen Style, ohne dass dies explizit angegeben werden muss.

Beispiel 11.23: *(Fortsetzung)* Mit Ausnahme der Zifferntaste sollen alle Tasten einen grünen Hintergrund und eine weiße Schrift bekommen.

XAML

```
<Window.Resources>
```

Hier der benannte Style für die Zifferntasten:

```
   <Style x:Key="myBtnStyle">
...
   </Style>
```

Und hier der Default-Style für alle Elemente vom Typ *Button*:

```
   <Style TargetType="{x:Type Button}">
     <Setter Property="Margin" Value="2" />
     <Setter Property="Background" Value="Green" />
     <Setter Property="FontWeight" Value="UltraBold" />
     <Setter Property="Foreground" Value="White" />
   </Style>
```

Ergebnis

Ohne weitere Änderungen im XAML-Code dürfte sich jetzt bereits folgender Anblick im Designer bieten (Bild 11.9).

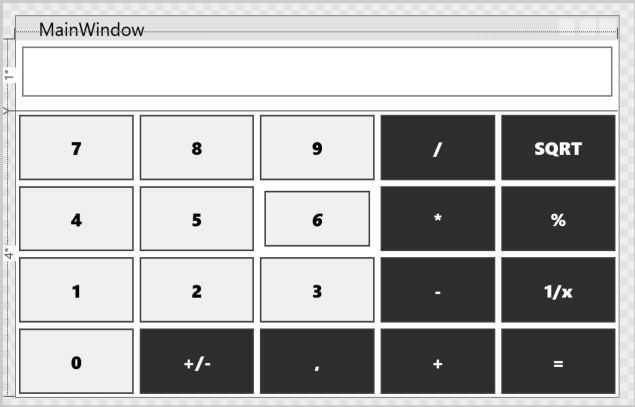

Bild 11.9 Buttons mit zwei unterschiedlichen Style-Varianten

HINWEIS: Hätten Sie die Style-Definition in der Datei *App.xaml* eingefügt, würde es sich um einen anwendungsweiten Style handeln – alle Windows bzw. die enthaltenen Schaltflächen würden dieses Outfit bekommen.

Doch was ist, wenn sich auf Ihrem Formular zum Beispiel ein paar *ToggleButton*-Controls befinden? Diese sind von obiger Style-Definition nicht betroffen, handelt es sich doch um eine andere Klasse.

11.5.4 Styles anpassen und vererben

Es gibt immer wieder Ausnahmen von der Regel, und so kommen Sie meist nicht darum herum, den Style einzelner Controls speziell anzupassen. Drei Varianten bieten sich dazu an:

- Sie überschreiben einzelne Attribute direkt im Element.
- Sie ersetzen den Style auf einer niedrigeren Ebene (statt *Application* z. B. in einem *StackPanel*).
- Sie vererben den Style und passen ihn unter neuem Namen an.

Styles anpassen (überschreiben)

Das Überschreiben von Styles ist eigentlich ganz intuitiv möglich. Geben Sie den Key des Styles an oder nutzen Sie einen Typ-Style wie bisher. Gleichzeitig erweitern Sie die Attributliste des betreffenden Elements, um die gewünschten Änderungen vorzunehmen.

Beispiel 11.24: *(Fortsetzung)* Die Taste „0“ soll rot hinterlegt werden (der Style gibt gelb vor).

XAML

```
<Button Name="button16" Style="{StaticResource myBtnStyle}"
         Background="Red" >0</Button>
```

Style ersetzen

Fällt ein komplettes Formular aus dem Rahmen oder möchten Sie einzelne Elemente einer Gruppe (*StackPanel*, *Grid* etc.) mit einem angepassten Style versehen, können Sie auch den zentral gültigen Style ersetzen. Definieren Sie dazu einen neuen *Resources*-Abschnitt und fügen Sie die neue Style-Definition in diesen ein.

Beispiel 11.25: *(Fortsetzung)* Ersetzen des zentralen Button-Styles

XAML

```
<Window x:Class="StyleSample.MainWindow"
...
```

Hier die übergreifende Definition:

```
 <Window.Resources>
   <Style TargetType="{x:Type Button}">
```

```
...
   </Style>
 </Window.Resources>
...
   <UniformGrid Name="uniformGrid1" Columns="5" Rows="4" Grid.Row="1">
```

Hier wird der Style **ersetzt**, d. h. alle obigen Einstellungen gehen verloren:

```
    <UniformGrid.Resources>
      <Style TargetType="{x:Type Button}">
        <Setter Property="Margin" Value="2" />
        <Setter Property="Background" Value="Blue" />
        <Setter Property="FontWeight" Value="UltraBold" />
        <Setter Property="Foreground" Value="White" />
      </Style>
    </UniformGrid.Resources>
    ...
```

HINWEIS: Elemente, die sich in der Hierarchie oberhalb des *UniformGrid*s befinden, sind nicht von dieser Anpassung betroffen, hier gilt wieder die zentrale Version.

Styles vererben

Wer schreibfaul ist und beispielsweise nicht den kompletten Style austauschen will, kann diesen auch einfach vererben. Dazu wird in die Definition des Styles das Attribut *BasedOn* aufgenommen, das per Binding auf den Basisstyle verweist.

Ob Sie in diesem neuen Style bestehende Eigenschaften überschreiben (einfach erneut definieren) oder neue Eigenschaften hinzufügen, bleibt Ihnen überlassen.

Beispiel 11.26: Vererben eines Styles

XAML

```
   <Window.Resources>
```

Das Original:

```
   <Style x:Key="myBtnStyle">
     <Setter Property="Control.Margin" Value="2" />
     <Setter Property="Control.Background" Value="Yellow" />
     <Setter Property="Control.FontWeight" Value="UltraBold" />
   </Style>
```

Der „Erbe“ mit geringfügiger Änderung:

```
   <Style x:Key="myBtnStyle2" BasedOn="{StaticResource myBtnStyle}">
```

Hier wird eine Eigenschaft überschrieben:

```
     <Setter Property="Control.Margin" Value="5" />
```

Hier wird eine neue Eigenschaft gesetzt:

```
    <Setter Property="Control.FontStyle" Value="Italic" />
  </Style>
</Window.Resources>
```

Ergebnis

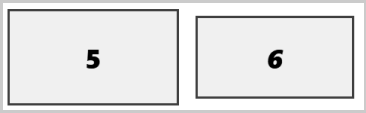

Bild 11.10
Links der Basis-Style, rechts der neue Style

Styleänderung per Code

Ihr Programm ist nicht darauf angewiesen, immer den gleichen Style zu verwenden. So ist es problemlos möglich, auch zur Laufzeit einen Style per Code neu zu setzen (z. B. unter bestimmten Bedingungen).

Beispiel 11.27: Der Style von *button10* wird geändert

C#

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
    // Ziffer 3
    button13.Style = (Style)FindResource("myBtnStyle2");
}
```

HINWEIS: Den Style bzw. dessen Instanz finden Sie mit der Methode *FindResource* über dessen Key.

HINWEIS: Bevor Sie sich weiter in diese Thematik vertiefen, werfen Sie zunächst einen Blick auf den folgenden Abschnitt; vielleicht können Sie damit bestimmte Aufgaben eleganter lösen.

11.6 Verwenden von Triggern

In unseren bisherigen Experimenten waren die vom Style vorgenommenen Änderungen immer statischer Natur, d. h. einmal gesetzt, blieb die Optik immer gleich. Doch dies kann sicher nicht der Weisheit letzter Schluss sein.

Wenn jetzt in Ihnen der Programmierer wieder durchkommt und Sie an C# und Ereignisprozeduren denken, vergessen Sie es gleich wieder. Für (fast) alle Aufgabenstellungen ist auch hier XAML die beste Lösung.

Für die Reaktion auf Eigenschaftsänderungen, Ereignisse, Datenänderungen etc. können Sie in WPF-Anwendungen sogenannte Trigger verwenden und damit zum Beispiel den Style ändern. Dabei sind Sie nicht auf einen einzelnen Trigger angewiesen, sondern Sie können der Trigger-Collection auch mehrere Ereignisse mit unterschiedlichen Bedingungen zuweisen.

Im Folgenden wollen wir uns die verschiedenen Triggerarten näher ansehen.

11.6.1 Eigenschaften-Trigger (Property Triggers)

Sicher kennen Sie auch das eine oder andere Programm, das exzessiv Gebrauch von diversen optischen Spielereien macht. Wird beispielsweise mit dem Mauscursor auf ein Control gezeigt, ändert sich dessen Rahmen oder die Hintergrundfarbe. Gleiches gilt für Eingabefelder, die den Fokus erhalten, etc. In all diesen Fällen ändern sich Eigenschaften (*IsMouseOver*, *IsFocused*), auf die Sie bei der konventionellen Programmierung mit Ereignismethoden reagieren können. Mit Eigenschaften-Triggern können Sie Ihren C#-Quellcode von derartigem Ballast befreien und direkt per XAML-Code Änderungen am Control vornehmen.

Beispiel 11.28: Eine *TextBox* soll auf *IsMouseOver* und *IsFocused* mit Farbänderungen reagieren.

XAML

```
<Window x:Class="TriggerSample.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:TriggerSample"
        mc:Ignorable="d"
        Title="MainWindow" Height="300" Width="300">

Einen Style für die TextBox erzeugen:

    <Window.Resources>
        <Style x:Key="myStyle" TargetType="{x:Type TextBox}">

Der Außenabstand soll immer 2 betragen:

            <Setter Property="Margin" Value="2" />

Hier werden die Trigger definiert:

            <Style.Triggers>

Unter der Bedingung ...

                <Trigger Property="IsMouseOver" Value="True">

... wird die folgende Eigenschaft gesetzt:

                    <Setter Property="Background" Value="Yellow" />
                </Trigger>
```

Unter der Bedingung ...

```
            <Trigger Property="IsFocused" Value="True">
```

... werden die folgenden Eigenschaften gesetzt:

```
                <Setter Property="Background" Value="Blue" />
                <Setter Property="Foreground" Value="White" />
            </Trigger>
        </Style.Triggers>
    </Style>
</Window.Resources>
<StackPanel>
```

Hier verwenden wir den Style:

```
    <TextBox Style="{StaticResource myStyle}" >Hallo</TextBox>
    <TextBox Style="{StaticResource myStyle}" >Hallo</TextBox>
    <TextBox Style="{StaticResource myStyle}" >Hallo</TextBox>
</StackPanel>
</Window>
```

Änderungen, die durch einen Trigger vorgenommen wurden, werden automatisch wieder rückgängig gemacht, wenn die Bedingung nicht mehr eingehalten wird (automatisches Wiederherstellen des Standardwertes).

Ergebnis

Bild 11.11 zeigt die Laufzeitansicht. Die erste TextBox hat den Eingabefokus, die zweite erfüllt die Bedingung *IsMouseOver=True* und die dritte TextBox ist im Standardzustand.

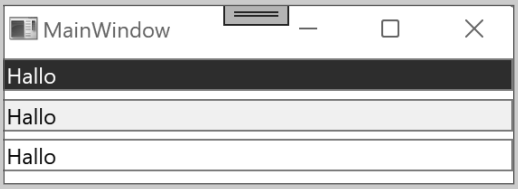

Bild 11.11
Trigger im Einsatz

Doch was ist, wenn Sie mehr als eine Bedingung benötigen? Auch das ist kein Problem, in diesem Fall erzeugen Sie einfach einen „multi-condition property trigger“.

Beispiel 11.29: Der *TextBox*-Hintergrund soll rot werden, wenn die *TextBox* den Eingabefokus besitzt und wenn kein Text enthalten ist.

XAML

```
    <Window.Resources>
        <Style x:Key="myStyle" TargetType="{x:Type TextBox}">
...
```

Einen Multi-Condition-Trigger definieren:

```
            <MultiTrigger>
                <MultiTrigger.Conditions>
```

Die beiden folgenden Bedingungen müssen zutreffen:

```
                <Condition Property="IsFocused" Value="True"/>
                <Condition Property="Text" Value="" />
            </MultiTrigger.Conditions>
```

Hier die Aktion:

```
                <Setter Property="Background" Value="Red" />
            </MultiTrigger>
        </Style.Triggers>
    </Style>
</Window.Resources>
```

11.6.2 Ereignis-Trigger

Ereignis-Trigger werden durch bestimmte Ereignisse (vom Typ *RoutedEvent*) ausgelöst. Im Gegensatz zu den Eigenschaften-Triggern können Sie über derartige Trigger jedoch direkt keine Eigenschaften ändern, Sie können „lediglich" Animationen starten, die sich wiederum auf Eigenschaften auswirken.

HINWEIS: Ereignis-Trigger setzen geänderte Eigenschaften nicht wieder zurück, dafür sind **Sie** als Programmierer verantwortlich (z. B. im Pendant des betreffenden Ereignisses).

Beispiel 11.30: Wird der Mauscursor über einen Button bewegt, wird dieser transparent.

XAML

Zunächst die Style-Definition:

```
    <Style TargetType="{x:Type Button}">
        <Style.Triggers>
```

Hier kommt unser Ereignis-Trigger:

```
            <EventTrigger RoutedEvent="Button.MouseEnter">
```

Wir reagieren mit einer Aktion über die Hintergründe:

```
                <EventTrigger.Actions>
                    <BeginStoryboard>
```

Hier die eigentliche Aktion – die Eigenschaft *Opacity* soll in 4 Sekunden von 1 auf 0.25 verringert werden:

```
                        <Storyboard>
                            <DoubleAnimation From="1" To="0.25"
Duration="0:0:4"
```

```
                                        Storyboard.TargetProperty="(Opacity)"/>
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger.Actions>
                </EventTrigger>
```

Beim *MouseLeave*-Ereignis gehen wir den umgekehrten Weg und stellen die ursprüngliche Transparenz wieder her:

```
                <EventTrigger RoutedEvent="Button.MouseLeave">
                    <EventTrigger.Actions>
                        <BeginStoryboard>
                            <Storyboard>
                                <DoubleAnimation From="0.25" To="1"
Duration="0:0:4"
                                                 Storyboard.TargetProperty="(Opacity)"/>
                            </Storyboard>
                        </BeginStoryboard>
                    </EventTrigger.Actions>
                </EventTrigger>
            </Style.Triggers>
        </Style>
```

11.6.3 Daten-Trigger

Mit diesen Triggern können Sie auf das Ändern beliebiger Eigenschaften reagieren. Die Verbindung zu den entsprechenden Eigenschaften stellen Sie per Bindung her. Als Reaktion auf eine Eigenschaftsänderung können Sie, wie auch bei den Eigenschaften-Triggern, mit einem *Setter*-Element bestimmte Eigenschaften ändern.

HINWEIS: Die durch den Trigger geänderten Eigenschaften werden automatisch zurückgesetzt, wenn die Bedingung nicht mehr erfüllt ist.

Beispiel 11.31: Enthält die *TextBox* mehr als zehn Zeichen, wird sie grün eingefärbt.

XAML

```
 <Window.Resources>
   <Style x:Key="myStyle" TargetType="{x:Type TextBox}">
...
      <DataTrigger Binding="{Binding RelativeSource={RelativeSource Self},
                    Path=Text.Length}" Value="10">
          <Setter Property="Background" Value="Green" />
       </DataTrigger>
     </Style.Triggers>
   </Style>
 </Window.Resources>
```

11.7 Einsatz von Templates

Im Folgenden möchten wir Sie zunächst mit ein paar Grundaussagen konfrontieren, bevor wir uns der Thematik „Templates“ bzw. „Vorlagen“ widmen:

- WPF-Controls haben prinzipiell keine Zeichenlogik, es handelt sich um Lookless Controls.
- WPF-Controls stellen lediglich eine Sammlung von Verhalten (Behaviors) dar.

Spinnen denn die Römer? Wo kommen denn sonst die ganzen optischen Spielereien her? Die Antwort auf dieses Paradoxon: Die Zeichenlogik eines Controls wird nur vom Layout/Styling bestimmt. Jedes Control besitzt ein Standardaussehen (Default-Template), das komplett ersetzt werden kann.

Hier haben Sie es mit der Spielwiese der Designer zu tun. Aus dem guten alten viereckigen Button kann ein gänzlich anderes Objekt werden, das jedoch nach wie vor das wesentliche Verhalten eines Buttons (Klick) besitzt. Aus Sicht des Programmierers kann dieser neue Button wie der Standard-Button verwendet werden. Damit ersparen Templates uns vielfach die Mühe, Controls umständlich abzuleiten und deren Zeichenlogik per C#-Code komplett neu zu implementieren.

HINWEIS: Im Gegensatz zu den Styles können Sie bei den Templates nicht nur die vorhandenen Eigenschaften beeinflussen, sondern das Control auch von Grund auf neu zusammenbauen.

11.7.1 Neues Template erstellen

Ein etwas komplexeres Beispiel soll die prinzipielle Vorgehensweise verdeutlichen. Eine Komplettübersicht dieses Themas können wir Ihnen an dieser Stelle leider nicht geben.

Beispiel 11.32: Erzeugen und Verwenden eines Templates

XAML

Unsere Schaltflächen sollen ellipsenförmig sein und einen Farbverlauf aufweisen. Ist die Maus über dem Control, soll sich die Schriftstärke ändern. Ein Niederdrücken der Schaltfläche führt zu einem umgekehrten Farbverlauf im Control.

```
<Window.Resources>
```

Zunächst erstellen wir einen neuen Type-Style für *Button*-Elemente (Sie können auch einen benannten Style verwenden):

```
   <Style TargetType="{x:Type Button}">
```

Hier greifen wir erstmals auf das Template zu. Die Definition ist etwas verschachtelt, da es sich um eine recht komplexe Zuweisung handelt:

```
<Setter Property="Template">
  <Setter.Value>
```

Der Eigenschaft *Template* wird ein *ControlTemplate* zugewiesen:

```
    <ControlTemplate TargetType="{x:Type Button}">
```

Für die innere Ausrichtung nutzen wir zunächst ein *Grid*:

```
      <Grid HorizontalAlignment="Stretch"
            VerticalAlignment="Stretch" ClipToBounds="False">
```

Und hier haben wir es schon mit der Optik zu tun. Wir erzeugen eine Ellipse mit einem Farbverlauf (dies ist der Defaultzustand):

```
      <Ellipse>
        <Ellipse.Fill>
          <LinearGradientBrush StartPoint="0,0" EndPoint="0,1" >
            <LinearGradientBrush.GradientStops>
              <GradientStop Color="#fff399" Offset="0.1"/>
              <GradientStop Color="#ffe100" Offset="0.5"/>
              <GradientStop Color="#feca00" Offset="0.9"/>
            </LinearGradientBrush.GradientStops>
          </LinearGradientBrush>
        </Ellipse.Fill>
      </Ellipse>
```

In unserem Button soll auch etwas angezeigt werden, dafür ist das *ContentPresenter*-Element verantwortlich:

```
    <ContentPresenter x:Name="PrimaryContent""
     HorizontalAlignment="Center" VerticalAlignment="Center"
```

Den eigentlichen Inhalt holen wir uns wiederum vom ursprünglichen Element (dem *Button*), deshalb auch die etwas umständliche Bindung:

```
     Content="{Binding Path=Content,
               RelativeSource={RelativeSource TemplatedParent}}" />
     </Grid>
```

Nicht nur die Standardanzeige unseres Buttons wollen wir beeinflussen, sondern auch die Reaktion auf Maus und Klicken:

```
      <ControlTemplate.Triggers>
```

Es wird geklickt, d. h. wir zeichnen einen neuen Hintergrund. Dazu benötigen wir allerdings den Namen der Ellipse:

```
        <Trigger Property="Button.IsPressed" Value="True">
         <Setter Property="Fill" TargetName="elli" >
```

Achtung: Diese Namen sind nur innerhalb des Templates verwendbar!

```
              <Setter.Value>
                <LinearGradientBrush StartPoint="0,1" EndPoint="0,0" >
                  <LinearGradientBrush.GradientStops>
                    <GradientStop Color="#fff399" Offset="0.1"/>
                    <GradientStop Color="#ffe100" Offset="0.5"/>
                    <GradientStop Color="#feca00" Offset="0.9"/>
                  </LinearGradientBrush.GradientStops>
                </LinearGradientBrush>
              </Setter.Value>
            </Setter>
          </Trigger>
```

Die Maus wird über das Control bewegt. In diesem Fall wird lediglich die Schriftstärke geändert:

```
          <Trigger Property="Button.IsMouseOver" Value="True">
            <Setter Property="FontWeight" Value="Bold" />
          </Trigger>
```

Hier könnten Sie noch auf weitere Eigenschaftsänderungen reagieren …

```
          </ControlTemplate.Triggers>
        </ControlTemplate>
      </Setter.Value>
    </Setter>
```

Last, but not least, setzen wir noch ein paar Eigenschaften:

```
    <Setter Property="Foreground" Value="Black" />
    <Setter Property="FontFamily" Value="Arial" />
    <Setter Property="FontSize" Value="14" />
  </Style>
</Window.Resources>
<StackPanel>
       <Button Height="50">Hallo User</Button>
       <Button Height="76" Margin="10" Width="113">Hallo User
       </Button>
   </StackPanel>
```

Ergebnis

Und wie sieht nun unsere neue Schaltfläche aus? Die Antwort sehen Sie in Bild 11.12.

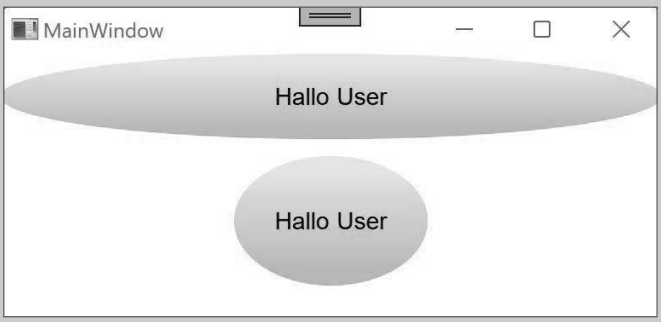

Bild 11.12
Buttons mit geändertem Template

Doch wie können Sie innerhalb des Templates auf die ursprüngliche Definition von Eigenschaften zugreifen? Hier hilft Ihnen ebenfalls Binding weiter.

Beispiel 11.33: Verwendung der *Button.Background*-Eigenschaft

XAML

```
<ControlTemplate TargetType="{x:Type Button}">
  <Grid HorizontalAlignment="Stretch" VerticalAlignment=vStretch"
        ClipToBounds="False">
    <Rectangle Fill="{TemplateBinding Property=Background}"/>
    <Ellipse Name="elli">
```

Ergebnis

Starten Sie das Programm mit dieser Änderung, taucht im Hintergrund zunächst der Default-Farbverlauf eines Buttons auf (so ist *Background* für einen *Button* auch gesetzt).

Bild 11.13 Der graue Hintergrund ist in den Ecken sichtbar.

Werfen wir noch einen Blick auf das *ContentPresenter*-Element in unserem obigen Template. Dessen Content stammt vom ursprünglichen Element ab und bietet damit ebenfalls die Möglichkeit, komplexe Controls „zusammenzubasteln“.

Beispiel 11.34: Wir definieren den zweiten Button mit einer Grafik, der Button übernimmt den vorliegenden Style.

XAML

```
<Button Height="76" Margin="10" Width="113">
    <StackPanel Orientation="Horizontal">
        <Image Source="Images/flash.png" Width="26" Height="26"
               Margin="0,0,10,0"/>
        <TextBlock VerticalAlignment="Center">Action</TextBlock>
    </StackPanel>
</Button>
```

Ergebnis

Das zusammengewürfelte Endergebnis (Grafik und Text per *Content*, Grundlayout per *Template*) präsentiert sich nach wie vor als vollwertiger Button:

Bild 11.14
Button mit Image und TextBox

11.7.2 Template abrufen und verändern

Möchten Sie von einem vorhandenen Control das Template abrufen und eventuell verändern, ist dies seit Visual Studio 2012 kein Problem mehr[3].

Sie können von jedem Control direkt aus dem Designer heraus eine Kopie des Default-Templates editieren. Markieren Sie das betreffende Control im WPF-Editor-Fenster und klicken Sie mit der rechten Maustaste. Über das Kontextmenü rufen Sie *Vorlage bearbeiten/ Kopie bearbeiten* auf:

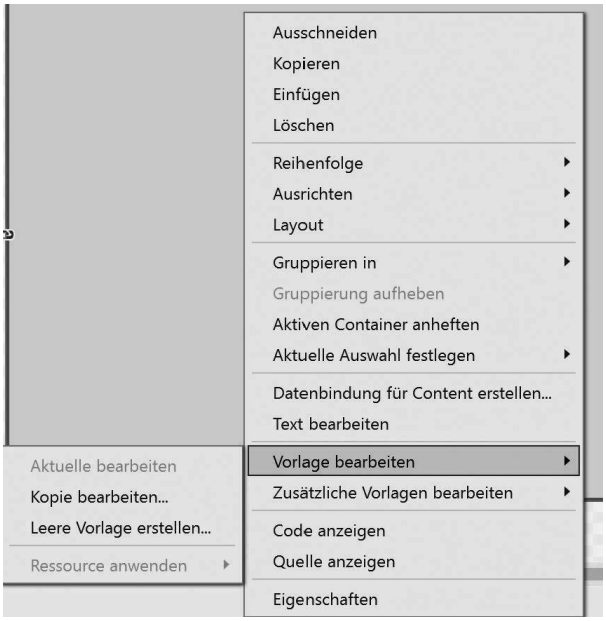

Bild 11.15 Kopie eines Templates erstellen

Im folgenden Dialog (Bild 11.16) bestimmen Sie, ob das Template per Name (benannt) oder per Typ zugeordnet werden soll und wo das Template erstellt wird. Auf das Speichern im aktuellen Window (Dieses Dokument) sollten Sie aus Gründen der Übersichtlichkeit verzichten. Mit der Option *Anwendung* wird das Template in der *app.xaml* erstellt.

[3] In früheren Versionen waren Sie auf die Anwendung *Expression Blend* angewiesen, mittlerweile ist der WPF-Editor in Visual Studio massiv aufgerüstet worden.

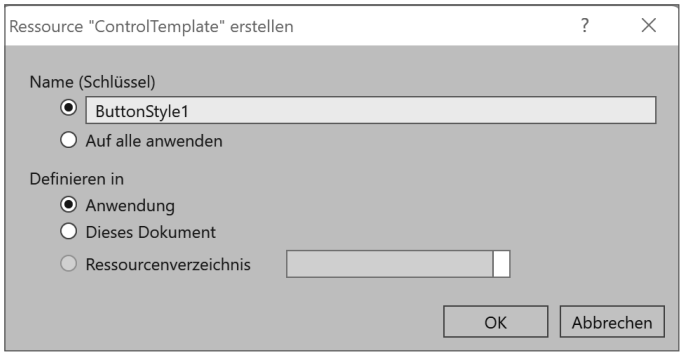

Bild 11.16 ControlTemplate erstellen

Beispiel 11.35: Die derart erstellte Kopie sieht wie folgt aus (Beispiel *Button*):

XAML

```
<Application.Resources>
```

Als Erstes wird ein *FocusVisualStyle* erstellt:

```
    <Style x:Key="FocusVisual">
        <Setter Property="Control.Template">
            <Setter.Value>
                <ControlTemplate>
                    <Rectangle Margin="2" StrokeDashArray="1 2"
             Stroke="{DynamicResource {x:Static SystemColors.ControlTextBrushKey}}"
             SnapsToDevicePixels="true" StrokeThickness="1"/>
                </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
```

Dann folgen Farbdefinitionen:

```
    <SolidColorBrush x:Key="Button.Static.Background" Color="#FFDDDDDD"/>
    <SolidColorBrush x:Key="Button.Static.Border" Color="#FF707070"/>
    <SolidColorBrush x:Key="Button.MouseOver.Background" Color="#FFBEE6FD"/>
    <SolidColorBrush x:Key="Button.MouseOver.Border" Color="#FF3C7FB1"/>
    <SolidColorBrush x:Key="Button.Pressed.Background" Color="#FFC4E5F6"/>
    <SolidColorBrush x:Key="Button.Pressed.Border" Color="#FF2C628B"/>
    <SolidColorBrush x:Key="Button.Disabled.Background" Color="#FFF4F4F4"/>
    <SolidColorBrush x:Key="Button.Disabled.Border" Color="#FFADB2B5"/>
    <SolidColorBrush x:Key="Button.Disabled.Foreground" Color="#FF838383"/>
```

Hier geht es mit der eigentlichen Definition des Buttons los, bei der zunächst einige Einstellungen zugewiesen werden:

```
    <Style x:Key="ButtonStyle1" TargetType="{x:Type Button}">
        <Setter Property="FocusVisualStyle" Value="{StaticResource FocusVisual}"/>
        <Setter Property="Background"
                Value="{StaticResource Button.Static.Background}"/>
        <Setter Property="BorderBrush"
                Value="{StaticResource Button.Static.Border}"/>
```

Hier wird direkt eine Systemressource genutzt. Die Anpassung erfolgt dynamisch, d. h. bei Änderungen in der Systemsteuerung:

```
        <Setter Property="Foreground"
            Value="{DynamicResource {x:Static SystemColors.ControlTextBrushKey}}"/>
```

Dann werden weitere Properties gesetzt:

```
        <Setter Property="BorderThickness" Value="1"/>
        <Setter Property="HorizontalContentAlignment" Value="Center"/>
        <Setter Property="VerticalContentAlignment" Value="Center"/>
        <Setter Property="Padding" Value="1"/>
```

Das Default-Template wird als Kopie über einen weiteren *Setter* definiert:

```
        <Setter Property="Template">
           <Setter.Value>
               <ControlTemplate TargetType="{x:Type Button}">
                   <Border x:Name="border" Background="{TemplateBinding
Background}"
                           BorderBrush="{TemplateBinding BorderBrush}"
                           BorderThickness="{TemplateBinding BorderThickness}"
                           SnapsToDevicePixels="true">
                       <ContentPresenter x:Name="contentPresenter"
Focusable="False"
                                HorizontalAlignment=
                                    "{TemplateBinding HorizontalContentAlignment}"
                                Margin="{TemplateBinding Padding}"
                                RecognizesAccessKey="True"
                                SnapsToDevicePixels=
                                    "{TemplateBinding SnapsToDevicePixels}"
                                VerticalAlignment=
                                     "{TemplateBinding VerticalContentAlignment}"/>
                   </Border>
```

Die Reaktion auf Eigenschaften-Trigger und Mausbewegungen:

```
                   <ControlTemplate.Triggers>
                        <Trigger Property="IsDefaulted" Value="true">
                            <Setter Property="BorderBrush" TargetName="border"
            Value="{DynamicResource {x:Static SystemColors.HighlightBrushKey}}"/>
                        </Trigger>
                        <Trigger Property="IsMouseOver" Value="true">
                            <Setter Property="Background" TargetName="border"
            Value="{StaticResource Button.MouseOver.Background}"/>
                            <Setter Property="BorderBrush" TargetName="border"
            Value="{StaticResource Button.MouseOver.Border}"/>
                        </Trigger>
                        <Trigger Property="IsPressed" Value="true">
                            <Setter Property="Background" TargetName="border"
            Value="{StaticResource Button.Pressed.Background}"/>
                            <Setter Property="BorderBrush" TargetName="border"
           Value="{StaticResource Button.Pressed.Border}"/>
                        </Trigger>
                        <Trigger Property="IsEnabled" Value="false">
                            <Setter Property="Background" TargetName="border"
```

```
            Value="{StaticResource Button.Disabled.Background}"/>
                        <Setter Property="BorderBrush" TargetName="border"
            Value="{StaticResource Button.Disabled.Border}"/>
                        <Setter Property="TextElement.Foreground"
                                TargetName="contentPresenter"
            Value="{StaticResource Button.Disabled.Foreground}"/>
                    </Trigger>
                </ControlTemplate.Triggers>
            </ControlTemplate>
            </Setter.Value>
        </Setter>
    </Style>
</Application.Resources>
```

HINWEIS: Ändern Sie in diesem Template beispielsweise die Farben im *IsMouseOver*-Trigger (siehe fett hervorgehobene Zeilen), dann wird sich das Aussehen aller Schaltflächen in Ihrer Anwendung ändern.

Hauptzielgruppe dieses Features dürfte jedoch nicht der Programmierer, sondern der Designer der Anwendung sein. Gleiches trifft ebenfalls auf das im folgenden Abschnitt behandelte Storyboard zu.

11.8 Transformationen, Animationen, StoryBoards

Im Folgenden wollen wir in einem „Schnelldurchlauf für Programmierer“ noch einen Blick auf WPF-typische Features werfen, auch wenn diese im Allgemeinen nicht zum Hauptarbeitsgebiet des Entwicklers gehören.

11.8.1 Transformationen

Mithilfe von Transformationen können Sie in WPF das optische Standardverhalten problemlos verändern, ohne sich um Templates oder Styles kümmern zu müssen. Sie können die Größe, Position, Drehung und Verzerrung der betroffenen Controls über die einfache Zuweisung einer entsprechenden Transformation ändern (statische Änderung).

HINWEIS: Damit sind diese Operationen auch die Vorstufe für einfache Animationen (dynamische Änderungen in Abhängigkeit von der Zeit).

Folgende Möglichkeiten stehen Ihnen zur Verfügung:

Transformation	Beschreibung
RotateTransform	Element um einen bestimmten Winkel drehen
ScaleTransform	Element vergrößern/verkleinern
SkewTransform	Element verformen
TranslateTransform	Element verschieben
MatrixTransform	Zusammenfassen der obigen Transformationen per 3x3-Transformationsmatrix

Lassen Sie uns nun an einfachen Beispielen die Wirkung der jeweiligen Transformation demonstrieren.

Drehen mit RotateTransform

Mit *RotateTransform* realisieren Sie eine Drehung des Elements im Uhrzeigersinn. Den Drehpunkt können Sie optional festlegen, per Default ist dies die linke obere Ecke.

Beispiel 11.36: Button um 40° drehen

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button"
            Height="50" Width="100" >
        <Button.RenderTransform>
            <RotateTransform Angle="40"/>
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis

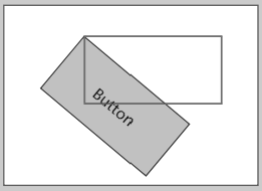

Bild 11.17
Um 40° gedrehter Button

Das *Angle*-Attribut bestimmt den Drehwinkel um den per *CenterX*- und *CenterY*-Attribut festgelegten Drehpunkt. Natürlich können Sie für eine andere Drehrichtung auch negative Werte übergeben.

HINWEIS: Durch die Rotation wird das Koordinatensystem des gedrehten Elements verändert!

Skalieren mit ScaleTransform

Soll ein Element skaliert werden, nutzen Sie eine *ScaleTransform*.

Beispiel 11.37: Button skalieren

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="300" Content="Button"
            Height="50" Width="100" >
        <Button.RenderTransform>
            <ScaleTransform ScaleX="1.5" ScaleY="2"/>
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="300" Height="50"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis

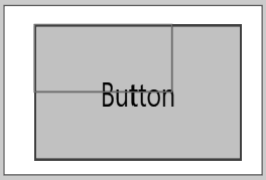

Bild 11.18
Button skalieren

Mit den *ScaleX*- und *ScaleY*-Attributen bestimmen Sie den Skalierungsfaktor für die X- bzw. Y-Achse. Mit *CenterX* und *CenterY* bestimmen Sie den Fixpunkt, von dem aus die Skalierung gestartet wird. Dies ist per Default die linke obere Ecke.

Verformen mit SkewTransform

Mit *SkewTransform* verformen Sie das Koordinatensystem um bestimmte Winkel.

Beispiel 11.38: Verformen des Koordinatensystems

XAML

```
<Canvas>
    <Button Canvas.Left="100" Canvas.Top="100" Content="Button"
            Height="50" Width="100" >
        <Button.RenderTransform>
            <SkewTransform AngleY="25" AngleX="15" />
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="100" Canvas.Top="100" Height="50"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis

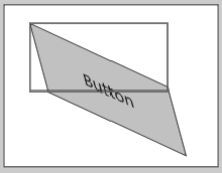

Bild 11.19
Verformung des Koordinatensystems

Den Verformungsgrad bestimmen Sie mit den Attributen *AngleX* und *AngleY*. *CenterX* und *CenterY* legen den Ursprungspunkt fest.

Verschieben mit TranslateTransform

Auch die Verschiebung eines Elements ist mit *TranslateTransform* kein Problem. Sie können mit den X- und Y-Attributen die horizontale bzw. vertikale Verschiebung bestimmen.

Beispiel 11.39: Button verschieben

XAML

```
<Canvas>
    <Button Canvas.Left="300" Canvas.Top="300" Content="Button"
            Height="50" Width="100" >
        <Button.RenderTransform>
            <TranslateTransform X="50" Y="25"/>
        </Button.RenderTransform>
    </Button>
    <Rectangle Canvas.Left="300" Canvas.Top="300" Height="50"
               Stroke="Red" Width="100" />
</Canvas>
```

Ergebnis

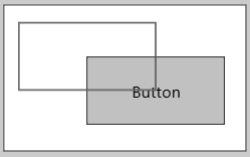

Bild 11.20
Button verschieben

Und alles zusammen mit TransformGroup

Sicher sind Sie auch schon versucht gewesen, mehrere der obigen Effekte gleichzeitig zu realisieren. Allerdings dürfte Ihnen die Syntaxprüfung des XAML-Editors dabei einen Strich durch die Rechnung gemacht haben.

HINWEIS: Um mehrere Transformationen gleichzeitig zuzuweisen, müssen Sie eine *TransformGroup* verwenden.

Beispiel 11.40: Mehrere Transformationen gleichzeitig anwenden

XAML

```
<Canvas>
    <Button Canvas.Left="500" Canvas.Top="100" Content="Button"
            Height="50" Width="100" >
        <Button.RenderTransform>
            <TransformGroup>
                <ScaleTransform ScaleX=".75" ScaleY="1.5"/>
                <RotateTransform Angle="45"></RotateTransform>
                <TranslateTransform X="50" Y="25"/>
            </TransformGroup>
```

```
            </Button.RenderTransform>
        </Button>
        <Rectangle Canvas.Left="500" Canvas.Top="100" Height="50"
                   Stroke="Red" Width="100" />
    </Canvas>
```

Ergebnis

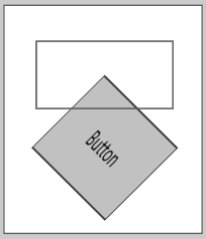

Bild 11.21
Transformationsgruppen

Doch Achtung: Hier spielt die Reihenfolge in der Gruppe eine bedeutende Rolle, wie folgende kleine Änderung (erst Drehung, dann Skalierung) zeigt. Ursache ist die Veränderung des Koordinatensystems des betroffenen Controls.

Beispiel 11.41: Änderung der Transformationsreihenfolge

XAML

```
<TransformGroup>
    <RotateTransform Angle="45"></RotateTransform>
    <ScaleTransform ScaleX=".75" ScaleY="1.5"/>
    <TranslateTransform X="50" Y="25"/>
</TransformGroup>
```

Ergebnis

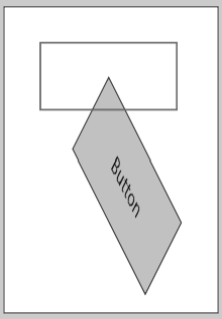

Bild 11.22
Transformationsgruppe mit anderer Reihenfolge

Hilfe durch den WPF-Editor

Wem die obigen Ausführungen zu kompliziert und wenig intuitiv waren, der kann es auch einfacher haben. Nutzen Sie einfach den in Visual Studio integrierten WPF-Editor, dieser ist mittlerweile sogar ganz brauchbar und bietet unter anderem auch die Möglichkeit, die *Rotate-Transformation* per Maus umzusetzen.

Im Eigenschaftenfenster können Sie zusätzlich die anderen Transformationsarten getrennt parametrieren:

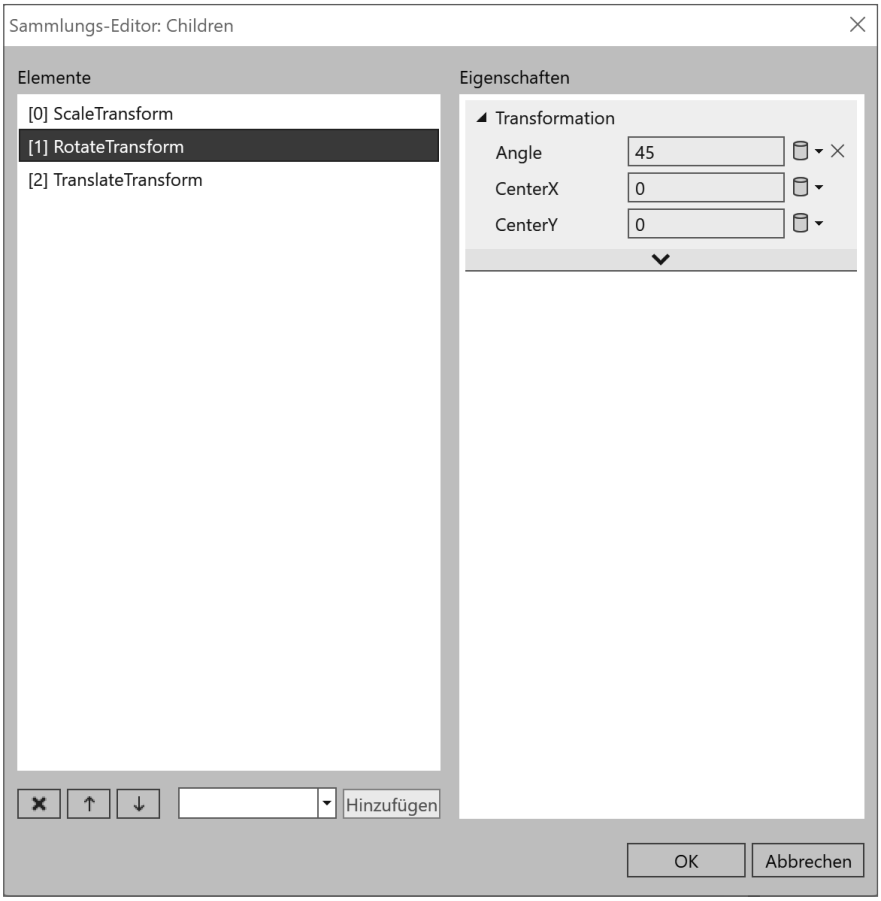

Bild 11.23 Editor für RotateTransform

11.8.2 Animationen mit dem StoryBoard realisieren

Für die Realisierung von Animationen werden in WPF sogenannte Storyboards verwendet, die wiederum einzelne oder mehrere Animationen (zeitliche Veränderungen von Eigenschaften) enthalten können. Storyboards können wiederum über bestimmte Ereignis-Trigger ausgelöst, angehalten, fortgesetzt oder auch beendet werden (alternativ natürlich auch per Code).

Ein erstes einfaches Beispiel soll die prinzipielle Vorgehensweise beim Animieren einer Eigenschaft (in diesem Fall der Transparenz) demonstrieren.

Beispiel 11.42: Ausblenden eines Buttons, wenn die Maus darüber bewegt wird

XAML

```
    <Canvas>
```

Zunächst den Button definieren:

```
        <Button Canvas.Left="700" Canvas.Top="100" Content="Button"
                Height="50" Width="100" Name="button1" >
```

Wir reagieren mit einem *Trigger* auf das Hineinbewegen der Maus:

```
            <Button.Triggers>
                <EventTrigger RoutedEvent="Button.MouseEnter">
```

Aufgrund des ausgelösten Trigger-Ereignisses wird das folgende *StoryBoard* ausgeführt:

```
                    <BeginStoryboard>
                        <Storyboard x:Name="Storyboard1">
```

Das *StoryBoard* enthält eine *DoubleAnimation* (Verändern einer *Double*-Eigenschaft) mit einer Zeitdauer (*Duration*) von 4 Sekunden:

```
                            <DoubleAnimation Duration="0:0:4"
```

Ziel der Eigenschaftsänderung ist *button1*:

```
                             Storyboard.TargetName="button1"
```

Die zu ändernde Eigenschaft ist *Opacity*:

```
                             Storyboard.TargetProperty="Opacity"
```

Die Eigenschaft wird in der oben genannten Zeitdauer von 1 auf 0 geändert:

```
                             From="1" To="0" />
                        </Storyboard>
                    </BeginStoryboard>
                </EventTrigger>
            </Button.Triggers>
        </Button>
        <Rectangle Canvas.Left="700" Canvas.Top="100" Height="50"
                   Stroke="Red" Width="100" />
    </Canvas>
```

Das war hoffentlich nicht allzu abschreckend. Es geht teilweise auch einfacher und alternativ könnten Sie auch mit sinnvollen Werkzeugen den obigen XAML-Code erstellen.

Animation per C#-Code realisieren

Im obigen Fall müssen Sie immer ein StoryBoard einsetzen, um die Animation(en) zu kapseln. Etwas einfacher geht es, wenn Sie lediglich eine Animation per C#-Code realisieren wollen. In diesem Fall erstellen Sie einfach eine Instanz der gewünschten Animation (davon gibt es

je nach Zieleigenschaft unterschiedliche), parametrieren diese und starten die Animation, indem Sie diese an die *BeginAnimation*-Methode des gewünschten Controls übergeben.

Beispiel 11.43: Eine einfache Animation per Code definieren und ausführen

C#

```
using System.Windows;
using System.Windows.Media.Animation;
...
    private void Button2_Click(object sender, RoutedEventArgs e)
    {
```

Instanz erstellen:

```
        DoubleAnimation animation = new DoubleAnimation();
```

Parametrieren:

```
            animation.From = 1;
            animation.To = 0;
```

Starten (Transparenz ändern):

```
            button2.BeginAnimation(OpacityProperty, animation);
        }
```

Das war doch gar nicht so schwierig, oder?

Animation per Code steuern

Vielleicht dämmert es Ihnen schon, komplexe Animationen bzw. die Zusammenfassung mehrerer Animationen als Storyboard sind kaum für die tägliche Praxis des C#-Entwicklers geeignet. Abgesehen davon, dass sie Unmengen von C#-Code erzeugen, fehlt bei vielen Animationen einfach die Vorstellungskraft. Dauernde Programmstarts zum Ausprobieren der Effekte zehren auch an den Nerven und kosten Zeit. Ganz nebenbei ist auch die Parametrierung vieler Eigenschaften mit C# eine Pein – hier kann XAML seine Vorteile deutlich ausspielen.

Viel besser ist es, die Storyboards mit einem Programm wie Microsoft Blend zu erstellen und nachträglich in die Anwendung einzufügen. Zum Starten der Animation können Sie entweder, wie bereits gezeigt, einen Trigger verwenden, oder Sie nutzen das *Storyboard* per Code. Dazu stellt die *Storyboard*-Klasse mehrere Methoden bereit, mit denen Sie die Animation gezielt kontrollieren können:

Methode	Beschreibung
Begin	Animationen des *Storyboards* starten.
Pause	Wiedergabe anhalten.
Resume	Wiedergabe fortsetzen.
Seek	Bei der Wiedergabe zu einer Position im *Storyboard* springen. Verwenden Sie einen *TimeSpan*-Wert.
Stop	Wiedergabe anhalten und Wiedergabeposition zurücksetzen.

HINWEIS: Auf das Ende der Animationen im *Storyboard* können Sie mit dessen *Completed*-Ereignis reagieren.

Beispiel 11.44: Als Ressource definierte Animation per Code starten

XAML

```
<Window x:Class="TransformationenSample.MainWindow"
...
        Title="MainWindow" Height="600" Width="1200">
```

Als Window-Ressource definieren wir ein *Storyboard*:

```
    <Window.Resources>
```

Achten Sie darauf, einen *Key* zu vergeben:

```
        <Storyboard x:Key="storyboard2">
          <DoubleAnimation Duration="0:0:4" Storyboard.TargetName="button3"
                           Storyboard.TargetProperty="Width"  To="300" />
        </Storyboard>
    </Window.Resources>
        <Canvas>
        <Button Canvas.Left="900" Canvas.Top="100" Content="Button"
                Height="50" Name="button3" Width="100"
                Click="Button3_Click">
        </Button>
...
```

C#

Zunächst den Namespace importieren:

```
...
using System.Windows.Media.Animation;
...
```

Zur Laufzeit können wir unser *Storyboard* suchen und mit der *Begin*-Methode starten:

```
       private void Button3_Click(object sender, RoutedEventArgs e)
        {
            Storyboard? sb = FindResource("storyboard2") as Storyboard;
            sb?.Begin();
```

Ist für das *Storyboard* kein *TargetName* vorgegeben, können Sie das *Storyboard* auch auf jedes andere Control anwenden, wenn Sie dieses an die *Begin*-Methode übergeben:

```
            sb?.Begin(button3);
        }
```

HINWEIS: Ein Klick auf eine andere Taste könnte beispielsweise mit *Storyboard.Stop* die Animation anhalten.

Selbstverständlich können Sie ein in den Ressourcen abgelegtes Storyboard auch per XAML einbinden bzw. starten. In diesem Fall benötigen Sie nicht eine Zeile C#-Code, können aber die Animationen (bzw. die übergeordneten Storyboards) zentral verwalten.

Beispiel 11.45: Alternative zum vorhergehenden Beispiel

XAML

```
<Window x:Class="Animation_Bsp.MainWindow"
...
    <Window.Resources>
        <Storyboard x:Key="storyboard2">
          <DoubleAnimation Duration="0:0:4" Storyboard.TargetName="button3"
                           Storyboard.TargetProperty="Width" To="300" />
        </Storyboard>
    </Window.Resources>
...
        <Button Canvas.Left="900" Canvas.Top="300" Content="Button"
                Height="23" Name="button4" Width="75" >
```

Per Trigger starten wir ein *Storyboard*:

```
            <Button.Triggers>
                <EventTrigger RoutedEvent="Button.MouseEnter">
```

Hier weisen wir die Ressource zu:

```
                    <BeginStoryboard Storyboard="{StaticResource storyboard2}" />
                </EventTrigger>
            </Button.Triggers>
        </Button>
    </Canvas>
</Window>
```

Haben Sie ein *Storyboard* ohne *TargetName* (universelle Verwendung), müssen Sie diesen beim Einbinden der Ressource angeben, um auch das Ziel der Animation zu bestimmen:

Beispiel 11.46: Ziel bestimmen

XAML

```
...
    <EventTrigger RoutedEvent="Button.MouseEnter">
        <BeginStoryboard Storyboard="{StaticResource storyboard4}"
                         Storyboard.TargetName="button4" />
    </EventTrigger>
...
```

Mehrere Animationen zusammenfassen

Dass eine Animation nur auf der linearen Änderung einer Eigenschaft basiert, dürfte wohl selten der Fall sein. Meist werden mehrere Eigenschaften gleichzeitig geändert. Auch das ist mit dem *Storyboard* kein Problem, wie es das folgende Beispiel zeigt:

Beispiel 11.47: *Storyboard* mit drei Animationen

XAML

```
<Window x:Class="TransformationenSample.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:TransformationenSample"
   mc:Ignorable="d"
   Title="MainWindow" Height="600" Width="1200">
    <Window.Resources>
      <Storyboard x:Key="storyboard3">
         <DoubleAnimation Duration="0:0:4"
            Storyboard.TargetProperty="Width" To="300" />
         <DoubleAnimation Duration="0:0:4"
            Storyboard.TargetProperty="RenderTransform.Angle" To="360" />
         <ColorAnimation  Duration="0:0:3"
            Storyboard.TargetProperty="Foreground.Color"
                 From="Rewd"  To="Blue" />
       </Storyboard>
    </Window.Resources>
```

Das soll zu diesem Thema genügen. Auch wenn WPF im Bereich „Animationen“ fast unbegrenzte Möglichkeiten bietet, so sprechen diese doch kaum den Programmierer, sondern eher den Designer der Anwendung an. Machen Sie sich also nicht die Mühe, komplexe Storyboards per XAML oder gar C#-Quellcode zu erstellen, sondern nutzen Sie die Vorteile von Blend für Visual Studio. Erstellen Sie damit interaktiv den entsprechenden XAML-Code und fügen Sie diesen in die Ressourcen Ihrer Anwendung ein. Damit ersparen Sie sich viele graue Haare und haben mehr Zeit für die eigentliche Anwendungsentwicklung.