

## Kapitel 12

# Steuerelemente

### **In diesem Kapitel:**

Einführung	236
Weitergeleitete Ereignisse (Routed Events)	237
Weitergeleitete Befehle (Routed Commands)	246
Eigenschaften der Abhängigkeiten	250
Eingaben	253
Schaltflächen	255
Bildlaufleisten und Schieberegler	257
Steuerelemente für die Texteingabe	261
Das Label-Element	265
Menüs	266
Werkzeugleisten (Toolbars)	271
Zusammenfassung	274

Mit Windows Presentation Foundation wollen wir die Benutzerschnittstellen unserer Applikationen deklarieren. Das funktioniert natürlich nicht ohne Steuerelemente wie Schaltflächen, Eingabefelder, Bildlaufleisten, usw. Diese Elemente stehen Ihnen auch in WPF zur Verfügung. Jedes WPF-Element enthält im .NET Framework 3.0 eine entsprechende Klasse, die das Element mit allen Eigenschaften, Methoden und Ereignissen abbildet.

In diesem Buch beschäftigen sich zwei Kapitel mit den Steuerelementen von WPF. In diesem Kapitel lernen Sie die Standardsteuerelemente, wie z. B. TextBox- oder Button-Elemente, kennen. In einem zweiten Kapitel werden die Elemente behandelt, die sich mit dem Anordnen von Steuerelementen (Layout) beschäftigen.

**BEGLEITDATEIEN**

Unter `.\Samples\Chapter12\` finden Sie die Beispieldateien für dieses Kapitel.

## Einführung

Steuerelemente sind kleine Einheiten der Benutzerschnittstelle einer Applikation. Normalerweise bestehen sie selbst wieder aus einer meist einfachen Benutzerschnittstelle (Aussehen des Steuerelements) und einer implementierten Logik (Verhalten). Häufig bilden die Steuerelemente eine Abstraktionsschicht in einem Framework für Benutzerschnittstellen. Dies ist in WPF allerdings nicht so, da die Steuerelemente hier nicht ihr Aussehen »implementieren«. Das Aussehen der WPF-Steuerelemente wird über Templates (Vorlagen) realisiert. Diese Templates oder Vorlagen können jederzeit ausgetauscht werden. Dadurch wird zwar das Aussehen eines Steuerelements modifiziert, sein Verhalten bleibt jedoch unverändert. Dieses Verhalten wird in Abbildung 12.1 dargestellt.

Das Steuerelement besteht weiterhin aus der Logik, die sein Verhalten festlegt. Hierzu gehören die Methoden, die Eigenschaften und Ereignisse, die ausgelöst werden können. Ein Anwender interagiert mit dem Steuerelement über diese Logik. Die Darstellung des Steuerelements ist in Abbildung 12.1 allerdings losgelöst von der Logik als Template (Vorlage) implementiert. Über diese Vorlage wird dem Anwender das Steuerelement mit seinem Inhalt dargestellt. Auch bei den WPF-Steuerelementen ist also Logik und Design getrennt. Eine solche Trennung ist bei Steuerelementen meistens nicht vollständig zu erreichen. Trotzdem macht diese Aufteilung Sinn, wie wir in diesem und im Kapitel über Vorlagen sehen werden.

Der Vorteil dieser Vorgehensweise in WPF liegt eigentlich auf der Hand. Wenn Sie aus einer normalen, rechteckigen Schaltfläche mit grauem Hintergrund eine elliptische Schaltfläche mit einem Bild und einem Farbgradienten machen wollen, so ist das möglich, ohne ein neues Steuerelement zu programmieren. Das Verhalten (Logik) der Schaltfläche soll erhalten bleiben. Was geändert werden muss, ist nur das Aussehen, welches über eine Vorlage abgebildet wird und separat geändert werden kann.

Alle WPF-Steuerelemente können Ereignisse auslösen, auf die ein Softwareentwickler in Ereignismethoden reagieren kann. In dieser Hinsicht unterscheidet sich WPF nicht von anderen Frameworks für Benutzerschnittstellen. Steuerelemente können natürlich auch Methoden enthalten, die ganz bestimmte Arbeiten erledigen. So können Sie z. B. mit einer `SelectAll`-Methode alle Elemente in einem `ListBox`-Steuerelement auswählen.

Die Ereignisse (*Events*), Befehle (*Commands*) und Eigenschaften (*Properties*) von WPF-Steuerelementen wollen wir in den folgenden Abschnitten noch etwas genauer betrachten.

**HINWEIS** Beachten Sie bitte auch bitte das im vorherigen Kapitel zu Ereignissen Gesagte, das sich auf die spezielle Eigenart von Visual Basic-Anwendungen bezieht, Ereignisprozeduren mit dem Handels-Schlüsselwort mit Ereignissen bestimmter Objekte zu verknüpfen.

## Weitergeleitete Ereignisse (Routed Events)

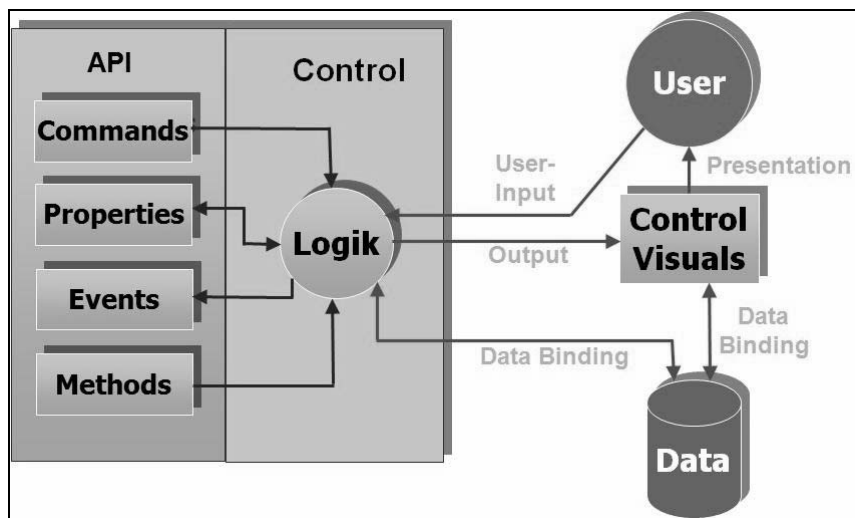


Abbildung 12.1 Aufbau der Steuerelemente in WPF

Ein sehr wichtiges Konzept für Steuerelemente unter Windows Presentation Foundation sind die *Routed Events*. Routed Events sind Ereignisse, die an verschiedene WPF-Elemente in der Hierarchie weiter geleitet werden können. Hierzu wollen wir uns einmal vorstellen, dass z.B. eine WPF-Schaltfläche aus mehreren Elementen bestehen kann: Auf der Schaltfläche kann sich ein Grid-Element mit zwei Zellen befinden. Eine Zelle enthält ein Canvas-Element mit mehreren Grafik-Objekten. Die zweite Zelle kann ein Border- und ein TextBlock-Element beinhalten. Alle Objekte in diesem Szenario (auch die grafischen Elemente) können nun z.B. ein MouseDown-Ereignis auslösen. Abbildung 12.2 zeigt ein entsprechendes Beispiel.

In WPF gibt es drei Ereignistypen:

- Direkte Ereignisse (*Direct Events* etwa: *direkte Ereignisse*)
- In der Hierarchie nach oben laufende Ereignisse (*Bubbling Events* etwa: *aufsprudelnde Ereignisse*)
- In der Hierarchie nach unten laufende Ereignisse (*Tunneling Events* etwa: *tunnelnde Ereignisse*)

Die drei Ereignistypen verhalten sich unterschiedlich und müssen jeweils an der richtigen Stelle angewendet werden.

Die Direct Events sind mit den normalen CLR-Ereignissen identisch. Ein solcher Event wird nur in dem Element verarbeitet, in dem er erzeugt wurde. Denken Sie bei diesen Ereignissen z.B. an ein `MouseLeave`- oder `MouseEnter`-Ereignis. Diese Ereignisse sind eigentlich nur für das jeweilige Element interessant, welches vom Mauszeiger betreten oder verlassen wird. Andere WPF-Elemente in der logischen Hierarchie werden sich nicht für diese Ereignisse interessieren.

Bubbling Events und Tunneling Events laufen nun im Gegensatz zu den Direct Events durch die logische Hierarchie und werden somit an andere WPF-Elemente weiter gegeben.

Ein Bubbling Event beginnt seine Reise durch die logische Hierarchie beim auslösenden Element und wird dann nach oben bis zum Wurzelement weiter geleitet. In welchem Element Sie das Ereignis nun bearbeiten, liegt ganz bei Ihnen. Sie können das Ereignis auch in mehreren WPF-Elementen verarbeiten. In diesem Fall implementieren Sie die Ereignismethode mehrfach unter verschiedenen Namen. Ein typisches Bubbling Event ist z.B. das `MouseDown`-Ereignis.

Ein Tunneling Event startet in der logischen Hierarchie oben beim Wurzelement und fällt dann nach unten bis zum auslösenden Element durch. Auch hier können Sie das Ereignis an beliebiger Stelle in der Hierarchie bearbeiten.

Wenn ein WPF-Element ein Tunneling Event und einen Bubbling Event auslöst, so wird zunächst immer das von oben beginnende Ereignis ausgelöst. Wir sprechen hier von einem *Vorschau*-Ereignis (Preview-Event). Ein ausgelöstes Vorschau-Ereignis können Sie sehr gut benutzen, um das Ereignis selbst zu blockieren, sodass es nicht durch die logische Hierarchie hindurch bis zum auslösenden Element wandert. Das Vorschau-Ereignis zum `MouseDown`-Ereignis heißt dann `PreviewMouseDown`. Vorschau-Ereignisse haben also immer die Vorsilbe `Preview`.

Die Elementhierarchie im hier verwendeten Beispiel (Abbildung 12.2) sieht vereinfacht folgendermaßen aus:

- Window
- Grid
- Canvas
- Ellipse
- Rectangle
- Border
- TextBlock

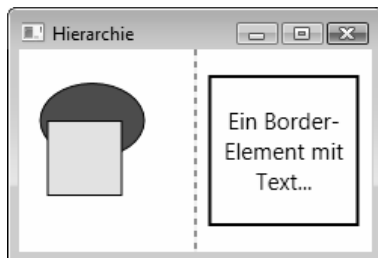


Abbildung 12.2 Ein Fenster mit einer Elementhierarchie

In dieser Hierarchie können Sie sehen, dass die beiden grafischen Elemente `Rectangle` und `Ellipse` auf einer Ebene liegen. Trotzdem liegen sie in der Darstellung übereinander, da das Rechteck im XAML-Code erst nach der Ellipse deklariert wurde. Wir wollen nun sehen, welche Ereignisse in dieser Hierarchie ausgelöst werden. Dazu implementieren wir im Beispiel die diversen Ereignismethoden für das Bubbling und das Tunneling `MouseDown`-Ereignis.

```
<Window x:Class="Hierarchie.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Hierarchie"
  Height="170" Width="250"
  MouseDown="OnWindow" PreviewMouseDown="OnWindowPrev"
  >
  <Grid ShowGridLines="True" MouseDown="OnGrid" PreviewMouseDown="OnGridPrev">
    <!-- Spalten für Grid-Element definieren -->
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <!-- Element für Grafik in der linken Grid-Zelle -->
    <Canvas Grid.Column="0" Width="100" Height="100"
      MouseDown="OnCanvas" PreviewMouseDown="OnCanvasPrev">
      <Ellipse Canvas.Left="5" Canvas.Top="5" Stroke="Black" Fill="Red" Width="70" Height="50"
        MouseDown="OnEllipse" PreviewMouseDown="OnEllipsePrev" />
      <Rectangle Canvas.Left="10" Canvas.Top="30" Stroke="Black" Fill="Yellow" Width="50" Height="50"
        MouseDown="OnRectangle" PreviewMouseDown="OnRectanglePrev"/>
    </Canvas>

    <!-- Rand- mit Text-Element in der rechten Grid-Zelle -->
    <Border Grid.Column="1" Width="100" Height="100" BorderThickness="2"
      BorderBrush="Black" MouseDown="OnBorder" PreviewMouseDown="OnBorderPrev">
      <TextBlock FontSize="15" TextWrapping="Wrap" TextAlignment="Center"
        MouseDown="OnTextBlock" PreviewMouseDown="OnTextBlockPrev"
        HorizontalAlignment="Center" VerticalAlignment="Center">
        Ein Border-Element mit Text...
      </TextBlock>
    </Border>
  </Grid>
</Window>
```

**Listing 12.1** Die logische Hierarchie für das Beispiel in Abbildung 12.2

Da wir noch nicht so viele XAML-Hierarchien erzeugt haben, wollen wir hier noch einmal etwas genauer auf das Listing 12.1 eingehen. Das Wurzelement ist hier ein `Window`-Element mit dem Klassennamen `Window1`. Für dieses Element wurden sowohl das `MouseDown`-Ereignis als auch das `PreviewMouseDown`-Ereignis implementiert. Das Bubbling Event wird in der Methode `OnWindow` und das dazugehörige Tunneling Event in der Methode `OnWindowPrev` bearbeitet. Den VB-Code zur XAML-Deklaration finden Sie in Listing 12.2.

Dort werden auch alle Ereignismethoden, welche dieses Beispiel benötigt, implementiert. In den einzelnen Ereignismethoden wird immer ein Informationstext an eine String-Variable angehängt. Hier werden keine MessageBox-Ausgaben benutzt, da die Erzeugung eines neuen Fensters den Durchlauf des MouseDown-Ereignisses beeinflusst. Um den Ereignisdurchlauf ohne irgendwelche Wechselwirkungen mit anderen WPF-Elementen zu gewährleisten, werden die in der String-Variablen gesammelten Informationen im Finalizer der Hauptfensterklasse mithilfe eines StreamWriter-Objekts in eine Datei geschrieben, die wir dann nach dem Schließen der Applikation mit dem Notepad von Windows anschauen können.

**HINWEIS** Wenn Sie für die Ausgabe der Informationen in den Ereignismethoden eine MessageBox benutzen, werden normalerweise die Ereignismethoden der Bubbling Events nicht aufgerufen.

Zurück zum XAML-Code (Listing 12.1). Innerhalb des Hauptfensters wird zunächst ein Grid-Element deklariert, für das auch das MouseDown- und das PreviewMouseDown-Ereignis implementiert werden. In diesem Fall werden die Methoden OnGrid bzw. OnGridPrev angesprochen. Über die Grid.ColumnDefinition-Eigenschaft werden dann zwei Spalten für das Grid definiert. Nun geht es weiter in der logischen Hierarchie mit den WPF-Elementen in den beiden Grid-Zellen. In der linken Zelle (Grid.Column="0") wird ein Canvas-Element deklariert, welches die beiden Grafikelemente Rectangle und Ellipse enthält. Für alle Elemente werden die beiden Ereignisse MouseDown und PreviewMouseDown implementiert. Schließlich wird die rechte Grid-Zelle zunächst mit einem Rand (Border) versehen, welcher dann ein TextBlock-Element mit den jeweiligen Ereignis-Aufrufen enthält.

```
Imports System
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Input
Imports System.IO

Namespace Hierarchie
    Partial Public Class Window1
        Inherits System.Windows.Window
        Dim strTest As String = ""

        Public Sub New()
            InitializeComponent()
        End Sub

        Protected Overrides Sub Finalize()
            MyBase.Finalize()
            ' Im Finalizer der Window1-Klasse werden die gesammelten
            ' Daten in eine Datei gespeichert.
            Dim sw As StreamWriter = New StreamWriter("C:\test.txt")
            sw.Write(strTest)
            sw.Flush()
            sw.Close()
        End Sub

        ' Nun kommen alle Ereignismethoden
        Private Sub OnGrid(ByVal sender As Object, _
            ByVal e As RoutedEventArgs)
            strTest = strTest + "Grid" & vbNewLine
        End Sub
    End Class
End Namespace
```

```
Private Sub OnGridPrev(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    strTest = strTest + "Grid - Preview" & vbCrLf
End Sub

Private Sub OnWindow(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    strTest = strTest + "Window" & vbCrLf
End Sub

Private Sub OnWindowPrev(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    strTest = strTest + "Window - Preview" & vbCrLf
End Sub

Private Sub OnCanvas(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    strTest = strTest + "Canvas" & vbCrLf
End Sub

Private Sub OnCanvasPrev(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    strTest = strTest + "Canvas - Preview" & vbCrLf
End Sub

Private Sub OnTextBlock(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    strTest = strTest + "Textblock" & vbCrLf
End Sub

Private Sub OnTextBlockPrev(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    strTest = strTest + "Textblock - Preview" & vbCrLf
End Sub

Private Sub OnBorder(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    strTest = strTest + "Border" & vbCrLf
End Sub

Private Sub OnBorderPrev(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    strTest = strTest + "Border - Preview" & vbCrLf
End Sub

Private Sub OnEllipse(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    strTest = strTest + "Ellipse" & vbCrLf
End Sub

Private Sub OnEllipsePrev(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    strTest = strTest + "Ellipse - Preview" & vbCrLf
End Sub
```

```

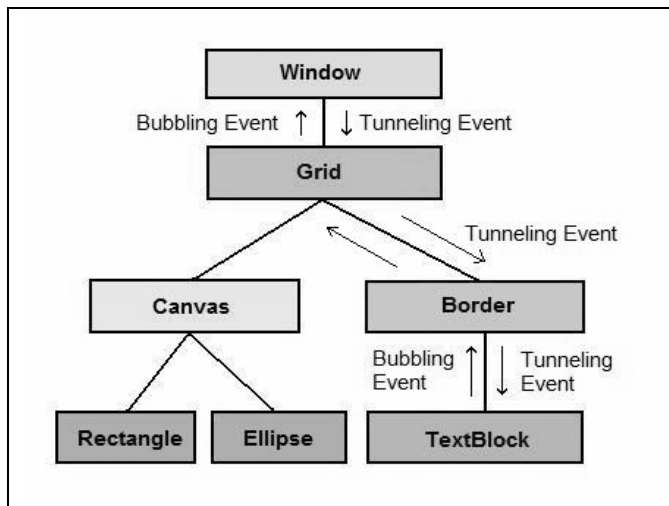
Private Sub OnRectangle(ByVal sender As Object, _
                        ByVal e As RoutedEventArgs)
    strTest = strTest + "Rectangle" & vbNewLine
End Sub

Private Sub OnRectanglePrev(ByVal sender As Object, _
                            ByVal e As RoutedEventArgs)
    strTest = strTest + "Rectangle - Preview" & vbNewLine
End Sub
End Class
End Namespace

```

**Listing 12.2** Der Code zum Beispiel in Abbildung 12.2

Die gesamte logische Hierarchie des Beispiels mit einem MouseDown- und einem PreviewMouseDown-Ereignis können Sie in Abbildung 12.3 verfolgen.



**Abbildung 12.3** Ereignisse in der logischen Hierarchie

Nun können wir das Beispielprogramm endlich aufrufen (Abbildung 12.2). Wenn wir mit der Maus auf den Text in der rechten Grid-Zelle klicken und danach die Applikation beenden, finden wir in der Datei »C:\Test.txt« folgende Informationen:

```

Window – Preview
Grid – Preview
Border – Preview
TextBlock – Preview
TextBlock
Border
Grid
Window

```

Sie können sehen, dass zuerst das »Tunneling Event« ausgelöst wird. Der Durchlauf beginnt im Wurzelement der Hierarchie, also im Hauptfenster der Applikation. Dann läuft das Preview-Ereignis durch den logischen Baum bis zum TextBlock-Element, welches das Ereignis ausgelöst hat. Nun läuft das »Bubbling Event« wieder den Weg zurück bis hin zum Wurzelement. In jeder Ereignismethode könnten Sie das Ereignis bearbeiten. Sie müssen aber die Weitergabe des Ereignisses nicht programmieren, denn dies erledigt WPF für Sie. Das Ereignis läuft natürlich auch dann durch die logische Hierarchie, wenn einzelne Ereignismethoden nicht implementiert werden.

Im zweiten Beispiel wollen wir nun auf das gelbe Rechteck in der linken Grid-Zelle klicken und zwar so, dass wir in dem Bereich klicken, in dem auch darunter die rote Ellipse liegt (Abbildung 12.4). Das Ergebnis in der Datei »C:\Test.txt« lautet nun:

```
Window - Preview
Grid - Preview
Canvas - Preview
Rectangle - Preview
Rectangle
Canvas
Grid
Window
```



Abbildung 12.4 Der zweite Versuch mit den Ereignissen

Hier wird das »Tunneling Event« zuerst vom Hauptfenster aus bis zum grafischen Element Rectangle durchlaufen. Wie Sie sehen, wird die Ereignismethode `OnEllipsePrev` des `Ellipse`-Elements nicht aufgerufen, da diese Ellipse in der logischen Hierarchie auf der gleichen Ebene wie das Rechteck liegt. Beide grafischen Objekte liegen im gleichen Canvas-Element. Auch hier wird dann das »Bubbling Event« durchlaufen, welches schließlich im Hauptfenster ankommt.

Welche Ereignismethoden in der Hierarchie aufgerufen werden, kann letztendlich von Ihnen gesteuert werden. Jedes weitergeleitete Ereignis erhält beim Aufruf der Ereignismethode ein Objekt vom Typ `RoutedEventArgs` oder ein Objekt, welches von dieser Klasse abgeleitet ist:

```
Private Sub OnCanvasPrev(ByVal sender As Object, _
                        ByVal e As RoutedEventArgs)
    strTest = strTest + "Canvas - Preview" & vbNewLine
End Sub
```

Sie können die Eigenschaft `Handled` im Parameter `e` benutzen, um den weiteren Durchlauf eines Ereignisses durch die logische Hierarchie zu unterbrechen:

```
Private Sub OnCanvasPrev(ByVal sender As Object, _
                        ByVal e As RoutedEventArgs)
    strTest = strTest + "Canvas - Preview" & vbNewLine
    e.Handled = True
End Sub
```

Als Standard wird in einer Ereignismethode für die Eigenschaft `Handled` der Wert `False` zurückgegeben. In diesem Fall wird das Ereignis weitergeleitet, wie wir es in den vorangegangenen Beispielen gesehen haben. Setzen Sie nun den Wert von `Handled` auf `True`, ist das Ereignis sozusagen »erledigt« und die Ereignisweiterleitung wird an dieser Stelle unterbrochen. Als Beispiel wollen wir die Eigenschaft `Handled` im Ereignis `PreviewMouseDown` des `Canvas`-Elements auf `True` setzen. Nun werden die in der Hierarchie darunter liegenden `Preview`-Eignismethoden und alle Methoden des `Bubbling Events` nicht aufgerufen, wie Sie an der ausgegebenen Test-Datei sehen können:

```
Window - Preview
Grid - Preview
Canvas - Preview
```

Direkte Ereignisse (`Direct Events`) werden nur für das Element verarbeitet, in dem sie auch ausgelöst wurden. In die Kategorie der direkten Ereignisse zählen z. B. das `MouseEnter`- oder das `MouseLeave`-Ereignis. Diese Ereignisse sind im Normalfall nur für das auslösende Element von Interesse.

Wenn Sie sich die Eigenschaften der Klasse `RoutedEventArgs` anschauen, werden Sie dort jedoch einige wichtige Informationen vermissen. Da wir ein Mausereignis verarbeitet haben, wollen wir meistens auch die Klickposition auswerten. Die Lösung des Problems ist sehr einfach. Wir müssen als zweiten Parameter in der Ereignismethode ein Element vom Typ `MouseEventArgs` annehmen. Listing 12.3 zeigt ein einfaches Fenster, welches wieder die beiden Ereignisse `MouseDown` und `PreviewMouseDown` implementiert (Listing 12.4). Auch hier benutzen wir im Code zunächst einen `String` in den Ereignismethoden, um die anfallenden Informationen zu sammeln. Da wir nun wissen, in welcher Reihenfolge die Ereignisse auftreten, können wir im »Bubbling Event« eine `MessageBox` mit dem Ergebnis anzeigen. Zusätzlich benötigen wir im Beispiel ein einfaches Rechteck, welches in einem `Canvas`-Element platziert wird.

Im `MouseEventArgs`-Objekt `e` gibt es allerdings immer noch keine `X`- und `Y`-Eigenschaften für die Position der Maus. Hier gibt es stattdessen eine interessante Vereinfachung für die Positionsabfrage. Wir benutzen die Methode `GetPosition` aus dem `MouseEventArgs`-Objekt. Beim Aufruf von `GetPosition` müssen Sie als einzigen Parameter ein Objekt in der Benutzerschnittstelle angeben, zu dem die Mauskoordinaten dann relativ umgerechnet werden sollen. Die Ausgabe der Koordinaten bezieht sich nun auf die obere linke Ecke des Hauptfensters. Im »Bubbling Event« beziehen wir uns in der Methode `GetPosition` auf das Rechteck mit dem Objektnamen `rect`. Der Unterschied der beiden Aufrufe ist im Ausgabebetext in der `MessageBox` leicht erkennbar.

```
<Window x:Class="EventArgs.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="EventArgs" Height="200" Width="250"
    MouseDown="OnWindow"
```

```
PreviewMouseDown="OnWindowPrev"  
>  
<Grid>  
  <Canvas>  
    <Rectangle Canvas.Left="30" Canvas.Top="30" Width="150" Height="100"  
      Fill="Red" Stroke="Black" Name="rect"/>  
  </Canvas>  
</Grid>  
</Window>
```

**Listing 12.3** Benutzung der MouseEventArgs-Klasse

```
Imports System  
Imports System.Windows  
Imports System.Windows.Input  
  
Namespace EventArgs  
  Partial Public Class Window1  
    Inherits System.Windows.Window  
    Private strTest As String = String.Empty  
  
    Public Sub New()  
      InitializeComponent()  
    End Sub  
  
    Public Sub OnWindow(ByVal sender As Object, _  
                       ByVal e As MouseEventArgs)  
      Dim pt As New Point  
      pt = e.GetPosition(Me)  
      strTest = strTest + "Bubble X: " &  
        & pt.X.ToString & "  
      "Y: " & pt.Y.ToString & vbNewLine  
      MessageBox.Show(strTest)  
    End Sub  
  
    Public Sub OnWindowPrev(ByVal sender As Object, _  
                            ByVal e As MouseEventArgs)  
      Dim pt As New Point  
      pt = e.GetPosition(rect)  
      strTest = strTest + "Tunnel X: " &  
        & pt.X.ToString & "  
      "Y: " & pt.Y.ToString & vbNewLine  
    End Sub  
  End Class  
End Namespace
```

**Listing 12.4** Benutzung der Klasse MouseEventArgs

Abschließend sei noch gesagt, dass nicht alle weitergeleiteten Ereignisse bis zum obersten Wurzel-Element der Anwendung laufen. So ist für das `MouseDown`-Ereignis z.B. ein `Button`-Element eine Schranke, die das Ereignis nicht ohne weiteres überwinden kann. Dadurch laufen bestimmte Ereignisse nicht sinnloserweise durch die gesamte Objekthierarchie.

## Weitergeleitete Befehle (Routed Commands)

Ein anderes wichtiges Konzept in WPF sind die weitergeleiteten Befehle (Routed Commands). Hier geht es um das Problem, dass eine Methode mit mehreren Elementen aus der Benutzerschnittstelle verbunden werden soll. Denken Sie z. B. an den Fall, dass Sie einen Menüpunkt und eine Schaltfläche in der Symbolleiste der Anwendung mit einer einzigen Ereignismethode verbinden wollen. Dies ist im Grunde genommen einfach zu realisieren, indem wir das Klickereignis der beiden Elemente mit der gleichen Methode im Code verbinden.

Nun wollen wir noch einen Schritt weiter gehen. Es ist häufig erforderlich, die beiden Elemente der Benutzerschnittstelle (Menüpunkt und Schaltfläche in der Symbolleiste) in bestimmten Situationen ein- oder auszuschalten. Normalerweise erledigen wir die Einstellungen für das Menü erst dann, wenn das Pop-up-Menü heruntergeklappt werden soll. Dafür gibt es auch die entsprechenden Ereignisse, die wir benutzen können. Nun gibt es aber ein Problem, denn die Schaltfläche in der Symbolleiste muss sofort bearbeitet werden, wenn sich der Status für die Benutzerschnittstelle ändert. Bauen wir also die Aktivierung und Deaktivierung der Schaltfläche in die gleiche Methode ein, welche die Logik für den Menüpunkt enthält, so wird die Schaltfläche zu spät bearbeitet. Wir müssen also eine eigene Behandlungsmethode für die Elemente in der Symbolleiste programmieren. Das ist natürlich keine gute Lösung.

Hier kommen nun die Routed Commands ins Spiel. Sie können mit diesem WPF-Konzept verschiedene Elemente der Benutzerschnittstelle mit Ereignismethoden verbinden, die dann auch entsprechend zeitnah aufgerufen werden.

Wir wollen das in einem Beispiel vertiefen. Im Hauptfenster der Applikation gibt es ein Menü in einem DockPanel-Element mit zwei Menüpunkten *Neu* und *Beenden*. Zusätzlich haben wir mitten im Fenster eine Schaltfläche *Neu*, die an die gleiche Ereignismethode gebunden werden soll wie der Menüpunkt. Der dazugehörige XAML-Code ist in Listing 12.5 zu sehen.

```
<Window x:Class="Commands.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Commands" Height="200" Width="300"
  >
  <DockPanel>
    <Menu DockPanel.Dock="Top">
      <MenuItem Header="Datei">
        <MenuItem Command="New" Header="Neu..." />
        <Separator />
        <MenuItem Click="OnBeenden" Header="Beenden" />
      </MenuItem>
    </Menu>

    <Button Command="New" Width="100" Height="25">Neu...</Button>
  </DockPanel>
</Window>
```

**Listing 12.5** Benutzerschnittstelle für das Beispiel mit Routed Commands

Was hier auffällt, ist die Eigenschaft `Command`, die im ersten `MenuItem`- und im `Button`-Element verwendet wird. Für den zweiten Menüpunkt *Beenden* wird dagegen eine ganz normale Ereignismethode für das Klickereignis implementiert.

Hinter der Eigenschaft `Command` steckt nun ein so genannter »Routed Command«. Wir wollen darum zunächst den VB-Code in Listing 12.6 anschauen.

```
Imports System
Imports System.Windows
Imports System.Windows.Input

Namespace Commands
    Partial Public Class Window1
        Inherits System.Windows.Window
        Private strDokument As String = "Das ist ein Test-Text!"
        Public Sub New()
            InitializeComponent()

            ' CommandBinding-Objekt anlegen und Ereignismethoden zuweisen
            Dim commandNew As New CommandBinding(ApplicationCommands.[New])
            AddHandler commandNew.Executed, AddressOf NeuHandler
            AddHandler commandNew.CanExecute, AddressOf NeuAusfuehrenHandler

            ' Neues CommandBinding hinzufügen
            Me.CommandBindings.Add(commandNew)
        End Sub

        Private Sub NeuHandler(ByVal sender As Object, _
                               ByVal e As ExecutedRoutedEventArgs)
            Dim strHelp As String = "Wollen Sie wirklich alles löschen?"
            Dim res As New MessageBoxResult
            res = MessageBox.Show(strHelp, "Test", MessageBoxButton.YesNo)
            If res = MessageBoxResult.Yes Then
                strDokument = String.Empty
            End If
        End Sub

        Private Sub NeuAusfuehrenHandler(ByVal sender As Object, _
                                         ByVal e As CanExecuteRoutedEventArgs)
            ' Darf der Befehl ausgeführt werden?
            e.CanExecute = strDokument.Length > 0
        End Sub

        Private Sub OnBeenden(ByVal sender As Object, _
                              ByVal e As RoutedEventArgs)
            Me.Close()
        End Sub
    End Class
End Namespace
```

**Listing 12.6** CommandBinding im Einsatz

Im Fensterobjekt wird eine `String`-Variable deklariert, in welcher sich ein kurzer Text befindet. Dieser Text soll hier im Beispiel unser Dokument darstellen. Das heißt, dass der `Neu`-Befehl als Menüpunkt und als Schaltfläche nur dann aktiviert sein soll, wenn sich in der `String`-Variablen auch tatsächlich ein Text befindet. Ist die `String`-Variable leer, darf der `Neu`-Befehl nicht ausgeführt werden. Hierzu benutzen wir ein Objekt der `CommandBinding`-Klasse, welches wir im Konstruktor des Hauptfensters anlegen. Bei der Instanziierung des Objekts wird im Konstruktor als Parameter der Wert `ApplicationCommands.[New]` übergeben. In WPF gibt es verschiedene, vordefinierte `CommandBinding`-Klassen, die Sie in Ihren Applikationen sofort benutzen können.

---

**WICHTIG** Es gibt folgende Grundklassen für Routed Commands in WPF:

- `ApplicationCommands`
- `New, Open, Save, SaveAs, Print, PrintPreview, Copy, Cut, Paste, Replace, SelectAll,...`
- `ComponentCommands`
- `MoveDown, MoveUp, MoveLeft, MoveRight, ScrollPageDown, ScrollPageUp,...`
- `EditingCommands`
- `AlignCenter, AlignJustify, Backspace, Delete, EnterLineBreak, MoveDownByLine,...`
- `MediaCommands`
- `BoostBass, ChannelUp, ChannelDown, FastForward, MuteVolume, NextTrack, Play, Stop,...`

Die obige Liste enthält nicht alle Befehle der jeweiligen `Commands`-Klassen. Die Klasse `ApplicationCommands` enthält als statische Eigenschaften für die diversen Befehle, die normalerweise im Menü einer Applikation zu finden sind. In der Klasse `ComponentCommands` finden Sie Befehle, die mit Komponenten zu tun haben. `EditingCommands` enthält die Befehle, die mit dem Ändern von Dokumenten zu tun haben. Schließlich gibt es noch die Klasse `MediaCommands`, die Befehle enthält, die mit Medien-dateien arbeiten.

Die vier `Commands`-Klassen enthalten nicht den Code, um die verschiedenen Befehle auszuführen (z. B. einen Text zu kopieren oder eine Datei zu speichern), sondern nur statische Eigenschaften, die es Ihnen ermöglichen, mehrere Elemente der Benutzerschnittstelle unter die Kontrolle einer Ereignismethode zu stellen.

Wenn Ihnen die vorgegebenen Befehle in den vier `Commands`-Klassen nicht ausreichen, können Sie eigene Befehlsklassen erzeugen.

---

Nach der Erzeugung des `CommandBinding`-Objekts in unserem Beispiel können Sie zwei Ereignismethoden definieren, die einerseits den Code aufrufen, der ausgeführt werden soll, wenn der Menüpunkt oder die Schaltfläche angeklickt werden (Ereignis: `Executed`) und andererseits die Steuerung für die Aktivierung und Deaktivierung der Elemente der Benutzerschnittstelle übernehmen (Ereignis: `CanExecute`). Diese Ereignisse, die hier ausgelöst werden, sind übrigens ganz normale Routed Events, die die logische Hierarchie der Benutzerschnittstelle durchlaufen.

Schließlich fügen wir das erzeugte `CommandBinding`-Objekt mit der Methode `Add` in die Liste der `CommandBindings` des Elternelements ein. Nun »weiß« unser Hauptfenster von den Bindungen der Schaltfläche und des Menüpunktes und kann diese bei Bedarf durch Aufruf der Ereignismethoden ausführen.

Die Ereignismethoden selbst erhalten ganz normalen Code. In der Methode `NeuHandler` wird der Anwender gefragt, ob er das Dokument (der `string strDokument`) löschen will. Wenn ja, wird die `String`-Variable geleert.

In der Methode `NeuAusfuehrenHandler` muss die Eigenschaft `CanExecute` des `CanExecuteRoutedEventArgs`-Objekt, der als Parameter übergeben wird, auf `True` gesetzt werden, wenn der Befehl ausgeführt werden darf. Ansonsten wird hier `False` zurückgegeben. Im Beispiel prüfen wir hierzu, ob die Länge des Textes in `strDokument` größer als Null ist.

Wenn wir die Applikation nun starten (Abbildung 12.5), ist der *Neu*-Befehl sowohl als Menüpunkt, sowie als Schaltfläche aktiviert, da in der String-Variable ein Text steht.

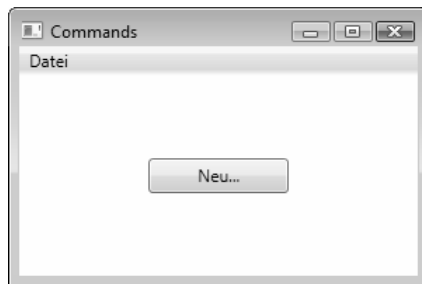


Abbildung 12.5 Nach dem Start des `CommandBindings`-Beispiels

Klicken Sie nun auf die Schaltfläche *Neu*. Die Abfrage, ob Sie löschen wollen, beantworten Sie mit »Ja«. Sobald das `MessageBox`-Fenster geschlossen wird, ändert sich auch der Status in der Benutzerschnittstelle. Die Schaltfläche und der Menüpunkt werden deaktiviert und der Befehl *Neu* kann nicht mehr ausgeführt werden.

Die Ereignismethode `NeuHandler` wird natürlich nur dann aufgerufen, wenn der Befehl selbst über das Menü oder die Schaltfläche ausgelöst wird. Es stellt sich jedoch die Frage, wie häufig die Methode `NeuAusfuehrenHandler` aufgerufen wird. Um dies zu prüfen, können wir in diese Ereignismethode zusätzlichen Code einbauen, der jedes Mal einen kleinen Text im Ausgabe-Fenster von Visual Studio ausgibt. Wir benutzen in diesem Fall kein `MessageBox`-Element für die Datenausgabe, um den Fluss der Ereignisse nicht zu ändern. Zuerst werden wir feststellen, dass die Ereignismethode nicht einfach regelmäßig aufgerufen wird (Polling). Die Ereignismethoden für die Aktivierung und Deaktivierung der Elemente werden immer dann aufgerufen, wenn »irgendwie etwas Wichtiges« in Ihrer Benutzerstelle passiert. So wird die Methode `NeuAusfuehrenHandler` z.B. aufgerufen, wenn Sie in das Hauptfenster klicken, wenn Sie Popup-Menüs herunterklappen, wenn Sie auf die Schaltfläche *Neu* klicken oder wenn die Applikation gestartet wird. Die Ereignismethode wird nicht aufgerufen, wenn Sie z.B. den Mauszeiger durch das Fenster oder über die Schaltfläche bewegen, ohne mit der Maus zu klicken.

Um alle Aktivierungsmethoden von Hand zu starten, müssen Sie die statische Methode `InvalidateRequerySuggested` aus der Klasse `CommandManager` aufrufen:

```
CommandManager.InvalidateRequerySuggested()
```

Ein solcher »außerordentlicher« Aufruf kann dann wichtig sein, wenn ein separater Thread bestimmte Rechenarbeiten erledigt hat, welche den Status der Benutzerschnittstelle beeinflussen. Beim Beenden eines Threads wird nicht automatisch die Methode `NeuAusfuehrenHandler` aufgerufen.

Das bedeutet natürlich, dass der Code in den Ereignismethoden für die Aktivierung und Deaktivierung der Steuerelemente möglichst kurz sein sollte, damit er sehr schnell ausgeführt werden kann. Sie müssen bedenken, dass es in der Applikation mehrere `CommandBindings` im Hauptfenster geben kann. In den oben skizzierten Fällen werden dann natürlich alle Ereignismethoden aufgerufen, die am entsprechenden Elternelement gebunden sind.

Weitergeleitete Ereignisse unterstützen die Trennung von Logik und Design. Der Softwareentwickler implementiert Befehle mit ganz bestimmten Namen. Der Designer kann diese Befehle mit den Steuerelementen in der Benutzerschnittstelle verbinden, ohne dass er einen Namen für eine spezielle Ereignismethode vergeben muss. Er benutzt einfach den Namen des gewünschten Befehls:

```
<MenuItem Command="New" Header="Neu..." />
<Button Command="Copy">Kopieren</Button>
```

## Eigenschaften der Abhängigkeiten

Kommen wir nun zu den Eigenschaften der Abhängigkeiten (Dependency Properties) bei den Steuerelementen von Windows Presentation Foundation.

Eigenschaften werden in der logischen Hierarchie von WPF weiter vererbt. Wird z.B. in einem `StackPanel`-Element eine bestimmte Schriftartgröße (Eigenschaft `FontSize`) gesetzt, so wird diese an alle Elemente weitergegeben, die in diesem Layout-Element enthalten sind. Dabei wird auch das Layout der Elemente neu berechnet, denn durch eine größere oder kleinere Schriftart für die Kindelemente im `StackPanel` können sich natürlich auch die Abmessungen und Positionen ändern (sofern es das Elternelement erlaubt). Das folgende Beispiel soll die Möglichkeiten, die sich durch die Abhängigkeitseigenschaften ergeben, demonstrieren (Listing 12.7 und Listing 12.8).

```
<Window x:Class="Depend.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Depend" Height="200" Width="650"
  >
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>

    <Button Margin="5" Tag="8" Grid.Row="0" Grid.Column="0" Click="WindowOnClick">Window: 8pt</Button>
    <Button Margin="5" Tag="16" Grid.Row="0" Grid.Column="1" Click="WindowOnClick">Window: 16pt</Button>
    <Button Margin="5" Tag="32" Grid.Row="0" Grid.Column="2" Click="WindowOnClick">Window: 32pt</Button>
```

```
<Button Margin="5" Tag="8" Grid.Row="1" Grid.Column="0" Click="ButtonOnClick">Button: 8pt</Button>
<Button Margin="5" Tag="16" Grid.Row="1" Grid.Column="1" Click="ButtonOnClick">Button: 16pt</Button>
<Button Margin="5" Tag="32" Grid.Row="1" Grid.Column="2" Click="ButtonOnClick">Button: 32pt</Button>
</Grid>
</Window>
```

**Listing 12.7** Test der Eigenschaften von Abhängigkeiten

```
Imports System
Imports System.Windows
Imports System.Windows.Controls
Imports System.Windows.Input

Namespace Depend
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
        End Sub

        Public Sub WindowOnClick(ByVal sender As Object, _
                                ByVal e As RoutedEventArgs)
            'FontSize-Eigenschaften des Fensters ändern
            Dim btn As New Button
            btn = CType(e.Source, Button)
            Me.FontSize = Convert.ToDouble(btn.Tag)
        End Sub

        Private Sub ButtonOnClick(ByVal sender As Object, _
                                  ByVal e As RoutedEventArgs)
            'FontSize-Eigenschaft einer Schaltfläche ändern
            Dim btn As New Button
            btn = CType(e.Source, Button)
            btn.FontSize = Convert.ToDouble(btn.Tag)
        End Sub
    End Class
End Namespace
```

**Listing 12.8** Die Schriftartgröße wird als Eigenschaft einer Abhängigkeit geändert

In diesem Beispiel werden sechs Schaltflächen dargestellt, welche die Schriftgrößen im Fenster oder auf den Schaltflächen selbst ändern können. Der Code für die Änderung der Schriftgrößen wurde in den Klick-Ereignismethoden `WindowOnClick` und `ButtonOnClick` untergebracht. Dort wird zunächst mit Hilfe des Methodenparameters `sender` die Schaltfläche ermittelt, auf die geklickt wurde. In der Eigenschaft `Tag` der einzelnen Schaltflächen ist die einzustellende Schriftgröße hinterlegt, die dann konvertiert und entweder für das gesamte Fenster oder nur für die jeweilige Schaltfläche gesetzt wird. Nach dem Start des Programms haben alle Schaltflächen die gleiche Schriftgröße (Abbildung 12.6).



Abbildung 12.6 Start des Testprogramms

Die Schaltflächen erben in diesem Beispiel die Schriftgröße des Elternelements, also des umgebenden Fensters. Klicken Sie nun auf die Schaltfläche *Button: 8pt*. Es wird nur die Schriftgröße dieser einen Schaltfläche geändert (Abbildung 12.7).

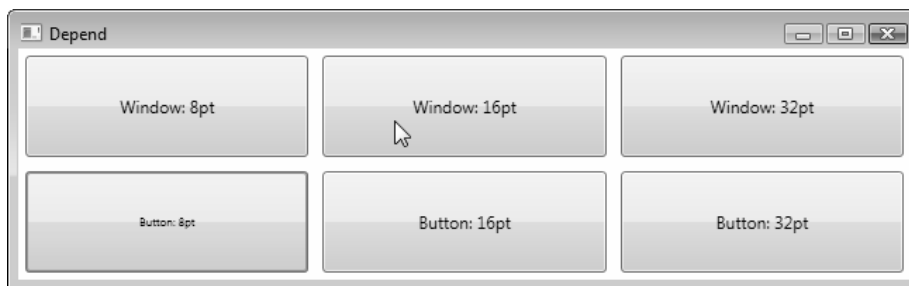


Abbildung 12.7 Schaltfläche mit kleiner Schriftgröße

Im nächsten Test klicken wir nun auf die Schaltfläche *Window: 16pt*. Nun sollte für das gesamte Fenster die Schriftgröße 16pt verwendet werden, außer für die Schaltfläche, die wir eben im ersten Test auf die kleine Schriftgröße (8pt) gesetzt haben (Abbildung 12.8):



Abbildung 12.8 Das Fenster mit 16pt-Schriftgröße

Die eingestellte Schriftgröße wurde bei diesem Versuch durch die gesamte logische Hierarchie der Benutzerschnittstelle weiter vererbt. Da für die Schaltfläche *Button: 8pt* bereits explizit eine andere Schriftgröße gesetzt wurde, ist an dieser Stelle die Vererbung unterbrochen worden. Klicken Sie nun noch auf die Schaltfläche *Button: 32pt* und Sie werden sehen, dass nur die Schriftart dieser Schaltfläche vergrößert wird (Abbildung 12.9).

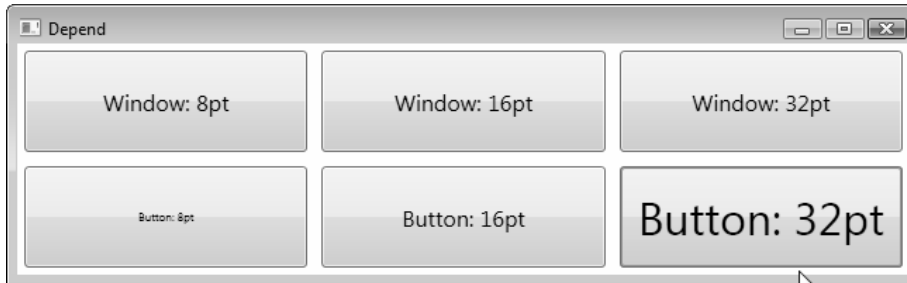


Abbildung 12.9 Der letzte Test

Ein weiterer Klick auf die Schaltfläche *Window: 8pt* würde nun durch die Vererbung der Eigenschaften alle Schaltflächen in der oberen Reihe und die Schaltfläche *Button: 16pt* in der unteren Reihe des Fensters ändern.

Da im letzten Beispiel ein `Grid`-Element benutzt wurde, um die Schaltfläche zu positionieren, hat sich die Größe der Kindelemente bei der Änderung der Schriftgröße nicht geändert. Sie können natürlich andere Szenarien definieren, bei denen die Größe der Kindelemente durch eine Layout-Berechnung von WPF neu ermittelt wird.

Wenn Sie die Vererbung einer Eigenschaft durch das explizite Setzen dieser Eigenschaft »unterbrochen« haben, wird der vorgegebene Wert für die Eigenschaft dieses Steuerelements benutzt und natürlich von dort an andere Steuerelemente, die sich in der Hierarchie darunter befinden, weitergegeben. Sie können das explizite Setzen der Eigenschaft wieder aufheben, indem Sie die Methode `ClearValue` anwenden:

```
Me.ClearValue(Window.FontSizeProperty)
```

Mit der obigen Codezeile können Sie die Eigenschaft `FontSize` wieder durch die gesamte Hierarchie, einschließlich des Steuerelements `Me`, vererben.

## Eingaben

Eingaben können mit der Maus, über die Tastatur und mit dem Stift gemacht werden. Die Mauseingaben werden immer zunächst zu dem Element der Benutzerschnittstelle geleitet, das sich genau unter dem Mauszeiger befindet. Handelt es sich um ein Routed Event, läuft das Ereignis nun durch die logische Hierarchie und wird ggf. in den verschiedenen Ereignismethoden abgearbeitet. Ein direktes Ereignis kann nur Element-spezifisch verarbeitet werden.


Maus-Ereignis	Weiterleitung	Aktion
<code>GotMouseCapture</code>	Bubbling	Das Element hält den Mausfokus fest
<code>LostMouseCapture</code>	Bubbling	Das Element hat den Mausfokus verloren
<code>MouseEnter</code>	Direkt	Der Mauszeiger wird in das Element hinein bewegt
<code>MouseLeave</code>	Direkt	Der Mauszeiger wird aus dem Element hinaus bewegt
<code>MouseDown</code> , <code>PreviewMouseDown</code>	Bubbling, Tunneling	Eine Maustaste wurde gedrückt
<code>MouseUp</code> , <code>PreviewMouseUp</code>	Bubbling, Tunneling	Eine Maustaste wurde losgelassen ▶

Maus-Ereignis	Weiterleitung	Aktion
MouseMove, PreviewMouseMove	Bubbling, Tunneling	Die Maus wurde bewegt
MouseWheel, PreviewMouseWheel	Bubbling, Tunneling	Das Mousrad wurde gedreht
MouseLeftButtonDown, PreviewMouseLeftButtonDown	Bubbling, Tunneling	Die linke Maustaste wurde gedrückt
MouseLeftButtonUp, PreviewMouseLeftButtonUp	Bubbling, Tunneling	Die linke Maustaste wurde losgelassen
MouseRightButtonDown, PreviewMouseRightButtonDown	Bubbling, Tunneling	Die rechte Maustaste wurde gedrückt
MouseRightButtonUp, PreviewMouseRightButtonUp	Bubbling, Tunneling	Die rechte Maustaste wurde losgelassen
QueryCursor	Bubbling	Ein Element fragt nach dem momentanen Mauscursor

**Tabelle 12.1** Maus-Ereignisse

Beim Programmieren mit der Maus gibt es noch eine interessante Eigenschaft, die aus der Klasse `UIElement` kommt, von der jedes WPF-Steuerelement abgeleitet ist. Die Eigenschaft heißt `IsMouseOver` und liefert einen booleschen Wert zurück. Die Eigenschaft `IsMouseOver` gibt immer dann der Wert `True` zurück, wenn sich der Mauszeiger über dem Steuerelement oder einem seiner sichtbaren Kindelemente befindet.

Bitte beachten Sie, dass das schon oft verwendete `Click`-Ereignis in Tabelle 12.1 der Maus-Ereignisse nicht enthalten ist. Ein `Click`-Ereignis kann nicht nur durch die Maus ausgelöst werden, sondern auch durch eine Tastatur. Außerdem können sich hinter einem `Click`-Ereignis mehrere Maus-Ereignisse verbergen (`MouseUp`, `MouseDown`). Darum finden Sie in der Klasse `Control` die beiden zusätzlichen Ereignisse `MouseDoubleClick` und `PreviewMouseDoubleClick`. Die Klasse `ButtonBase` implementiert dann das `Click`-Ereignis. `ButtonBase` ist die Basisklasse der Klassen `Button`, `RadioButton` und `CheckBox`.

Auch für die Tastatur stehen diverse Ereignisse zur Verfügung (Tabelle 12.2). Hier spielt das Konzept des Eingabefokus eine entscheidende Rolle. Nur das Steuerelement enthält Eingaben von der Tastatur, welches den Eingabefokus hat. Der Fokus kann durch Anklicken mit der Maus, durch die -Taste oder durch Navigieren mit den Cursor-Tasten zu einem bestimmten Steuerelement gebracht werden.

Tastatur-Ereignis	Weiterleitung	Aktion
GotFocus, PreviewGotFocus	Bubbling, Tunneling	Ein Element erhält den Eingabefokus
LostFocus, PreviewLostFocus	Bubbling, Tunneling	Ein Element verliert den Eingabefokus
KeyDown, PreviewKeyDown	Bubbling, Tunneling	Eine Taste wird gedrückt
KeyUp, PreviewKeyUp	Bubbling, Tunneling	Eine Taste wird losgelassen
TextInput, PreviewTextInput	Bubbling, Tunneling	Ein Element empfängt eine Texteingabe

**Tabelle 12.2** Tastatur-Ereignisse

In Tabelle 12.3 können Sie schließlich die Ereignisse für den Stift des Tablett-PCs sehen.

Stift-Ereignis	Weiterleitung	Aktion
GotStylusCapture	Bubbling	Das Element hält den Stiffokus fest
LostStylusCapture	Bubbling	Das Element verliert den Stiffokus
StylusDown, PreviewStylusDown	Bubbling, Tunneling	Der Stift berührt den Bildschirm über einem Element
StylusUp, PreviewStylusUp	Bubbling, Tunneling	Der Stift wird vom Bildschirm über einem Element hoch gehoben
StylusEnter	Direkt	Der Stift wird in das Element hinein bewegt
StylusLeave	Direkt	Der Stift wird aus dem Element hinaus bewegt
StylusInRange, PreviewStylusInRange	Bubbling, Tunneling	Der Stift befindet sich dicht über dem Bildschirm
StylusOutOfRange, PreviewStylusOutOfRange	Bubbling, Tunneling	Der Stift befindet sich außerhalb des Bildschirmfassungsabstands
StylusMove, PreviewStylusMove	Bubbling, Tunneling	Der Stift wird über das Element bewegt
StylusInAirMove, PreviewStylusInAirMove	Bubbling, Tunneling	Der Stift wird über das Element bewegt, berührt dabei aber nicht den Bildschirm
StylusSystemGesture, PreviewStylusSystemGesture	Bubbling, Tunneling	Es wird eine Stift-Geste erkannt
TextInput, PreviewTextInput	Bubbling, Tunneling	Das Element empfängt eine Texteingabe

**Tabelle 12.3** Stift-Ereignisse

## Schaltflächen

Schaltflächen sind einfache Steuerelemente, die angeklickt werden können. Hierbei unterscheiden wir verschiedene Typen: Neben der Standard-Schaltfläche (Button) gibt es in WPF auch noch die Steuerelemente Kontrollkästchen (CheckBox) und Auswahlkästchen (RadioButton). Die drei Klassen sind von ButtonBase abgeleitet und stellen das Click-Ereignis zur Verfügung. Die Anwendung dieser Elemente ist sehr einfach:

```
<Button Click="OnButtonClicked" Name="btn">Button</Button>
<CheckBox Click="OnCheckboxClicked" Name="check">CheckBox</CheckBox>
<RadioButton Click="OnRadiobuttonClicked" Name="radio">RadioButton</RadioButton>
```

Die Methode für das Click-Ereignis der Schaltfläche kann folgendermaßen implementiert werden:

```
Private Sub OnButtonClick(ByVal sender As Object, ByVal e As RoutedEventArgs)
    MessageBox.Show("Schaltfläche geklickt!")
End Sub
```

Entsprechende Methoden können auch für die beiden anderen Steuerelemente erstellt werden. Wenn Sie aus dem Code auf die Steuerelemente zugreifen wollen, müssen diese mit einem Namen versehen werden.

Die drei Schaltflächenarten können nicht nur einen einfachen Text enthalten, sondern sie können beliebige andere Elemente, auch Grafik, darstellen. Im folgenden Listing 12.9 werden die drei Elemente mit einem erweiterten Inhalt dargestellt.

```
<Window x:Class="ButtonTest.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Test mit Schaltflächen" Height="200" Width="300"
  >
  <Grid>
    <StackPanel Width="150">

      <Button>
        <TextBlock>
          <Ellipse Margin="0,0,5,0" Width="20" Height="10" Fill="Red" Stroke="Black" />
          Button
          <Ellipse Margin="5,0,0,0" Width="20" Height="10" Fill="Green" Stroke="Black" />
        </TextBlock>
      </Button>

      <CheckBox>
        <TextBlock Foreground="White" FontSize="18">
          <TextBlock.Background>
            <LinearGradientBrush StartPoint="0,0" EndPoint="1,0">
              <GradientStop Color="Blue" Offset="0" />
              <GradientStop Color="Red" Offset="1" />
            </LinearGradientBrush>
          </TextBlock.Background>
          CheckBox
        </TextBlock>
      </CheckBox>

      <RadioButton>
        <TextBlock FontSize="20" FontFamily="Algerian">
          RadioButton
        </TextBlock>
      </RadioButton>

    </StackPanel>
  </Grid>
</Window>
```

**Listing 12.9** Steuerelemente mit erweitertem Inhalt

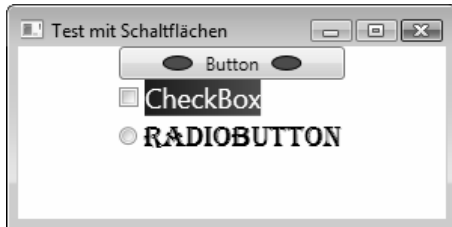


Abbildung 12.10 Steuerelemente mit erweitertem Inhalt

In Abbildung 12.10 sehen Sie das Ergebnis unserer Programmierung. Einige der oben benutzten WPF-Elemente werden Sie in den folgenden Kapiteln noch näher kennen lernen. Das Beispiel soll Ihnen hier nur zeigen, dass es mit WPF sehr einfach ist, auch komplexe grafische Effekte mit den vorhandenen Steuerelementen zu nutzen.

Für alle Schaltflächenarten wird in Listing 12.9 ein TextBlock-Element verwendet, um eine weitere Gestaltung und Formatierung durchzuführen. Die normale Schaltfläche wird mit grafischen Grundelementen (Ellipsen) erweitert. Für das CheckBox-Element wird ein neuer Hintergrund mit einem Farbverlauf (LinearGradientBrush) definiert und für das RadioButton-Element wurde eine andere Schriftart ausgewählt.

## Bildlaufleisten und Schieberegler

In WPF finden Sie Steuerelemente, mit denen Sie einen Wert aus einem vorgegebenen Bereich auswählen können. Zu diesen Steuerelementen gehören die Klassen `Slider` und `ScrollBar`. Die Benutzung der Slider-Elemente ist in Listing 12.10 dargestellt.

```
<Window x:Class="Schieber.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Schieberegler" Height="280" Width="300"
  >
  <Grid>
    <Canvas>
      <Slider Name="sldVert" Orientation="Vertical" Margin="15,15,15,15"
        Minimum="0" Maximum="100" Height="200" Value="0" ValueChanged="OnValueVert" />
      <Slider Name="sldHorz" Orientation="Horizontal" Margin="60,15,15,15"
        Minimum="0" Maximum="100" Width="200" Value="0" ValueChanged="OnValueHorz" />

      <Label Name="lblHorz" FontSize="15" Margin="120,30,15,15" />
      <Label Name="lblVert" FontSize="15" Margin="30,100,15,15" />
    </Canvas>
  </Grid>
</Window>
```

Listing 12.10 Slider-Elemente

```
Imports System
Imports System.Windows

Namespace Schieber
  Partial Public Class Window1
```

```
Inherits System.Windows.Window

Public Sub New()
    InitializeComponent()
End Sub

Private Sub OnValueHorz(ByVal sender As Object, ByVal e As RoutedEventArgs)
    lblHorz.Content = sldHorz.Value.ToString
End Sub

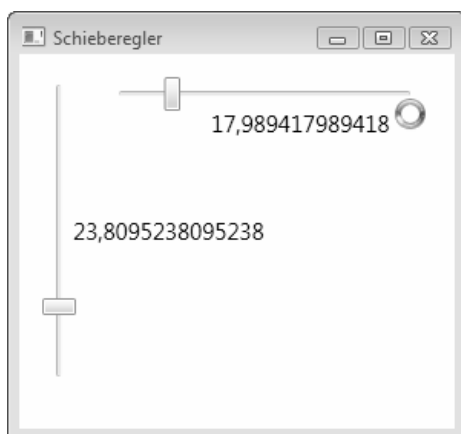
Private Sub OnValueVert(ByVal sender As Object, ByVal e As RoutedEventArgs)
    lblVert.Content = sldVert.Value.ToString
End Sub

End Class
End Namespace
```

**Listing 12.11** Ereignismethoden für die Schieberegler

Im Beispiel werden zwei Schieberegler deklariert, die in horizontaler und vertikaler Richtung verlaufen (Listing 12.10). Dies wird über die Eigenschaft `Orientation` eingestellt. Das hier verwendete Canvas-Element dient nur zur Positionierung aller Elemente und muss für unser Beispiel nicht weiter betrachtet werden. Für die Schieberegler werden jeweils ein `Minimum`- und ein `Maximum`-Wert festgelegt. Die Ausgaben der `Value`-Eigenschaft bewegen sich genau in diesem vorgegebenen Bereich. In unserem Beispiel wurden zusätzlich zwei `Label`-Elemente definiert, welche die Eigenschaft `Value` der beiden Schieberegler ausgeben.

Nun wollen wir zwei Ereignismethoden im Code implementieren (Listing 12.11). Das Ereignis `ValueChanged` der beiden Schieberegler wird auf die Methoden `OnValueHorz` und `OnValueVert` geleitet. Dort wird die Ausgabe der `Value`-Eigenschaft der Schieberegler in einen `String` konvertiert und im dazugehörigen `Label`-Element dargestellt. Die `Value`-Eigenschaft des `Slider`-Elements liefert übrigens einen Wert vom Typ `Double` zurück, wie Sie es beim Ausführen des Beispielprogramms sofort erkennen können (Abbildung 12.11).



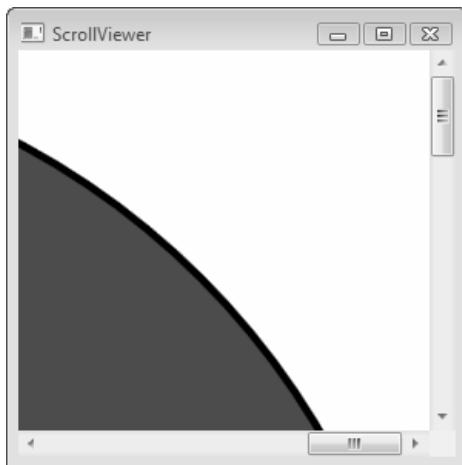
**Abbildung 12.11** Zwei Schieberegler im Einsatz

Mit den Bildlaufleisten (`ScrollBar`) können Sie ganz ähnlich wie mit den Schiebereglern (`Slider`) arbeiten. Bildlaufleisten bieten zusätzlich die Eigenschaften `LargeChange` und `SmallChange` an, mit denen Sie das Blättern durch ein größeres Dokument steuern können.

Ein sehr interessantes Steuerelement von WPF ist das `ScrollViewer`-Element, mit dem es sehr leicht möglich ist, einen Inhalt mit Bildlaufleisten darzustellen, wenn dieser nicht in das Elternelement passt (Listing 12.12). Der darzustellende Inhalt wird als Kindelement im `ScrollViewer`-Element deklariert. Ist das Kindelement größer als der verfügbare Platz, werden die benötigten Bildlaufleisten dargestellt (Abbildung 12.12) und können benutzt werden, ohne dass eine weitere Programmierung erforderlich ist.

```
<Window x:Class="Scroller.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="ScrollViewer" Height="300" Width="300"
  >
  <Grid>
    <ScrollViewer HorizontalScrollBarVisibility="Auto" VerticalScrollBarVisibility="Auto">
      <Ellipse Width="1000" Height="1000" Fill="Red" Stroke="Black" StrokeThickness="5" />
    </ScrollViewer>
  </Grid>
</Window>
```

**Listing 12.12** Viel Inhalt mit den `ScrollViewer`-Element



**Abbildung 12.12** Benutzung des `ScrollViewer`-Elements

Das `ScrollViewer`-Element kann in beliebigen Layout-Elementen eingesetzt werden. So können Sie z.B. die Zelle eines `Grid`-Elements sehr einfach mit Bildlaufleisten versehen. Auch die Ausgabe von Texten kann von einem `ScrollViewer`-Element unterstützt werden. Hierbei ist es meistens sinnvoll, die horizontale Bildlaufleiste auszuschalten und nur die senkrechte Leiste zu benutzen. Listing 12.13 zeigt ein Beispiel mit einem `Grid`-Element, bestehend aus zwei Zeilen und zwei Spalten, in dem eine Ellipse, ein Bild und ein Text in den einzelnen Zellen ausgegeben werden. Für die Ausgabe des Textes in der unteren `Grid`-Zeile wird die horizontale Bildlaufleiste ausgeschaltet und der Zeilenumbruch wird für das `TextBlock`-Element aktiviert.

Versuchen Sie nach dem Start des Beispielprogramms das Fenster zu vergrößern. Sobald der Inhalt vollständig in einer `Grid`-Zelle dargestellt werden kann, verschwinden die Bildlaufleisten (Abbildung 12.13).

```

<Window x:Class="Scroller2.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="ScrollViewer" Height="300" Width="300"
  >
  <Grid>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>

    <ScrollViewer Grid.Column="0" Grid.Row="0" HorizontalScrollBarVisibility="Auto"
      VerticalScrollBarVisibility="Auto">
      <Ellipse Width="200" Height="200" Fill="Red" Stroke="Black" StrokeThickness="3" />
    </ScrollViewer>

    <ScrollViewer Grid.Column="1" Grid.Row="0" HorizontalScrollBarVisibility="Auto"
      VerticalScrollBarVisibility="Auto">
      <Image Source="IC5146.Jpg" />
    </ScrollViewer>

    <ScrollViewer Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="2" VerticalScrollBarVisibility="Auto"
      HorizontalScrollBarVisibility="Disabled" >
      <TextBlock TextWrapping="Wrap" FontSize="20">
        Hier werden mehrere ScrollViewer-Elemente von Windows Presentation Foundation eingesetzt.
        Es ist nun sehr einfach möglich, große Elemente mit Bildlaufleisten darzustellen.
      </TextBlock>
    </ScrollViewer>
  </Grid>
</Window>

```

**Listing 12.13** ScrollViewer-Elemente in einem Grid



**Abbildung 12.13** Mehrere ScrollViewer-Elemente

## Steuerelemente für die Texteingabe

Das einfachste Steuerelement für die Texteingabe ist das `TextBox`-Element, mit dem Sie normalerweise eine Zeile Text eingeben können. Durch Setzen der Eigenschaft `AcceptsReturn` auf `True` ist es jedoch möglich, mehrere Textzeilen einzugeben. Den eingegebenen Text können Sie über die Eigenschaft `Text` abfragen oder vorgeben.

Eine weitere Variante bei der Texteingabe stellt das `PasswordBox`-Element dar. Die Texte in diesem Steuerelement werden nicht lesbar dargestellt. Außerdem wird dieses Element über die Eigenschaft `Password` abgefragt.

**HINWEIS** Denken Sie daran, dass Kennworte, die in einem normalen `String`-Objekt gespeichert werden, solange im Speicher stehen, bis der Garbage Collector den Speicher löscht bzw. überschreibt. Kennworte sollten darum in `SecureString`-Objekten abgelegt werden. Wenn diese nicht mehr benötigt werden, kann der Speicher mit dem Kennwort sofort gelöscht werden. Außerdem wird das Kennwort im Speicher verschlüsselt abgelegt, sodass es nicht mit einem Debugger oder aus der Swap-Datei ausgelesen werden kann.

Die Anwendung des `TextBox`-Elements wird in Listing 12.14 gezeigt. Wenn Sie die Anwendung starten, können Sie die vorgegebenen Texte modifizieren (Abbildung 12.14). Bei der Eingabe eines Kennworts (ganz unten) wird mit der Eigenschaft `PasswordChar` das Zeichen »#« gesetzt. Dies ist das Zeichen, welches als Platzhalter für die Kennwortzeichen im `PasswordBox`-Element ausgegeben wird.

```
<Window x:Class="TextBoxen.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Textboxen" Height="300" Width="300"
  >
  <Grid>
    <StackPanel Orientation="Vertical">
      // Einfache TextBox
      <Label Margin="5">Einfache TextBox:</Label>
      <TextBox Name="text1" Margin="5">Einfache TextBox</TextBox>

      // Mehrzeilige TextBox
      <Label Margin="5">TextBox mit mehreren Zeilen:</Label>
      <TextBox Name="text2" AcceptsReturn="True" Margin="5" Height="70">Mehrere Zeilen</TextBox>

      // TextBox für ein Kennwort
      <Label Margin="5">Kennwort-Eingabe:</Label>
      <PasswordBox Name="text3" PasswordChar="#" Margin="5" />
    </StackPanel>
  </Grid>
</Window>
```

**Listing 12.14** Verschiedene Texteingaben

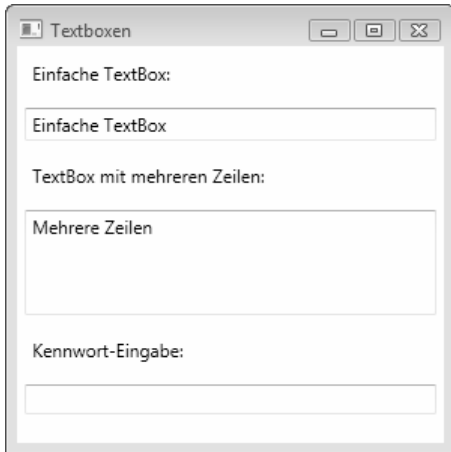


Abbildung 12.14 TextBox-Elemente und PasswordBox-Element

Ein TextBox-Element erlaubt nur die einfache Eingabe von Buchstaben. Wesentlich flexibler ist dagegen das RichTextBox-Element. Hier können Texte und grafische Elemente gemischt werden (Listing 12.15).

```
<Window x:Class="RichText.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="RichTextBox" Height="140" Width="500"
  >
  <Grid>
    <RichTextBox>
      <FlowDocument>
        <Paragraph>
          <Run xml:space="preserve" FontSize="20">Ein RichTextBox-Element </Run>
          <Ellipse Width="50" Height="20" Fill="Red" Stroke="Black" />
          <Run xml:space="preserve" FontSize="20"> kann neben einem Text auch </Run>
          <Run FontWeight="Bold" FontStyle="Italic" FontSize="20">grafische Elemente</Run>
          <Run xml:space="preserve" FontSize="20"> enthalten.</Run>
          <Polyline Stroke="Green" StrokeThickness="3" Points="0,0 20,20 40,0 60,20 80,0 100,20" />
        </Paragraph>
      </FlowDocument>
    </RichTextBox>
  </Grid>
</Window>
```

Listing 12.15 Text und Grafik in einem RichTextBox-Element

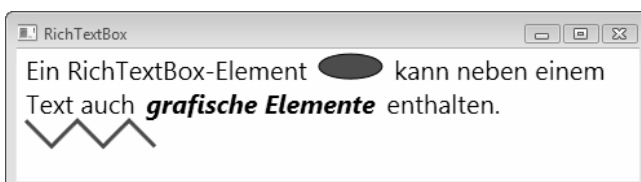


Abbildung 12.15 Das RichTextBox-Element mit Text und Grafik

Innerhalb des `RichTextBox`-Elementes haben wir zunächst ein `FlowDocument`-Element deklariert, welches dann ggf. mehrere `Paragraph`-Elemente enthalten kann. Jedes `Paragraph`-Element stellt einen kompletten Abschnitt des Dokuments dar und kann beliebige andere Elemente enthalten. Dazu gehören Texte, grafische Elemente und auch normale Steuerelemente. Texte werden in `Run`-Elementen vorgegeben, die sehr vielfältige Formatierungsanweisungen enthalten können. Um Zeichnungen innerhalb der Texte darzustellen, werden an den entsprechenden Stellen einfach die normalen grafischen WPF-Elemente benutzt (Listing 12.15).

Sie können die einzelnen Elemente, die in einem `RichTextBox`-Element enthalten sind, auch zur Laufzeit modifizieren. Hierzu müssen Sie die Elemente, die verändert werden sollen, mit einem Namen versehen. Danach können Sie aus dem Code ganz normal auf die Elemente zugreifen. Listing 12.16 zeigt den XAML-Teil eines Beispiels.

```
<Window x:Class="RichText2.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Änderbares Rechteck" Height="190" Width="400"
  >
  <Grid>
    <StackPanel Orientation="Vertical">
      // Eingabe von Breite und Höhe des Rechtecks
      <StackPanel Orientation="Horizontal">
        <Label FontSize="20">Breite:</Label>
        <TextBox Name="textBreite" FontSize="20" Width="100" TextChanged="OnTextChanged">50</TextBox>
        <Label FontSize="20">Höhe:</Label>
        <TextBox Name="textHoehe" FontSize="20" Width="100" TextChanged="OnTextChanged">20</TextBox>
      </StackPanel>

      // Text mit einem Rechteck
      <RichTextBox Margin="10" Height="100">
        <FlowDocument>
          <Paragraph>
            <Run xml:space="preserve" FontSize="20">Hier ist ein </Run>
            <Rectangle Name="rect" Width="50" Height="20" Fill="Red" Stroke="Black"
              StrokeThickness="2" />
            <Run FontSize="20">, welches in Breite und Höhe geändert werden kann.</Run>
          </Paragraph>
        </FlowDocument>
      </RichTextBox>
    </StackPanel>
  </Grid>
</Window>
```

**Listing 12.16** Ein Text mit einem variablen Rechteck

Zunächst erzeugen wir zwei `TextBox`-Elemente mit den Namen »textBreite« und »textHoehe« im oberen Bereich eines `StackPanel`-Elements. Das `TextChanged`-Ereignis dieser Eingabeelemente wird an die Methode `OnTextChanged` gebunden (Listing 12.17). In unteren Teil deklarieren wir ein `RichTextBox`-Element, welches einen Text und das zu steuernde Rechteck mit dem Namen »rect« enthält.

```

Imports System
Imports System.Windows

Namespace RichText2
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnTextChanged(ByVal sender As Object, _
            ByVal e As RoutedEventArgs)

            ' Prüfen, ob alle Elemente vorhanden sind
            If textBreite Is Nothing Or _
                textHoehe Is Nothing Or _
                rect Is Nothing Then Return

            Dim dBreite As Double
            Dim DHoehe As Double
            Dim bTest As Boolean

            ' Fehlerhafte Eingaben werden mit TryParse erkannt
            bTest = Double.TryParse(textBreite.Text, dBreite)
            bTest = Double.TryParse(textHoehe.Text, DHoehe)

            ' Alle Werte sind OK: Rechteck ändern
            If bTest And dBreite > 0.0 And DHoehe > 0.0 Then
                rect.Width = dBreite
                rect.Height = DHoehe
            End If

        End Sub
    End Class
End Namespace

```

**Listing 12.17** Steuerung des Rechtecks im RichTextBox-Element

Wenn wir die Ereignismethode programmieren, müssen wir allerdings ein bisschen aufpassen. Zuerst wird in der logischen Hierarchie die `TextBox` für die Breite deklariert und dementsprechend wird sie zur Laufzeit des Programms auch als erste erzeugt und dargestellt. Nun wird bei der Initialisierung dieser `TextBox` im XAML-Code die Zahl »50« zugewiesen und dadurch das `TextChanged`-Ereignis ausgelöst. In der Ereignismethode müssen wir also berücksichtigen, dass beim ersten Aufruf die zweite `TextBox` (Höhe) und das Rechteck im `RichTextBox`-Element noch gar nicht existieren. Darum werden in der Ereignismethode die einzelnen Objekte auf `nothing` getestet.

Außerdem müssen wir damit rechnen, dass ein Anwender unseres Programms fehlerhafte Eingaben tätigt. Dazu gehören negative Zahlen oder auch Buchstaben. Aus diesem Grund wird die Methode `TryParse` aus dem Typ `Double` benutzt, um die Eingaben zu prüfen, bevor das Rechteck manipuliert wird (Abbildung 12.16).

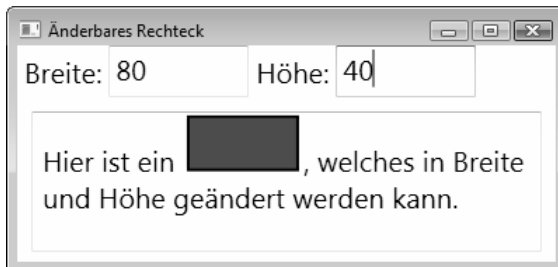


Abbildung 12.16 Das Rechteck wurde geändert

## Das Label-Element

Das Label-Element haben Sie in den vorangegangenen Beispielen schon kennen gelernt. Es dient in erster Linie dazu, vor anderen Steuerelementen einen Text auszugeben, wie in Listing 12.16 bereits gezeigt wurde. Im ersten Moment sieht es so aus, als ob das Label-Element überflüssig wäre und durch eine einfache TextBox simuliert werden könnte. Bei einem Label kommt jedoch als neue Möglichkeit die Verarbeitung des Fokus hinzu.

Sie sollten ihre Benutzerschnittstellen immer so aufbauen, dass der Anwender auch mit der Tastatur die verschiedenen Steuerelemente in einem Dialogfenster ansteuern kann. Hierbei spielt das Label-Element eine große Rolle. Sie können für jedes Label eine Kurztaste (Access Key) definieren, die der Anwender in Kombination mit der Alt-Taste zur Ansteuerung des Label-Elements verwenden kann. Da nun das Label selbst jedoch nicht den Eingabefokus erhalten kann, wird dieser an das im Label gebundene Element weitergegeben.

In Listing 12.18 wird statt dem normalen Text im Label ein AccessText-Element benutzt. Der Buchstabe hinter dem Unterstreichungszeichen ist die Kurztaste für das Label-Element. Sie können also den Eingabefokus der TextBox für die Breite zuordnen, indem Sie die Tastenkombination `[Alt] [B]` eingeben. Das Ziel-element, dem der Eingabefokus zugewiesen werden soll, wird in der Target-Eigenschaft des Label-Elements angegeben. Hier müssen Sie folgende Syntax benutzen:

```
Target="{Binding ElementName=textBreite}"
```

Als ElementName wird das Steuerelement angegeben, welches den Eingabefokus enthalten soll.

**HINWEIS** Die Kurztasten werden in der Benutzerschnittstelle erst dann durch die Unterstreichungszeichen gekennzeichnet, wenn Sie die Alt-Taste drücken (Abbildung 12.17).

Wie Sie in Listing 12.18 sehen können, ist das Element AccessText nicht unbedingt erforderlich, um die Kurztaste zu definieren. Das AccessText-Element ist dann sehr nützlich, wenn Sie die Textausgabe formatieren oder die vielfältigen Möglichkeiten mit Animationen benutzen wollen.

```
<Window x:Class="Labels.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Label" Height="100" Width="450"
  >
```

```

<Grid>
  <StackPanel Orientation="Horizontal" Height="40">
    <Label FontSize="20" Target="{Binding ElementName=textBreite}">
      <AccessText>_Breite:</AccessText>
    </Label>
    <TextBox Name="textBreite" FontSize="20" Width="100" ToolTip="Breitenangabe">50</TextBox>

    <Label FontSize="20" Target="{Binding ElementName=textHoehe}">
      _Höhe:
    </Label>
    <TextBox Name="textHoehe" FontSize="20" Width="100" ToolTip="Höhenangabe">20</TextBox>

    <Button FontSize="20" Margin="5,0,0,0" Click="OnAusgabe" ToolTip="Ausgabe der Daten">
      <AccessText>_Ausgabe...</AccessText>
    </Button>
  </StackPanel>
</Grid>
</Window>

```

**Listing 12.18** Label-Elemente und Schaltflächen mit Kurztasten



**Abbildung 12.17** Die dargestellten Kurztasten in der Benutzerschnittstelle

Natürlich können Sie die Reihenfolge der Steuerelemente für das Drücken der Tabulator-Taste über die Eigenschaft `TabIndex` festlegen, welche in jedem Steuerelement vorhanden ist. Außerdem haben wir in Listing 12.18 die Eigenschaft `ToolTip` verwendet, um kleine Hilfstexte auf den Steuerelementen darzustellen, wenn Sie mit der Maus darüber verweilen.

## Menüs

Viele Windows-Applikationen werden durch hierarchisch angeordnete Menüs gesteuert. Wir unterscheiden zwischen *Popup-Menüs*, die am oberen Rand eines Fensters angeordnet sind (Hauptmenü) und den *Kontext-Menüs*, die dann erscheinen, wenn Sie mit der linken Maustaste auf ein Steuerelement klicken. Ein Menü enthält im Allgemeinen mehrere `MenuItem`-Elemente.

Das folgende Beispiel (Listing 12.19) zeigt den Code für ein einfaches Hauptmenü eines noch einfacheren Textverarbeitungsprogramms. In diesem Menü wurden alle gängigen Befehle einer Anwendung dargestellt, jedoch hier nicht immer implementiert.

```

<Window x:Class="TestMenue.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Ein Menü" Height="300" Width="400"
  >

```

```

<Window.Resources>
  // Bild für Befehl "Neu"
  <Path x:Key="imgNeu" Fill="White" Stroke="Black" StrokeThickness="1"
    Data="M 3,3 L 3,15 14,15 14,3 Z M 10,3 L 14,7" />

  // Bild für Befehl "Beenden"
  <Path x:Key="imgBeenden" Fill="Red" Stroke="Black" StrokeThickness="1"
    Data="M 7,13 L 2,7 5,7 5,0 9,0 9,7 12,7 z" />
</Window.Resources>

<DockPanel>
  <Menu DockPanel.Dock="Top">
    <MenuItem Header="_Datei">
      <MenuItem Header="_Neu..." Command="New" InputGestureText="Strg+N" Icon="{StaticResource
imgNeu}" />
      <Separator />
      <MenuItem Header="_Öffnen..." />
      <MenuItem Header="_Speichern" />
      <MenuItem Header="Speichern _unter..." />
      <Separator />
      <MenuItem Header="_Beenden" Click="OnBeenden" Icon="{StaticResource imgBeenden}" />
    </MenuItem>

    <MenuItem Header=" _Bearbeiten">
      <MenuItem Header=" _Ausschneiden" Command="Cut" InputGestureText="Strg+X" />
      <MenuItem Header=" _Kopieren" Command="Copy" InputGestureText="Strg+C" />
      <MenuItem Header=" _Einfügen" Command="Paste" InputGestureText="Strg+V" />
      <Separator />
      <MenuItem Header=" _Alles markieren" Command="SelectAll" InputGestureText="Strg+A" />
    </MenuItem>

    <MenuItem Header=" _Hilfe">
      <MenuItem Header=" _Hilfe..." />
      <Separator />
      <MenuItem Header=" _Über..." />
    </MenuItem>
  </Menu>

  <TextBox Name="text" AcceptsReturn="True" TextWrapping="Wrap" VerticalScrollBarVisibility="Auto"
/>
</DockPanel>
</Window>

```

**Listing 12.19** Ein einfaches Hauptmenü

Damit das Hauptmenü an der oberen Fensterkante anliegt, benutzen wir als Layout-Element ein `DockPanel` und setzen die Eigenschaft `DockPanel.Dock` auf den Wert `Top`. Nun können wir `MenuItem`-Elemente in das Hauptmenü einfügen. Jeder Menüpunkt wird über mehrere wichtige Eigenschaften bestimmt. Zunächst wird die Eigenschaft `Header` benutzt, um den Text für den Menüpunkt festzulegen. In diesem Text können Sie wiederum ein Unterstreichungszeichen benutzen, um die Kurztaste (in Kombination mit der `Alt`-Taste) für den Menüpunkt festzulegen.

Da ein Menüpunkt normalerweise einen Befehl in der Anwendung auslöst, müssen wir eine Verbindung mit den entsprechenden Ereignismethoden herstellen. Die kann entweder über die Eigenschaft `Command` mit einem `CommandBinding`-Objekt oder durch direktes Binden des `Click`-Ereignisses an die Ereignismethode realisiert werden. Im Beispiel wurden beide Methoden benutzt (Listing 12.20). Die benötigten `CommandBinding`-Objekte werden im Konstruktor-Code des Fensters implementiert. Die einzelnen Ereignismethoden, die an die Menübefehle gebunden sind, führen sehr einfachen Code aus und müssen nicht weiter erläutert werden.

Wenn Sie den Text für die Kurztaste des Menübefehls ändern wollen, benutzen Sie hierzu die Eigenschaft `InputGestureText`.

Horizontale Trennelemente zwischen zwei Menüpunkten werden durch ein `Separator`-Element dargestellt.

Oft möchten wir im linken Bereich des Menüpunktes ein kleines Bild (Icon) anzeigen, welches dann auch auf den Schaltflächen der Werkzeugleiste benutzt werden kann. Hier müssen wir etwas vorgreifen, denn wir benutzen »Ressourcen«, um diese Bilder zu definieren. Im `Block Window.Resources` deklarieren wir zwei `Path`-Elemente, denen wir über die Eigenschaft `Key` einen Namen zuweisen.

Nun können wir die definierten Grafiken über die Eigenschaft `Icon` des `MenuItem`-Elements zuweisen. Die Bilder sollten die Größe 17x17 Einheiten nicht überschreiten, ansonsten wird der Menüpunkt sehr hoch bzw. breit.

**HINWEIS**

Sie können natürlich auch Pixel-Grafiken (BMP, JPG, o.ä.) über die `Icon`-Eigenschaft in einem Menüpunkt darstellen. Verwenden Sie dann ein `Image`-Element für die Eigenschaft `MenuItem.Icon`. Ggf. müssen Sie auch hier die Größe der Grafik mit einem `Viewbox`-Element anpassen.

Nach dem Hauptmenü deklarieren wir noch ein `TextBox`-Element mit dem Namen »text«, das die Textverarbeitungsmöglichkeit realisieren soll. Die Befehle *Einfügen*, *Kopieren*, *Ausschneiden* und *Alles markieren* werden auf diese `TextBox` angewendet, um eine möglichst einfache Implementierung zu erhalten.

Starten Sie nun die Anwendung, und »spielen« Sie etwas mit den Befehlen in den Menüs *Datei* und *Bearbeiten*. Beobachten Sie, wie die Menüpunkte durch die implementierten `CommandBinding`-Elemente ein- und ausgeschaltet werden. Benutzen Sie auch die möglichen Kurztasten, die wir für das Menü definiert haben. Die laufende Anwendung ist in Abbildung 12.18 zu sehen.

```
Imports System
Imports System.Windows
Imports System.Windows.Input

Namespace TestMenue
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
            text.Focus()

            ' Befehl: Neu
            Dim cbNeu As CommandBinding
            cbNeu = New CommandBinding(ApplicationCommands.[New])
```

```
AddHandler cbNeu.Executed, AddressOf OnNeu
AddHandler cbNeu.CanExecute, AddressOf OnNeuCanExecute
Me.CommandBindings.Add(cbNeu)

' Befehl: Ausschneiden
Dim cbAusschneiden As CommandBinding
cbAusschneiden = New CommandBinding(ApplicationCommands.Cut)
AddHandler cbAusschneiden.Executed, AddressOf OnAusschneiden
AddHandler cbAusschneiden.CanExecute, AddressOf OnAusschneidenCanExecute
Me.CommandBindings.Add(cbAusschneiden)

' Befehl: Einfügen
Dim cbEinfuegen As CommandBinding
cbEinfuegen = New CommandBinding(ApplicationCommands.Paste)
AddHandler cbEinfuegen.Executed, AddressOf OnEinfuegen
AddHandler cbEinfuegen.CanExecute, AddressOf OnEinfuegenCanExecute
Me.CommandBindings.Add(cbEinfuegen)

' Befehl: Kopieren
Dim cbKopieren As CommandBinding
cbKopieren = New CommandBinding(ApplicationCommands.Copy)
AddHandler cbKopieren.Executed, AddressOf OnKopieren
AddHandler cbKopieren.CanExecute, AddressOf OnAusschneidenCanExecute
Me.CommandBindings.Add(cbKopieren)
End Sub

Private Sub OnNeu(ByVal Sender As Object, ByVal e As RoutedEventArgs)
    ' Text löschen
    text.Clear()
End Sub

Private Sub OnNeuCanExecute(ByVal sender As Object, _
    ByVal e As CanExecuteRoutedEventArgs)
    ' Neu-Befehl nur ausführen, wenn TextBox einen Text enthält
    e.CanExecute = (text.Text.Length > 0)
End Sub

Private Sub OnBeenden(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    'Fenster schließen
    Me.Close()
End Sub

Private Sub OnAusschneiden(ByVal sender As Object, _
    ByVal e As RoutedEventArgs)
    ' Befehl: Ausschneiden
    text.Cut()
End Sub

Private Sub OnAusschneidenCanExecute(ByVal sender As Object, _
    ByVal e As CanExecuteRoutedEventArgs)
    e.CanExecute = (text.Text.Length > 0)
End Sub
```

```
Private Sub OnEinfuegen(ByVal sender As Object, _
                        ByVal e As RoutedEventArgs)
    ' Befehl: Einfügen
    text.Paste()
End Sub

Private Sub OnEinfuegenCanExecute(ByVal sender As Object, _
                                  ByVal e As CanExecuteRoutedEventArgs)
    ' Einfügen-Befehl nur ausführen, wenn Text in der Zwischenablage ist
    e.CanExecute = Clipboard.ContainsText
End Sub

Private Sub OnKopieren(ByVal sender As Object, _
                       ByVal e As RoutedEventArgs)
    ' Befehl: Kopieren
    text.Copy()
End Sub

Private Sub OnAllesMarkieren(ByVal sender As Object, _
                              ByVal e As RoutedEventArgs)
    'Befehl: Alles markieren
    text.SelectAll()
End Sub
End Class
End Namespace
```

**Listing 12.20** Implementierung der Befehle im Hauptmenü



**Abbildung 12.18** Hauptmenü mit Bildern und Kurztasten

Die Benutzung von kontextsensitiven Menüs funktioniert ganz ähnlich wie die Deklaration eines Hauptmenüs. Die gewünschten Menüpunkte werden in dem Steuerelement als `ContextMenu`-Element deklariert, in dem sie benutzt werden sollen. Listing 12.21 zeigt die Erweiterung des letzten Beispiels. Durch Anklicken der `TextBox` mit der rechten Maustaste erscheint das Menü mit den beiden Befehlen *Einfügen* und *Neu*.

```
...
    Private Sub OnSuchenStart(ByVal sender As Object, _
                             ByVal e As RoutedEventArgs)
        Dim iPos As Integer
        iPos = text.SelectionStart + text.SelectionLength
        Dim strText As String
        strText = text.Text
        iPos = strText.IndexOf(textSuchen.Text, iPos)
        If iPos >= 0 Then
            text.Select(iPos, textSuchen.Text.Length)
        Else
            text.Select(0, 0)
        End If
    End Sub
End Sub
...
```

**Listing 12.21** Erweiterter XAML-Code für ein kontextsensitives Menü im TextBox-Element

Der implementierte Code bleibt unverändert, da die Menüpunkte aus dem Kontextmenü über CommandBinding-Elemente an die entsprechenden Ereignismethoden gebunden sind. Auch die Aktivierung und Deaktivierung der Menüpunkte funktioniert durch diese Bindung wie erwartet. Natürlich können Sie auch in einem Kontextmenü kleine Bildchen auf der linken Seite darstellen, in dem Sie die Icon-Eigenschaft entsprechend setzen.

## Werkzeugleisten (Toolbars)

Ein ebenfalls häufig verwendetes Element in Benutzerschnittstellen von Windows-Anwendungen ist die Werkzeugleiste (Toolbar). Die WPF-Werkzeugleiste kann nicht nur Schaltflächen, sondern auch andere Steuerelemente, wie TextBox- oder ComboBox-Elemente, enthalten. Wir wollen das letzte Beispiel mit dem Menü nun mit einer Werkzeugleiste erweitern. Dazu implementieren wir den folgenden XAML-Code zur Definition von Menü- und TextBox-Element aus dem vorherigen Beispiel (Listing 12.22):

```
...
<ToolBarTray DockPanel.Dock="Top">
    <ToolBar>
        <Button Command="New" Content="{StaticResource imgNeu}" />
        <Separator />
        <Label>_Suchen:</Label>
        <TextBox Name="textSuchen" Width="100" />
        <Button Click="OnSuchenStart">
            <AccessText>_Start</AccessText>
        </Button>
        <Separator />
        <Button Click="OnBeenden" Content="{StaticResource imgBeenden}" />
    </ToolBar>
</ToolBarTray>
...
```

**Listing 12.22** Die Anwendung mit einer Werkzeugleiste

Wir beginnen mit einem `ToolBarTray`-Element, welches das Layout der gesamten Werkzeugleiste kontrolliert. Damit die Werkzeugleiste direkt unter dem Hauptmenü erscheint, setzen wir die Eigenschaft `DockPanel.Dock` wieder auf den Wert `Top`. Im Layout-Element platzieren wir jetzt ein `ToolBar`-Element, in dem dann die verschiedenen Steuerelemente der Werkzeugleiste positioniert werden können.

Zunächst benötigen wir eine Schaltfläche für den *Neu*-Befehl und verwenden das bereits definierte Bildchen aus den vorhandenen Windows-Ressourcen. Danach kommt eine senkrechte Trennlinie (Separator).

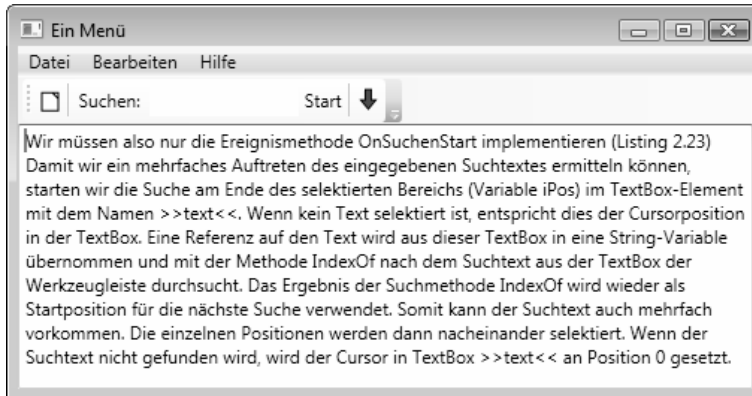
Wir wollen den Text in unserer `TextBox` nach einem bestimmten Text durchsuchen können. Dazu erzeugen wir im `ToolBar`-Element nun erst ein `Label`-Element mit dem Text *Suchen*, dann eine `TextBox` mit einer festen Breite und schließlich eine Schaltfläche mit der Aufschrift *Start*. Diese Schaltfläche ist mit der Ereignismethode `OnSuchenStart` (Listing 12.23) verbunden. Nach einer weiteren Trennlinie wird noch eine Schaltfläche deklariert, mit der das Programm beendet werden kann.

Auch für die Werkzeugleiste können wir die bereits existierenden `CommandBinding`-Elemente oder Ereignismethoden verwenden.

Wir müssen also nur die Ereignismethode `OnSuchenStart` implementieren (Listing 12.23). Damit wir ein mehrfaches Auftreten des eingegebenen Suchtextes ermitteln können, starten wir die Suche am Ende des selektierten Bereichs (Variable `iPos`) im `TextBox`-Element mit dem Namen »text«. Wenn kein Text selektiert ist, entspricht dies der Cursor-Position in der `TextBox`. Eine Referenz auf den Text wird aus dieser `TextBox` in eine `String`-Variable übernommen und mit der Methode `IndexOf` nach dem Suchtext aus der `TextBox` der Werkzeugleiste durchsucht. Das Ergebnis der Suchmethode `IndexOf` wird wieder als Startposition für die nächste Suche verwendet. Somit kann der Suchtext auch mehrfach vorkommen. Die einzelnen Positionen werden dann nacheinander selektiert. Wenn der Suchtext nicht gefunden wird, wird der Cursor in `TextBox` »text« an Position 0 gesetzt (Abbildung 12.19).

```
...
Private Sub OnSuchenStart(ByVal sender As Object, _
                        ByVal e As RoutedEventArgs)
    Dim iPos As Integer
    iPos = text.SelectionStart + text.SelectionLength
    Dim strText As String
    strText = text.Text
    iPos = strText.IndexOf(textSuchen.Text, iPos)
    If iPos >= 0 Then
        text.Select(iPos, textSuchen.Text.Length)
    Else
        text.Select(0, 0)
    End If
End Sub
...
```

**Listing 12.23** Der Zusatzcode für den Suchen-Befehl



**Abbildung 12.19** Der Text-Editor mit einer Werkzeugleiste

Mehrere Werkzeugleisten können nacheinander in einem `ToolBarTray`-Element deklariert werden (siehe Abbildung 12.20):

```
<ToolBarTray DockPanel.Dock="Top">
  <ToolBar>
    <!-- Die erste Toolbar -->
  </ToolBar>
  <!-- Die zweite Toolbar -->
  <ToolBar>
    <Button Command="Cut">Ausschneiden</Button>
    <Button Command="Copy">Kopieren</Button>
    <Button Command="Paste">Einfügen</Button>
  </ToolBar>
</ToolBarTray>
```

Die einzelnen Werkzeugleisten können Sie nebeneinander oder untereinander anordnen. Eine Verschiebung an den rechten, linken oder unteren Rand des Fensters ist in dieser Anwendung jedoch nicht möglich.



**Abbildung 12.20** Zwei Werkzeugleisten in der Anwendung

## Zusammenfassung

Steuerelemente sind in jedem Framework für Benutzerschnittstellen vorhanden, so auch in Windows Presentation Foundation. Auch hier werden Steuerelemente über Klassen implementiert, die über Eigenschaften und Methoden verfügen. Diese nutzt der Programmierer, um die Steuerelemente zu kontrollieren.

Wichtig für Steuerelemente sind die Konzepte *Routed Events* (weitergeleitete Ereignisse) und *Routed Commands* (weitergeleitete Befehle). Diese Konzepte erleichtern die Programmierung von hierarchischen Benutzerschnittstellen erheblich. Auch die Weitergabe von Eigenschaften stellt eine große Erleichterung für den Softwareentwickler dar.

Denken Sie auch daran, dass WPF-Steuerelemente eine Trennung von Design und Logik durchführen, sodass das Aussehen von Steuerelementen sehr leicht geändert werden kann. Diese Möglichkeit betrachten wir in einem späteren Kapitel noch genauer.

Die einzelnen Steuerelemente von WPF wurden hier nicht im Detail durchgesprochen. Das würde den Rahmen dieses Buches leider sprengen.