

Teil C

Windows Presentation Foundation (WPF)

In diesem Teil:

WPF – Einführung und Grundlagen	207
Steuerelemente	235
Layout	275
Grafische Grundelemente	305

Kapitel 11

WPF – Einführung und Grundlagen

In diesem Kapitel:

Einführung	208
Aller Anfang ist schwer ...	208
Architektur	209
Das .NET Framework ab Version 3.0	210
Vektorgrafik	211
Trennung von Design und Logik	211
Der WPF-Designer	221
Logischer und visueller Baum	223
Die XAML-Syntax im Überblick	225
Ein eigenes XAMLPad	228
Zusammenfassung	233

Einführung

Die Bibliotheken, die in Microsoft Windows für die Grafikausgaben und die Benutzerschnittstelle verantwortlich sind (GDI32.DLL und USER32.DLL), gibt es schon seit den Anfangszeiten von Windows. Damals waren es noch 16-Bit-Bibliotheken. Diese DLLs sind im Laufe der letzten 20 Jahre immer wieder migriert, angepasst und natürlich stark erweitert worden.

Nun stehen wieder große Erweiterungen für Grafik und Benutzerschnittstelle an. Windows Presentation Foundation (kurz: WPF, immer noch unter ihrem Codenamen *Avalon* bekannt) bietet neue Möglichkeiten, die bisher unter Windows entweder gar nicht, oder nur mit Zusatzbibliotheken zur Verfügung standen. Aus diesem Grund wurde das alte Grafik-Subsystem nicht erweitert, stattdessen wurde ein neues Subsystem implementiert, das der Anwender sowohl mit Windows XP,¹ Windows Server 2003/2008 und Windows Vista benutzen kann. In Windows XP und Windows Server 2003 müssen die neuen Bibliotheken extra installiert werden. In Windows Vista sind diese bereits enthalten und können nach der Installation des Betriebssystems sofort genutzt werden.

In diesem Kapitel wollen wir die Grundlagen von Windows Presentation Foundation besprechen. Und glauben Sie mir, es gibt viel zu berichten. Um Ihnen nur einen kleinen Vorgeschmack auf das Kommende zu geben, hier einige Stichworte: Vektorgrafiken, Fließkomma-Koordinaten, deklarative Programmierung, weitergeleitete Kommandos, weitergeleitete Eigenschaften, Abhängigkeitseigenschaften, XAML und vieles mehr.

HINWEIS

Bevor es mit WPF losgeht, noch ein Hinweis in eigener Sache. Basis des kompletten WPF-Teils ist das Buch *WPF Crashkurs* von Bernd Marquardt.² Bernd hat alle Texte dieses Teils geschrieben, und von ihm stammen auch die C#-Originalbeispiele, auf denen die Visual Basic-Beispiele dieses Buchteils basieren. Auch wenn ich meinen Dank schon im Vorwort angebracht habe – ich möchte es nicht versäumen, mich noch einmal an der dafür am besten geeigneten Stelle für das Überlassen seiner Texte und seiner C#-Beispiele zu bedanken! Dank gilt an dieser Stelle auch nochmals meinem Mitarbeiter Jürgen Heckhuis, der die Beispiele von C# nach Visual Basic 2008 »übersetzt« hat.

BEGLEITDATEIEN

Unter `.\Samples\Chapter11\` finden Sie die Beispieldateien für dieses Kapitel.

Aller Anfang ist schwer ...

Wo soll man eigentlich anfangen, wenn so vieles neu oder anders ist? Beginnen wir mit einer kurzen Beschreibung von WPF. Windows Presentation Foundation ist ein neues grafisches Subsystem für Microsoft Windows. Es bietet eine Unterstützung und eine einheitliche Sichtweise auf:

¹ Mindestens Service Pack 2 ist für Windows XP erforderlich.

² *Windows Presentation Foundation Crashkurs*, ISBN: 978-3-86645-504-7, erschienen bei Microsoft Press. Sie erreichen den Autor über seine Website www.gosky.de.

- Benutzerschnittstellen
- 2D-Grafiken
- 3D-Grafiken
- Dokumente
- Medien (Bilder, Filme,...)

Aus diesem Grund besteht WPF aus einer Reihe von Diensten, die der Softwareentwickler in Anspruch nehmen kann, um seine Softwarelösung zu bauen. Diese Dienste können folgendermaßen unterteilt werden:

- Basis-Dienste
 - XAML, Eingaben, Eigenschaften-System, Ereignisse, Zugänglichkeit
- Benutzerschnittstellen-Dienste
 - Anwendungs-Dienste, Verteilung, Steuerelemente, Layout, Datenbindung
- Medien-Dienste
 - 2D-Grafik, 3D-Grafik, Audio, Video, Texte, Fotos, Animationen, Effekte
- Dokument-Dienste
 - XPS-Dokumente, Open Packaging

Alle diese Dienste können Sie gleichzeitig und ohne Einschränkungen in einer WPF-Applikation benutzen. WPF bietet Ihnen für alle Dienste ein einheitliches Programmiermodell, eine einheitliche Schnittstelle und eine einfache Integration der Dienste in ihre Applikation.

Architektur

In Abbildung 11.1 sehen Sie den Aufbau von WPF. Der untere Teil der Grafik zeigt den Teil des Gesamtsystems, der noch als *unmanaged* Code, also Maschinencode vorliegt. Dazu zählen in erster Linie das Betriebssystem, die Treiber, DirectX und der Desktop Window Manager (DWM). Auf das Betriebssystem setzt das .NET Framework 2.0 auf. Hierbei handelt es sich um *managed* Code. Wie Sie weiter aus der Grafik sehen können, setzt das Presentation Framework mit seinen Neuerungen auf das Betriebssystem mit DirectX und das .NET Framework 2.0 auf. Das bedeutet, dass ein Einsatz von Windows Presentation Foundation das Vorhandensein des .NET Frameworks 2.0 voraussetzt.

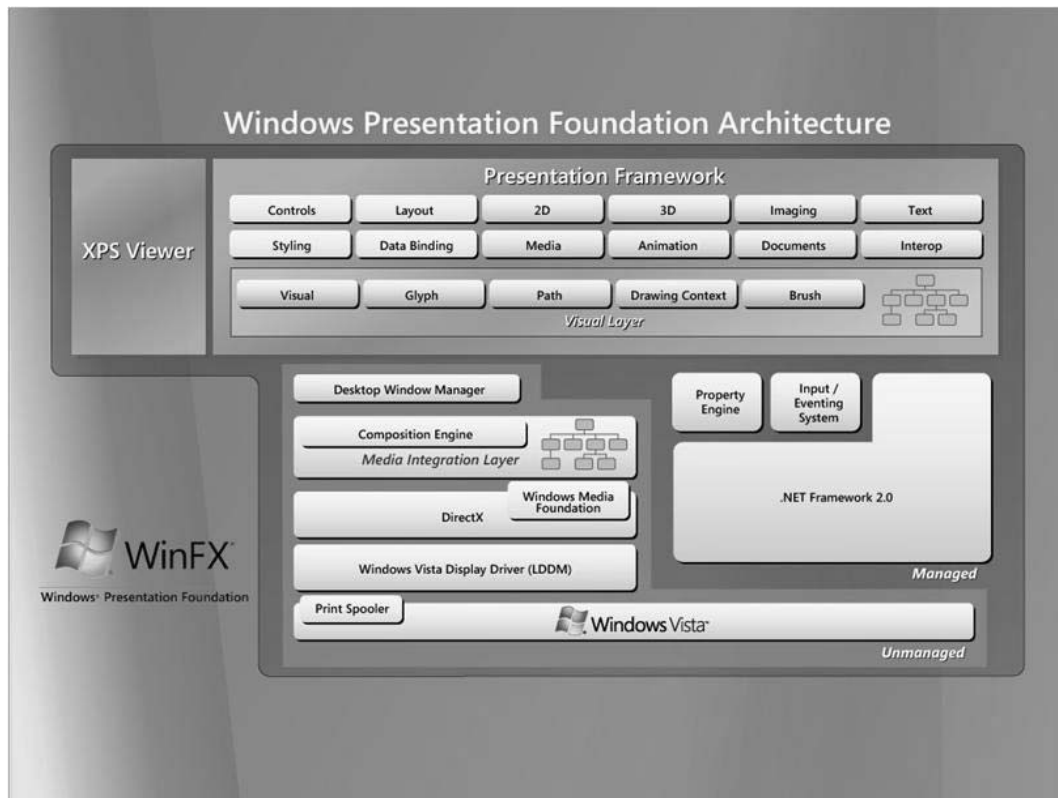


Abbildung 11.1 Die Architektur von Windows Presentation Foundation

Alle Klassen, die im .NET Framework 2.0 vorhanden sind, können mit WPF genutzt werden. Hier müssen Sie nichts Neues lernen.

Anders dagegen im Presentation Framework selbst. Hier gibt es tausende von neuen Klassen, Strukturen, Aufzählungen und Ereignissen. In Abbildung 11.1 können Sie die einzelnen WPF-Dienste, die oben erwähnt wurden, wieder erkennen.

Das .NET Framework ab Version 3.0

Im Zusammenhang mit Windows Presentation Foundation werden Sie oft den Namen »*.NET Framework 3.0*« hören. Hierbei handelt es sich um vier große Komponenten, die von Microsoft zu einem Produkt, nämlich dem *.NET Framework 3.0*, zusammengefasst wurden. Dazu gehören:

- Windows Presentation Foundation (WPF)
- Windows Communication Foundation (WCF)
- Windows Workflow Foundation (WF)
- Windows Card Services (WCS)

Dieses Buch behandelt als einzigen .NET 3.0-Schwerpunkt das Thema WPF – zum einen, weil es mit der vorherigen Version von Visual Studio schlicht nicht möglich war, WPF-Projekte zu entwickeln – erst mit Visual Studio 2008 steht ein WPF-Designer erstmals in einem Visual Studio-Produkt serienmäßig zur Verfügung. Zum anderen ist über das Thema WPF und Visual Basic im deutschsprachigen Buchmarkt bislang noch nicht viel zu bekommen.

WPF, .NET Framework 3.5 und Version Targeting

Wenn Sie WPF-Anwendungen entwickeln, bedeutet das nicht notwendigerweise, dass Sie in Ihren Anwendungen auf LINQ, das ja das .NET Framework 3.5 voraussetzt, verzichten müssen. Sie können natürlich auch eine neue WPF-Anwendung erstellen und diese auf dem .NET Framework 3.5 oder höheren Versionen³ basieren lassen.

Die Version 3.0 ist allerdings die kleinste Version, auf der Ihre WPF-Anwendungen basieren müssen. Windows XP SP2 ist damit das älteste Betriebssystem, für das Sie WPF-Anwendungen entwickeln können.

Vektorgrafik

Im Vergleich zu unseren gewohnten Windows-Forms-Anwendungen gibt es einige grundlegende konzeptionelle Neuerungen, die mit Windows Presentation Foundation eingeführt wurden. Zunächst einmal werden alle WPF-Elemente (Ausnahme: Pixel-Bilder, Bitmaps,...) als Vektorgrafik dargestellt und verarbeitet. Dies bedeutet, dass eine Schaltfläche keine Bitmap mehr ist, sondern dass diese Schaltfläche als Rechteck mit abgerundeten Ecken, einer Füllfarbe und einem Text in der Mitte an die Grafikkarte übermittelt und dort entsprechend dargestellt wird. Die Objekte »Rechteck mit Füllfarbe« und »Text« können normalerweise schneller an die Grafik-Hardware übertragen werden, als mehr oder weniger große Bitmaps. Die weitere Darstellung der Elemente übernimmt die Grafik-Hardware (GPU, Graphical Processing Unit), die mittlerweile in modernen PCs extrem leistungsfähig ist.

In einem Szenario mit vektororientierter Grafik ist es sinnvoll, alle Koordinaten als Fließkommazahlen anzugeben, da diese Grafiken nun beliebig skalierbar, also in der Größe änderbar, sind. Die Koordinatenwerte können Sie in WPF sowohl als Integerzahlen, als auch als Fließkommazahlen in einfacher und in doppelter Genauigkeit angeben.

Trennung von Design und Logik

Wenn Sie bisher eine Benutzerschnittstelle entworfen und programmiert haben, dann wurden normalerweise Design und Logik schnell vermischt. In einer Windows Forms-Anwendung wird in einer Klasse, die von `System.Windows.Forms.Form` abgeleitet wird, in der Methode `InitializeComponent` das Aussehen eines Fensters in Form von Visual Basic .NET-Code festgelegt. Nun hat ein Grafik-Designer mit Programmcode sehr

³ Sobald diese verfügbar werden.

wenig im Sinn. Der Grafiker benutzt diverse Werkzeuge, um schöne, bunte Grafiken zu erstellen. Er wird aber wohl kaum VB-Code schreiben wollen, um das Aussehen eines Fensters oder Steuerelements zu verändern. Andererseits ist es unmöglich, den logischen Teil der Applikation, also z. B. die Datenzugriffe oder das Berechnen von Zahlen mit einem Grafikwerkzeug durchzuführen.

Auf unserer Welt gibt es leider nur wenige Programmierer, die gleichzeitig auch hervorragende Designer sind. Das kann man natürlich auch anders herum betrachten: Es gibt kaum Top-Designer, die auch noch perfekt programmieren können.

Hier stoßen also zwei Welten aneinander, für die es gilt, eine für beide Seiten akzeptable Brücke zu errichten. Auf der einen Seite befindet sich der Grafiker mit seinen Werkzeugen, die irgendetwas abliefern, das der Softwareentwickler auf der anderen Seite mit der Applikationslogik »unterfüttern« kann. Dabei sollte sich der Grafiker so wenig wie möglich mit Programmierung auseinandersetzen müssen, und der Entwickler benötigt kaum eine Ahnung über die eingesetzten Designerwerkzeuge.

Ziel ist letztendlich, dass ein Top-Designer zusammen mit einem Top-Softwareentwickler eine Top-Applikation erstellt!

1. Beginnen wir im alten Stil und schreiben eine WPF-Applikation nur mit Visual Basic .NET-Code. Wir führen zunächst einmal keine Trennung von Design und Code ein. Es handelt sich natürlich um eine weitere Version des bekannten *HalloWelt*-Programms. Die Vorgehensweise ist folgende:
2. Starten Sie Visual Studio 2008 und wählen Sie *Datei > Neu > Projekt* aus.
3. Im Dialogfeld *Neues Projekt* wählen Sie als Programmiersprache Visual Basic aus. Darunter wählen Sie ein leeres »Windows«-Projekt aus.
4. Als Referenzen fügen Sie Verweise auf folgende Bibliotheken ein: System, WindowBase, PresentationFramework und PresentationCore. Benutzen Sie hierzu im Projektmappen-Explorer durch Anklicken mit der rechten Maustaste den Befehl *Verweis hinzufügen*. Wählen Sie die angegebenen Referenzen im Dialogfeld aus.
5. Fügen Sie nun im Projektmappen-Explorer eine neue VB-Code-Datei mit dem Namen »Hallo.vb« hinzu. Benutzen Sie wieder die rechte Maustaste mit dem Befehl *Hinzufügen > Neues Element*.
6. Tippen Sie nun den Code aus Listing 11.1 ein.
7. In den Projekt-Eigenschaften stellen Sie auf der Seite *Anwendung* den Ausgabebetyp *Windows-Application* ein.
8. Führen Sie das Programm aus, indem Sie aus dem Menü *Debuggen* den Befehl *Starten ohne Debugging* aufrufen.

```
Imports System
Imports System.Windows

Namespace Hallo

    Public Class Hallo
        Inherits System.Windows.Window
    End Class
End Namespace
```

```
<STAThread(> _
Shared Sub main()
    Dim w As New Window With {.Title = "Hallo, Welt!", .Width = "200", .Height = "200"}
    w.Show()

    Dim app As New Application
    app.Run()
End Sub

End Class
End Namespace
```

Listing 11.1 Das erste WPF-Programm nur in Visual Basic

Wenn wir das Programm aus Listing 11.1 laufen lassen, erscheint ein ziemlich eintöniges Fenster (Listing 11.2) mit dem Titel »Hallo, Welt!«.

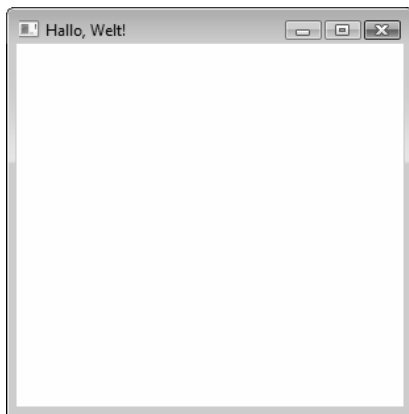


Abbildung 11.2 Das erste WPF-Fenster

Schauen wir uns den Code des Beispiels nun etwas genauer an. Nach dem Einfügen der benötigten Namensräume aus den Referenzen definieren wir einen eigenen Namensraum mit der Klasse »Hallo«. In WPF muss vor der statischen Main-Methode das Attribut [STAThread] angewendet werden, um das Threading-Modell für die Applikation auf »Single-Threaded Appartment« zu setzen. Dies ist ein Relikt aus alten COM-Zeiten, aber es sorgt ggf. für eine Kompatibilität in die Welt des *Component Object Model*.

In der Main-Methode erzeugen wir zunächst ein Window-Objekt, welches über die Eigenschaft Title den Text für die Titelzeile erhält. Die Methode Show aus der Window-Klasse sorgt für die Darstellung des Fensters auf dem Bildschirm. Wenn wir die beiden letzten Code-Zeilen der Main-Methode nicht eingeben und unser Programm starten, dann werden wir das Fenster nur sehr, sehr kurz sehen. Die Applikation wird nämlich sofort wieder beendet. Hier kommen nun diese beiden letzten Zeilen ins Spiel. Im erzeugten Application-Objekt wird die Methode Run aufgerufen, um für dieses Hauptfenster eine Meldungsschleife (Message Loop) zu erzeugen. Nun kann unser kleines Programm Meldungen empfangen. Das Fenster bleibt solange sichtbar, bis diese Meldungsschleife beendet wird, z.B. durch Anklicken der Schließ-Schaltfläche oben rechts in der Titelzeile des Fensters.

Nun haben wir in diesem Beispiel allerdings den Designer arbeitslos gemacht, denn wir haben als Softwareentwickler das Design im VB-Code implementiert. Eigentlich besteht das Design dieser Applikation nur aus einer Zeile:

```
Dim w As New Window With {.Title = "Hallo, Welt!", .Width = "200", .Height = "200"}
```

Alles andere möchte ich in diesem einfachen Beispiel einmal als Applikationslogik bezeichnen. Da ein Designer jedoch nicht mit VB-Code oder anderen Programmiersprachen arbeiten will, muss eine andere »Sprache« her, mit der man Benutzeroberflächen und Grafiken »deklarieren« kann.

In Windows Presentation Foundation wird hierzu die Extensible Application Markup Language (XAML, sprich: »gsämmel«) benutzt.

XAML: Extensible Application Markup Language

XAML ist, wie XML, eine hierarchisch orientierte Beschreibungssprache. Wir können mit XAML Benutzeroberflächen oder Grafiken deklarieren. Kommen wir wieder zu unserem ersten Beispiel (Abbildung 11.2) zurück. Als nächstes deklarieren wir nun den Designer-Teil mit XAML. Vorab jedoch noch ein kleiner Hinweis: Ein Designer wird natürlich den XAML-Code nicht »von Hand« eintippen, so wie wir es nun im nächsten Beispiel machen. Er wird stattdessen Werkzeuge verwenden, welche das erstellte Design schließlich als XAML-Code zur Verfügung stellen. In Listing 11.2 können Sie nun den erforderlichen XAML-Code für unser erstes Beispielprogramm sehen.

```
<Window x:Class="Hallo2.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Hallo, Welt!"
  Height="250"
  Width="250"
  >
</Window>
```

Listing 11.2 Unser Fenster in XAML

Was wir in Listing 11.2 sehen, ist nicht gerade überwältigend! Diese wenigen Zeilen XAML-Code machen im Grunde genommen nichts anderes, als ein Window-Objekt zu deklarieren, den Titel des Fensters auf »Hallo, Welt!« zu setzen und die Größe des Fensters auf 250 mal 250 Einheiten zu definieren. Hierzu werden die Eigenschaften Title, Width und Height auf die gewünschten Werte gesetzt.

Alle Eigenschaften in XAML werden mit Texten gefüllt. Darum müssen die Werte in Anführungszeichen gesetzt werden. Um XAML-Code zu nutzen, sollten Sie zwei Namensräume deklarieren:

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

Visual Studio 2008 fügt diese Namensräume automatisch in Ihre .NET Framework 3.0-Projekte in die XAML-Dateien ein.

Dieses Projekt wurde mit Visual Studio angelegt, und darum übernimmt Visual Studio auch den Rest der Arbeit. Im Hintergrund wird nämlich eine weitere Datei erzeugt, die VB-Code enthält und unter anderem ein Application-Objekt erstellt und die dazugehörige Run-Methode aufruft. Dieser, von Visual Studio erzeugte Code, soll uns aber gar nicht weiter interessieren. Wenn wir das Programm starten, werden wir wieder das gleiche Fenster wie im ersten Beispiel sehen.

Für das zweite Beispiel ist die Vorgehensweise mit Visual Studio 2008 folgende:

1. Starten Sie Visual Studio 2008 und wählen Sie *Datei > Neu > Projekt* im Datei-Menü aus.
2. Im Dialogfeld *Neues Projekt* wählen Sie als Programmiersprache Visual Basic aus. Darunter wählen Sie ein *WPF-Anwendung*-Projekt aus.
3. Öffnen Sie nun im Projektmappen-Explorer die XAML-Datei mit dem Namen *Window1.xaml*.
4. Tippen Sie nun den Code aus Listing 11.2 ein. Ändern Sie den Code, welchen Visual Studio erzeugt hat, einfach ab.
5. Führen Sie das Programm aus, indem Sie aus dem Menü *Debuggen* den Befehl *Starten ohne Debugging* aufrufen.

Nun haben wir die Deklaration der Benutzerschnittstelle in XAML hinterlegt. Dieser XAML-Code kann durch Grafikwerkzeuge erzeugt werden.

Jetzt aber wieder zurück zum Programmieren. Wir kommen nun auf die Applikationslogik zurück. Im nächsten Beispiel wollen wir eine minimale Logik implementieren. Für die Benutzerschnittstelle verwenden wir wiederum XAML, für die Logik wird Visual Basic eingesetzt.

Das Beispiel soll zeigen, wie eine Benutzerschnittstelle und die Applikationslogik in einer WPF-Applikation zusammenarbeiten. Dazu deklarieren wir mitten im Fenster eine Schaltfläche in einer bestimmten Größe (Listing 11.3). Das ist unsere Benutzerschnittstelle, die von einem Designer erstellt wurde.

```
<Window x:Class="Hallo3.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Hallo 3" Height="200" Width="200">
  <Grid>
    <Button Click="OnClick" Width="120" Height="40" Margin="28,58,21,64"
      >Bitte anklicken!</Button>
  </Grid>
</Window>
```

Listing 11.3 Die Benutzerschnittstelle in XAML

Innerhalb der Deklaration für das Hauptfenster wird ein weiteres WPF-Element definiert: die Schaltfläche. In XAML heißt dieses Element `Button`. Eine Schaltfläche hat natürlich Eigenschaften, die Sie setzen können. Hier wird die Breite (`Width`) der Schaltfläche mit 120 Einheiten angegeben und die Höhe (`Height`) mit 40 Einheiten. Der Text auf der Schaltfläche lautet: »Bitte anklicken!«. Wenn wir den XAML-Code eingeben und das Programm starten, passiert natürlich beim Klicken auf die Schaltfläche zunächst einmal gar nichts.

Wenn der Anwender des Programms auf die Schaltfläche klickt, dann soll jedoch ein Text ausgegeben werden. Sie ahnen, welcher Text? Natürlich »Hallo, Welt!«. Das ist die Applikationslogik, welche ein Softwareentwickler erstellt (Listing 11.4). Die Verbindung zwischen dem Button-Element und dem VB-Code wird über ein Ereignis hergestellt. In XAML (Listing 11.3) wird im Button-Element das Click-Ereignis benutzt, welches auf die Ereignismethode `OnClick` zeigt. Diese Methode wird nun in Visual Basic implementiert (Listing 11.4).

Dieses Projekt wurde ebenfalls mit Visual Studio 2008 erzeugt. Zunächst einmal werden nur die Namensräume `System` und `System.Windows` benötigt. Visual Studio erzeugt fast den gesamten Code aus Listing 11.4, wir müssen nur die Methode `OnClick` (im unteren Bereich) eingeben.

```
Imports System
Imports System.Windows

Namespace Hallo3
    Partial Public Class Window1
        Inherits System.Windows.Window

        ''' <summary>
        ''' Logik für dieses Beispiel
        ''' </summary>

        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnClick(ByVal sender As System.Object, ByVal e As _
            System.Windows.RoutedEventArgs)
            MessageBox.Show("Hallo, Welt!")
        End Sub
    End Class
End Namespace
```

Listing 11.4 Die Applikationslogik in Visual Basic

Im Konstruktor der Fensterklasse wird die Methode `InitializeComponent` aufgerufen. Auch der Code dieser Methode wird in einer separaten VB-Codedatei von Visual Studio generiert. Der VB-Befehl, der hier zum Einsatz kommt, um die verschiedenen Dateien für den Compiler korrekt zusammenzuführen, heißt `partial`. Der gesamte Code der Klasse kann auf mehrere VB-Dateien verteilt werden: Ein Teil der Klasse ist der Code aus Listing 11.4 und der andere, unsichtbare Teil ist der von Visual Studio erzeugte Zusatzcode, der die Methode `InitializeComponent` und eine Art Abbildung des XAML-Codes aus Listing 11.3 in Visual Basic enthält. Beim Übersetzen werden beide Dateien zusammengefügt. Aus diesem Grund erhalten Sie auch eine Fehlermeldung vom Compiler, wenn Sie das Beispiel übersetzen, ohne vorher die Methode `OnClick` einzugeben.

Die Benutzerschnittstelle, die in XAML definiert wurde, wird übrigens nicht von einem XML-Interpreter abgearbeitet, also interpretiert. Das wäre sicherlich zu langsam. Auch XAML wird kompiliert. Letztendlich entsteht daraus ganz normaler MSIL-Code, wie wir ihn aus jedem .NET-Programm gewohnt sind. Dieser »Zwischen-Code«, der in der EXE-Datei steht, wird dann vom JIT-Compiler (Just-In-Time) zur Laufzeit des Programms in die Maschinensprache der jeweiligen Zielplattform übersetzt.

In die Ereignismethode (Listing 11.4) werden zwei Parameter übergeben: Von Typ `object` bekommen wir die Variable `sender`, die uns angibt, welches Objekt das Ereignis ausgelöst hat. Der Parameter `e` vom Typ `EventArgs` gibt uns zusätzliche Daten an, die zu dem jeweiligen Ereignis gehören. Diese Art der Ereignisprogrammierung kennen wir schon aus Windows Forms. In der Methode selbst wird dann einfach die Methode `MessageBox.Show` mit dem gewünschten Text aufgerufen.

Das Ergebnis unserer Bemühungen zeigt Abbildung 11.3.



Abbildung 11.3 XAML und Logik werden ausgeführt

Wir sind aber noch nicht ganz am Ziel unserer Wünsche! Das ganze Programm nützt uns nichts, wenn wir die Objekte, die wir in XAML deklariert haben, nicht aus unserem Applikationscode manipulieren können. Die Schaltfläche »Bitte anklicken!« ist ein Element vom Typ `Button` und enthält diverse Eigenschaften und Methoden. Wie kann man diese nun aus dem VB-Code aufrufen?

Hierzu wollen wir das letzte Beispiel wieder ein bisschen ändern. Zunächst muss die Schaltfläche einen Namen bekommen. Dies erledigen wir durch Setzen der `Name`-Eigenschaft (Listing 11.5). Es handelt sich hierbei um die einzige Änderung am XAML-Code.

```
<Window x:Class="Hallo4.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Hallo 4" Height="200" Width="200">
  <Button Name="btn" Click="OnClick" Width="120" Height="40"
    Margin="34,61,24,61">Bitte anklicken!</Button>
</Window>
```

Listing 11.5 Die Schaltfläche bekommt einen Namen

Da die Schaltfläche nun einen Namen (`btn`) hat, können wir sie ganz normal aus dem VB-Code heraus ansprechen und die Eigenschaften neu setzen oder Methoden des Objektes `btn` aufrufen. Im Beispiel (Listing 11.6) ändern wir mit der Eigenschaft `Content` zunächst den Text, der auf der Schaltfläche ausgegeben wird. Schließlich werden die `FontSize` und die `Foreground` neu gesetzt. In allen Fällen wird vor der jeweiligen Eigenschaft der Name der Schaltfläche, der in XAML definiert wurde, angegeben.

```

Imports System
Imports System.Windows

Namespace Hallo4
    Partial Public Class Window1
        Inherits System.Windows.Window
        Public Sub New()
            InitializeComponent()
        End Sub

        Private Sub OnClick(ByVal sender As System.Object, _
            ByVal e As System.Windows.RoutedEventArgs)
            MessageBox.Show("Hallo Welt!")

            ' Nach der MessageBox: Schaltfläche ändern
            With btn
                .Content = "Guten Tag!"
                .FontSize = 20
                .Foreground = Brushes.Red
            End With
        End Sub
    End Class
End Namespace

```

Listing 11.6 Die erweiterte Applikationslogik

Abbildung 11.4 zeigt das Fenster nach dem Ausführen der Ereignismethode. Die Eigenschaften der Schaltfläche wurden entsprechend geändert, nachdem im MessageBox-Element die *OK*-Schaltfläche angeklickt wurde.



Abbildung 11.4 Darstellung des Fensters nach dem MessageBox.Show-Aufruf

Nun können wir »beide Richtungen« zwischen XAML und Code benutzen. Um aus XAML die Applikationslogik »aufzurufen«, verwenden wir Ereignisse. Diese stehen uns in den vielen WPF-Elementen in großer Anzahl zur Verfügung. Wir implementieren Ereignismethoden, welche die Applikationslogik enthalten. Um umgekehrt die in XAML deklarierten WPF-Elemente zur Laufzeit zu verändern, benutzen wir das ganz normale Objektmodell dieser Elemente und rufen die Eigenschaften und Methoden aus dem Programmcode auf. Die einzelnen Objekte werden durch ihre Namen identifiziert. Die in XAML deklarierten Eigenschaftswerte sind also die Initialeinstellungen bei der Darstellung der WPF-Elemente.

Wie eben bereits erwähnt wurde, stellt die XAML-Deklaration der Benutzerschnittstelle den Startzustand der Anwendung dar. Sie können allerdings, wenn erforderlich, die Startwerte sofort aus dem Programmcode ändern, indem Sie eine Methode für das Ereignis Loaded implementieren. Dies wird in einem Beispiel in Listing 11.7 (XAML-Teil) und Listing 11.8 (VB-Teil) gezeigt.

HINWEIS Aus diesem Beispiel wurde der nicht benötigte VB-Code, der von den Visual Studio-Erweiterungen erzeugt wurde, entfernt, um das Listing etwas kürzer zu halten.

Für das Hauptfenster mit dem Klassennamen Window1 wird das Ereignis Loaded an die Methode OnLoaded gebunden. Der Code, der in der entsprechenden VB-Methode implementiert ist, wird nun direkt nach dem Konstruktoraufruf von Window1 ausgeführt. Sobald das Fenster auf dem Bildschirm erscheint, enthält es bereits die vergrößerte Schaltfläche, da die beiden Eigenschaften Width und Height in der Methode OnLoaded entsprechend gesetzt wurden. Wird die Schaltfläche danach angeklickt, so wird die Ereignismethode OnClick aufgerufen und ausgeführt.

```
<Window x:Class="Hallo5.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="Hallo 5" Height="300" Width="300"
  Loaded="OnLoaded">
  <Button Name="btn" Click="OnClick" Width="120" Height="40"
    Margin="34,61,24,61">Bitte anklicken!</Button>
</Window>
```

Listing 11.7 Fenster mit einem Loaded-Ereignis

```
Imports System
Imports System.Windows

Namespace Hallo5
  Partial Public Class Window1
    Inherits System.Windows.Window

    Public Sub New()
      InitializeComponent()
    End Sub

    Private Sub OnClick(ByVal sender As System.Object, _
      ByVal e As System.Windows.RoutedEventArgs)
      MessageBox.Show("Hallo Welt!")

      ' Nach der MessageBox: Schaltfläche ändern
      With btn
        .Content = "Guten Tag!"
        .FontSize = 20
        .Foreground = Brushes.Red
      End With
    End Sub
  End Class
End Namespace
```

```

Private Sub OnLoaded(ByVal sender As System.Object, _
                    ByVal e As System.Windows.RoutedEventArgs)
    btn.Height = 80
    btn.Width = 150
End Sub
End Class
End Namespace

```

Listing 11.8 Der Code in OnLoaded wird zuerst ausgeführt



Abbildung 11.5 Nach der Ausführung der OnLoaded-Methode

WICHTIG Grundsätzlich können wir sagen, dass alles, was in XAML angegeben und deklariert werden kann, auch in VB.NET mithilfe von Programmcode erzeugt werden kann. Umgekehrt gilt das allerdings nicht!

Wir können nun das Design der letzten Beispielanwendung ändern, ohne den Applikationscode zu modifizieren. Dazu wollen wir etwas ziemlich Verrücktes machen: Die Schaltfläche soll schräg im Fenster stehen und wir benötigen dazu eine Transformation, die wir in Kapitel 14 noch ausführlich behandeln werden. Benutzen Sie den entsprechenden XAML-Code mit der Transformation im Moment einfach so, wie in Listing 11.9 aufgeführt.

```

<Window x:Class="Hallo5a.Window1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Hallo 5a" Height="300" Width="300"
    Loaded="Window_Loaded">
    <Button Name="btn" Click="OnClick" Width="120" Height="40" Margin="34,61,24,61">
        Bitte anklicken!
        <Button.RenderTransform>
            <SkewTransform AngleX="10" />
        </Button.RenderTransform>
    </Button>
</Window>

```

Listing 11.9 Eine schräge Schaltfläche

Die Logik, die in Visual Basic implementiert wurde (Ereignismethoden `OnLoaded`, `OnClick`), bleibt unverändert und wird für dieses Beispiel gar nicht mehr aufgelistet. Wir benutzen weiterhin den Code aus Listing 11.8. Wenn Sie das Programm starten, sehen Sie zunächst die schräge Schaltfläche (Abbildung 11.6 links); die Funktionalität hinter dieser Schaltfläche ist jedoch unverändert geblieben. Ein Anklicken löst die `MessageBox.Show`-Methode aus und danach werden die Eigenschaften der nun schräg liegenden Schaltfläche wie bisher geändert (Abbildung 11.6 rechts).



Abbildung 11.6
Eine Applikation mit
geändertem Design

Wir haben nun also eine Trennung zwischen dem Design der Applikation und der Logik erreicht. Der Softwareentwickler kann die Logik der Applikation ändern, ohne das Design (versehentlich) zu modifizieren. Umgekehrt kann ein Designer das Aussehen der Anwendung ändern, ohne dass der Programmcode ihm dabei ständig »in die Quere kommt«.

WICHTIG Wir können also sagen: XAML + Code = Anwendung.

Der WPF-Designer

Natürlich müssten Sie eine Anwendung nicht unbedingt komplett codieren – bislang haben wir ja sowohl Programmlogik als auch den XAML-Designpart in guter alter Entwicklermanier mit wenn auch verschiedenen text-basierten Editoren erstellt.

Zumindest beim Design-Part war das in Windows Forms anders. Wann immer es galt, Windows Forms-Anwendungen zu gestalten, war der WinForms-Designer von der Partie – sollte das nicht in WPF genau so sein? Sollte man meinen. Tatsache ist, dass der WPF-Designer einen WPF-Entwickler sicherlich in die Lage versetzt, rudimentärste WPF-Formulare zu erstellen. Wenn es aber darum geht, komplexere Objektverschachtelungen aus designtechnischer Sicht zu bauen, sind die Grenzen schnell erreicht. Den Komfort eines WinForms-Designers – beispielsweise beim interaktiven Erstellen von Werkzeugleisten oder Dropdown-Menüs – werden Sie beim derzeitigen Stand des WPF-Designers vergeblich suchen. Und von einer Unterstützung der viel weiter reichenden Fähigkeiten von WPF (Animationen, 3D) kann aus designertechnischer Sicht schon gar keine Rede sein.

Logischer und visueller Baum

In Windows Presentation Foundation gibt es zwei Hierarchien für die WPF-Elemente, die von großer Wichtigkeit sind. Beginnen wir mit dem logischen Baum (*Logical Tree*). Der logische Baum bildet den Zusammenhang zwischen den Objekten ab. Diese Hierarchie ist entscheidend für die Vererbung von Eigenschaften zwischen den Elementen. Innerhalb dieser Hierarchie gibt es verschiedene Methoden, um durch den Baum zu navigieren:

- `GetParent`
- `GetChildren`
- `FindLogicalNode`

Die drei genannten statischen Methoden finden Sie in der Klasse `LogicalTreeHelper`.

```
<Window x:Class="LogicalTree.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="LogicalTree" Height="300" Width="300"
  >
  <Grid>
    <Button Name="btn" Width="200" Height="30" Click="OnClick">Hallo</Button>
  </Grid>
</Window>
```

Listing 11.10 Eine Beispiel-Hierarchie

Die logische Hierarchie für den XAML-Code in Listing 11.10 ist einfach:

- `Window1`-Element
- `Grid`-Element
- `Button`-Element

Diese Hierarchie können wir mit den Methoden `GetParent` und `GetChildren` durchlaufen. Die Methoden geben Objekte vom Typ `DependencyObject` zurück, wie das folgende Listing 11.11 zeigt:

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

Namespace LogicalTree
  Partial Public Class Window1
    Inherits System.Windows.Window
```

```

Public Sub New()
    InitializeComponent()
End Sub

Private Sub onClick(ByVal sender As System.Object, _
    ByVal e As System.Windows.RoutedEventArgs)

    ' Richtung Wurzel-Element das Eltern-Element
    Dim Grid As DependencyObject
    Grid = LogicalTreeHelper.GetParent(CType(sender, FrameworkElement))
    MessageBox.Show(Grid.GetType().ToString)

    ' Richtung Wurzel-Element das nächste Eltern-Element
    Dim Window As DependencyObject
    Window = LogicalTreeHelper.GetParent(Grid)
    MessageBox.Show(Window.GetType().ToString)

    ' Alle Kindelemente des Grids
    Dim obj As DependencyObject
    For Each obj In LogicalTreeHelper.GetChildren(Grid)
        MessageBox.Show(obj.GetType().ToString)
    Next

    ' Ein bestimmtes Element suchen
    Dim depobj As DependencyObject
    depobj = LogicalTreeHelper.FindLogicalNode(Window, "btn")

    Dim btn As New Button
    btn = CType(depobj, Button)
    btn.Content = "WPF"
End Sub
End Class
End Namespace

```

Listing 11.11 Abfragen der Hierarchie aus Listing 11.10

In Listing 11.11 können Sie sehen, wie die Hierarchie abgearbeitet werden kann. In diesem einfachen Fall geben wir die Typen aus der Hierarchie nur in Meldungsfenstern aus. Im zweiten Teil des VB-Codes wird ermittelt, welche Kindelemente es im Grid-Element gibt. Hier wird nur ein Element, nämlich die Schaltfläche, gefunden. Im dritten Teil des Beispiels wird ein bestimmtes Element mit der Methode `LogicalTreeHelper.FindLogicalNode` aufgesucht. Das gefundene Objekt wird in ein `Button`-Element konvertiert. Dann können wir das Objekt benutzen und verschiedene Eigenschaften ändern.

Die zweite wichtige Hierarchie, die visuelle Hierarchie (Visual Tree) entscheidet darüber, wie die einzelnen Elemente »gerendert«, also dargestellt werden. Eine Schaltfläche ist nicht einfach nur eine Bitmap, die in der Grafikkarte dargestellt wird, sondern besteht aus einem Rechteck mit abgerundeten Ecken, einem Hintergrund und einem Inhalt (der wiederum beliebig kompliziert gestaltet sein kann). Die visuelle Hierarchie entscheidet also über die Darstellung der einzelnen Teile eines gesamten Elements. Hier wird auch die Reihenfolge in der Z-Richtung berücksichtigt, d.h., wie die Elemente übereinander liegen. Weiterhin spielt die visuelle Hierarchie für das Hit-Testing und die Transformationen der WPF-Elemente eine große Rolle.

Die XAML-Syntax im Überblick

Dieser Abschnitt gibt Ihnen einen kurzen Überblick über die XAML-Syntax. Der XAML-Code soll auch mit normalem VB-Code verglichen werden. Viele Leser werden sicherlich schon Erfahrung mit XML gesammelt haben. Trotzdem soll hier mit einigen Beispielen die Syntax von XAML erläutert werden.

Eine XAML-Hierarchie beginnt immer mit einem Wurzelement, einem Window- oder Page-Element. Dort werden die benötigten XML-Namensräume definiert.

Nun wird die Hierarchie der Benutzerschnittstelle deklariert. Alle Elemente, die in XAML benutzt werden können, existieren als normale Klassen im .NET Framework 3.0. Als Programmierer wissen wir, dass Klassen unter anderem Eigenschaften, Methoden und Ereignisse enthalten. Wir wollen nun betrachten, wie sich das in XAML verhält.

```
<Button Name="btnClear" Width="80" Height="25" Click="OnClear">Löschen</Button>
```

In der obigen XAML-Zeile wird ein Element vom Typ Button mit dem Namen btnClear deklariert. Zur Laufzeit wird also ein Objekt vom Typ Button erzeugt. Die Eigenschaften Width und Height werden gesetzt und außerdem wird das Click-Ereignis mit der Ereignismethode OnClear verbunden. Der Text auf der Schaltfläche wird über die Default-Eigenschaft des Button-Elements gesetzt. Das Gleiche könnten Sie mit folgendem VB-Code erreichen:

```
Dim btnClear As New Button
btnClear.Width = 80
btnClear.Height = 25
btnClear.Content = "Löschen"
AddHandler btnClear.Click, AddressOf OnClear
Me.Content = btnClear
```

Auf Grund des Inhaltsmodells von WPF müssen Sie in der letzten VB-Zeile die Schaltfläche als Inhalt in das Elternelement einbringen. In XAML wird das einfach durch die Deklarationshierarchie erledigt:

```
<Window ...
  Width="300" Height="300">
  <!-- Die folgende Schaltfläche ist der Inhalt des Fensters -->
  <Button Click="OnClear">Test</Button>
</Window>
```

Die Ereignismethode OnClear wird im VB-Code implementiert und hat normalerweise folgende Kopfzeile:

```
Public Sub OnClear(ByVal sender As System.Object, _
                  ByVal e As System.Windows.RoutedEventArgs)
```

Angehängte Eigenschaften (*attached properties*) werden Sie dann benutzen, wenn Sie auf Eigenschaften des Elternelements zugreifen müssen:

```
<Grid>
...
  <Button Grid.Row="0" Grid.Column="0">Button 1</Button>
  <Button Grid.Row="1" Grid.Column="0">Button 2</Button>
</Grid>
```

In den obigen Zeilen wird von den Button-Deklarationen aus auf die Eigenschaften Row und Column des äußeren Grid-Elements zugegriffen.

Oftmals werden Sie in XAML Elemente deklarieren, ohne deren Standorteigenschaft zu benutzen. In diesem Fall können Sie eine gekürzte Schreibweise benutzen. Statt

```
<TextBox Name="text" Width="100"></TextBox>
```

können Sie auch schreiben

```
<TextBox Name"text" Width"100" />
```

Häufig müssen Sie einer Eigenschaft nicht einfach nur eine Zahl oder einen Text, sondern ein komplexes Element, welches wiederum eigene Eigenschaften besitzt, zuweisen. Als Beispiel wollen wir der Eigenschaft RenderTransform einer Schaltfläche (Button) ein Element vom Typ RotateTransform zuweisen, für welches die Eigenschaften Angle, CenterX und CenterY gesetzt werden sollen:

```
<Button Name="btnClear" Width="80" Height="25" Click="OnClear">
  Löschen
  <Button.RenderTransform>
    <RotateTransform Angle="25" CenterX="40" CenterY="12.5" />
  </Button.RenderTransform>
</Button>
```

Zunächst werden für die Schaltfläche selbst einige Eigenschaften »direkt« gesetzt (Name, Width,...). In der Button-Klasse gibt es die Eigenschaft RenderTransform, der nun ein RotateTransform-Objekt zugewiesen werden soll. Darum wird innerhalb der Hierarchie das RotateTransform-Element deklariert und die Eigenschaften Angle, CenterX und CenterY werden gesetzt. Der entsprechende VB-Code sieht folgendermaßen aus:

```
' Button-Element erzeugen
Dim btnClear As New Button
btnClear.Width = 80
btnClear.Height = 25
btnClear.Content = "Löschen"
AddHandler btnClear.Click, AddressOf OnClear

' Rotation erzeugen
Dim rot As New RotateTransform
rot.Angle = 25
rot.CenterX = 40
rot.CenterY = 12.5
```

```
' Rotation der Schaltfläche zuweisen  
btnClear.RenderTransform = rot  
Me.Content = btnClear
```

Diese Hierarchien können beliebig tief geschachtelt werden. Wie Sie am letzten Beispiel sehen können, ist XAML bei der Definition von Hierarchien meistens wesentlich einfacher und übersichtlicher als eine normale Programmiersprache.

Oft gibt es in den WPF-Objekten Eigenschaften, denen Sie mehrere Elemente eines Typs zuweisen können. In eine `ListBox` können Sie mehrere `ListBoxItem`-Elemente einfügen:

```
<ListBox>  
  <ListBox.Items>  
    <ListBoxItem>Test1</ListBoxItem>  
    <ListBoxItem>Test2</ListBoxItem>  
    <ListBoxItem>Test3</ListBoxItem>  
  </ListBox.Items>  
</ListBox>
```

Die drei `ListBoxItem`-Elemente werden in einem `Collection`-Objekt angelegt und der `ListBox`-Eigenschaft `Items` zugewiesen. In dem gezeigten Fall gibt es auch noch eine einfachere Schreibweise:

```
<ListBox>  
  <ListBoxItem>Test1</ListBoxItem>  
  <ListBoxItem>Test2</ListBoxItem>  
  <ListBoxItem>Test3</ListBoxItem>  
</ListBox>
```

Zum Vergleich auch hier der passende VB-Code zur Erzeugung der `ListBox`:

```
Dim lb As New ListBox  
Dim item As New ListBoxItem  
item.Content = "Test1"  
lb.Items.Add(item)  
item = New ListBoxItem  
item.Content = "Test2"  
lb.Items.Add(item)  
item = New ListBoxItem  
item.Content = "Test3"  
lb.Items.Add(item)  
Me.Content = lb
```

Mit XAML können Sie oft sehr einfach ganze Listen von Objekten deklarieren und zuweisen. Im folgenden Beispiel wird ein `MeshGeometry3D`-Element mit Daten initialisiert:

```
<MeshGeometry3D Positions="0,0,0 5,0,0 0,0,5"  
  TriangleIndices="0 2 1"  
  Normals="0,1,0 0,1,0 0,1,0" />
```

In diesem Beispiel wird die Eigenschaft `Positions` mit drei Elementen vom Typ `Point3D` initialisiert. Jedes `Point3D`-Element wird wiederum mit drei `double`-Zahlen initialisiert, die jeweils durch Kommata getrennt in der Liste angegeben werden. Ganz ähnlich wird die Eigenschaft `Normals` gesetzt. Für die Eigenschaft `TriangleIndices` umfasst die Liste nur drei Zahlen, die einfach hinzugefügt werden. Der VB-Code zu diesem XAML-Beispiel sieht folgendermaßen aus:

```
Dim mesh As New MeshGeometry3D
mesh.Positions.Add(New Point3D(0.0, 0.0, 0.0))
mesh.Positions.Add(New Point3D(5.0, 0.0, 0.0))
mesh.Positions.Add(New Point3D(0.0, 0.0, 5.0))
mesh.TriangleIndices.Add(0)
mesh.TriangleIndices.Add(2)
mesh.TriangleIndices.Add(1)
meshNormals.Add(New Vector3D(0.0, 1.0, 0.0))
meshNormals.Add(New Vector3D(0.0, 1.0, 0.0))
meshNormals.Add(New Vector3D(0.0, 1.0, 0.0))
```

Wie Sie in diesem Beispiel leicht erkennen können, gibt es für die Eigenschaften `Positions`, `TriangleIndices` und `Normals` der `MeshGeometry3D`-Klasse immer eine `Add`-Methode, welche die Daten aus den XAML-Listenangaben korrekt verarbeitet und hinzufügt.

Mit diesen wenigen Beispielen haben wir die wichtigsten Syntaxelemente von XAML kennen gelernt und sollten in der Lage sein, die XAML-Hierarchien in diesem Buch zu lesen und zu verstehen. Mit etwas Übung werden Sie dann auch eigenen XAML-Code erstellen können.

HINWEIS Denken Sie daran, dass in Zukunft die XAML-Hierarchien nicht »von Hand« eingegeben, sondern von grafisch orientierten Werkzeugen erzeugt werden (wenn auch nicht gerade vom eingebauten WPF-Designer – Microsoft stellt aber weitere Werkzeuge zur Verfügung, über die Sie sich unter <http://www.microsoft.com/expression/> informieren können).

Ein eigenes XAMLPad

Im Windows-Software Development Kit (SDK) wird ein kleines Werkzeug mitgeliefert, welches eine XAML-Hierarchie in einem Fenster darstellen kann. Das Werkzeug heißt XAMLPad (Abbildung 11.8). Wir wollen zum Schluss dieses Kapitels versuchen, unser eigenes einfaches *MeinXAMLPad* zu entwickeln.

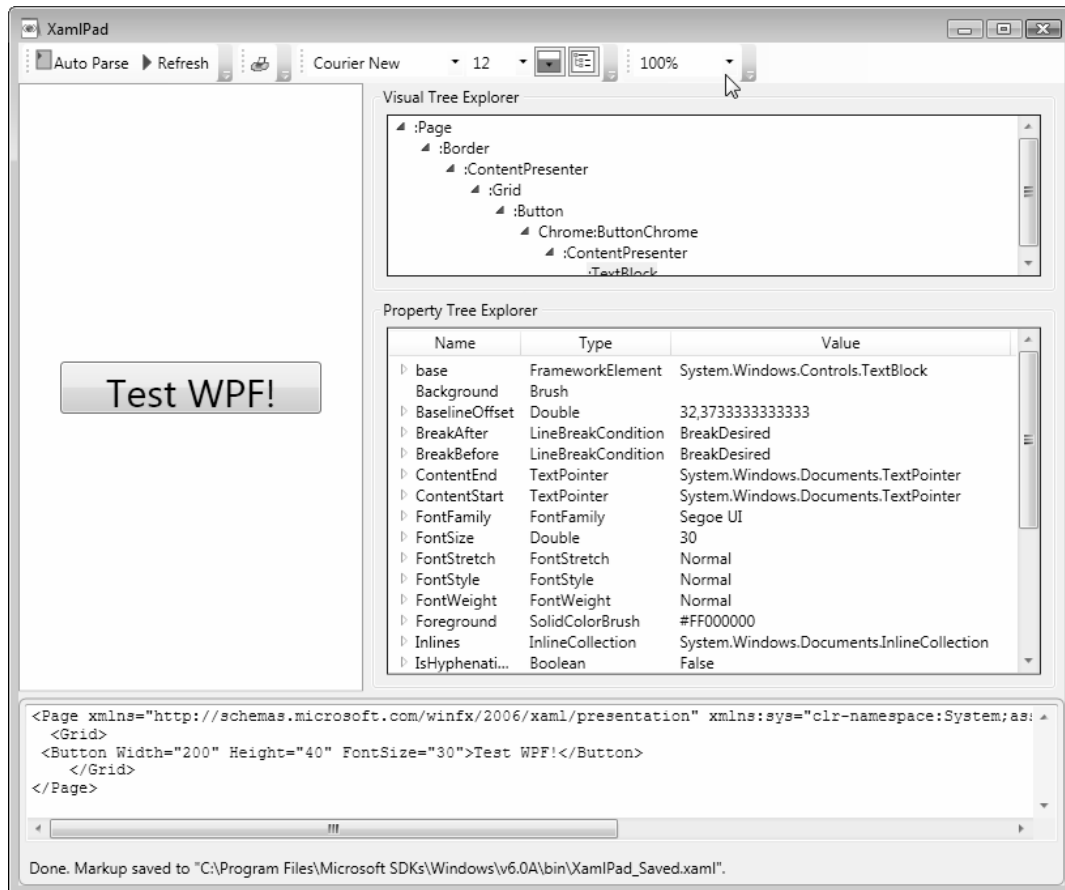


Abbildung 11.8 Das XAMLPad aus dem Windows Software Development Kit

In Abbildung 11.8 wird im unteren Bereich des Fensters eine XAML-Hierarchie eingegeben, die im oberen Bereich visualisiert wird. Das Programm eignet sich gut, um ein bisschen mit XAML zu experimentieren und die Hierarchien für ein WPF-Element zu erforschen. Es eignet sich allerdings nicht dazu, WPF-Applikationen zu entwickeln und zu testen. Sie können mit diesem Werkzeug keinen VB-Code definieren, der z. B. als Ereignismethode für das Click-Ereignis der Schaltfläche aufgerufen werden kann.

Zunächst wollen wir eine einfache Benutzerschnittstelle wie in Listing 11.12 definieren.

```

<Window x:Class="MeinXAMLPad.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="MeinXAMLPad" Height="500" Width="600"
  >
  <Grid>
  <Grid.RowDefinitions>
  <RowDefinition />
  <RowDefinition />

```

```

    <RowDefinition Height="40"/>
  </Grid.RowDefinitions>

  <Grid Name="gridCode" Grid.Column="0" Grid.Row="0" />

  <TextBox Grid.Column="0" Grid.Row="1" Name="textCode"
    VerticalScrollBarVisibility="Visible" HorizontalScrollBarVisibility="Visible"
    FontFamily="Lucida Console" FontSize="14" AcceptsReturn="True" />

  <StackPanel Grid.Column="0" Grid.Row="2" Orientation="Horizontal">
    <Button Name="btnClear" Width="80" Margin="5" Click="OnClear">Löschen</Button>
    <Button Name="btnCompile" Width="80" Margin="5" Click="OnCompile">Darstellen</Button>
    <Button Name="btnExit" Width="80" Margin="5" Click="OnExit">Beenden</Button>
  </StackPanel>
</Grid>
</Window>

```

Listing 11.12 Die Benutzerschnittstelle für MeinXAMLPad

Die hier verwendeten WPF-Steuerelemente (Grid, StackPanel, Button und TextBox) werden wir an späterer Stelle noch genauer kennen lernen. Das verwendete Grid-Element enthält in unserem Beispiel drei Zellen untereinander. Jede Zelle kann weitere WPF-Elemente enthalten. Die oberste Zelle enthält wiederum ein Grid-Element mit dem Namen gridCode. Dieses Grid wird später die WPF-Elemente darstellen, welche wir im darunter liegenden Textfeld eingeben.

Die Texteingabe wird über ein mehrzeiliges TextBox-Element realisiert. Um aus dem VB-Code auf das Element zugreifen zu können, vergeben wir auch hier einen Namen: textCode. Außerdem setzen wir einige weitere Eigenschaften, um die Eingabe etwas zu vereinfachen. Wichtig ist, die Eigenschaft AcceptsReturn auf True zu setzen, damit wir problemlos mehrzeilige Texte eingeben können. Weiterhin können Sie die Benutzbarkeit der Bildlaufleiste (Scrollbar) mit den Eigenschaften VerticalScrollBarVisibility und HorizontalScrollBarVisibility einschalten. Eine andere Schriftart (FontFamily) mit einer etwas größeren FontSize rundet das Bild ab.

Im unteren Teil des Fensters legen wir nun ein StackPanel-Element mit horizontaler Anordnung an. Hier befinden sich mehrere Schaltflächen, die *MeinXAMLPad* steuern. Alle Schaltflächen sind mit Ereignismethoden verbunden, deren Funktion sich aus dem Namen ergibt. Die interessanteste Schaltfläche mit dem Namen btnCompile sorgt für die Übersetzung und Darstellung des eingegebenen XAML-Codes. Der dazugehörige VB-Code ist in Listing 11.13 zu sehen.

```

Imports System
Imports System.Windows
Imports System.Windows.Markup
Imports System.Xml
Imports System.IO

Namespace MeinXAMLPad
  Partial Public Class Window1
    Inherits Windows.Window

```

```
'Das folgende Element enthält den Code für unser WPF-Element
Dim uie As UIElement
Dim code As UIElement
Public Sub New()
    uie = Nothing
    InitializeComponent()
    InitTextBox()
End Sub

Private Sub InitTextBox()
    'Starttext in die Textbox schreiben
    textCode.Text = _
    "<Grid xmlns=""http://schemas.microsoft.com/winfx/2006/xaml/presentation"" _
    & vbNewLine
    textCode.AppendText _
    (" xmlns:x=""http://schemas.microsoft.com/winfx/2006/xaml"" _
    & vbNewLine)
    textCode.AppendText(" >" & vbNewLine & vbNewLine & "</Grid>" & vbNewLine)
    textCode.Focus()
End Sub

Private Sub OnClear()
    If Not uie Is Nothing Then
        'Existierende WPF-Elemente entfernen
        gridCode.Children.Remove(uie)
        uie = Nothing
    End If
    InitTextBox()
End Sub

Private Sub OnCompile()
    'WPF-Element in Grid darstellen
    If Not uie Is Nothing Then
        gridCode.Children.Remove(uie)
    End If

    'WPF-Code aus TextBox übersetzen
    uie = GetTheCode(textCode.Text)

    If Not uie Is Nothing Then
        'WPF im Grid darstellen
        gridCode.Children.Add(uie)
    End If
End Sub

Private Function GetTheCode(ByVal strText As String) As UIElement
    uie = Nothing
    Try
        ' Stream aus Eingabetext erzeugen
        Dim sr As StringReader = New StringReader(strText)
        ' Text aus der Box als XmlReader darstellen
        Dim xr As XmlReader = XmlReader.Create(sr)
        ' Laden und übersetzen des XAML-Codes, Zuordnen zum UIElement
        code = CType(XmlReader.Load(xr), UIElement)
    End Try
End Function
```

```

        Catch ex As Exception
            'Einfachste Fehlerbehandlung
            MessageBox.Show("Fehler im XAML-Code:" _
                & vbNewLine & vbNewLine & ex.Message)
        End Try
        Return code
    End Function

    Private Sub OnExit()
        Me.Close()
    End Sub
End Class
End Namespace

```

Listing 11.13 Die Implementierung der Logik für MeinXAMLPad

Bei der Initialisierung der Applikation wird aus dem Konstruktor der Fensterklasse die Methode `InitTextBox` aufgerufen. Hier wird nur ein XAML-Grundgerüst mit den erforderlichen Namensräumen in das `TextBox`-Element geschrieben, damit wir beim Testen nicht so viel tippen müssen. Der Code für die Schaltfläche *Beenden* ist eigentlich klar. Hier schließen wir einfach das Hauptfenster der Applikation.

Kommen wir nun zur Ereignismethode `OnCompile`, die natürlich die Hauptarbeit macht. Der übersetzte XAML-Code wird in einem `UIElement` gespeichert, welches in der Fensterklasse deklariert wird. Das Objekt `UIElement` liegt in der WPF-Klassenhierarchie ziemlich weit oben:

- `System.Object`
 - `System.Windows.Threading.DispatcherObject`
 - `System.Windows.DependencyObject`
 - `System.Windows.Media.Visual`
 - **`System.Windows.UIElement`**
 - `System.Windows.FrameworkElement`
 - `System.Windows.Controls.Control`
 - `System.Windows.Controls.ContentControl`
 - `System.Windows.Controls.Primitives.ButtonBase`
 - `System.Windows.Controls.Button`

Sie sehen in der obigen Liste, dass ein normales WPF-Steuer-element (hier: `Button`-Element) von `UIElement` abgeleitet wird, sodass wir in unserem Beispiel ein Objekt von diesem Typ für die Speicherung des eingegebenen XAML-Segments gut benutzen können. Wenn bereits ein `UIElement` dargestellt wurde, dann wird dieses Element in `OnCompile` zunächst gelöscht, indem für das Kindelement in `gridCode` die `Remove`-Methode aufgerufen wird. Nun kann der eingegebene XAML-Code über die Methode `GetTheCode` verarbeitet werden. In dieser Methode wird der Text aus dem Eingabefeld `textCode` als `StringReader`-Element geöffnet. Der erzeugte Datenstrom wird nun als `XmlReader` mit der `Create`-Methode geöffnet. Diese Schritte sind erforderlich, da das nun verwendete `XmlReader`-Element XML als Eingabe für die `Load`-Methode benötigt. Dieser Methodenaufruf im `XmlReader` erzeugt nun das `UIElement`, welches schließlich als Kindelement im Grid `gridCode` dargestellt wird.

Wenn Sie die Schaltfläche *Löschen* anklicken, passieren in der Ereignismethode `OnClear` zwei Dinge. Sollten bereits WPF-Elemente im `Grid`-Element `gridCode` dargestellt werden, dann müssen diese mit der `Remove`-Methode gelöscht werden. Nun sehen Sie wieder ein leeres Ausgabefenster; das Element `gridCode` enthält kein `UIElement` mehr. Außerdem wird das `TextBox`-Element wieder initialisiert.

MeinXAMLPad ist nun fertig. Der Aufruf der Applikation zeigt uns Abbildung 11.9. Im Eingabefeld unserer Applikation haben wir ein `StackPanel`-Element mit einer Schaltfläche, einem Eingabefeld und einem Kontrollkästchen eingegeben. Natürlich ist die Deklaration von Benutzerschnittstellen mit unserem Werkzeug nicht so angenehm. Zum einen vermissen wir natürlich die `IntelliSense`-Möglichkeit aus `Visual Studio 2008`, außerdem können wir keine Ereignismethoden definieren und der XAML-Code wird in nacktem Schwarz dargestellt. Wenn Sie Spaß daran haben, können Sie dieses Beispiel natürlich weiter entwickeln.

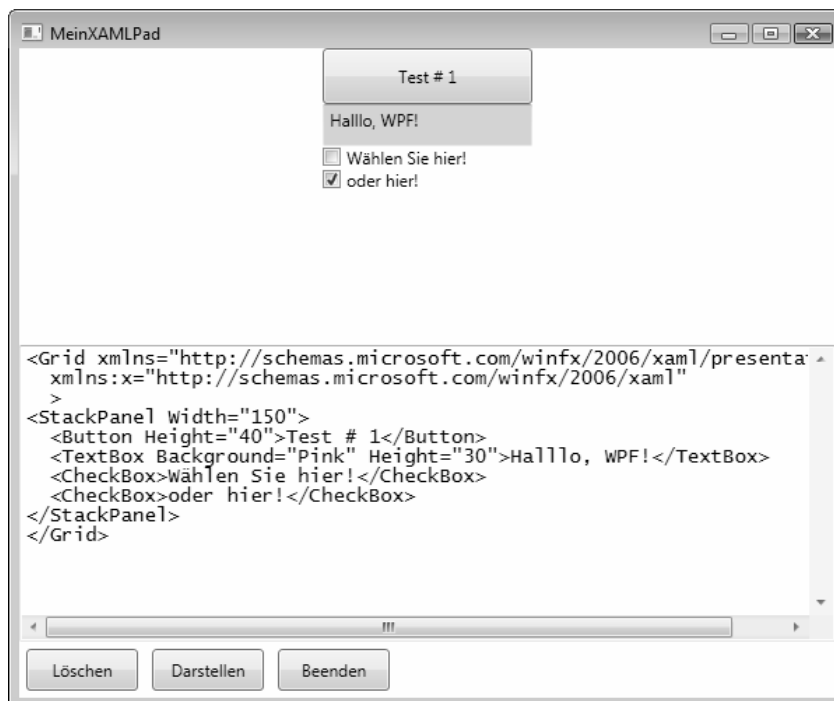


Abbildung 11.9 MeinXAMLPad in der Anwendung

Zusammenfassung

In diesem ersten Kapitel haben Sie die Grundlagen von Windows Presentation Foundation kennen gelernt. Sie wissen nun, dass Code und Design getrennt voneinander vorliegen. Der Code wird in Visual Basic implementiert, während das Design der Benutzeroberfläche in XAML deklariert wird.

Alle WPF-Elemente, egal ob Steuerelemente oder grafische Figuren, werden als Vektorgrafik ausgegeben. Dadurch wird immer eine hervorragende Darstellungsqualität erzielt, auch wenn Elemente der Benutzeroberfläche extrem vergrößert werden.

Die Verbindung von Design und Code wird über den Aufruf von Ereignismethoden aus XAML bzw. über die mit Namen versehenen Elemente aus dem Programmcode vollzogen.

Wenn Sie mit WPF arbeiten, wird niemals XAML-Code zur Laufzeit »interpretiert«. Letztendlich liegt auch die mit XAML deklarierte Benutzeroberfläche als MSIL-Code in der auszuführenden Datei vor. Dieser wird dann zur Laufzeit vom JIT-Compiler der Common Language Runtime in Maschinencode übersetzt. Somit sollten WPF-Benutzerschnittstellen ähnlich schnell wie bereits bekannte .NET Frameworks arbeiten.

Die Trennung von Code und Logik mit WPF bringt uns folgenden großen Vorteil: Sie bringen einen Top-Designer mit einem Top-Softwareentwickler zusammen und die beiden entwickeln für Sie eine Top-Software!