Kapitel 6

Einführung in Windows Presentation Foundation

In diesem Kapitel:

Klaus Löffelmann, Sarika Calla Purohit: Microsoft Visual Basic 2010 - Das Entwicklerbuch, Microsoft Press 2010 (ISBN 978-3-86645-535-1)

Das Auge isst mit. Diesen Spruch habe ich mir von meiner Mutter schon anhören müssen, als ich als kleiner Junge den Tisch decken musste – na ja, sagen wir: entsprechende Versuche veranstaltet habe, diesen ihren ästhetischen Bedürfnissen entsprechend zu decken. Mein eigener Grundsatz »Form follows function«, in leicht abgeänderter Version im Deutschen eher als »Funktion vor Design« bekannt, widersprach diesem Grundsatz meiner Ma und war – ich gebe es offen zu – sicherlich auch Beweggrund, mich seit Visual Basic 4.0 eher auf Windows Forms-Anwendungen als auf Webanwendungen zu spezialisieren. Und mich im Übrigen auch nicht als Innenarchitekt zu versuchen. Viele Webanwendungen lassen sich sicherlich grafisch und designtechnisch ansprechend, auf alle Fälle aber sehr viel ansprechender als Windows Forms-Anwendungen designen. Leider aber nur von den richtigen Leuten, zu denen ich zugegeben leider nicht gehöre.

Was ist Windows Presentation Foundation?

Und dann brachte Microsoft mit .NET Framework 3.0 das Grafikframework *Windows Presentation Foundation* heraus. Erklärtes Ziel war es eigentlich, und daraus leitet sich auch der Name ab, eine Technik zu schaffen, mit der Informationen in grafisch vollendeter Form vor allem präsentiert werden können. Aber nicht nur. WPF, wie Windows Presentation Foundation gerne abgekürzt genannt wird, sollte auch dazu dienen, diese Grafik als Benutzerschnittstelle zu nutzen, also um mit dem Benutzer zu interagieren – mit Maus, mit Stift oder, wie es mit Windows 7, dem Nachfolger von Windows Vista der Fall ist, auch mit fingerberührungsempfindlichen Displays auf Notebooks[1] oder speziellen Tischdisplays.

Und das bedeutet für uns Windows Forms-Programmierer: umdenken, und zwar *heftigst*, wie wir Westfalen sagen. Der deutsche Spruch passt nicht mehr, denn nunmehr gilt: *Funktion ist Design*. Mit dem englischen Idiom sind wir immer noch gut aufgehoben, denn es ändert sich nichts an »Form follows function«.

WPF unterlag in den letzten zwei Jahren einem kleinen, von keinem und dann wieder von allen vorangetriebenen Evolutionsprozess. Denn obwohl Microsoft WPF ursprünglich niemals als designierten Nachfolger von Windows Forms konzipiert hatte, zeichnet es sich immer mehr ab, dass genau das passiert: Viele der immer noch umfangreichen Visual Basic 6.0- und C++-COM-Anwendungen müssen auf eine neue Plattform migriert werden, da viele dieser Anwendungen unter Vista, Windows 7 und Windows Server 2008 erhebliche Probleme bereiten und unter den immer häufiger anzutreffenden 64-Bit-Versionen von Windows überhaupt nicht mehr funktionieren. Die einzige Lösung dazu lautet: Die alten Anwendungen müssen auf bzw. zu .NET migriert werden; Visual Basic 6.0 und VBA vorzugsweise auf .NET-Versionen von Visual Basic. Und während dieses Migrationsprozesses, während also die Konzeptionen für die beste Vorgehensweise einer solchen Umstellung von den Tausenden verschiedenen Entwicklerteams, die es da draußen gibt, ausgearbeitet wird, stellt sich diesen erfahrungsgemäß immer wieder die gleiche Frage: Wie sollen die neuen Frontends und Benutzeroberflächen gestaltet werden? In diesem Prozess kommen immer mehr Teams zu dem Schluss, dass sich WPF *durchaus* auch als UI-Ersatz[2] für Windows Forms eignet. Die langfristigen Benefits, die sich daraus ergeben, scheint man als wichtiger zu erachten als die Nachteile und den erheblichen Zusatzaufwand, den man vor allen Dingen zurzeit noch bei der Ablösung von Windows Forms durch WPF hinnehmen muss. Und dieser Zeitaufwand liegt vor allem an folgendem:

---

[1] Demo gefällig? YouTube hilft, und *http://tinyurl.com/6gaoln* zeigt wo.

[2] UI: Abkürzung für User Interface, zu Deutsch *Benutzeroberfläche.*

- Der Einarbeitungsaufwand in die Technik von WPF ist ganz erheblich. Das gilt vor allen Dingen für Teams, die aus der Visual Basic 6.0-Welt kommen und die sich ohnehin erst an das objektorientierte Programmierkonzept gewöhnen müssen. Und aufgepasst: Die Rede ist dabei von objekt*orientiert* – viele Entwickler, die selbst wirklich alles aus Visual Basic 6.0 herausgeholt haben und dessen Tücken und Fallen aus dem Effeff kennen, haben dennoch – konzeptbedingt – bestenfalls objekt*basiert* programmieren können. Das ist ein großer Unterschied, gerade was den bestmöglichen Aufbau einer Anwendung anbelangt, der in .NET-Versionen von Visual Basic – objektorientiert – zu einem komplett anderen Ansatz führen kann, bei dem im Vergleich zur Ausgangsanwendung kein Stein auf dem anderen verbleibt.
- Der Aufbau von Formularen, oder das, was dem am nächsten kommt, unterliegt einem ganz anderen Konzept als in Windows Forms. Es gibt hier – ähnlich wie bei HTML zur Beschreibung von Webseiten – eine an XML angelehnte Sprache namens XAML (sprich: gsämmel), die dafür zuständig ist. Hier ist viel Lernaufwand erforderlich, insbesondere dann, wenn einem schon das Konzept von Dialekten wie HTML, die den Seitenaufbau beschreiben, eher fremd ist.
- Windows Forms verwendet einen Designer, der altbewährt ist. Selbst der .NET-Designer für Windows Forms-Anwendungen hat bereits fünf Jahre auf dem Buckel, und schon der allererste Designer dieser .NET-Version basierte konzeptionell weitestgehend auf dem, was zuletzt in Visual Basic 6.0 zu finden war. Der WPF-Designer hat – wohlwollend ausgedrückt – selbst in .NET 4.0 noch einiges an Verbesserungspotenzial, und so sollten Sie sich darauf einrichten, dass Sie viele Dinge, die Sie unter Windows Forms natürlich interaktiv mit Designerunterstützung machen würden, für WPF direkt in XAML eintippen müssen. Während Sie die Ergebnisse Ihrer Arbeit anfangs oft mit positivem Erstaunen kommentieren werden, sollten Sie sich allerdings nicht darüber wundern, dass Ihnen auf dem Weg dahin sicherlich der eine oder andere Fluch entgleiten wird.

HINWEIS Wenn Sie professionelle WPF-Anwendungen entwickeln wollen, werden Sie um den Einsatz einer speziell auf WPF abgestimmten Designersoftware nicht herumkommen. Ihr Name: Expression Blend – und es gibt sie inzwischen in der vierten Version. Der unverständliche Nachteil: Expression Blend ist NICHT in Visual Studio 2010 enthalten – Sie müssen es wohl oder übel dazukaufen. Spätestens hier bietet sich der Kauf eines so genannten MSDN-Abos an: Schon in der MSDN Professional Edition dieses Abonnements sind sowohl Visual Studio 2010 Professional als auch Expression Blend 4 enthalten. Und natürlich erhalten Sie dazu alle Betriebssysteme, Server und die gängigsten Anwendungsprogramme aus dem Hause Microsoft für Entwicklungszwecke obendrein. Mehr zum Thema MSDN erfahren Sie unter *http://tinyurl.com/3a9dkm7*.

Was ist so neu an WPF?

Ein guter Freund sagte mir einst, er hasse an vielen Computerbüchern, dass sie bestimmte Sachverhalte nicht einfach Punkt für Punkt an simplen, einfachen Beispielen erklären. Und er bezog sich damit indirekt auch auf die Rohfassung des Textes, den Sie gerade lesen. An dieser Stelle, so seine Meinung, seien schon zwei Seiten ins Land gegangen, und er wisse immer noch nicht, wie man eine einfache Linie mit WPF auf dem Bildschirm malt – das könne doch nicht so schwer sein. Und da hat er vielleicht irgendwie Recht, gleichzeitig aber auch nicht verstanden, was WPF ist. Denn seine Frage impliziert schon, dass er meint, dass die Rendering[3]-Engine von WPF so funktionieren würde wie das, was er kenne, sodass beispielsweise aus einer Linie ein Rechteck und aus einem Rechteck eine Schaltfläche entstünde. So war es aber nicht gedacht.

[3] Lustigerweise gibt es keine wirklich passende Übersetzung, die das Konzept des Renderings des »Entstehenlassens oder des Umsetzens« so ganz passend wiedergibt. Was gemeint ist: *http://tinyurl.com/34omj3v*

Dummerweise ist es nämlich völlig anders. Wenn Sie aber verstehen wollen, wie WPF funktioniert, sollten Sie etwas über die Entstehungsgeschichte von WPF wissen. Warum? Ganz einfach und als Metapher erzählt: Sonst geht es Ihnen wie diesem Zeitgenossen, der bislang nur das Waschbrett kannte. Und der sich beim erstmaligen Anblick der Waschmaschine fragt, wie man so etwas Unergonomisches bauen könne wie eine Waschmaschine, bei der die Waschreibe auf der Innenseite einer Trommel angeordnet sei und man zum Waschen der Wäsche ja in diese hineinkriechen müsse.

Das einfache Beispiel, das der vorhin erwähnte Freund verlangt, kann er bekommen. Doch nicht ohne den angekündigten Exkurs in Sachen *Geschichte von Windows* und wie dessen grafische Elemente bislang den Weg auf den Bildschirm fanden.

25 Jahre Windows, 25 Jahre gemalte Schaltflächen

Windows-Anwendungen sind mittlerweile seit mehr als 25 Jahren zugegen. Und sind wir ehrlich: Auch wenn es Windows 1.0 und 2.0[4] schon *gab*, wurde erst Windows 2.11 als Bundle-Version mit Adobe Pagemaker etwas bekannter. Und erst mit Windows 3.0 kam der Mainstream-Durchbruch dank neuem Look & Feel bei der Bedienung von Anwendungen, dessen grundsätzliches Konzept sich später allerdings selbst mit Windows 95 und Windows NT sowie über Windows 2000 und Windows XP hinweg im Wesentlichen kaum änderte.

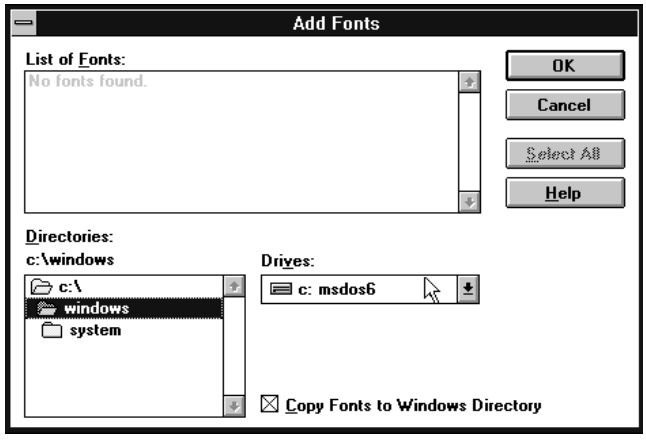

Abbildung 6.1 Der Dialog zum Hinzufügen von Schriftarten in einem englischen Windows 3.0 ...

Sicherlich: Windows ist vor allen Dingen, was die auf Windows NT basierenden Versionen anbelangt, im Lauf der Zeit deutlich stabiler geworden; Windows 95 brachte genau wie Windows NT 4.0 viele neue grafische Elemente, die mit Windows XP nochmals bunter, moderner und mit einer Menge an Farbverläufen gekrönt wurden. Doch eines hat sich an und für sich seit der ersten Version von Windows nicht geändert: die Art und Weise, wie die Grafik ihren Weg auf die Mattscheibe findet.

[4] Mehr dazu liefert *http://www.winhistory.de/more/win1.htm*.

Abbildung 6.2 ... und in Windows Vista. Hier ist deutlich zu sehen, dass sich über all die Jahre wirklich nicht viel am Prinzip der Windows-Bedienung und des Renderings von Windows Forms-Dialogen geändert hat

Das geschieht nämlich im Grunde genommen bei 99% aller Programme mithilfe von GDI (*Graphics Device Interface*) von Windows. Diese API[5] beinhaltet Funktionen, mit der zweidimensionale Zeichenfunktionen (das Malen von Linien, Rechtecken, Kreisen und verschiedenen Pinselstärken, Formen und Mustern) möglich sind, die durch Grafikkarten unterstützt werden. *Unterstützung von Grafikkarten* bedeutet dabei, dass die Treiber der Grafikkarte bestimmte Befehle beispielsweise zum Zeichnen einer Linie nicht selbst durch Algorithmen und mithilfe des Prozessors umsetzen, sondern sie vielmehr an die Grafikkarte weiterreichen können, damit die Grafikkarte diese Linie selbst malt – natürlich viel, viel schneller, als es der Prozessor Ihres Computers jemals könnte. Und wie das in der Praxis ausschaute, daran soll uns das folgende kleine Beispiel noch einmal erinnern:

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2008 Entwicklerbuch\Kapitel 06\SimpleSampleGDIPlus
```

Öffnen Sie dort die Projektmappe (.SLN-Datei) *SimpleSampleGDIPlus*.

[5] *API* bedeutet *Application Programming Interface,* in etwa Anwendungsschnittstelle zum Programmieren.

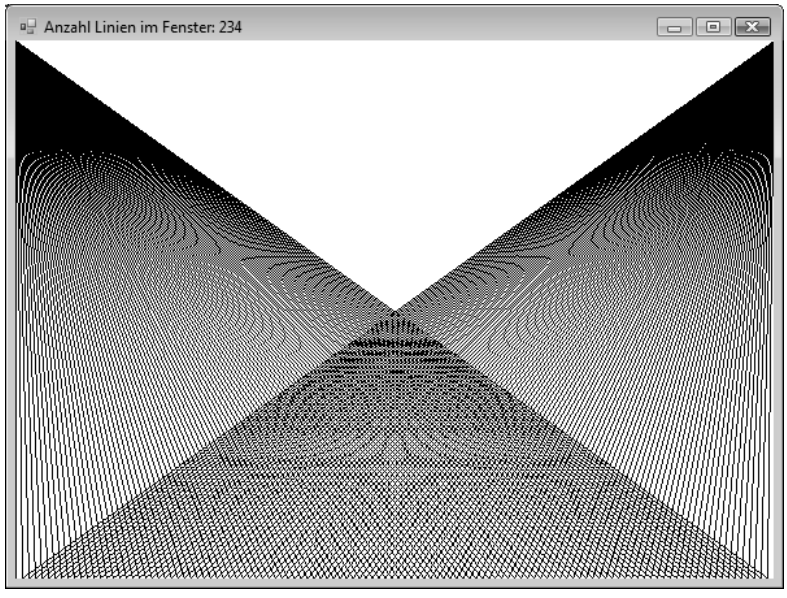

Abbildung 6.3 Das Ergebnis nach einem Doppelklick ins Fenster – eine Reihe von aufgefächerten Linien zeigt eindrucksvoll den bei Fernsehmachern so gefürchteten Moiré[6]-Effekt

Die Figur, die dort gezeichnet wird (Abbildung 6.3), interessiert uns allerdings nur am Rande. Vielmehr soll Augenmerk darauf gelegt werden, *wie* die Linien auf den Bildschirm kommen und wie Windows dafür sorgt, dass sie dort verbleiben. Und dazu schauen wir uns erst einmal das kleine, zur Abbildung gehörige Listing an, das die Grafik beim Doppelklick ins Fenster bringt:

```
Public Class Form1

    Private mydoppelGeklickt As Boolean

    'OnLoad überschreiben - damit das Load-Ereignis behandeln, ...
    Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
        MyBase.OnLoad(e)
        '...und hier den Fenstertext setzen.
        Me.Text = "Doppelklicken Sie in das Fenster, um die Grafik darzustellen."
    End Sub

    'Wir hätten auch das Load-Ereignis behandeln können...
    Private Sub Form1_Load(ByVal sender As Object, ByVal e As System.EventArgs) _
                Handles Me.Load
        'aber warum ein Ereignis der Instanz in der Instanz behandeln,
        'die das Ereignis auslöst? Dann lieber an der Stelle eingreifen,
        'die das Ereignis auslösen, und das ist OnLoad.
    End Sub
    'Grafik wird ins Fenster gemalt, sobald der Anwender doppelklickt.
    Protected Overrides Sub OnDoubleClick(ByVal e As System.EventArgs)
        MyBase.OnDoubleClick(e)
        Dim g As Graphics = Graphics.FromHwnd(Me.Handle)
```

[6] Sprich: Moareh – mehr dazu zeigt *http://tinyurl.com/bt6zjf*.

```
        DrawDemo(g)
        mydoppelGeklickt = True
    End Sub

    'Und hier wird gemalt.
    Public Sub DrawDemo(ByVal g As Graphics)

        'Erstmal den Fensterinhalt löschen.
        g.Clear(Color.White)

        'tatsächliche Breite des Clientbereichs ermitteln und merken
        Dim tatsächlicheBreiteGemerkt = Me.ClientSize.Width
        Dim linienZähler = 0

        'Jeden 5. Pixel berücksichtigen
        For x = 0 To tatsächlicheBreiteGemerkt Step 5

            'Erst die Linie von links oben nach rechts unten malen.
            g.DrawLine(Pens.Black, 0, 0, x, Me.ClientSize.Height)

            'Und dann noch eine von rechts oben nach links unten.
            g.DrawLine(Pens.Black, tatsächlicheBreiteGemerkt, _
                               0, tatsächlicheBreiteGemerkt - x, _
                               Me.ClientSize.Height)

            'Linienzähler aktualisieren
            linienZähler += 2
        Next

        'Im Fenstertitel über die Anzahl gemalter Linien informieren.
        Me.Text = "Anzahl Linien im Fenster: " & linienZähler
    End Sub
End Class
```

Zum Malen von Inhalten in GDI und GDI+ (zu GDI+ etwas später mehr) ist dabei Folgendes zu sagen: Diese sind höchst volatil. Wenn Sie nämlich nicht selbst dafür sorgen, dass der Fensterinhalt, nachdem er zerstört wurde, weil etwas anderes über das Fenster geschoben wurde oder jemand das Fenster vergrößert oder verkleinert hat, restauriert wird, dann ist der Inhalt eben weg, wie Abbildung 6.4 anschaulich zeigt. Hier wurde das Fenster zunächst ein paar Pixel horizontal und vertikal vergrößert, und anschließend hat sich auch noch der der Windows-Taschenrechner über das Fenster geschoben.

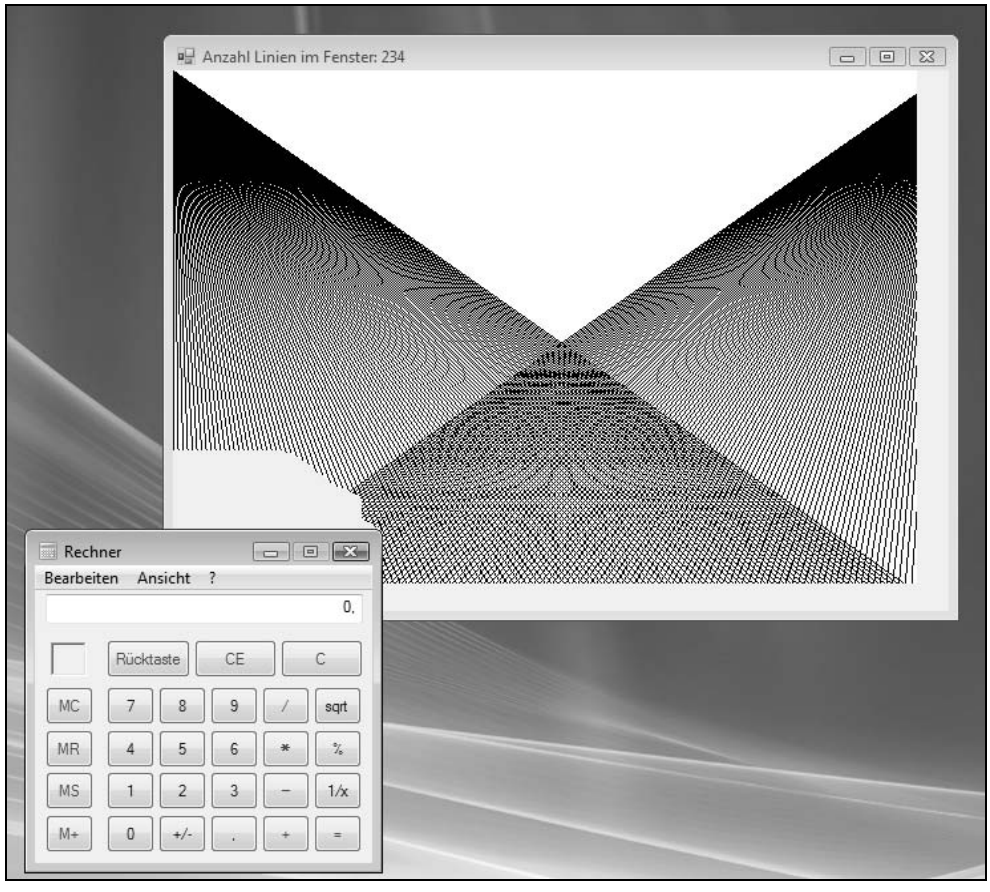

Abbildung 6.4 Um das Aktualisieren des Fensterinhalts muss sich der Entwickler unter GDI oder GDI+ selbst kümmern. Tut er es nicht, gibt es beim Vergrößern oder beim Verdecken des Fensters Effekte, wie hier in der Abbildung zu sehen.

HINWEIS Ab Windows Vista müssen Sie die Aero-Darstellung ausschalten, um das ursprüngliche Verhalten von GDI nachvollziehen zu können. Unter Windows Vista und Windows 7 bekommt jede Anwendung bei Aero ohnehin einen dedizierten Videospeicherbereich zugewiesen, sodass sich Fenster aus Anwendungssicht gar nicht mehr überlappen können, und Sie dieses Verhalten auch gar nicht bemerken. Deswegen übrigens »wirkt« Vista auch soviel langsamer als Windows 7: Ganz vereinfacht ausgedrückt, sind Zugriff auf GDI-Funktionen unter Windows 7 besser optimiert und können sich nicht gegenseitig so sehr beeinflussen. Darüber hinaus sind Vista-GDI-Operationen nicht hardwareunterstützt, ab Windows 7 ist das wieder der Fall. Bei Windows Vista können sich Anwendungen beim Zugriff auf das GDI obendrein schon mal gegenseitig behindern, und wirken daher nicht so flüssig wie unter Windows 7. Die reine Performance von Anwendungen ist aber unter Windows Vista oftmals sogar schneller als unter Windows 7; die Bildschirmdarstellung von GDI-Operationen ist unter Windows 7 hingegen schneller, da hardwareunterstützt.

Damit solche Probleme wie in Abbildung 6.4 nicht passieren können, muss der Entwickler selbst Hand anlegen: Im `Paint`-Ereignis, das dann ausgelöst wird, wenn sich irgendetwas vor das Fenster legt und das Neuzeichnen des Inhalts beim Wiederhervorholen des Fensters erforderlich macht, sorgt er dafür, dass die Anwendung darüber informiert wird, dass der Fensterinhalt aktualisiert werden muss. Ein Ergänzen der folgenden Zeilen würde in unserem Beispiel also bereits für Abhilfe schaffen.

```
'Wird ausgelöst, wenn der Fensterinhalt zerstört wurde
 'und neugezeichnet werden muss
 Protected Overrides Sub OnPaint(ByVal e As System.Windows.Forms.PaintEventArgs)
     MyBase.OnPaint(e)

     'Nur wenn der Doppelklick bereits stattgefunden hat.
     If mydoppelGeklickt Then
         '
         DrawDemo(e.Graphics)
     End If
 End Sub
```

Gegebenenfalls muss er auch ein entsprechendes `Resize`-Ereignis behandeln, das ausgelöst wird, wenn sich die Fenstergröße durch Verkleinern oder Vergrößern des Fensters ändert.

Und dieses Prinzip des Darstellens von Dokumenteninhalten gilt für Windows Forms seit der ersten Windows-Version bis heute. Und nicht nur für Grafiken, die in Fenstern dargestellt werden, sondern für alles. Und dann wiederum doch nur für Grafiken, die in Fenstern dargestellt werden, denn vielen Leuten ist es überhaupt nicht bewusst, wieso Windows Windows heißt. Warum? Weil nicht nur Dokumenteninhalte jeglicher Form in Fenstern, die auch als solche zu erkennen sind, dargestellt werden. Vielmehr ist nämlich jedes Element in Windows – mit wenigen Ausnahmen – ein Fenster. Eine Schaltfläche ist ein Fenster. Eine ListBox ist ein Fenster. Ein geöffnetes Kombinationslistenfeld sind zwei Fenster – die TextBox ist eines und die Liste darunter ein weiteres. Streng genommen ist auch ein ToolTip oder eine Sprechblase des Systray (Infobereich) ein Fenster, und alle diese Windows-Klassen müssen dafür sorgen, dass sie ihren Inhalt neu malen, in dem Moment, wo dieser durch Überlappung mit einem anderen Fenster zerstört wurde. Die Grafikkarte hilft dabei hardwaremäßig nur wenig:

- Sie kopiert zwar Bildblöcke in rasender Geschwindigkeit.
- Sie zeichnet auch Grundlinienfiguren wie Linien, Rechtecke, Ellipsen und füllt diese mit Inhalten. Das macht sie bis heute aber leider nur, wenn die nativen Zeichenbefehle des Betriebssystems, genau gesagt von GDI, verwendet werden. Die .NET-Grafikbefehle werden von einer moderneren Grafikbibliothek umgesetzt: GDI+. Diese lässt leider auf vielen Grafikkarten die entsprechende Hardwareunterstützung vermissen, sodass sie zwar die deutlich komfortableren Grafikoperationen anbietet, diese sind aber *deutlich* langsamer als ihre GDI-Pendents.

HINWEIS Zwei Ausnahme betreffen einerseits das Rendern von Text unter .NET. Hier gibt es den `TextRenderer`, der tatsächlich als einziges direktes Grafikobjekt von GDI Gebrauch macht. Zum Zeichnen bestimmter UI-Elemente gibt es darüber hinaus auch die `ControlPaint`-Klasse im Namespace `System.Windows.Forms`, deren statische Methoden teilweise auch von GDI Gebrauch machen (zum Beispiel `DrawReversibleFrame` zum invertierten Zeichnen eines Rahmens oder `DrawReversibleLine` zum invertierten Zeichnen einer Linie).

Spielende Protagonisten

Grafikkarten können allerdings schon seit etlichen Jahren erheblich mehr als nur diese »alten Kamellen« in Sachen Grafikrendering. Das so genannte DirectX gibt es bereits seit Windows 95, und die Älteren unter uns erinnern sich sicherlich noch an den ersten Egoshooter *Wolfenstein 3D*, der zwar einen sehr umstrittenen Spielgegenstand lieferte, dessen Entwickler der Firma id Software aber nichtsdestotrotz demonstrierten, was eigentlich aus einem Bürocomputer der frühen 1990er-Jahre in Sachen Spiele und damit natürlich in Sachen

Grafik und Sound herauszuholen war. Das grundsätzliche Problem der damaligen Zeit war allerdings, dass für diese enorm hohe Performance eine Anwendung wirklich alleine und ohne störende »Mitbewerber« laufen musste. Und das war, obwohl Windows 3.11 schon als Quasi-Multitasking-Betriebssystem existierte, eben nur unter MS-DOS möglich. DirectX erst lieferte die nötigen Voraussetzungen, damit das – inzwischen war Windows 95 auf dem Markt – auch innerhalb eines Multitasking-Betriebssystems möglich wurde. Und auch hier war es wieder die Firma id Software, die letzten Endes – und auch erst mit der Version 3.0 von DirectX – durch die Portierung des Computerspiels Doom auf Windows 95 zum Geburtshelfer von DirectX wurde. Denn erst mit dieser Version gab es genügend Spieleentwickler, die DirectX als echte Multimedia-Alternative zum Stand-alone-Konzept von MS-DOS wahrnahmen, und DirectX gewann die nötige Akzeptanz.

Und gerade die Gamer sahen von diesem Zeitpunkt an atemberaubende Benutzeroberflächen für ihre heiß geliebten Spiele, die nur durch dreidimensionale, mit virtueller Realität ausgestattete KI-Systeme in Sciencefiction-Filmen übertroffen wurden. Mal ehrlich: Für welchen Windows-Benutzer der damaligen Zeit war der Film *Enthüllung* mit Michael Douglas und Demi Moore kein benutzeroberflächentechnischer Hochgenuss – wenn auch mit einem leicht bitteren Beigeschmack, nämlich zu wissen, dass man sich 45 Minuten später wieder am seit Jahren bekannten und deshalb recht eintönigen User Interface von Windows NT 4.0 SP4a der realen Welt einfinden musste.

Und genau dieses Problem sollte uns Windows-Benutzern dann noch die nächsten ca. zehn Jahre erhalten bleiben, sodass sich viele Benutzer und sicherlich nicht nur die Softwareentwickler unter uns sehnlichst wünschten, wenigstens eine kleine Bewegung in Richtung Spiele-UIs erleben zu dürfen. Doch wir sollten von Windows 2000, dessen zahlreichen Service Packs und auch von Windows XP und dessen leider weniger zahlreichen Service Packs enttäuscht bleiben. Doch dann kamen Windows Vista und .NET Framework 3.0.

Mit Windows Vista gab es nämlich auch endlich ein neues Grafiksystem, das die Darstellung von bestimmten Dingen über eine Funktionsbibliothek auf der Basis von DirectX erlaubte – genau das war Windows Presentation Foundation, kurz WPF genannt. WPF-basierende Anwendungen laufen am flüssigsten und am – wenn ich dieses Wort einmal kreieren darf – *hardwareunterstütztesten* ab Windows Vista. Mit einer besonderen Version kommen aber auch Windows XP-Benutzer in den Genuss, WPF-basierende Anwendungen einzusetzen, aber eben nicht in dieser hochwertigen Darstellungsvollendung wie der von Windows Vista.

Zum Zeitpunkt, zu dem diese Zeilen entstehen, ist die Multimedia-Entwicklergemeinde, was DirectX anbelangt, inzwischen bei der Version 11 angelangt. Diese beinhaltet die neusten und aufwändigsten Multimedia-Funktionsbibliotheken, die es jemals unter einer Windows Version gab. Und wer die Entwicklung von Windows Vista mitbekommen hat, der weiß nicht nur, dass DirectX seit Version 10 ausschließlich Vista-Benutzern und -Entwicklern vorbehalten ist, sondern auch, was es für ein unsäglicher Akt war, endlich in den Genuss stabiler NVIDIA- und ATI-Grafikkartentreiber für die volle DirectX 10-Unterstützung zu gelangen.

Theoretisch konnten wir Entwickler mit WPF, das – der Vollständigkeit halber erwähnt – übrigens schon auf Basis von DirectX 9 funktioniert, seit .NET Framework 3.0 bzw. Windows Vista auch anspruchsvolle Anwendungen entwickeln. Die Betonung liegt auf theoretisch, denn es gab ein kleines Problem: Es war zu diesem Zeitpunkt nämlich kein wirklich funktionierendes Tool verfügbar, mit dem die Entwicklung möglich gewesen wäre. Lediglich ein hastig zusammengefrickelter Aufsatz für Visual Studio 2005 musste zum Lernen und für die ersten Gehversuche herhalten; an eine richtige Entwicklung von WPF-Anwendungen war zu diesem Zeitpunkt noch nicht zu denken.

Klaus Löffelmann, Sarika Calla Purohit: Microsoft Visual Basic 2010 - Das Entwicklerbuch, Microsoft Press 2010 (ISBN 978-3-86645-535-1)

All diese Ereignisse der jüngeren WPF-Geschichte sorgten so dafür, dass wir erst mit Windows Vista SP1 und mit Visual Studio 2008 SP1 bzw. .NET Framework 3.5 SP1 so richtig in WPF loslegen konnten. Naja, vielleicht nicht sofort. Denn nachdem wir uns das bislang gültige Konzept noch mal in Erinnerung gerufen haben, wie Grafik mit GDI bzw. GDI+ ihren Weg auf den Bildschirm findet, und was man machen muss, damit eine Grafik auch persistent bleibt (oder zumindest so wirkt), wollen wir meinen Kollegen Marcus jetzt auch noch mal mit der notwendigen Demo versorgen, die zeigt, wie man in WPF einerseits Linien zeichnet, und wieso es andererseits um das Zeichnen von Linien mit WPF – so, wie wir es von GDI kennen – gar nicht gehen darf.

BEGLEITDATEIEN Die Begleitdateien zum folgenden Beispiel finden Sie im Verzeichnis:

```
...\VB 2010 Entwicklerbuch\Kapitel 06\SimpleSampleWPF
```

Öffnen Sie dort die Projektmappe (.SLN-Datei) *SimpleSampleWPF*.

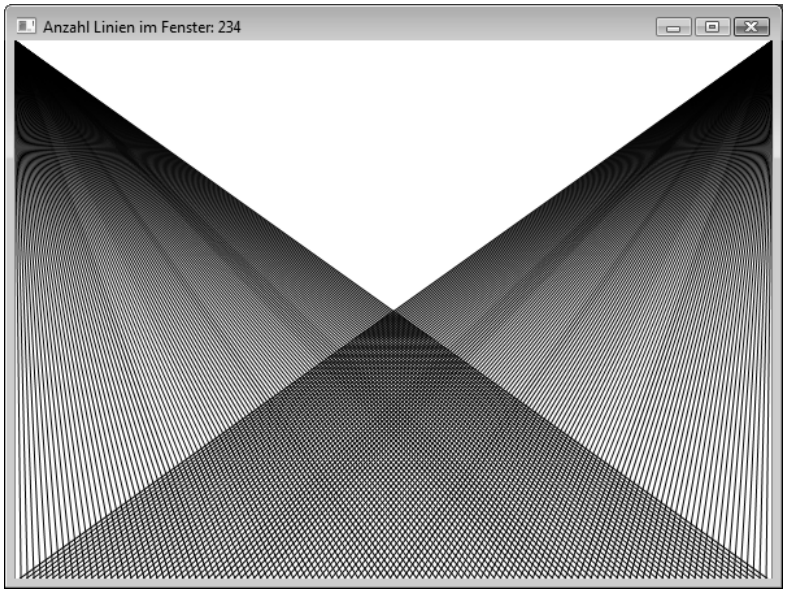

Abbildung 6.5 Die WPF-Version des Linienzaubers sieht erst mal der GDI-Version recht ähnlich ...

Auf den ersten Blick scheint das Ergebnis sich von der GDI+-Version nicht sonderlich zu unterscheiden. Die Linien wirken aber irgendwie sauberer gezeichnet, und im Gegensatz zu GDI+ läuft das Zeichnen jedenfalls unter Vista ordentlicher, mit weniger Flimmern und subjektiv ein wenig schneller ab.

Erste wirkliche Prinzipunterschiede fallen aber bereits auf, wenn Sie das gleiche Spielchen mit diesem Fenster machen, wie wir es versuchsweise in Verbindung mit Abbildung 6.4 (Seite 196) gemacht haben. Jetzt geht beim Überlappen mit einem anderen Fenster nichts mehr verloren – auch wenn der eigentliche Malvorgang, anders als im ersten Beispiel, im `SizeChanged`-Ereignis des Fensters passiert und dementsprechend für das Neuzeichnen des Inhalts beim Vergrößern oder Verkleinern bereits gesorgt ist, wie das folgende Listing dieses Beispiels zeigt:

```
Imports System.Windows.Media.Animation

Class Window1

    Private Sub Window1_SizeChanged(ByVal sender As Object, ByVal e As
System.Windows.SizeChangedEventArgs) Handles Me.SizeChanged
        'meineLeinwand kommt aus der XAML-Definition
        'dort haben wir das vordefinierte Grid rausgeschmissen
        'und durch die Leinwand (Canvas) ersetzt.
        meineLeinwand.Children.Clear()

        'tatsächliche Breite der Leinwand ermitteln und merken
        Dim tatsächlicheBreiteGemerkt = meineLeinwand.ActualWidth
        Dim linienZähler = 0

        'Jeden 5. Pixel berücksichtigen
        For x = 0 To tatsächlicheBreiteGemerkt Step 5

            'Neues Linienobjekt anlegen und Parameter setzen
            Dim eineLinie As New Line()
            'Parameter ensprechend setzen
            eineLinie.Stroke = Brushes.Black
            eineLinie.X1 = 0
            eineLinie.Y1 = 0
            eineLinie.X2 = x
            eineLinie.Y2 = meineLeinwand.ActualHeight
            eineLinie.HorizontalAlignment = HorizontalAlignment.Left
            eineLinie.VerticalAlignment = VerticalAlignment.Center
            eineLinie.StrokeThickness = 1
            'Der Leinwand hinzufügen
            meineLeinwand.Children.Add(eineLinie)

            'Die entgegenlaufende Linie definieren
            eineLinie = New Line()
            'Wieder Parameter entsprechend setzen
            eineLinie.Stroke = Brushes.Black
            eineLinie.X1 = tatsächlicheBreiteGemerkt
            eineLinie.Y1 = 0
            eineLinie.X2 = tatsächlicheBreiteGemerkt - x
            eineLinie.Y2 = meineLeinwand.ActualHeight
            eineLinie.HorizontalAlignment = HorizontalAlignment.Left
            eineLinie.VerticalAlignment = VerticalAlignment.Center
            eineLinie.StrokeThickness = 1
            'Und der Leinwand hinzufügen
            meineLeinwand.Children.Add(eineLinie)

            'Linienzähler aktualisieren
            linienZähler += 2
        Next

        Me.Title = "Anzahl Linien im Fenster: " & linienZähler
    End Sub
    .
    .
    .
End Class
```

Tatsache ist: Anders als bei GDI oder GDI+ ist das Gezeichnete hier persistent. Und das wird umso deutlicher, wenn Sie abermals einen Doppelklick in die Grafik machen, denn dann sehen Sie, dass jede einzelne Linie beginnt, sich zu animieren, und damit einen Effekt generiert, der sich wirklich sehen lassen kann (Abbildung 6.6).

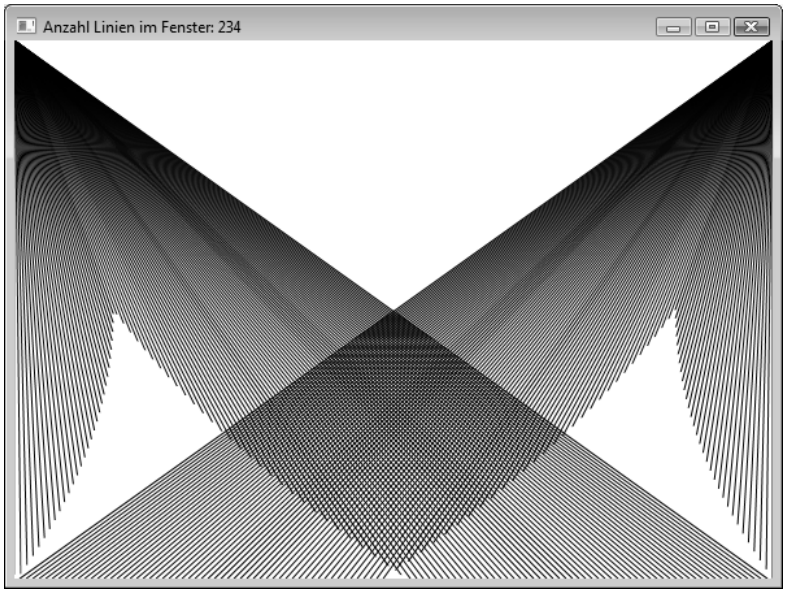

Abbildung 6.6 ...hinterlässt aber nach einem Doppelklick einen völlig anderen Eindruck, denn damit startet nämlich ein zwar einfacher Animationsvorgang, der aber jede einzelne der im Fenster enthaltenen 234 Linien betrifft!

Sie können nur ahnen, welchen Aufwand man in GDI+ bzw. GDI betreiben müsste, um diesen Animationseffekt so flimmerfrei wie hier im Beispiel hinzubekommen. Man müsste mit Doppelpufferung und Zwischenspeicherung des Anzeigebildes arbeiten, und man müsste die aktuellen Längen jeder einzelnen Linie zwischenspeichern und selbst kontrollieren. Nicht so in WPF:

```
Private Sub Window1_MouseDoubleClick(ByVal sender As Object, _
                    ByVal e As System.Windows.Input.MouseButtonEventArgs) Handles Me.MouseDoubleClick
        Dim count = 0

        'Funktioniert natürlich nur, da sich nur
        'Linien auf dem Canvas-Objekt befinden.
        For Each lineItem As Line In meineLeinwand.Children
            'Y2-Eigenschaft wird animiert, daher der Name.
            'Die Klasse DoubleAnimation animiert Eigenschaften
            'vom Typ Double.

            'Animiert wird von der aktuellen Höhe der Linie
            'bis zur halben Höhe der Linie.
            Dim Y2Animation As New DoubleAnimation(lineItem.ActualHeight, _
                        lineItem.ActualHeight / 2, TimeSpan.FromSeconds(3))

            'Hier wird festgelegt, dass die Animation automatisch wird,
            'wenn sie einmal komplett durchlaufen wurde.
            Y2Animation.RepeatBehavior = RepeatBehavior.Forever
```

```
            'Bestimmt, dass die DoubleAnimation hoch- und durch
            'AutoReverse=True anschließed wieder runterzählt.
            Y2Animation.AutoReverse = True

            'Die Animation für jede weitere Linie wird um
            '500 Millisekunden verzögert zur vorherigen begonnen,
            Y2Animation.BeginTime = New TimeSpan(500000 * count)

            'Animation starten ...
            lineItem.BeginAnimation(Line.HeightProperty, Y2Animation)
            count += 1
        Next
    End Sub
```

Jede einzelne Linie ist hier in WPF nicht nur eine simple Methode, die bewirkt, dass Veränderungen am Grafikspeicher vorgenommen werden, dessen Repräsentation dann wieder den Eindruck einer gemalten Linie bewirkt. Vielmehr ist eine Linie, wie wir sie hier verwenden, abgeleitet von einer Klasse, die schon unfassbar viel an Infrastruktur mitbringt. Die nicht nur dafür sorgt, dass eine Instanz dieser Klasse sich zu gegebener Zeit rendert und aktualisiert, sondern die sich auch in eine Animationsinfrastruktur einfügt und sich damit wie viele der meisten anderen Objekte, die WPF als »sichtbares Ding« kennt, also als so genanntes Visual, verhält und zu steuern vermag.

Abbildung 6.7 Eine TextBox, die auf eine animierte Stoffsimulation projiziert wird und dabei voll funktionsfähig bleibt – eine der eindrucksvollsten WPF-Demos derzeit, da mit klugem Kopf, aber minimalem Aufwand in die Tat umgesetzt

Und Sie haben hier wirklich nur die Spitze des Eisbergs gesehen. Zu was WPF seit dem Service Pack 1 von .NET 3.5 in der Lage ist, zeigt eine Demo von David Teitlebaum, die Sie unter *http://tinyurl.com/5kmpz7* einsehen können, auf Channel 9 übrigens sehr eindrucksvoll. Die Demodateien können Sie übrigens über David Teitlebaums Blog abrufen – sie sind allerdings leider nur in C# verfügbar.

Wie WPF Designer und Entwickler zusammenbringt

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-Code festgelegt. Nun hat ein Grafikdesigner mit Programmcode sehr wenig im Sinn. Der Grafiker benutzt diverse Werkzeuge, um schöne, bunte Grafiken zu erstellen. Er wird aber wohl kaum Visual Basic-Code schreiben wollen, um das Aussehen eines Fensters oder Steuerelements zu verändern. Andererseits ist es unmöglich, den logischen Teil der Applikation, also zum Beispiel die Datenzugriffe oder das Berechnen von Zahlen, mit einem Grafikwerkzeug durchzuführen.

Leider gibt es nur wenige Programmierer, die gleichzeitig auch hervorragende Designer sind. Das kann man natürlich auch anders herum betrachten: Es gibt wenige Topdesigner, die auch noch perfekt programmieren können.

Hier stoßen also zwei Welten aufeinander, 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 sollte ohne größeres Wissen über die eingesetzten Designerwerkzeuge auskommen. Ziel ist letztendlich, dass ein Topdesigner zusammen mit einem Topsoftwareentwickler eine Topapplikation erstellt.

1. Beginnen wir im alten Stil und schreiben eine WPF-Applikation nur mit Visual Basic-Code. Wir führen zunächst einmal keine Trennung von Design und Code ein. Es handelt sich natürlich um eine weitere Version des bekannten *Hallo Welt!*-Programms. Die Vorgehensweise ist folgende:
2. Starten Sie Visual Studio 2010 und wählen Sie *Datei/Neu/Projekt* aus.
3. Im Dialog *Neues Projekt* wählen Sie als Programmiersprache *Visual Basic* aus. Im großen Listenfeld entscheiden Sie sich für *Leeres Projekt*.
4. Als Referenzen fügen Sie Verweise auf folgende Bibliotheken ein: `System`, `WindowBase`, `PresentationFramework`, `System.Xaml` und `PresentationCore`. Benutzen Sie hierzu im Projektmappen-Explorer durch Anklicken mit der rechten Maustaste den Befehl *Verweis hinzufügen*. Wählen Sie die angegebenen Referenzen im Dialog aus.
5. Fügen Sie nun im Projektmappen-Explorer eine neue Visual Basic-Codedatei vom Typ *Klasse* mit dem Namen »Hallo.vb« hinzu. Benutzen Sie wieder die rechte Maustaste und den Befehl *Hinzufügen/Neues Element*.
6. Tippen Sie nun den Code des nächsten Listings ein.
7. In den Projekt-Eigenschaften stellen Sie auf der Registerkarte *Anwendung* den folgenden *Anwendungstyp* ein: *Windows Forms-Anwendung*.

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

        <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
```

Wenn wir das Programm laufen lassen, erscheint ein ziemlich eintöniges Fenster mit dem Titel »Hallo, Welt!« (Abbildung 6.8).

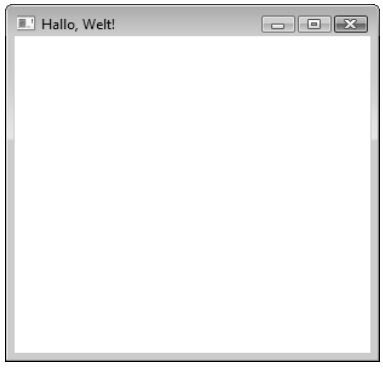

Abbildung 6.8 Das erste WPF-Fenster

Schauen wir uns den Code des Beispiels nun etwas genauer an. Nach dem Einfügen der benötigten Namensräume aus den Referenzen definieren wir einen eigenen Namespace 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 Apartment« zu setzen. Dies ist ein Relikt aus alten COM-Zeiten, aber es sorgt gegebenenfalls 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 Codezeilen 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 so lange sichtbar, bis diese Meldungsschleife beendet wird, zum Beispiel durch Anklicken der Schließenschaltfläche rechts oben in der Titelzeile des Fensters.

Nun haben wir in diesem Beispiel allerdings den Designer arbeitslos gemacht, denn wir haben als Softwareentwickler das Design im Visual Basic-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 Visual Basic-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 (sprich: »gsämmel«) benutzt.

XAML: Extensible Application Markup Language

XAML ist, wie XML, eine hierarchisch orientierte Beschreibungssprache. Wir können mit XAML Benutzerschnittstellen oder Grafiken deklarieren. Kommen wir wieder zu unserem ersten Beispiel (Abbildung 6.8) zurück. Als Nächstes deklarieren wir nun den Designerteil 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 folgenden Listing 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>
```

Was wir hier 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 × 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 2010 fügt diese Namensräume automatisch in Ihre .NET Framework 4.0-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 Visual Basic-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 2010 folgende:

1. Starten Sie Visual Studio 2010 und wählen Sie *Datei/Neu/Projekt* aus.
2. Im Dialog *Neues Projekt* wählen Sie als Programmiersprache *Visual Basic* aus. Im großen Listenfeld entscheiden Sie sich für *WPF-Anwendung*.
3. Öffnen Sie nun im Projektmappen-Explorer die XAML-Datei mit dem Namen *MainWindow.xaml*.
4. Tippen Sie nun den Code des letzten Listings ein. Ändern Sie dabei den Code, welchen Visual Studio erzeugt hat, einfach entsprechend.
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 (siehe folgendes Listing). 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>
```

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

Dieses Projekt wurde ebenfalls mit Visual Studio 2010 erzeugt. Zunächst einmal werden nur die Namensräume `System` und `System.Windows` benötigt. Visual Studio erzeugt fast den gesamten Code aus dem folgenden Listing, 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
```

Im Konstruktor der Fensterklasse wird die Methode `InitializeComponent` aufgerufen. Auch der Code dieser Methode wird in einer separaten Visual Basic-Codedatei von Visual Studio generiert. Der Visual Basic-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 Visual Basic-Dateien verteilt werden: Ein Teil der Klasse ist der Code aus dem Listing und der andere, unsichtbare Teil ist der von Visual Studio erzeugte Zusatzcode, der die Methode `InitializeComponent` und eine Art Abbildung des XAML-Codes 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 Zwischencode, 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 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 `RoutedEventArgs` 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 6.9.

Abbildung 6.9 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 Visual Basic-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. 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>
```

Da die Schaltfläche nun einen Namen (`btn`) hat, können wir sie ganz normal aus dem Visual Basic-Code heraus ansprechen und die Eigenschaften neu setzen oder Methoden des Objekts `btn` aufrufen. Im Beispiel (nächstes Listing) ändern wir mit der Eigenschaft `Content` zunächst den Text, der auf der Schaltfläche ausgegeben wird. Schließlich werden die Fontgröße (`FontSize`) und die Vordergrundfarbe (`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
```

Abbildung 6.10 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 6.10 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 den folgenden beiden Codelistings gezeigt.

Für das Hauptfenster mit dem Klassennamen `Window1` wird das Ereignis `Loaded` an die Methode `OnLoaded` gebunden. Der Code, der in der entsprechenden Visual Basic-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. Zunächst der XAML-Code:

```
<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>
```

Und der Visual Basic-Code:

```
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

        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
```

Abbildung 6.11 Nach der Ausführung der OnLoaded-Methode

WICHTIG Grundsätzlich können wir sagen, dass alles, was in XAML angegeben und deklariert werden kann, auch in Visual Basic 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 so genannte Transformation. Benutzen Sie den entsprechenden XAML-Code mit der Transformation im Moment einfach so, wie in folgendem Listing 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="OnLoaded">
        <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>
```

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 dem letzten Listing. Wenn Sie das Programm starten, sehen Sie zunächst die schräge Schaltfläche (Abbildung 6.12 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 6.12 rechts).

Abbildung 6.12 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«.

Ereignisbehandlungsroutinen in WPF und Visual Basic

Im Zusammenhang mit dem Designer möchte ich auf das »Verdrahten« von Ereignisbehandlungsroutinen in WPF-Anwendungen ein kleines bisschen näher eingehen. In Windows Forms-Anwendungen sind Visual Basic-Entwickler es gewohnt, Ereignisprozeduren per Doppelklick auf das entsprechende Steuerelement zu verdrahten – das vorherige Kapitel hat gezeigt, wie es geht. Das funktioniert in WPF-Anwendungen mit dem Designer prinzipiell ebenfalls. In beiden Fällen fügt Visual Basic dabei den Rumpf der Ereignisroutine mit einem geeigneten Namen ein, die automatisch die korrekte Signatur für das Behandeln des Ereignisses erhält. Das `Handles`-Schlüsselwort zeigt dem Visual Basic-Compiler an, dass hier die Infrastruktur für das Hinzufügen der Ereignisdelegatenliste in die `Form`- (bzw. – für WPF – `Window`-)Klasse eingefügt werden muss.

Bei WPF-Anwendungen gibt es in Ergänzung dazu auch eine Möglichkeit, die Sie auch bei der Entwicklung in jedem Fall vorziehen sollten, nämlich im XAML-Code selbst zu bestimmen, welche Ereignisprozedur für ein Ereignis eines bestimmten Objekts aufgerufen werden soll. Das funktioniert ähnlich einfach wie ein Doppelklick auf ein Steuerelement, wie Abbildung 6.13 zeigt.

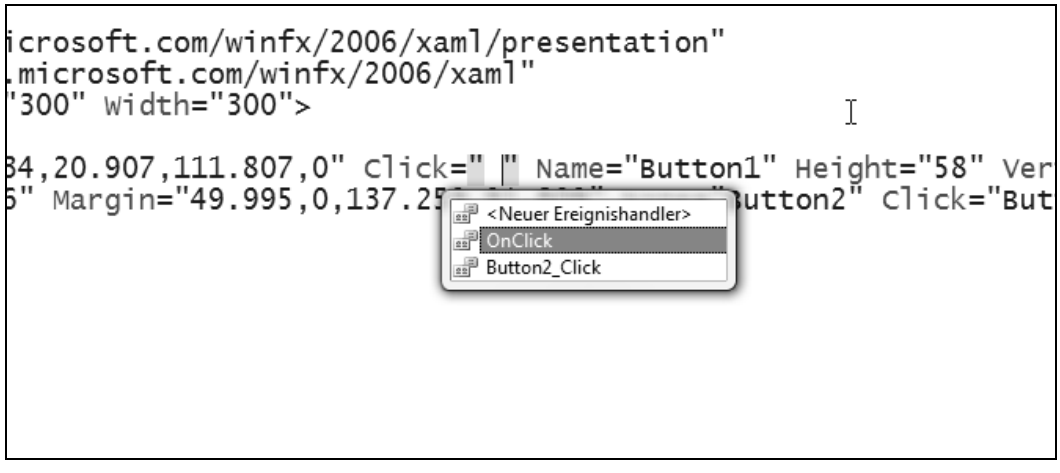

Abbildung 6.13 Ereignisprozeduren lassen sich in WPF auch in Visual Basic-Programmen, anders als Sie es von Windows Forms-Anwendungen gewohnt sind, per XAML-Code »verdrahten«

Sie können den Namen des Ereignisses direkt in die XAML-Elementdefinition hineinschreiben, und IntelliSense hilft Ihnen anschließend, das Ereignis entweder mit einer bereits vorhandenen Methode (die natürlich über die entsprechende Signatur des Ereignisses verfügen muss) zu verknüpfen oder den Rumpf einer neuen Ereignisbehandlungsroutine einzufügen. In letzterem Fall würden Sie in der IntelliSense-Objekteliste einfach den ersten Eintrag <*Neuer Ereignishandler*> auswählen.

Die XAML-Syntax im Überblick

Dieser Abschnitt gibt Ihnen einen kurzen Überblick über die XAML-Syntax. Der XAML-Code soll auch mit normalem Visual Basic-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 Wurzelelement, 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 in .NET Framework 4.0. 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 Visual Basic-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 Visual Basic-Zeile die Schaltfläche als Inhalt in das übergeordnete Element 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 Visual Basic-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 übergeordneten Elements 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 (mehr zum Thema *Grids* erfahren Sie später).

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 Visual Basic-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 Visual Basic-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 Kommas 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 Visual Basic-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)
 mesh.Normals.Add(New Vector3D(0.0, 1.0, 0.0))
 mesh.Normals.Add(New Vector3D(0.0, 1.0, 0.0))
 mesh.Normals.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 generiert 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).

ImageResizer – das praktische WPF-Beispiel

Wie schon im vorherigen Kapitel soll auch in diesem das Motto gelten: Am besten funktioniert das Erlernen eines neuen Themenblocks und neuer Features von Visual Studio, wenn Sie sich ein Beispiel schnappen und dieses direkt ausprobieren. Und für WPF und die entsprechenden Tools haben wir – so meinen wir jedenfalls – uns etwas ganz Nettes einfallen lassen, das nicht nur ein typisches Beispiel darstellt, sondern dessen fertiges Ergebnis Sie später auch noch täglich nutzen können. Naja, viele von Ihnen jedenfalls:

Es beginnt mit Neugier – eine der stärksten Triebfedern des Menschen, welche seit dem Aufkommen der sozialen Netzwerke ein Freudenfeuerwerk nach dem anderen knallen lässt. Wenn ich meine Urlaubsbilder noch am gleichen Tag den neidischen Kollegen zuhause präsentieren will, ist das technisch kein Problem. Es gibt tausend Möglichkeiten, die Bilder im Internet zu verteilen, doch bergen sie alle ein Problem. Dank der Auflösung von vielen Millionen Pixel moderner Kameras sind die Bilder zwar gestochen scharf, doch das spiegelt sich auch in der Größe der Dateien wider. Und diese Tatsache ist die Grundlage der Idee für das Projekt dieses Kapitels.

Was kann dieses Kapitel Ihnen bieten? Es kann Ihnen einen schnellen, mühelosen und vor allem praktischen Einstieg in die WPF-Anwendungsentwicklung bieten. Wenn Sie bisher mit Windows Forms programmiert und diese geschätzt haben, so wird Ihnen dieses Kapitel helfen, alte Gewohnheiten – wenn auch nicht komplett – hinter sich zu lassen oder sie zumindest um WPF-Eigenarten und die des WPF-Designers zu ergänzen und damit die Tür zu einer neuen Welt zu öffnen.

Sie werden erfahren, wie Sie es schaffen, die Zeit für das manuelle Verkleinern der Bilder zu sparen und Ihre Kollegen dennoch neidisch zu machen. Das Ergebnis, der ImageResizer (Abbildung 6.14), ist ein WPF basiertes Projekt, welches uns mit möglichst effektiver Auslastung der Leistungsressourcen des Computers und minimalem Aufwand des Benutzers erlaubt, Bilder im Dateityp und in der Größe anzupassen.

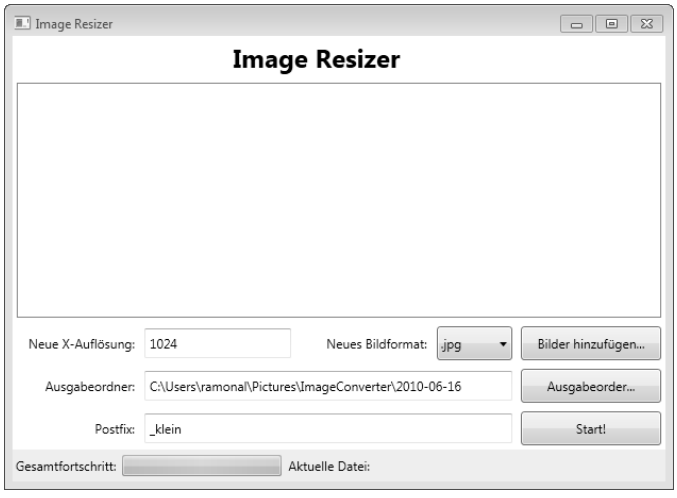

Abbildung 6.14 Oberfläche des Programms

1. Starten Sie Visual Studio, falls Sie das noch nicht getan haben. Sie finden sich auf der Startseite wieder. Hier befindet sich alles, was Sie für einen schnellen Start brauchen: Zugriff auf Ihre letzten Projekte, die Möglichkeit, Projekte zu öffnen und neue Projekte anzulegen. Klicken Sie auf *Neues Projekt* und wählen Sie im daraufhin folgenden Dialog den Typ *WPF-Anwendung* aus. Geben Sie dem Projekt den Namen **ImageResizer**, und bestätigen Sie den Dialog mit *OK*.

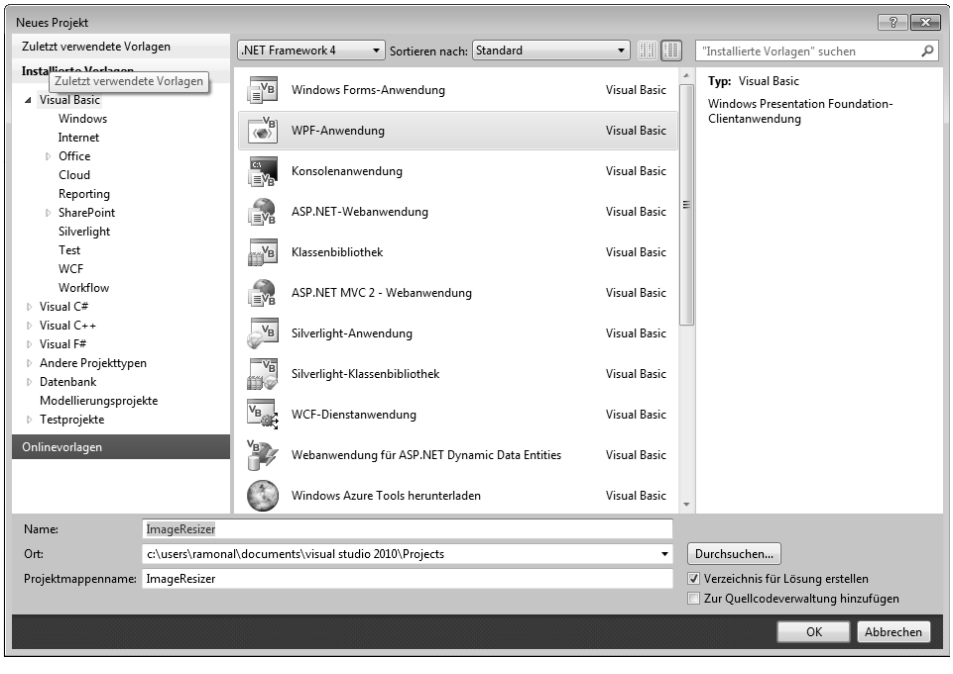

Abbildung 6.15 Auf der Startseite ein neues Projekt anlegen

Standardmäßig besitzt eine neue WPF-Anwendung die Projektdateien *MainWindow* und *Application*. Dabei stellt *MainWindow* eine Klassendatei dar, die den Code des zurzeit noch einzigen Fensters der WPF-Anwendung enthält. Hier wird sich der größte Teil der Programmlogik abspielen, und wir werden unser Projekt durch weitere Projektdateien dieses Typs ergänzen. Dieser Dateityp kann sowohl XAML-Code als auch Programmcode (in einer genannten Code-Behind-Datei) beinhalten.

In *Application* können globale Einstellungen für das Projekt definiert werden, wie beispielsweise das Aussehen der Schaltflächen, aber auch Methoden, die unter anderem das Starten und Beenden der Anwendung betreffen. Hier steht also alles, was das Projekt im Ganzen betrifft. Auch für diesen Projektdateityp gilt, dass er XAML-Code sowie eine Code-Behind-Datei enthalten kann.

Visual Studio hat das Fenster *MainWindow* in der Entwurfsansicht geöffnet, sodass die Oberfläche bearbeitet werden kann. Die Entwurfsansicht teilt sich in den Designer und in den zugrunde liegenden Code, vergleichbar mit dem Entwurf einer Webseite: Unten steht gewissermaßen der HTML-Code und oben der Entwurf, wie er später im Webbrowser aussehen wird. Im Falle von WPF ist die Auszeichnungssprache, wie schon erwähnt, XAML, und der Designer übernimmt das Anzeigen des Fensters zur Laufzeit. Das Aussehen des Fensters wird durch Tags und deren Verschachtelung sowie Anordnung bestimmt.

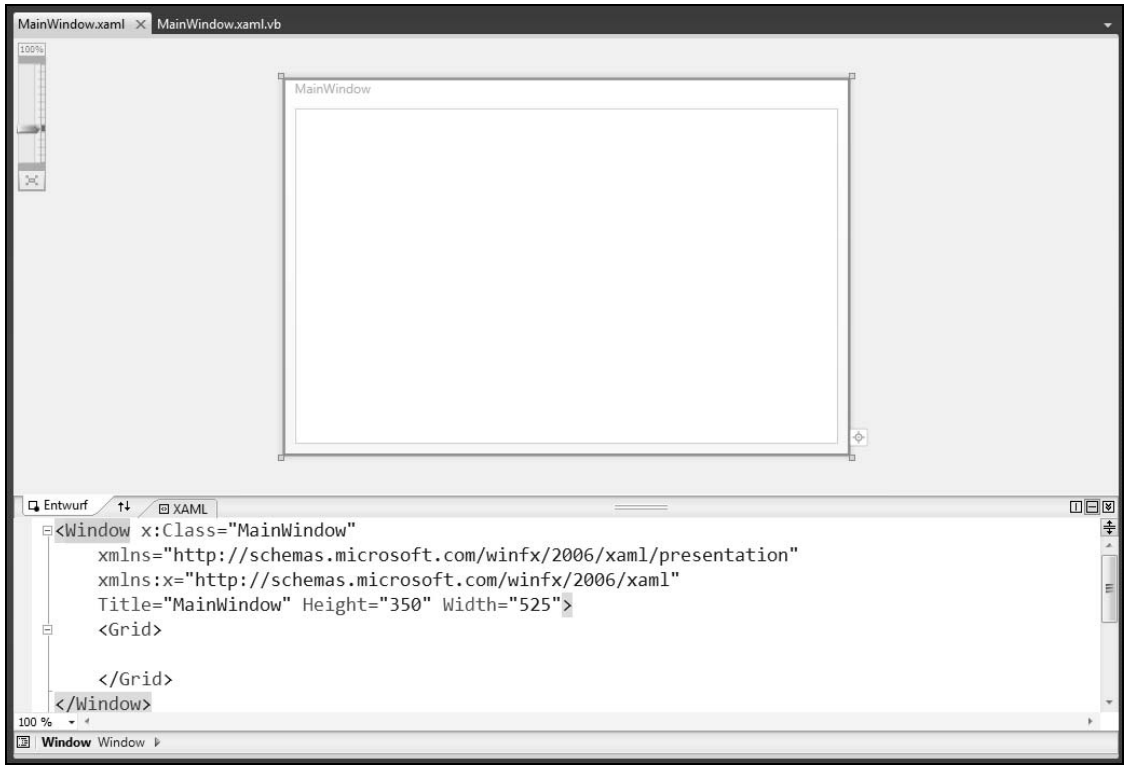

Abbildung 6.16 WPF-Fenster im Designermodus

Der Code wie in Abbildung 6.16 zu sehen, lässt sich folgendermaßen übersetzen:

```
<Window x:Class="MainWindow">…</Window>
```

Durch die Tags `<Window>…</Window>` wird ein WPF-Fenster definiert, welches den Namen *MainWindow* trägt. Alles was zwischen diesen Tags definiert ist, befindet sich im Fenster.

```
<Window x:Class="MainWindow"
        Title="MainWindow" Height="350" Width="525">

</Window>
```

Hier werden drei Eigenschaften gesetzt: der Titel des Fensters (es heißt zunächst *MainWindow*), die Höhe (350 Pixel) und die Breite (525 Pixel).

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

Umgangssprachlich enthalten diese beiden Codezeilen die Grundlage für die Sprache, durch die das Fenster sein Aussehen erhält. Es werden zwei Namensräume registriert, in denen alle WPF-Objekte definiert sind. Dieses Prinzip sollten Sie sich dann merken, da Sie auf diese Art und Weise später Zugriff beispielsweise auf Ihre eigenen Steuerelemente im XAML-Code nehmen können. Sie registrieren die Assembly, in der Ihr Programm läuft, und geben ihr ein Präfix, unter dem sie zu erreichen ist.

```
xmlns:loc="clr-namespace:ImageResizer"
```

Nun sind alle Elemente der Assembly im Fenster über `<loc:NameDesElements></loc:NameDesElements>` zu erreichen und können so eingebunden werden.

```
<Window x:Class="MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Im Fenster geschachtelt ist ein *WPF-*`Grid`. Das `Grid` ist der einzige Inhalt eines Fensters oberster Ebene. Das ist ein großer Unterschied zwischen Windows Forms und WPF: Während Sie in Windows Forms in ein Form (was ja das Äquivalent zu einem Fenster in WPF bildet) beliebig viele Steuerelemente anordnen können, funktioniert das in WPF mit einem Fenster nicht. Jedes Fenster kann nur ein Steuerelement beherbergen und in 90% der Fälle besteht der Inhalt eines Fensters in WPF aus einem `Grid`. Das hat folgenden Grund: Das WPF-`Grid` ist das, wozu die HTML-Tabelle (`<table></table>`) jahrelang missbraucht wurde: Es teilt seine Fläche einerseits in Zeilen und Spalten auf und kann andererseits, im Unterschied zum Fenster selbst, untergeordnete Elemente anordnen.

HINWEIS Im Grunde genommen gibt es in WPF nur zwei Sorten von Steuerelementen. Die erste Art sind solche, die einen Inhalt (also ein weiteres Steuerelement) beinhalten können. *Inhalt* heißt auf Englisch *Content*, und deswegen haben solche Steuerelemente eine `Content`-Eigenschaft. Die zweite Sorte bilden Steuerelemente, die dazu dienen, andere Steuerelemente zu schachteln und in einem bestimmten Stil anzuordnen; sie fungieren also als *Container* für andere Steuerelemente, stellen selbst aber nichts visuell dar. Diese Trennung gibt es so in Windows Forms nicht. Hier kann jedes Steuerelement auch als Container fungieren (so und gerade auch das Form selbst), und daran muss man sich als Entwickler unter WPF erst einmal gewöhnen.

Übrigens, Sie können das einmal selbst ausprobieren: Löschen Sie zunächst das `Grid` im Fenster, und ziehen Sie eine Schaltfläche aus der Toolbox in das Fenster. Das funktioniert. Versuchen Sie es mit einer weiteren Schaltfläche, werden Sie scheitern – der Designer verweigert die Anordnung der zweiten Schaltfläche. Das Fenster selbst gehört nämlich auch zu den `Content`-Steuerelementen, das heißt, es kann nur ein einziges Steuerelement beherbergen. Damit *muss* Ihr einziges Steuerelement ein Container-Steuerelement sein, und das ist standardmäßig eben ein `Grid`-Steuerelement, weil es am flexibelsten layoutet werden kann.

Das bisher beschriebene Wissen über WPF soll als theoretische Grundlage reichen. Doch bevor Sie irgendetwas praktisch umsetzen, machen Sie sich eine Vorstellung davon, wie Ihre Oberfläche aussehen soll. Bitte unterschätzen Sie diesen Schritt nicht, er wird Ihnen beim Designen der Oberfläche mit WPF im Nachhinein viel Arbeit ersparen können.

Überlegen Sie sich:

- **Welche Informationen braucht das Programm und welche Steuerelemente resultieren daraus?**
  - Bilder-/Ordnerpfade müssen eingelesen werden:
    → TextBox, Button
  - Der Ausgabeordner muss eingelesen werden:
    → TextBox, Button
  - Eingabe eines Postfixes:
    → TextBox
  - Der Dateityp muss ausgewählt werden:
    → ComboBox
  - Die Auflösung muss eingelesen werden:
    → TextBox
  - Vorgang starten:
    → Button
- **Welche Steuerelemente brauche ich, um dem Benutzer Informationen anzuzeigen?**
  - Beschriftung der TextBoxen:
    → Label
  - Liste aller ausgewählten Bilder
    → ListBox

- **Welche zusätzlichen Steuerelemente wie Beschriftungen, Überschriften, Fortschrittsanzeigen und Ähnliches brauche ich?**
  - Überschrift im Fenster

    → TextBlock
  - Statusanzeige mit einem Fortschrittsbalken für den Bearbeitungsfortschritt und das Anzeigen des aktuellen Bildes

    → StatusBar, ProgressBar, TextBlock
- **Wie sollen sich die Steuerelemente verhalten, wenn der Benutzer das Fenster größer zieht?**
  - Die Überschrift und die StatusBar bleiben statisch. Auch die Eingabefelder sollen nicht wachsen, die Liste für die Dateinamen dennoch schon. Die ListBox soll sich dynamisch vergrößern.
- **Wie ordne ich die Steuerelemente am unmissverständlichsten an?**

Nachdem Sie sich die ersten vier Punkte klargemacht haben, sollten Sie sich im letzten Schritt Ihre Ideen bildlich vorstellen, was natürlich am besten mit Stift und Papier geht. Malen Sie ihr Fenster auf, wie es nachher aussehen soll. Und dann überlegen Sie sich, wie Sie den Entwurf am besten in WPF umsetzen können, zum Beispiel und am besten mithilfe eines `Grid`, welches Ihnen die Definition von Zeilen und Spalten erlaubt.

Einen Entwurfsvorschlag für die Aufteilung der Oberfläche des Projekts ImageResizer, an dem Sie sich im Folgenden orientieren sollten, zeigt Abbildung 6.17.

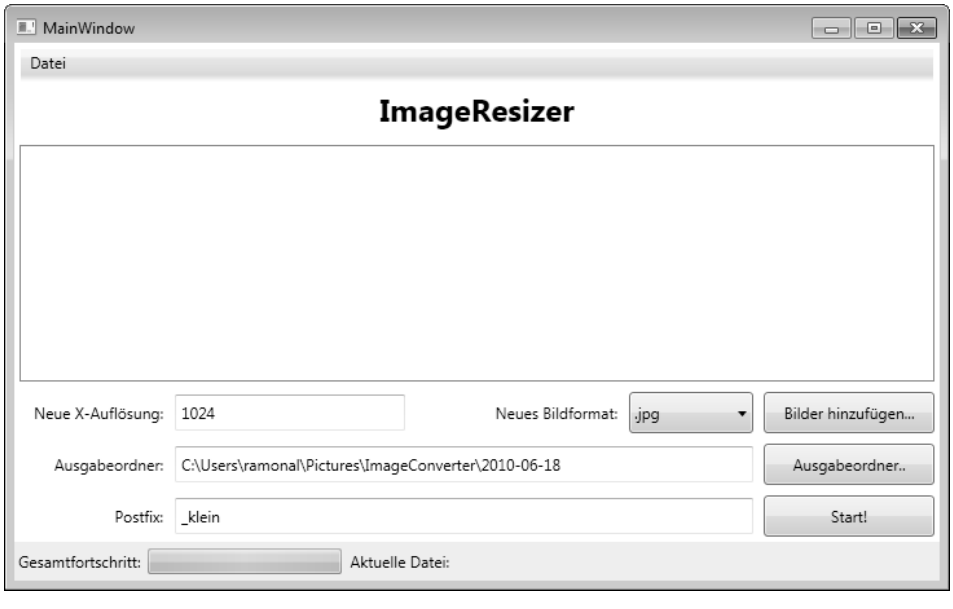

Abbildung 6.17 Die Oberfläche im Entwurf

Eine mögliche Aufteilung der Oberfläche in Zeilen und Spalten sehen Sie in Abbildung 6.18. Die Oberfläche ist in 6 Zeilen und 5 Spalten aufgeteilt.

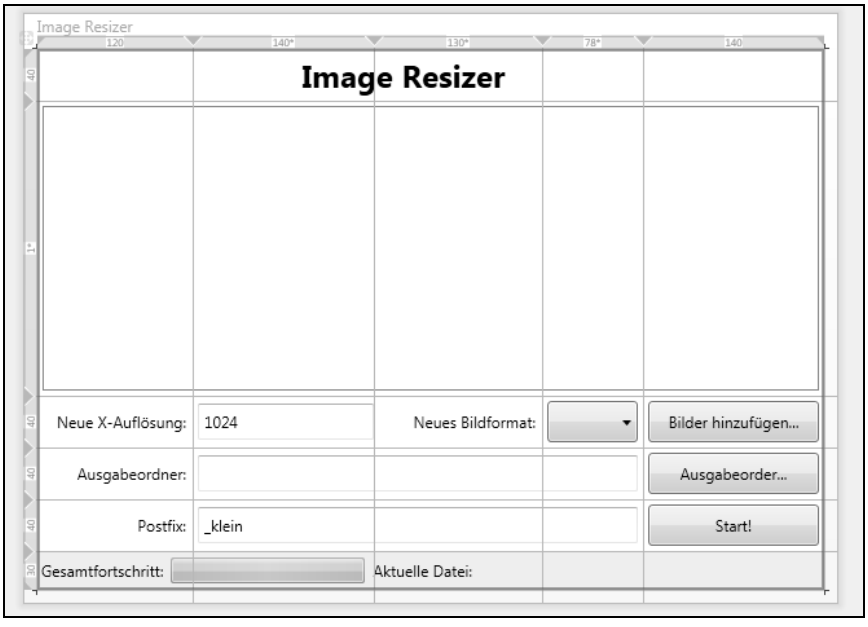

Abbildung 6.18 Einteilung des Entwurfs der Oberfläche in Zeilen und Spalten

Um eine Oberfläche in WPF umzusetzen, gibt es prinzipiell zwei Möglichkeiten: den Designer und den Code. Ich persönlich würde Ihnen empfehlen, so viel wie möglich Code händisch zu definieren, weil daraus ein sauberer, pflegbarer und verständlicher Code resultiert, der sich auch später einfach debuggen lässt. Das mag am Anfang eher mühsam sein, doch es erspart einem die unter Umständen aufwändige Fehlersuche im vom Designer erzeugten Code. Sollten Sie eine komplizierte und verschachtelte Oberfläche im Designer konzipiert haben, kann das zur Laufzeit ein ungewolltes Ergebnis produzieren, dessen Ursache Sie dann im Code wieder finden müssen. Ein Designer ist universell und kann Ihre Ideen bis zu einem gewissen Grad umsetzen, jedoch vielleicht nicht auf die beste Art und Weise.

Definieren von Grid-Spalten und -Zeilen zum Anordnen von Steuerelementen

Trotzdem kann er lästige Tipparbeit übernehmen. Zum Beispiel beim Definieren der Zeilen und Spalten des `Grid`.

2. In der Entwurfsansicht des *MainWindow* in Visual Studio klicken Sie auf das Fenster und vergrößern es mithilfe des kleinen blauen Rechtecks rechts unten, bis es Ihnen passend erscheint.

Abbildung 6.19 Vergrößern Sie das Fenster, bis es Ihnen groß genug erscheint.

3. Anschließend klicken Sie auf das im Fenster befindliche `Grid` und bewegen den Mauszeiger an dessen blauen Rahmen entlang. Sie sehen eine Linie zum Definieren einer Zeile bzw. Spalte. Um eine Zeile oder Spalte Wirklichkeit werden zu lassen, suchen Sie sich die richtige Dimension der Zeile oder Spalte aus und klicken Sie an die Stelle.

Abbildung 6.20 Das Grid im Designer in Spalten und Zeilen aufteilen

4. Setzen Sie dieses Verfahren fort, bis Sie eine ähnliche Aufteilung wie in Abbildung 6.21 erreicht haben.

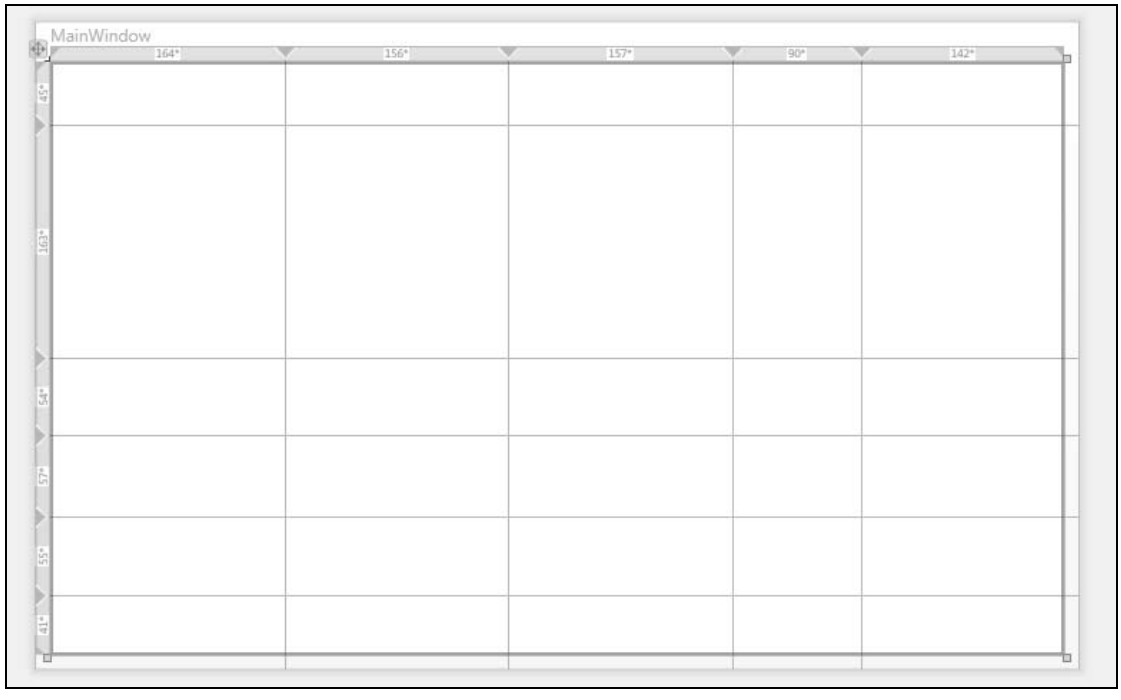

Abbildung 6.21 Grid nach Aufteilung in Zeilen und Spalten

Sie sehen: Die erste Spalte ist zu groß, die letzte etwas zu klein. Das Feintuning der Aufteilung kann nun sehr einfach und schnell im XAML-Code vorgenommen werden. Sie sehen, dass der WPF-Designer einiges an XAML-Code produziert hat:

```
<Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="45*" />
            <RowDefinition Height="163*" />
            <RowDefinition Height="54*" />
            <RowDefinition Height="57*" />
            <RowDefinition Height="55*" />
            <RowDefinition Height="41*" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="164*" />
            <ColumnDefinition Width="156*" />
            <ColumnDefinition Width="157*" />
            <ColumnDefinition Width="90*" />
            <ColumnDefinition Width="142*" />
        </Grid.ColumnDefinitions>
    </Grid>
```

Das Zeichen »*« neben den Werteangaben für die Zeilenhöhe bzw. Spaltenbreite bedeutet hier so viel wie »anteilig«. Die erste Zeile erhält anteilig »45« vom Ganzen, wohingegen die zweite mit »163 Teilen anteilig am Ganzen« eindeutig besser dran ist. »Anteilig« (also mit »*«) werden Spalten und Zeilen im Grid immer, wenn Sie sie mit dem WPF-Designer zusammenklicken. Möchten Sie feste Pixelbreiten, löschen Sie das Sternchen im XAML-Code einfach raus.

HINWEIS Prinzipiell geht das auch mit dem Designer (Abbildung 6.22), dauert aber unnötig lange, deswegen wollen wir hier im Beispiel nicht näher darauf eingehen.

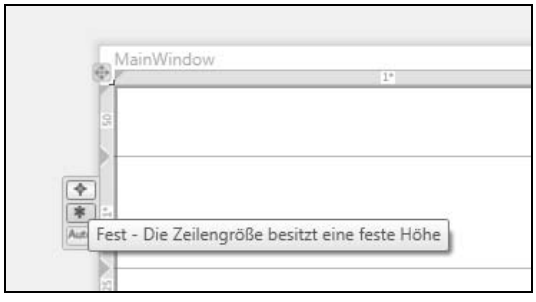

Abbildung 6.22 Nur der Vollständigkeit halber erwähnt – feste Spalten- oder Zeilenmaße erreichen Sie auch über den Designer; nur sehr viel umständlicher und langwieriger

Im Entwurf liegt die Überschrift in der ersten `Grid`-Zeile und die Statusleiste in der letzten. Diese beiden Zeilen sollen statisch bleiben.

5. Platzieren Sie die Einfügemarke im Quellcode und ändern Sie die Angaben der Höhe in der ersten Zeile auf *40* und in der letzten Zeile auf *30*.
6. Die Eingabefelder liegen in den Zeilen drei, vier und fünf. Setzen Sie auch die Höhe dieser Zeilen auf *40*.
7. Es bleibt nur noch die Zeile mit der `ListBox`, die sich dynamisch vergrößern soll. Ändern Sie die Höhenangabe hier auf »*«. Nun bleibt alles bei seiner festen Größe und nur die `ListBox` mit den Dateinamen wächst später beim Vergrößern des Fensters dynamisch mit.

```
<Grid.RowDefinitions>
    <RowDefinition Height="40" />
    <RowDefinition Height="*" />
    <RowDefinition Height="40" />
    <RowDefinition Height="40" />
    <RowDefinition Height="40" />
    <RowDefinition Height="30" />
</Grid.RowDefinitions>
```

8. Wiederholen Sie die gleiche Vorgehensweise mit den Spalten. Überlegen Sie sich, welche Spalte gleich bleibend groß sein soll, und welche Spalten sich dynamisch vergrößern sollen. In der ersten Spalte befinden sich in der Entwurfsansicht die Textblöcke zur Beschriftung der Eingabefelder und in der letzten Spalte die Schaltflächen zum Bedienen des Programms. Diese beiden Spalten sollten sich nicht vergrößern, da weder die Schaltflächen noch die Textblöcke besonders hübsch oder sinnvoll sind, wenn diese breit gezogen dargestellt werden. Die restlichen Spalten sollen dynamisch mitwachsen. Ändern Sie die Werte der Breite in der ersten und letzten Spalte auf *120* und *140*, und gleichen Sie auch die dynamischen Werte wie unten stehend an:

```
<Grid.ColumnDefinitions>
    <ColumnDefinition Width="120" />
    <ColumnDefinition Width="140*" />
    <ColumnDefinition Width="130*" />
    <ColumnDefinition Width="80*" />
    <ColumnDefinition Width="140" />
</Grid.ColumnDefinitions>
```

Der Grundstein für die Oberflächengestaltung ist gelegt und die Oberfläche bereit für die Anordnung der Steuerelemente in der entstandenen Aufteilung.

WICHTIG Die erste Zeile eines `Grid` ist immer die 0, daraufhin folgt die 1. usw. Wenn Sie Steuerelementen eine Zeile des `Grid` zuweisen, denken Sie also immer daran, dass Sie von *null* an zählen müssen. Das Gleiche gilt natürlich auch für Spalten im `Grid`.

»Begin at the beginning«, the King said very gravely, »and go on till you come to the end: then stop!«[7]

Dieses Zitat von Lewis Carroll wird richtungsweisend bei der weiteren Vorgehensweise sein und nun befolgt, indem als Erstes die Überschrift ihren Platz auf der Oberfläche findet.

9. Bewegen Sie den Mauszeiger an den linken Rand der Visual Studio-IDE und blenden Sie durch Klicken des Toolbox-Symbols die Toolbox ein.

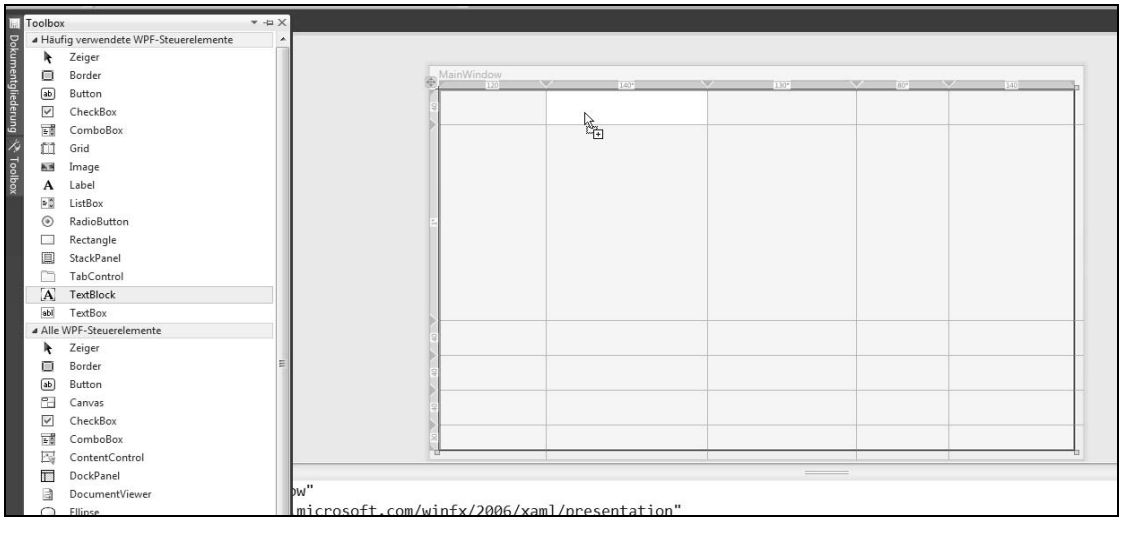

Abbildung 6.23 Ziehen eines *TextBlock* aus der Toolbox auf die Oberfläche

[7] Zu Deutsch: »Fange beim Anfang an«, sagte der König ernsthaft, »und lies, bis du ans Ende kommst, dann halte inne.«

10. Positionieren Sie per Ziehen und Ablegen, wie in Abbildung 6.23 zu sehen, einen `TextBlock` in der obersten Zeile des `Grid`. Der `TextBlock` sollte sich nun ungefähr an der Stelle befinden, wie sie Abbildung 6.24 zeigt. Klicken Sie den `TextBlock` mit der rechten Maustaste an und wählen Sie *Layout zurücksetzen/Alle*.

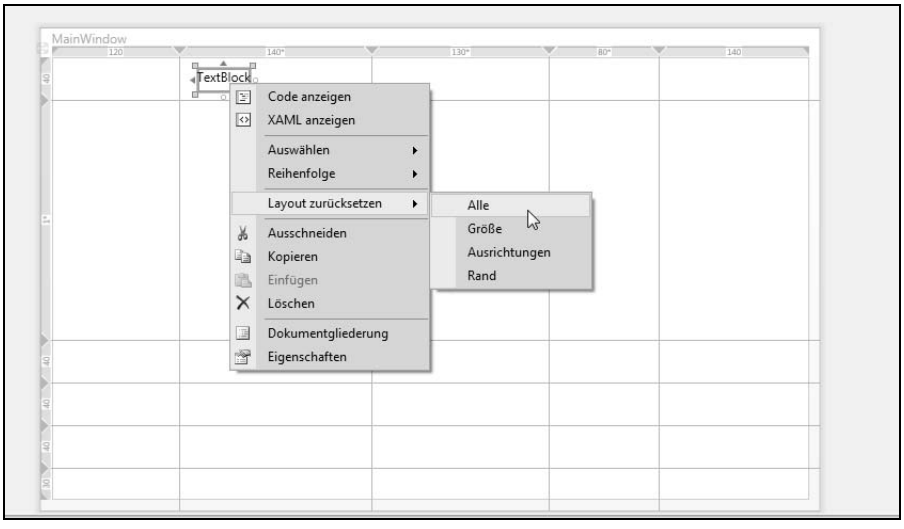

Abbildung 6.24 Positionierung des *TextBlock* und Anpassen des Layouts

11. Das Ergebnis ist ein `TextBlock`, der die Zelle vollständig ausfüllt. Ändern Sie rechts im Eigenschaftenfenster den Wert der Eigenschaft `Text` von *TextBlock* auf **ImageResizer**. Das Eigenschaftenfenster können Sie auch aufrufen, indem Sie das Steuerelement mit der rechten Maustaste anklicken und den Menüeintrag *Eigenschaften* auswählen.

Abbildung 6.25 Die Text-Eigenschaft im Eigenschaftenfenster ändern

12. Ändern Sie auch die Eigenschaften in der Kategorie *Text*, bis sie denen in Abbildung 6.26 entsprechen.

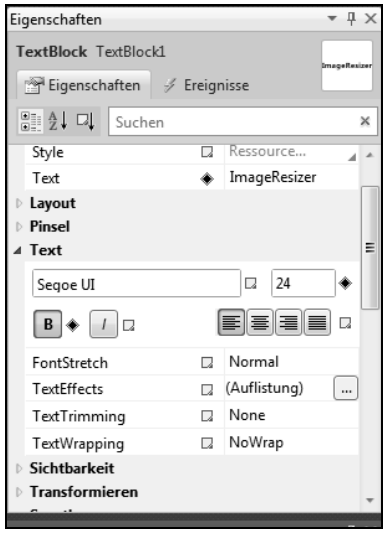

Abbildung 6.26 Anpassen der Eigenschaften der Kategorie *Text* im Eigenschaftenfenster

13. Die Überschrift soll sich in der Mitte der obersten Zeile befinden und sich über alle 5 Spalten erstrecken. Um dieses Ergebnis zu erreichen, müssen folgende Eigenschaften in der Kategorie *Layout* geändert werden: `HorizontalAlignment` erhält den Wert *Center*, `ColumnSpan` den Wert *5* und `Column` den Wert *0* (Abbildung 6.27).

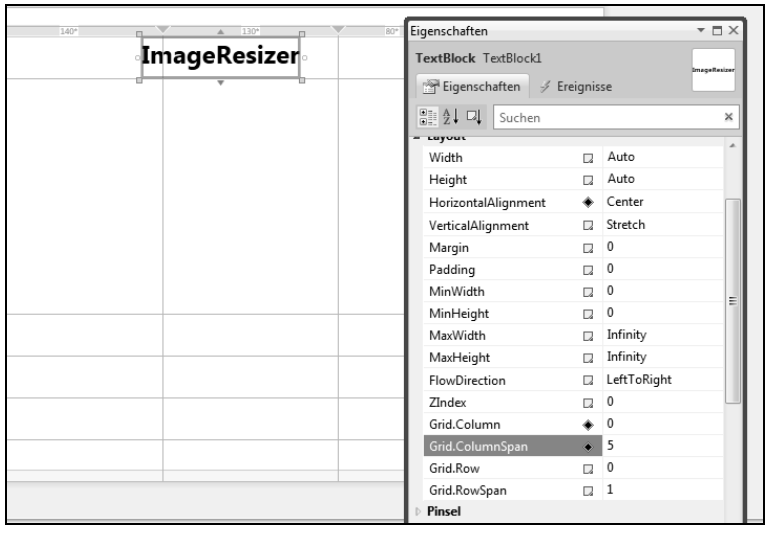

Abbildung 6.27 Anpassen der Eigenschaften der Kategorie Layout

Der `TextBlock` befindet sich in Spalte 0, erstreckt sich über 5 Spalten und positioniert sich zentriert. All diese Eigenschaften finden sich auch im XAML-Code wieder. Der WPF-Designer hat XAML-Code generiert, der unsere Änderungen ausdrückt:

```
<TextBlock Grid.Column="0" Name="TextBlock1" Text="ImageResizer" FontSize="24" FontWeight="Bold"
HorizontalAlignment="Center" Grid.ColumnSpan="5" />
```

Als Nächstes wird die `ListBox` für die Anzeige der Dateinamen auf die Oberfläche gebracht. Dazu verwenden wir dieses Mal nicht den WPF-Designer, sondern, um zu trainieren und zu zeigen, wie es schneller und einfacher gehen kann, direkt den XAML-Editor:

14. Positionieren Sie die Einfügemarke im XAML-Code eine Zeile unterhalb der eben eingefügten `TextBox` und tippen Sie **<Lis**. Sie werden sehen, dass IntelliSense auch hier aktiv ist und Ihnen entsprechende Vorschläge macht (Abbildung 6.28).

Abbildung 6.28 IntelliSense lässt Sie auch im XAML- Editor nicht alleine

15. Wählen Sie aus der Liste `ListBox` und tippen Sie **>**. Visual Studio komplettiert Ihre Eingabe mit dem entsprechenden Endetag. Vervollständigen Sie die Zeile folgendermaßen:

```
<ListBox Name="FilenamesListBox" Grid.ColumnSpan="5"></ListBox>
```

Die `ListBox` ist im Code Behind nun durch den Namen *FilenamesListBox* ansprechbar – das, was Sie also als Äquivalent aus Windows Form kennen, wenn Sie Namen für Elemente vergeben, um sie aus dem Visual Basic-Code heraus anzusprechen, haben wir gerade direkt in XAML gemacht. Wie Sie den jeweiligen Code in der Code-Behind-Datei hinterlegen, werden wir später noch sehen.

Darüber hinaus haben wir mit der `Grid.ColumnSpan`-Eigenschaft im XAML-Code direkt mitbestimmt, dass sich die `ListBox` über die kompletten 5 definierten Spalten des `Grid` erstrecken soll. Jedoch gibt es noch ein Problem: Sie liegt zunächst noch mit dem `TextBlock` in einer Zeile, die Standardeinstellung für die Zeile ist nämlich 0. Steht also *keine* explizite Zeilenangabe im Code, befindet sich das Steuerelement in der obersten (der nullten) Zeile des `Grid`. Das Gleiche gilt für die Spalten. Zudem wäre es schön, wenn die `ListBox` nicht am Rand »kleben«, sondern einen gewissen Abstand nach außen einhalten würde. Die Eigenschaft `Margin` kümmert sich darum.

16. Ergänzen Sie die Codezeile um folgende Angaben:

```
<ListBox Name="FilenamesListBox" Grid.ColumnSpan="5" Grid.Row="1" Margin="4,4,4,4"></ListBox>
```

TIPP Anstelle, wie im Beispiel, alle vier Ränder für die `Margin`-Eigenschaft einzeln einzugeben, können Sie, falls die Werte alle gleich lauten, auch die verkürzte Notierung verwenden, also in etwa wie folgt:

```
<ListBox Name="FilenamesListBox" Grid.ColumnSpan="5" Grid.Row="1" Margin="4"></ListBox>
```

Das spart Zeit und Platz und trägt obendrein zur Übersichtlichkeit bei.

Das Resultat sollte nun in etwa wie in Abbildung 6.29 dargestellt aussehen.

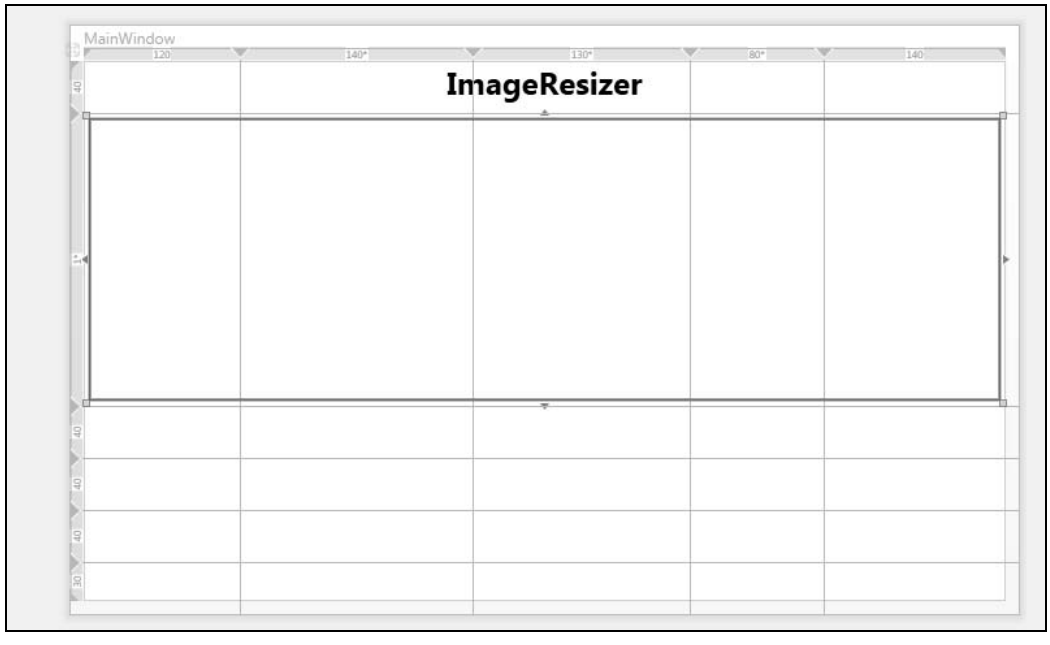

Abbildung 6.29 Die Oberfläche nach dem Einfügen des *TextBlock* und der *ListBox*

Dieses Wissen reicht für die weitere Oberflächengestaltung aus. Sie können nun Steuerelemente sowohl per Designer auf die Oberfläche bringen als auch per Hand implementieren. Welche Vorgehensweise Sie im Folgenden bevorzugen, ist Ihnen natürlich selbst überlassen.

Als Nächstes sollen die Schaltflächen ihren Weg auf die Oberfläche finden. Jeder `Button` hat die Eigenschaft *Content*, deren Inhalt die Beschriftung der Schaltfläche vorgibt.

17. Fügen Sie drei Steuerelemente des Typ `Button` mit folgenden Eigenschaften ein:

Button-Eigenschaften				
Name	**Row**	**Column**	**Content**	**Margin**
AddImagesButton	2	4	Bilder hinzufügen...	4
ChooseOutputFolderButton	3	4	Ausgabeordner...	4
StartButton	4	4	Start!	4

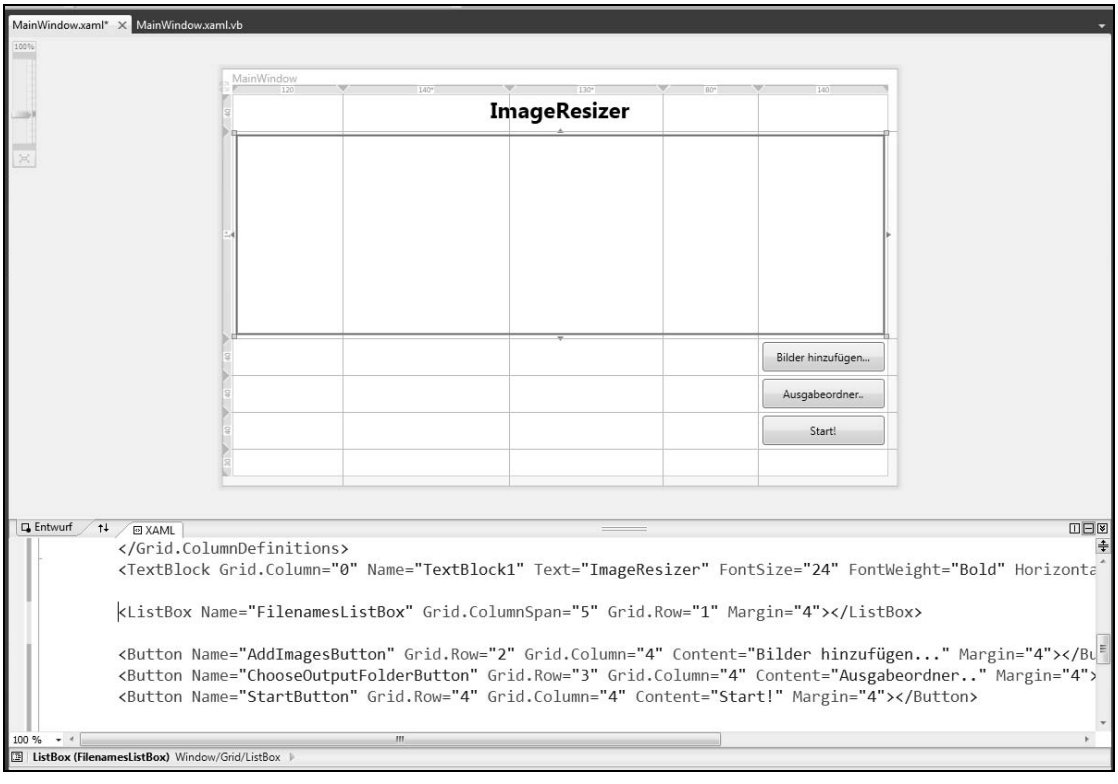

Abbildung 6.30 Fügen Sie der Oberfläche drei Schaltflächen hinzu

Als Nächstes kümmern wir uns um die Beschriftungen der Elemente im Fenster. Zu diesem Zweck platzieren Sie vier Steuerelemente vom Typ `Label` im `Grid`. Die Beschriftung wird ebenfalls wieder mit der Eigenschaft `Content` festgesetzt. Die Eigenschaften für die vier Labels können Sie der folgenden Tabelle entnehmen oder aber auch aus dem Entwurf ableiten. Die Eigenschaft `Name` der Labels muss nicht gesetzt werden, da sie rein dekorative Zwecke erfüllen und später nicht vom Code aus verändert werden müssen. Der Designer gibt den Steuerelementen ungeachtet dessen standardmäßig immer einen Namen.

HINWEIS Falls Sie den Code für die Labels per Hand eingeben, denken Sie daran, dass der Standardwert für Zeilen und Spalten 0 ist und deswegen bei den ersten drei Labels die Spaltenangabe nicht mitgetippt werden muss.

Die Eigenschaften `HorizontalContentAlignment` und `VerticalContentAlignment` kümmern sich um die Positionierung des Inhalts *innerhalb* des Steuerelements (links, zentriert, rechts, oben, unten).

Content	Row	Column	HorizontalContentAlignment	VerticalContentAlignment
Neue X-Auflösung:	2	0	Right	Center
Ausgabeordner:	3	0	Right	Center
Postfix:	4	0	Right	Center
Neues Bildformat:	2	2	Right	Center

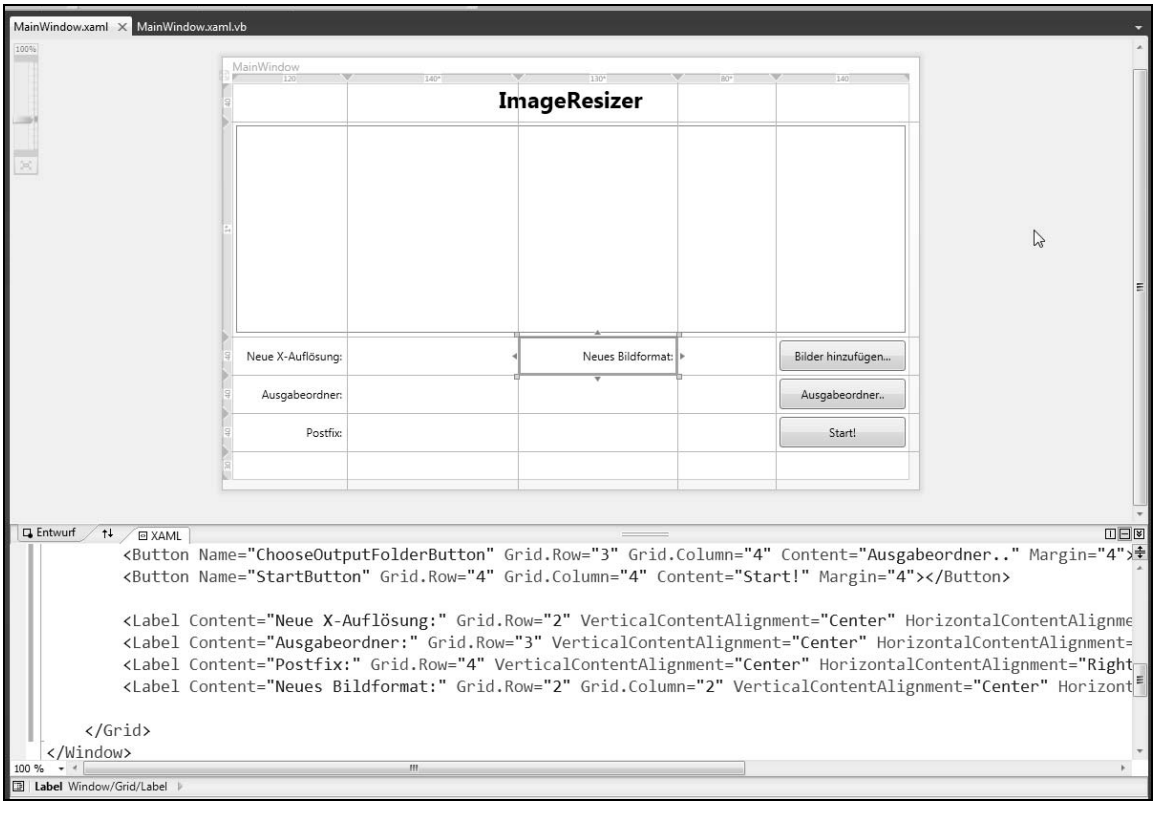

Abbildung 6.31 Fügen Sie der Oberfläche vier Labels hinzu

Um später zur Laufzeit die Eingaben des Benutzers entgegenzunehmen, fehlen noch Eingabefelder, und um die kümmern wir uns im Folgenden.

18. Fügen Sie dem Fenster drei Steuerelemente vom Typ `TextBox` mit folgenden Eigenschaften hinzu:

Name	Row	Column	Grid.ColumnSpan	Text	Margin	VerticalContentAlignment
X_ResTextBox	2	1	(Eigenschaft weglassen)	1024	4,6,0,6	Center
OutputFolderTextBox	3	1	3	(leer)	4,6	Center
PostfixTextBox	4	1	3	_klein	4,6	Center

Die Werte für den Außenabstand (`Margin`) sind hier anders angegeben als bisher. Ist nur eine Dezimalzahl spezifiziert, gilt der Wert für alle 4 Außenabstände – diese Eingabehilfe haben wir vorhin schon kennen gelernt; bei *zwei* Werten dagegen werden jeweils die Werte für den Abstand rechts und links bzw. oben und unten zusammengefasst: Die erste Zahl bezieht sich auf den Abstand der rechten und linken Seiten, die zweite auf den Abstand oben und unten. Sind alle 4 Angaben vorhanden, so wirken die Zahlen wie folgt auf die Seiten: links, oben, rechts, unten – also einmal im Uhrzeigersinn um das Steuerelement herum.

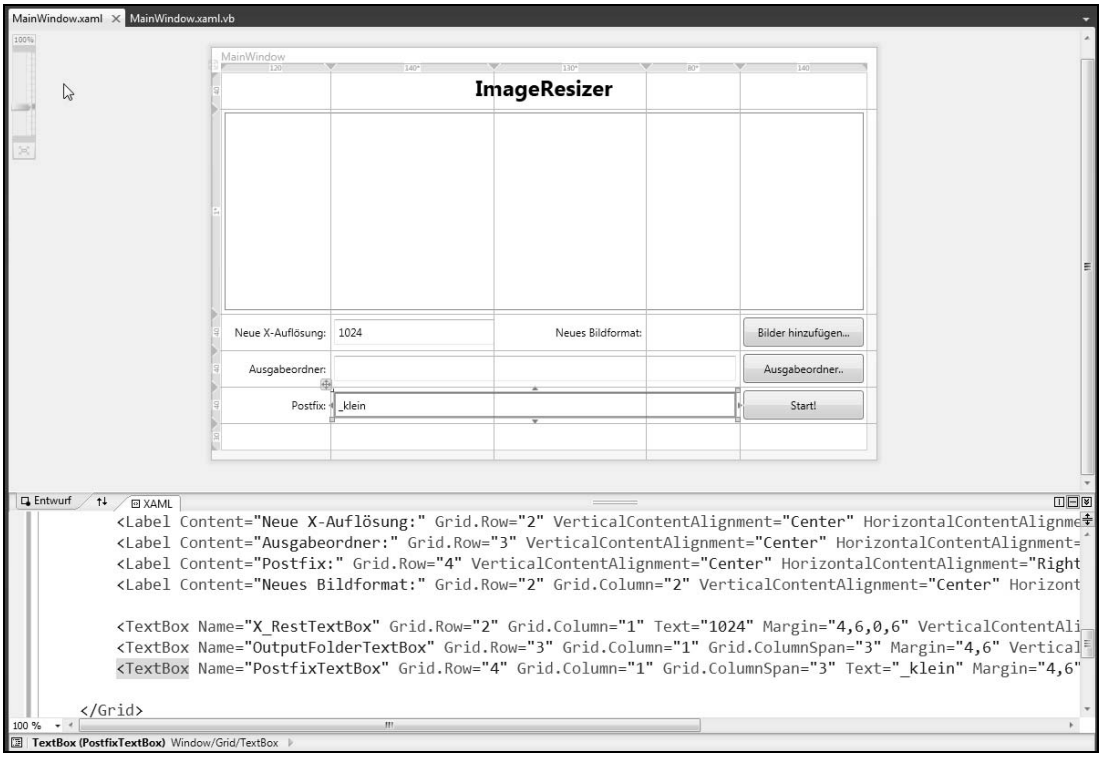

Abbildung 6.32 Fast fertig: Der Oberfläche fehlen jetzt nur noch zwei Elemente

Was noch fehlt, ist die Auswahlmöglichkeit für den Typ des Bildes.

19. Fügen Sie dafür den folgenden Code im XAML-Editor ein:

```
<ComboBox Grid.Column="3" Grid.Row="2" Name="FormatComboBox" Margin="4"
          VerticalContentAlignment="Center">
    <ComboBoxItem>.jpg</ComboBoxItem>
```

```
        <ComboBoxItem>.bmp</ComboBoxItem>
        <ComboBoxItem>.png</ComboBoxItem>
        <ComboBoxItem>.tif</ComboBoxItem>
    </ComboBox>
```

Was macht dieser Code? Er fügt eine `ComboBox` in die Zelle in Zeile 3 und Spalte 4 ein, nennt sie *FormatComboBox* und gibt ihr den Außenabstand 4. Der Inhalt wird angewiesen, sich zu zentrieren. Es werden 4 Listenelemente definiert, die dann zur Laufzeit zur Auswahl stehen – diese Vorgehensweise entspricht also der, wenn Sie im Code der `Items`-Auflistung der `ComboBox` Elemente mit der `Add`-Methode hinzufügen würden.

Um die Oberfläche zu komplettieren, fehlt nun noch die `StatusBar` in der letzten Zeile.

20. Ergänzen Sie diese wie folgt:

```
    <StatusBar Grid.ColumnSpan="5" Grid.Row="5" Name="StatusBar1">
        <StatusBarItem>
            <TextBlock Text="Gesamtfortschritt:"/>
        </StatusBarItem>
        <StatusBarItem>
            <ProgressBar Height="20" Width="150" Name="TotalProgressBar" />
        </StatusBarItem>
        <StatusBarItem>
            <TextBlock Text="Aktuelle Datei:" Name="CurrentFileTextBlock" />
        </StatusBarItem>
    </StatusBar>
```

Die `StatusBar` schachtelt hier drei untergeordnete Elemente, davon zwei Textblöcke und eine `ProgressBar`. Die `ProgressBar` namens *TotalProgressBar* wird zur Laufzeit den Fortschritt der Verkleinerung bzw. Umwandlung der Bilder anzeigen, und der TextBlock *CurrentFileTextBlock* wird darüber Auskunft geben, welche Datei gerade bearbeitet wird.

Damit ist die Oberfläche komplettiert und enthält nun unsere insgesamt 18 Steuerelemente. Und falls Sie XAML vorher noch nicht kannten, haben Sie nebenbei eine Menge über den Umgang mit dieser Auszeichnungssprache gelernt.

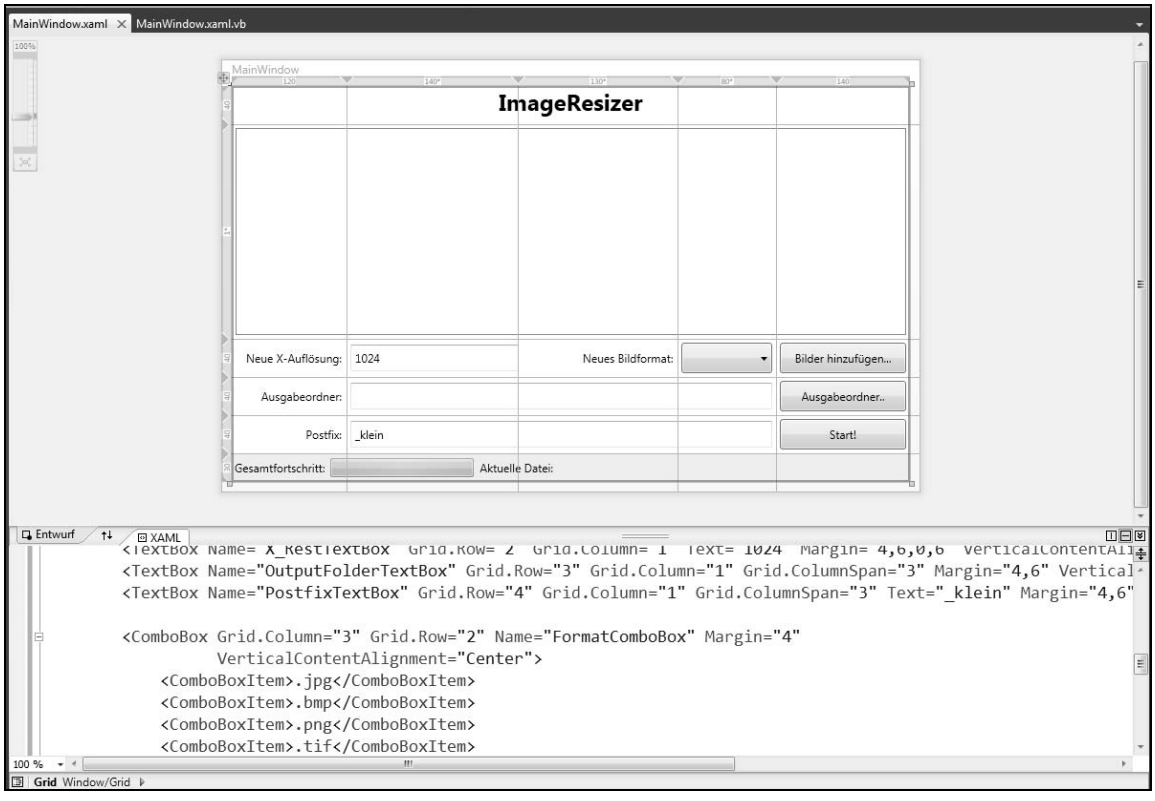

Abbildung 6.33 Die fertige Oberfläche

Nachträgliches Einfügen einer neuen Zeile

Nicht selten kommt nun die Stelle, wo der Fertigungsprozess Ihre Kreativität so angekurbelt hat, dass Ihnen noch eine (ultimative) Idee für die Oberflächengestaltung kommt. Wie wäre es beispielsweise mit einem Menü für den *ImageResizer*, mit einem Eintrag, mit dessen Hilfe das Programm beendet werden kann, alle Eingaben zurückgesetzt oder bestimmte Einstellungen geändert werden können? Doch leider ist auf unserem so toll durchgeplanten Fenster dafür nun leider kein Platz mehr. Alle Zeilen sind vollständig belegt und alle Steuerelemente müssten beim Einfügen einer neuen Zeile eine Position nach unten verschoben werden, was ziemlich aufwändig ist. Doch kein Problem ohne Lösung: Der Designer hilft in diesem Fall schneller als der XAML-Editor (einer der wenigen Fälle). Dabei gibt es einen kleinen Trick, den Sie im Folgenden nachvollziehen können:

21. Klicken Sie im Designer auf das `Grid` und platzieren Sie den Mauszeiger so auf dem blauen Rahmen, dass Sie mit der Linie die erste Zeile quasi halbieren, etwa wie in Abbildung 6.34 zu sehen.

Abbildung 6.34 Einfügen einer neuen Zeile: Schritt 1

22. Klicken Sie, um eine neue Zeile zu definieren. Sie werden sehen, dass der WPF-Designer die Definition der Zeilen angepasst hat. Die erste Zeile, die vormals die Höhe 40 hatte, ist nun in zwei Zeilen aufgeteilt, wobei Sie vielleicht die Mitte genauer getroffen haben als hier und nun über zwei Zeilen mit der Höhe 20 verfügen.

```
<Grid.RowDefinitions>
            <RowDefinition Height="17" />
            <RowDefinition Height="23" />
            <RowDefinition Height="*" />
            <RowDefinition Height="40" />
            <RowDefinition Height="40" />
            <RowDefinition Height="40" />
            <RowDefinition Height="30" />
        </Grid.RowDefinitions>
```

Auch hat der WPF-Designer die Zeilenangaben aller Steuerelemente automatisch für Sie angepasst.

Abbildung 6.35 Einfügen einer neuen Zeile: Die Zeilenangaben der darunter liegenden Zeilen wurden automatisch angepasst.

Das ging doch schnell, oder nicht? Das gleiche Prinzip lässt sich natürlich auch für Spalten anwenden. Passen Sie die neuen Zeilen in ihrer Höhe an, sodass die oberste Zeile den Wert *35* annimmt und die folgende den Wert *40*. Der Designer hat die Angaben der Position des `TextBlock` dahingehend geändert, dass seine vorherige Position im `Grid` unverändert ist:

```
<TextBlock Grid.Column="1" Name="TextBlock1" Text="ImageResizer" FontSize="24" FontWeight="Bold"
HorizontalAlignment="Center" Grid.ColumnSpan="2" Margin="159,0,37,0" Grid.RowSpan="2" />
```

Die Eigenschaft `RowSpan` hat nun den Wert *2*, das heißt, dass sich der `TextBlock` nun über beide Zeilen erstreckt. Gleichzeitig hat der Designer festgestellt, dass der `TextBlock` für eine Spalte zu breit ist und den Platz von zwei Spalten benötigt. Auch besitzt der `TextBlock` einen Abstand nach links und rechts, welchen der Designer in der Eigenschaft `Margin` festgehalten hat. Das ist ein Beispiel für Code des Designers, wie er im Buche steht. Der Designer hat den `TextBlock` in die zweite Spalte (also `Column="1"` wegen der nullbasierten Zählung) geschoben, ihm zwei Spalten Platz zugewiesen und ihm des Weiteren noch einen Außenabstand verpasst, der ihn genau dahin positioniert, wo er sich vor der Zeilenaufteilung auch befunden hat. Der `TextBlock` steht an der richtigen Position, und doch bleibt die Frage, ob diese Art der Positionierung nicht verwirrend oder umständlich ist, denn: Der `TextBlock` sollte sich eigentlich nur in der ersten Zeile befinden und sich in der Mitte *aller* 5 Spalten positionieren. Um das zu erreichen, ändern Sie den Code wie folgt:

- Löschen Sie die Angaben für `Margin`.
- Löschen Sie die Angaben für `RowSpan`.
- Ändern Sie die Angabe für `ColumnSpan` auf *5*.
- Ändern Sie die Angabe für `Column` auf *0*.
- Setzen Sie den Wert für `Row` auf *1*.

Das Ergebnis sieht damit in etwa wie folgt aus:

```
<TextBlock Grid.Column="0" Grid.Row="1" Grid.ColumnSpan="5" HorizontalAlignment="Center"
                  Name="TextBlock1" Text="ImageResizer" FontSize="24" FontWeight="Bold" />
```

Sie könnten einwenden, dass sich ein solches Verhalten als »Pingelei« bezeichnen ließe. Doch behalten Sie im Hinterkopf: Sollten Sie einmal gezwungen sein, den Code einer komplexen Benutzeroberfläche, die mit dem Designer erstellt wurde, per Hand zu ändern, werden Sie viel Verständnis für die Denkweise des Designers und somit die Strukturierung des Codes aufbringen müssen, um ihn und all seine Konsequenzen vollständig zu verstehen.

Einmal das Hauptmenü, bitte!

23. Um die eigentliche Idee des Menüs umzusetzen, fügen Sie im Code oberhalb des `TextBlock` mit der Überschrift folgenden Code hinzu.

```
<Menu Grid.ColumnSpan="5" Margin="5"></Menu>
```

Wer Menüs aus Windows Forms und dem dazugehörigen Designer schätzt, wird im WPF-Designer auch in der vorliegenden Version von Visual Studio 2010 eher enttäuscht werden. Das Menü lässt sich nur per Hand im XAML-Editor aufbauen; Designerunterstützung: Fehlanzeige. Das hat aber auch einen Vorteil: Das Aufbauen eines Menüs per Codeeingabe geht schnell von der Hand und ist außergewöhnlich flexibel, da seine Struktur jeglichem Kreativitätsanspruch standhält: Das Menü weist Menüelemente auf, die wiederum ebenfalls Menüelemente oder auch jedes andere Steuerelement enthalten können. Sie könnten also rein theoretisch einen Menüeintrag erstellen, der eine Combobox beinhaltet, die beim Öffnen abspielende Videos zeigt – das Konzept der `Context`-Eigenschaft macht es möglich! Menüelemente haben eine Eigenschaft namens `Header`, deren Wert im Normalfall den Text des Menüelements definiert, aber eben auch Bilder, Videos und jedes andere Steuerelement enthalten kann.

Der klassische Fall der Menüführung über den Menüeintrag *Datei* zum Befehl *Beenden* wird also umgesetzt so aussehen:

```
    <Menu Grid.ColumnSpan="5" Margin="5">
          <MenuItem Header="Datei">
              <MenuItem Header="Beenden"></MenuItem>
          </MenuItem>
      </Menu>
```

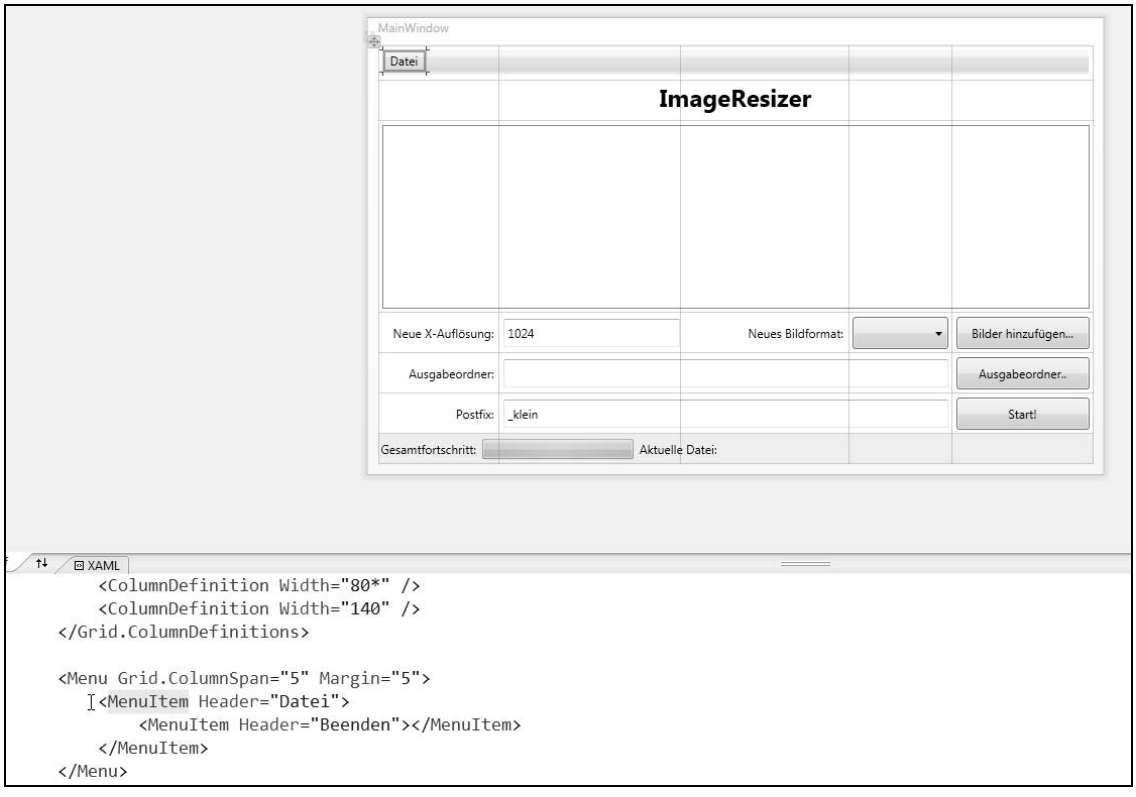

Abbildung 6.36 Das eben hinzugefügte Menü: Code und Ansicht im Designer

Das Element mit der Aufschrift *Beenden* ist also ein untergeordnetes Element des Elements mit der Aufschrift *Datei* und wird zur Laufzeit als Unterpunkt zu sehen sein. Probieren Sie es aus:

24. Ergänzen Sie den Code entsprechend dem oberen Listing, und starten Sie das Programm, indem Sie wahlweise F5 drücken oder aber über den Menüeintrag *Debuggen* den Befehl *Debugging starten* wählen.

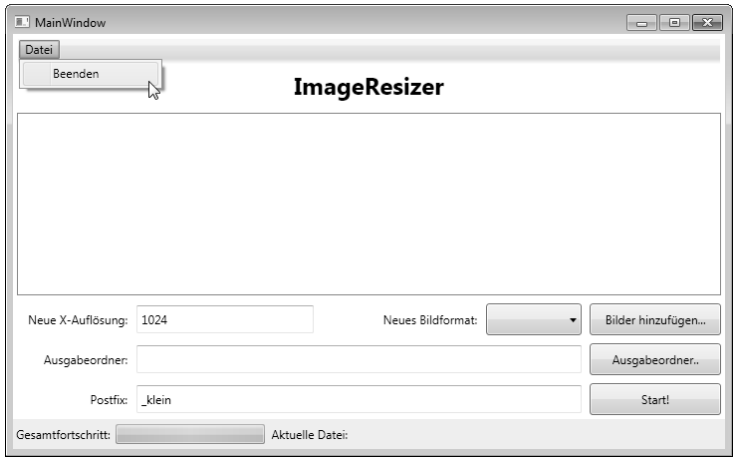

Abbildung 6.37 Das Menü zur Laufzeit

Steuerelemente mit Ereignissen verknüpfen

Da schön auszusehen ja bekanntlich nicht reicht, geht es jetzt an die Funktionalität. Beenden Sie zunächst das Debugging über das »x«-Symbol im Fenster oder die entsprechende Schaltfläche in Visual Studio.

25. Platzieren Sie die Einfügemarke im Menüelement *Beenden* und geben Sie dem Menüelement den Namen *MenuItemBeenden*. Schreiben Sie dann weiter **Click=**. IntelliSense zeigt Ihnen alle verfügbaren Methoden, die sich bereits im Code Behind befinden und die mit der Signatur des `Click`-Ereignisses des `MenuItem` übereinstimmen und gibt Ihnen darüber hinaus die Möglichkeit, einen neuen Methodenrumpf für die Behandlung des `Click`-Ereignisses dort hinzuzufügen. Da das Fenster noch keine Methoden besitzt, sehen Sie für unser Beispiel lediglich die Möglichkeit, einen neuen Methodenrumpf einzufügen. Nehmen Sie diese Möglichkeit wahr, und klicken Sie doppelt auf `<Neuer Ereignishandler>` (Abbildung 6.38).

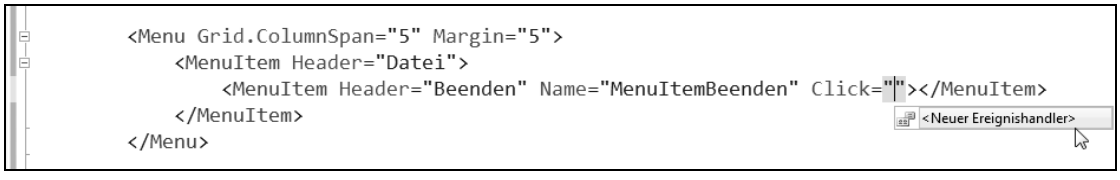

Abbildung 6.38 Das *Click*-Ereignis des Menüelements mit einer Methode verknüpfen

26. Drücken Sie F7, um in die Code-Behind-Datei zu gelangen oder klicken Sie im Projektmappen-Explorer doppelt auf die Datei *MainWindow.xaml.vb*, um sie zu öffnen. Sie werden feststellen, dass Visual Studio den Methodenrumpf eingefügt hat.

27. Vervollständigen Sie den Methodenrumpf wie folgt, und starten Sie dann das Debugging:

```
Private Sub MenuItemBeenden_Click(ByVal sender As System.Object,
    ByVal e As System.Windows.RoutedEventArgs)
        Me.Close()
    End Sub
```

Wenn Sie nun zur Laufzeit auf das Menüelement *Beenden* klicken, wird das Fenster (und damit das Programm) geschlossen.

Laden der Standardeinstellungen

Der Ordner, in dem sich die bearbeiteten Bilder sehr wahrscheinlich wiederfinden sollen, ist der standardmäßige, für diesen Zweck vorgesehene Windows-Ordner *Eigene Bilder* bzw. *Bilder* (bzw. zur übersichtlicheren Strukturierung ein Unterordner davon) des angemeldeten Benutzers. Dieser Pfad soll beim Laden des Fensters schon voreingestellt sein. Auch soll standardmäßig das Bildformat *.jpg* ausgewählt sein.

Sie können Methoden auch direkt über den Code mit einem Ereignis verdrahten. Tippen Sie dazu den Methodenrumpf der Ereignisroutine und verknüpfen Sie diese mittels des Schlüsselworts `Handels` mit dem Ereignis Ihrer Wahl:

```
Private Sub MainWindow_Loaded(ByVal sender As Object,
                              ByVal e As System.Windows.RoutedEventArgs) Handles Me.Loaded

    End Sub
```

Diese Routine ist mit dem `Loaded`-Ereignis des Fensters verknüpft, was bedeutet, dass der Code ausgeführt wird, sobald das Fenster geladen ist – der ideale Zeitpunkt zum Voreinstellen einiger Werte.

HINWEIS Richtig geübte OOPler unter Ihnen werden übrigens vergeblich nach einer überschreibbaren Methode `OnLoaded` in der `Windows`-Klasse suchen. Wieso einige der ereignisauslösenden Methoden der `Windows`-Klasse (und ihrer Basisklassen – `Windows` ist von `FrameworkElement` abgeleitet) überschreibbar sind, und andere nicht, kann ich Ihnen auch nicht verraten. Tatsache ist, dass das `Loaded`-Ereignis nur über das Binden des eigentlichen Ereignisses erreichbar ist. Wichtig ist das zu wissen, wenn Sie es von Windows Forms gewohnt sind, sich beim Finden der Ereignisse auf die IntelliSense-Liste der überschreibbaren Methoden zu verlassen, die Sie erhalten, wenn Sie innerhalb einer Windows-Klasse **overrides** gefolgt von einem Leerzeichen in den Editor eintippen.

Übernehmen Sie weiterhin den oben stehenden Code, falls noch nicht geschehen, und vervollständigen Sie ihn wie folgt:

```
Class MainWindow

    Private Sub MainWindow_Loaded(ByVal sender As Object,
                                  ByVal e As System.Windows.RoutedEventArgs) Handles Me.Loaded

        'JPeg vorselektieren
        FormatComboBox.SelectedIndex = 0

        'Eigene Bilder + ImageConverter + Tagesdatum als Vorgabeverzeichnis bestimmen
        Dim myOutputFolder As String
        myOutputFolder = Environment.GetFolderPath(Environment.SpecialFolder.MyPictures)
        myOutputFolder &= "\ImageConverter\" & Now.ToString("yyyy-MM-dd")
        OutputFolderTextBox.Text = myOutputFolder
    End Sub

    Private Sub MenuItemBeenden_Click(ByVal sender As System.Object, ByVal e As
System.Windows.RoutedEventArgs)
        Me.Close()
    End Sub

End Class
```

28. Starten Sie das Programm mit F5, und der Pfad zu Ihrem Bilderordner sollte sich in der entsprechenden `TextBox` vorfinden (Abbildung 6.39).

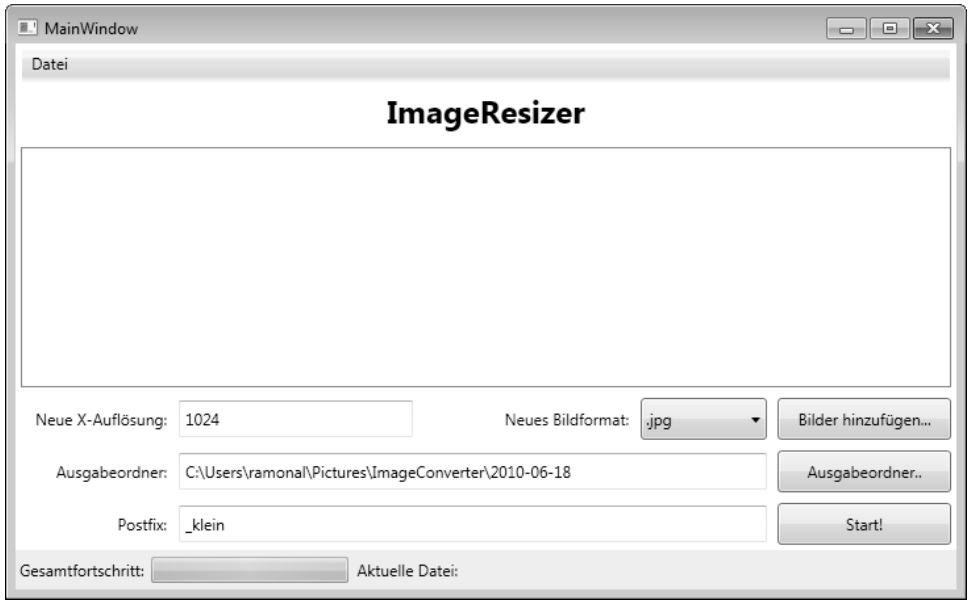

Abbildung 6.39 Beim Laden des Fensters wird das Bildformat vorselektiert und der Ausgabeordner auf den benutzereigenen Bilderordner voreingestellt

Organisation ist alles: den Ausgabepfad frei definierbar machen

Falls der voreingestellte Ausgabeordner zugunsten von Übersicht und Ordnung geändert werden soll, muss ein geeigneter und benutzerfreundlicher Dialog her. Dieser Dialog wird uns von .NET Framework zur Verfügung gestellt, sodass der Aufwand auf ein paar Einstellungen beschränkt wird.

29. Verknüpfen Sie das `Click`-Ereignis der Schaltfläche *ChooseOutputFolderButton* mit einer entsprechenden Routine. Auf welche Weise, ob per `Handles`-Schlüsselwort oder via XAML-Code, das ist Ihnen überlassen. Im XAML-Code sieht das so aus:

```
<Button Name="ChooseOutputFolderButton" … Click="ChooseOutputFolderButton_Click"></Button>
```

Die Klasse, die den Dialog zur Verfügung stellt, heißt `FolderBrowserDialog`. Um sie instanziieren und verwenden zu können, muss allerdings ein wenig Vorarbeit geleistet werden, denn den Dialog gibt es (noch) nicht in der WPF-Funktionsbibliothek. Er ist aber unter Windows Forms verfügbar, und es ist durchaus legitim, uns Funktionalitäten für unsere Benutzeroberfläche bei den Windows Forms-Jungs »auszuborgen«. Und das geht so:

30. Klicken Sie zu diesem Zweck mit der rechten Maustaste auf das Projekt *ImageResizer* (nicht auf die Projektmappe!) und wählen Sie *Verweis hinzufügen* (Abbildung 6.40).

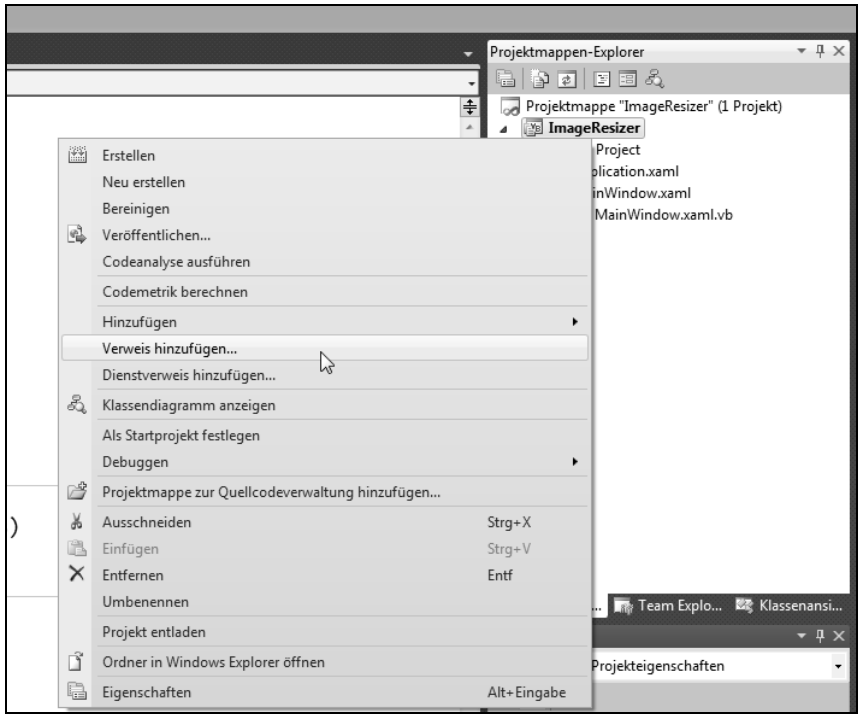

Abbildung 6.40 Um einen Verweis hinzuzufügen, klicken Sie mit der rechten Maustaste auf das Projekt und wählen den entsprechenden Menüeintrag aus

31. Suchen Sie nun im sich öffnenden Dialog auf der Registerkarte *.NET* die Assembly *System.Windows.Forms* (Abbildung 6.41). Wählen Sie diese aus, und bestätigen Sie den Dialog mit *OK*.

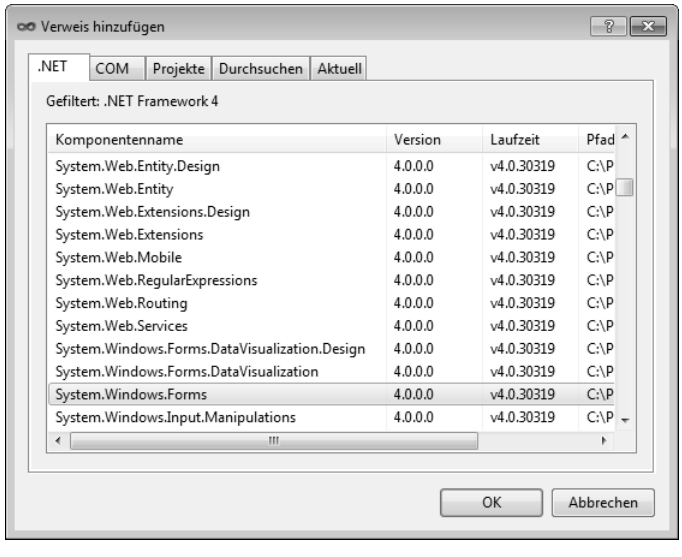

Abbildung 6.41 Binden Sie den Assembly-Verweis auf *System.Windows.Forms* ein

Zurück im Code des Fensters *MainWindow* sollen Klassen aus der Assembly verwendet werden, die wir gerade eingebunden haben. Um das möglich zu machen, tippen Sie folgenden Code ein. Dabei wichtig: Er muss noch vor der Klassendefinition stehen!

```
Imports System.Windows.Forms

Class MainWindow
        …
End Class
```

Da die Klasse, die uns die Ordnerauswahl auf dem Bildschirm anzeigen lässt, nun erreichbar ist, können wir Sie mit entsprechendem Code instanziieren und anschließend verwenden. Wir erlauben dem Benutzer, mit dem Wert der Eigenschaft `ShowNewFolderButton` neue Ordner im Dialog anzulegen, und geben dem Dialog über die Eigenschaft `Description` den Titel *Zielverzeichnis wählen*. Der Dialog wird mit der Methode `ShowDialog` aufgerufen und das Programm wartet so lange mit der Ausführung des weiteren Codes, bis der Dialog ein Ergebnis erzielt hat: Er wurde entweder abgebrochen oder bestätigt. Wird der Dialog mit *OK* bestätigt, ist der neue Ordner gewählt, und der Pfad kann im entsprechenden Eingabefeld angezeigt werden. Den Code dazu sehen Sie im Folgenden:

```
  Private Sub ChooseOutputFolderButton_Click(ByVal sender As System.Object,
                                             ByVal e As System.Windows.RoutedEventArgs)

        Dim folderBrowser = New FolderBrowserDialog
        folderBrowser.ShowNewFolderButton = True
        folderBrowser.Description = "Zielverzeichnis wählen"
        Dim dr = folderBrowser.ShowDialog
        If dr = Forms.DialogResult.OK Then
            OutputFolderTextBox.Text = folderBrowser.SelectedPath
        End If
    End Sub
```

Starten Sie einen Testlauf via F5.

Letzte Vorbereitungen treffen: Bilder hinzufügen

Die Bühne steht, doch die Protagonisten fehlen: die Bilder, die bearbeitet werden sollen. Beim Klicken auf die Schaltfläche soll ein Dialog das Hinzufügen der Bilder ermöglichen. Verdrahten Sie als Erstes das `Click`-Ereignis der Schaltfläche *AddImagesButton* mit einer Methode. Dort wird der altbekannte Datei-Auswahl-Dialog aufgerufen. Da dieser in zwei Versionen in verschiedenen Bibliotheken zur Verfügung steht, nämlich in Win32 und in Forms, muss er voll referenziert werden, um Verwechslungen auszuschließen:

```
Private Sub AddImagesButton_Click(ByVal sender As System.Object, _
                                  ByVal e As System.Windows.RoutedEventArgs) _
Handles AddImagesButton.Click

        'Wir verwenden den OpenFileDialog aus Win32, weil er für WPF "gemacht" ist -
        'müssen ihn aber voll qualifizieren, weil es ihn in .Forms und .Win32 gibt.
        Dim fod = New Microsoft.Win32.OpenFileDialog
    End Sub
```

Beim Aufrufen des Dialogs sind einige Einstellungen für den Programmablauf wichtig. Damit das Programm ordnungsgemäß laufen kann, müssen die Dateipfade und Dateien existieren sowie die Dateinamen gültig sein und vom Betriebssystem akzeptiert werden. Den Code dafür sehen Sie im Folgenden:

```
'nur existierende Dateien und Pfade dürfen gewählt werden.
fod.CheckPathExists = True
fod.CheckFileExists = True

'nur gültige Win32 Dateinamen werden akzeptiert
fod.ValidateNames = True
```

Um dem Benutzer die Handhabung angenehm zu gestalten, soll er mehrere Bilder gleichzeitig auswählen können:

```
'es dürfen mehrere Dateien ausgewählt werden
fod.Multiselect = True
```

Der Filter des Dialogs wird mit Bilddateitypen bestückt und standardmäßig soll der Dateityp *.jpg* ausgewählt sein:

```
'der Filter soll nach Bilddateitypen filtern
fod.Filter = "JPeg-Dateien (*.jpg; *.jpeg)|*.jpg; *.jpeg" &
             "|Bitmap-Dateien (*.bmp)|*.bmp" &
             "|TIFF-Dateien (*.tif;*.tiff)|*.tif;*.tiff" &
             "|PNG-Dateien (*.png)|*.png" &
             "|Alle Dateien (*.*)|*.*"

'der Filtert steht standardmäßig auf "*.jpg"
fod.DefaultExt = "*.jpg"
```

Ist der Dialog nun bestätigt, müssen die Pfade der ausgewählten Dateien als Elemente in die `ListBox`. Im nachfolgenden Code entdecken Sie ein neues Feature von Visual Basic 2010: mehrzeilige Anweisungs-Lambdas. Mehr zu diesen nützlichen neuen Helfern erfahren Sie übrigens in Kapitel 18. Hier wird für jedes Element in der Liste der gewählten Dateinamen die kleine nachfolgende Routine ausgeführt: Das Element wird der Elementauflistung der `ListBox` hinzugefügt:

```
'Den DialogResult des Dialogs speichern
Dim dr = fod.ShowDialog
'und auf Erfolg überprüfen
If dr = True Then
    Array.ForEach(fod.FileNames, Sub(filename)
                                     FilenamesListbox.Items.Add(filename)
                                 End Sub)
End If
```

Endspurt: das Verkleinern der Bilder

Nun stehen alle zur Ausführung benötigten Informationen zur Verfügung, und die eigentliche Bearbeitung der Bilder kann beginnen. Dazu macht das Programm folgende Schritte:

- Einlesen der Informationen
- Bearbeiten der Bilder
- Speichern der Bilder
- Anzeigen des Fortschritts für den Benutzer

Hier gibt es einen Haken: In den WPF-Libraries existiert, anders als in Windows Forms, keine Bild-Klasse, mit der man Bilder speichern könnte. Also, eigentlich jedenfalls nicht. Mit diesem Buch haben Sie gleichzeitig auch einen voll funktionsfähigen Workaround erworben, der es Ihnen ermöglicht, auch in WPF Bilder zu speichern. Die vom Autor geschriebene Bibliothek mit dieser und einigen anderen Funktionalitäten finden Sie begleitend im Unterverzeichnis *WriteableBitmapManager* der Projektmappe dieses Beispiels (*ImageResizer*). Sie haben natürlich die Möglichkeit, die Funktionalitäten daraus auch in Ihren anderen Projekten zu verwenden.

Sie werden darüber hinaus auch mit dem Nachvollziehen der folgenden Schritte lernen, wie Sie Programmbibliotheken einer Projektmappe prinzipiell hinzufügen und die entsprechenden Referenzen setzen, damit Sie sie von Ihren Projekten heraus verwenden können.

Hinzufügen eines Projekts zu einer bestehenden Projektmappe und Setzen der Verweise

32. Kopieren Sie das Projekt *WriteableBitmapManager* in ein Verzeichnis Ihrer Wahl, beispielsweise in das Projektverzeichnis des *ImageResizer*, das Sie standardmäßig in Ihrem Windows-Dokumenteordner im Verzeichnis *Visual Studio 2010\Projects* finden.

33. Wechseln Sie zu Visual Studio und klicken Sie im Projektmappen-Explorer die Projektmappe *ImageResizer* mit der rechten Maustaste an.

34. Klicken Sie auf *Hinzufügen/Vorhandenes Projekt* (Abbildung 6.42).

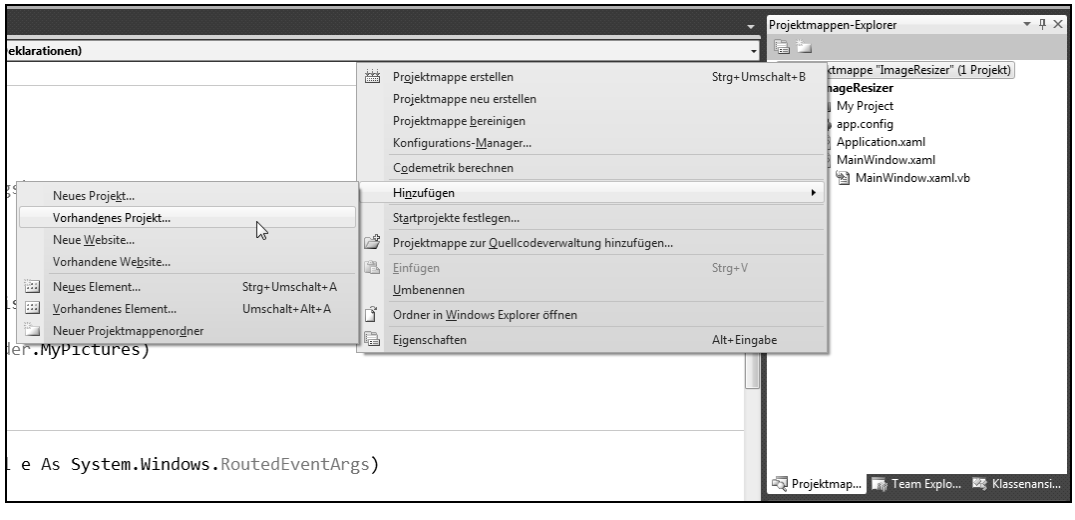

Abbildung 6.42 Hinzufügen eines vorhandenen Projekts zur Projektmappe

35. Navigieren Sie zu dem entsprechenden Ordner, wählen Sie die Projektdatei des Projekts *WriteableBitmapManager*, und bestätigen Sie den Dialog mit *Öffnen* (Abbildung 6.43).

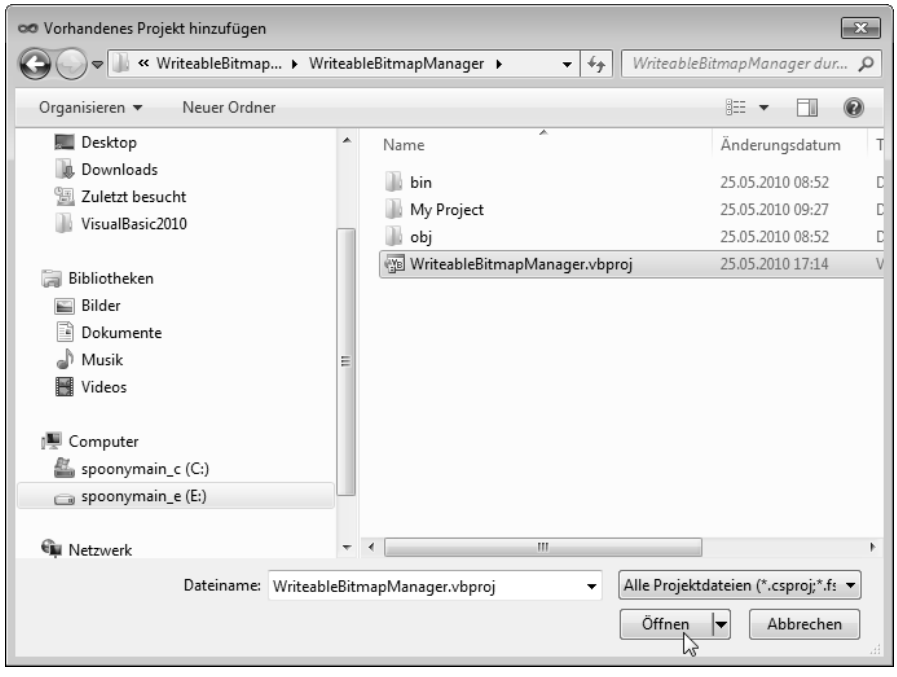

Abbildung 6.43 Wählen Sie die Projektdatei des *WriteableBitmapManager*

Das Projekt befindet sich nun in der Projektmappe.

36. Verfahren Sie wie oben beschrieben, um dem Projekt *ImageResizer* einen Verweis auf das Projekt *WriteableBitmapManager* hinzuzufügen. Wählen Sie dafür im Dialog die Registerkarte *Projekte* und das Projekt *WriteableBitmapManager* und klicken Sie *OK*.

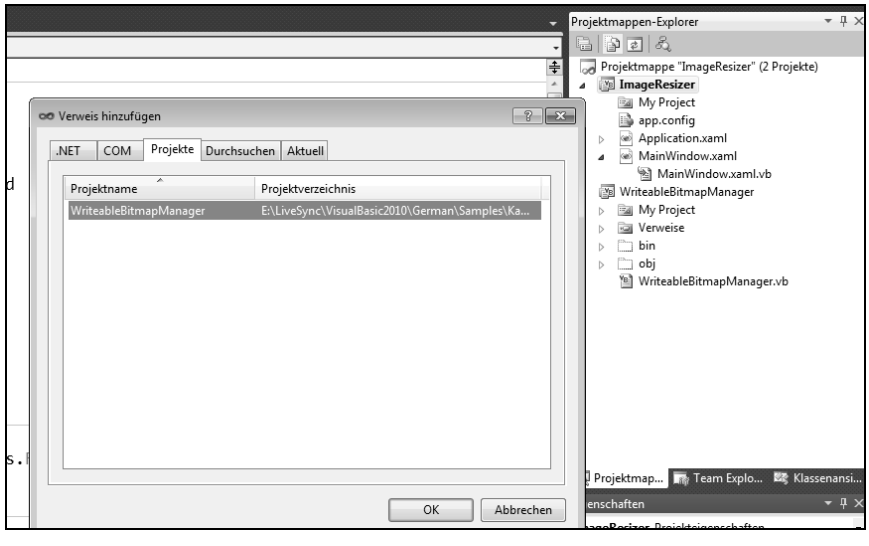

Abbildung 6.44 Einen Verweis auf das Projekt hinzufügen

37. Ergänzen Sie den Code im Fenster *MainWindow* um folgende (fett gedruckte) Zeile:

```
Imports System.Windows.Forms
Imports ActiveDevelop.Wpf.Imaging
Class MainWindow
…
End Class
```

Das Herzstück des Programms: die Funktionalität des Verkleinerns

Der Vorgang des Verkleinerns wird durch die Schaltfläche *StartButton* ausgelöst. Verdrahten Sie den `Click`-Event der Schaltfläche mit einer Methode. Für das Verkleinern der Bilder sind erst einmal folgende Informationen wichtig:

- Wie groß ist das neue Bild?
- Welchen Dateityp soll es erhalten?
- Welche Bilder sollen verkleinert werden?

Die Informationen dazu werden aus den Eingaben der Benutzer ermittelt. Geben Sie folgenden Code in den Methodenrumpf ein:

```
Dim myReduceToX = Integer.Parse(X_ResTextBox.Text)
Dim newExtension = DirectCast(FormatComboBox.SelectedItem, ComboBoxItem).Content.ToString
```

Initialisieren Sie die Fortschrittsanzeige in der `ProgressBar`, indem Sie den maximalen Fortschritt ermitteln und einen Zähler schaffen, der den bisher gemachten Fortschritt zwischenspeichert:

```
Me.TotalProgressBar.Maximum = FilenamesListbox.Items.Count
Dim progressCount = 0
```

Die folgenden Schritte müssen für jedes Bild ausgeführt werden:

- Den Namen des gerade bearbeiteten Bildes auf der Oberfläche anzeigen
- Den Dateinamen des Bildes in einer neuen Instanz der `FileInfo` Klasse speichern, die später Funktionalitäten zum Verwalten bzw. Bearbeiten zur Verfügung stellt
- Die Klasse `WriteableBitmapManager` instanziieren und anweisen, eine dem Bild entsprechende Kopie anzulegen
- Errechnen des Faktors, um den das Bild verkleinert werden soll
- Die Kopie des Bildes um den errechneten Faktor verkleinern, dabei die Zeit stoppen
- Umbenennen des Bildes entsprechend der Auswahl des Dateityps
- Speichern des Bildes mithilfe der `WriteableBitmapManager`-Instanz im angegebenen Ordner mit dem angegebenen Postfix
- Den Fortschrittszähler um eins erhöhen und den aktuellen Fortschritt in der `ProgressBar` anzeigen

Um die Funktionalität der `FileInfo`-Klasse nutzen zu können, muss noch folgender Namespace importiert werden:

```
Imports System.IO
```

Fügen Sie dem Projekt *ImageResizer* einen weiteren Verweis hinzu, indem Sie das Projekt mit der rechten Maustaste anklicken, *Verweis hinzufügen* auswählen und auf der Registerkarte *.NET* die Bibliothek *System.Drawing* auswählen. Umgesetzt in Code sieht die Prozedur nun so aus:

HINWEIS Der folgende Code, der sich in der `StartButton_Click`-Methode befindet, ist vielleicht ein wenig zu viel zum Abtippen. Wenn Sie Ihr Projekt bis hierhin nachgebaut haben, können Sie sich die Tipparbeit im Folgenden sparen und im Beispielverzeichnis dieses Kapitels die Textdatei *ImageResizer_SButtonClick.txt* öffnen, die den Code vollständig enthält und diesen von dort aus in die Ereignismethode kopieren.

```
Private Sub StartButton_Click(ByVal sender As System.Object, ByVal e As System.Windows.RoutedEventArgs)

        'neue Breite einlesen:
        Dim myReduceToX = Integer.Parse(X_ResTextBox.Text)
        'Dateityp einlesen
        Dim newExtension = DirectCast(FormatComboBox.SelectedItem, ComboBoxItem).Content.ToString

        'Zahl der Berechnungen ermitteln und der Fortschrittsanzeige zuweisen
        Me.TotalProgressBar.Maximum = FilenamesListBox.Items.Count
        'Zähler für bisher fertig gestellte Berechnungen = Fortschritt
        Dim progressCount = 0

        'Für jedes Bild ..
        For Each filenameItem As String In FilenamesListBox.Items
            'den Dateinamen anzeigen
            CurrentFileTextBlock.Text = filenameItem
            'und speichern
            Dim fileInfo As New FileInfo(filenameItem)

            'den Manager instanziieren und eine Kopie des Bildes anfertigen
            Dim currentPicture As New WriteableBitmapManager(filenameItem)
            Dim mySmallWbm = currentPicture.CreateCompatible(myReduceToX)
            'den Verkleinerungsfaktor errechnen
            Dim pixelFakt As Double = currentPicture.WriteableBitmap.PixelWidth / myReduceToX

            'Stoppuhr anschmeißen
            Dim sw = Stopwatch.StartNew
            'Das Bild verkleinern
            For y = 0 To mySmallWbm.WriteableBitmap.PixelHeight - 1
                For x = 0 To myReduceToX - 1
                    mySmallWbm.SetPixel(x, y,
                        currentPicture.GetPixel(CInt(Math.Truncate(pixelFakt * x)),
                                                CInt(Math.Truncate(pixelFakt * y))))
                Next
            Next

            'und aktualisieren
            mySmallWbm.UpdateBitmap()
            'Stoppuhr stoppen
            sw.Stop()

            'Den Dateityp anpassen
            Dim newFilename = fileInfo.Name.Replace(fileInfo.Extension, "")
            'den Ordner und das Postfix einbauen
```

```
            newFilename = OutputFolderTextBox.Text &
                          "\" & newFilename & PostfixTextBox.Text & newExtension

            'Speichern
            mySmallWbm.SaveBitmap(newFilename)

            'und den Fortschritt anzeigen
            progressCount += 1
            TotalProgressBar.Value = progressCount

        Next
    End Sub
```

Starten Sie mit F5 einen Testlauf des Programms.

Von der Holz- zur Businessklasse

Nach dem Verkleinern der Bilder mit dem Programm kann ich ohne hellseherische Fähigkeiten mit großer Wahrscheinlichkeit behaupten, dass Sie den Ausgabeordner gesucht haben und sich die verkleinerten Bilder angeschaut haben.

Um dem Programm den letzten Schliff an Bedienkomfort zu geben, wäre es nicht schön, die kleinen Bilder sofort als Vorschau zu sehen? Kein Problem. Klicken Sie zunächst mit der rechten Maustaste auf das Projekt *ImageResizer*, wählen Sie *Hinzufügen* und klicken Sie auf *Neues Element* (Abbildung 6.45).

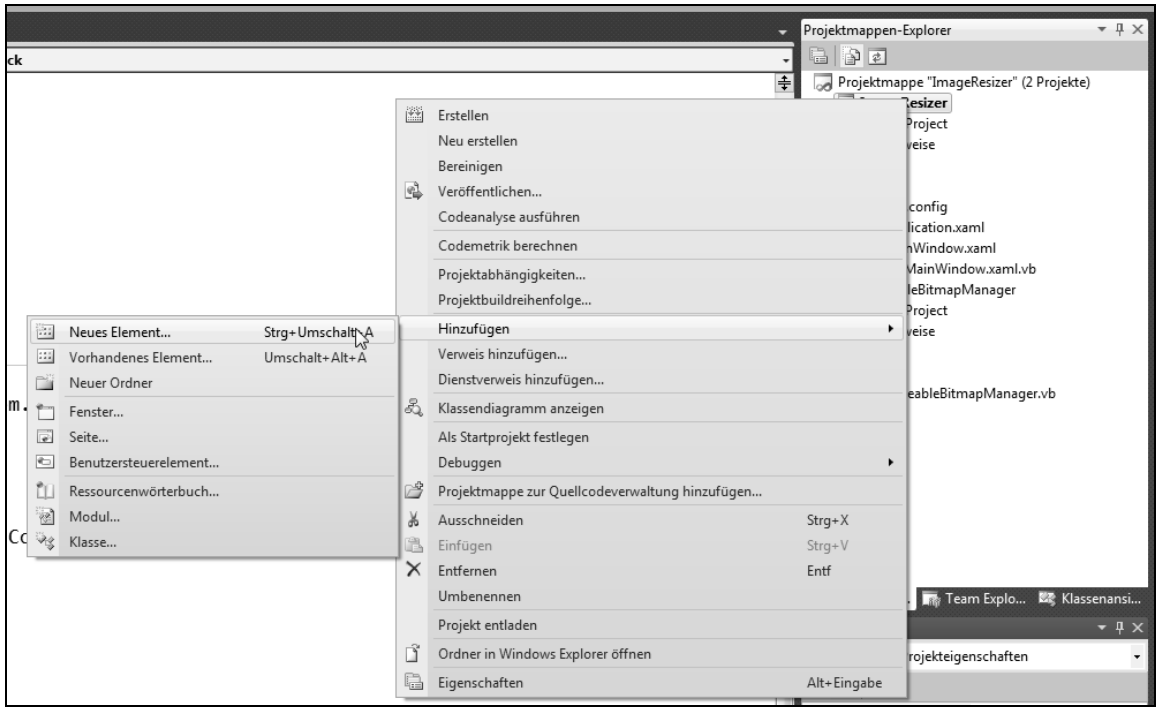

Abbildung 6.45 Ein neues Element hinzufügen

Wählen Sie im daraufhin angezeigten Dialog den Eintrag *Fenster (WPF)* und geben Sie ihm den Namen **BitmapViewer.xaml** (Abbildung 6.46). Bestätigen Sie den Dialog mit *Hinzufügen*.

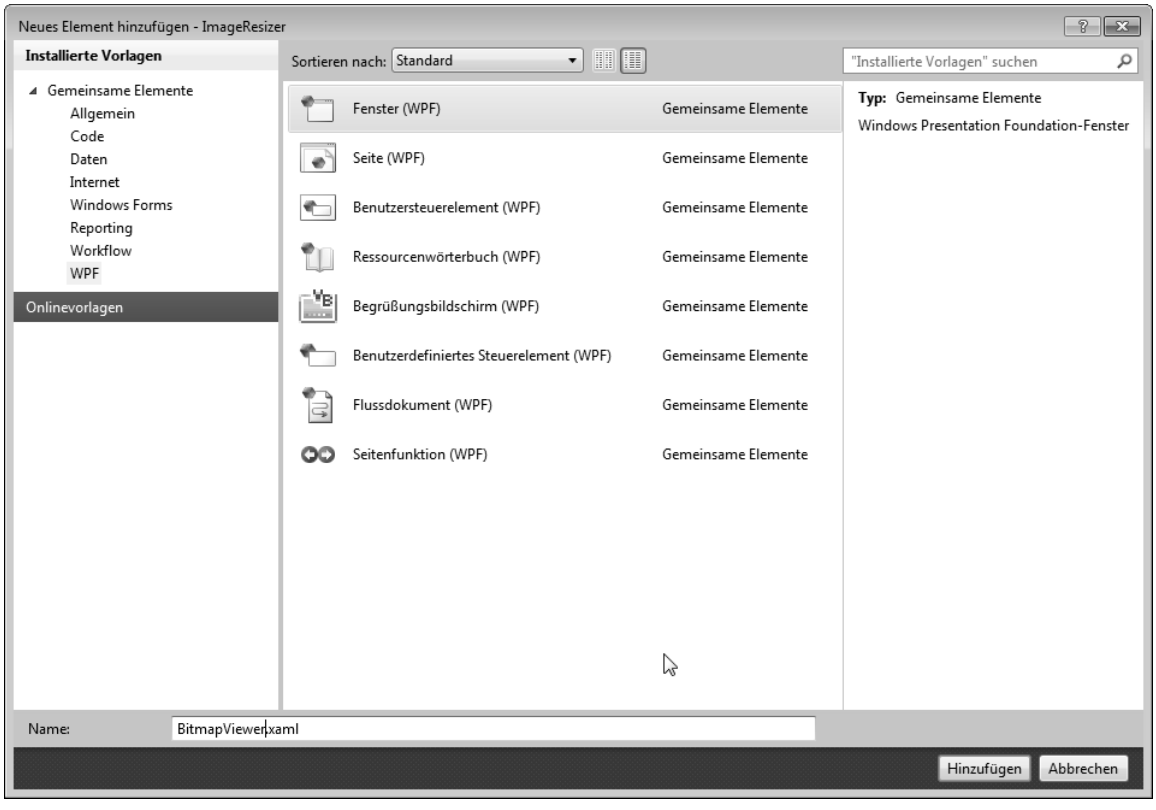

Abbildung 6.46 Ein neues Fenster hinzufügen

Das neue Fenster braucht nicht viel an Oberfläche. Fügen Sie in das `Grid` ein Steuerelement des Typs `ScrollViewer` ein und platzieren Sie darin ein `Image`. Setzen Sie die Eigenschaft `Stretch` des `Image` auf *None*, damit das Bild in seiner Originalgröße angezeigt wird, und geben Sie ihm den Namen **PreviewImage.**

Ziehen Sie das Fenster größer, bis es Ihnen groß genug erscheint, und setzen Sie die Startposition auf die Mitte des Bildschirms mithilfe der Eigenschaft `WindowStartupLocation` und dem Wert *CenterScreen*:

```
<Window x:Class="BitmapViewer"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="BitmapViewer" Height="448" Width="656" WindowStartupLocation="CenterScreen">
    <Grid>
        <ScrollViewer Name="ScrollViewer1" HorizontalScrollBarVisibility="Visible">
            <Image Name="PreviewImage" Stretch="None" />
        </ScrollViewer>
    </Grid>
</Window>
```

Wechseln Sie nun in die Code Behind-Datei, fügen Sie eine Methode namens *ShowPicture* ein, die als Übergabeparameter eine Variable vom Typ `WriteableBitmapManager` entgegennimmt sowie eine Variable vom Typ `String`. Implementieren Sie folgenden Code, um das Vorschaufenster zu komplettieren:

```
Public Sub ShowPicture(ByVal wbm As WriteableBitmapManager, ByVal statusText As String)
       PreviewImage.Width = wbm.WriteableBitmap.Width
       PreviewImage.Height = wbm.WriteableBitmap.Height
       PreviewImage.Source = wbm.WriteableBitmap
       Me.Title = statusText
   End Sub
```

Auch in diesem Fenster fehlt noch der Namespace für die Klasse `WriteableBitmapManager`.

TIPP Um fehlende Namespaces zu importieren, können Sie sich auch der Fehlerkorrektur bedienen, wie in Abbildung 6.47 zu sehen. Das erspart Ihnen nicht nur das Scrollen zum Anfang der Codedatei, sondern auch das Suchen nach dem entsprechenden Namespace-Namen, den es zu importieren gilt.

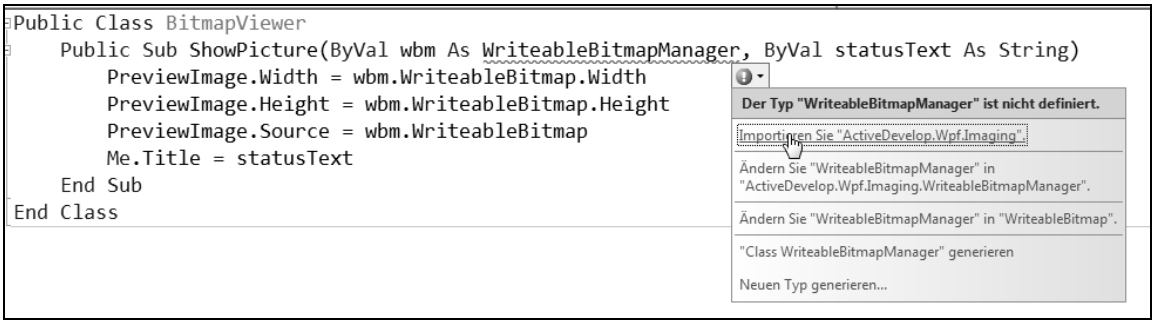

Abbildung 6.47 Visual Studio macht Ihnen Verbesserungsvorschläge bei auftretenden Fehlern, zum Beispiel bei fehlenden Verweisen

Fügen Sie die folgenden (fett gedruckten) Codezeilen in die Verkleinerungsroutine ein, um das Fenster aufzurufen, die einzelnen Bilder anzuzeigen und das Fenster wieder zu schließen:

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

       'Das Vorschaufenster instanziieren und anzeigen
       Dim picViewer As New BitmapViewer
       picViewer.Show()

       'neue Breite einlesen:
       Dim myReduceToX = Integer.Parse(X_ResTextBox.Text)
       'Dateityp einlesen
               …

       'Für jedes Bild ..
       For Each filenameItem As String In FilenamesListBox.Items
           'den Dateinamen anzeigen
           CurrentFileTextBlock.Text = filenameItem
```

```
                    …
            'und aktualisieren
            mySmallWbm.UpdateBitmap()
            'Stoppuhr stoppen
            sw.Stop()

            'das verkleinerte Bild anzeigen
            picViewer.ShowPicture(mySmallWbm, "Dauer der Konvertierung in ms: " &
                        sw.ElapsedMilliseconds.ToString("#,##0.00"))


                …
        Next

        picViewer.Close()

    End Sub
```

Wenn Sie das Programm nun starten, werden Sie feststellen, dass das Vorschaufenster zwar ordnungsgemäß in seiner Größe vorhanden ist, jedoch als einziges Bild das als zuletzt konvertierte angezeigt wird. Wie kommt das?

Da das ganze Programm einzig und allein auf einem einzigen `Thread` läuft, ist es während der Berechnung so beschäftigt, dass es die Oberfläche vernachlässigt und keine Kapazität zum Aktualisieren der Oberfläche freigibt. Wenn man die Berechnung nun auf einen anderen `Thread` verlegt, wäre die Kapazität zum Aktualisieren der Oberfläche frei, doch das soll noch nicht Bestandteil dieses Kapitels sein: Ausführliche Informationen und Erklärungen über das Thema Threading finden Sie in Kapitel 29.

Fürs Erste lösen wir das Problem, indem wir gezielt die Windows-Nachrichtenschleife triggern, ihr also die Möglichkeit geben, aufgelaufene Nachrichten »mal eben« zu verarbeiten, obwohl wir ansonsten die gesamte Rechenleistung des UI-Threads blockieren. Nehmen Sie folgende Änderung vor:

```
                …
            picViewer.ShowPicture(mySmallWbm, "Dauer der Konvertierung in ms: " &
                        sw.ElapsedMilliseconds.ToString("#,##0.00"))

            'Böser Workaround - mehr dazu im Threading-Kapitel:
            System.Windows.Forms.Application.DoEvents()
                …
        Next

        picViewer.Close()

    End Sub
```

Wie beim Speichern eines Bildes gibt es dafür auch bei WPF leider keine explizite Funktionalität, wohl auch, da das externe Triggern der Nachrichtenwarteschleife eher unter »Improvisation« und »Bitte, liebe Kinder, bitte nicht zuhause nachmachen« fällt. Wir leihen uns an dieser Stelle wieder eine Funktion aus der Windows Forms-Library, nämlich die `DoEvents`-Methode der `Application`-Klasse. Die Threading-Variante in Kapitel 29 zeigt dann später, wie es wirklich geht.

Klaus Löffelmann, Sarika Calla Purohit: Microsoft Visual Basic 2010 - Das Entwicklerbuch, Microsoft Press 2010 (ISBN 978-3-86645-535-1)