Kapitel 1
Einführung in die WPF

Lehnen Sie sich zurück. In diesem Kapitel werden Sie »gebootet«. Nach einem Blick auf die WPF im .NET Framework und einem Schnelldurchlauf durch die Windows-Programmiergeschichte erfahren Sie mehr über die Architektur und Konzepte der WPF.

Die Windows Presentation Foundation (WPF) ist Teil des .NET Frameworks seit der Version 3.0 und Teil von .NET Core seit Version 3.0. Wie sich die WPF in die Struktur von .NET Framework 4.8 und .NET Core 3.0 eingliedert und welche Stärken und Eigenschaften die WPF hat, lesen Sie in Abschnitt 1.1.

In Abschnitt 1.2 erfahren Sie in einem Schnelldurchlauf der letzten Jahrzehnte, wie sich die Programmierung unter Windows entwickelt hat. Wir beginnen bei Windows 1.0 und enden bei der WPF und Windows 10.

Die WPF nutzt zum Zeichnen DirectX. Wie die dafür benötigte Architektur aussieht und funktioniert, lesen Sie in Abschnitt 1.3. Sie lernen dabei auch die zentralen Assemblies der WPF kennen, die Sie später auch in all Ihren WPF-Projekten wiederfinden.

In Abschnitt 1.4, dem letzten Abschnitt dieses Kapitels, finden Sie einen Überblick über die zentralen Konzepte der WPF. Dazu gehört die XML-basierte Beschreibungssprache XAML, mit der Sie in der WPF Ihre Benutzeroberflächen deklarieren. Ebenfalls lernen Sie in diesem Abschnitt die Konzepte wie Dependency Properties, Routed Events oder Styles und Templates näher kennen. Diese Konzepte tauchen in den folgenden Kapiteln immer wieder auf.

1.1 Die WPF und .NET

Mit der *Windows Presentation Foundation* (WPF) steht seit der Einführung des .NET Frameworks 3.0 im Jahr 2006 ein mächtiges Programmiermodell zum Entwickeln von Benutzeroberflächen zur Verfügung.

1.1.1 .NET Core 3.0 und .NET Framework 4.8

Die WPF wurde mit dem .NET Framework 3.0 eingeführt. Mit dem aktuellen .NET Framework 4.8 hat Microsoft angekündigt, dass dies die letzte Hauptversion des .NET Frameworks sein wird. Zukünftige Entwicklungen und Innovationen finden auf .NET Core statt.

.NET Core ist Microsofts moderne Cross-Platform-Implementierung von .NET, die auf Linux, macOs und Windows läuft. Mit .NET Core 3.0 werden auch mit der WPF entwickelte Desktop-Anwendungen unterstützt. Doch Moment, heißt das, dass Ihre WPF-Anwendung auch unter Linux und macOS läuft?

Nein, dies ist nicht der Fall. Abbildung 1.1 zeigt die Übersicht von .NET Core 3.0. Es gibt dort einen Teil, der cross-platform läuft. Zu diesem Teil gehören *ASP.NET Core* zum Entwickeln von Web-Anwendungen, *Entity Framework* und *Entity Framework Core* zum Zugriff auf Daten und *ML.NET* zum Nutzen von Machine-Learning-Methoden in .NET-Applikationen. Die WPF und Windows Forms (*WinForms*) werden als zusätzliche Bibliotheken unterstützt, die nur unter Windows verfügbar sind. Der Grund dafür ist, dass die WPF und WinForms viele Abhängigkeiten zu Windows haben. Somit lässt sich die WPF nur nutzen, um Desktop-Applikationen für Windows zu entwickeln. Desktop-Applikationen für Linux und macOs lassen sich mit der WPF und .NET Core 3.0 nicht entwickeln.

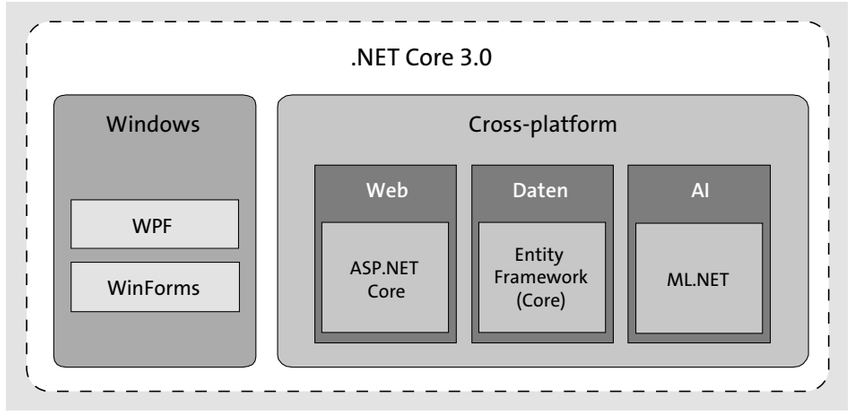

Abbildung 1.1 .NET Core 3.0 und WPF

Jetzt stellt sich die Frage, welches Framework Sie zum Entwickeln Ihrer WPF-Anwendung einsetzen sollten: .NET Framework 4.8 oder .NET Core 3.0? Visual Studio 2019 bietet Ihnen Projektvorlagen für beide Plattformen. Doch die Empfehlung ist klar.

Wenn Sie eine neue WPF-Applikation entwickeln, sollten Sie definitiv .NET Core 3.0 wählen. .NET Core 3.0 bietet Ihnen im Gegensatz zum .NET Framework 4.8 verschiedene Vorteile. Beispielsweise lässt sich Ihre WPF-Applikation gebündelt zusammen .NET Core ausliefern, wodurch auf dem Zielrechner kein .NET installiert sein muss. Ebenso werden viele der neueren C#-Features ab C# 8.0 nur unter .NET Core 3.0 unterstützt, nicht unter .NET Framework 4.8. Der Grund ist, dass einige Sprach-Features Erweiterungen an der Common Language Runtime (CLR) voraussetzen, und diese Erweiterungen würden im .NET Framework zu einigen Kompatibilitätsproblemen führen. Die Erweiterungen sind somit nur in .NET Core verfügbar.

.NET Core ist die Zukunft von .NET. Microsoft wird das .NET Framework in Zukunft weiter unterstützen, aber die Innovation findet in .NET Core statt. Falls Sie bestehende WPF-Anwendungen mit dem .NET Framework entwickelt haben und Sie keine großen Änderungen planen, können Sie diese auf .NET Framework lassen. Falls Sie jedoch weitere Features planen, empfiehlt sich eine Migration von .NET Framework zu .NET Core. Der Code bleibt dabei gleich. Lediglich die Projektdateien (*.csproj*) müssen angepasst werden. Am einfachsten können Sie den Unterschied der Projektdateien sehen, indem Sie neue WPF-Projekte in Visual Studio 2019 sowohl mit .NET Framework als auch mit .NET Core erstellen und dann die Projektdateien entsprechend vergleichen.

In diesem Buch wird .NET Core 3.0 zum Entwickeln der WPF-Anwendungen verwendet. Den Code, den Sie dabei lernen, können Sie natürlich auch in einer auf .NET Framework basierenden WPF-Anwendung nutzen. Die Konzepte sind dieselben, nur der genutzte Unterbau ist ein anderer.

1.1.2 Die WPF und Visual Studio 2019

Für die WPF bietet Visual Studio 2019 verschiedene Projektvorlagen und einen intelligenten Oberflächen-Designer. Neben Visual Studio 2019 gibt es weitere Werkzeuge, die Sie beim Entwickeln Ihrer Anwendungen unterstützen. Besonders zu erwähnen ist das Design-Tool *Blend*, das zusammen mit Visual Studio 2019 auf Ihrem Rechner installiert wird. Sie können Blend zum Erstellen reichhaltiger Benutzeroberflächen mit der WPF einsetzen. Blend legt dabei ein spezielles Augenmerk auf das Design der Anwendung.

Natürlich verwenden Sie zum Programmieren von WPF-Anwendungen nach wie vor Visual Studio. Die zusätzlichen Programme wie Blend bieten Ihnen allerdings beim Benutzeroberflächen-Design etwas mehr Unterstützung als Visual Studio. Sie finden in Blend ähnliche Funktionen und Werkzeuge wie in einem Grafikprogramm, beispielsweise Farbpaletten, Timelines für Animationen, Pens, Pencils und vieles mehr. Damit kann ein Grafikdesigner arbeiten, ohne den vom Programm generierten Code zwingend kennen und verstehen zu müssen.

1.1.3 Die WPF als zukünftiges Programmiermodell

Bei manchen Programmierern, die die technische Entwicklung im Hause Microsoft nicht mitverfolgt haben, sorgte die Nachricht von einem weiteren Programmiermodell für Benutzeroberflächen als Teil des .NET Frameworks 3.0 zuerst für etwas Verwirrung. Schließlich enthielt das .NET Framework seit Version 1.0 mit Windows Forms ein bewährtes Programmiermodell zur Entwicklung von Desktop-Anwendungen. Insbesondere im .NET Framework 2.0 wurde Windows Forms stark verbessert und erfreute sich großer Beliebtheit. Viele Entwickler stellten sich demzufolge die Frage, was das neue Programmiermodell für Vorteile bringen würde und warum sie in Zukunft die WPF anstelle von Windows Forms einsetzen sollten.

Wer damals bereits erste Gehversuche mit den von Microsoft als Download bereitgestellten Vorabversionen des .NET Frameworks 3.0 – den sogenannten Community Technology Previews (CTPs) – hinter sich hatte, der wusste, dass mit der WPF auch scheinbar komplexe Aufgaben (wie die Programmierung von animierten, rotierenden 3D-Objekten mit Videos auf der Oberfläche) relativ einfach und schnell umsetzbar sind (siehe Abbildung 1.2, welche aus Kapitel 14, »3D-Grafik«, stammt).

Abbildung 1.2 Ein 3D-Würfel, auf dem ein Video abläuft

Allerdings trat beim Versuch, eine etwas umfangreichere Anwendung mit der WPF zu entwickeln, meist die erste Ernüchterung auf. Die Einstiegshürde ist bei der WPF relativ mächtig, das heißt, sie ist um einiges höher als jene bei Windows Forms. Die ersten Erfolgserlebnisse mit der WPF werden Sie zwar schon bald haben, für die professionelle und erfolgreiche Entwicklung müssen Sie jedoch auch die Konzepte und Hintergründe der WPF verstanden haben. Dazu gehören unter anderem Layout, Dependency Properties, Routed Events sowie Styles und Templates. Haben Sie diesen Punkt erreicht, können Sie mit der WPF auch sehr komplexe Anwendungen erstellen, ohne auf komplizierte weitere Technologien zugreifen zu müssen.

In diesem Buch werde ich, um die Konzepte zu erläutern, auch einige Beispiele aus der Anwendung *FriendStorage* herauspicken. FriendStorage ist eine kleine Anwendung, die Sie beim Studieren der WPF-Konzepte unterstützt. In FriendStorage können Sie eine Liste mit Freunden speichern und für jeden Freund verschiedene persönliche Daten und ein Bild erfassen (siehe Abbildung 1.3). Die Anwendung zeigt auf der rechten Seite eine Liste Ihrer Freunde an. Auf der linken Seite sehen Sie die Details zum aktuell in der Liste ausgewählten Freund.

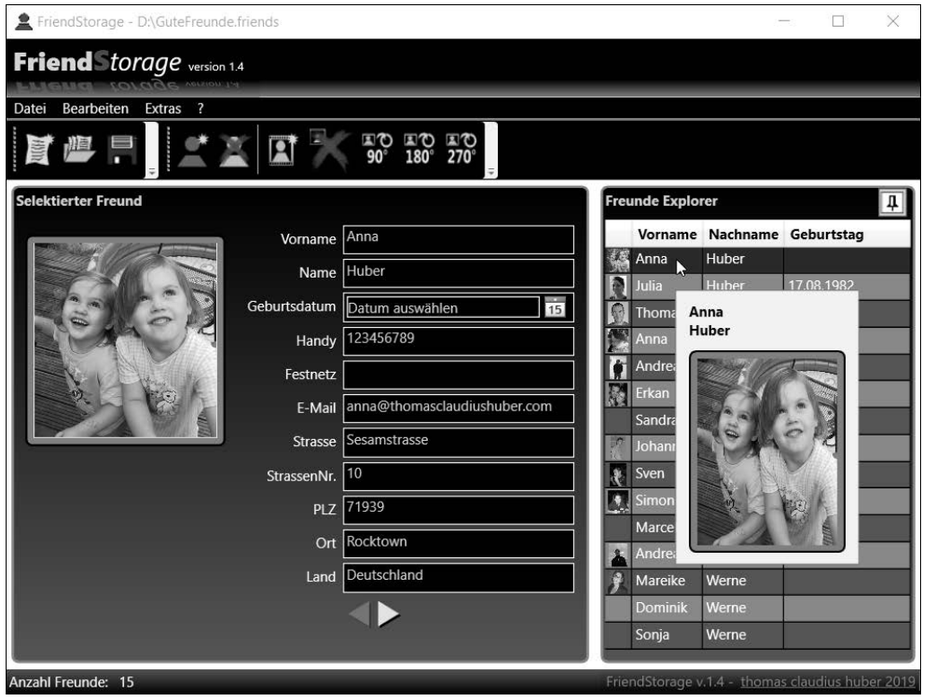

Abbildung 1.3 Die Anwendung »FriendStorage« speichert Listen mit Freunden. Sie verwendet verschiedene Features der WPF wie Commands, Styles, Trigger, Animationen und Data Binding.

Bisher konnten grafisch hochwertige Anwendungen nur mit weiteren einarbeitungsintensiven Technologien wie DirectX erstellt werden. Mit der WPF stellt Microsoft im Zeitalter von Windows 10 ein einheitliches, zeitgemäßes Programmiermodell zur Verfügung, das zur Entwicklung moderner Desktop-Anwendungen keine Kenntnisse über komplexe, weitere Technologien wie eben DirectX erfordert. Und das Besondere an dem einheitlichen Programmiermodell der WPF ist, dass Sie für die Verwendung von Animationen, Data Bindings oder Styles immer die gleichen Konstrukte verwenden, egal ob Sie damit 2D-, 3D-, Text- oder sonstige Inhalte beeinflussen wollen. Haben Sie also einmal gelernt, wie Sie ein 2D-Element animieren, können Sie das Erlernte auch auf 3D-Objekte anwenden.

1.1.4 Stärken und Eigenschaften der WPF

Mit 2D, 3D und Text beziehungsweise Dokumenten wurden schon einige der Stärken der WPF angeschnitten. Im Gegensatz zu Windows Forms bietet die WPF viele weitere, vorteilhafte Eigenschaften, die sich nicht nur auf die Erstellung reichhaltiger Benutzeroberflächen aus Sicht des Grafikers beziehen. Unter anderem sind dies erweitertes Data Binding, verbes-

serte Layoutmöglichkeiten, ein flexibles Inhaltsmodell, eine verbesserte Unterstützung für Audio/Video, integrierte Animationen, Styles, Templates und vieles mehr. In Tabelle 1.1 finden Sie eine Handvoll der wohl bedeutendsten Eigenschaften der WPF.

Eigenschaft	Beschreibung
Flexibles Inhaltsmodell	Die WPF besitzt ein flexibles Inhaltsmodell. In bisherigen Programmiermodellen, wie Windows Forms, konnte beispielsweise ein Button lediglich Text oder ein Bild als Inhalt enthalten. Mit dem flexiblen Inhaltsmodell der WPF kann ein Button – genau wie viele andere visuelle Elemente – einen beliebigen Inhalt haben. So ist es beispielsweise möglich, in einen Button ein Layout-Panel zu setzen und darin wiederum mehrere 3D-Objekte und Videos unterzubringen. Ihrer Kreativität sind keine Grenzen gesetzt.
Layout	Die WPF stellt einige Layout-Panels zur Verfügung, um Controls in einer Anwendung dynamisch anzuordnen und zu positionieren. Aufgrund des flexiblen Inhaltsmodells lassen sich die Layout-Panels der WPF auch beliebig ineinander verschachteln, wodurch Sie in Ihrer Anwendung auch ein sehr komplexes Layout erstellen können.
Styles	Ein Style ist eine Sammlung von Eigenschaftswerten. Diese Sammlung lässt sich einem oder mehreren Elementen der Benutzeroberfläche zuweisen, wodurch deren Eigenschaften dann die im Style definierten Werte annehmen. Sie definieren einen Style üblicherweise als Ressource, um ihn beispielsweise mehreren Buttons zuzuweisen, die alle die gleiche Hintergrundfarbe und die gleiche Breite haben sollen. Ohne Styles müssten Sie auf jedem Button diese Properties setzen. Mit Styles setzen Sie lediglich die Style-Property auf den Buttons, was sogar implizit passieren kann; mehr dazu folgt in Kapitel 11, »Styles, Trigger und Templates«.
Trigger	Trigger erlauben es Ihnen, auf deklarativem Weg festzulegen, wie ein Control auf bestimmte Eigenschaftsänderungen oder Events reagieren soll. Mit Triggern können Sie bereits deklarativ Dinge erreichen, für die Sie ansonsten einen Event Handler benötigen würden. Trigger definieren Sie meist in einem Style oder einem Template.
»lookless« Controls	Custom Controls sind bei der WPF »lookless«, das heißt, Sie trennen ihre visuelle Erscheinung von ihrer eigentlichen Logik und ihrem Verhalten. Das Aussehen eines Controls wird dabei mit einem `ControlTemplate` beschrieben. Das Control selbst definiert kein Aussehen, sondern nur Logik und Verhalten – daher »lookless«. Durch Ersetzen des `ControlTemplates` (durch Setzen der `Template`-Property der Klasse `Control`) lässt sich das komplette Aussehen eines Controls anpassen.

Tabelle 1.1 Wichtige Eigenschaften der WPF

Eigenschaft	Beschreibung
»lookless« Controls (Forts.)	Aufgrund dieser Flexibilität und der Tatsache, dass die meisten Controls der WPF als Custom Control implementiert sind, müssen Sie keine Subklassen mehr erstellen, um lediglich das Aussehen anzupassen. Es sind folglich weniger eigene Custom Controls als in bisherigen Programmiermodellen notwendig.
Daten	Die Elemente in Ihrer Applikation können Sie mit Data Binding an verschiedene Datenquellen binden. Dadurch ersparen Sie sich die Programmierung von Event Handlern, die die Benutzeroberfläche oder die Datenquelle bei einer Änderung aktualisieren. Außer durch Data Binding können Sie mit Data Templates das Aussehen Ihrer Daten auf der Benutzeroberfläche definieren.
2D- und 3D-Grafiken	3D-Grafiken können Sie in der WPF auf dieselbe Weise zeichnen und animieren wie 2D-Grafiken. Dazu stellt die WPF viele Zeichenwerkzeuge bereit, wie beispielsweise die verschiedenen Brushes. Auch eine Benutzerinteraktion mit 3D-Elementen ist möglich.
Animationen	Die WPF besitzt einen integrierten Mechanismus für Animationen. Während Sie bisher für Animationen einen Timer und einen dazugehörigen Event Handler verwendet haben, ist dies jetzt wesentlich einfacher realisiert, wie Sie in Kapitel 15, »Animationen«, sehen werden.
Audio/Video	Audio- und Video-Elemente lassen sich einfach in Ihre Applikation einbinden. Dafür stehen verschiedene Klassen zur Verfügung.
Text & Dokumente	Die WPF stellt eine umfangreiche API zum Umgang mit Text und Dokumenten bereit. Es werden fixe und fließende Dokumente unterstützt. *Fixe Dokumente* unterstützen eine gleichbleibende, fixierte Darstellung des Inhalts – ähnlich wie PDF-Dokumente –, während *fließende Dokumente* ihren Inhalt an verschiedene Faktoren anpassen, wie beispielsweise an die Größe des Fensters. Zum Anzeigen der Dokumente stehen verschiedene Controls zur Verfügung.

Tabelle 1.1 Wichtige Eigenschaften der WPF (Forts.)

Neben all den in Tabelle 1.1 dargestellten Eigenschaften, die Sie in einzelnen Kapiteln dieses Buches wiederfinden, ist eine weitere große Stärke der WPF die verbesserte Unterstützung des Entwicklungsprozesses zwischen dem Designer einer Benutzeroberfläche und dem Entwickler, der die eigentliche Geschäftslogik implementiert. Dies wird durch die XML-basierte Beschreibungssprache *Extensible Application Markup Language* (XAML, sprich »Semmel«) erreicht, die mit der WPF eingeführt wurde. XAML wird in der WPF zur deklarativen Beschrei-

bung von Benutzeroberflächen eingesetzt. Sie können Ihre Benutzeroberflächen zwar auch weiterhin in C# erstellen, profitieren dann aber nicht von den Vorteilen, die Ihnen XAML bietet. So dient XAML beispielsweise im Entwicklungsprozess als Austauschformat zwischen Designer und Entwickler. Es gibt mittlerweile zig Programme, die XAML-Dateien öffnen können und zum Editieren einen grafischen Editor bereitstellen, wodurch ein Designer Ihre Benutzeroberfläche wie eine Grafik »designen« kann. Sie werden XAML in den Kapiteln dieses Buchs mit allen Tricks und Kniffen kennenlernen. Insbesondere in Kapitel 3, »XAML«, dreht sich alles um diese Markup-Sprache.

1.1.5 Auf Wiedersehen, GDI+

Außer durch ihr Modell, das vorsieht, Benutzeroberflächen mit XAML zu erstellen, unterscheidet sich die WPF auch aus rein technologischer Sicht von Windows Forms und den bisherigen Programmiermodellen unter Windows. Erstmals baut die grafische Darstellung nicht auf der GDI-Komponente (*Graphics Device Interface*) der Windows-API auf, wie das bei Windows Forms und vorherigen Programmiermodellen der Fall war (Windows Forms verwendet GDI+, eine verbesserte Variante von GDI). Stattdessen greift die WPF zur Darstellung des Fensterinhalts auf DirectX zurück.

Ja, Sie haben richtig gelesen. Für das Zeichnen der Pixel (Rendering) macht die WPF von dem bisher meist nur in Spielen verwendeten DirectX Gebrauch. *DirectX* ist eine aus mehreren APIs bestehende Suite, die auf Windows-Rechnern die Kommunikation zwischen Hardware und Software ermöglicht. Dadurch lassen sich die Möglichkeiten der in heutigen Computern mittlerweile standardmäßig eingebauten Grafikkarten mit 3D-Beschleunigern richtig ausnutzen. Bisher, so auch unter Windows Forms, blieben die meisten Möglichkeiten der vorhandenen Hardware völlig ungenutzt.

Den guten alten Grafikschnittstellen GDI und GDI+ kehrt die WPF also den Rücken zu. Verständlich, denn DirectX ist natürlich weitaus attraktiver und leistungsfähiger. Zu Ehren der klassischen Windows-API und der darin enthaltenen GDI-Komponente wagen wir einen ganz kurzen Rückblick ins Jahr 1985, als alles begann.

1.2 Von Windows 1.0 zur Windows Presentation Foundation

Als im November 1985 die erste Version von Windows auf den Markt kam – eine grafische Erweiterung zum damaligen Betriebssystem MS-DOS –, gab es nur eine Möglichkeit, Windows-Anwendungen zu schreiben: Mit der Programmiersprache C wurde auf die Funktionen der ebenfalls in C geschriebenen Windows-Programmierschnittstelle – kurz Windows-API – zugegriffen, um Fenster zu erstellen und um weitere Systemfunktionalität zu verwenden. Mit der Umstellung auf eine 32-Bit-Architektur wurden die Bibliotheken der Windows-API angepasst und erweitert. Sie tragen seitdem die Namen *gdi32.dll*, *kernel32.dll* und *user32.dll*. Man spricht in diesem Zusammenhang statt von der »Windows-API« auch von der »Win32-API«.

1.2.1 Die ersten Wrapper um die Windows-API

Da bei der direkten Verwendung der Windows-API viele Funktionsaufrufe auf sehr niedrigem, detailliertem Betriebssystem-Level notwendig waren, was zu Unmengen von Code führte, fiel bei der zeitintensiven Programmierung der Blick auf das Wesentliche sehr schwer, weil man so viele Details programmieren musste. Es war nur eine Frage der Zeit, bis die ersten Programmbibliotheken entstanden, die die Aufrufe der Windows-API kapselten und diese Aufrufe zu logischen, abstrakteren Einheiten zusammenfassten.

Für C++ entwickelte Microsoft die *Microsoft Foundation Classes* (*MFC*) als objektorientierte »Wrapper«-Bibliothek um die Windows-API. Borland brachte mit der *Object Window Library* (*OWL*) ein Konkurrenzprodukt auf den Markt. Auch im Zeitalter von .NET werden von Windows Forms die Funktionen der Windows-API gekapselt. Man könnte also sagen, dass die Programmierung von Windows-Applikationen seit der Einführung von Windows 1.0 im Jahr 1985 in den Grundzügen gleich geblieben ist – im Hintergrund wurde seit eh und je auf die Programmbibliotheken der Windows-API zugegriffen.

1.2.2 Windows Forms und GDI+

Die Aufrufe der zur Windows-API gehörenden Programmbibliothek GDI+ kapselt Windows Forms hauptsächlich in der Klasse `System.Drawing.Graphics`. Jedes Control in Windows Forms nutzt ein `Graphics`-Objekt zum Zugriff auf GDI+. In Abbildung 1.4 ist zu sehen, dass GDI+ die entsprechenden Befehle an die Grafikkarte weitergibt, um das Control auf den Bildschirm zu zeichnen. (Das Zeichnen auf den Bildschirm nennt man *Rendering*.)

Ein einzelnes Control von Windows Forms und Win32 wird aus Windows-Sicht als ein Fenster betrachtet. Jedes Fenster wird über einen Window-Handle (`HWND`-Datentyp in C/C++, `System.IntPtr` in .NET) referenziert und besitzt einen bestimmten Bereich auf dem Bildschirm, auf den es zeichnen darf. Auf die Bereiche eines anderen Window-Handles darf das Fenster nicht zeichnen.

Die WPF schlägt einen neuen, zeitgemäßen Weg ein und lässt die »Altlasten« vergangener Jahrzehnte hinter sich. Ein Control hat bei der WPF keinen Window-Handle mehr;[1] es kann somit auch auf die Pixel anderer Controls zeichnen, wodurch beispielsweise Transparenzeffekte möglich sind.

1 Es gibt einige Ausnahmen, wie beispielsweise ein Window, das in einen Top-Level-Handle gesetzt wird. Auch ein Kontextmenü wird bei der WPF in einen Window-Handle gesetzt, damit es immer im Vordergrund ist.

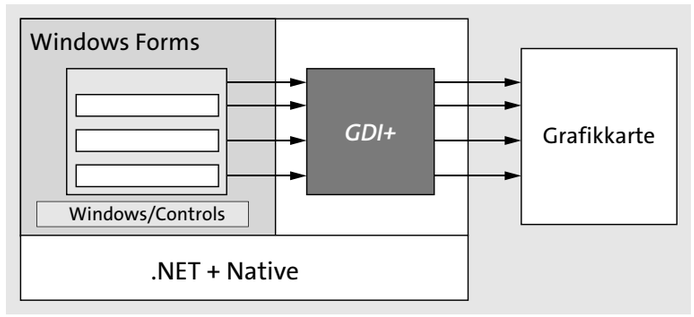

Abbildung 1.4 In Windows Forms werden Controls mit GDI+ gezeichnet.

1.2.3 Die Windows Presentation Foundation

Mit der Entwicklung der WPF begann Microsoft bereits, bevor im Jahr 2001 die erste Version des .NET Frameworks erschien. Damals war den Entwicklern und Entscheidern bei Microsoft bereits klar, dass .NET die Zukunft sein würde. Somit entschieden sie sich, auch die WPF in Managed Code statt in nativem Code zu implementieren.

Während die bisherigen Programmiermodelle von Microsoft für Benutzeroberflächen meist nur dünne Wrapper um die Windows-API darstellten (wie eben auch Windows Forms), ist die WPF das erste, umfangreiche Programmiermodell für Benutzeroberflächen, das fast vollständig in .NET geschrieben ist.

Eines der Designziele der WPF war es, nicht auf den vielen in die Jahre gekommenen Funktionen der Windows-API aufzubauen. So setzt die WPF beispielsweise für die Darstellung DirectX anstelle von GDI+ ein, um die Leistung der heutigen Grafikkarten nicht nur in Spielen, sondern auch in »gewöhnlichen« Windows-Anwendungen voll und ganz auszureizen. Dabei werden die Komponenten einer WPF-Anwendung nicht mehr durch das Betriebssystem, sondern durch die WPF selbst unter Verwendung von DirectX gezeichnet. Ein einzelnes Control der WPF besitzt nicht wie ein Window-Handle unter Win32 seinen Bereich, in dem es zeichnen darf. Somit kann ein Control der WPF wie gesagt über die Pixel eines anderen Controls zeichnen, was Transparenzeffekte ermöglicht.

Durch die Tatsache, dass die WPF alles selbst zeichnet, sind das flexible Inhaltsmodell oder Dinge wie Templates – mit denen Sie das Erscheinungsbild von Controls individuell an Ihre Bedürfnisse anpassen können – überhaupt erst möglich.

Nach der Einführung des .NET Frameworks 3.0 war es noch so, dass Windows Forms mit Controls wie der `DataGridView` Komponenten besaß, die man in der WPF vergeblich suchte. Auch der Windows-Forms-Designer in Visual Studio 2005 war dem damals zur Verfügung stehenden WPF-Designer weit voraus. Somit lautete der Grundsatz zu dieser Zeit, Anwendungen, die ohne verschiedene Medienelemente und ohne grafische Kunststücke auskamen, weiterhin mit Windows Forms zu entwickeln.

Mit dem .NET Framework 4.0 erhielt die WPF ein `DataGrid`, und es gab bereits genügend Controls aus der dritten Reihe von altbekannten Herstellern wie beispielsweise Infragistics, DevExpress oder Telerik. Mit dem aktuellen .NET Framework 4.8 und .NET Core 3.0 wurde insbesondere der WPF-Designer in Visual Studio 2019 verbessert. Auch das Schreiben von XAML-Code ist jetzt noch einfacher, da die IntelliSense-Unterstützung deutlich besser ist. Mit Visual Studio 2019 wird auch das Design-Tool Blend installiert.

Hinweis

Im Gegensatz zu Visual Studio erlaubt *Blend* das Definieren von Animationen über eine rein grafische Benutzeroberfläche, die keinerlei XAML-Kenntnisse voraussetzt. Es gibt in Blend ein Timeline-Fenster ähnlich wie das aus Adobe Flash, in dem sich eine Animation über einen Zeitraum mit einfachen Mausklicks definieren lässt.

Neben der grafischen Unterstützung von Animationen besitzt Blend viele designerfreundliche Merkmale, wie Farbpaletten, eine Werkzeugleiste mit aus Grafikprogrammen bekannten Werkzeugen wie Stift, Pinsel, Radiergummi etc.

Blend wird allerdings auch lediglich als Programm für den Feinschliff der Benutzeroberfläche betrachtet und wird Visual Studio somit keinesfalls ersetzen. In diesem Buch wird mit der WPF und XAML programmiert und folglich außer Visual Studio kein anderes Programm genutzt. Wenn Sie später Blend verwenden, werden Sie nach der Lektüre dieses Buchs den von Blend und auch von anderen Programmen generierten XAML-Code verstehen.

Die damaligen Gründe für Windows Forms – mehr Controls und bessere Designunterstützung in Visual Studio 2005 – sind aus meiner Sicht heute nicht mehr gegeben. Auch für typische datenintensive Geschäftsanwendungen ist die WPF bestens geeignet, da sie mit Features wie dem umfangreichen Data Binding optimale Unterstützung bietet.

Haben Sie sich für die WPF entschieden, stehen Ihnen für Ihre bereits entwickelten Win32- und Windows-Forms-Anwendungen verschiedene Interoperabilitätsmöglichkeiten mit der WPF zur Verfügung. In Visual Studio gibt es zwar keinen integrierten Migrationsmechanismus von Windows Forms/Win32 zur WPF – die Programmiermodelle sind einfach zu unterschiedlich –, doch die in .NET enthaltenen Klassen für Interoperabilität zwischen WPF und Windows Forms/Win32 bieten Ihnen verschiedene Möglichkeiten, Ihre älteren Anwendungen nach und nach zu migrieren. In Kapitel 20, »Interoperabilität«, erfahren Sie mehr über verschiedene Interoperabilitätsszenarien und mögliche Migrationsstrategien.

1.3 Die Architektur der WPF

Nachdem Sie jetzt bereits einige Eigenschaften und Hintergründe der WPF kennen, ist es an der Zeit, einen Blick auf die Architektur der WPF zu werfen. Der Kern der WPF besteht aus drei Bibliotheken:

- *MilCore.dll*
- *PresentationCore.dll*
- *PresentationFramework.dll*

Obwohl Microsoft sich entschieden hat, die WPF in .NET statt in nativem Code zu implementieren, setzt die WPF aus Performance-Gründen auf einer in nativem Code geschriebenen Schicht namens *Media Integration Layer* (*MIL*) auf. Die Kernkomponente des Media Integration Layers ist die *MilCore.dll*.

Wie Abbildung 1.5 zeigt, ist MilCore unter anderem für die Darstellung von 2D- und 3D-Inhalten, Bildern, Videos, Text und Animationen verantwortlich. Zur Darstellung der Informationen auf dem Bildschirm greift MilCore auf die Funktionalität von DirectX zu, um die Leistung der Grafikhardware voll auszunutzen.

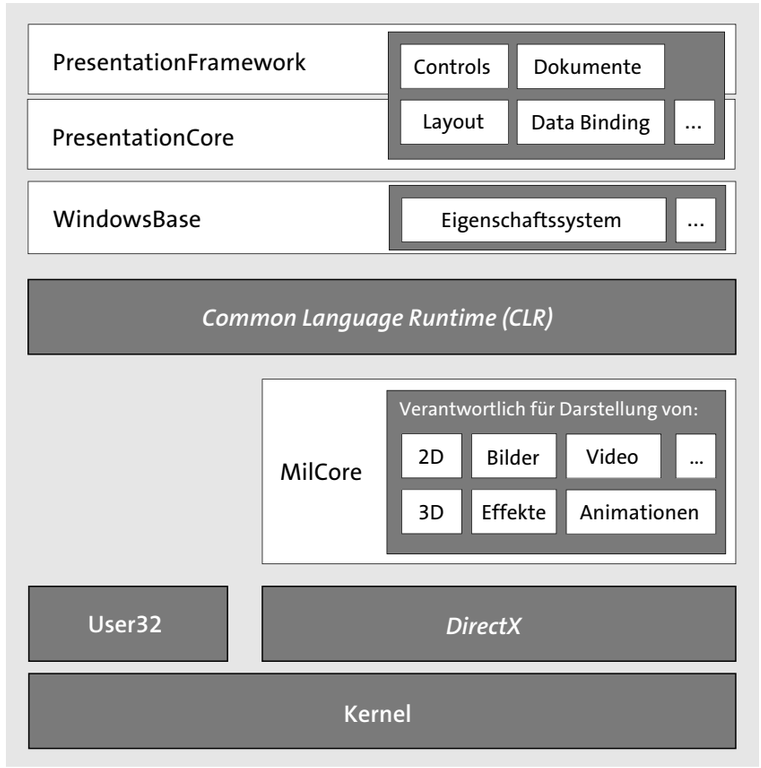

Abbildung 1.5 Die WPF-Architektur mit den Hauptkomponenten »PresentationFramework«, »PresentationCore« und »MilCore«

Beim Entwickeln einer WPF-Anwendung werden Sie mit MilCore nicht direkt in Kontakt kommen. Die Programmierschnittstelle, die zur Programmierung von WPF-Anwendungen genutzt wird, liegt komplett in Managed Code vor. Die Assemblies *PresentationCore.dll* und *PresentationFramework.dll* bilden dabei die zentralen Bausteine der WPF. Aufgrund ihrer Implementierung in Managed Code sind sie in Abbildung 1.5 oberhalb der Laufzeitumgebung von .NET – der Common Language Runtime (CLR) – positioniert. PresentationCore und PresentationFramework enthalten Logik für Controls, Layout, Dokumente oder Data Binding. Darüber hinaus kapselt PresentationCore die Aufrufe der nativen MilCore-Komponente. Beide Komponenten bauen auf der Assembly *WindowsBase.dll* auf, die nicht Teil der WPF ist, sondern Klassen für alle mit dem .NET Framework 3.0 neu eingeführten Programmiermodelle enthält. So finden Sie in WindowsBase beispielsweise Klassen für das erweiterte Eigenschaftssystem mit Dependency Properties, das bei der WPF und auch bei der Windows Workflow Foundation (WF) verwendet wird.

Sehen wir uns genauer an, was die drei Komponenten der WPF (MilCore, PresentationCore und PresentationFramework) und auch die WindowsBase-Komponente enthalten, was ihre Aufgaben sind und – was wohl am spannendsten ist – wie sie diese Aufgaben meistern.

1.3.1 MilCore – die »Display Engine«

Die in nativem Code geschriebene Komponente *MilCore.dll* kapselt DirectX. MilCore ist in der WPF für die Darstellung von 3D, 2D, Text, Video, Bildern, Effekten und Animationen zuständig. Prinzipiell alles, was in einer WPF-Anwendung gezeichnet wird, basiert auf der Funktionalität von MilCore und DirectX.

Ein wohl entscheidender Vorteil, den Ihre WPF-Anwendung durch MilCore erreicht, ist die vektorbasierte Darstellung. MilCore stellt alle Inhalte vektorbasiert dar; dadurch können Sie Ihre Anwendungen beliebig skalieren, ohne dass diese an »Schärfe« verlieren.

Im Folgenden werfen wir einen Blick darauf, wie MilCore die Aufgabe wahrnimmt, die einzelnen Elemente Ihrer Anwendung auf dem Bildschirm darzustellen.

Den darzustellenden Bildschirminhalt verwaltet MilCore in Form einer Baumstruktur, als sogenannten *Composition Tree* (siehe Abbildung 1.6). Dieser Baum besteht aus einzelnen Knoten (*Composition Nodes*), die Metadaten und Zeichnungsinformationen enthalten. Bei Änderungen am Composition Tree generiert MilCore die entsprechenden DirectX-Befehle, die die Änderungen visuell umsetzen und mithilfe der Grafikkarte auf dem Bildschirm darstellen. Die vielen Composition Nodes werden also durch MilCore zu einem großen, zu zeichnenden Bild zusammengesetzt; dieser Prozess wird auch als *Composition* bezeichnet. Das zusammengesetzte Bild wird dann auf dem Bildschirm dargestellt.

In früheren Modellen, wie Windows Forms, hatte jedes Control seinen eigenen Ausschnitt, in dem es sich selbst zeichnen durfte. Der Ausschnitt war über ein Window-Handle (HWND) definiert. Über den Ausschnitt kam das Control nicht hinaus. Dieses System wird als Clipping-

System bezeichnet, da in ihm einfach am Rand abgeschnitten beziehungsweise geclippt wird und jedes Control seinen eigenen Ausschnitt hat, auf dem es sich zeichnen darf. Bei einem Composition-System hingegen wird nicht abgeschnitten. Stattdessen darf jedes Control überall zeichnen, und am Ende wird alles zu einem großen Bild zusammengefügt. Mit der Composition in MilCore kann ein Control auch über die Pixel eines anderen Controls zeichnen, wodurch Effekte wie Halbtransparenz erst möglich werden.

Der Composition Tree ist mit allen Zeichnungsinformationen zwischengespeichert. Dadurch kann die Benutzeroberfläche sehr effektiv und schnell neu gezeichnet werden, auch dann, wenn Ihre Anwendung gerade beschäftigt ist. Die Zeiten von visuell eingefrorenen Anwendungen sind also vorbei.

Der zur Laufzeit mit den Zeichnungsinformationen bestückte und in der nativen MilCore-Komponente lebende Composition Tree besitzt auf der .NET-Seite ein Pendant, den sogenannten *Visual Tree*. Der Visual Tree setzt sich aus allen visuellen Elementen einer WPF-Anwendung zusammen. Das Prinzip der Entwicklung von WPF-Anwendungen und -Controls besteht im Grunde darin, mit XAML und/oder prozeduralem Code eine Hierarchie von visuellen Elementen zu erzeugen, wie etwa `Window`, `TextBox` und `Button`. Die einzelnen Controls in dieser Hierarchie setzen sich wiederum aus einfacheren visuellen Elementen zusammen, wie `Rectangle`, `TextBlock` oder `Border`. Alle visuellen Elemente haben die gemeinsame Basisklasse `Visual`. Die gesamte Hierarchie, einschließlich der einfacheren visuellen Elemente, wird daher als *Visual Tree* bezeichnet.

Über einen zweiseitigen Kommunikationskanal sind der Visual Tree und der Composition Tree miteinander verbunden, wie in Abbildung 1.6 zu sehen ist. Über diesen Kommunikationskanal werden Änderungen auf die jeweils andere Seite übertragen. Dabei sind die beiden Bäume nicht zu 100 % identisch; zum Beispiel können einem Knoten im Visual Tree mehrere Knoten im Composition Tree entsprechen. Objekte mit hohem Speicherplatzbedarf, wie etwa Bitmaps, werden gemeinschaftlich verwendet. Wie Abbildung 1.6 zeigt, verwendet MilCore DirectX, das die entsprechenden Befehle an die Grafikkarte gibt, wodurch die Darstellung auf dem Bildschirm (also das Rendering) erfolgt.

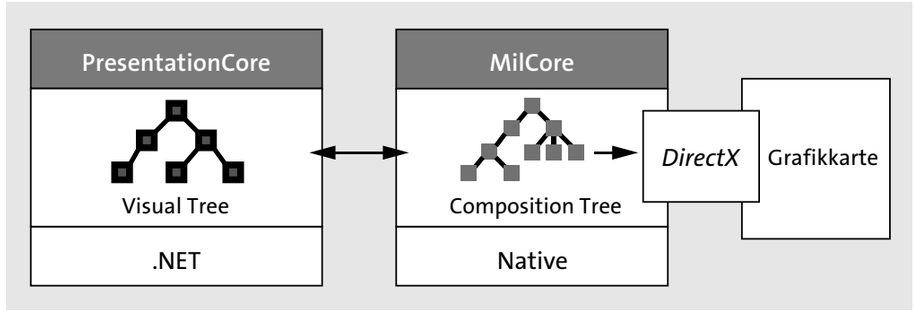

Abbildung 1.6 Die Kommunikation zwischen der .NET-Komponente »PresentationCore« und der auf DirectX aufbauenden Komponente »MilCore«

Hinweis

Tatsächlich verwendet die WPF im Hintergrund einen weiteren Thread, der für das Rendering (also das Zeichnen auf den Bildschirm) verantwortlich ist. Eine WPF-Anwendung besitzt somit immer zwei Threads:

- einen UI-Thread, der Benutzereingaben mit der Maus oder Tastatur entgegennimmt, Events entgegennimmt und Ihren Code ausführt
- einen Render-Thread, der für das Zeichnen der Inhalte verantwortlich ist

Wie bereits erwähnt, werden Sie die Programmbibliothek *MilCore.dll* nicht direkt verwenden, sondern über .NET-Assemblies indirekt auf sie zugreifen. MilCore wird Ihnen daher nicht begegnen, zumal die Bibliothek nicht öffentlich ist. Somit stellen die drei .NET-Assemblies den Teil dar, mit dem Sie bei der Entwicklung von WPF-Anwendungen in Berührung kommen. Im Folgenden sehen wir uns diese drei Assemblies kurz an.

1.3.2 WindowsBase

Die Assembly *WindowsBase.dll* enthält die Basislogik für Windows-Anwendungen und ist in .NET geschrieben. Unter anderem ist in der Assembly WindowsBase die Logik für das erweiterte Eigenschaftssystem implementiert, das aus den sogenannten Dependency Properties besteht. Dependency Properties werden in Kapitel 7, »Dependency Properties«, genauer betrachtet. Weiter enthält WindowsBase Low-Level-Klassen, die notwendig sind, um beispielsweise in Ihrer WPF-Anwendung die Nachrichtenschleife zu starten. Die Assemblies PresentationCore und PresentationFramework bauen beide auf WindowsBase auf.

1.3.3 PresentationCore

In PresentationCore ist auf der .NET-Seite die Verbindung der beiden Baumstrukturen Visual Tree und Composition Tree (auf MilCore-Seite) in der abstrakten Klasse `System.Windows.Media.Visual` implementiert. Der Visual Tree einer WPF-Anwendung besteht aus Objekten von Subklassen der Klasse `Visual`. Die Klasse `Visual` enthält private Methoden zur Kommunikation mit dem auf MilCore-Seite bestehenden Composition Tree. `Visual` dient als Basisklasse für jene Klassen, die in der WPF visuell dargestellt werden sollen.

Neben `Visual` enthält PresentationCore einige weitere interessante Klassen, unter anderem die Klasse `UIElement`, die in der WPF von `Visual` ableitet und eine konkrete Implementierung für visuelle Elemente darstellt. Mehr zu `Visual`, `UIElement` und weiteren Klassen möchte ich Ihnen an dieser Stelle noch nicht verraten; ich werde sie in Kapitel 2, »Das Programmiermodell«, und in den folgenden Kapiteln ausführlicher behandeln.

1.3.4 PresentationFramework

Die wichtigsten Klassen der WPF für die Entwicklung der grafischen Benutzerschnittstelle wie auch der Funktionalität einer Anwendung befinden sich in der Assembly *PresentationFramework.dll*. Zu ihnen zählen Klassen für Controls, Dokumente, Layout-Panels, Benutzerführung und Animationen sowie Klassen zum Einbinden von Videos und Bildern. Oft wird diese Assembly allein als »die« WPF bezeichnet, da die anderen Assemblies, WindowsBase und PresentationCore, nur die Basis für ein UI-Framework bieten. Die WPF ist ein solches UI-Framework, das in der Assembly PresentationFramework implementiert ist.

In Kapitel 2, »Das Programmiermodell«, werden Sie sehen, dass die Assemblies *PresentationCore.dll*, *PresentationFramework.dll* und auch die Assembly *WindowsBase.dll* standardmäßig in Ihrem WPF-Projekt referenziert werden.

1.3.5 Vorteile und Stärken der WPF-Architektur

Durch die Architektur der WPF und der darunterliegenden Komponente MilCore als Wrapper um DirectX ergeben sich viele Vorteile gegenüber Windows Forms und älteren Win32-Technologien. Die wichtigsten fasse ich an dieser Stelle kurz zusammen:

- **Volle Ausnutzung der Hardware**
  Weil sie auf DirectX aufbauen, können WPF-Anwendungen die in heutigen Rechnern bereits standardmäßig vorhandenen leistungsfähigen Grafikkarten voll ausnutzen. Falls die Kraft der Grafikkarte nicht ausreicht, wird von Hardware- auf Software-Rendering umgestellt.
- **Die WPF »zeichnet« selbst**
  In einer WPF-Anwendung werden die visuellen Elemente durch die WPF gezeichnet und nicht durch das Betriebssystem, wie es in Windows Forms und älteren Win32-Technologien der Fall war. Dies erlaubt zum einen verschiedene Effekte, wie die bereits erwähnte Transparenz. Neben solchen Effekten ist zum anderen durch das selbstständige Zeichnen der WPF die Anwendung des flexiblen Inhaltsmodells möglich, wodurch Sie visuelle Elemente beliebig ineinander verschachteln können. Zu guter Letzt können Sie, weil ja die WPF zeichnet, das visuelle Erscheinungsbild von Controls mit Styles und Templates Ihren Wünschen entsprechend anpassen.
- **Zwischengespeicherte Zeichnungsdaten**
  Durch die zwischengespeicherten Zeichnungsdaten eines visuellen Elements ist ein effektives Neuzeichnen möglich. Die Informationen des Visual Trees werden mit dem Composition Tree abgeglichen, den MilCore zur Verfügung stellt. Ist ein Neuzeichnen notwendig, weil vielleicht ein anderes Fenster das Fenster Ihrer Applikation verdeckt hat, können dafür die im Composition Tree bereits vorhandenen zwischengespeicherten Zeichnungsinformationen verwendet werden. Auch Änderungen an einem visuellen Element – und somit an einem Knoten des Visual Trees – können sehr effizient neu gezeichnet werden. Ändern Sie die Border-Linie eines Buttons, wird diese Änderung an MilCore übertragen

und der entsprechende DirectX-Befehl zur Neuzeichnung der Border-Linie erstellt. In Windows Forms wäre ein Neuzeichnen des ganzen Buttons notwendig.

- **Vektorbasierte Grafiken**
  Die Inhalte Ihrer Anwendung werden durch die WPF vektorbasiert gezeichnet. Somit ist Ihre Anwendung beliebig skalierbar und wirkt auch vergrößert nicht pixelig.

Tipp

Ob eine WPF-Anwendung die Hardware voll ausnutzt, hängt von den Eigenschaften der Grafikkarte und von der installierten DirectX-Version ab. Die WPF teilt Rechner aufgrund dieser Umstände in drei Ebenen ein:

- **Ebene 0** – Es ist DirectX kleiner als Version 7.0 installiert. Es findet somit keine Hardwarebeschleunigung statt. Das ganze Rendering (Zeichnen) findet in der Software statt. Das ist nicht sehr leistungsstark.
- **Ebene 1** – eingeschränkte Hardwarebeschleunigung durch die Grafikkarte. DirectX ist mindestens in der Version 7.0 installiert, aber in einer Version kleiner als 9.0.
- **Ebene 2** – Fast alle Grafikfeatures der WPF verwenden die Hardwarebeschleunigung der Grafikkarte. Auf dem Rechner ist mindestens DirectX 9.0 installiert.

In einer WPF-Anwendung können Sie prüfen, auf welcher Ebene Ihre Anwendung läuft. Nutzen Sie dazu die statische `Tier`-Property der Klasse `RenderCapability` (Namespace: `System.Windows.Media`). Die `Tier`-Property ist vom Typ `int`. Allerdings müssen Sie aus diesem `int` das sogenannte High-Word extrahieren, damit Sie die eigentliche Ebene erhalten. Sie extrahieren ein High-Word (auch hWord genannt), indem Sie eine Bit-Verschiebung um 16 Bits durchführen. Folgende Zeile zeigt, wie es geht:

```
int ebene = RenderCapability.Tier >> 16;
```

Auf meinem Rechner enthält der Integer `ebene` den Wert `2`. Meine Grafikkarte ist somit gut, und es ist mindestens DirectX in der Version 9.0 installiert. Demnach findet bei WPF-Anwendungen eine Hardwarebeschleunigung statt.

Neben der Architektur baut die WPF auch auf spannenden Konzepten auf. So können Sie beispielsweise eine Benutzeroberfläche deklarativ mit der XML-basierten Beschreibungssprache XAML erstellen. Einige dieser Konzepte, die uns Programmierern das Leben erleichtern sollen, werden im nächsten Abschnitt betrachtet.

1.4 Konzepte

In Abschnitt 1.3, »Die Architektur der WPF«, haben Sie bereits einen kleinen Blick hinter die Kulissen der WPF werfen können. In diesem Abschnitt erfahren Sie mehr über ein paar der grundlegenden Konzepte der WPF, die auf der .NET-Seite verankert sind. Neben der deklarativen Sprache XAML sind dies unter anderem Dependency Properties, Routed Events, Com-

mands, Styles, Templates und 3D. Hier dargestellte und weitere Konzepte – wie Layout, Ressourcen und Data Binding – werden in späteren Kapiteln separat betrachtet. An dieser Stelle erhalten Sie lediglich einen kleinen Einblick. Es soll dabei (noch) nicht jedes Detail beleuchtet werden; betrachten Sie diesen Abschnitt somit als eine kleine Schnupperrunde, die Sie locker angehen können.

1.4.1 XAML

Die *Extensible Application Markup Language* (*XAML*) ist eine XML-basierte Beschreibungssprache, mit der Sie Objektbäume erstellen können. Zur Laufzeit werden aus den in XAML deklarierten XML-Elementen .NET-Objekte erzeugt.

Hinweis

Mit anderen Worten: XAML ist ein Serialisierungsformat. Zur Laufzeit werden die Inhalte einer XAML-Datei deserialisiert und die entsprechenden Objekte erzeugt.

Bei der WPF wird XAML für die Beschreibung von Benutzeroberflächen eingesetzt. Dafür definieren Sie in XAML Controls, Styles, Animationen oder 3D-Objekte, um nur einige der Möglichkeiten zu nennen. Folgender Codeschnipsel stellt bereits einen gültigen XAML-Ausschnitt dar und erstellt einen Button mit kursiver Schriftart, einem Rand von 10 Einheiten und dem Inhalt »OK«:

```
<Button Name="btnOk" FontStyle="Italic" Margin="10"> OK
</Button>
```

Listing 1.1 Beispiele\K01\01 Button\MainWindow.xaml

Event Handler und sonstige Logik werden üblicherweise in einer in C# geschriebenen Codebehind-Datei eingefügt. Das `Button`-Element aus Listing 1.1 wird vom XAML-Parser der Klasse `Button` aus dem CLR-Namespace `System.Windows.Controls` zugeordnet. Die Details werden wir uns in Kapitel 3, »XAML«, genauer ansehen. Abbildung 1.7 zeigt den Button in einen Fenster. Der Button hat einem Rand von 10, kursive Schriftart und den Inhalt »OK«.

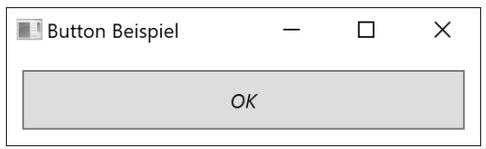

Abbildung 1.7 Der erstellte Button

Anstatt Ihre Benutzeroberflächen mit XAML zu definieren, ist natürlich auch der prozedurale Weg mit C# möglich. Der XAML-Ausschnitt aus Listing 1.1 entspricht folgendem C#-Code:


```
var btnOk = new System.Windows.Controls.Button
{
  FontStyle = System.Windows.FontStyles.Italic,
  Margin = new System.Windows.Thickness(10),
  Content = "Ok"
};
```

Listing 1.2 Ein Button in C#

Obwohl Sie Ihre Benutzeroberfläche auch rein in C# erstellen können – und dies in Ausnahmen bei komplexen Oberflächen manchmal auch sinnvoll ist –, profitieren Sie dann natürlich nicht von den Vorteilen, die Ihnen XAML bietet:

- Sie können mit XAML die Darstellung Ihrer Anwendung besser von der dahinterliegenden Businesslogik trennen. Üblicherweise definieren Sie dazu in XAML die Beschreibung Ihrer Oberfläche und setzen die eigentliche Logik in eine Codebehind-Datei, die in einer prozeduralen Sprache wie C# programmierte Methoden und Event Handler enthält. Das Prinzip der Codebehind-Datei kennen Sie vielleicht aus ASP.NET; auch dort werden Darstellung und Logik auf diese Weise getrennt.
- Die Beschreibung einer Benutzeroberfläche in XAML ist wesentlich kompakter und übersichtlicher als die Erstellung in C#. XAML schraubt somit die Komplexität Ihres Codes nach unten.
- Eine XAML-Datei ist ideales Futter für einen Designer, der Ihrer Benutzeroberfläche mit Tools wie Blend mehr Leben einhaucht und eine zeitgemäße Darstellung verleiht.
- In XAML erstellte Benutzeroberflächen werden zur Designzeit stets aktuell im WPF-Designer dargestellt. Für in C# erstellte Benutzeroberflächen gilt das dagegen nicht; diese sehen Sie erst zur Laufzeit.
- In XAML lässt sich jede öffentliche .NET-Klasse verwenden, die einen Default-Konstruktor besitzt.
- Wenn Sie bereits mit Windows Forms und Visual Studio programmiert haben, dann wissen Sie, dass in Visual Studio zu jeder `Form` Code vom Windows-Forms-Designer generiert wird. Seit .NET Framework 2.0 wird dieser generierte Code in eine partielle Klasse ausgelagert. Sobald händisch etwas an diesem generierten Code geändert wird, hat der Windows-Forms-Designer meist Probleme, die `Form` wieder korrekt darzustellen, da er eine bestimmte Formatierung des C#-Codes voraussetzt. In XAML besteht dieses Problem nicht: Sie können in XAML beliebige Änderungen durchführen, die sofort vom WPF-Designer dargestellt werden. Nehmen Sie umgekehrt Änderungen im WPF-Designer vor, werden diese gleich in XAML übernommen.

Wie Sie sehen, bietet XAML einige Vorteile gegenüber C#. Bedenken Sie jedoch, dass XAML die Sprache C# nicht ersetzen wird. XAML ist eine deklarative Sprache, in der Sie beispiels-

weise keine Methoden definieren können. Dennoch eignet sich eine deklarative Sprache bestens für die Definition von Benutzeroberflächen.

In Kapitel 3, »XAML«, werden wir XAML genau durchleuchten und uns die Syntax und Möglichkeiten dieser deklarativen, XML-basierten Sprache ansehen. An dieser Stelle wenden wir uns einem weiteren Konzept zu, auf dem die WPF aufbaut, den Dependency Properties.

1.4.2 Dependency Properties

Dependency Properties sind eines der wichtigsten Konzepte der WPF. In der WPF lassen sich Properties auf verschiedene Arten setzen: entweder auf dem üblichen Weg direkt auf einem Objekt in C#, in XAML oder über Styles, Data Binding oder Animationen. Einige Properties werden sogar durch eine Eltern-Kind-Beziehung »vererbt«. Ändern Sie den Wert der `FontSize`-Property eines `Window`-Objekts, wird der gesetzte Wert wie von Geisterhand – im Hintergrund hat natürlich die WPF ihre Hände im Spiel – auch von im Fenster enthaltenen `Button`- und `TextBox`-Objekten verwendet.

Eine Dependency Property ist also von mehreren Quellen in Ihrer Anwendung und im System abhängig – daher der Name »Dependency« (dt. »Abhängigkeit«). Wenn eine Dependency Property nicht gesetzt ist, hat sie einen Default-Wert.

Dependency Properties sind bei der WPF die Grundlage für Styles, Animationen, Data Binding, Property-Vererbung und vieles mehr. Mit einer normalen .NET-Property können Sie keinen Gebrauch von diesen »Diensten« der WPF machen. Glücklicherweise sind die meisten Properties der Elemente der WPF als Dependency Properties implementiert und lassen sich somit mit Animationen, Data Bindings oder Styles verwenden.

Die wohl wichtigste Eigenschaft einer Dependency Property ist ihr integrierter Benachrichtigungsmechanismus für Änderungen, wodurch die WPF beobachten kann, wann sich ihr Wert ändert. Dies macht sie auch als Quelle für ein Data Binding ideal.

Dependency Properties werden in der Laufzeitumgebung der WPF in der sogenannten *Property Engine* registriert, die die Grundlage für diverse Möglichkeiten ist, wie etwa die Property-Vererbung.

Zusammengefasst bieten Dependency Properties folgenden Mehrwert gegenüber klassischen .NET-Properties:

- Sie verfügen über einen integrierten Benachrichtigungsmechanismus.
- Sie besitzen einen Default-Wert.
- Sie besitzen Metadaten, die unter anderem die Information enthalten, ob durch eine Änderung des Wertes ein Neuzeichnen des visuellen Elements notwendig ist.
- Sie verfügen über eine integrierte Validierung.
- Sie bieten Property-Vererbung über den Visual Tree.

- Viele Dienste der WPF (wie Animationen oder Styles) lassen sich nur mit Dependency Properties verwenden. Mit normalen Properties wäre es ohne weiteren Code nicht möglich, zu bestimmen, welche Quelle (Animation, Style, lokaler Wert etc.) den endgültigen Wert einer Dependency Property festlegt.
- Sie können als Attached Property implementiert auch auf anderen Elementen gesetzt werden.

Aus Entwicklersicht besteht eine Dependency Property aus einer klassischen .NET-Property – wenn diese auch optional ist – und aus einem öffentlichen, statischen Feld vom Typ `DependencyProperty`. Dieses Feld stellt den Schlüssel zum eigentlichen Wert der Property dar.

```
public class MyClass:DependencyObject
{
  public static readonly DependencyProperty FontSizeProperty
  = DependencyProperty.Register("FontSize"
      ,typeof(double)
      ,typeof(Button)
      ,new FrameworkPropertyMetadata(11.0
      ,FrameworkPropertyMetadataOptions.Inherits
      |FrameworkPropertyMetadataOptions.AffectsRender));
  public double FontSize
  {
      get { return (double)GetValue(FontSizeProperty); }
      set { SetValue(FontSizeProperty, value); }
  }
}
```

Listing 1.3 Implementierung einer Dependency Property

Die Methoden `GetValue` und `SetValue`, die in Listing 1.3 in den `get`- und `set`-Accessoren der .NET-Property aufgerufen werden, sind in der Klasse `DependencyObject` definiert, von der die dargestellte Klasse erbt. Jede Klasse, die Dependency Properties speichern möchte, muss von `DependencyObject` abgeleitet sein.

Der obere Codeausschnitt soll Ihnen nur eine kleine Vorstellung davon geben, wie die Implementierung einer Dependency Property aussieht. In Kapitel 7, »Dependency Properties«, werden wir diese Implementierung ausführlich betrachten. Denken Sie an dieser Stelle noch an das Motto dieses Kapitels: »Lehnen Sie sich zurück«. Wir werden uns später alles noch sehr genau anschauen.

Auf den ersten Blick werden Sie aufgrund der Kapselung durch eine normale .NET-Property nicht bemerken, dass Sie auf eine Dependency Property zugreifen. Beispielsweise ist die `FontSize`-Property der `Button`-Klasse als Dependency Property implementiert, die sich auf-

grund der Kapselung durch eine »normale« .NET-Property auch wie eine solche verwenden lässt:

```
var btn = new System.Windows.Controls.Button();
btn.FontSize = 15.0;
```

Hinweis

Im Gegensatz zu einer normalen .NET-Property kann die als Dependency Property implementierte `FontSize` in Animationen oder Styles verwendet werden.

Neben der Kombination eines statischen Feldes vom Typ `DependencyProperty` mit einer normalen .NET-Property als Wrapper treten Dependency Properties auch als Attached Properties auf. Das Besondere an einer Attached Property ist, dass sie Teil einer Klasse ist, aber auf Objekten anderer Klassen gesetzt wird. Dies mag zunächst etwas verwunderlich klingen, wird aber insbesondere bei Layout-Panels verwendet. Kindelemente müssen somit nicht mit unnötig vielen Eigenschaften für jedes Layout-Panel überladen werden, da die Definition der Dependency Properties im Panel selbst liegt. Wie kann das gehen?

Das Panel definiert nur den Schlüssel für einen Wert. Dieser Schlüssel ist ein statisches Feld vom Typ `DependencyProperty`. Objekte, die Dependency Properties speichern, müssen zwingend vom Typ `DependencyObject` sein. Diese Klasse enthält vereinfacht gesehen eine Art Hashtable, in der mit der Methode `SetValue` Schlüssel/Wert-Paare gespeichert werden. Alle Controls der WPF leiten von dieser Klasse ab und besitzen somit intern eine solche Art Hashtable. Möchten Sie auf einem Button eine Layout-Property speichern, so wird der Wert in der Hashtable des `Button`-Objekts unter dem in der `Panel`-Klasse definierten Schlüssel gespeichert. Nimmt das `Panel`-Objekt das Layout vor, so kann es mit dem Schlüssel die für das Layout benötigten Werte der einzelnen Controls abrufen.

XAML definiert für die Attached Properties eine eigene Syntax. In Listing 1.4 wird ein Grid mit zwei Zeilen definiert. Ein Grid ist eines der Layout-Panels der WPF, das Elemente in Zeilen und Spalten anordnet. Im Grid in Listing 1.4 wird eine TextBox der ersten und ein Button der zweiten Zeile zugeordnet. Dazu wird die in der Klasse `Grid` als Attached Property implementierte `Grid.Row` auf der `TextBox` und auf dem `Button` gesetzt:

```
<Grid>
 <Grid.RowDefinitions>
  <RowDefinition/>
  <RowDefinition/>
 </Grid.RowDefinitions>
 <TextBox Grid.Row="0" Text="In Zeile 1 des Grid" Margin="10"/>
 <Button Grid.Row="1" Content="In Zeile 2 des Grid" Margin="10"/>
</Grid>
```

Listing 1.4 Beispiele\K01\02 DependencyProperties\MainWindow.xaml

Nimmt das `Grid` nun das Layout vor, so kann es auf jedem einzelnen Element die `Grid.Row` durch Aufruf von `GetValue` abfragen und so die Elemente in die entsprechende Zeile setzen. Abbildung 1.8 zeigt das `Grid` aus Listing 1.4 in einem Fenster.

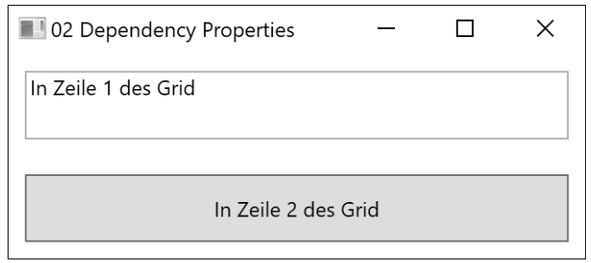

Abbildung 1.8 Durch die Attached Properties werden hier eine TextBox und ein Button in einem Grid untereinander in zwei Zeilen angeordnet.

Obwohl Sie Dependency Properties wahrscheinlich meist nur bei der Implementierung eigener Controls benötigen, trägt ihr Verständnis natürlich zu einem effektiveren Umgang mit der WPF bei. Beim Einstieg in die WPF sorgen Dependency Properties bei den meisten Entwicklern für Missverständnisse und manchmal auch für etwas Frust. Dies liegt meist daran, dass am Anfang nicht ganz klar ist, wofür die Dependency Properties letztlich gut sind. So lassen sie das Bild von .NET ein wenig komplizierter erscheinen. Sehen Sie sich nochmals den Mehrwert von Dependency Properties gegenüber klassischen .NET-Properties am Anfang dieses Abschnitts an. Behalten Sie an dieser Stelle im Hinterkopf, dass Sie die in der WPF integrierten Animationen, Styles und vieles mehr nur mit Dependency Properties verwenden können. Auch das Prinzip der Attached Properties – die Sie unter anderem in Kapitel 6, »Layout«, noch öfter sehen werden – ist nur dank Dependency Properties möglich.

Da die Dependency Properties ein zentrales Element der WPF sind, dieser Abschnitt sicher einige Fragen offen lässt und Ihren Wissensdurst in Sachen Dependency Properties gewiss und hoffentlich nicht gestillt hat, ist ihnen in diesem Buch ein eigenes Kapitel gewidmet. In Kapitel 7, »Dependency Properties«, werden Sie alle Details zu Dependency Properties und ihrer Implementierung erfahren.

1.4.3 Routed Events

Neben Dependency Properties bilden Routed Events ein in der WPF durchgängig verwendetes Konzept. Bei der WPF besteht das Prinzip der Entwicklung von Benutzeroberflächen darin, eine Hierarchie von Elementen zu erzeugen; die Hierarchie wird als *Element Tree* bezeichnet. Die Elemente im Element Tree stehen in einer Eltern-Kind-Beziehung. Ein Button kann als Inhalt ein StackPanel – ein Layout-Container der WPF – enthalten, und darin könnte sich ein einfaches Rectangle-Objekt befinden.

Stellen Sie sich vor, was passiert, wenn ein Benutzer auf das im StackPanel der Button-Instanz enthaltene Rectangle klickt. Bekommt der Button dann auch eine Benachrichtigung über das ausgelöste `MouseLeftButtonDown`-Event? In bisherigen Programmiermodellen wird er nicht benachrichtigt, denn bisher war es so, dass das Element, das im Vordergrund steht und auf dem der Fokus liegt, auch das Event empfängt – in diesem Fall das `MouseLeftButtonDown`-Event.

In der WPF ist die klassische Behandlung von Events nicht ausreichend, da sich auch einfache Controls wie ein Button aus mehreren anderen visuellen Elementen zusammensetzen und nach dem klassischen Prinzip diese Elemente das Event absorbieren würden. Der `Button` selbst erhielte somit eventuell gar keine Information darüber, dass etwas »in ihm« geklickt wurde. Um zu unserem Button mit einem StackPanel und einem Rectangle zurückzukommen, bedeutet dies, dass der Button, wenn ein Benutzer auf das Rectangle oder das StackPanel klickt, nach klassischem Event Handling selbst kein `MouseLeftButtonDown`-Event mitbekommt. Und genau das ist der Punkt, an dem die Routed Events der WPF ins Spiel kommen. Routed Events können bei der WPF eine von drei verschiedenen Routing-Strategien verwenden:

- *Tunnel* – Das Event wird von oben durch den Visual Tree in niedrigere Hierarchiestufen geroutet.
- *Bubble* – Das Event »blubbert« von einem im Visual Tree tiefer liegenden Element nach oben.
- *Direct* – Das Event wird nur auf dem geklickten visuellen Element ausgelöst. Diese Strategie gleicht der bei Events, die Sie aus der bisherigen Programmierung mit .NET 2.0 und früher kennen – mit der Ausnahme, dass sich diese Events auch in einem sogenannten `EventTrigger` verwenden lassen.

Das `MouseLeftButtonDown`-Event der WPF besitzt die Strategie `Bubble`. Wird in unserem Button auf das Rectangle geklickt, »blubbert« das Event nach oben zum StackPanel und von dort zum Button. Der Button selbst löst beim Empfang des `MouseLeftButtonDown`-Events sein eigenes `Click`-Event aus, das auch die `Bubble`-Strategie besitzt und weiter nach oben blubbert.

Tunneling Events und Bubbling Events treten oft in Paaren auf. So gibt es zum Bubbling Event `MouseLeftButtonDown` ein passendes Gegenstück, das Tunneling Event `PreviewMouseLeftButtonDown`. Abbildung 1.9 verdeutlicht den Button und das darin enthaltene StackPanel mit dem Rectangle als Inhalt. Sie sehen, wie zuerst die Tunneling Events – konventionsgemäß mit dem Präfix »Preview« benannt – von oben nach unten und wie anschließend die Bubbling Events von unten nach oben ausgelöst werden. Jedes Element, das auf dieser Route liegt, kann beispielsweise für das `MouseLeftButtonDown`-Event einen Event Handler installieren.

In Abbildung 1.10 sehen Sie ein Fenster, das unseren Button enthält. Im Button ist ein StackPanel (schwarz), das wiederum ein Rectangle (rot) enthält. Auf dem Window, auf dem Stack-

Panel, auf dem Button und auf dem Rectangle wurden Event Handler für die Events `PreviewMouseLeftButtonDown` und `MouseLeftButtonDown` installiert, die das Event und den Sender in einer ListView ausgeben.

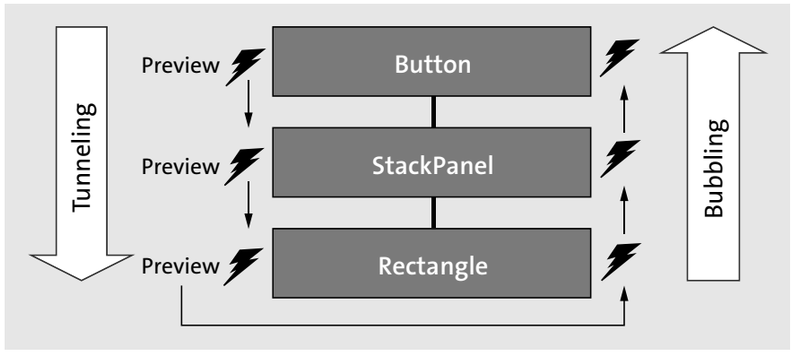

Abbildung 1.9 Bei der WPF werden Routed Events im Element Tree von oben nach unten getunnelt und blubbern von unten nach oben.

In Abbildung 1.10 wurde direkt auf das Rectangle geklickt. Die ListView unterhalb des Buttons wird anschließend durch die Event Handler auf den einzelnen Elementen mit Event-Informationen gefüllt. Darin lässt sich gut erkennen, wie das `PreviewMouseLeftButtonDown`-Event vom Window bis zum Rectangle getunnelt wird. Anschließend blubbert das `MouseLeftButtonDown`-Event vom Rectangle im Element Tree nach oben zurück zum `Window`.

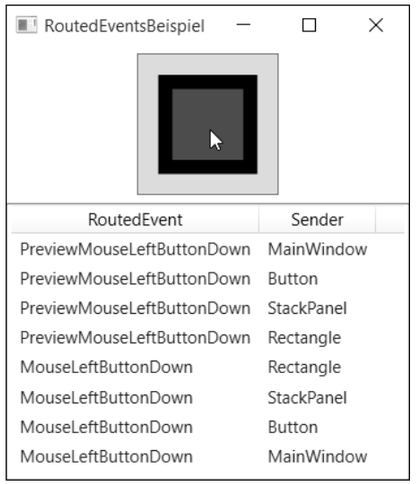

Abbildung 1.10 Das Event wird vom MainWindow bis zum geklickten roten Rectangle getunnelt und blubbert wieder nach oben zum MainWindow.

In Abbildung 1.11 sehen Sie, wie das Event geroutet wird, wenn nicht auf das rote Rectangle, sondern auf das schwarze StackPanel geklickt wird.

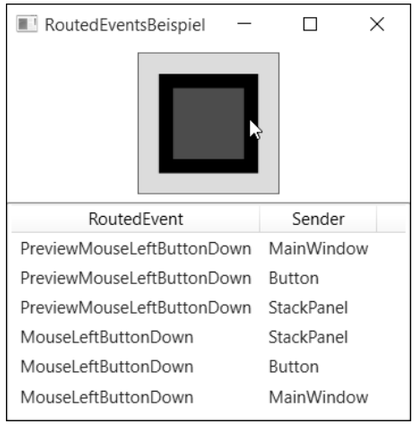

Abbildung 1.11 Das Event wird vom MainWindow bis zum geklickten schwarzen StackPanel getunnelt und blubbert wieder nach oben zum MainWindow.

Ähnlich wie bei den mithilfe von Dependency Properties implementierten Attached Properties lassen sich Routed Events auch auf visuellen Elementen setzen, die das Routed Event nicht selbst definieren. Es wird dann von *Attached Events* gesprochen.

Stellen Sie sich folgendes simples Beispiel vor: Sie haben ein Fenster, das neun Buttons enthält. Jeder Button soll beim Klicken eine Messagebox mit einer `String`-Repräsentation seines Inhalts anzeigen. Mit gewöhnlichen Events installieren Sie für diese Aufgabe pro Button einen Event Handler für das `Click`-Event, oder Sie verwenden einen gemeinsamen Event Handler, den Sie dennoch mit dem `Click`-Event jedes Buttons verbinden müssen.

Wie Sie bereits erfahren haben, besitzt das `Click`-Event die Strategie `Bubble`. Somit haben Sie die Möglichkeit, nur einen Event Handler für das `Button.ClickEvent` auf dem Window-Objekt zu installieren. Wird auf einen Button geklickt, blubbert das `Click`-Event vom Button nach oben zum Window-Objekt. Das Window-Objekt erhält das Event und ruft den Event Handler auf. Allerdings befindet sich in der `sender`-Variablen des Event Handlers immer das Window-Objekt. Sie erhalten über die `Source`-Property der bei Routed Events verwendeten `RoutedEventArgs` eine Referenz auf den geklickten Button und können so die MessageBox mit dem Namen anzeigen.

Wie Sie sehen, eröffnen sich Ihnen mit Routed Events interessante neue Möglichkeiten. In Kapitel 8, »Routed Events«, erfahren Sie alle notwendigen Details zu Routed Events und lernen, wie Sie Routed Events in eigenen Klassen implementieren.

1.4.4 Commands

Ein weiteres Konzept der WPF ist die integrierte Infrastruktur für Commands. Im Gegensatz zu Events erlauben Commands eine bessere Trennung der Präsentationsschicht von der Anwendungslogik. Sie geben auf einem Element in XAML nicht direkt einen Event Handler an, sondern lediglich ein Command.

Stellen Sie sich vor, Sie werden beauftragt, einen kleinen Texteditor zu programmieren. Darin benötigen Sie unter anderem Funktionalität zum Ausschneiden, Kopieren und Einfügen von Text. Typischerweise kann ein Benutzer diese Funktionen aus dem Menü, dem Kontextmenü oder der Toolbar aufrufen. In unserem Beispiel soll es der Einfachheit halber je Funktion ein MenuItem wie auch einen Button geben.

Klassischerweise würden Sie für jedes MenuItem und jeden Button einen Event Handler implementieren, der die entsprechende Funktion ausführt. Haben Sie den Event Handler für das Ausführen des Kopierens auf dem MenuItem und auch auf dem Button erstellt, so müssen Sie darüber hinaus Ihre Controls entsprechend aktivieren und deaktivieren. Denn nur, wenn Text in der Textbox markiert ist, sollen die Controls für die Kopierfunktionalität auch aktiviert sein. Auch dies lässt sich mit den entsprechenden Event Handlern ausprogrammieren. Neben dem Aktivieren und Deaktivieren möchten Sie eventuell die Tastenkombinationen Strg+C unterstützen. Hier ist ebenso ein weiterer Event Handler notwendig, der auf das `KeyDown`-Event reagiert und das entsprechende Kopieren des markierten Textes in die Zwischenablage auslöst.

Wenn Sie ein weiteres Control zu Ihrem Texteditor hinzufügen, das die Copy-Funktionalität unterstützen soll (beispielsweise einen Button in einer Toolbar), dann müssen Sie auch für dieses Control wieder alle Event Handler installieren und dafür sorgen, dass das Control richtig aktiviert und deaktiviert wird. Möchten Sie die Controls, die an der Kopierfunktionalität teilnehmen, nicht fest in Ihrem Code verdrahten, wartet schon mehr Aufwand auf Sie.

Glücklicherweise gibt es in der WPF eine eigene Infrastruktur für Commands, die Ihnen genau solche Aufgaben wesentlich erleichtert und Ihnen im Gegensatz zu den Events eine bessere Entkopplung Ihrer Benutzeroberfläche von der Anwendungslogik erlaubt. Klären wir zunächst, was ein Command überhaupt ist. Bei einem Command handelt es sich um ein Objekt einer Klasse, die das Interface `ICommand` (Namespace: `System.Windows.Input`) implementiert, das die Methoden `Execute` und `CanExecute` und das Event `CanExecuteChanged` definiert:

- `Execute` löst das Command aus.
- `CanExecute` gibt einen booleschen Wert zurück, der angibt, ob das Command überhaupt ausgelöst werden kann.
- `CanExecuteChanged` wird ausgelöst, wenn `CanExecute` beim nächsten Aufruf vermutlich einen anderen Wert zurückgibt.

Einem Button können Sie ein `ICommand`-Objekt über die `Command`-Property zuweisen. Das `ICommand` wird automatisch ausgelöst, sobald Sie auf den Button klicken. Gibt die `CanExecute`-Methode des Commands `false` zurück, so wird die `IsEnabled`-Property des Buttons automatisch auf `false` gesetzt.

Mit der Klasse `RoutedCommand` besitzt die WPF bereits eine pfannenfertige `ICommand`-Implementierung. Es gibt bereits vordefinierte `RoutedCommand`-Objekte. So besitzt die Klasse `ApplicationCommands` statische Properties wie `Copy` oder `Paste`, die `RoutedCommand`-Instanzen enthalten. Der Knackpunkt bei den Routed Commands ist, dass eine `RoutedCommand`-Instanz die Logik für die Behandlung des Commands nicht selbst enthält. Das heißt, die `Execute`-Methode der `RoutedCommand`-Klasse enthält nicht die Logik, die beispielsweise für einen Kopiervorgang notwendig ist.

Die `Execute`-Methode eines `RoutedCommand`s löst im Hintergrund vereinfacht gesagt nur eine Suche am Element Tree entlang nach sogenannten `CommandBinding`-Objekten aus. Jedes Control hat eine `CommandBindings`-Property, die mehrere `CommandBinding`-Objekte enthalten kann.

Ein `CommandBinding`-Objekt besitzt eine Referenz auf ein `ICommand` und definiert unter anderem die Events `Executed` und `CanExecute`. Das Event `Executed` eines `CommandBinding`s wird ausgelöst, wenn das `RoutedCommand` ausgeführt wird. Vor dem `Executed`-Event wird immer das Event `CanExecute` ausgeführt. Im Event Handler dieses Events setzen Sie die Property `CanExecute` der `CanExecuteRoutedEventArgs` auf `false`, damit das Command nicht ausgeführt werden kann und ein Button, dessen `Command`-Property das entsprechende Command enthält, beispielsweise automatisch deaktiviert wird.

Die Suche nach einem `CommandBinding` und damit nach der Logik, die für ein Command ausgeführt werden soll, beginnt meist bei dem fokussierten Control. Allerdings lässt sich auf einem MenuItem oder auf einem Button auch explizit die `CommandTarget`-Property auf ein bestimmtes Control setzen, wodurch die Suche bei diesem `CommandTarget` beginnt. Wird kein `CommandBinding` auf dem Zielelement gefunden, wird im Element Tree auf dem nächsthöheren Element nach einem `CommandBinding` für das ausgelöste Command gesucht. Die Suche endet, wenn ein `CommandBinding`-Objekt gefunden wurde oder die Suche beim Wurzelelement angelangt ist.

Hinweis

Das Zielelement eines Commands ist das in der `CommandTarget`-Property eines MenuItems angegebene Element. Ist die `CommandTarget`-Property des MenuItems `null`, wird als Zielelement des Commands das Element mit dem Tastatur-Fokus verwendet. Beim Zielelement beginnt dann die Suche nach `CommandBinding`-Objekten aufwärts im Element Tree.

Die große Stärke der WPF besteht nun darin, dass viele Controls für die in der WPF vordefinierten Commands – wie eben `ApplicationCommands.Copy` – bereits `CommandBinding`-Objekte und somit vordefinierte Logik besitzen. Eine TextBox hat für das Command `ApplicationCommands.Copy` ein `CommandBinding` definiert, das im `CanExecute`-Event die Property `CanExecute` der `CanExecuteEventArgs` auf `false` setzt, falls in der Textbox kein Text markiert ist. Wird das `ApplicationCommands.Copy` ausgeführt, wird im `Executed`-Event-Handler des in der TextBox enthaltenen `CommandBinding`s der selektierte Text in die Zwischenablage kopiert.

Die in den Controls der WPF bereits vordefinierten `CommandBinding`-Instanzen erlauben es Ihnen, einen funktionsfähigen Texteditor ohne prozeduralen Code rein in XAML zu erstellen (siehe Listing 1.5).

```
<StackPanel>
  <Menu>
    <MenuItem Header="Copy" Command="ApplicationCommands.Copy"/>
    <MenuItem Header="Cut" Command="ApplicationCommands.Cut"/>
    ...
  </Menu>
  <TextBox MinHeight="100" TextWrapping="Wrap" Margin="5">
  ...
</StackPanel>
```

Listing 1.5 Beispiele\K01\03 Commands\MainWindow.xaml

Hat der Benutzer im Texteditor aus Listing 1.5 keinen Text markiert, sind die MenuItems für Copy und Cut deaktiviert (siehe Abbildung 1.12). Sobald er Text selektiert, werden die MenuItems und Buttons aktiviert.

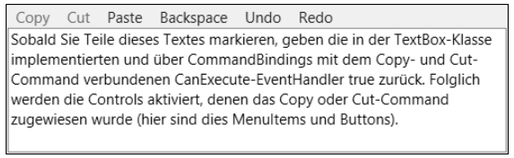

Abbildung 1.12 Der Texteditor mit den Befehlen im Menü

In Kapitel 9, »Commands«, werden wir die Infrastruktur der Commands genauer unter die Lupe nehmen. Sie werden anhand der FriendStorage-Anwendung sehen, wie eigene Routed Commands implementiert werden, und Sie lernen die vordefinierten Built-in-Commands der WPF kennen. Zudem erfahren Sie in Kapitel 9 mehr zum sogenannten Model-View-View-Model-Pattern (MVVM), das auf Commands und Data Binding basiert.

1.4.5 Styles und Templates

Mit Styles und Templates lassen sich die Controls der WPF sehr einfach anpassen. Ein Style definiert lediglich eine Sammlung von Werten für Properties. Meistens wird ein Style als Ressource erstellt, damit er sich auf mehrere Elemente anwenden lässt. In Listing 1.6 wird ein Style für Buttons erstellt, der die `Width`-, `Height`- und `Template`-Property setzt.

```
<Window ...>
  <Window.Resources>
    <Style TargetType="Button">
      <Setter Property="Width" Value="75"/>
      <Setter Property="Height" Value="30"/>
      <Setter Property="Background" Value="Black"/>
      <Setter Property="Foreground" Value="White"/>
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate TargetType="Button">
            <Grid>
              <Border Background="{TemplateBinding Background}"
                      CornerRadius="5"/>
                <ContentPresenter HorizontalAlignment="Center"
                                  VerticalAlignment="Center"/>
            </Grid>
          </ControlTemplate>
        </Setter.Value>
      </Setter>
    </Style>
  </Window.Resources>
  <StackPanel>
    <Button Content="OK"/>
    <Button Content="Abbrechen" Margin="5"/>
  </StackPanel>
</Window>
```

Listing 1.6 Beispiele\K01\04 StylesUndTemplates\MainWindow.xaml

Mit dem Style in Listing 1.6 wird die `Template`-Property für Buttons in diesem `Window`-Objekt gesetzt. Die Buttons behalten ihre ganz normale Funktionalität, lösen beim Klicken das `Click`-Event aus usw., werden aber durch das `ControlTemplate` nicht wie gewöhnliche Buttons dargestellt, wie Abbildung 1.13 zeigt. Dieser Ausschnitt gibt Ihnen nur einen kleinen Vorgeschmack darauf, wie einfach Sie das Aussehen der Controls der WPF komplett verändern können, indem Sie ein neues Template definieren.

Abbildung 1.13 Mit einem Template angepasste Buttons

1.4.6 3D

Mit der WPF können Sie 3D-Inhalte einfach in Ihre Anwendungen integrieren. 3D-Objekte lassen sich vollständig in XAML definieren. Der dreidimensionale Inhalt wird durch das Element `Viewport3D` dargestellt. Listing 1.7 enthält ein `Viewport3D`-Element. Darin wird ein Würfel erstellt (siehe Abbildung 1.14). Die Details lernen Sie in Kapitel 14, »3D-Grafik«, kennen.

```
<Viewport3D Height="250" Width="250">
  <Viewport3D.Camera>
 <PerspectiveCamera Position="0.5 0.5 4" LookDirection="0 0 -1"/>
  </Viewport3D.Camera>
  <Viewport3D.Children>
    <ModelVisual3D>
      <ModelVisual3D.Content>
        <DirectionalLight Color="White" Direction="0,0,-1"/>
      </ModelVisual3D.Content>
    </ModelVisual3D>
    <ModelVisual3D>
      <ModelVisual3D.Content>
        <GeometryModel3D>
          <GeometryModel3D.Geometry>
            <MeshGeometry3D Positions="0,0,1 1,0,1 1,1,1 0,1,1
                                       1,0,0 0,0,0 0,1,0 1,1,0
                                       1,0,1 1,0,0 1,1,0 1,1,1
                                       0,0,0 0,0,1 0,1,1 0,1,0
                                       0,1,1 1,1,1 1,1,0 0,1,0
                                       1,0,1 0,0,1 0,0,0 1,0,0"
                          TriangleIndices="0 1 2 2 3 0
                                           4 5 6 6 7 4
                                           8 9 10 10 11 8
                                           12 13 14 14 15 12
                                           16 17 18 18 19 16
                                           20 21 22 22 23 20"
                          TextureCoordinates="0,1 1,1 1,0 0,0
                                              0,1 1,1 1,0 0,0
                                              0,1 1,1 1,0 0,0
                                              0,1 1,1 1,0 0,0
                                              0,1 1,1 1,0 0,0
                                              0,1 1,1 1,0 0,0"/>
          </GeometryModel3D.Geometry>
          <GeometryModel3D.Material>
            <DiffuseMaterial>
              <DiffuseMaterial.Brush>
                <ImageBrush ImageSource="thomas.jpg"/>
```

```
            </DiffuseMaterial.Brush>
          </DiffuseMaterial>
        </GeometryModel3D.Material>
      </GeometryModel3D>
    </ModelVisual3D.Content>
    <ModelVisual3D.Transform>
      <RotateTransform3D>
        <RotateTransform3D.Rotation>
          <AxisAngleRotation3D Axis="0.5 1 0" Angle="30"/>
        </RotateTransform3D.Rotation>
      </RotateTransform3D>
    </ModelVisual3D.Transform>
  </ModelVisual3D>
 </Viewport3D.Children>
</Viewport3D>
```

Listing 1.7 Beispiele\K01\05 3D\MainWindow.xaml

Abbildung 1.14 Einfaches 3D-Objekt mit einem Bild

Hinweis

Obwohl die 3D-API der WPF auf DirectX aufbaut, ist sie nicht zum Entwickeln von Spielen gedacht. Dafür sollten Sie Managed DirectX verwenden, das wesentlich performanter ist. Die 3D-API der WPF ist allerdings ein recht einfaches Mittel, um Ihrer Anwendung mit 3D-Effekten etwas Pep zu verleihen oder um beispielsweise Geschäftsdaten in 3D darzustellen.

1.5 Zusammenfassung

Die WPF ist ein umfangreiches Programmiermodell, mit dem sich professionelle Desktop-Anwendungen entwickeln lassen. Die WPF wird im .NET Framework ab Version 3.0 und auch in .NET Core ab Version 3.0 unterstützt. .NET Core ist die Zukunft, und neue WPF-Anwendungen sollten mit .NET Core entwickelt werden.

Mit Visual Studio 2019 lassen sich WPF-Anwendungen für .NET Framework und .NET Core erstellen.

Während bisherige Programmiermodelle unter Windows nur dünne Wrapper um die Windows-API waren – so auch Windows Forms –, ist die WPF eine UI-Bibliothek, die in .NET entwickelt wurde und nicht mehr auf der Windows-API aufbaut, sondern Inhalte selbst via DirectX zeichnet.

In der WPF können Sie Oberflächen mit der XML-basierten Beschreibungssprache XAML definieren. XAML wird als Austauschformat zwischen Designer und Entwickler verwendet. Doch auch, wenn Sie allein, ohne Designer, eine Anwendung erstellen, erlaubt Ihnen XAML eine bessere Strukturierung Ihrer Anwendung und eine bessere Trennung zwischen Ihrer Benutzeroberfläche und Ihrer Programmlogik.

Visuelle Elemente werden in der WPF nicht durch das Betriebssystem, sondern durch die WPF selbst gezeichnet. Dazu wird die auf DirectX aufsetzende MilCore-Komponente verwendet.

Die WPF zeichnet die Inhalte Ihrer Anwendung vektorbasiert. Dadurch ist Ihre Anwendung beliebig skalierbar und wird auch bei höherer Auflösung nicht pixelig dargestellt.

Die WPF besitzt ein flexibles Inhaltsmodell, wodurch Sie in jedes visuelle Element andere visuelle Elemente packen können.

Die WPF bietet integrierte Unterstützung für Animationen, 2D- und 3D-Grafiken, Layout, Data Binding und vieles mehr.

Mit der WPF können Sie nicht nur einfacher grafisch hochwertige Benutzeroberflächen erstellen als zuvor. Konzepte wie Dependency Properties, Routed Events und Commands bieten Ihnen auch bei der Entwicklung von reinen Geschäftsanwendungen, die grafisch nicht so anspruchsvoll sind, viele Möglichkeiten. Mit Templates können Sie zudem das Aussehen eines Controls komplett neu definieren.

Benutzeroberflächen können Sie nur aus XAML, nur aus C# (oder einer anderen prozeduralen Sprache wie VB.NET) oder aus einer Mischung aus XAML und C# (in einer Codebehind-Datei) erstellen.

Im nächsten Kapitel werden wir uns das Programmiermodell der WPF ansehen. Sie werden unter anderem die wichtigsten Klassen der WPF kennenlernen, und wir werden die erste Windows-Anwendung mit der WPF implementieren.

Kapitel 3
XAML

XAML (sprich: »Semmel«) ist eine XML-basierte Beschreibungssprache. Sie wird in der WPF und in UWP/WinUI zur Definition von Benutzeroberflächen eingesetzt. In diesem Kapitel erfahren Sie mehr über die Syntax und die Funktionsweise von XAML.

Die *Extensible Application Markup Language* (XAML) wurde zusammen mit der WPF eingeführt. XAML ist eine XML-basierte Beschreibungssprache, die bei der WPF zur Erstellung von Benutzeroberflächen eingesetzt wird. Obwohl mit dem Programmiermodell der WPF das Design der Benutzeroberfläche auch allein in C# erstellt werden kann, ist XAML die bevorzugte Art. Der ausschlaggebende Grund ist, dass XAML eine bessere Trennung von Benutzeroberflächendesign und -logik ermöglicht. Allerdings muss vorweg gesagt werden, dass Sie in XAML nichts tun können, was nicht auch in C# möglich ist. XAML ist lediglich ein Serialisierungsformat. Zur Laufzeit werden aus den in XAML angegebenen Elementen Objekte erzeugt. Warum XAML dennoch eingeführt wurde, erfahren Sie in Abschnitt 3.1, »<Warum>XAML?</Warum>«.

Die folgenden Abschnitte erklären, wie XML-Namespaces von XAML den CLR-Namespaces zugeordnet werden und wie Sie XAML um zusätzliche CLR-Namespaces erweitern. Sie erfahren, auf welche Arten Sie Properties in XAML setzen können, wie Type-Converter funktionieren und welche Markup-Extensions existieren. Darüber hinaus gehe ich auf Spracherweiterungen von XAML sowie auf die Klassen `XamlWriter` und `XamlReader` ein, mit denen Sie zur Laufzeit Objekte in eine XAML-Datei serialisieren und aus einer XAML-Datei deserialisieren können.

3.1 <Warum>XAML?</Warum>

Im Entwicklungsprozess einer Anwendung wird meist zu Beginn festgelegt, wie die fertig implementierte Benutzeroberfläche aussehen soll. Im traditionellen Entwicklungsprozess ist eine Person – die im Weiteren als »Designer« bezeichnet wird – dafür zuständig, das Design der Benutzeroberfläche mithilfe eines Werkzeugs festzulegen, zum Beispiel mit einem Grafikprogramm. Das Ergebnis ist ein Entwurf der Benutzeroberfläche, der in irgendeiner Form gespeichert wird, etwa als Bilddatei.

Die erstellten Dateien gibt der Designer an den Entwickler weiter. Dieser hat nun die Aufgabe, die Anwendungslogik zu implementieren und die Benutzeroberfläche gemäß dem Entwurf umzusetzen, den er vom Designer erhalten hat. Der Entwickler baut in Visual Studio die vom Designer entworfene Benutzeroberfläche nach. Dabei versucht er, das Design zu treffen, das vom Designer spezifiziert wurde.

Der Entwickler erledigt also nochmals die ganze Arbeit, die der Designer bereits getan hat. Dabei trifft er in der Praxis mit seinem Design der Benutzeroberfläche oft nicht die Vorstellungen des Designers. Insbesondere, wenn eine komplexe Benutzeroberfläche mit visuellen Effekten und Animationen entworfen wird, kann es zu vielen Missverständnissen zwischen Designer und Entwickler kommen. Dabei ist es die doppelte Erfassung des Designs der Benutzeroberfläche – einmal durch den Designer und ein zweites Mal durch den Entwickler –, die im Entwicklungsprozess zu Reibungen führt. Aufgrund unterschiedlicher Formate ist die doppelte Erfassung leider nicht vermeidbar. Ein gemeinsames Format würde Abhilfe schaffen.

Genau an dieser Stelle kommt XAML ins Spiel. Als deklarative Beschreibungssprache für Benutzeroberflächen verbessert XAML die Zusammenarbeit, indem es als Austauschformat dient.

Bei der Entwicklung mit der WPF erstellt der Designer die Benutzeroberfläche in einem Designwerkzeug. Aus diesem Designwerkzeug exportiert er XAML. Mittlerweile besitzen viele Anwendungen eine Exportfunktion für XAML, sei es standardmäßig oder über ein Plug-in. Beispielsweise lässt sich auch aus Adobe Illustrator XAML exportieren. Microsoft stellt für Designer unter anderem das Programm Blend zur Verfügung, das mit Visual Studio installiert wird. In diesem Programm findet der Designer typische Grafikwerkzeuge wieder, mit denen er die Benutzeroberfläche erstellen und als XAML abspeichern kann.

Die vom Designer als XAML gespeicherte Benutzeroberfläche importiert der Entwickler in Visual Studio. Er muss die Benutzeroberfläche nicht wie im traditionellen Entwicklungsprozess nochmals erstellen, sondern kann direkt die XAML-Datei des Designers verwenden und diese nun mit der dazugehörigen Logik versehen, die er in C# implementiert.

In der Praxis hat sich auch oft der umgekehrte Weg bewährt. Dabei erstellt der Entwickler im ersten Schritt die Anwendung mit vollständiger Logik und einem rudimentären GUI. Die ganze Anwendung gibt er im zweiten Schritt an den Designer weiter. Dieser öffnet in seinem Design-Tool Blend das erhaltene Projekt (*.csproj* oder *.vbproj*). Der Designer verschönert in Blend das GUI mit neuen Templates für die Controls, fügt hier und da eine Animation ein, verbindet eventuell Teile der GUI mittels Data Binding mit der vom Entwickler implementierten Logik, erstellt Farbverläufe für den Hintergrund etc., bis er schließlich das gewünschte Design getroffen hat.

In weiteren Schritten öffnet der Entwickler das Projekt wieder in Visual Studio und führt seine Unit Tests durch, um zu sehen, ob der Designer nicht versehentlich seine Programmlogik manipuliert hat.

Tipp

Da sich mit Blend auch *.csproj*-Dateien öffnen lassen, ist es oft auch üblich, dass ein Entwickler Visual Studio und Blend parallel betreibt und in beiden Programmen dasselbe Projekt geöffnet hat. In Blend kann er dann auf einfache Weise Farbverläufe, Animationen oder Templates anpassen. Visual Studio merkt automatisch, dass sich die Dateien außerhalb von Visual Studio geändert haben, und lädt sie bei Bedarf neu.

Mit der Aufteilung in Designer und Entwickler hat sich für die Architektur von WPF-Anwendungen ein Pattern bewährt, das auf dem Model-View-Controller aufbaut. Das sogenannte Model-View-ViewModel-Pattern (MVVM) ermöglicht die beste Trennung von UI und Logik. In Abschnitt 9.6, »Das Model-View-ViewModel-Pattern (MVVM)«, finden Sie mehr Informationen zum MVVM-Pattern und eine kleine Beispielanwendung, die dieses Pattern einsetzt.

Außer seiner Funktion als Austauschformat zwischen Designer und Entwickler bietet XAML gegenüber der prozeduralen Programmierung einer Benutzeroberfläche weitere Vorteile. An dieser Stelle sind die wichtigsten:

- Sie benötigen weitaus weniger Code als in C#. Darüber hinaus lässt sich die Benutzeroberfläche einfacher und schneller implementieren, da XAML leicht lesbar und besser strukturierbar ist als prozeduraler Code.
- XAML ist ein vertrautes Konzept. Wenn Sie über Erfahrung mit HTML oder anderen webbasierten Beschreibungssprachen verfügen, werden Sie sich in XAML relativ schnell zurechtfinden. Allerdings möchte ich XAML nicht mit Sprachen wie HTML vergleichen. XAML ist im Grunde ein zusätzlicher Weg, um .NET-Objekte zu erzeugen.
- XAML ist erweiterbar (»extensible«). Sie können beispielsweise in XAML Objekte von Ihren eigenen Klassen erstellen.
- Neben Microsoft bieten viele Dritthersteller in ihren Programmen Unterstützung für XAML an. An dieser Stelle seien Adobe Illustrator und Zam3D von Electric Rain erwähnt.
- Eine XAML-Datei ist eine XML-basierte Repräsentation von .NET-Objekten. Es gibt somit keinen Performance-Nachteil gegenüber einer Implementierung der Benutzeroberfläche in C#. Jede .NET-Klasse können Sie in XAML verwenden, wenn die Klasse einen öffentlichen Default-Konstruktor (parameterlos) besitzt.
- Sie können eine XAML-Datei zur Laufzeit dynamisch laden, um die darin deklarierten Objekte in Ihrer Anwendung zu nutzen.

3.2 Elemente und Attribute

XAML basiert auf XML und besteht somit aus XML-Elementen und XML-Attributen. Wie XML-Dokumente müssen auch XAML-Dokumente wohlgeformt sein. Wohlgeformt ist eine

XAML-Datei genau dann, wenn sie einige Voraussetzungen erfüllt. Die wichtigsten Voraussetzungen sind:

- Eine XAML-Datei hat genau ein Wurzelelement, das alle anderen Elemente umschließt.
- Auf ein öffnendes Element folgt ein schließendes Element, wie die Überschrift von Abschnitt 3.1, »<Warum>XAML?</Warum>«, demonstriert. Ein leeres Element können Sie auch direkt mit einem / am Ende des Elements schließen, was so aussieht: `<Warum/>`
- Elemente müssen richtig verschachtelt sein. Wird innerhalb des Elements `<Button>` das Element `<Image>` geöffnet, muss das schließende Element `</Image>` vor dem schließenden Element `</Button>` stehen.

Listing 3.1 zeigt ein gültiges XAML-Dokument, das die obigen Voraussetzungen erfüllt:

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <Button Content="OK"/>
</Window>
```

Listing 3.1 Beispiele\K03\01 XAMLEinfuehrung\MainWindow.xaml

Aus XAML werden wie folgt Objekte erstellt:

- **Elemente werden .NET-Klassen zugeordnet** – Zur Laufzeit werden aus den XML-Elementen in XAML Objekte der entsprechenden .NET-Klassen erzeugt. Dazu müssen die .NET-Klassen zwingend einen öffentlichen Default-Konstruktor (parameterlos) besitzen. Da die XML-Elemente in XAML .NET-Klassen zugeordnet sind und aus ihnen Objekte dieser Klassen erzeugt werden, bezeichnet man sie auch als Objektelemente.
- **Attribute werden .NET-Properties und -Events zugeordnet** – In Listing 3.1 ist auf dem `Button`-Element das XML-Attribut `Content` gesetzt, das der `Content`-Property der `Button`-Klasse zugeordnet ist. Im vorherigen Kapitel wurde auf `Button`-Elementen das XML-Attribut `Click` gesetzt, das keiner Property, sondern dem Event `Click` der `Button`-Klasse zugeordnet ist. In der Codebehind-Datei wurde der entsprechende Event Handler erwartet. XML-Attribute können folglich entweder einer .NET-Property oder einem Event zugeordnet sein. Die zu .NET-Properties zugeordneten XML-Attribute werden auch als *Property-Attribut*, die zu .NET-Events zugeordneten XML-Attribute als *Event-Attribut* bezeichnet.

In Listing 3.1 wurde ein Objekt der Klasse `Window` erstellt, das eine `Button`-Instanz enthält. Das `Window`-Element bildet das Wurzelelement und umschließt das `Button`-Element. Im `Button`-Element wird mit dem Attribut `Content` die `Content`-Property des `Button`-Objekts gesetzt. Statt `<Button Content="OK"> </Button>` wird die abgekürzte Schreibweise `<Button Content="OK"/>` verwendet, die auch als *Empty-Element-Syntax* bezeichnet wird. Zum Setzen der `Content`-Property wäre auch `<Button>OK</Button>` möglich, doch dazu folgt mehr in Abschnitt 3.4, »Properties in XAML setzen«.

Der `XAML`-Code aus Listing 3.1 entspricht folgendem Code in C#:

```
var w = new System.Windows.Window();
var b = new System.Windows.Controls.Button();
b.Content = "OK";
w.Content = b;
```

Listing 3.2 Alternative Darstellung in C#

Hinweis

Da die in XAML definierten XML-Elemente .NET-Klassen zugeordnet werden, ist in XAML die korrekte Groß-/Kleinschreibung zwingend notwendig.

Sicherlich fragen Sie sich, wie aus den XML-Elementen in XAML Objekte von den richtigen .NET-Klassen erstellt werden können. Woher weiß der XAML-Parser, dass er aus einem `Button`-Element ein Objekt der Klasse `Button` aus dem Namespace `System.Windows.Controls` erstellen muss? Die Antwort liegt in den XML-Namespaces, die Sie in XAML auf dem Wurzelelement mit dem Attribut `xmlns` angeben.

3.3 Namespaces

Die Elemente in einem XAML-Dokument werden .NET-Klassen zugeordnet. Diese Zuordnung erfolgt dabei über gewöhnliche XML-Namespaces, die im Hintergrund auf CLR-Namespaces zeigen. Ein XAML-Dokument besitzt normalerweise zwei oder mehr XML-Namespaces: den XML-Namespace der WPF und den XML-Namespace von XAML. Definieren Sie weitere XML-Namespaces, die auf Ihre eigenen CLR-Namespaces zeigen, um eigene Klassen in XAML zu verwenden. In den anschließenden Abschnitten klären wir den Mythos um Namespaces und betrachten Folgendes:

- **Der XML-Namespace der WPF** – Dieser XML-Namespace wird mehreren CLR-Namespaces der WPF zugeordnet, wodurch Sie in XAML ohne Weiteres .NET-Klassen wie `Button` oder `Window` verwenden können.
- **Der XML-Namespace für XAML** – Er ist dem CLR-Namespace `System.Windows.Markup` zugeordnet und enthält zudem einige Compiler-Direktiven, wie das bereits bekannte `x:Class`-Attribut.
- **Über den Namespace-Alias** – Für XML-Namespaces lassen sich beliebige Aliasse vergeben. Lesen Sie diesen Abschnitt unbedingt, falls Sie mit XML nicht allzu vertraut sind.
- **XAML mit eigenen CLR-Namespaces erweitern** – Der letzte Teil zeigt, wie Sie in XAML auf eigene Klassen aus Ihren CLR-Namespaces zugreifen.

3.3.1 Der XML-Namespace der WPF

In Listing 3.1 wurden in XAML ein `Window`-Element und ein `Button`-Element deklariert. Zur Laufzeit werden aus diesen Elementen Objekte der Klassen `System.Windows.Window` und `System.Windows.Controls.Button` erstellt. Auf dem `Window` ist mit dem `xmlns`-Attribut der XML-Namespace `http://schemas.microsoft.com/winfx/2006/xaml/presentation` definiert. Das ist der XML-Namespace der WPF. Sie werden unter diesem URL keine Webseite finden. Es ist lediglich ein Namespace, der XML-Elemente eindeutig qualifiziert. Sie können das mit .NET-Namespaces vergleichen, die Klassen eindeutig qualifizieren. Aufgrund der Eindeutigkeit werden in XML und damit auch in XAML für Namespaces URLs verwendet.

Das Wurzelelement – im Fall von Listing 3.1 das `Window`-Element – muss in XAML über mindestens ein `xmlns`-Attribut verfügen, um sich selbst voll zu qualifizieren. Die in XAML deklarierten Elemente `Window` und `Button` aus Listing 3.1 werden durch das `xmlns`-Attribut auf dem Wurzelelement dem XML-Namespace der WPF zugeordnet.

Der XML-Namespace der WPF ist mit mehreren CLR-Namespaces verbunden, unter anderem auch mit den CLR-Namespaces `System.Windows` und `System.Windows.Controls`, in denen sich die Klassen `Window` und `Button` befinden. Die Verbindung des XML-Namespaces zu den CLR-Namespaces ist in den Assemblies der WPF hartkodiert.

Die Assemblies ordnen den XML-Namespace der WPF bestimmten CLR-Namespaces zu, indem sie auf Assembly-Ebene mehrere `XmlnsDefinitionAttribute`s (Namespace: `System.Windows.Markup`) definieren. Das `XmlnsDefinitionAttribute` enthält für die Zuordnung eine Property `ClrNamespace` und eine Property `XmlNamespace`, die beide vom Typ `string` sind. Die Zuordnung von XML- zu CLR-Namespaces wird als Namespace-Mapping bezeichnet.

Der XAML-Parser durchsucht zur Erstellung der Objekte alle von Ihrem Projekt referenzierten Assemblies nach Attributen vom Typ `XmlnsDefinitionAttribute`, deren `XmlNamespace`-Property den XML-Namespace der WPF enthält. Von jeder gefundenen `XmlnsDefinitionAttribute`-Instanz wird die `ClrNamespace`-Property ausgelesen.

In den gefundenen CLR-Namespaces sucht der XAML-Parser die entsprechende Klasse und erstellt ein Objekt dieser Klasse. Im Fall des `Window`s und des `Button`s aus Listing 3.1 findet der XAML-Parser die beiden Klassen in den CLR-Namespaces `System.Windows` (`Window`) und `System.Windows.Controls` (`Button`) und erstellt von diesen Klassen die Objekte.

Listing 3.3 zeigt einen Ausschnitt einer kleinen Konsolenanwendung, die mit Reflection alle `XmlnsDefinitionAttribute`s ausliest, die auf den Assemblies *PresentationCore*, *PresentationFramework*, *WindowsBase* und *WindowsFormsIntegration* gesetzt sind. Die CLR-Namespaces, die dem XML-Namespace der WPF zugeordnet sind, werden zu einer Liste hinzugefügt und an der Konsole ausgegeben.

```
var assemblies = new Assembly[4];
assemblies[0] = typeof(UIElement).Assembly;        // PresentationCore
assemblies[1] = typeof(FrameworkElement).Assembly; // PresentationFramework
```



```
assemblies[2] = typeof(Dispatcher).Assembly;       // WindowsBase
assemblies[3] = typeof(WindowsFormsHost).Assembly; // WindowsFormsIntegration
var list = new List<string>();
foreach (var assembly in assemblies)
  foreach (object o in assembly.GetCustomAttributes(false))
    if (o is System.Windows.Markup.XmlnsDefinitionAttribute)
    {
      var xmlnsAttr = o as XmlnsDefinitionAttribute;
      if (xmlnsAttr.XmlNamespace.Equals(
        "http://schemas.microsoft.com/winfx/2006/xaml/presentation")
        && !list.Contains(xmlnsAttr.ClrNamespace))
          list.Add(xmlnsAttr.ClrNamespace);
    }
list.Sort();
foreach (string s in list)
  Console.WriteLine(s);
```

Listing 3.3 Beispiele\K03\02 XAMLtoCLRNamespaces\Program.cs

Die Konsolenausgabe aus Listing 3.3 zeigt die gemappten CLR-Namespaces, die Sie in dem folgenden Hinweis-Kasten finden.

Hinweis

Der XML-Namespace der WPF, `http://schemas.microsoft.com/winfx/2006/xaml/presentation`, wird folgenden CLR-Namespaces zugeordnet:

```
System.Diagnostics
System.Windows
System.Windows.Automation
System.Windows.Controls
System.Windows.Controls.Primitives
System.Windows.Data
System.Windows.Documents
System.Windows.Forms.Integration
System.Windows.Ink
System.Windows.Input
System.Windows.Media
System.Windows.Media.Animation
System.Windows.Media.Effects
System.Windows.Media.Imaging
System.Windows.Media.Media3D
System.Windows.Media.TextFormatting
System.Windows.Navigation
```

`System.Windows.Shapes`
`System.Windows.Shell`

Da die Beziehung zwischen dem XML-Namespace der WPF und den zugeordneten CLR-Namespaces eine 1:n-Beziehung ist, mussten die Entwickler der WPF strikt darauf achten, dass die Klassennamen über alle zugeordneten CLR-Namespaces hinweg eindeutig sind. Ansonsten könnte ein Objektelement in XAML nicht eindeutig einer Klasse zugeordnet werden. Daher gibt es in den CLR-Namespaces, die dem XML-Namespace der WPF zugeordnet sind, beispielsweise nur im Namespace `System.Windows.Controls` eine `Button`-Klasse.

3.3.2 Der XML-Namespace für XAML

Auf einem XML-Element können Sie beliebig viele weitere XML-Namespaces deklarieren. Allerdings muss jeder weitere XML-Namespace ein Alias besitzen. Standardmäßig besitzt ein XAML-Dokument noch einen zweiten XML-Namespace auf dem Wurzelelement: den XML-Namespace für XAML. Dieser hat üblicherweise ein `x` als Alias:

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

Der XML-Namespace für XAML ist mit dem `XmlnsDefinitionAttribute` dem CLR-Namespace `System.Windows.Markup` zugeordnet. Neben den Klassen aus diesem CLR-Namespace verfügt der XML-Namespace von XAML über spezielle Direktiven für den XAML-Compiler und -Parser. Diese Direktiven werden auch *XAML-Spracherweiterungen* genannt und sehen im XAML-Dokument auf den ersten Blick wie Properties auf Objektelementen aus. Tatsächlich sind es aber keine Properties, sondern eben Instruktionen für den XAML-Compiler und -Parser.

Ein Beispiel für eine solche Spracherweiterung ist das `x:Class`-Attribut, das im vorherigen Kapitel bereits verwendet wurde. Das `x:Class`-Attribut legt in XAML auf dem Wurzelelement den Namen der partiellen Klasse fest, die in die generierte Datei (*g.cs*) geschrieben wird. Die Klasse in der generierten Datei wird beim Kompilieren mit der partiellen Klasse der Codebehind-Datei verbunden. Um das Attribut `Class` dem XAML-Namespace zuzuordnen, muss es mit dem Alias dieses Namespaces versehen werden. Das Alias für den XML-Namespace von XAML ist üblicherweise ein `x`, somit müssen Sie – wie es im vorherigen Kapitel bereits geschehen ist – `x:Class` schreiben, um das `Class`-Attribut diesem Namespace zuzuordnen. In Abschnitt 3.7, »XAML-Spracherweiterungen«, lernen Sie weitere Spracherweiterungen von XAML kennen.

Hinweis

Wenn Sie in den folgenden Listings in diesem Buch einen Ausschnitt einer XAML-Datei sehen, der das Wurzelelement nicht enthält, können Sie immer davon ausgehen, dass auf dem Wurzelelement der XML-Namespace der WPF als Default-Namespace (ohne Alias) und der XML-Namespace von XAML mit dem Alias `x` definiert sind.

3.3.3 Über den Namespace-Alias

Der XML-Namespace der WPF ist in einem XAML-Dokument immer der Default-Namespace. Der Default-Namespace ist jener, der kein Alias besitzt. Dies ist sinnvoll, da der Großteil der Elemente in einer XAML-Datei aus dem XML-Namespace der WPF stammt und somit nicht mit einem Alias als Präfix versehen werden muss. Dennoch können Sie natürlich auch für den XML-Namespace der WPF ein beliebiges Alias setzen, was der Ausschnitt aus Listing 3.4 zeigt:

```
<WPF:Window x:Class="AliasFuerWPFxmlns.MainWindow"
  xmlns:WPF="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <WPF:StackPanel>
    <WPF:Button Content="OK"/>
    <WPF:TextBox Text="Hallo"/>
  </WPF:StackPanel>
</WPF:Window>
```

Listing 3.4 Beispiele\K03\03 AliasFuerWPFxmlns\MainWindow.xaml

Elemente, die kein Präfix besitzen, sind immer dem Default-Namespace zugeordnet, der einfach ohne Alias mit `xmlns=...` gesetzt wird. Wie Listing 3.4 zeigt, ordnen Sie ein Element einem anderen Namespace als dem Default-Namespace zu, indem Sie folgende Syntax verwenden:

```
<Alias:Element/>
```

3.3.4 XAML mit eigenen CLR-Namespaces erweitern

Das »X« in XAML steht wie bereits erwähnt für *extensible* (»erweiterbar«). In XAML sind Sie also nicht auf die Elemente in den XML-Namespaces der WPF und XAML beschränkt. Sie können weitere XML-Namespaces hinzufügen, die auf zusätzliche CLR-Namespaces zeigen – das bereits bekannte Namespace-Mapping. Mit einem Namespace-Mapping lässt sich in XAML jede beliebige .NET-Klasse instanziieren, die einen öffentlichen Default-Konstruktor besitzt.

Hinweis

Für generische Klassen (Generics) besteht leider bisher keine Möglichkeit, diese in XAML zu instanziieren. Lediglich für Wurzelelemente existiert mit der Spracherweiterung `x:TypeArguments` eine Option.

Um einen zusätzlichen CLR-Namespace in XAML zu nutzen, definieren Sie üblicherweise auf dem Wurzelelement ein weiteres `xmlns`-Attribut mit einem beliebigen, allerdings eindeutigen Alias. Als XML-Namespace geben Sie keinen URL an, sondern eine spezielle Syntax, die

der XAML-Parser versteht. Aus der speziellen Syntax erkennt der XAML-Parser den zugeordneten CLR-Namespace wie auch die Assembly, die den CLR-Namespace enthält. Die Syntax sieht wie folgt aus:

```
xmlns:mn="clr-namespace:MeinCLRNamespace;assembly=MeineAssembly"
```

`mn` ist das frei wählbare Alias für den zugeordneten CLR-Namespace. Objekte aus dem CLR-Namespace `MeinCLRNamespace` lassen sich in XAML nun mit Objektelementen in der Form von `<mn:Klassenname>` erstellen. Im Namespace-Mapping geben Sie als Assembly lediglich den Namen der Assembly an, keinen Pfad und keine Dateierweiterung. Die Assembly muss sich natürlich in den Projektverweisen befinden. Beachten Sie auch, dass Sie für die Angabe des CLR-Namespaces einen »:« verwenden und für die Angabe der Assembly ein »=«. Das Folgende ist ein Beispiel für ein Mapping des `System`-Namespaces, der in .NET Core in der Assembly `System.Runtime` liegt:

```
xmlns:sys="clr-namespace:System;assembly=System.Runtime"
```

Hinweis

Im .NET Framework liegt der `System`-Namespace mit Klassen wie `String` nicht in der Assembly `System.Runtime`, sondern in der Assembly `mscorlib`. Das heißt, die .NET-Framework-Variante für das Namespace-Mapping sieht wie folgt aus:

```
xmlns:sys="clr-namespace:System;assembly=mscorlib"
```

Mit der Zuordnung des `System`-Namespaces aus der Assembly `System.Runtime` lassen sich in XAML nun Objekte der Klassen aus dem `System`-Namespace erstellen. Dazu nutzen Sie für die Objektelemente das gewählte Alias als Präfix. Im Namespace-Mapping oben ist dies das Alias `sys`.

Wenn Sie im Namespace-Mapping einen CLR-Namespace aus dem Projekt verwenden, in dem auch Ihre XAML-Datei definiert ist, verzichten Sie im `xmlns`-Attribut auf die Angabe von `assembly=...` und den Strichpunkt hinter dem angegebenen CLR-Namespace. Folgende XAML-Datei mappt den `System`-Namespace und auch den CLR-Namespace des Projekts, in dem sich die XAML-Datei befindet (`CustomNamespacesInXAML`):

```
<Window x:Class="CustomNamespacesInXAML.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:sys="clr-namespace:System;assembly=System.Runtime"
xmlns:local="clr-namespace:CustomNamespacesInXAML">
  <ListBox>
    <sys:String>Ein einfacher String</sys:String>
    <local:RepeatString Repeats="2" StringToRepeat="Hallo"/>
```



```
  </ListBox>
</Window>
```

Listing 3.5 Beispiele\K03\04 CustomNamespacesInXAML\MainWindow.xaml

In Listing 3.5 wird im `Window` eine `ListBox` erstellt. Die `ListBox` wird mit einem `String`-Objekt und einem Objekt der Klasse `RepeatString` gefüllt. `RepeatString` liegt im Namespace `CustomNamespacesInXAML`. Dieser CLR-Namespace ist Teil des aktuellen Projekts, wodurch beim Namespace-Mapping auf die Angabe der Assembly verzichtet wird. Auf dem `RepeatString`-Objekt werden die Properties `Repeats` und `StringToRepeat` gesetzt. Die Klasse `RepeatString` sieht wie folgt aus (siehe Listing 3.6):

```
public class RepeatString
{
  public string StringToRepeat { get; set; }
  public int Repeats { get; set; }
  public override string ToString()
  {
    StringBuilder sb = new StringBuilder();
    sb.Append(StringToRepeat);
    for (int i = 0; i < Repeats; i++)
      sb.Append(StringToRepeat);
    return sb.ToString();
  }
}
```

Listing 3.6 Beispiele\K03\04 CustomNamespacesInXAML\RepeatString.cs

Beachten Sie in der Klasse `RepeatString` die Properties, die genau den Attributen entsprechen, die in Listing 3.5 in XAML auf dem `RepeatString`-Element gesetzt wurden. Die Klasse `RepeatString` überschreibt die Methode `ToString`. In der `ToString`-Methode wird der in der `StringToRepeat`-Property gespeicherte Wert in einer `for`-Schleife so oft zu einem `StringBuilder`-Objekt hinzugefügt, bis der `Repeats`-Wert erreicht ist. Der im `StringBuilder`-Objekt enthaltene `String` wird als Ergebnis der `ToString`-Methode zurückgegeben.

Hinweis

In der Klassenhierarchie in Kapitel 2, »Das Programmiermodell«, wurde bereits erwähnt, dass alles in der WPF Sichtbare ein `Visual` ist. Von `Visual` leitet `UIElement` ab. Für alle Objekte, die nicht vom Typ `UIElement` sind und zu einem `Window` oder (wie in Listing 3.5) zu einer `Listbox` hinzugefügt werden, wird die `ToString`-Methode aufgerufen. Der zurückgegebene Wert wird in einem automatisch erstellten `TextBlock`-Objekt angezeigt.

In Listing 3.5 wird somit für das `String`- wie auch für das `RepeatString`-Objekt ein `TextBlock`-Objekt erstellt, das den Rückgabewert der Methode `ToString` anzeigt.

Das in Listing 3.5 deklarierte `Window`-Objekt zeigt den Inhalt wie vermutet an. Das ebenfalls in XAML erstellte `RepeatString`-Objekt wiederholt den angegebenen String `Hallo` zweimal, was zu der Ansicht aus Abbildung 3.1 führt.

Abbildung 3.1 Das Window-Objekt aus Listing 3.5 in Aktion

Das Beispiel zeigt, dass sich weitere Assemblies und CLR-Namespaces leicht in XAML einbinden lassen.

Tipp

Falls Sie beispielsweise ein Framework entwickeln, können Sie auf den Assemblies Ihres Frameworks natürlich auch `XmlnsDefinitionAttribute`s definieren, die XML-Namespaces CLR-Namespaces zuordnen. Referenziert der Benutzer Ihres Frameworks Ihre Assemblies, muss er nur noch den XML-Namespace angeben. So ist das ja auch bei den Klassen der WPF der Fall.

Obwohl die `xmlns`-Attribute üblicherweise auf dem Wurzelelement erstellt werden, können Sie auch auf Kindelementen `xmlns`-Attribute setzen. Das in Listing 3.7 deklarierte `Window`-Objekt ist analog zu dem in Abbildung 3.1 dargestellten.

```
<Window x:Class="CustomNamespacesInXAML.MainWindow"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml">
  <ListBox>
    <String xmlns="clr-namespace:System;assembly=System.Runtime">
      Ein einfacher String
    </String>
    <local:RepeatString
      xmlns:local="clr-namespace:CustomNamespacesInXAML"
      Repeats="2" StringToRepeat="Hallo"/>
  </ListBox>
</Window>
```

Listing 3.7 Beispiele\K03\05 CustomNamespacesInXAML2\MainWindow.xaml

Genauso wie ein Wurzelelement können Sie auch jedes Objektelement mit einem `xmlns`-Attribut voll qualifizieren, wie dies in Listing 3.7 zu sehen ist. Allerdings ist es in der Praxis die übliche, übersichtlichere und auch elegantere Vorgehensweise, alle `xmlns`-Attribute auf dem Wurzelelement mit entsprechenden Aliassen zu deklarieren.

Hinweis

Sinnvoll ist die Definition eines XML-Namespaces ohne Alias (`xmlns="..."`) auf einem Objektelement anstelle eines XML-Namespaces mit Alias (`xmlns:alias="..."`) auf dem Wurzelelement genau dann, wenn das Objektelement viele weitere Objektelemente enthält, die im gleichen XML-Namespace wie das Objektelement selbst liegen. Dadurch können Sie sich einige Aliasse als Präfix auf den Objektelementen sparen. Würde in einem solchem Fall der XML-Namespace auf dem Wurzelelement mit dem Alias angelegt, müssten alle Elemente in diesem Objektelement mit einem Alias versehen werden.

3.4 Properties in XAML setzen

Wird auf einem Element in XAML ein XML-Attribut definiert, wird dieses entweder einer Property oder einem Event zugeordnet. Mit einem XML-Attribut lässt sich also eine Property setzen. XAML bietet neben dieser sogenannten Attribut-Syntax weitere Möglichkeiten, Properties zu setzen. In diesem Abschnitt sehen wir uns die folgenden Möglichkeiten an:

- **die Attribut-Syntax** – Auf einem Element wird ein XML-Attribut definiert, das die .NET-Property setzt.
- **die Property-Element-Syntax** – Es wird speziell für eine Property ein eigenes XML-Element erstellt. Dieses XML-Element ist jetzt kein Objektelement, sondern ein Property-Element, da es einer Property zugeordnet wird. Diese Syntax erlaubt es, einer Property ein komplexes Objekt zuzuweisen.
- **die Content-Property** – Auf Klassen kann mit dem `ContentPropertyAttribute` eine Default-Property angegeben werden. Diese Default-Property wird gesetzt, wenn sich etwas in einem Objektelement befindet.
- **die Attached-Property-Syntax** – Mit dieser Syntax werden Attached Properties gesetzt. Dies sind Properties, die in einer Klasse definiert und auf Objekten anderer Klassen gesetzt werden.

3.4.1 Die Attribut-Syntax

Properties wurden in diesem Kapitel bereits auf verschiedenen in XAML erstellten Objekten gesetzt. Auf einem Objektelement definieren Sie zum Setzen einer Property ein Attribut mit dem Namen der Property. Diesem Attribut weisen Sie den gewünschten Wert zu, den Sie in Anführungszeichen setzen. Auf folgendem Button setzen Sie die `Content`-Property auf »OK«. Dazu verwenden Sie auf dem `Button`-Element ein `Content`-Attribut:

```
<Button Content="OK"/>
```

Die Syntax, die oben zum Setzen der `Content`-Property verwendet wurde, wird als Attribut-Syntax bezeichnet. Wie an der Attribut-Syntax zu erkennen ist, lässt sich der `Content`-Property ein `String`-Wert zuweisen. Wenn die Property einen primitiven Typ (`bool`, `int`, `double`, `float`, `char` etc.) oder einen Aufzählungswert verlangt, wird der in der Attribut-Syntax angegebene String automatisch in den entsprechenden Typ gecastet. Wollen Sie in XAML einer Property keinen primitiven Typ und keinen Aufzählungswert, sondern ein »richtiges« Objekt zuweisen, ist dies mit der Attribut-Syntax nicht ohne Weiteres möglich. Verwenden Sie stattdessen die Property-Element-Syntax.

Hinweis

Um ein Objekt mit der Attribut-Syntax zuzuweisen, gibt es einerseits noch die Möglichkeit der (später beschriebenen) Markup-Extensions und andererseits die Type-Converter. Type-Converter konvertieren den im XAML-Attribut angegebenen String in das von der Property benötigte Objekt. Dadurch können Sie beispielsweise der `Background`-Property eines `Button`-Elements mit der Attribut-Syntax den String `Blue` zuweisen. Im Hintergrund erstellt ein Type-Converter automatisch einen `SolidColorBrush`, der der `Background`-Property des `Button`-Objekts zugewiesen wird. Der Button wird somit blau dargestellt.

3.4.2 Die Property-Element-Syntax

Um einer Property in XAML ein komplexes Objekt zuzuweisen, das sich mit der Attribut-Syntax nicht zuweisen lässt, verwenden Sie die Property-Element-Syntax. Wie der Name der Property-Element-Syntax schon vermuten lässt, erstellen Sie zum Setzen einer Property anstelle eines XML-Attributs ein eigenständiges XML-Element. Dieses XML-Element erzeugt kein Objekt, wie dies die bisherigen als Objektelement bezeichneten XML-Elemente in XAML tun. Stattdessen verweist dieses XML-Element auf eine Property eines Objekts. Es wird somit Property-Element genannt. Folgender Code-Ausschnitt erstellt einen Button und setzt den Wert der `Content`-Property des `Button`-Objekts mit der Property-Element-Syntax:

```
<Button>
  <Button.Content>
    OK
 </Button.Content>
</Button>
```

Ein Property-Element besteht aus dem Namen der Klasse und der eigentlichen Property. Klassenname und Property sind durch einen Punkt getrennt. Das allgemeingültige Format der Property-Element-Syntax lautet demzufolge `<Klassenname.Propertyname>`.

Hinweis

Am Anfang dieses Kapitels wurde beschrieben, dass in XAML Elemente .NET-Klassen zugeordnet werden. Attribute werden .NET-Properties und -Events zugeordnet. Die Property-Element-Syntax bildet eine Ausnahme: Bei ihr wird das Property-Element einer .NET-Property und nicht einer Klasse zugeordnet.

Hinweis

Auf einem Property-Element können Sie keine XML-Attribute setzen. XML-Attribute verweisen in XAML immer auf Properties oder Events eines Objekts. Ein Property-Element definiert allerdings kein Objekt, sondern lediglich eine Property. Folglich kann ein Property-Element keine XML-Attribute besitzen.

Die Property-Element-Syntax erlaubt es, einer Property ein komplexes Objekt zuzuweisen und nicht nur primitive Typen oder Aufzählungswerte. Folgende Loose-XAML-Datei weist unter Verwendung der Property-Element-Syntax der `Content`-Property eines Buttons ein `Image`-Objekt und der `Background`-Property einen `LinearGradientBrush` zu:

```
<Button>
  <Button.Content>
    <Image Source="thomas.png" Height="120" Margin="5"/>
  </Button.Content>
  <Button.Background>
    <LinearGradientBrush>
      <GradientStop Offset="0" Color="Black"/>
      <GradientStop Offset="1" Color="White"/>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

Listing 3.8 Beispiele\K03\06 PropertyElementSyntax\MainWindow.xaml

Abbildung 3.2 zeigt den in Listing 3.8 erstellten Button.

Abbildung 3.2 Ein Button, dessen Content-Property ein Image und dessen Background-Property einen LinearGradientBrush enthält

3.4.3 Die Content-Property (Default-Property)

Um die Darstellung von XAML etwas kompakter zu machen, definieren viele Klassen der WPF eine Property, die in XAML per Default gesetzt wird, wenn sich etwas direkt innerhalb eines Objektelements befindet und nicht explizit über die Property-Element-Syntax einer Property zugeordnet ist. Die `TextBox`-Klasse definiert die `Text`-Property als Default-Property. Um die `Text`-Property einer Textbox zu setzen, können Sie somit auch auf die Property-Element-Syntax verzichten. Zusammen mit der Attribut-Syntax haben Sie drei Möglichkeiten, die `Text`-Property einer Textbox zu setzen:

```
<!-- Normale Attribut-Syntax -->
<TextBox Text=".NET rockt"/>
<!-- Property-Element-Syntax -->
<TextBox>
  <TextBox.Text>
    .NET rockt
  </TextBox.Text>
</TextBox>
<!-- Implizite Verwendung der Text-Property (Content-Property)-->
<TextBox>
  .NET rockt
</TextBox>
```

Listing 3.9 Beispiele\K03\07 ContentProperty\MainWindow.xaml

Die per Default gesetzte Property wird als Content-Property bezeichnet. Zum Festlegen der Content-Property wird auf Klassenebene das Attribut `ContentPropertyAttribute` (Namespace: `System.Windows.Markup`) deklariert, dessen Konstruktor einen `string` mit dem Namen der gewünschten Default-Property entgegennimmt. Die Klasse `TextBox` setzt die `Text`-Property als Content-Property:

```
[ContentProperty("Text")]
public class TextBox:TextBoxBase,...
```

Beachten Sie in Listing 3.9 beim untersten `TextBox`-Objekt, dass zum Setzen der `Text`-Property die Property-Element-Syntax einfach weggelassen wird. Der XAML-Parser sucht folglich auf der `TextBox`-Klasse nach dem Attribut `ContentProperty`, findet im `ContentProperty`-Attribut die `Text`-Property und weist den nicht in einem Property-Element angegebenen Wert innerhalb des `TextBox`-Elements der `Text`-Property zu.

Von der Content-Property der Klassen `Window` und auch `ListBox` wurde in diesem Kapitel bereits implizit Gebrauch gemacht. Die Content-Property von `Window` heißt – rein zufälligerweise – `Content`, die der Klasse `ListBox` heißt `Items`. Ohne die Content-Property müssten Sie zum Zuweisen von Objekten fast immer die Property-Element-Syntax verwenden, wodurch Ihre XAML-Dokumente bestimmt einige Zeilen größer wären.

Die Content-Property der Klasse `Button` heißt `Content`, also so wie die Content-Property der Klasse `Window`. Das ist nicht verwunderlich, denn beide Klassen erben diese Property von `ContentControl`. Auf der Klasse `ContentControl` ist auch das `ContentProperty`-Attribut mit der Property `Content` definiert. Den Button aus Listing 3.8 könnten Sie somit auch so wie in Listing 3.10 mit zwei Zeilen weniger erstellen (verzichten Sie dazu auf die Property-Elemente `<Button.Content>` und `</Button.Content>`):

```
<Button>
  <Image Source="thomas.png" Height="120" Margin="5"/>
  <Button.Background>
    <LinearGradientBrush>
      <GradientStop Offset="0" Color="Black"/>
      <GradientStop Offset="1" Color="White"/>
    </LinearGradientBrush>
  </Button.Background>
</Button>
```

Listing 3.10 Beispiele\K03\08 ContentPropertyButton\MainWindow.xaml

Das `Image`-Objekt wird in Listing 3.10 implizit der `Content`-Property des `Button`-Objekts zugewiesen.

3.4.4 Die Attached-Property-Syntax

Für die Attached Properties, eine spezielle Implementierung der in Kapitel 7 beschriebenen Dependency Properties, besitzt XAML eine eigene Syntax. Das Besondere an Attached Properties ist, dass sie in einer Klasse definiert, aber auf Objekten anderer Klassen gespeichert werden.

Diese Möglichkeit wird insbesondere bei Layout-Panels genutzt. In Listing 3.11 werden die Attached Properties `Left` und `Top` der `Canvas`-Klasse verwendet, um einen Button in einem Canvas absolut anzuordnen:

```
<Canvas Width="300" Height="300">
  <Button Canvas.Left="10" Canvas.Top="40" Content="OK"/>
</Canvas>
```

Listing 3.11 Beispiele\K03\09 AttachedProperties\MainWindow.xaml

Die C#-Variante des XAML-Ausschnitts aus Listing 3.11 benötigt etwas mehr Code (siehe Listing 3.12):

```
var c = new Canvas();
c.Width = 300;
c.Height = 300;
```

```
var b = new Button();
b.Content = "OK";
Canvas.SetLeft(b, 10);
Canvas.SetTop(b, 40);
c.Children.Add(b);
```

Listing 3.12 C#-Variante zu Listing 3.11

In Kapitel 6, »Layout«, werden Sie die Attached-Property-Syntax intensiv einsetzen. In Kapitel 7, »Dependency Properties«, erfahren Sie, wie Sie eigene Attached Properties implementieren.

3.5 Type-Converter

XAML-Dokumente sind oft weitaus kompakter als C#-Code. Ein Grund dafür ist unter anderem die Verwendung von Type-Convertern. Wird ein XAML-Dokument geparst, werden die mit der Attribut-Syntax angegebenen Strings für Properties mit primitiven Typen oder Aufzählungen automatisch gecastet. Für Objekte anderer Typen ist die Verwendung der Property-Element-Syntax notwendig, es sei denn, für den Typ existiert ein Type-Converter oder Sie verwenden eine Markup-Extension. Letztere stelle ich Ihnen im nächsten Abschnitt vor.

Type-Converter konvertieren den in einem XML-Attribut angegebenen String in das Objekt, das die dahinterliegende .NET-Property benötigt. Anstelle der Property-Element-Syntax ist es somit möglich, ein Objekt direkt mit der Attribut-Syntax einer Property zuzuweisen.

Ein Type-Converter ist ein Objekt der Klasse `TypeConverter` (Namespace: `System.ComponentModel`) beziehungsweise einer Subklasse. Die Klasse `TypeConverter` existiert bereits seit .NET 1.0 und enthält unter anderem die virtuelle Methode `ConvertFrom`, die sich in Subklassen überschreiben lässt. Die `ConvertFrom`-Methode gibt den konvertierten Wert zurück.

In XAML wandelt ein Type-Converter den mittels Attribut-Syntax angegebenen String in den Typ der Property um. XAML-Code wirkt dadurch weitaus kompakter als C#-Code, da die Angabe eines einfachen Strings in der Attribut-Syntax zur Erzeugung eines eventuell auch komplexeren Objekts genügt. Die Property-Element-Syntax ist nicht erforderlich.

Allerdings verursachen Type-Converter insbesondere bei XAML-Einsteigern den Eindruck, dass XAML fast magisch funktioniere. Daher entmystifizieren wir im Folgenden das Spiel der Type-Converter in XAML und betrachten folgende Punkte:

- **vordefinierte Type-Converter** – Es gibt bei der WPF bereits zahlreiche Type-Converter, wodurch XAML kompakt erscheint.
- **eigene Type-Converter implementieren** – Um die Funktionsweise von Type-Convertern zu verstehen, implementieren wir einen eigenen Type-Converter.

- **Type-Converter in C# verwenden** – Natürlich lassen sich Type-Converter auch in C# verwenden. Dieser Abschnitt zeigt, wie's geht.

3.5.1 Vordefinierte Type-Converter

Die `Button`-Klasse erbt von der Klasse `FrameworkElement` die Property `Margin`. Die `Margin`-Property gibt den Rand eines `FrameworkElements` an und ist vom Typ `Thickness` (Namespace: `System.Windows`).

In XAML setzen Sie die `Margin`-Property eines Buttons wie folgt:

```
<Button Margin="10" Content="OK"/>
```

Listing 3.13 Beispiele\K03\10 ThicknessTypeConverter\MainWindow.xaml

Die C#-Variante zur Erstellung eines gleichen `Button`-Objekts sehen Sie in Listing 3.14:

```
var button = new Button
{
  Margin = new Thickness(10),
  Content = "OK"
};
```

Listing 3.14 Beispiele\K03\10 ThicknessTypeConverter\MainWindow.xaml.cs

Der String `10` wird in XAML demnach automatisch in ein `Thickness`-Objekt konvertiert. Wie ist dies möglich? Zur Klasse `Thickness` existiert eine Klasse `ThicknessConverter`, die aus einem String ein `Thickness`-Objekt erzeugen kann. Die Verbindung der `Thickness`-Klasse zur Klasse `ThicknessConverter` ist über das `TypeConverterAttribute` definiert, das auf der Klasse `Thickness` gesetzt ist. Mehr zum `TypeConverterAttribute` lesen Sie im nächsten Abschnitt. Die XAML-Syntax zum Setzen der `Margin`-Property müsste ohne einen Type-Converter wie folgt aussehen:

```
<Button Content="OK">
  <Button.Margin>
    <Thickness Bottom="10"
               Left="10"
               Right="10"
               Top="10"/>
  </Button.Margin>
</Button>
```

Listing 3.15 Beispiele\K03\10 ThicknessTypeConverter\MainWindow.xaml

Das Erzeugen des `Thickness`-Objekts in Listing 3.15 ist allerdings auch nur möglich, da für alle primitiven Typen und Aufzählungen standardmäßig bereits Type-Converter existieren. Nur dadurch können die Strings `10` in `double`-Werte umgewandelt und den Properties `Bottom`, `Left`, `Right` und `Top` des `Thickness`-Objekts zugewiesen werden.

Type-Converter erhöhen also nicht nur die Lesbarkeit von XAML, indem sie weitaus kompakteren Code ermöglichen, sondern sind auch die Grundlage, um in XAML überhaupt Objekte mit primitiven Typen und Aufzählungen erstellen zu können.

[!]

Achtung

Die Type-Converter greifen nur, wenn die Property auch tatsächlich von dem entsprechenden Typ ist. Wollen Sie einer Property vom Typ der Aufzählung `Visibility` einen Wert zuweisen, genügt es, diesen Wert als String anzugeben. Wollen Sie einer Property vom Typ `object` einen Wert der Aufzählung `Visibility` zuweisen, müssen Sie den Wert explizit mit der `x:Static`-Markup-Extension aus der `Visibility`-Aufzählung holen, damit er nicht als String interpretiert wird:

```
... PropertyVomTypObject="{x:Static Visibility.Visible}" ...
```

Die WPF besitzt viele vordefinierte Type-Converter. Beispielsweise gibt es Type-Converter für die Klassen `Brush` oder `Color`, die Sie in Kapitel 13, »2D-Grafik«, kennenlernen werden.

3.5.2 Eigene Type-Converter implementieren

Im Folgenden werden Sie anhand eines kleinen Beispiels lernen, wie Sie einen eigenen `TypeConverter` implementieren. Dazu erstellen wir zunächst eine Klasse `Address`, die die Properties `Street`, `StreetNumber`, `ZipCode` und `City` enthält (siehe Listing 3.16). Alle Properties haben primitive Typen.

```
public class Address
{
  public string Street { get; set; }
  public string StreetNumber { get; set; }
  public int ZipCode { get; set; }
  public string City { get; set; }
}
```

Listing 3.16 Beispiele\K03\11 SimpleTypeConverter\Address.cs

Neben der Klasse `Address` wird die Klasse `Friend` erstellt, die zwei Properties besitzt: `Name` und `Address` (siehe Listing 3.17). Die Property `Address` ist vom Typ der in Listing 3.16 dargestellten `Address`-Klasse und kann folglich nicht mehr einfach über die Attribut-Syntax gesetzt werden. Da zum Test in XAML `Friend`-Objekte erstellt werden, wird in der `Friend`-Klasse neben

der Definition der Properties `Name` und `Address` die `ToString`-Methode überschrieben. Dadurch lassen sich in XAML erstellte `Friend`-Objekte in einem `Window`-Objekt anzeigen.


```
public class Friend
{
  public string Name { get; set; }
  public Address Address { get; set; }
  public override string ToString()
  {
    if (Address != null)
      return string.Format("Name: {0}\nAdresse: {1} {2} {3} {4}"
                    , Name
                    , Address.Street
                    , Address.StreetNumber
                    , Address.ZipCode
                    , Address.City);
    return Name;
  }
}
```

Listing 3.17 Beispiele\K03\11 SimpleTypeConverter\Friend.cs

In Listing 3.18 wird in XAML innerhalb eines `Window`-Objekts ein `Friend`-Objekt mit einer Adresse erzeugt. Zum Setzen der `Address`-Property muss die Property-Element-Syntax verwendet werden.

```
<Window ...
xmlns:local="clr-namespace:SimpleTypeConverter" ...>
...
  <local:Friend Name="Jimmy Johnson">
    <local:Friend.Address>
      <local:Address Street="Milchstrasse"
                     StreetNumber="8"
                     ZipCode="8553"
                     City="Mooncity"/>
    </local:Friend.Address>
  </local:Friend>
</Window>
```

Listing 3.18 Beispiele\K03\11 SimpleTypeConverter\MainWindow.xaml

Abbildung 3.3 zeigt das in Listing 3.18 erzeugte Fenster mit dem `Friend`-Objekt als Inhalt. Das Ergebnis der `ToString`-Methode ist im Fenster sichtbar.

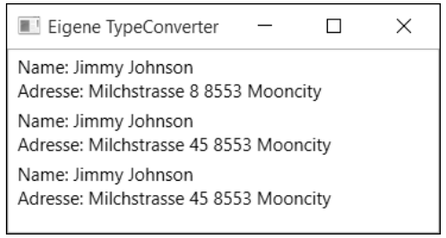

Abbildung 3.3 Der Rückgabewert der ToString-Methode eines Friend-Objekts

Während in Listing 3.18 das Property-Element `<local:Friend.Address>` verwendet wurde, um die Adresse zu setzen, wäre doch auch die in Listing 3.19 dargestellte Erzeugung eines `Friend`-Objekts mit der Initialisierung der `Address`-Property über die Attribut-Syntax hilfreich:

```
<local:Friend Name="Jimmy Johnson"
  Address="Milchstrasse 45 8553 Mooncity"/>
```

Listing 3.19 Beispiele\K03\11 SimpleTypeConverter\MainWindow.xaml

Damit der XAML-Parser jedoch aus dem in Listing 3.19 angegebenen String `Milchstrasse 45 8553 Mooncity` ein `Address`-Objekt erzeugen kann, wird ein Type-Converter benötigt. Dazu wird in diesem Beispiel die Klasse `AddressConverter` erstellt, die von `System.ComponentModel.TypeConverter` abgeleitet ist und die für den XAML-Parser benötigte Methode `ConvertFrom` überschreibt:

```
public class AddressConverter : TypeConverter
{
  public override object ConvertFrom(ITypeDescriptorContext
   context,System.Globalization.CultureInfo culture,object value)
  {
    if (value is string)
    {
      string[] array = value.ToString().Split(' ');
      if (array.Length == 4)
      {
        return new Address
        {
          Street = array[0],
          StreetNumber = array[1],
          ZipCode = int.Parse(array[2]),
          City = array[3]
        };
      }
    }
```



```
    return base.ConvertFrom(context, culture, value);
  }
  ...
}
```

Listing 3.20 Beispiele\K03\11 SimpleTypeConverter\AddressConverter.cs

Ist in der überschriebenen `ConvertFrom`-Methode der in der Parametervariablen `value` enthaltene Wert ein `string`, wird dieser nach Leerzeichen gesplittet. Aus dem daraus erhaltenen String-Array wird ein `Address`-Objekt erzeugt, das als Rückgabewert der `ConvertFrom`-Methode zurückgegeben wird.

Wie an der Signatur der `ConvertFrom`-Methode erkennbar ist, können Sie innerhalb der Methode zusätzlich die `CultureInfo` prüfen, um eventuell für verschiedene Länder und Regionen den Rückgabewert entsprechend zu formatieren. Dies ist bei dem hier erstellten `Address`-Objekt jedoch nicht notwendig und kann vernachlässigt werden.

Die erstellte `AddressConverter`-Klasse muss jetzt noch mit der `Address`-Klasse verbunden werden. Dazu wird auf der `Address`-Klasse das Attribut `System.ComponentModel.TypeConverterAttribute` gesetzt, das den Typ des zu verwendenden Type-Converters als Konstruktor-Parameter entgegennimmt:

```
[TypeConverter(typeof(AddressConverter))]
public class Address {
...
```

Listing 3.21 Beispiele\K03\11 SimpleTypeConverter\Address.cs

Mit dem Code in Listing 3.21 ist die Verbindung zwischen der `Address`-Klasse und der `AddressConverter`-Klasse hergestellt. Wird das XAML-Dokument deserialisiert, ruft der XAML-Parser für eine mittels Attribut-Syntax gesetzte `Address`-Property – wie in Listing 3.19 zu sehen ist – die `ConvertFrom`-Methode der `AddressConverter`-Klasse auf. Er erhält als Rückgabewert der `ConvertFrom`-Methode das von der `Address`-Property verlangte `Address`-Objekt.

Mit der erstellten `AddressConverter`-Klasse stehen Ihnen jetzt drei Möglichkeiten offen, um in XAML ein `Friend`-Objekt mit einer Adresse zu erzeugen (siehe Listing 3.22):

```
<!-- Ohne Verwendung des AddressConverters -->
<local:Friend Name="Jimmy Johnson">
  <local:Friend.Address>
    <local:Address Street="Milchstrasse"
                   StreetNumber="8"
                   ZipCode="8553"
                   City="Mooncity"/>
 </local:Friend.Address>
</local:Friend>
```

```
<!-- Nutzung des AddressConverters - Attribut-Syntax -->
<local:Friend Name="Jimmy Johnson" Address="Milchstrasse 45 8553
Mooncity"/>
<!-- Nutzung des AddressConverters - Property-Element-Syntax-->
<local:Friend Name="Jimmy Johnson">
  <local:Friend.Address>
    <local:Address>
      Milchstrasse 45 8553 Mooncity
    </local:Address>
  </local:Friend.Address>
</local:Friend>
```

Listing 3.22 Beispiele\K03\11 SimpleTypeConverter\MainWindow.xaml

Beachten Sie in Listing 3.22 die dritte Möglichkeit. Die `Address`-Klasse hat keine Content-Property definiert, dennoch enthält das `Address`-Element in Listing 3.22 als direkten Inhalt einen String. Der XAML-Parser sucht zunächst auf der `Address`-Klasse nach dem `ContentPropertyAttribute`, findet aber keines. Bevor er eine Exception auslöst, sucht er nach dem Type-Converter der `Address`-Klasse, um den im `Address`-Element enthaltenen String in ein `Address`-Objekt zu konvertieren. Der XAML-Parser findet die `AddressConverter`-Klasse und erstellt aus dem String das `Address`-Objekt, das anschließend der `Address`-Property des `Friend`-Objekts zugewiesen wird.

Hinweis

Der Code aus Listing 3.22 funktioniert zwar zur Laufzeit, aber der Designer in Visual Studio kann die Freunde nicht darstellen. Damit auch der Designer zufrieden ist, muss der eingesetzte `TypeConverter` – in diesem Beispiel der `AddressConverter` – neben der `ConvertFrom`-Methode auch die `CanConvertFrom`-Methode überschreiben. Mit dieser Methode muss der `TypeConverter` mitteilen, dass er einen String konvertieren kann, da in XAML zunächst alles ein String ist. Das Überschreiben der `CanConvertFrom`-Methode sieht somit wie folgt aus:

```
public override bool CanConvertFrom(ITypeDescriptorContext context,
  Type sourceType)
{
  if (sourceType == typeof(string))
    return true;
  return base.CanConvertFrom(context, sourceType);
}
```

Hinweis

Besitzt eine Klasse keine Content-Property, ist es in XAML dennoch möglich, in einem Objektelement dieser Klasse Text direkt einzufügen, ohne ein Property-Element zu verwenden.

Der XAML-Parser sucht in diesem Fall den zur Klasse gehörenden Type-Converter, um aus dem im Objektelement enthaltenen Text ein Objekt der Klasse zu erstellen, die dem Objektelement zugeordnet ist. Findet der XAML-Parser keinen Type-Converter, löst er eine Exception aus.

3.5.3 Type-Converter in C# verwenden

Um in C# einen Type-Converter zu nutzen, verwenden Sie die Klasse `TypeDescriptor` (Namespace: `System.ComponentModel`). Listing 3.23 erzeugt mit der Klasse `TypeDescriptor` ein `Address`-Objekt. Dabei wird implizit die Klasse `AddressConverter` verwendet.

```
Address a = (Address)TypeDescriptor.GetConverter(typeof(Address))
              .ConvertFrom("Milchstrasse 45 8553 Mooncity");
```

Listing 3.23 Beispiele\K03\11 SimpleTypeConverter\MainWindow.xaml.cs

Die statische Methode `GetConverter` der Klasse `TypeDescriptor` prüft das `TypeConverterAttribute` auf dem übergebenen Typ. Hier ist der übergebene Typ `Address`. Die Klasse `Address` setzt mit dem `TypeConverterAttribute` die Klasse `AddressConverter` als Type-Converter. Die Methode `GetConverter` erstellt folglich ein Objekt der Klasse `AddressConverter` und gibt dieses zurück. Auf dem zurückgegebenen `AddressConverter`-Objekt wird die `ConvertFrom`-Methode mit dem zu konvertierenden String aufgerufen. Das von `ConvertFrom` zurückgegebene Objekt muss noch entsprechend gecastet werden.

Die Klasse `TypeDescriptor` und die statische Methode `GetConverter` sind insbesondere dann hilfreich, wenn Sie den Type-Converter erst zur Laufzeit ermitteln wollen. Der XAML-Parser weiß beispielsweise nichts vom hier erstellten `AddressConverter`. Folglich muss er ihn zur Laufzeit ermitteln. Dazu kann der XAML-Parser die `TypeDescriptor`-Klasse verwenden und die statische `GetConverter`-Methode aufrufen.

Ist der Type-Converter bereits zur Kompilierzeit bekannt, können Sie natürlich auch direkt ein Objekt der entsprechenden `TypeConverter`-Klasse erstellen:

```
Address a = (Address)new AddressConverter()
              .ConvertFrom("Milchstrasse 45 8553 Mooncity");
```

Hinweis

Das Eigenschaften-Fenster in Visual Studio verwendet übrigens auch Type-Converter, um die dort eingegebenen Strings in genau diejenigen Objekte zu konvertieren, die von den Properties verlangt werden.

Tipp

In Kapitel 2, »Das Programmiermodell«, wurde beschrieben, dass unter anderem die `Width`- und `Height`-Properties – beide sind vom Typ `double` – der Klasse `FrameworkElement` in logischen Einheiten angegeben werden (was 1/96 Inch entspricht). In XAML lassen sich dank Type-Convertern für `Width` und `Height` und auch für andere Properties, die logische Einheiten verlangen, sogenannte »qualifizierte« `double`-Werte angeben, beispielsweise wie folgt:

```
<Button Width="10cm"/>
```

Der Button wird jetzt 10 cm breit dargestellt. Sie haben folgende Werte, um einen `double` zu qualifizieren:

- **`px`** – der Default; wird verwendet, wenn Sie nichts angeben. `px` sind logische Einheiten (1/96 Inch) und nicht, wie die Bezeichnung vermuten lässt, Pixel.
- **`in`** – Angabe von Inches; **1 in = 96 px**
- **`cm`** – Angabe von Zentimetern; **1 cm = 96 ÷ 2,54 px**. Der oben gezeigte Button ist also nach dieser Formel 10 × 96 ÷ 254 px breit, was gerundet 378 px (logischen Einheiten) entspricht.
- **`pt`** – Angabe in Points; **1 pt = 96 ÷ 72 px**. Die Einheit Points ist eine allgemeine Einheit zur Angabe von Schriftgrößen. Ein Point entspricht dabei exakt 1/72 Inch. Damit entspricht eine logische Einheit 0,75 Points (= 96 ÷ 72). Eine Angabe von 36 Point ist also genau ein halber Inch.

Das Qualifizieren eines `doubles` mit einem der oben dargestellten Werte funktioniert nur, da die `Width`- und `Height`-Properties von `FrameworkElement` – und auch viele andere Properties – mit einem `TypeConverterAttribute` ausgestattet sind. Das `TypeConverterAttribute` kann nicht nur auf Klassenebene, sondern auf allen Mitgliedern einer Klasse stehen, da das Attribut selbst mit `AttributeTargets.All` definiert ist. Die in `FrameworkElement` definierte `Width`-Property hat die folgende Signatur:

```
[TypeConverter(typeof(LengthConverter)),...]
public double Width
{ ... }
```

Um den oben deklarierten Button mit einer Breite von 10 cm in C# zu setzen, greifen Sie entweder direkt auf die Klasse `LengthConverter` (Namespace: `System.Windows`) zurück, oder Sie verwenden wie folgt die `TypeDescriptor`-Klasse, um an den `LengthConverter` zu gelangen:

```
Button btn = new Button();
TypeConverter tc =
  TypeDescriptor.GetProperties(btn)["Width"].Converter;
btn.Width = (double)tc.ConvertFromString("10cm");
```


3.6 Markup-Extensions

Mit der Attribut-Syntax lassen sich einer Property nur primitive Typen und Aufzählungswerte zuweisen. Mit den Type-Convertern haben Sie eine Möglichkeit gesehen, auch Properties mit komplexeren Typen über die Attribut-Syntax zu initialisieren. Die Markup-Extensions verschaffen Ihnen eine weitere Möglichkeit, Objekte über die Attribut-Syntax und über die Property-Element-Syntax zu erstellen und einer Property zuzuweisen. Markup-Extensions bieten Ihnen zudem die Option, nicht unbedingt ein neues Objekt erstellen zu müssen, sondern beispielsweise mit einem Data Binding ein bereits existierendes Objekt zu referenzieren. Eine Markup-Extension verweist aus XAML auf eine Klasse, die von der abstrakten Klasse `MarkupExtension` (Namespace: `System.Windows.Markup`) abgeleitet ist und deren Methode `ProvideValue` implementiert.

Im Folgenden sehen wir uns drei Bereiche an:

- **Verwenden von Markup-Extensions in XAML und C#** – Dieser Abschnitt zeigt, was es mit Markup-Extensions auf sich hat.
- **XAML Markup-Extensions** – Dieser Abschnitt gibt Ihnen einen Überblick über die Markup-Extensions, die im XML-Namespace von XAML und somit im CLR-Namespace `System.Windows.Markup` definiert sind.
- **Markup-Extensions der WPF** – Der dritte Abschnitt gibt Ihnen eine Übersicht über die Markup-Extensions, die in den CLR-Namespaces der WPF definiert sind.

3.6.1 Verwenden von Markup-Extensions in XAML und C#

Wie bereits erwähnt wurde, sind Markup-Extensions Klassen, die von der abstrakten Klasse `MarkupExtension` (Namespace: `System.Windows.Markup`) abgeleitet sind und deren Methode `ProvideValue` implementieren. Markup-Extensions werden beim Verwenden der Attribut-Syntax in geschweifte Klammern eingeschlossen. Der Ausschnitt in Listing 3.24 nutzt Markup-Extensions mit der Attribut-Syntax:

```
<StackPanel>
  <Button Height="{x:Static SystemParameters.IconHeight}"
          Background="{x:Null}"
          Content="{Binding ElementName=slider,Path=Value}"
          Margin="10"/>
  <Slider x:Name="slider" Minimum="0" Maximum="100" Value="0" Margin="10"/>
</StackPanel>
```

Listing 3.24 Beispiele\K03\12 MarkupExtensionsAttributSyntax\MainWindow.xaml

`Static`, `Null` und `Binding` verweisen auf Klassen, die von der Klasse `MarkupExtension` abgeleitet sind. Die Klassen heißen dabei `StaticExtension`, `NullExtension` und `Binding`. Der XAML-Par-

ser sucht automatisch in den entsprechenden .NET-Namespaces nach Klassen, die dem ersten Wort in den geschweiften Klammern entsprechen. Dabei hängt der XAML-Parser automatisch ein »Extension« an den Klassennamen an, wenn er die entsprechende Klasse nicht findet und der Klassenname noch kein `Extension`-Suffix besitzt. Von den Klassen wird eine Instanz erzeugt. Das in Listing 3.24 erzeugte `Binding`-Objekt setzt die Properties `ElementName` und `Path`.

Hinweis

Beachten Sie, dass für die gesetzten Properties einer Markup-Extension bei der Attribut-Syntax keinerlei Anführungszeichen oder Hochkommas verwendet werden. Stattdessen wird der Wert für eine Eigenschaft direkt hinter das Gleichheitszeichen gesetzt. Einzelne Properties werden mit einem Komma voneinander getrennt:

```
"{Binding ElementName=slider,Path=Value}"
```

Abbildung 3.4 zeigt das Fenster mit dem XAML-Inhalt aus Listing 3.24.

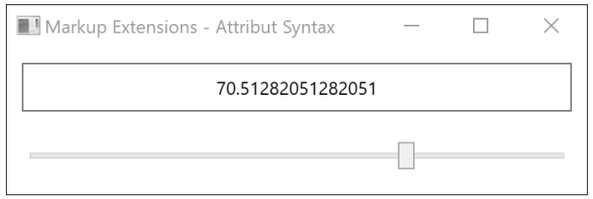

Abbildung 3.4 Die Background-Property des Buttons ist null.
Der Content des Buttons ist an den Wert des Slider-Controls gebunden.

Hinweis

Wenn Sie in XAML die geschweiften Klammern in einem XML-Attribut verwenden, sucht der XAML-Parser immer nach einer `MarkupExtension`-Klasse, die dem ersten Wort in den geschweiften Klammern entspricht. Falls Sie einer Property mit der Attribut-Syntax tatsächlich einen in geschweiften Klammern eingeschlossenen String zuweisen möchten, müssen Sie vor den String als Escape-Sequenz ein leeres, geschweiftes Klammerpaar setzen. Dadurch sucht der XAML-Parser nicht nach der Klasse und interpretiert alles Folgende als String.

Folgende Zeile weist der `Content`-Property eines Buttons den String `{OK}` zu:

```
<Button Content="{}{OK}"/>
```

Ohne das leere Klammerpaar vor `{OK}` würde der XAML-Parser nach einer Klasse `OK` suchen, nicht fündig werden und folglich eine Exception auslösen. Nur bei der Attribut-Syntax interpretiert der XAML-Parser ein geschweiftes Klammernpaar als Markup-Extension. Einen Button mit dem Text `{OK}` könnten Sie somit auch wie folgt erstellen:

```
<Button>{OK}<Button/>
```


Den Button aus Listing 3.24 erzeugen Sie in C# wie folgt:

```
var btn = new System.Windows.Controls.Button();
btn.Height = System.Windows.SystemParameters.CaptionHeight;
btn.Background = null;
var b = new System.Windows.Data.Binding();
b.ElementName = "slider";
b.Path = new System.Windows.PropertyPath("Value");
btn.SetBinding(System.Windows.Controls.Button.ContentProperty, b);
```

Listing 3.25 Der Button aus Listing 3.24 in C#

Anstatt Markup-Extensions in XAML mithilfe der geschweiften Klammern und der Attribut-Syntax zu verwenden, können Sie Markup-Extensions auch als gewöhnliche Objektelemente erstellen. Listing 3.26 ist analog zu Listing 3.24, allerdings werden die Markup-Extensions nicht mit der Attribut-Syntax zugewiesen, sondern jetzt als Objektelemente erstellt:

```
<Window x:Class="MarkupExtensionsObjectElementSyntax.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Markup Extensions - ArrayExtension">
  <StackPanel>
    <Button Margin="10">
      <Button.Content>
        <Binding ElementName="slider" Path="Value"/>
      </Button.Content>
      <Button.Height>
        <x:Static Member="SystemParameters.IconHeight"/>
      </Button.Height>
      <Button.Background>
        <x:Null/>
      </Button.Background>
    </Button>
    <Slider x:Name="slider" Minimum="0" Maximum="100" Value="0" Margin="10"/>
  </StackPanel>
</Window>
```

Listing 3.26 Beispiele\K03\13 MarkupExtensionsObjectElementSyntax\MainWindow.xaml

Wie Listing 3.26 zeigt, sind die Markup-Extensions `Static` und `Null` im XAML-Namespace definiert, die Markup-Extension `Binding` dagegen im XML-Namespace der WPF.

3.6.2 XAML-Markup-Extensions

Tabelle 3.1 zeigt die Markup-Extensions von XAML, wobei angenommen wird, dass der XML-Namespace von XAML wie üblich über ein x-Alias verfügt.

Markup-Extension	Beschreibung

`x:Array`	Nutzen Sie die `x:Array`-Markup-Extension, um in XAML ein .NET-Array zu erstellen. Dabei geben Sie den Typ des Arrays über die `Type`-Property an. Folgender Codeausschnitt zeigt die Definition eines Integer-Arrays in XAML. Dabei wird vorausgesetzt, dass auf dem Wurzelelement ein Namespace-Mapping zum `System`-Namespace mit dem Alias `sys` existiert, um in XAML auf die `Int32`-Struktur zugreifen zu können. Der Typ des Arrays wird übrigens mit der Markup-Extension `x:Type` festgelegt, die ebenfalls in dieser Tabelle beschrieben wird.
	`<x:Array Type="{x:Type sys:Int32}">`
	`  <sys:Int32>2</sys:Int32>`
	`  <sys:Int32>4</sys:Int32>`
	`  <sys:Int32>8</sys:Int32>`
	`</x:Array>`
	Listing 3.27 Beispiele\K03\14 MarkupExtensionsArrayExtension\MainWindow.xaml

`x:Null`	Die `x:Null`-Markup-Extension verwenden Sie, um einer Property eine `null`-Referenz zuzuweisen. `<Button Background="{x:Null}"/>`

`x:Reference`	Die `x:Reference`-Markup-Extension nutzen Sie, um in XAML einer Property direkt eine Referenz eines anderen in XAML definierten Elements zuzuweisen. Dies ist eine Alternative zu einem Data Binding:
	`<Label Content="_Vorname" Target=`
	`"{x:Reference txtFirstName}"/>`
	`<TextBox x:Name="txtFirstName"/>`

`x:Static`	Nutzen Sie die `x:Static`-Markup-Extension, um auf statische Properties, Felder und Konstanten in einer Aufzählung oder Klasse zuzugreifen. Falls sich die Klasse nicht im Default-Namespace befindet, fügen Sie vor dem Klassennamen das Alias des XML-Namespaces an, der auf den entsprechenden CLR-Namespace zeigt.

Tabelle 3.1 XAML-Markup-Extensions

Markup-Extension	Beschreibung
`x:Static` (Forts.)	Im folgenden Beispiel wurde der `System`-Namespace dem XML-Namespace mit dem Präfix `sys` zugeordnet. Der Inhalt eines Buttons wird auf den Wert der statischen Property `MaxValue` der Klasse `System.Int32` gesetzt. `<Button Content="{x:Static sys:Int32.MaxValue}"/>` **Listing 3.28** Beispiele\K03\15 MarkupExtensionsStaticExtension\MainWindow.xaml
`x:Type`	Erstellt eine Instanz der Klasse `System.Type`. Die `x:Type`-Markup-Extension ist das Pendant zum `typeof`-Operator in C#. Insbesondere bei Styles und Templates verwenden Sie die `x:Type`-Markup-Extension, da Sie dort angeben müssen, auf welchen Typ sich der Style oder das Template bezieht. `<Style TargetType="{x:Type TextBox}">` Dazu finden Sie mehr in Kapitel 11, »Styles, Trigger und Templates«.

Tabelle 3.1 XAML-Markup-Extensions (Forts.)

3.6.3 Markup-Extensions der WPF

Mit der `Binding`-Klasse haben Sie bereits eine Markup-Extension der WPF kennengelernt. Daneben gibt es ein paar weitere. Tabelle 3.2 zeigt einen Überblick über die WPF-Markup-Extensions.

Markup-Extension	Beschreibung
`Binding`	Definieren Sie mit der `Binding`-Markup-Extension ein Data Binding in XAML. Listing 3.24 und Listing 3.26 machten bereits von dieser Extension Gebrauch.
`ComponentResourceKey`	Mit `ComponentResourceKey` referenzieren Sie Ressourcen durch Angabe eines Typs und einer Ressource-ID. Dabei können die Ressourcen in einer anderen Assembly liegen.
`DynamicResource`	Mit der `DynamicResource`-Markup-Extension weisen Sie einer Property den Wert einer Ressource zu. Wird zur Laufzeit die Ressource geändert, wird auch automatisch der Wert der Property geändert.

Tabelle 3.2 Markup-Extensions der WPF

Markup-Extension	Beschreibung
`RelativeSource`	Diese Markup-Extension wird im Zusammenhang mit einem Binding verwendet. Die `Binding`-Klasse enthält eine Property mit dem gleichen Namen. Folgender Codeausschnitt setzt den Inhalt eines Buttons auf den Wert der `ActualWidth`-Property des Buttons: `<Button Content="{Binding Path=ActualWidth,RelativeSource ={`**`RelativeSource Mode=Self`**`}}"/>` **Listing 3.29** Beispiele\K03\16 MarkupExtensionsRelativeSource\MainWindow.xaml
`StaticResource`	Mit der `StaticResource`-Markup-Extension weisen Sie einer Property den Wert einer Ressource zu. Im Gegensatz zu `DynamicResource` wird der Wert der Property nicht geändert, wenn sich die Ressource ändert. Mehr zu `DynamicResource` und `StaticResource` erfahren Sie in Kapitel 10, »Ressourcen«.
`TemplateBinding`	Mit dieser Markup-Extension binden Sie Properties in einem `ControlTemplate` an Properties des Controls, auf das das `ControlTemplate` angewendet wird. Mehr Informationen dazu erhalten Sie in Kapitel 11, »Styles, Trigger und Templates«.
`ThemeDictionary`	Verwenden Sie diese Markup-Extension, falls Sie zu bestimmten Windows-Themes eigene Styles verwenden möchten. Beispielsweise können Sie damit Buttons unter dem Windows-10-Theme *Aero2* immer blau darstellen.

Tabelle 3.2 Markup-Extensions der WPF (Forts.)

Mit Markup-Extensions werden Sie es in diesem Buch noch oft zu tun haben. An die Syntax mit den geschweiften Klammern, die auf den ersten Blick vielleicht etwas seltsam erscheint, werden Sie sich relativ schnell gewöhnen.

Hinweis zu eigenen Markup-Extensions

Für sehr spezielle Fälle möchten Sie eventuell eine eigene Markup-Extension erstellen. Dazu kreieren Sie eine Subklasse von `MarkupExtension`. In der Subklasse müssen Sie lediglich die `ProvideValue`-Methode programmieren, die den ermittelten Wert zurückgibt.

Kapitel 15
Animationen

In bisherigen Programmiermodellen wurden Animationen meist mit einem Timer implementiert. Bei der WPF sind zum Erstellen von Animationen keine Timer notwendig. Mit zahlreichen Animationsklassen lassen sich Animationen sogar rein in XAML auf deklarative Weise erstellen.

Eine Animation ist im Grunde nur eine Illusion für das menschliche Auge. Die wohl bekannteste Animation ist das Kino. 25 leicht veränderte Bilder werden pro Sekunde angezeigt. Das menschliche Auge wird dabei getäuscht, denn es nimmt nicht mehr die 25 einzelnen Bilder wahr, sondern registriert eine flüssige Bewegung.

Bei der WPF beschreibt eine Animation nicht wie im Kino einzelne Bilder, sondern die Änderung des Wertes einer Dependency Property über einen bestimmten Zeitraum hinweg. Beispielsweise wird die `Width`-Property eines Buttons in einer Sekunde vom Wert 100 zum Wert 200 animiert.

Obwohl Animationen auch heute noch vielen Entwicklern wie eine Spielerei vorkommen, können sie – gezielt eingesetzt – tatsächlich Mehrwert schaffen. Das Internet ist voll von animierten Inhalten. In Windows-Anwendungen lässt sich die Navigation mit Animationen verbessern, Buttons können animierte `MouseOver`-Effekte haben, Informationen können animiert eingeblendet werden usw. Auch die Anwendung *FriendStorage* verwendet eine Animation, um den Freunde-Explorer ein- und auszublenden. Es gibt zahlreiche Anwendungsgebiete.

Zum Animieren von Dependency Properties besitzt die WPF zahlreiche Klassen, die sich im Namespace `System.Windows.Media.Animation` befinden. Die Klassen besitzen reichlich Properties und sind somit bestens für XAML geeignet.

Mit den Klassen der WPF werden drei Animationsarten unterstützt:

- **Basis-Animationen** (die auch als From/To/By-Animationen bezeichnet werden)
- **Keyframe-Animationen**
- **Pfad-Animationen**

In Abschnitt 15.1, »Animationsgrundlagen«, erhalten Sie einen Überblick über die Voraussetzungen für Animationen und erfahren einiges über die drei Animationsarten, über Animationsklassen, Timelines, Clocks und das Interface `IAnimatable`.

In Abschnitt 15.2 und Abschnitt 15.3 erfahren Sie alle Details zu Basis-Animationen. Dabei werden diese zuerst in C# und dann in XAML erstellt. In Abschnitt 15.4 lernen Sie die Keyframe-Animationen kennen, bevor in Abschnitt 15.5 mit den Pfad-Animationen die dritte Animationsart vorgestellt wird.

In Abschnitt 15.6 erfahren Sie mehr zu den Easing Functions, die auch als Beschleunigungsfunktionen bezeichnet werden. Damit lassen sich Animationen mit einfachen Effekten wie einem Sprung- oder einem Federeffekt versehen, um realer zu wirken.

Der letzte Abschnitt zeigt, wie Sie Low-Level-Animationen implementieren. Dies sind Animationen, bei denen Sie im Code alles selbst erledigen müssen, damit sich etwas bewegt. Falls Sie bereits mit Adobe Flash Animationen erstellt haben, entdecken Sie im letzten Abschnitt Mechanismen, die der `OnEnterFrame`-Funktion aus Flash ähneln.

15.1 Animationsgrundlagen

Bevor wir Animationen sowohl in C# als auch in XAML erstellen, sollten Sie die notwendigen Grundlagen für Animationen beherrschen. Dieser Abschnitt vermittelt Ihnen in vier Bereichen das notwendige Wissen:

- **Voraussetzungen für Animationen** – Hier erfahren Sie, welche Voraussetzungen erfüllt sein müssen, damit Sie die Animationsklassen der WPF für eine Animation verwenden können.
- **Übersicht der Animationsarten und -klassen** – Die drei Animationsarten lernen Sie hier kennen; Sie erhalten auch einen Überblick über die Animationsklassen.
- **Timelines und Clocks** – Eine Animation hängt immer mit einer Timeline (Zeitlinie) und einer Clock (Uhr) zusammen; was es mit beiden auf sich hat, wird hier gezeigt.
- **Das Interface `IAnimatable`** – Das Interface `IAnimatable` definiert einige für Animationen interessante Methoden, aber auch Methoden, die Sie kennen müssen.

15.1.1 Voraussetzungen für Animationen

Mit einer Animation wird der Wert einer Property verändert. Damit eine Property sich mit den Animationsklassen der WPF animieren lässt, müssen drei Voraussetzungen erfüllt sein:

- Die Property muss als Dependency Property implementiert sein.
- Die Klasse, auf der die Dependency Property animiert wird, muss von `DependencyObject` erben und das Interface `IAnimatable` implementieren.
- Für den Typ der Property muss ein passender Animationstyp existieren. Falls kein passender Animationstyp existiert, müssen Sie eine eigene Subklasse von `AnimationTimeline` erstellen.

Sind alle drei Voraussetzungen erfüllt, hält Sie nichts mehr davon ab, die Animationsklassen der WPF zu nutzen.

Hinweis

Die Klassen im Namespace `System.Windows.Media.Animation` und die darin implementierte Logik werden auch als *Animationssystem* bezeichnet.

Hinweis

Ein Wert lässt sich auch ohne die Animationsklassen animieren. Klassisch funktioniert dies mit einem Timer. Die WPF bietet allerdings mit der Klasse `CompositionTarget` eine effektivere Variante als einen Timer an. Für die mit der Klasse `CompositionTarget` erstellten sogenannten *Low-Level-Animationen* gelten die in diesem Abschnitt dargestellten Voraussetzungen nicht. Sie müssen ähnlich wie im Callback eines Timers die Property-Werte für jedes Bild selbst setzen. Details zur Low-Level-Animation vermittelt Abschnitt 15.7.

15.1.2 Übersicht über die Animationsarten und -klassen

Das Animationssystem der WPF unterstützt drei unterschiedliche Animationsarten:

- **Basis-Animationen** – Eine Basis-Animation ändert den Wert einer Property über einen bestimmten Zeitraum. Dabei werden üblicherweise ein Start- und ein Zielwert sowie eine Zeitdauer angegeben. Basis-Animationen werden aufgrund der verwendeten Properties oft auch als *From/To/By-Animationen* bezeichnet.
- **Keyframe-Animationen** – Bei einer Keyframe-Animation werden nur die wichtigsten Schlüsselbilder (Keyframes) zu bestimmten Zeitpunkten angegeben. Die WPF berechnet die Bilder dazwischen. Durch die Schlüsselbilder muss eine Animation als Ganzes nicht linear verlaufen, sondern kann eben in unterschiedliche Richtungen gehen. Dadurch ist diese Animationsart weitaus mächtiger als die Basis-Animation. Für eine Keyframe-Animation stehen vier Arten von Schlüsselbildern zur Verfügung: `LinearKeyFrames`, `SplineKeyFrames`, `EasingKeyFrames` und `DiscreteKeyFrames`.
- **Pfad-Animationen** – Eine Pfad-Animation animiert einen bestimmten Wert entlang eines geometrischen Pfades. Dies ermöglicht es Ihnen, beispielsweise ein Objekt an einem bestimmten Pfad entlangwandern zu lassen.

Eine Animation beeinflusst den Wert einer Dependency Property. Mit dem Animationssystem der WPF geben Sie im Grunde nur die Werte zu bestimmten Zeitpunkten an. Die WPF berechnet dazwischenliegende Werte. Dieser Berechnungsprozess wird auch als *Interpolation* bezeichnet.

Da eine Animation den Wert einer Property beeinflusst, muss die Animation bei der Interpolation auch Werte bereitstellen, die dem Typ der Property entsprechen. Beispielsweise muss eine Animation für eine Property vom Typ `Color` auch `Color`-Werte erzeugen. Eine Animation für eine Property vom Typ `Double` muss `Double`-Werte erzeugen. Daher gibt es für verschiedene Typen verschiedene Animationsklassen. Für eine Basis-Animation vom Typ `Color` verwenden Sie die Klasse `ColorAnimation`, für eine Basis-Animation vom Typ `Double` die Klasse `DoubleAnimation`.

Die WPF besitzt Animationsklassen für 22 verschiedene Typen (siehe Tabelle 15.1).

.NET-Datentypen	WPF-Datentypen
Boolean	Color
Byte	Matrix
Char	Point
Decimal	Point3D
Double	Quaternion
Int16	Rect
Int32	Rotation3D
Int64	Size
Object	Thickness
Single	Vector
String	Vector3D

Tabelle 15.1 Die 22 für Animationen unterstützten Typen

Alle Animationsklassen der WPF folgen einem genauen Schema. Für jeden der in Tabelle 15.1 dargestellten Typ existiert eine abstrakte Klasse `<Typ>AnimationBase`, wobei `<Typ>` durch den jeweiligen Typ aus Tabelle 15.1 ersetzt wird. Für den Typ `Double` finden Sie beispielsweise die abstrakte Klasse `DoubleAnimationBase`. Von der Basisklasse `<Typ>AnimationBase` finden Sie für jeden der 22 Typen maximal drei Subklassen – eine für jede Animationsart:

- **`<Typ>Animation`** – die **Basis-Animation** für den angegebenen Typ; beispielsweise verwenden Sie zum Animieren einer `double`-Property die `DoubleAnimation`-Klasse.
- **`<Typ>AnimationUsingKeyFrames`** – die **Keyframe-Animation** für den angegebenen Typ; beispielsweise verwenden Sie zum Animieren einer `double`-Property die Klasse `DoubleAnimationUsingKeyFrames`.

- **`<Typ>AnimationUsingPath`** – die **Pfad-Animation** für den angegebenen Typ; Beispiel: die Klasse `DoubleAnimationUsingPath`

Hinweis

Im Namespace `System.Windows.Media.Animation` finden Sie für `KeyFrame`-Animationen fünf weitere Klassen für unterschiedliche Typen nach folgendem Schema (wobei Sie `<Typ>` wieder durch den jeweiligen Typ ersetzen):

- `<Typ>KeyFrameCollection`
- `<Typ>KeyFrame` (die abstrakte Basisklasse für die unteren vier)
- `Discrete<Typ>KeyFrame`
- `Linear<Typ>KeyFrame`
- `Easing<Typ>KeyFrame`
- `Spline<Typ>KeyFrame`

Wenn Sie die zur Verfügung stehenden Typen aus Tabelle 15.1 betrachten, werden Sie feststellen, dass beispielsweise eine Basis-Animation nicht für jeden Typ sinnvoll ist. Aufgrund dieser Tatsache hat eben nicht jede `<Typ>AnimationBase`-Klasse für jede der drei Animationsarten eine Subklasse. Beispielsweise leitet für den Typ `String` nur die Klasse `StringAnimationUsingKeyFrames` von `StringAnimationBase` ab. Mit der Keyframe-Animation geben Sie an, welchen Wert eine `String`-Property zu einem bestimmten Zeitpunkt hat. Es ergibt keinen Sinn, den String »Auto« zum String »Fahrrad« zu animieren, da nichts dazwischenliegt. Eine Berechnung beziehungsweise Interpolation der Werte dazwischen ist nicht möglich. Folglich gibt es keine `StringAnimation`- (Basis-Animation) und auch keine `StringAnimationUsingPath`-Klasse (Pfad-Animation).

Da nicht jeder Typ für jede Animationsart geeignet ist, gibt es bei der WPF die folgende Anzahl an Animationsklassen:

- 22 `<Typ>AnimationBase`-Klassen
- 17 `<Typ>Animation`-Klassen. Es gibt keine Basis-Animation für die Typen `Boolean`, `Char`, `Matrix`, `Object` und `String`.
- 22 `<Typ>AnimationUsingKeyFrames`-Klassen
- 3 `<Typ>AnimationUsingPath`-Klassen, und zwar nur für die Typen `Double`, `Matrix` und `Point`

Hinweis

Mit den `KeyFrame`-Klassen sieht es ähnlich aus wie mit den Animationsklassen. Sie finden bei der WPF:

- 22 `<Typ>KeyFrameCollection`-Klassen
- 22 `<Typ>KeyFrame`-Klassen

- 22 `Discrete<Typ>KeyFrame`-Klassen
- 17 `Linear<Typ>KeyFrame`-Klassen
- 17 `Spline<Typ>KeyFrame`-Klassen

Für die Typen `Boolean`, `Char`, `Matrix`, `Object` und `String` gibt es keine `Linear<Typ>KeyFrame`- und keine `Spline<Typ>KeyFrame`-Klasse. Da es für diese Typen auch keine Basis- und keine Pfad-Animation gibt, lassen sie sich nur mit einer `KeyFrame`-Animation und diskreten Keyframes animieren. Zwischen diskreten Keyframes (Schlüsselbildern) findet keine Interpolation statt. Sie sehen später in Abschnitt 15.4, »Keyframe-Animationen«, ein Beispiel dazu.

Hinweis

Wenn Sie die Animationsklassen betrachten, ist sicherlich die erste Überlegung, warum die Klassen nicht als generische Klassen implementiert wurden. Eine Klasse wie `AnimationBase<T>` hätte doch als Basisklasse genügt. Das womöglich wichtigste Argument gegen generische Klassen war, dass diese nicht vollständig von XAML unterstützt werden. Da die Klassen eben nicht generisch sind, lassen sich Animationen auch komplett in XAML erstellen.

15.1.3 Timelines und Clocks

Alle Animationen beziehungsweise die `<Typ>AnimationBase`-Klassen leiten über die abstrakte Klasse `AnimationTimeline` von der abstrakten Klasse `Timeline` ab (siehe Abbildung 15.1).

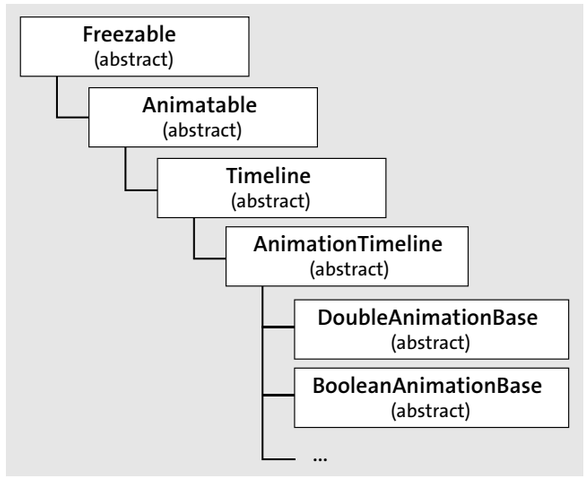

Abbildung 15.1 Die Animationsklassen in der Klassenhierarchie der WPF

Die Klasse `Timeline` repräsentiert eine Zeitlinie mit einer bestimmten Länge (sie wird in der `Duration`-Property angegeben). Die `Timeline`-Klasse besitzt einige Properties, die wir im Folgenden anhand der `DoubleAnimation`-Klasse betrachten:

- **`AccelerationRatio`** – ein Wert zwischen `0` (Default) und `1`, der die Beschleunigung mit einer Prozentangabe festlegt. Wenn auch die `DecelerationRatio`-Property gesetzt ist, darf die Summe aus beiden 1 nicht überschreiten.
- **`AutoReverse`** – Setzen Sie diese Property auf `true`, damit die `Timeline` nach dem Erreichen des Endwertes wieder rückwärts zum Startwert durchlaufen wird. Der Default-Wert ist `false`.
- **`BeginTime`** – ist vom Typ `Nullable<TimeSpan>`. Definiert den Startzeitpunkt für die Timeline. Der Default-Wert ist `0:0:0`.
- **`DecelerationRatio`** – ist ein Wert zwischen `0` (Default) und `1`, der die »Bremsrate« in Prozent festlegt.
- **`Duration`** – ist vom Typ `Duration`. Legt die Länge der Timeline und damit die Zeitdauer der Animation fest. Der Default-Wert ist der Wert des statischen Feldes `Duration.Automatic`, das einer Sekunde entspricht.
- **`FillBehavior`** – ist vom Typ der Aufzählung `FillBehavior`. Legt fest, ob der Wert einer animierten Dependency Property beibehalten wird, nachdem das Ende der Animation erreicht wurde.
- **`Name`** – definiert den Namen der `Timeline`.
- **`RepeatBehavior`** – ist vom Typ der Struktur `RepeatBehavior`. Legt fest, ob und wie oft die Timeline wiederholt wird. Der Default-Wert ist ein `RepeatBehavior`-Objekt mit dem Wert `1` in der `Count`-Property, wodurch die Timeline nicht wiederholt wird.
- **`SpeedRatio`** – ist vom Typ `double` (Default ist `1`). Legt den Faktor für die Zeit dieser Timeline relativ zur Eltern-Timeline fest. Falls die Timeline keine Eltern-Timeline hat, definiert dieser Faktor die Default-Geschwindigkeit der Timeline.

Hinweis

In der WPF lassen sich Timelines auch verschachteln, um komplexere Animationen zu erstellen. Wenn Sie bereits Anwendungen mit Adobe Flash entwickelt haben, sollte Ihnen das Prinzip von Timelines bekannt sein. Auch in Flash werden Timelines verschachtelt.

In Flash kann sich innerhalb der Wurzel-Timeline (`_root`) ein Movieclip-Objekt befinden, das wiederum seine eigene Timeline besitzt und wiederum weitere Movieclip-Objekte haben kann etc.

Verschachtelte Timelines bei der WPF lernen Sie später mit der Klasse `Storyboard` kennen.

Die Klasse `Timeline` verfügt neben den gezeigten Properties über eine Attached Property `DesiredFrameRate`. Darüber legen Sie optional fest, wie viele Bilder pro Sekunde ablaufen sollen. Der angegebene Wert ist nützlich, um Ressourcen zu sparen. Setzen Sie die `DesiredFrameRate` auf `1`, wird lediglich ein Bild pro Sekunde generiert.

Neben den Properties definiert die Klasse `Timeline` ein paar interessante Events und Methoden. Das `Completed`-Event ist sehr nützlich, um am Ende einer Animation etwas auszuführen.

Eine sehr wichtige Methode der `Timeline`-Klasse ist die `CreateClock`-Methode. Sie gibt ein `Clock`-Objekt zurück, das das eigentliche Arbeitstier hinter einer Animation ist. Die Timeline selbst ist nur die Beschreibung einer Animation, das `Clock`-Objekt führt letztlich die Animation durch. Das `Clock`-Objekt kennt unter anderem die aktuelle Zeit relativ zur Timeline (`CurrentTime`-Property) und den aktuellen Fortschritt (`CurrentProgress`-Property).

Hinweis

Im Hintergrund besitzt die WPF einen Zeitmanager, der die Zeit der Animationen regelt, indem er alle aktiven `Clock`-Objekte aktualisiert. Der Zeitmanager macht dies regelmäßig bei sogenannten *Ticks* (Augenblicken). Ein `Clock`-Objekt besitzt also nicht selbst den Mechanismus, um seine `CurrentTime`-Property zu ändern, sondern dies erfolgt durch einen bei der WPF versteckten, aber zentralen Zeitmanager.

Von `Timeline` ist neben `MediaTimeline` (die Thema von Kapitel 16, »Audio und Video«, ist) und `TimelineGroup` auch die Klasse `AnimationTimeline` abgeleitet. `AnimationTimeline` ist die Basis für alle `<Typ>AnimationBase`-Klassen. `AnimationTimeline` definiert unter anderem die abstrakte Read-only-Property `TargetPropertyType` vom Typ `Type`. Subklassen, wie `DoubleAnimationBase`, überschreiben diese Property und geben den Typ zurück, den sie animieren können. `DoubleAnimationBase` gibt `typeof(double)` zurück.

Die Klasse `AnimationTimeline` überdeckt auch die `CreateClock`-Methode der Klasse `Timeline` und gibt ein `AnimationClock`-Objekt zurück. `AnimationClock` erbt von `Clock` und besitzt die zusätzliche Methode `GetCurrentValue`, die den aktuellen Wert zu einem bestimmten Zeitpunkt liefert.

Damit hätten wir die Grundlagen zu Animationen geklärt. Die einzelnen Klassen und Properties sehen wir uns gleich genauer an. Der C#-Code, um beispielsweise eine Animation für einen `double`-Wert zu erstellen, ist relativ simpel:

```
var doubleAnimation = new DoubleAnimation
{
  From = 50,
  To = 100
};
```


Jetzt stellt sich die Frage, wie das `DoubleAnimation`-Objekt auf eine bestimmte Dependency Property vom Typ `Double` angewendet wird, um diese von 50 nach 100 zu animieren. Hier kommt das Interface `IAnimatable` ins Spiel.

15.1.4 Das Interface »IAnimatable«

Eine der drei zu Beginn dieses Abschnitts gezeigten Voraussetzungen für Animationen ist, dass das Objekt mit der zu animierenden Property das Interface `IAnimatable` implementiert. Das Interface `IAnimatable` besitzt die zum Starten einer Animation wichtigen Methoden `ApplyAnimationClock` und `BeginAnimation`. Das Folgende ist eine komplette Darstellung des Interfaces:

```
public interface IAnimatable
{
  void ApplyAnimationClock(DependencyProperty dp,
                           AnimationClock clock);
  void ApplyAnimationClock(DependencyProperty dp,
                           AnimationClock clock,
                           HandoffBehavior handoffBehavior);
  void BeginAnimation(DependencyProperty dp,
                      AnimationTimeline animation);
  void BeginAnimation(DependencyProperty dp,
                      AnimationTimeline animation,
                      HandoffBehavior handoffBehavior);
  object GetAnimationBaseValue(DependencyProperty dp);
  bool HasAnimatedProperties { get; }
}
```

Wie anhand der Methodensignaturen leicht zu erkennen ist, lassen sich nur Dependency Properties animieren. Sie merken: Der Einfluss von Dependency Properties bei der WPF erstreckt sich durch alle wichtigen Teile des Frameworks.

Hinweis

In den Metadaten einer Dependency Property finden Sie die Property `IsAnimationProhibited`. Hat diese den Wert `true`, lässt sich die Dependency Property nicht animieren. `IsAnimatedProhibited` ist in der Klasse `UIPropertyMetadata` definiert. `FrameworkPropertyMetadata` erbt von `UIPropertyMetadata`.

`IAnimatable` wird von den Klassen `UIElement` und `ContentElement` implementiert. Weiter gibt es die direkt von `Freezable` abgeleitete, abstrakte Klasse `Animatable`, die ebenfalls `IAnimatable` implementiert. Von `Animatable` leiten verschiedenste Klassen ab, für 2D-Grafik unter ande-

rem die Klassen `Brush`, `Geometry` oder `Drawing`. Auch 3D-Grafik-Klassen, wie `Model3D` oder `GeometryD`, erben von `Animatable` und lassen sich somit für Animationen nutzen.

Um eine Animation zu starten, haben Sie zwei Möglichkeiten:

- Sie erstellen von Ihrer `AnimationTimeline` mit `CreateClock` ein `AnimationClock`-Objekt. Dieses übergeben Sie der `ApplyAnimationClock`-Methode.
- Sie übergeben Ihre `AnimationTimeline`-Instanz direkt der `BeginAnimation`-Methode.

Wenn Sie die `BeginAnimation`-Methode aufrufen, wird die WPF intern ein `AnimationClock`-Objekt erstellen, da dieses für den Fortschritt in der Animation verantwortlich ist. Das explizite Erstellen eines `AnimationClock`-Objekts und der anschließende Aufruf von `ApplyAnimationClock` können aus mehreren Gründen sinnvoll sein:

- Sie wollen mit einer einzigen `AnimationClock` mehrere Dependency Properties animieren, beispielsweise die Höhe und Breite eines Buttons.
- Sie wollen Ihre Animation steuern (pausieren, stoppen, starten), was in C# nur beim Zugriff auf das `AnimationClock`-Objekt möglich ist.

Mit `BeginAnimation` wird also intern ein `AnimationClock`-Objekt erstellt und die Animation gestartet. Um beispielsweise die Breite eines Buttons in einer Sekunde von 50 auf 100 zu animieren, reicht folgender C#-Code aus:

```
var doubleAnimation = new DoubleAnimation
{
  From = 50,
  To = 100
};
btnToAnimate.BeginAnimation(Button.WidthProperty, doubleAnimation);
```

Es ist an der Zeit, sich die Basis-Animationen und die Properties der `Timeline`-Klasse genauer anzusehen.

15.2 Basis-Animationen in C#

Im vorherigen Abschnitt sind Sie bereits den drei Animationsarten begegnet. Hier sehen wir uns die aufgrund ihrer Properties auch als From/To/By-Animationen bezeichneten Basis-Animationen an. Basis-Animationen werden durch die Animationsklassen `<Typ>Animation` beschrieben. Die `<Typ>Animation`-Klassen erweitern die `<Typ>AnimationBase`-Klasse um die Properties `By`, `From`, `To`, `IsAdditive` und `IsCumulative`. In diesem Abschnitt betrachten wir diese fünf Properties und jene aus der `Timeline`-Klasse anhand der Klasse `DoubleAnimation`. Dabei animieren wir mit einer `DoubleAnimation`-Instanz die `Width`-Property eines Buttons, wie dies bereits im vorherigen Abschnitt teilweise gezeigt wurde.

Hinweis

Wir beschränken uns hier auf das Erstellen von Basis-Animationen in C#. Im nächsten Abschnitt werden Basis-Animationen in XAML angelegt.

15.2.1 Start- und Zielwert mit »From«, »To« und »By«

Die `<Typ>Animation`-Klassen besitzen die Properties `From`, `To` und `By`, um den Start- und den Zielwert einer Animation festzulegen. Diese Properties sind allesamt vom generischen Typ `Nullable<T>`, wobei der generische Typparameter dem zu animierenden Typ entspricht. Bei der `ColorAnimation`-Klasse sind die Properties vom Typ `Nullable<Color>`, bei der `DoubleAnimation`-Klasse vom Typ `Nullable<Double>`.

Der Button, dessen `Width`-Property im Folgenden animiert wird, befindet sich in einem Canvas (siehe Listing 15.1). Im Canvas wird er mit der Größe aus seiner `DesiredSize`-Property dargestellt.

```
<Window ... Loaded="Window_Loaded">
  <Canvas>
    <Button x:Name="btn" Content="OK"/>
  </Canvas>
</Window>
```

Listing 15.1 Beispiele\K15\01 BasisAnimationInCSharp\MainWindow.xaml

Für das `Loaded`-Event des `Windows` wurde in Listing 15.1 der Event Handler `Window_Loaded` angegeben. In der Codebehind-Datei wird darin ein `DoubleAnimation`-Objekt mit dem Startwert `100` und dem Endwert `200` erzeugt (siehe Listing 15.2). Auf dem `Button`-Objekt wird die `BeginAnimation`-Methode aufgerufen. Als erster Parameter wird die `Button.WidthProperty` und als zweiter das `DoubleAnimation`-Objekt übergeben.

```
private void Window_Loaded(object sender, RoutedEventArgs e)
{
  var doubleAnimation = new DoubleAnimation
  {
    From = 100,
    To = 200
  };
  btn.BeginAnimation(Button.WidthProperty, doubleAnimation);
}
```

Listing 15.2 Beispiele\K15\01 BasisAnimationInCSharp\MainWindow.xaml.cs

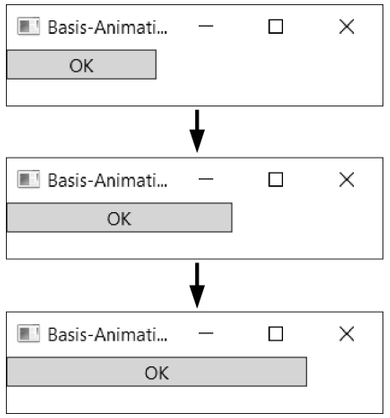

Abbildung 15.2 Ein animierter Button

Sobald das Fenster geladen wird, wird der Button animiert. Dabei springt er zu Beginn von seiner `DesiredSize` auf die Breite von 100. Die `Duration`-Property der `DoubleAnimation` wurde in Listing 15.2 nicht gesetzt. Folglich dauert die Animation aufgrund des Default-Wertes für die `Duration`-Property genau eine Sekunde. Es findet eine lineare Interpolation statt. Nach 0,1 Sekunden hat die `Width`-Property des Buttons den Wert 110, nach 0,2 Sekunden den Wert 120 und nach einer Sekunde den Wert 200. Abbildung 15.2 zeigt den Button nach 0,0, 0,5 und 1,0 Sekunden.

Hinweis

Beachten Sie in Listing 15.2, dass das `DoubleAnimation`-Objekt weder das Zielobjekt für die Animation noch die zu animierende Property kennt. Erst durch den Aufruf von `BeginAnimation` auf dem `Button`-Objekt werden das Zielobjekt (der Button) und die zu animierende Property (`Button.WidthProperty`) bekannt. Das `DoubleAnimation`-Objekt beschreibt somit nur die Animation und lässt sich daher auch in weiteren `BeginAnimation`-Aufrufen verwenden, um eine Dependency Property vom Typ `Double` zu animieren. Wenn Sie in Listing 15.2 unterhalb des Aufrufs von `BeginAnimation` auf dem `Button`-Objekt noch folgende Zeile einfügen, wird auch die Höhe des Fensters von 100 nach 200 animiert:

```
this.BeginAnimation(Window.HeightProperty, da);
```

15.2.2 Animation nur mit »To«

Anstatt bei einer Basis-Animation die Properties `From` und `To` zu setzen, ist auch nur das Setzen von `To` möglich. Dann wird als Startwert der aktuelle Wert der zu animierenden Dependency Property verwendet:


```
var doubleAnimation = new DoubleAnimation
{
  To = 200
};
btn.BeginAnimation(Button.WidthProperty, doubleAnimation);
```

Auf dem Button enthält die `Width`-Property den Default-Wert `Double.NaN`. Dieser Wert ist allerdings als Startwert für die Animation ungültig. Die Animation kann zwischen `Double.NaN` und 200 keine Werte berechnen oder interpolieren; Sie erhalten eine `AnimationException`. Die `Width`-Property lokal auf dem Button zu setzen schafft Abhilfe:

```
<Button x:Name="btn" Width="50" Content="OK"/>
```

Die `Width`-Property des Buttons wird jetzt linear vom lokalen Wert 50 zu dem in der Animation angegebenen Wert 200 animiert.

Hinweis

Das Weglassen der `From`-Property wird in der Praxis häufig verwendet, um flüssige Übergänge von Animationen zu schaffen. Stellen Sie sich vor, Sie möchten die `Width`-Property eines Buttons beim Event `MouseEnter` vergrößern und beim `MouseLeave` wieder verkleinern. Wenn Sie bei der Animation im `MouseLeave`-Event die `From`-Property setzen, wird die `Width`-Property des Buttons immer abrupt auf diesen Wert springen und von dort animiert. Wenn die Animation in `MouseEnter` den Button vollständig vergrößert, fällt dies nicht auf, aber das `MouseLeave`-Event kann ja stattfinden, bevor die Animation im `MouseEnter`-Event den Button vollständig vergrößert hat. Durch Weglassen der `From`-Property im `MouseLeave`-Event nimmt die Animation den aktuellen Wert der `Width`-Property als Startwert, wodurch die Animation flüssig ist.

15.2.3 Animation nur mit »From«

Neben einer Animation, die nur eine gesetzte `To`-Property enthält, ist auch eine Animation mit lediglich einer gesetzten `From`-Property möglich:

```
var doubleAnimation = new DoubleAnimation
{
  From = 200
};
btn.BeginAnimation(Button.WidthProperty, doubleAnimation);
```

Der Zielwert der Animation ist der aktuelle Wert der `Width`-Property. Diese darf natürlich auch hier nicht den Default-Wert `Double.NaN` enthalten. Lokales Setzen hilft auch hier weiter:

```
<Button x:Name="btn" Width="50" Content="OK"/>
```

Die Breite des Buttons wird jetzt vom Wert 200 zum Wert 50 animiert.

15.2.4 Animation mit »From« und »By«

Anstatt den Zielwert einer Animation mit der `To`-Property zu definieren, können Sie auch die `By`-Property der `<Typ>Animation`-Klassen nutzen. Die Summe aus der `From`-Property und der `By`-Property ergibt den Zielwert der Animation, der in folgendem Codeausschnitt somit bei 200 liegt:

```
var doubleAnimation = new DoubleAnimation
{
  From = 100,
  By = 100
};
btn.BeginAnimation(Button.WidthProperty, doubleAnimation);
```

Hinweis

Auch bei der Angabe von `By` kann das `From` weggelassen werden. Es wird dann der aktuelle Wert der zu animierenden Dependency Property als Startwert verwendet. Die Summe aus diesem Startwert und dem Wert der `By`-Property ergibt dann den Zielwert.

Werden auf einem `<Typ>Animation`-Objekt sowohl die `By`- als auch die `To`-Property gesetzt, hat der Wert der `By`-Property keine Bedeutung. Es wird der Wert der `To`-Property als Zielwert der Animation genutzt.

Wie dieser Abschnitt gezeigt hat, gibt es einige unterschiedliche Kombinationen für Basis-Animationen. Tabelle 15.2 hält fest, wie der Start- und der Zielwert einer Animation abhängig von den gesetzten Properties des `<Typ>Animation`-Objekts definiert werden.

Gesetzte Property	Auswirkung
`From` und `To`	Die Animation startet beim Wert der `From`-Property und endet beim Wert der `To`-Property.
`From` und `By`	Die Animation startet beim Wert der `From`-Property. Der Endwert ist die Summe aus dem Wert der `From`-Property und dem Wert der `By`-Property.
`From`	Die Animation startet beim Wert der `From`-Property. Der Endwert ist der Wert der zu animierenden Dependency Property, der vor der Animation aufgrund des Vorrangrechts gültig war.

Tabelle 15.2 Auswirkungen der Properties »From«, »To« und »By«

Gesetzte Property	Auswirkung
`To`	Der Startwert der Animation entspricht dem aufgrund des Vorrangrechts aktuellen Wert der zu animierenden Dependency Property (das ist beispielsweise ein lokaler Wert oder ein Wert aus einer vorherigen Animation). Der Endwert der Animation entspricht dem Wert der `To`-Property.
`By`	Der Startwert der Animation entspricht dem aufgrund des Vorrangrechts aktuellen Wert der zu animierenden Dependency Property. Der Endwert der Animation entspricht der Summe aus dem Startwert und dem Wert der `By`-Property.

Tabelle 15.2 Auswirkungen der Properties »From«, »To« und »By« (Forts.)

15.2.5 Dauer, Startzeit und Geschwindigkeit

Zum Definieren der Dauer besitzt die Klasse `Timeline` die Property `Duration` vom Typ `Duration`. Die `Duration`-Struktur kann ein Objekt der Struktur `TimeSpan` kapseln. Dazu übergeben Sie dem Konstruktor einfach das `TimeSpan`-Objekt:

```
var doubleAnimation = new DoubleAnimation
{
  From = 100,
  To = 200,
  Duration = new Duration(TimeSpan.Parse("0:0:2"))
};
```

Die Struktur `TimeSpan` besitzt zahlreiche statische Methoden, wie `FromSeconds` oder `FromMilliSeconds`, um ein `TimeSpan`-Objekt zu erstellen. In den folgenden Codeausschnitten wird allerdings immer die statische `Parse`-Methode verwendet, die einen String entgegennimmt. Diese `Parse`-Methode wird auch vom `TimeSpanConverter` aufgerufen. Der übergebene String ist somit für die spätere XAML-Betrachtung von Bedeutung. Der für die `Duration`-Property in XAML verwendete `DurationConverter` nutzt intern den `TimeSpanConverter` und kommt somit mit exakt denselben Strings zurecht.

Die Angabe des Zeit-Strings erfolgt nach folgendem Schema:

```
Stunden:Minuten:Sekunden.Sekundenbruchteile
```

`0:0:2` bedeutet zwei Sekunden. Die `Parse`-Methode verlangt nicht zwingend alle Werte. Doch Vorsicht: Mit dem String `1` definieren Sie nicht eine Sekunde, sondern eine Stunde. Mit dem String `0:5` definieren Sie fünf Minuten.

Hinweis

Vor den Stunden lassen sich zusätzlich auch Tage angeben:

`Tage.Stunden:Minuten:Sekunden.Sekundenbruchteile`

Der String `3.5:0:0` bedeutet drei Tage und fünf Stunden. Für Animationen ist die Angabe von Tagen aber wohl nicht notwendig.

Typischerweise sind die meisten Animationen nicht länger als ein paar Sekunden. Um Sekunden zu definieren, müssen Sie als Stunden und Minuten immer explizit `0` angeben. Die Syntax `0:0:1` steht für eine Sekunde, `0:0:1.5` für eineinhalb Sekunden und `0:0:0.5` oder auch `0:0:.5` für eine halbe Sekunde.

Die Struktur `Duration` wurde eingeführt, da für die Zeitdauer einer Timeline zwei weitere Werte möglich sind, die sich mit einem `TimeSpan`-Objekt nicht ausdrücken lassen. Die Struktur `Duration` enthält die beiden statischen Properties `Automatic` und `Forever`, die beide `Duration`-Objekte zurückgeben. `Automatic` ist der Default-Wert für die `Duration`-Property einer Timeline. Der Wert `Automatic` entspricht einem `Duration`-Objekt mit einer `TimeSpan` von einer Sekunde (`0:0:1`).

`Duration.Forever` bedeutet, dass die Zeitdauer der Timeline bis zum Ende der Zeit reicht (wann auch immer das sein mag). Die WPF kann zwischen jetzt und dem Ende der Zeit natürlich keine Werte interpolieren, somit ergibt dieser Wert auf unserer einfachen `DoubleAnimation` keinen Sinn. Anders ist dies auf Timelines, die weitere Timelines enthalten. Das Storyboard, das später noch beschrieben wird, ist eine solche Timeline, die weitere Timelines enthält.

Ein `Duration`-Objekt lässt sich nur über den Konstruktor mit einem `TimeSpan`-Objekt erstellen. Über die Read-only-Property `HasTimeSpan` erfahren Sie, ob Ihr `Duration`-Objekt über eine Zeitdauer verfügt, und über die Read-only-Property `TimeSpan` erhalten Sie das `TimeSpan`-Objekt.

Neben der für die Dauer notwendigen `Duration`-Property besitzt die `Timeline`-Klasse die Property `BeginTime` vom Typ `Nullable<TimeSpan>`. Sie beschreibt die Startzeit einer Timeline (Default ist `0:0:0`) und erlaubt es, ein Offset zu definieren. Folgende Animation wird erst eine Sekunde nach dem Aufruf von `BeginAnimation` gestartet und dauert somit insgesamt drei Sekunden:

```
var doubleAnimation = new DoubleAnimation
{
  From = 100,
  To = 200,
  BeginTime = TimeSpan.Parse("0:0:1"),
  Duration = new Duration(TimeSpan.Parse("0:0:2"))
};
```


`BeginTime` nimmt auch negative Zeitspannen entgegen. Folgende Animation wird genau in der Mitte gestartet, wodurch sie nur eine Sekunde dauert:

```
var doubleAnimation = new DoubleAnimation
{
  From = 100,
  To = 200,
  BeginTime = TimeSpan.Parse("-0:0:1"),
  Duration = new Duration(TimeSpan.Parse("0:0:2"))
};
```

Neben `Duration` und `BeginTime` lässt sich auch die Geschwindigkeit einer Animation regeln, was sich wiederum auf die Zeitdauer auswirkt. Dafür setzen Sie die Property `SpeedRatio` (vom Typ `double`). Ein Wert von `1` bedeutet »normale Geschwindigkeit«. Ein Wert von `0.5` bedeutet »halb so schnell«, wodurch die Animation doppelt so lange dauert. Ein Wert von `2` bedeutet »doppelt so schnell«, wodurch die Animation nur halb so lange dauert.

Bei der folgenden Animation ist in der `Duration`-Property eine Dauer von 2 Sekunden angegeben. Die `SpeedRatio`-Property ist auf `10` gesetzt, wodurch die Animation zehnmal so schnell abläuft. Die tatsächliche Dauer beträgt also 0,2 Sekunden.

```
var doubleAnimation = new DoubleAnimation
{
  From = 100,
  To = 200,
  Duration = new Duration(TimeSpan.Parse("0:0:2")),
  SpeedRatio = 10
};
```

Tipp

Die `SpeedRatio`-Property ist interessant, wenn Animationen vorliegen, deren Geschwindigkeit der Benutzer bestimmen kann. Beispielsweise lässt sich die `SpeedRatio`-Property an die `Value`-Property eines Sliders binden, damit der Benutzer die Geschwindigkeit festlegen kann.

15.2.6 Rückwärts und Wiederholen

Oftmals ist erwünscht, dass eine Animation nach dem Erreichen des Endes zurück zum Startwert läuft. Um dies zu erreichen, setzen Sie die `AutoReverse`-Property auf `true` (Default `false`). Die folgende `DoubleAnimation` erreicht nach zwei Sekunden den Wert `200` und ist nach exakt vier Sekunden wieder beim Wert `100` angelangt. Mit `AutoReverse` verdoppelt sich also die Zeitdauer einer Timeline.

```
var doubleAnimation = new DoubleAnimation
{
  From = 100,
  To = 200,
  Duration = new Duration(TimeSpan.Parse("0:0:2")),
  AutoReverse = true
};
```

Neben dem Rückwärtsgang gibt es auch die Möglichkeit, eine Animation zu wiederholen: Dazu nutzen Sie die `RepeatBehavior`-Property vom Typ `RepeatBehavior`. Die Struktur `RepeatBehavior` definiert entweder die Anzahl von Wiederholungen oder die Zeitdauer, in der wiederholt wird.

Dem `RepeatBehavior`-Konstruktor übergeben Sie entweder ein `double` für die Anzahl von Wiederholungen oder ein `TimeSpan`-Objekt für die Zeitdauer. Beim ersten Ansatz erhalten Sie den `double` anschließend über die `Count`-Property des `RepeatBehavior`-Objekts. Die `HasCount`-Property gibt `true` zurück. Beim zweiten Verfahren erhalten Sie das `TimeSpan`-Objekt über die `Duration`-Property, und `HasDuration` gibt `true` zurück.

Hinweis

Es ist tatsächlich so, dass die `Duration`-Property der `RepeatBehavior`-Struktur ein `TimeSpan`-Objekt zurückgibt. Lassen Sie sich davon nicht beirren.

Folgende Animation hat die Anzahl `2` als `RepeatBehavior` gesetzt. Die Animation wird somit zweimal durchlaufen. Da `AutoReverse` ebenfalls auf `true` steht, beträgt die Gesamtzeit acht Sekunden.

```
var doubleAnimation = new DoubleAnimation
{
  From = 100,
  To = 200,
  Duration = new Duration(TimeSpan.Parse("0:0:2")),
  AutoReverse = true,
  RepeatBehavior = new RepeatBehavior(2)
};
```

Sie könnten in obiger Animation die `RepeatBehavior`-Property auch wie folgt setzen, um dasselbe Ergebnis zu erzielen:

```
doubleAnimation.RepeatBehavior = new RepeatBehavior(TimeSpan.Parse("0:0:8"));
```

Setzen Sie `RepeatBehavior` beispielsweise auf `0:0:5`, wird die `Width`-Property des Buttons nach zwei Sekunden den Wert 200 erreichen, nach vier Sekunden wieder beim Wert 100 sein und nach fünf Sekunden beim Wert 150 stehen bleiben.

Tipp

`RepeatBehavior` besitzt eine statische Property `Forever` vom Typ `RepeatBehavior`. Wenn Sie diesen Wert der `RepeatBehavior`-Property Ihrer Timeline zuweisen, wird Ihre Animation ständig wiederholt:

```
doubleAnimation.RepeatBehavior = RepeatBehavior.Forever;
```

15.2.7 Die Gesamtlänge einer Timeline

Aus den bisher erwähnten Properties `Duration`, `SpeedRatio`, `AutoReverse` und `RepeatBehavior` ergibt sich die gesamte Länge/Dauer einer Timeline. Ist `RepeatBehavior` auf `Forever` gesetzt, ist die Gesamtlänge unendlich. Enthält `RepeatBehavior` einen `TimeSpan`-Wert, lässt sich die Gesamtlänge der Timeline wie folgt beschreiben:

Länge = BeginTime + RepeatBehavior

Enthält `RepeatBehavior` einen `double`-Wert, gilt für die Gesamtlänge der Timeline folgende Formel:

$$Länge = BeginTime + RepeatBehavior \times \frac{Duration \times (AutoReverse\ ?\ 2:1)}{Speedratio}$$

15.2.8 Wiederholen mit neuen Werten

Die `<Typ>Animation`-Klassen definieren neben den Properties `From`, `By` und `To` die beiden Properties `IsAdditive` und `IsCumulative`. Diese beiden Properties sind per Default `false`. Sie sind beim Wiederholen einer Animation von Bedeutung:

- **`IsAdditive`** – Setzen Sie diese Property auf `true`, wird der aktuelle Wert der zu animierenden Dependency Property zu den Properties `From` und `To` addiert.
- **`IsCumulative`** – Setzen Sie diese Property auf `true`, wird bei einer mit `RepeatBehavior` wiederholten Animation die Differenz zwischen der `To`- und der `From`-Property ermittelt. Diese Differenz wird zum Wert der `From`-Property addiert, woraus sich der für die Wiederholung neue Startwert ergibt. Die Differenz wird ebenfalls zur `To`-Property addiert, womit auch der für die Wiederholung neue Endwert feststeht. Die Animation läuft beim zweiten Durchlauf somit flüssig weiter.

`IsAdditive` funktioniert nur bei Aufrufen von `BeginAnimation`, die erneut stattfinden können, nicht jedoch bei Animationen, die mit `RepeatBehavior` wiederholt werden. `IsCumulative` dagegen funktioniert nur bei solchen Animationen, die mit `RepeatBehavior` wiederholt werden. Werfen wir einen kurzen Blick auf die beiden Properties. Dazu verwenden wir einen Button und animieren dessen `Width`-Property:

```
<Button x:Name="btn" Width="50" Content="OK"/>
```

Wird folgender Code mehrmals ausgeführt, springt der Button immer auf den Wert 100 zurück und wird zum Wert 200 animiert:

```
var doubleAnimation = new DoubleAnimation
{
  From = 100,
  To = 200
};
btn.BeginAnimation(Button.WidthProperty, doubleAnimation);
```

Wenn Sie die IsAdditve-Property auf dem oberen DoubleAnimation-Objekt auf true setzen, wird der aktuelle Wert der Width-Property immer zu den From- und To-Properties der DoubleAnimation addiert. Die erste Ausführung des Codes animiert die Breite (50) des Buttons somit von 150 zu 250. Die zweite Ausführung animiert die Breite des Buttons, die jetzt 250 beträgt, vom Wert 350 zum Wert 450 usw.

Im Gegensatz zu IsAdditive, das nichts mit RepeatBehavior zu tun hat, funktioniert die Property IsCumulative nur mit Wiederholungen anhand der RepeatBehavior-Property. Folgende Animation läuft zweimal. Der Wert der Width-Property wird vom Wert 100 zum Wert 200 animiert, springt dann abrupt zum Wert 100 zurück und wird nochmals zum Wert 200 animiert:

```
var doubleAnimation = new DoubleAnimation
{
  From = 100,
  To = 200,
  RepeatBehavior = new RepeatBehavior(2)
};
btn.BeginAnimation(Button.WidthProperty, doubleAnimation);
```

Wird auf obiger DoubleAnimation die IsCumulative-Property auf true gesetzt, wird beim zweiten Durchlauf die Differenz der To- und From-Property (= 100) sowohl zur From- als auch zur To-Property addiert, wodurch sich für den zweiten Durchlauf der Startwert 200 und der Zielwert 300 ergeben. Folglich wird die Width-Property des Buttons flüssig vom Wert 100 zum Wert 300 animiert.

Was glauben Sie, was passiert, wenn zusätzlich – wie im folgenden Codeausschnitt – die AutoReverse-Property auf true gesetzt wird?

```
var doubleAnimation = new DoubleAnimation
{
  From = 100,
  To = 200,
  IsCumulative = true,
  AutoReverse = true,
  RepeatBehavior = new RepeatBehavior(2)
```



```
};
btn.BeginAnimation(Button.WidthProperty, doubleAnimation);
```

Die Width-Property des Buttons wird mit obiger Animation vom Wert 100 zum Wert 200 animiert, wird von 200 zurück zum Wert 100 animiert und springt dann abrupt auf den Wert 200, wird von dort zum Wert 300 animiert und läuft rückwärts zum Wert 200, wo die Animation endet.

15.2.9 Beschleunigen und Abbremsen

Normalerweise findet zwischen dem Start- und dem Zielwert einer Basis-Animation eine lineare Interpolation statt. Bei einer Animation vom Wert 100 zum Wert 200 und einer Dauer von einer Sekunde ist nach 0,1 Sekunden der Wert 110 erreicht, nach 0,2 Sekunden der Wert 120 usw. Die Animation läuft mit einer konstanten Geschwindigkeit ab.

In der realen Welt sind Animationen mit konstanten Geschwindigkeiten aufgrund physischer Kräfte wie Trägheit oder Gravitation kaum anzutreffen. Ein Auto fährt nicht gleich mit 100 km/h los, sondern wird beschleunigt. Ein Ball rollt nicht mit 100 km/h über den Fußballrasen und bleibt plötzlich liegen, vielmehr wird er immer langsamer.

Startet eine Animation bei der WPF mit konstanter Geschwindigkeit, wirkt dies meist etwas unnatürlich. Dies ändern Sie, indem Sie Ihre Animationen mit den in der Timeline-Klasse definierten Properties AccelerationRatio und DecelerationRatio beschleunigen und abbremsen.

Die Properties AccelerationRatio und DecelerationRatio setzen Sie auf einen Wert zwischen 0 und 1. Per Default sind beide Properties 0, was einer konstanten Geschwindigkeit entspricht. Die Summe aus beiden Werten darf 1 nicht überschreiten, ansonsten erhalten Sie eine InvalidOperationException.

> **Hinweis**
>
> Das Beschleunigen und Abbremsen hat keine Auswirkung auf die Dauer einer Animation. Lediglich der berechnete Wert zu einem bestimmten Zeitpunkt ist ein anderer. Der Zielwert wird unabhängig vom Beschleunigen oder Abbremsen immer zum selben Zeitpunkt erreicht.

Setzen Sie AccelerationRatio auf 1, wird vom Start bis zum Ende der Animation beschleunigt. Der ermittelte Wert nach 0,1 Sekunden für eine Animation von 100 nach 200 von einer Sekunde ist 101 (und nicht wie linear 110), nach 0,2 Sekunden 104 usw. Setzen Sie AccelerationRatio auf 0.5, wird vom Start bis zur Hälfte der Zeit (Duration-Property) beschleunigt, die andere Hälfte wird mit konstanter Geschwindigkeit durchgeführt. Setzen Sie die DecelerationRatio-Property auf 0.5, wird in der zweiten Hälfte der Animation abgebremst. AccelerationRatio beginnt also immer beim Start einer Animation, DecelerationRatio arbeitet immer auf das Ende hin. Abbildung 15.3 zeigt die Auswirkungen einiger Einstellungen.

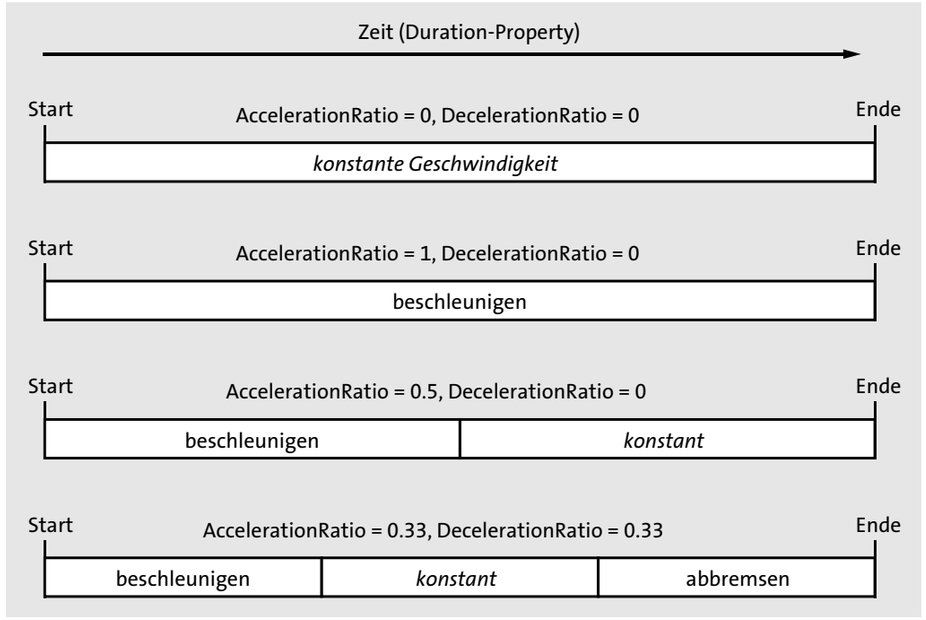

Abbildung 15.3 Auswirkungen von »AccelerationRatio« und »DecelerationRatio«

Hinweis

Haben Sie die `AccelerationRatio` auf `1` und die `AutoReverse`-Property auf `true` gesetzt, findet beim Rückwärtsgang der Animation eine Umkehrung der Beschleunigung statt. Eine solche negative Beschleunigung entspricht einem Bremsvorgang.

15.2.10 Das Füllverhalten einer Animation

Die letzte Property einer Timeline, die Sie in diesem Abschnitt kennenlernen, ist die Property `FillBehavior`. Sie ist vom Typ der Aufzählung `FillBehavior` und legt fest, was mit dem Wert der animierten Dependency Property passieren soll, wenn das Ende der Animation erreicht wurde. Die Aufzählung `FillBehavior` hat lediglich zwei Werte:

- **`HoldEnd`** – Die Animation behält den Wert nach dem Ende. Dies ist der Default-Wert der `FillBehavior`-Property und üblicherweise das gewünschte Verhalten. Wird die Breite eines Buttons vom Wert 100 zum Wert 200 animiert, bleibt er auch nach der Animation auf dem Wert 200 stehen.

- **`Stop`** – Erreicht die Animation das Ende, wird der Wert verworfen, und es wird wieder der aufgrund des Vorrangrechts für Dependency Properties gültige Wert ermittelt. Das ist üblicherweise der Wert, der vor der Animation bestand.

Tipp

In Abschnitt 7.2.12, »Den Wert einer Dependency Property ermitteln«, wurde das Vorrangsrecht beschrieben. Darin wurde gezeigt, dass Animationen Vorrang vor einem lokal gesetzten Wert haben. Dies muss so sein, da Animationen ansonsten einen lokalen Wert überhaupt nicht ändern könnten. Mit dem per Default definierten Wert `HoldEnd` für die `FillBehavior`-Property hält eine Animation den Wert auch fest, nachdem ihr Ende bereits überschritten wurde. Folglich zeigt das Setzen eines lokalen Wertes keine Auswirkungen, da der Wert der Animation Vorrang hat. Auf dem in diesem Abschnitt animierten Button wird die folgende Zeile nach einer durchgeführten Animation keine Auswirkung haben, da der lokale Wert im Vorrangsrecht von Dependency Properties hinter jenem der Animation liegt:

```
btn.Width = 40;
```

Um den lokalen Wert dennoch zu setzen, muss die Animation von der `Width`-Property entfernt werden. Dazu rufen Sie auf dem Button-Objekt die `BeginAnimation` auf und übergeben als zweiten Parameter für die `AnimationTimeline` einfach eine `null`-Referenz. Nach diesem Aufruf lässt sich der lokale Wert setzen:

```
btn.BeginAnimation(Button.WidthProperty, null);
btn.Width = 40;
```

Während ein lokaler Wert nicht mehr einfach gesetzt werden kann, ohne die Animation zu entfernen, ist eine neue Animation immer möglich. Die Methode `BeginAnimation` und auch die später beschriebene Methode `ApplyAnimationClock` besitzen beide eine Überladung, die als dritten Parameter einen Wert der Aufzählung `HandOffBehavior` entgegennimmt. Dieser Wert legt fest, was passiert, wenn für die zu animierende Property bereits eine Animation läuft. Die Aufzählung `HandOffBehavior` enthält nur zwei Werte:

- **`SnapshotAndReplace`** (Default) – Neue Animationen auf derselben Dependency Property ersetzen existierende Animationen.
- **`Compose`** – Neue Animationen werden mit existierenden kombiniert, indem sie einfach ans Ende angefügt werden.

Achtung

Wenn Sie `Compose` als `HandOffBehavior` verwenden, müssen Sie dafür sorgen, dass Sie die von den Animationen verwendeten `Clock`s auch wieder entfernen. Ansonsten kann es bei vielen `Clock`-Objekten zu Performance-Problemen kommen. Sie entfernen alle Animationen auf einer Property, indem Sie `BeginAnimation` oder `ApplyAnimationClock` aufrufen und als zweiten Parameter eine `null`-Referenz übergeben.

Um eine spezifische `AnimationClock` zu entfernen, rufen Sie die `Remove`-Methode auf dem `ClockController` auf. Diesen finden Sie in der `Controller`-Property der `AnimationClock`-Klasse.

15.2.11 Eine Animation mit »AnimationClock« steuern

Die bisher gezeigten Animationen hatten keinerlei Interaktionsmöglichkeit. In diesem letzten Abschnitt zu Basis-Animationen in C# erfahren Sie, wie Sie Animationen zur Laufzeit steuern.

Wie bereits in den Animationsgrundlagen zu Beginn dieses Kapitels erwähnt wurde, steckt hinter einer Animation ein `Clock`-Objekt, das den aktuellen Fortschritt und sonstige zeitliche Informationen kennt. Ein `Clock`-Objekt erlaubt das Steuern einer Animation.

Beim Aufruf von `BeginAnimation` wird intern automatisch ein `AnimationClock`-Objekt erstellt. Das `AnimationClock`-Objekt besitzt drei wichtige Properties: `CurrentTime`, `CurrentProgress` und `CurrentState`. In der `Timeline`-Property befindet sich eine Referenz der Timeline, von der diese `AnimationClock` erstellt wurde.

Hinweis

Die `CurrentState`-Property der `Clock`-Klasse ist vom Typ der Aufzählung `ClockState`. Diese definiert die Werte `Active`, `Filling` und `Stopped`. Gerade laufende Animationen mit dem Wert `Active` werden auch als *aktive Animationen* bezeichnet.

Eine aktive `AnimationClock` wird im Hintergrund vom Zeitmanager (TimeManager) aktualisiert, wodurch eine Animation fortschreitet. Der Zeitmanager aktualisiert alle aktiven `Clock`-Objekte in kurz hintereinander liegenden Zeitpunkten, die auch als *Ticks* (engl. für »Augenblicke«) bezeichnet werden. Der Zeitmanager tickt also wie ein Timer und aktualisiert mit jedem Tick alle aktiven `Clock`-Objekte. Die Anzahl der Ticks pro Sekunden hängt von der Leistungsfähigkeit des Systems ab. Wenn die aktualisierte `Clock` eine `AnimationClock` ist, holt sie den aktuellen Wert von der `GetCurrentValue`-Methode Ihrer Timeline. Wenn Sie während einer Animation auf Ihrem `DependencyObject` die Methode `GetValue` aufrufen, erhalten Sie den aktuellen Wert von der `AnimationClock`.

Wenn Sie mit `BeginAnimation` eine Animation starten, wird zwar intern ein `AnimationClock`-Objekt erzeugt, allerdings haben Sie keinen Zugriff auf dieses Objekt. Um ein `AnimationClock`-Objekt zu erhalten, rufen Sie auf Ihrer `AnimationTimeline` die `CreateClock`-Methode auf. Zum Starten der Animation rufen Sie auf dem zu animierenden Objekt nicht `BeginAnimation`, sondern die in `IAnimatable` definierte `ApplyAnimationClock`-Methode auf. Als ersten Parameter übergeben Sie die zu animierende Dependency Property, als zweiten Parameter das `AnimationClock`-Objekt, das Sie von `CreateClock` erhalten haben:


```
var doubleAnimation = new DoubleAnimation();
...
AnimationClock clock = doubleAnimation.CreateClock();
imgBall.ApplyAnimationClock(Canvas.TopProperty, clock);
```

Animationen lassen sich mit einem `AnimationClock`-Objekt steuern. Die Klasse `AnimationClock` besitzt eine Property `Controller`, die ein `ClockController`-Objekt zurückgibt, das letztlich das Steuern ermöglicht.

Hinweis

Bei verschachtelten Timelines gibt nur die `Controller`-Property der zur Wurzel-Timeline gehörenden `AnimationClock` einen `ClockController` zurück. Die Controller-Properties von `AnimationClocks`, die zu Kind-Timelines gehören, geben `null` zurück. Hier sind noch keine Timelines verschachtelt, folglich sind wir immer auf der Wurzel-Timeline.

Die Klasse `ClockController` hat nur zwei Properties: `Clock` enthält das `Clock`-Objekt, zu dem der `ClockController` gehört, `SpeedRatio` erlaubt das interaktive Steuern der Geschwindigkeit. Natürlich lässt sich mit einem `ClockController`-Objekt weitaus mehr machen, als die Geschwindigkeit zu ändern. Sie finden in der Klasse `ClockController` neben den zwei Properties einige Methoden:

- **`Begin`** – setzt die `Clock` beim nächsten Tick des Zeitmanagers wieder auf den Startzeitpunkt.
- **`Pause`** – stoppt den Fortschritt der `Clock`.
- **`Remove`** – entfernt die `Clock` und den `ClockController` von den Dependency Properties, die mit ihr animiert wurden. Lokale Werte lassen sich dann wieder setzen. Der Aufruf von `Remove` findet üblicherweise im `Completed`-Event der `AnimationClock` statt.
- **`Resume`** – Eine zuvor mit `Pause` gestoppte `Clock` läuft weiter.
- **`Seek`** – springt beim nächsten Tick des Zeitmanagers an eine bestimmte Stelle in einer Animation. `Seek` nimmt als ersten Parameter ein `TimeSpan`-Objekt entgegen und als zweiten einen Wert der Aufzählung `TimeSpanSeek`. `TimeSpanSeek` enthält lediglich zwei Werte: `BeginTime`, um relativ zur Anfangszeit (`BeginTime`-Property) zu suchen, oder `TimeSpanSeek.Duration`, um relativ zur tatsächlichen Dauer (`Duration`-Property) zu suchen.
- **`SeekAlignedToLastTick`** – springt sofort an eine bestimmte Stelle in einer Animation und wartet nicht auf den nächsten Tick des Zeitmanagers.
- **`SkipToFill`** – setzt die `CurrentTime`-Property der `Clock` an das Ende der aktiven Periode.
- **`Stop`** – stoppt die `Clock`.

Sehen wir uns einige der Methoden an einem kleinen Beispiel an. Listing 15.3 enthält ein Grid. Im Grid befindet sich ein Canvas mit zwei Image-Objekten. Das Image-Objekt mit dem Namen `imgBall` soll animiert werden. Die Anwendung enthält zur Steuerung der Animation

vier Buttons und einen Slider. Beachten Sie, dass alle vier Buttons denselben `Click`-Event-Handler haben.

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition/>
    <RowDefinition Height="Auto"/>
  </Grid.RowDefinitions>
  <Canvas Width="250" Height="185">
    <Image Height="185" Source="fussballthomas.png"
      Canvas.Left="40"/>
    <Image x:Name="imgBall" Width="25" Canvas.Top="10"
      Canvas.Left="140" Source="teamgeist.png"/>
  </Canvas>
  <DockPanel Grid.Row="1">
    <Button Margin="5" Click="Button_Click" Content="Start"/>
    <Button Margin="5" Click="Button_Click" Content="Stop"/>
    <Button Margin="5" Click="Button_Click" Content="Pause"/>
    <Button Margin="5" Click="Button_Click" Content="Weiter"/>
    <Slider x:Name="sli" Minimum="0.01" Maximum="2"
    AutoToolTipPlacement="BottomRight" AutoToolTipPrecision="2"/>
  </DockPanel>
</Grid>
```

Listing 15.3 Beispiele\K15\02 AnimationenKontrollierenCSharp\MainWindow.xaml

Abbildung 15.4 zeigt das User-Interface der Anwendung aus Listing 15.3. Mit den Buttons lässt sich der Fußball steuern, mit dem Slider die Geschwindigkeit bestimmen.

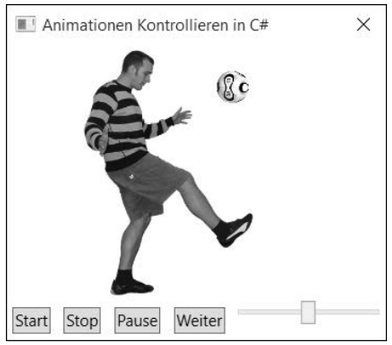

Abbildung 15.4 Gesteuerte Animation eines Fußballs

In der Codebehind-Datei wird im `Window_Loaded`-Event-Handler ein `DoubleAnimation`-Objekt erstellt, das zum Animieren der `Canvas.TopProperty` des `imgBall`-Objekts verwendet wird (sie-

he Listing 15.4). Beachten Sie, dass die `AccelerationRatio`-Property über den Wert `1` verfügt, wodurch der Ball beim Herunterfallen beschleunigt wird. `AutoReverse` ist `true`, wodurch die Animation rückwärtsläuft und der Ball wieder langsamer wird, wenn er sich vom Fuß weg nach oben bewegt. Auf dem `DoubleAnimation`-Objekt wird die `CreateClock`-Methode aufgerufen und das erhaltene `AnimationClock`-Objekt in der Klassenvariablen `_clock` gespeichert. Auf dem `imgBall`-Objekt wird die `ApplyAnimationClock`-Methode mit der `Canvas.TopProperty` und der `_clock`-Referenz aufgerufen. Der Slider wird zuletzt an die `SpeedRatio`-Property des `ClockControllers` gebunden.

Im `Button_Click`-Event-Handler wird, je nach geklicktem Button, auf dem `ClockController`-Objekt die entsprechende Methode aufgerufen, um die Animation zu steuern.

```
public partial class MainWindow : Window
{ ...
  private AnimationClock _clock = null;
  private void Window_Loaded(object sender, RoutedEventArgs e)
  {
    // Animation erstellen und starten, AnimationClock speichern
    var doubleAnimation = new DoubleAnimation
    {
      To = 110,
      RepeatBehavior = RepeatBehavior.Forever,
      AutoReverse = true,
      AccelerationRatio = 1,
      Duration = new Duration(TimeSpan.Parse("0:0:0.25"))
    };
    _clock = doubleAnimation.CreateClock();
    imgBall.ApplyAnimationClock(Canvas.TopProperty, _clock);
    // Slider an SpeedRatio des Controllers binden
    var binding = new Binding
    {
      Source = _clock.Controller,
      Path = new PropertyPath("SpeedRatio")
    };
    sli.SetBinding(Slider.ValueProperty, binding);
  }
  private void Button_Click(object sender, RoutedEventArgs e)
  {
    var btn = e.Source as Button;
    switch (btn.Content.ToString())
    {
      case "Stop":
        _clock.Controller.Stop();
```

```
        break;
      case "Pause":
        _clock.Controller.Pause();
        break;
      case "Start":
        _clock.Controller.Begin();
        break;
      case "Weiter":
        _clock.Controller.Resume();
        break;
      default:
        break;
    }
  }
}
```

Listing 15.4 Beispiele\K15\02 AnimationenKontrollierenCSharp\MainWindow.xaml.cs

15.2.12 Animationen in FriendStorage

Auch FriendStorage verwendet Basis-Animationen, um den Freunde-Explorer ein- und auszublenden (siehe Abbildung 15.5).

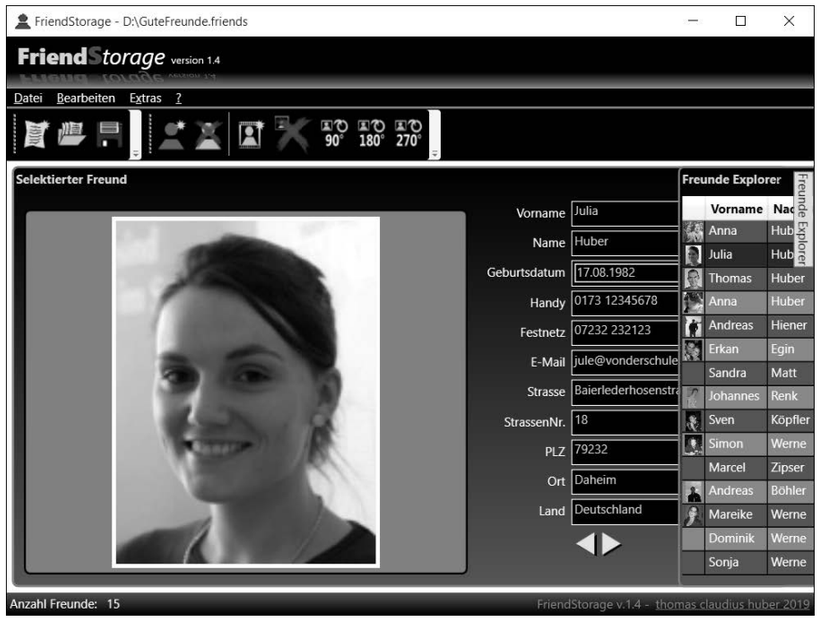

Abbildung 15.5 Der animierte Freunde-Explorer auf der rechten Seite

Zum Ausblenden des Freunde-Explorers wird die `X`-Property eines `TranslateTransform`-Objekts animiert, das sich in der `RenderTransform`-Property des Grids mit dem Freunde-Explorer befindet. Das Grid mit dem Freunde-Explorer wird durch die Animation des `TranslateTransform`-Objekts nach rechts außerhalb des sichtbaren Bereichs geschoben (siehe Listing 15.5). Im Event Handler für das `Completed`-Event der `DoubleAnimation` wird die `Visibility`-Property des Grids mit dem Freunde-Explorer (`layer1`) auf `Collapsed` gesetzt.

```
void HandleLayerOMouseEnter(object sender, RoutedEventArgs e)
{
  // layer1-Grid ausblenden
  if (!btnPinIt.IsChecked.GetValueOrDefault()
  && layer1.Visibility == Visibility.Visible)
  {
    // 1. Zielwert für die Animation setzen
    double to = layer1.ColumnDefinitions[1].Width.Value;
    // 2. layer1Trans.X zum ermittelten Zielwert animieren
    // und Event Handler für Completed-Event installieren
    var ani = new DoubleAnimation(to,
      new Duration(TimeSpan.FromMilliseconds(500)));
    ani.Completed += new Event Handler(ani_Completed);
   layer1Trans.BeginAnimation(TranslateTransform.XProperty, ani);
  }
}
void ani_Completed(object sender, EventArgs e)
{
  // 3. layer1-Grid ausblenden
  layer1.Visibility = Visibility.Collapsed;
}
```

Listing 15.5 Beispiele\FriendStorage\MainWindow.xaml.cs

Tipp

Am Ende von Kapitel 6, »Layout«, finden Sie eine ausführliche Beschreibung des Layouts von FriendStorage. Dort finden Sie zusätzlich zum animierten Ausblenden des Freunde-Explorers auch den Code zum animierten Einblenden.

15.3 Basis-Animationen in XAML

Um eine Animation zu starten, wird in C# die Methode `BeginAnimation` oder `ApplyAnimationClock` aufgerufen. Aus XAML sind keine Methodenaufrufe möglich. Dennoch ist es mög-

lich, Animationen vollständig in XAML zu erstellen. Die Animationsklassen besitzen ausreichend Properties, die sich einfach aus XAML setzen lassen:

```
<DoubleAnimation To="1" Duration="0:0:4"
  AutoReverse="True" RepeatBehavior="Forever"/>
```

Allerdings fehlt der obigen `DoubleAnimation` noch etwas: Sie kennt weder das Zielobjekt noch die Ziel-Property. In C# wird das Zielobjekt identifiziert, indem auf ihm die `BeginAnimation`- oder `ApplyAnimationClock`-Methode aufgerufen wird. Der Methode wird die Ziel-Property als Parameter übergeben. In XAML existiert eine andere Variante, die Sie gleich sehen werden.

Über die Definition von Zielobjekt und Ziel-Property hinaus stellt sich die grundlegende Frage, wo das `DoubleAnimation`-Element überhaupt untergebracht wird. Dazu gibt es in XAML nur eine Möglichkeit, nämlich in einem Trigger. Jeder Trigger kann sogenannte `TriggerActions` enthalten, die eine Animation starten können.

Alle Trigger erben von `TriggerBase` eine `EnterActions`- und eine `ExitActions`-Property vom Typ `TriggerAction`. `EnterActions` wird aufgerufen, wenn ein Trigger aktiviert wird, `ExitActions`, wenn er deaktiviert wird. Die Klasse `EventTrigger` besitzt eine `Actions`-Property, die ebenfalls vom Typ `TriggerAction` ist.

Von der abstrakten Klasse `TriggerAction` gibt es drei direkte Subklassen (siehe Abbildung 15.6). Die Klasse `SoundPlayerAction` werden Sie in Kapitel 16, »Audio und Video«, kennenlernen. Sie wird verwendet, um *.wav*-Dateien abzuspielen. Die Klasse `BeginStoryboard` dient zum Starten eines Storyboards und ist das, was wir hier benötigen. Von der abstrakten Klasse `ControllableStoryboardAction` leiten einige Klassen ab, um eine mit `BeginStoryboard` gestartete Animation zu steuern.

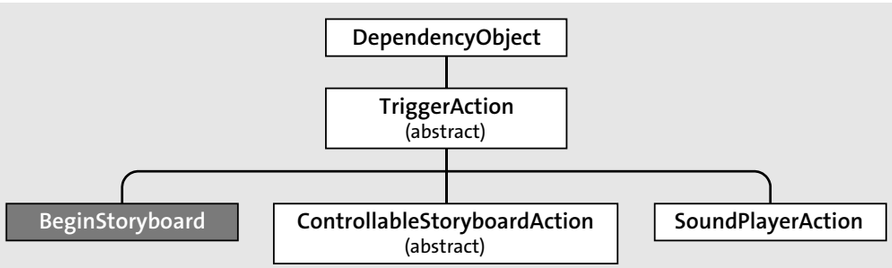

Abbildung 15.6 TriggerActions in der Klassenhierarchie der WPF

Die Klasse `BeginStoryboard` besitzt die Property `Storyboard` vom Typ `Storyboard`. Ihr weisen Sie das Storyboard zu, das durch dieses `BeginStoryboard`-Objekt gestartet werden soll. Klären wir, was genau ein Storyboard ist.

Von der abstrakten Klasse `Timeline` gibt es drei Subklassen (siehe Abbildung 15.7). `AnimationTimeline` und deren Subklassen (`<Typ>AnimationBase`) haben Sie bereits kennengelernt. `MediaTimeline` dient dem Abspielen von Audio und Video. `TimelineGroup` ist selbst wieder abstrakt und definiert eine `Children`-Property vom Typ `TimelineCollection`. `Storyboard` leitet über `ParallelTimeline` von `TimelineGroup` ab und besitzt somit auch die `Children`-Property und kann mehrere Timelines enthalten.

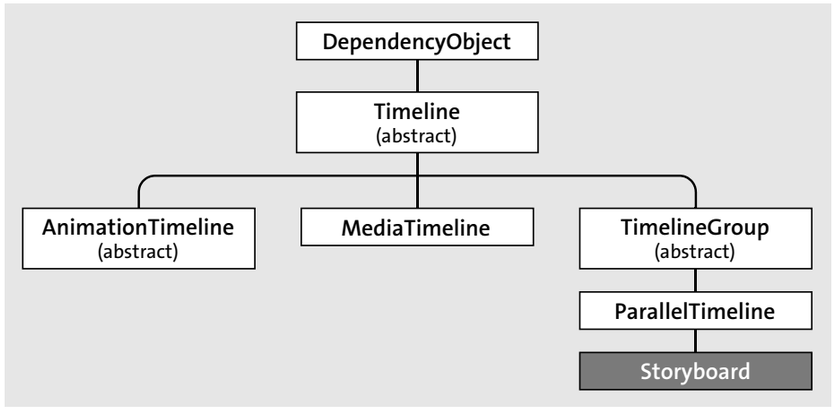

Abbildung 15.7 Die Klasse »Storyboard« in der Klassenhierarchie der WPF

Die Klasse `Storyboard` besitzt neben der aus `TimelineGroup` geerbten `Children`-Property die Attached Properties `TargetName` (vom Typ `string`) und `TargetProperty` (vom Typ `PropertyPath`). Mit ihnen definieren Sie auf einer im `Storyboard` enthaltenen `AnimationTimeline` das Zielobjekt und die zu animierende Property. Beachten Sie, dass die `TargetProperty`-Property vom Typ `PropertyPath` ist. Es lassen sich somit für die zu animierende Property auch komplexe Pfade angeben, wie sie vom Data Binding her bekannt sind.

Hinweis

Ein Storyboard kann wiederum weitere Timelines und somit auch weitere Storyboards enthalten, wodurch sich eine komplexe Hierarchie von Timelines schaffen lässt. Obwohl Sie innerhalb eines Storyboards weitere Storyboards erstellen können, sollten Sie dies nicht tun. Verwenden Sie innerhalb eines Storyboards für tiefere Verschachtelungen von Timelines die Klasse `ParallelTimeline`, die im Gegensatz zur Klasse `Storyboard` etwas schlanker ist.

Mit den Klassen `BeginStoryboard` und `Storyboard` »im Werkzeugkoffer« lassen sich jetzt die ersten Animationen in XAML erstellen.

15.3.1 Eine einfache Animation in XAML

Um eine einfache Animation in XAML zu starten, wird eine `TriggerAction` vom Typ `BeginStoryboard` den `EnterActions` oder `ExitActions` eines `Triggers` zugewiesen. Der EventTrigger unterscheidet sich von den anderen Triggern: Er unterstützt nur seine eigene `Actions`-Property vom Typ `TriggerAction`. Diese sehen wir uns jetzt an.

In Listing 15.6 wird ein `BeginStoryboard`-Element der `Actions`-Property (die als Content-Property gesetzt ist) eines EventTriggers zugewiesen. Der EventTrigger lauscht auf das `Loaded`-Event des Grids. Das `BeginStoryboard` enthält das `Storyboard`-Element, das wiederum eine `DoubleAnimation` enthält. Beachten Sie, dass auf dem `DoubleAnimation`-Objekt die Attached Properties `Storyboard.TargetName` und `Storyboard.TargetProperty` gesetzt sind, um das Zielobjekt und die zu animierende Dependency Property festzulegen.

Hinweis

Um in XAML eine Animation zu starten, müssen Sie immer ein Storyboard verwenden.

Im Grid befinden sich zwei Image-Objekte, die übereinandergezeichnet werden. Beide Image-Objekte enthalten ein fast identisches Bild, das ein Glas zeigt: einmal leer und einmal voll. Durch die Animation der `Opacity`-Property des oberen Bildes ergibt sich ein Fülleffekt des Glases (siehe Abbildung 15.8).

```
<Grid Width="380" Height="214" Background="Black">
  <Grid.Triggers>
    <EventTrigger RoutedEvent="Grid.Loaded">
      <BeginStoryboard>
        <Storyboard>
          <DoubleAnimation Storyboard.TargetName="bierVoll"
            Storyboard.TargetProperty="Opacity" To="1"
            Duration="0:0:4" AutoReverse="True"
            RepeatBehavior="Forever"/>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Grid.Triggers>
  <Image Source="bierGlasLeer.jpg"/>
  <Image Source="bierGlasVoll.jpg" x:Name="bierVoll"
    Opacity="0"/>
</Grid>
```

Listing 15.6 Beispiele\K15\03 DasErsteBier\MainWindow.xaml

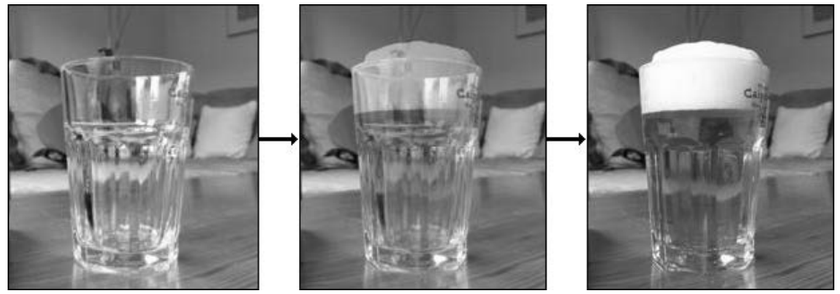

Abbildung 15.8 Animation der Opacity-Property

In Listing 15.6 wurden sowohl die `TargetName`- als auch die `TargetProperty`-Property gesetzt. Die `TargetName`-Property ist optional. Ist sie nicht angegeben, wird die Animation auf dem Element ausgeführt, das den Trigger besitzt. In Listing 15.6 ist dieses Element das Grid. Aber in Listing 15.6 soll die Animation ja auf dem Image mit dem Namen `bierVoll` durchgeführt werden. Folglich muss dazu die `TargetName`-Property gesetzt werden.

Hinweis

Sie finden in den Buch-Beispielen im Ordner *Beispiele\K15\04 DasZweiteBier.xaml* noch eine Animation, die mehrere Bilder und somit mehrere Zustände des Bierglases zeigt.

Tipp

Für die `Duration`-Property existiert ein Type-Converter (`DurationConverter`), der die Angabe eines `TimeSpan`-Strings oder der Strings `Automatic` oder `Forever` erlaubt.

Für die `RepeatBehavior`-Property besteht ebenfalls ein Type-Converter (`RepeatBehaviorConverter`). Sie können für `RepeatBehavior` einen `TimeSpan`-String, den String `Forever` oder die Anzahl der Wiederholungen angeben. Letzteres tun Sie, indem Sie einen `double`-Wert angeben, auf den ein `x` folgt. `2x` bedeutet »zweimal durchlaufen«, `1.5x` »eineinhalbmal«.

Anstatt direkt die aus `FrameworkElement` geerbte `Triggers`-Property zu verwenden, werden Animationen auch oft in einem `Style` untergebracht. Dann wird die `TargetName`-Property nicht gesetzt, sondern lediglich die `TargetProperty`-Property. Listing 15.7 zeigt einen Codeausschnitt, den Sie bereits aus Kapitel 11, »Styles, Trigger und Templates«, kennen. Der Style für Image-Objekte enthält zwei EventTrigger. Beachten Sie, dass sich die Attached Property `TargetProperty` auch direkt auf dem Storyboard setzen lässt, wodurch darin liegende Timelines automatisch diese `TargetProperty` verwenden, wenn Sie nicht explizit einen anderen Wert für die `TargetProperty`-Property definieren.

```
<StackPanel Orientation="Horizontal">
  <StackPanel.Resources>
    <Style TargetType="Image">
      <Setter Property="Width" Value="100"/>
      <Setter Property="Margin" Value="2"/>
      <Style.Triggers>
        <EventTrigger RoutedEvent="MouseEnter">
          <BeginStoryboard>
            <Storyboard TargetProperty="Width">
              <DoubleAnimation To="120" Duration="0:0:0.5"
                DecelerationRatio="1"/>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
        <EventTrigger RoutedEvent="MouseLeave">
          <BeginStoryboard>
            <Storyboard TargetProperty="Width">
              <DoubleAnimation To="100" Duration="0:0:0.5"
                DecelerationRatio="1"/>
            </Storyboard>
          </BeginStoryboard>
        </EventTrigger>
      </Style.Triggers>
    </Style>
  </StackPanel.Resources>
  <Image Source="fontaene.jpg"/>
  <Image Source="thomas.jpg"/>
  <Image Source="wasserfall.jpg"/>
  <Image Source="sandfigur.jpg"/>
  <Image Source="schwein.jpg"/>
</StackPanel>
```

Listing 15.7 Beispiele\K15\05 EventTriggerInStyle\MainWindow.xaml

Die Bilder im StackPanel aus Listing 15.7 werden beim Event `MouseEnter` gezoomt und beim Event `MouseLeave` wieder verkleinert (siehe Abbildung 15.9). Beachten Sie, dass in Listing 15.7 beide `DoubleAnimation`-Objekte keine `From`-Property setzen. Es wird somit immer die aktuelle `Width`-Property genutzt. Die Animation ist dadurch auch dann flüssig, wenn die Maus schnell über die Bilder hinwegbewegt wird.

Abbildung 15.9 Animierte Image-Objekte, die auf »MouseOver« reagieren

In Listing 15.7 wurden im `Image`-Style zwei EventTrigger für die Events `MouseEnter` und `MouseLeave` erstellt. Das gleiche Ergebnis lässt sich auch mit einem Property-Trigger für die `IsMouseOver`-Property erzielen. Die Animationen werden dann in der `EnterActions`- und `ExitActions`-Property gesetzt. Der Property-Trigger in Listing 15.8 zeigt die gleiche Funktion wie die beiden EventTrigger aus Listing 15.7. Ob Sie den Style aus Listing 15.7 oder den aus Listing 15.8 verwenden, ist reine Geschmacksache.

```
<Style.Triggers>
  <Trigger Property="IsMouseOver" Value="True">
    <Trigger.EnterActions>
      <BeginStoryboard>
        <Storyboard TargetProperty="Width">
          <DoubleAnimation To="120" Duration="0:0:0.5"
            DecelerationRatio="1"/>
        </Storyboard>
      </BeginStoryboard>
    </Trigger.EnterActions>
    <Trigger.ExitActions>
      <BeginStoryboard>
        <Storyboard TargetProperty="Width">
          <DoubleAnimation To="100" Duration="0:0:0.5"
            DecelerationRatio="1"/>
        </Storyboard>
      </BeginStoryboard>
    </Trigger.ExitActions>
  </Trigger>
</Style.Triggers>
```

Listing 15.8 Beispiele\K15\06 PropertyTriggerInStyle\MainWindow.xaml

15.3.2 Das Storyboard als Timeline-Container

Bisher wurde zur `Children`-Property des Storyboards nur eine `Timeline` bzw. eine `DoubleAnimation` hinzugefügt. Mit mehreren Timelines lassen sich komplexere Animationen erstellen. In den Anfangszeiten von Adobe Flash (das damals noch von Macromedia entwickelt wurde) gab es viele Internetseiten, die eine Art Intro hatten, bei dem Texte wie am Anfang eines Kinofilms abliefen. Mit einem Storyboard und ein paar darin enthaltenen Timelines ist dieser Effekt auch mit der WPF einfach zu realisieren.

Listing 15.9 enthält ein Grid mit vier TextBoxen namens `txt1`, `txt2`, `txt3` und `txt4`. Diese werden durch das Storyboard im EventTrigger des Grids nacheinander eingeblendet (siehe Abbildung 15.10). Beachten Sie in Listing 15.9, dass die einzelnen `DoubleAnimation`-Objekte unterschiedliche Werte für die `BeginTime`-Property enthalten, wodurch sie zu verschiedenen Zeitpunkten starten.

```
<Grid Width="300" Height="100" Background="Black">
  <Grid.Resources>
    <Style TargetType="TextBlock">
      <Setter Property="HorizontalAlignment" Value="Center"/>
      <Setter Property="VerticalAlignment" Value="Center"/>
      <Setter Property="Foreground" Value="White"/>
      <Setter Property="Opacity" Value="0"/>
      <Setter Property="FontSize" Value="14"/>
    </Style>
  </Grid.Resources>
  <Grid.Triggers>
    <EventTrigger RoutedEvent="Grid.Loaded">
      <BeginStoryboard>
        <Storyboard TargetProperty="Opacity">
          <DoubleAnimation To="1" BeginTime="0:0:2"
            Duration="0:0:2" AutoReverse="True"
            Storyboard.TargetName="txt1"
            DecelerationRatio="1"/>
          <DoubleAnimation BeginTime="0:0:6" To="1"
            Duration="0:0:2" AutoReverse="True"
            Storyboard.TargetName="txt2"
            DecelerationRatio="1"/>
          <DoubleAnimation BeginTime="0:0:12" To="1"
            Duration="0:0:2" Storyboard.TargetName="txt3"
            DecelerationRatio="1"/>
          <DoubleAnimation BeginTime="0:0:14" To="1"
            Storyboard.TargetName="txt4"
            DecelerationRatio="1"/>
        </Storyboard>
      </BeginStoryboard>
    </EventTrigger>
  </Grid.Triggers>
  <TextBlock  Name="txt1" Text="Rheinwerk Verlag präsentiert"/>
  <TextBlock Name="txt2"
    Text="eine Thomas Claudius Huber Produktion"/>
  <StackPanel Height="40">
    <TextBlock Name="txt3" FontWeight="Bold" FontSize="16"
      Text="Windows Presentation Foundation"/>
    <TextBlock Name="txt4" Text="das umfassende Handbuch"/>
  </StackPanel>
</Grid>
```

Listing 15.9 Beispiele\K15\07 FilmIntro\MainWindow.xaml

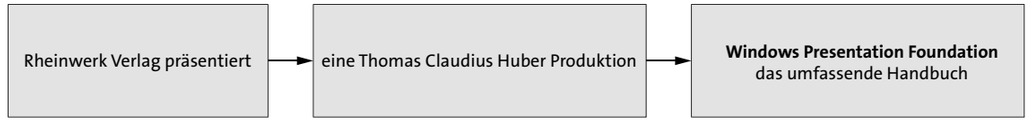

Abbildung 15.10 Ein Trailer, wie man ihn aus den Neunzigerjahren von Flash-Intros kennt

Tipp

Da das Storyboard selbst auch eine Timeline ist, lassen sich darauf auch Properties wie `RepeatBehavior` setzen. Die Dauer eines Storyboards richtet sich automatisch immer nach dem Ende der längsten Kind-Timeline.

Die `Duration`-Property mit dem Wert `Duration.Automatic` bedeutet für das Storyboard in dem Fall nicht mehr eine Dauer von einer Sekunde, sondern eine Dauer, die sich automatisch nach der am längsten andauernden Timeline richtet.

Mit der Möglichkeit, Timelines zu verschachteln, wird die Klasse `Storyboard` natürlich auch für Animationen in C# interessant. Um die Animation aus Listing 15.9 in C# zu erstellen, erzeugen Sie ein `Storyboard`-Objekt, setzen darauf die `TargetName`-Property und fügen anschließend die einzelnen `DoubleAnimation`-Objekte zur `Children`-Property hinzu:

```
var sb = new Storyboard();
sb.SetValue(Storyboard.TargetPropertyProperty,"Opacity");
var da =new DoubleAnimation();
da.SetValue(Storyboard.TargetNameProperty, "txt1");
...
sb.Children.Add(da);
```

Nachdem Sie das Storyboard und all die Kinder initialisiert haben, stellt sich die Frage, wie Sie die Animation beziehungsweise das Storyboard starten. Die `BeginAnimation`-Methode nimmt eine `AnimationTimeline` entgegen, aber `Storyboard` ist keine solche. `AnimationTimeline` und `Storyboard` sind lediglich beide vom Typ `Timeline`. Dabei definiert `AnimationTimeline` die eigentliche Animation für eine bestimmte Property eines bestimmten Objekts. Das Storyboard ist »nur« ein Container für mehrere Timelines.

Die Klasse `Storyboard` besitzt Methoden wie `Begin`, `Pause` und `Stop`. Rufen Sie die `Begin`-Methode auf, um das Storyboard zu starten. Dabei geben Sie im einfachsten Fall ein FrameworkElement mit, das sich im selben `NameScope` befinden muss wie die Zielobjekte der im Storyboard enthaltenen Animationen. Hat eine Animation im Storyboard kein Zielobjekt gesetzt, wird das an die `Begin`-Methode übergebene Objekt als Zielobjekt verwendet. Nach dem Aufruf von `Begin` startet das Storyboard. Im Fall von Listing 15.9 ist das Grid das Zielobjekt:

```
sb.Begin(grid);
```

[+]

Tipp

Die Klassen `FrameworkElement` und `FrameworkContentElement` besitzen beide mehrere Überladungen einer Methode namens `BeginStoryboard`. Diese nimmt im einfachsten Fall ein `Storyboard` entgegen und kapselt lediglich den oben dargestellten Aufruf der `Begin`-Methode eines Storyboards. Statt

```
sb.Begin(grid);
```

ist somit auch folgende Zeile möglich:

```
grid.BeginStoryboard(sb);
```

`BeginStoryboard` von `FrameworkElement` macht intern nichts, außer `Begin` auf dem übergebenen `Storyboard` aufzurufen. Als Parameter an `Begin` wird die `FrameworkElement`-Instanz selbst übergeben (`this`).

Ihnen sollte jetzt klar sein, warum in XAML um das Storyboard noch das `BeginStoryboard`-Element benötigt wird, das auf den ersten Blick überflüssig erscheint. Das `Storyboard`-Element erstellt das `Storyboard`, und `BeginStoryboard` sorgt dafür, dass beim Aktivieren des Triggers die `Begin`-Methode des Storyboards aufgerufen und dadurch die Animation gestartet wird.

15.3.3 Animationen mit »ControllableStoryboard« steuern

Auch XAML bietet die Möglichkeit, Animationen zu steuern. Neben der Klasse `BeginStoryboard` erbt auch die abstrakte Klasse `ControllableStoryboardAction` von `TriggerAction`. Von `ControllableStoryboardAction` sind sieben Klassen abgeleitet, die alle die Aufrufe der Methoden (`Pause`, `Stop`, `Remove` etc.) des in `BeginStoryboard` angegebenen `Storyboard`-Objekts kapseln:

- `PauseStoryboard`
- `RemoveStoryboard`
- `ResumeStoryboard`
- `SeekStoryboard`
- `SetStoryboardSpeedRatio`
- `SkipStoryboardToFill`
- `StopStoryboard`

Damit das Ganze funktioniert, setzen Sie lediglich die `Name`-Property Ihres `BeginStoryboard`-Elements. Die abstrakte Klasse `ControllableStoryboardAction` besitzt die Property `BeginStoryboardName` (Typ `String`). Diese Property setzen Sie beispielsweise auf einem `PauseStoryboard`-Element auf den String, den Sie auch in der `Name`-Property des `BeginStoryboard`-Elements angegeben haben. Schon ist die Verbindung zwischen `BeginStoryboard` und den `ControllableStoryboards` geschaffen, und Animationen lassen sich auch rein in XAML mit Triggern steuern.

Listing 15.10 enthält wieder die bereits im C#-Abschnitt verwendete Fußball-Animation (siehe Abbildung 15.11). Das Grid in Listing 15.10 enthält mehrere Trigger, die auf das Routed Event `Button.Click` reagieren. Je nach geklicktem Button wird die entsprechende `ControllableTriggerAction` ausgeführt.

```
<Grid> ...
<Grid.Triggers>
  <EventTrigger RoutedEvent="Button.Click"
    SourceName="btnStart">
    <BeginStoryboard Name="beginStoryboard">
      <Storyboard TargetProperty="(Canvas.Top)"
        TargetName="imgBall">
        <DoubleAnimation AutoReverse="True" From="10" To="110"
          RepeatBehavior="Forever" Duration="0:0:0.25"
          AccelerationRatio="1"/>
      </Storyboard>
    </BeginStoryboard>
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.Click" SourceName="btnStop">
    <StopStoryboard BeginStoryboardName="beginStoryboard"/>
  </EventTrigger>
  <EventTrigger RoutedEvent="Button.Click" SourceName="btnPause">
    <PauseStoryboard BeginStoryboardName="beginStoryboard"/>
  </EventTrigger>
 <EventTrigger RoutedEvent="Button.Click" SourceName="btnResume">
    <ResumeStoryboard BeginStoryboardName="beginStoryboard"/>
```

```
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.Click" SourceName="btn2x">
      <SetStoryboardSpeedRatio SpeedRatio="2"
        BeginStoryboardName="beginStoryboard"/>
    </EventTrigger>
    <EventTrigger RoutedEvent="Button.Click" SourceName="btn1x">
      <SetStoryboardSpeedRatio SpeedRatio="1"
        BeginStoryboardName="beginStoryboard"/>
    </EventTrigger>
  </Grid.Triggers>
  <Canvas Width="250" Height="185">
    <Image Height="185" Source="fussballthomas.png"
      Canvas.Left="40"/>
    <Image x:Name="imgBall" Width="25" Canvas.Top="10"
      Canvas.Left="140" Source="teamgeist.png"/>
  </Canvas>
  <DockPanel Grid.Row="1" LastChildFill="False">
    <Button Margin="5" x:Name="btnStart" Content="Start"/>
    <Button Margin="5" x:Name="btnStop" Content="Stop"/>
    <Button Margin="5" x:Name="btnPause" Content="Pause"/>
    <Button Margin="5" x:Name="btnResume" Content="Weiter"/>
    <Button Margin="5" x:Name="btn1x" Content="1x"/>
    <Button Margin="5" x:Name="btn2x" Content="2x"/>
  </DockPanel>
</Grid>
```

Listing 15.10 Beispiele\K15\08 AnimationKontrollierenInXAML\MainWindow.xaml

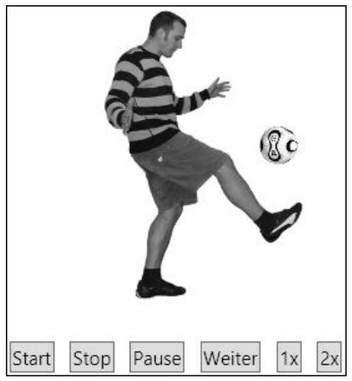

Abbildung 15.11 Gesteuerte Animation eines Fußballs – rein in XAML