HANSER

Leseprobe

Jörg Wegener

WPF 4.5 und XAML

Grafische Benutzeroberflächen für Windows inkl. Entwicklung von Windows Store Apps

Herausgegeben von Dr. Holger Schwichtenberg

ISBN (Buch): 978-3-446-43467-7

ISBN (E-Book): 978-3-446-43541-4

Weitere Informationen oder Bestellungen unter

http://www.hanser-fachbuch.de/978-3-446-43467-7

sowie im Buchhandel.

3 Layout

Das Erscheinungsbild einer Anwendung wird maßgeblich vom verwendeten Layout beeinflusst. Es bestimmt darüber, in welcher Weise Steuerelemente auf der Oberfläche einer Anwendung positioniert und ausgerichtet werden. Es ist ebenfalls dafür verantwortlich, dass zur Laufzeit auf Veränderungen reagiert wird, die sich auf die Inhalte auswirken können. Das kann beispielsweise bei Änderungen der Fenstergröße oder der Inhalte der Fall sein.

Im Idealfall bemerkt eine Anwendung diese Änderungen automatisch und reagiert darauf mit einer Anpassung oder einer Umpositionierung der betroffenen Steuerelemente. Soll das Fenster beispielsweise in einen 1/3- und einen 2/3-Bereich unterteilt werden, so müssten bei einer Änderung der Fenstergröße auch die Größen und Positionen der Steuerelemente neu berechnet werden. Dieser Aufwand, der allein für die Ausrichtung oder die Aufteilung der Steuerelemente notwendig ist, sollte wohl heutzutage nicht mehr zum Alltag eines Entwicklers gehören. Daher gibt es bereits in verschiedenen Plattformen Unterstützung für derartige Aufgaben, die aber auch meist sehr unterschiedliche Ansätze verfolgt haben. Meist wurde nur ein einziges Layout unterstützt, das (wenn überhaupt) nur mittels Parameter angepasst werden konnte. Unter Windows Forms konnten Steuerelemente nur über ein einfaches Koordinatensystem absolut in einem Fenster positioniert werden (einzige Ausnahme bildete hier das `FlowPanel`). Höhere Flexibilität erreichte man durch zusätzliche Eigenschaften oder Panels, die zwar viel Programmieraufwand vermieden haben, doch im Ergebnis immer noch recht unflexibel waren. Das `SplitPanel` konnte beispielsweise für eine automatische Unterteilung der Oberfläche verwenden werden, doch hatten diese Ansätze immer den Nachteil, dass sie nur die Lösung für eine bestimmte Aufgabenstellung waren und nicht das Problem grundsätzlich gelöst haben.

3.1 Panels für das Layout verwenden

In der WPF versuchte man, das starre Konstrukt des Layouts regelrecht aufzubrechen. Es sollte dem Entwickler nicht mehr länger ein einziges Layout angeboten werden, das sich starr und unflexibel verhält. Stattdessen werden ihm gleich mehrere unterschiedliche Layouts angeboten. Jedes dieser Layouts ist dabei sehr einfach gehalten und immer nur auf eine

bestimmte Art der Positionierung und Ausrichtung spezialisiert. Größere Anpassungen im Verhalten sind somit nur begrenzt möglich. Diese sind aber auch kaum notwendig, da man die Flexibilität vor allem durch das geschickte Kombinieren unterschiedlicher Layouts erhält. Selbst das Erstellen eigener, neuer Layouts ist problemlos möglich, wenngleich diese Möglichkeit bei der angebotenen Flexibilität eher die Ausnahme bleiben sollte.

Die Layouts finden sich in Form von sog. *Panels* wieder. Diese positionieren und richten die Steuerelemente immer auf ihre eigene Art und Weise aus. Ein `StackPanel` stapelt beispielsweise grundsätzlich die Steuerelemente der Reihe nach auf, während bei einem `DockPanel` diese an die jeweiligen Seiten des Fensters angedockt werden, wie man es von Symbolleisten oder Werkzeugleisten kennt.

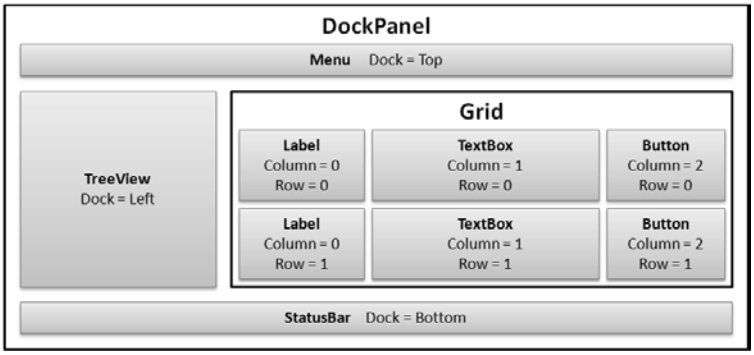

Bild 3.1 Unterschiedliche Layout-Systeme können problemlos miteinander kombiniert werden und so ein komplexeres Layout ergeben.

In Bild 3.1 ist exemplarisch der Aufbau verschiedener Panels zu sehen, die in Kombination das Layout einer Anwendung definieren. Hier wurde zuerst ein `DockPanel` zur Ausrichtung der grundlegenden Steuerelemente der Navigation verwendet (`Menu`, `TreeView` und `StatusBar`). Der verbleibende Platz auf der rechten Seite wird vom Panel `Grid` genutzt, das die Steuerelemente in tabellarischer Form ausrichtet.

Listing 3.1 Die Struktur der Anwendung in XAML

```
<DockPanel>
  <Menu DockPanel.Dock="Top" />
  <TreeView DockPanel.Dock="Left" />
  <StatusBar DockPanel.Dock="Bottom" />
  <Grid>
    <!-- etc. -->
  </Grid>
</DockPanel>
```

Ein Panel tritt in der WPF grundsätzlich für den Benutzer nicht in Erscheinung. Dadurch unterscheidet es sich grundlegend von den Panels aus Windows Forms, bei denen sich ein Panel meist als Box oder als Container auf dem Bildschirm präsentierte. Das Einzige, was der Benutzer von einem Panel in der WPF sieht, ist das Ergebnis der Ausrichtung und der Positionierung der dort enthaltenen Steuerelemente.

Selbstverständlich wird auch weiterhin das Layout durch den Zweck der Anwendung bestimmt: Eine Datenbankanwendung wird eher Formulare und Listen verwenden, wohingegen eine Multimedia-Anwendung meist andere Möglichkeiten der Darstellung nutzen wird. Um dieser breiten Palette an Szenarien gerecht zu werden, können die unterschiedlichen Panels problemlos miteinander kombiniert oder ineinander verschachtelt werden. Ein `StackPanel` kann so die Steuerelemente in einem `Grid` stapeln, das wiederum von einem `DockPanel` an den linken Seitenrand ausgerichtet wird.

Die Verwendung von Panels ist keineswegs optional. Damit in einem Fenster mehr als nur ein einzelnes Steuerelement platziert werden kann, wird ein Panel für die Ausrichtung benötigt. Ansonsten wüsste das Fenster nicht, in welcher Weise die Steuerelemente positioniert und ausgerichtet werden sollten. Die Klasse `Window` bietet daher für ihren Inhalt lediglich die Eigenschaft `Content` an – und die ist sogar nur vom Typ `object`. Dadurch ist sie zwar nicht auf einen bestimmten Typ festgelegt, kann dadurch aber auch nur ein einzelnes Steuerelement aufnehmen. Damit in einem Fenster mehrere Steuerelemente platziert werden können, ist de facto die Verwendung eines Panels notwendig.

Listing 3.2 Inhalte eines Fensters manuell im Programmcode zuweisen

```
Window win = new Window();
StackPanel stackpanel1 = new StackPanel();
stackpanel1.Children.Add(new Button());
stackpanel1.Children.Add(new Label());
win.Content = stackpanel1;
```

Wie in dem Beispielcode zu sehen ist, müssen die Inhalte zuerst einem Panel zugewiesen werden, das für die Ausrichtung und die Positionierung verantwortlich sein soll. Erst dieses Panel kann als Inhalt dem Fenster zugewiesen werden. Das Erstellen und Zuweisen von Steuerelementen im Programmcode wird im Alltag vermutlich seltener vorkommen. Dafür konnte man in diesem Beispiel deutlich erkennen, dass man mit einem Panel fast beliebig viele Steuerelemente platzieren kann. Oder metaphorisch ausgedrückt: Wenn die WPF eine gute Fee ist, erhält man mit einem Fenster nur einen einzelnen freien Wunsch. Entscheidet man sich statt eines Steuerelementes für ein Panel, so erhält man dagegen beliebig viele weitere freie Wünsche.

Die Klasse `Window` wurde von `ContentControl` abgeleitet, in der auch die Eigenschaft `Content` definiert worden ist. Neben der Klasse `Window` sind auch weitere Klassen von `ContentControl` spezialisiert, wie beispielsweise die Steuerelemente `Button` oder `GroupBox`. Die Verwendung von Panels zieht sich daher wie ein roter Faden durch das gesamte System. Wann immer über die Eigenschaft `Content` nur ein einfacher Inhalt zugewiesen werden kann, kann man sich ein Panel zu Hilfe nehmen.

In Bild 3.2 ist der Aufbau mehrerer Panels im Vergleich zu sehen. Hierbei wird das `DockPanel` für eine grundlegende Einteilung des Fensters verwendet. Hierüber wird als Erstes die `ToolBar` an den oberen Seitenrand angedockt. Der restliche Platz im Fenster wird mit einem Grid in zwei Bereiche (bzw. Zellen) aufgeteilt.

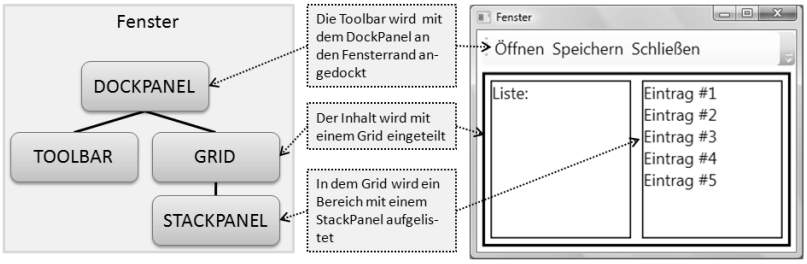

Bild 3.2 Theorie und Praxis im Vergleich: links die Struktur und rechts das Ergebnis

Der Umstieg mag zu Beginn etwas schwerfallen, da gewohnte Techniken radikal über Bord geworfen wurden. Gab es unter Windows Forms ausschließlich die absolute Positionierung, so hat man es hier mit einer Vielzahl an neuen Layouts zu tun. Die meisten Features existieren aber auch weiterhin in der WPF – sie sind nur entweder anders umgesetzt, oder sie sind an anderer Stelle wieder zu finden. Eine genaue Übersetzung der Features ist jedoch nicht möglich, da die Unterschiede zwischen Windows Forms und der WPF zu gravierend sind. Dennoch soll die folgende Auflistung einen kleinen Überblick der Veränderungen geben.

Tabelle 3.1 Positionierung und Ausrichtung im Vergleich mit Windows Forms

Windows Forms	Windows Presentation Foundation
Absolute Positionierung	Das Panel `Canvas` eignet sich als Ersatz. Absolute Positionierung ist aber grundsätzlich das Mittel zweiter Wahl. In erster Linie sollten die neuen Layouts verwendet werden, mit denen die Steuerelemente automatisch ausgerichtet und positioniert werden. Hierfür stehen mehrere Panels zur Verfügung, wie beispielsweise das `Grid` oder das `DockPanel`.
Dock-Eigenschaft	Die Eigenschaft `Dock` gibt es nicht mehr. Stattdessen wird diese Aufgabe vom `DockPanel` übernommen, das die Steuerelemente an die entsprechenden Seiten des Fensters andockt.
Anchor-Eigenschaft	Anchors existieren in der WPF in dieser Form nicht. Stattdessen wird eine Kombination aus horizontaler/vertikaler Ausrichtung und Abständen (Margins) verwendet.
SplitPanel	Mithilfe des `Grids` und des `GridSplitters` kann eine Unterteilung vorgenommen werden.

Es gibt aber ein paar Leitlinien, die den Umgang und auch den Umstieg erheblich erleichtern. Werden diese befolgt, wird das Arbeiten mit der WPF deutlich einfacher, da man nicht länger gegen das System kämpft. Der nachträgliche Aufwand durch Änderungen oder Anpassungen kann so zumindest auf ein Minimum reduziert werden.

Abhängig von der jeweiligen Situation gibt es natürlich auch hier wieder die berühmte Ausnahme von der Regel. Man sollte daher selber von Fall zu Fall entscheiden, welche der Regeln angewendet werden können und welche nicht. Vor allem Grafiken oder Zeichnungen (die in Kapitel 10 beschrieben werden) sind hiervon fast grundsätzlich ausgenommen.

- Panels sind immer nur für die eigenen Steuerelemente verantwortlich.
  Nur die Steuerelemente, die direkt unterhalb des Panels definiert worden sind, werden

für die Ausrichtung berücksichtigt. Andere Steuerelemente, die sich außerhalb dieses Kontextes befinden, bleiben dagegen außen vor und können sich sogar in der Zuständigkeit eines anderen Panels befinden.

- Absolute Positionierungen sollten vermieden werden.
  Bei der absoluten Positionierung werden Steuerelemente fest in einem Koordinatensystem bewegt. Wurden ihre Position und ihre Größe einmal festgelegt, ist eine Veränderung nur mit zusätzlichem Programmieraufwand möglich. Wird vom Anwender die Größe des Fensters verändert, ist eine Korrektur der Ausrichtung nur sehr schwer möglich.

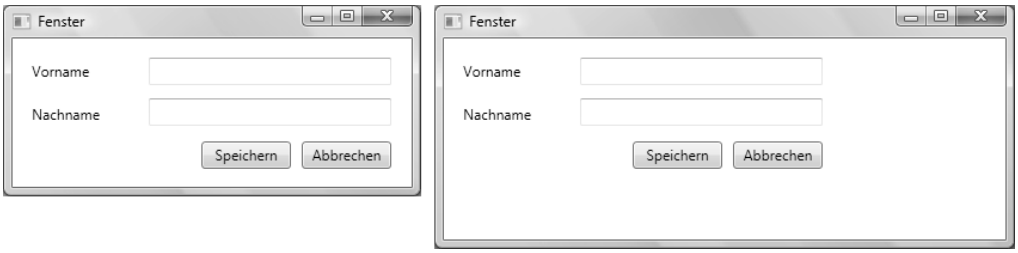

Bild 3.3 Wegen der absoluten Positionierung werden die Steuerelemente nicht angepasst, und ein unnötiger Freiraum entsteht.

Beim Positionieren mit Blend oder Visual Studio versuchen die Designer meist automatisch, die Steuerelemente mit `HorizontalAlignment` und `VerticalAlignment` an den rechten oder unteren Seitenrand „anzuheften“. Das kann schnell zu ungewollten Ergebnissen führen, wenn die falschen Steuerelemente diese Einstellung erhalten.

- Die Größe eines Steuerelementes wird durch seinen Inhalt bestimmt.
  Die Panels, aber auch die dort enthaltenen Steuerelemente, haben in der WPF eines gemeinsam: Sie versuchen, die Steuerelemente immer optimal anzuzeigen. Dabei richten sie sich hauptsächlich nach dem Platzbedarf ihrer Inhalte: Enthält eine Schaltfläche einen längeren Text, wird ihm auch mehr Platz zugesprochen. Das wirkt sich vorteilhaft aus, wenn der Benutzer die Schriftgröße systemweit ändert oder die Anwendung in unterschiedlichen Sprachen ausgeliefert werden soll.

  Auf derartige Veränderungen selber zu reagieren, würde einen enormen Aufwand bedeuten. Die WPF reagiert hingegen automatisch auf diese Veränderungen – jedenfalls solange man nicht mit absoluten Größenangaben arbeitet.

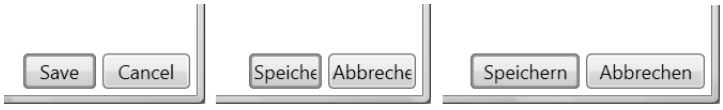

Bild 3.4 Feste Größenangaben (links und Mitte) können sich nicht nach dem Platzbedarf ihrer Inhalte richten (rechts).

In Bild 3.4 sind zwei Schaltflächen zu sehen, die mit festen Größenangaben platziert worden sind. Sobald sich der Inhalt der Schaltflächen ändert, beispielsweise durch den Wechsel der Sprache, können längere Texte nicht mehr adäquat angezeigt werden. Normalerweise können die Schaltflächen ihre Breite anhand ihrer Inhalte automatisch anpassen. Hierfür dürfen aber keine Werte für `Width` und `Height` angegeben werden. Ein entspre-

chender Abstand der Beschriftung vom Rand der Schaltflächen kann mit der Angabe von `Padding` erreicht werden.

- Die Positionierung der Steuerelemente wird alleine vom Panel vorgenommen.
  Für den Benutzer sind Panels grundsätzlich unsichtbar; lediglich ihre Auswirkung auf die Positionierung der Steuerelemente wird für ihn sichtbar. Ein Panel hat in der WPF ausschließlich die Aufgabe, Steuerelemente in akkurater Weise zu positionieren. Damit das auch im Sinne des Entwicklers passiert, werden hierfür gleich mehrere Panels angeboten, die miteinander kombiniert oder deren Verhalten über Parameter angepasst werden können.

3.2 StackPanel

Das `StackPanel` ist sicherlich eine der einfachsten Ausführungen eines Panels. Wie der Name vielleicht vermuten lässt, stapelt das Panel die enthaltenen Elemente einfach in der Reihenfolge auf, in der sie definiert worden sind. In dem folgenden Beispiel werden der Reihe nach die vier Schaltflächen aufgereiht.

Listing 3.3 Die einfache Verwendung eines StackPanels

```
<Window xmlns="..." xmlns:x="...">
  <StackPanel>
    <Button>Eins</Button>
    <Button>Zwei</Button>
    <Button>Drei</Button>
    <Button>Vier</Button>
  </StackPanel>
</Window>
```

Da die Schaltflächen keinerlei Angaben zu ihrer Breite, Höhe oder Ausrichtung enthalten, streckt das `StackPanel` daher alle auf die maximal mögliche Breite. Die Höhe der Schaltflächen wird dagegen nicht beeinflusst.

Die verfügbare Fläche wird vom `StackPanel` relativ stur der Reihe nach befüllt – schließlich ist es ja nur auf diese eine Aufgabe spezialisiert. Der verbleibende Freiraum (hier unten) wird standardmäßig freigehalten. Sollte der Raum dagegen nicht ausreichend sein, so verschwinden einfach die Elemente am Seitenrand.

Bild 3.5 Ein StackPanel stapelt die Schaltflächen der Reihe nach auf.

Auf das Verhalten vom `StackPanel` kann nur sehr wenig Einfluss genommen werden. Lediglich die Ausrichtung der Elemente kann über die Eigenschaft `Orientation` bestimmt werden:

Listing 3.4 Die Ausrichtung auf Horizontal ändern

```
<StackPanel Orientation="Horizontal">
  <Button>Eins</Button>
  <Button>Zwei</Button>
  <Button>Drei</Button>
  <Button>Vier</Button>
</StackPanel>
```

Anders als im ersten Beispiel werden die Schaltflächen in Bild 3.6 nicht mehr vertikal, sondern horizontal gestapelt. Dafür wird jetzt auch die Höhe der Schaltflächen angepasst und nicht mehr ihre Breite. Werden darüber hinaus weitere Features benötigt, wie beispielsweise das Umkehren der Reihenfolge von rechts nach links oder das Ausfüllen der restlichen Freifläche mit einem Steuerelement, wäre der Einsatz von einem `DockPanel` empfehlenswert, da dieses ganz ähnliche Eigenschaften besitzt.

Wie jedes andere Panel kann auch ein `StackPanel` dazu verwendet werden, den Inhalt eines Steuerelements zu steuern. So kann beispielsweise eine Schaltfläche nicht nur einen einfachen Text darstellen, sondern dank eines `StackPanel` auch mit einer Grafik erweitert werden.

Listing 3.5 Das StackPanel für die Darstellung von Steuerelementen verwenden

```
<Button>
  <StackPanel Orientation="Vertical">
    <Image Source="Clipboad.gif" />
    <TextBlock>Clipboard</TextBlock>
  </StackPanel>
</Button>

<Button>
  <StackPanel Orientation="Horizontal">
    <TextBlock Margin="0,0,4,0"
        FontWeight="Bold">Name:</TextBlock>
    <TextBlock>Jörg Wegener</TextBlock>
  </StackPanel>
</Button>
```

Das Ergebnis der Ausrichtung lässt sich durchaus sehen. In Bild 3.6 sind die beiden Schaltflächen abgebildet, die durch ein Symbol und die Formatierungen deutlich besser hervorgehoben werden können.

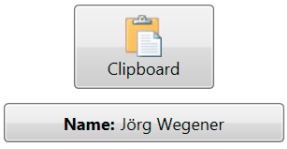

Bild 3.6 Ein StackPanel kann auch die Darstellung einer Schaltfläche erweitern.

3.3 DockPanel

Mit einem `DockPanel` lassen sich beliebige Elemente an die Seiten eines Fensters anheften. Dieses Feature mag an die bekannte Eigenschaft `Dock` der Steuerelemente unter Windows Forms erinnern. In der Tat verhalten sich beide Techniken erstaunlich ähnlich, doch wie immer steckt hier der Teufel wieder im Detail.

Listing 3.6 Einfache Verwendung eines DockPanel

```
<DockPanel>
  <Button>Eins</Button>
  <Button>Zwei</Button>
  <Button>Drei</Button>
  <Button>Vier</Button>
</DockPanel>
```

Wie man in Bild 3.8 sieht, verhält sich das `DockPanel` ohne weitere Angaben ganz ähnlich wie ein `StackPanel`: Die enthaltenen Schaltflächen werden einfach der Reihe nach am linken Seitenrand angeheftet. Außerdem wird ihre Höhe auf das mögliche Maximum gesetzt. Lediglich die vierte Schaltfläche, die in der Auflistung als Letztes angegeben wurde, bildet eine Ausnahme und nimmt die gesamte verbleibende Fläche ein.

Bild 3.7 Ein DockPanel ohne Angaben zur Ausrichtung der Steuerelemente

Sollen die Schaltflächen dagegen wie Menüs oder Symbolleisten an unterschiedliche Seiten des Fensters angeheftet werden, so kann die Ausrichtung individuell für jedes Steuerelement einzeln angepasst werden.

Listing 3.7 Steuerelemente mit einem DockPanel an unterschiedliche Seiten anheften

```
<DockPanel>
  <Button DockPanel.Dock="Top">Oben</Button>
  <Button DockPanel.Dock="Bottom">Unten</Button>
  <Button DockPanel.Dock="Left">Links</Button>
  <Button DockPanel.Dock="Right">Rechts</Button>
  <Button>Inhalt</Button>
</DockPanel>
```

Mithilfe des Attached Properties `Dockpanel.Dock`, das von der Klasse `DockPanel` definiert worden ist, können die Schaltflächen jeweils an die gewünschten Seiten des Fensters angeheftet werden. In diesem Beispiel wurden die vier Schaltflächen einfach an die jeweiligen Seiten des Fensters angeheftet. Da im vorherigen Beispiel diese Angabe noch fehlte, wurden die Schaltflächen einfach der Reihe nach an den linken Rand angeheftet. Das entspricht auch dem Standardverhalten des `DockPanel`.

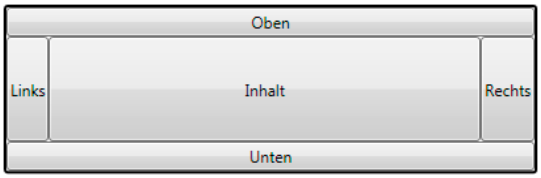

Bild 3.8 Das DockPanel heftet die Steuerelemente an den gewünschten Seitenrand an.

Wie auch im vorherigen Beispiel wird wieder ein Element für die verbleibende Freifläche genutzt. Das `DockPanel` verwendet hierfür ausschließlich das Element, das als letztes in der Auflistung angegeben wurde – in diesem Fall wäre das die Schaltfläche mit der Beschriftung „Inhalt“. Anstelle der Schaltfläche kann hier aber auch ein weiteres Panel platziert werden, wie beispielsweise ein `Grid` oder ein `StackPanel`, das die weiteren Inhalte in einer anderen Form unterteilen kann.

Sollte man aber vielleicht aus alter Gewohnheit versehentlich die Reihenfolge ändern, so können dadurch seltsame Ergebnisse entstehen. Wird das Steuerelement für den Inhalt nicht als letztes definiert, so wird dieses einfach an den linken Seitenrand angeheftet. Dagegen wird mit dem letzten Steuerelement die verbleibende Freifläche gefüllt – selbst wenn bei diesem die Richtung zum Anheften explizit angegeben wurde.

In der Regel ist es wünschenswert, dass die verbleibende Fläche für weitere Steuerelemente genutzt werden kann. In so einem Fall müsste das letzte Steuerelement wiederum ein Panel sein, das die übrigen Steuerelemente aufnehmen kann. Der Freiraum kann beispielsweise von einem `Grid` weiter aufgeteilt werden, wohingegen sich die übrigen Steuerelemente, wie das Menü oder die Symbolleisten, unauffällig an die Seiten des Fensters anheften lassen. Ein `DockPanel` eignet sich aber auch gut, um Steuerelemente zu gruppieren. Als Beispiel sei hier ein Steuerelement aus Outlook genannt: Hier werden die unterschiedlichen Ansichten und Aufgaben in Kategorien zusammengefasst.

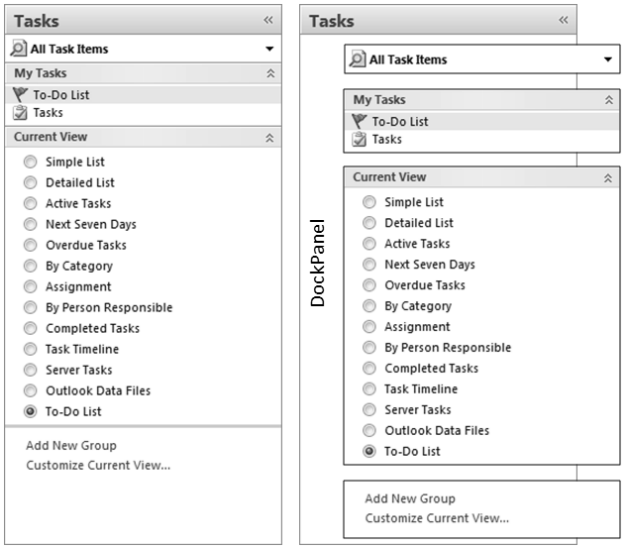

Bild 3.9 Die Anatomie eines DockPanels – die jeweiligen Steuerelemente ließen sich in der WPF einfach über ein DockPanel ausrichten.

In Bild 3.9 ist das Steuerelement abgebildet, das die Kategorien mit den Titeln *My Tasks* und *Current View* enthält. Die jeweiligen Kategorien ließen sich in der WPF beispielsweise problemlos mit dem Steuerelement `Expander` implementieren, mit dem sich der Inhalt ein- oder ausblenden lässt. Das `DockPanel` würde diese dann der Reihe nach am oberen Rand andocken, wohingegen der letzte `Expander` die verbleibende Freifläche einnehmen würde. Da die Höhe der jeweiligen `Expander` durch ihren Inhalt bestimmt wird, nimmt jede Kategorie gerade so viel Platz ein, wie für ihre Darstellung notwendig ist.

Listing 3.8 Beispiel von Expandern, die mit einem DockPanel ausgerichtet werden

```
<DockPanel>
  <Expander DockPanel.Dock="Top" Header="My Tasks" />
  <Expander DockPanel.Dock="Top" Header="Current View" />
</DockPanel>
```

Es mag jedoch auch Fälle geben, bei denen nicht die verbleibende Fläche mit dem letzten Steuerelement ausgefüllt werden soll. Das Verhalten vom `DockPanel` lässt sich aber mit der Eigenschaft `LastChildFill` beeinflussen. Wird hier der Wert auf `False` gesetzt, so verhält sich das `DockPanel` fast wie ein `StackPanel` und lässt die verbleibende Fläche unberührt.

Wie eingangs erwähnt, bietet sich ein `DockPanel` an, um Symbolleisten oder eine Statusbar an die Seiten des Fensters anzuheften. Im folgenden Beispiel werden diese beiden Steuerelemente mit einem `DockPanel` an den oberen und unteren Seitenrand angeheftet.

Listing 3.9 DockPanel zum Anheften eines Menüs und der Symbolleisten

```
<DockPanel>
  <ToolBarTray DockPanel.Dock="Top">
    <ToolBar>
      <Button>Neu</Button>
      <Button>Öffnen</Button>
      <Button>Speichern</Button>
    </ToolBar>
  </ToolBarTray>
  <StatusBar DockPanel.Dock="Bottom">Status: OK</StatusBar>
  <Grid>
    <TextBox>Das ist der Inhalt</TextBox>
  </Grid>
</DockPanel>
```

Hier wird das Steuerelement `ToolBarTray` an den oberen Rand des Fensters angedockt. Ein `ToolBarTray` dient als übergeordnetes Steuerelement für mehrere Symbolleisten, die darin besser organisiert werden können. Es ist aber auch möglich, ohne dieses Steuerelement die Symbolleisten einzeln an den Seitenrand anzuheften. In Bild 3.10 ist das Ergebnis vom `DockPanel` zu sehen.

Bild 3.10 Das DockPanel heftet Symbolleiste und Statuszeile an den Fensterrand an.

In der Standardausführung eignet sich ein `DockPanel` allerdings nur zum „statischen" Andocken an den Seitenrand. Die bekannten Werkzeugfenster unter Visual Studio oder Office, die man nach Belieben am Seitenrand andocken oder auch frei im Raum bewegen kann, werden in der WPF leider weder vom `DockPanel` noch von einem anderen Steuerelement unterstützt. Man sollte annehmen, dass ein solches Feature in der heutigen Zeit bereits zum Standardrepertoire gehören sollte. Hier gibt es von Drittanbietern entsprechende Lösungen, die sich in Optik und im Verhalten an den Werkzeugfenstern in Visual Studio anlehnen, wie beispielsweise die Lösung von Actipro Software.

3.4 WrapPanel

Das `WrapPanel` besitzt ebenfalls deutliche Ähnlichkeiten zum `StackPanel`, da es auch die enthaltenen Steuerelemente der Reihe nach stapelt. Allerdings geht das `WrapPanel` dabei etwas behutsamer mit dem verfügbaren Platz um und setzt beispielsweise nicht die Höhe der Steuerelemente auf ein mögliches Maximum. Sollte zudem der Platz nicht ausreichend sein, so wird einfach eine neue Zeile angefangen. Dieser Umbruch (oder auch „Wrap") geschieht in der Standardeinstellung automatisch am rechten Seitenrand.

Listing 3.10 Einfache Verwendung eines WrapPanels

```
<WrapPanel>
  <Button>Eins</Button>
  <Button>Zwei</Button>
  <Button>Drei</Button>
  <Button>Vier</Button>
</WrapPanel>
```

Wie auf der linken Seite in Bild 3.11 zu sehen ist, werden die vier Buttons der Reihe nach von links nach rechts aufgereiht. Sollte dabei das rechte Zeilenende erreicht werden, wird vom `WrapPanel` eine neue Zeile angefangen. Sowohl der verbleibende Freiraum am rechten Zeilenende als auch der am unteren Seitenende werden dabei aber nicht ausgefüllt. Selbstverständlich kann auch hier die Ausrichtung vom `WrapPanel` wieder mit einer Eigenschaft beeinflusst werden. Hierfür muss der Wert von `Orientation` geändert werden, der standardmäßig auf „Horizontal" gesetzt ist. Auf der rechten Seite in Bild 3.11 ist das Ergebnis der vertikalen Ausrichtung zu sehen.

Bild 3.11 Das WrapPanel mit horizontaler oder vertikaler Ausrichtung

Anders als bei einem `StackPanel` oder bei einem `DockPanel` scheinen aber dieses Mal weder die Breite noch die Höhe der Schaltflächen verändert worden zu sein. Dieser Eindruck täuscht in diesem Beispiel, da die Inhalte der Schaltflächen eine ähnliche Größe besit-

zen. Verändert man aber die Höhe der Schaltflächen durch eine explizite Angabe, kann das Verhalten vom `WrapPanel` besser verdeutlicht werden.

Listing 3.11 WrapPanel mit Steuerelementen mit expliziter Höhe

```
<WrapPanel>
  <Button>Normal</Button>
  <Button Height="30">Höher</Button>
  <Button Height="15">Kleiner</Button>
  <Button>Vier</Button>
</WrapPanel>
```

Obwohl in der ersten Schaltfläche die Höhenangabe fehlt, sieht man in Bild 3.11, dass das Steuerelement trotzdem auf die gleiche Höhe der zweiten Schaltfläche angepasst worden ist. Das `WrapPanel` versucht hier, eine einheitliche Zeilenhöhe zu erreichen, wodurch die Darstellung harmonischer werden soll, als wenn jedes Steuerelement in der Zeile eine unterschiedliche Höhe besitzt. Bitte beachten Sie, dass die Höhe nur dann automatisch angepasst wird, wenn diese nicht explizit gesetzt worden ist. Im Falle der dritten Schaltfläche wurde ebenfalls die Höhe gesetzt, allerdings auf einen wesentlich niedrigeren Wert. In diesem Fall lässt das `WrapPanel` die Höhe unberührt. Das Verhalten wirkt sich ausschließlich auf die aktuelle Zeile aus. Die vierte Schaltfläche, die sich bereits in der nächsten Zeile befindet, ist von dieser Anpassung nicht mehr betroffen und wird in ihrer regulären Höhe dargestellt.

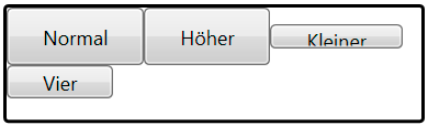

Bild 3.12 Steuerelemente ohne explizite Höhenangabe, wie in diesem Fall die erste Schaltfläche, werden automatisch in ihrer Größe angepasst.

Ein `WrapPanel` kann daher eher mit der Textausgabe unter Word oder unter HTML verglichen werden, wo ebenfalls die Textzeilen auf eine einheitliche Höhe angepasst werden, um ein ruhiges und harmonisches Bild zu erhalten. Das Verhalten lässt sich von außen über die beiden Eigenschaften `ItemWidth` und `ItemHeight` beeinflussen, mit denen entweder die Breite oder die Höhe der Einträge definiert werden können. Das `WrapPanel` orientiert sich so nicht mehr an die Größe der Steuerelemente und gibt die jeweiligen Zeilen (oder Spalten) in einer Einheitsgröße aus.

3.5 Grid

Das `Grid` teilt die Oberfläche wie eine Tabelle in mehrere Zeilen und Spalten ein. Das Verhalten von einem `Grid` kann daher am ehesten mit den Tabellen aus HTML verglichen werden, da diese eine ganz ähnliche Funktion aufweisen.

Listing 3.12 Definition eines einfachen Grids (ohne Einteilung in Zeilen und Spalten)

```
<Grid>
  <Button>Eins</Button>
  <Button>Zwei</Button>
  <Button>Drei</Button>
  <Button>Vier</Button>
</Grid>
```

Ohne zusätzliche Angaben hat das `Grid` aber keinerlei Kenntnis darüber, wie die Schaltflächen ausgerichtet und positioniert werden sollen. Das Ergebnis fällt daher eher ernüchternd aus, wie man auf der linken Seite in Bild 3.13 erkennen kann. Scheinbar wird nur die letzte Schaltfläche angezeigt, die zudem noch auf die gesamte Fläche des `Grid` vergrößert wurde. Tatsächlich werden aber alle vier Schaltflächen angezeigt und auf dieselbe einheitliche Größe gebracht. Da sie jedoch der Reihe nach angezeigt werden, verdecken die Schaltflächen immer die jeweils vorherigen. Der Anschein einer einzelnen Schaltfläche ist daher trügerisch.

Um effektiver mit einem `Grid` arbeiten zu können, sollte dieses zuerst in Spalten und Zeilen eingeteilt werden. Hierfür stehen die beiden Eigenschaften `ColumnDefinitions` und `RowDefinitions` zur Verfügung. Die jeweiligen Steuerelemente können dann über die beiden Attached Properties `Grid.Column` und `Grid.Row` an die gewünschten Positionen in dem `Grid` platziert werden.

Listing 3.13 Definition eines Grids (mit Enteilung in Zeilen und Spalten)

```
<Grid>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>

  <Button Grid.Column="0" Grid.Row="0">Eins</Button>
  <Button Grid.Column="1" Grid.Row="0">Zwei</Button>
  <Button Grid.Column="0" Grid.Row="1">Drei</Button>
  <Button Grid.Column="1" Grid.Row="1">Vier</Button>
</Grid>
```

Um einen spürbaren Effekt zu erzielen, sollte man entweder bei `RowDefinitions` oder bei `ColumnDefinitions` mehr als nur einen Eintrag definieren, da sich das `Grid` ansonsten unverändert verhält.

Bild 3.13 Ein Grid ohne und mit Einteilung in Spalten und Zeilen

Auf der rechten Seite in Bild 3.13 ist das Ergebnis dieser Erweiterung zu sehen. Das `Grid` ist nun in vier gleiche Zellen aufgeteilt. Die Deklaration unterscheidet sich aber deutlich von anderen Techniken, wie beispielsweise der aus HTML. Es gibt keine Tags, die explizit eine neue Zeile oder eine neue Zelle erstellen. Vielmehr werden die Elemente der Reihe nach aufgelistet und ausschließlich über die Attached Properties ihrer Position zugeordnet. Das hat zwar den Vorteil, dass die Definition platzsparender erfolgt und man auch nicht explizit leere Zellen definieren muss, doch trägt es andererseits auch nicht gerade zur Übersichtlichkeit bei. Zumindest ist die Reihenfolge der Steuerelemente im `Grid` unabhängig von ihrer Darstellung, sodass man diese sogar bedenkenlos umkehren könnte, ohne dass dadurch die Darstellung beeinflusst wird.

Das Positionieren der Steuerelemente in einem `Grid` ist in XAML mit den Attached Properties einfach und unkompliziert. Um denselben Vorgang jedoch im Programmcode abzubilden, sind einige Zeilen zusätzlich erforderlich.

Listing 3.14 Ein Grid im Programmcode erstellen

```
// Grid erstellen; in Spalten und Zeilen einteilen
Grid grid = new Grid();
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.ColumnDefinitions.Add(new ColumnDefinition());
grid.RowDefinitions.Add(new RowDefinition());
grid.RowDefinitions.Add(new RowDefinition());

// Inhalte erstellen und im Grid positionieren
Button btn1 = new Button();
btn1.Content = "Eins";
Grid.SetColumn(btn1, 0);
Grid.SetRow(btn1, 0);

// etc...
```

Auch hier muss das `Grid` zuerst in Spalten und Zeilen eingeteilt werden. Diese Angabe erfolgt über die Eigenschaften `ColumnDefinitions` und `RowDefinitions`. Die Schaltflächen können danach über die statischen Methoden `SetColumn()` und `SetRow()` an die gewünschte Position in dem `Grid` gesetzt werden.

Das Erscheinungsbild der Spalten und Zeilen lässt sich aber noch weiter beeinflussen. Neben Höhen- und Breitenangaben lassen sich auch bestimmte Mindest- und Höchstgrößen festlegen, in denen sich die Spalten und Zeilen bewegen dürfen.

Listing 3.15 Grid mit Angaben zu Mindestgrößen

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="60" />
    <ColumnDefinition MinWidth="650" />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition Height="40" />
    <RowDefinition MaxHeight="80" />
  </Grid.RowDefinitions>

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

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

In diesem Beispiel wurden für die Spalten und Zeilen gleich mehrere Eigenschaften gesetzt. So wurde die Breite der ersten Spalte auf exakt 60 Pixel gesetzt. Die zweite Spalte dagegen kann den verbleibenden Platz einnehmen, muss allerdings eine Mindestbreite von 650 Pixeln einnehmen. Bei den Zeilen sehen die Einstellungen ähnlich aus: Die erste Zeile wurde auf eine exakte Höhe von 40 Pixel gesetzt. Die zweite Zeile kann wiederum den verbleibenden Platz in Anspruch nehmen, solange eine Höhe von 80 Pixeln nicht überschritten wird.

Feste Angaben zur Höhe und Breite, bzw. die Mindest- und Maximalwerte, sind jedoch mit Vorsicht einzusetzen, da sie die Ausrichtung und Positionierung des Layouts stark einschränken. Sie sollten daher nur im Ausnahmefall angewendet werden, wenn ein bestimmtes Steuerelement eine gewisse Mindestfläche zur Darstellung benötigt, um korrekt arbeiten zu können.

Panels verwenden normalerweise immer die gesamte zur Verfügung stehende Fläche, um darauf ihre Steuerelemente auszurichten. Beim `Grid` fällt das besonders auf, da die einzelnen Zellen auf die Gesamtgröße des `Grid` verteilt werden. Das wirkt sich selbstverständlich auch auf die Inhalte aus: Die Größe der Zellen wächst oder schrumpft proportional mit. Dieser Effekt kann sich als unschön erweisen, wenn das Grid für Formulare verwendet werden soll, da die Steuerelemente in den jeweiligen Zellen ebenfalls auf die maximale Fläche vergrößert werden. Eingabefelder oder Beschriftungen können dadurch ungewohnt wuchtig erscheinen. Die Darstellung lässt sich aber mit den Eigenschaften `HorizontalAlignment` oder `VerticalAlignment` anpassen. In Bild 3.14 ist der Vergleich zu sehen, wie diese Eigenschaften die Darstellung verbessern können. Im rechten Grid wurde die Eigenschaft `VerticalAlignment` auf den Wert „Top“ gesetzt.

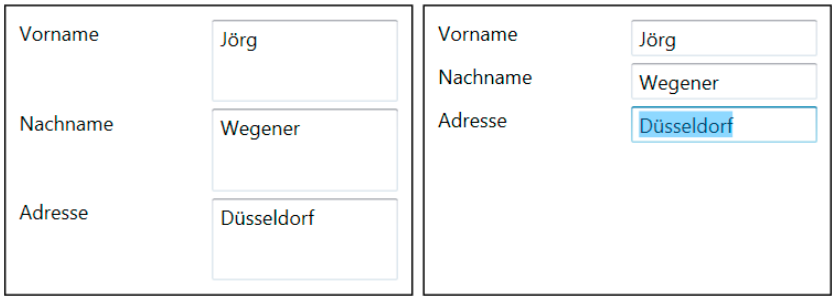

Bild 3.14 Grid mit unterschiedlicher vertikaler Ausrichtung

Diese Möglichkeit ist deutlich flexibler, als wenn man die Höhe und Breite vom `Grid` fest definieren würde. Das Grid kann dadurch auch weiterhin auf den Platzbedarf seiner Steuerelemente eingehen und entsprechend mitwachsen. Zudem müssten diese Werte beim Entwurf der Oberfläche nicht immer wieder korrigiert werden.

Neben den absoluten Höhen- und Breitenangaben lassen sich Spalten und Zeilen aber auch mit relativen Größenangaben steuern. Bei der Umsetzung entschied man sich allerdings gegen eine Verwendung der sonst üblichen Prozentwerte. Stattdessen können freie Werte

angegeben werden, die jeweils in Relation zur Gesamtbreite stehen. Hierbei müssen die angegebenen Werte nicht wie bei Prozentangaben immer zusammen den Wert 100 ergeben – vielmehr wird die Basis automatisch aus der Summe der einzelnen Werte berechnet. Dieser Unterschied macht sich auch schon in der Deklaration bemerkbar. Um sich deutlich von der prozentualen Aufteilung zu unterscheiden, wird anstelle des Prozentzeichens ein Stern (*) verwendet. Eine Kombination aus festen und variablen Spaltenbreiten ist ebenfalls möglich. Hierbei werden aber zuerst die fest vorgegebenen Breitenangaben vom Gesamtwert abgezogen. Erst dieser verbleibende Wert wird über die relativen Größenangaben aufgeteilt.

Listing 3.16 Grid mit relativen Größenangaben

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="2*" />
    <ColumnDefinition Width="3*" />
  </Grid.ColumnDefinitions>
  <Button Grid.Column="0" Content="2*" />
  <Button Grid.Column="1" Content="3*" />
</Grid>
```

In diesem Beispiel wird die Gesamtbreite des Grids durch den Wert fünf dividiert. Dieser Wert errechnet sich aus den Breiten der einzelnen Spalten, die hierfür aufaddiert werden. Die linke Spalte erhält demnach eine Breite von 2/5 und die rechte Spalte eine von 3/5 von der Gesamtbreite.

Diese Definitionen dagegen im Code vorzunehmen, ist ein wenig aufwendiger, da ein einfacher `Double` als Datentyp für die Höhen und Breiten nicht mehr ausreichend ist. Hierfür wurde eigens ein neuer Datentyp namens `GridLength` definiert, der die Größenangabe sowohl als Pixel als auch als Verhältnis (Stern) speichern kann.

Listing 3.17 Größenverhältnisse im Programmcode vornehmen (Stern-Notation)

```
Grid grid = new Grid();
ColumnDefinition column = new ColumnDefinition();
column.Width = new GridLength(3, GridUnitType.Star);
grid.ColumnDefinitions.Add(column);
```

Der Konstruktor der Klasse `GridLength` ist mehrfach überladen. Standardmäßig nimmt dieser einen einfachen `Double`-Wert als Parameter an, über den die Größe in der Maßeinheit Pixel definiert werden kann. In diesem Beispiel wurde jedoch der alternative Konstruktor verwendet, bei dem die Maßeinheit mit einem zweiten Parameter auf die Stern-Notation gesetzt werden kann. Die hier verwendeten Datentypen, wie `GrindLength` oder `GridUnitType`, haben im Namen allesamt „Grid“ als Präfix. Dadurch wird symbolisiert, dass das Grid das einzige Steuerelement ist, das dieses Feature unterstützt.

Sollen darüber hinaus die Inhalte über mehrere Spalten oder über mehrere Zeilen angezeigt werden, können die gewünschten Zellen miteinander verbunden werden. Diese Angabe kann ebenfalls für jedes Steuerelement einzeln getroffen werden.

Listing 3.18 Aufteilung der Steuerelemente über mehrere Spalten und Zeilen

```
<Grid ShowGridLines="True">
  <ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
    <ColumnDefinition />
    <ColumnDefinition />
    <ColumnDefinition />
  </ColumnDefinitions>
  <RowDefinitions>
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
  </RowDefinitions>

  <Button Content="A"
      Grid.Column="0" Grid.ColumnSpan="3"
      Grid.Row="0" Grid.RowSpan="3" />
  <Button Content="B"
      Grid.Column="2" Grid.ColumnSpan="3"
      Grid.Row="2" Grid.RowSpan="3" />
  <Button Content="C"
      Grid.Column="0"
      Grid.Row="2" Grid.RowSpan="3" />
  <Button Content="D"
      Grid.Column="4"
      Grid.Row="0" Grid.RowSpan="3" />
</Grid>
```

Mit den Attached Properties `Grid.ColumnSpan` und `Grid.RowSpan` kann die Positionierung des Steuerelementes über mehrere Spalten und Zeilen definiert werden. Die Schaltfläche mit der Beschriftung „A“ aus diesem Beispiel wird dadurch über drei Spalten und drei Zeilen hinweg im Grid platziert.

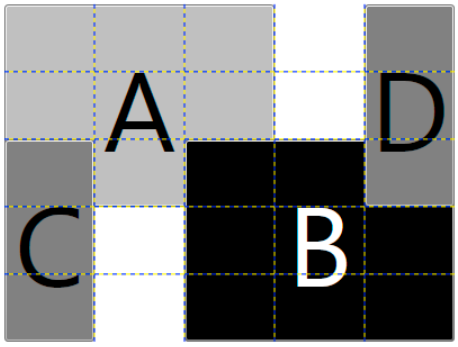

Bild 3.15 Ein Grid mit vier Schaltflächen

Man sollte beachten, dass sich diese Angaben ausschließlich auf die Inhalte (in diesem Beispiel einer Schaltfläche) auswirken und nicht auf das Grid oder auf eine einzelne Zelle selbst. So können problemlos auch weitere Inhalte in derselben Zelle platziert werden, oder

es kann zu Überschneidungen aus anderen Zellen kommen, so wie in Bild 3.15 deutlich zu sehen ist. Ein Steuerelement hat also kein exklusives Recht auf seine Zelle(n). Vielmehr dient das Grid lediglich als Orientierung für die Ausrichtung.

Nützlich während des Entwurfs können auch die Hilfslinien sein, die zur Orientierung im Grid eingeschaltet werden können. Obwohl alle Panels in der WPF grundsätzlich unsichtbar sind, erweist sich gerade die Positionierung der Steuerelemente mit den Spalten- und Zeilendefinitionen als besonders schwierig. Die Hilfslinien heben hierzu die jeweiligen Zellbereiche hervor, sodass die eigenen Definitionen besser kontrolliert werden können. Hierfür muss lediglich die Eigenschaft `ShowGridLines` auf den Wert „True“ gesetzt werden.

In der Regel besitzen Panels keinerlei Möglichkeit zur Interaktion mit dem Anwender. So werden Steuerelemente automatisch positioniert und ausgerichtet, ohne dass der Anwender den Vorgang von außen beeinflussen könnte. Einzig das Grid bietet dem Anwender die Möglichkeit, die Spaltenbreite oder auch die Zeilenhöhe anzupassen. Dieses Feature wird über ein eigenes Steuerelement namens `GridSplitter` implementiert.

Unter Windows Forms konnte man den Splitter frei noch auf der Oberfläche positionieren, da dieser mit den meisten Steuerelementen problemlos umgehen konnte. Der `GridSplitter` hingegen ist ausschließlich auf die Verwendung innerhalb eines `Grid` entwickelt worden und funktioniert dementsprechend weder mit anderen Panels noch mit Steuerelementen. Der `GridSplitter` wird hierfür in einer leeren Zelle platziert, die als Trennung zwischen zwei weiteren Zellen fungiert, die jeweils die Inhalte beherbergen. Der Anwender hat dadurch die Möglichkeit, die Aufteilung der Zellen selbst zu bestimmen.

Listing 3.19 Verwendung eines GridSplitters

```
<Grid>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="1*" />
    <ColumnDefinition Width="6" />
    <ColumnDefinition Width="2*" />
  </Grid.ColumnDefinitions>
  <TextBox
      Grid.Column="0" Grid.Row="0" />
  <GridSplitter
      Grid.Column="1" Grid.Row="0"
      HorizontalAlignment="Stretch" />
  <TextBox Grid.Column="2" Grid.Row="0" />
</Grid>
```

Hier wurde der `GridSplitter` zwischen zwei Spalten platziert. Die Änderung der Spaltengröße übernimmt der `GridSplitter` dabei automatisch – zusätzlicher Programmcode ist hierfür nicht erforderlich. Man sollte lediglich darauf achten, dass der `GridSplitter` die gesamte Fläche der Zelle einnimmt – was nämlich standardmäßig nicht der Fall ist. Das Verhalten muss daher explizit über die Eigenschaft `HorizontalAlignment` angegeben werden.

Eine vertikale Aufteilung ist selbstverständlich auch möglich. Hierfür muss die Eigenschaft `ResizeDirection` auf den Wert `Rows` gesetzt werden. Standardmäßig ist hier der Wert `Columns` hinterlegt.

Listing 3.20 Ein GridSplitter für die Aufteilung in Zeilen

```
<GridSplitter
    Grid.Column="0" Grid.Row="1"
    VerticalAlignment="Stretch"
    ResizeDirection="Columns" />
```

Bei der Verwendung eines `GridSplitter` muss man Folgendes beachten: Grundsätzlich kann bei den beiden Inhaltszellen mit der Stern-Notation gearbeitet werden (wie auch im Beispiel zu sehen war). Sobald der Anwender die Aufteilung mit dem `GridSplitter` anpasst, werden die Werte jedoch wieder in absolute Pixel umgerechnet.

Leider wird in Visual Studio und in Blend das Grid viel zu häufig für eine absolute Positionierung missbraucht. Hierbei werden, ohne dass jegliche Spalten- oder Zeilenaufteilung definiert werden, die Steuerelemente einfach der Reihe nach in die Standardzelle (die immer vorhanden ist) platziert. Diese werden dann mit der Hilfe von `HorizontalAlignment`, `VerticalAlignment` und `Margin` absolut im Fenster positioniert. Für eine absolute Positionierung ist allerdings das `Canvas` wesentlich besser geeignet.

3.6 UniformGrid

Das `UniformGrid` ist die stark vereinfachte Variante vom `Grid`. Hierüber lassen sich lediglich die Anzahl der Spalten und der Zeilen festlegen, die gleichmäßig über die Gesamtgröße verteilt werden. Man hat also keine Möglichkeit, eine gewünschte Zeile oder Spalte größer oder kleiner zu definieren. Auch können Steuerelemente nicht explizit an eine bestimmte Zelle platziert werden, da das `UniformGrid` diese einfach der Reihe nach von der linken oberen zur rechten unteren Zelle platziert.

Listing 3.21 Die Verwendung des UniformGrid

```
<Window>
  <UniformGrid Columns="2" Rows="3">
    <Button Content="links oben" />
    <Button Content="rechts oben" />
    <Button Content="links mitte" />
    <Button Content="rechts mitte" />
    <Button Content="links unten" />
    <Button Content="rechts unten" />
  </UniformGrid>
</Window>
```

Das `UniformGrid` kann daher als eine Art Kombination aus `Grid` und `WrapPanel` angesehen werden, bei dem die Inhalte zwar in einer Tabellenstruktur angeordnet werden, dies allerdings automatisch wie im `WrapPanel` vorgenommen wird.

Sollten weniger Steuerelemente angegeben sein als Zellen definiert wurden, werden die restlichen Zellen des `UniformGrid` einfach freigelassen. Anders sieht es hingegen aus, wenn mehr Steuerelemente als Zellen angegeben wurden: In diesem Fall wird die Fläche

unterhalb des UniformGrids, auf der die Steuerelemente platziert werden, einfach logisch erweitert. Der Inhalt wird also über die Grenzen des UniformGrid hinaus platziert.

3.7 Canvas

Das Canvas bietet als einziges Panel die Möglichkeit zur absoluten Positionierung von Steuerelementen an. Es sollte aber deswegen nicht als letztes Überbleibsel aus den Zeiten von Windows Forms angesehen werden. Auch zukünftig behält das Canvas seine Daseinsberechtigung, da vor allem Grafiken oder Zeichnungen hierüber dargestellt werden. Die Funktionsweise ist daher recht schnell erklärt: Die Steuerelemente werden in einem Koordinatensystem manuell positioniert. Eine automatische Umpositionierung oder Ausrichtung wird dagegen nicht angeboten.

Listing 3.22 Positionierung von Steuerelementen mit einem Canvas

```
<Canvas>
  <Button
      Canvas.Left="10"
      Canvas.Top="10"
      Width="250" Height="25"
      Content="OK" />
</Canvas>
```

Die beiden Attached Properties Canvas.Left und Canvas.Top geben die absolute Position der Schaltfläche innerhalb vom Canvas an. Alternativ können die Angaben auch mit Bezug auf den rechten oder unteren Rand erfolgen. Hierfür wären dann die Attached Properties Canvas.Right und Canvas.Bottom zu verwenden.

Listing 3.23 Steuerelemente am rechten, unteren Seitenrand eines Canvas positionieren

```
<Canvas>
  <Button
      Canvas.Right="10"
      Canvas.Bottom="10"
      Width="250" Height="25"
      Content="OK" />
</Canvas>
```

Eine Kombination beider Angaben ist ebenfalls erlaubt, solange sie nicht widersprüchlich angegeben werden. Eine gleichzeitige Angabe von Canvas.Left und Canvas.Right ist beispielsweise nicht möglich, da ein Steuerelement entweder nur am linken oder rechten Seitenrand ausgerichtet sein kann. Der Compiler wirft hierbei aber keine Fehlermeldung, sondern ignoriert einfach einen der beiden Werte. Weitere Angaben sind im Canvas nicht notwendig, da die Angaben zur Höhe und zur Breite von den Steuerelementen selbst definiert werden können. In den Beispielen wurde die Größe der Schaltfläche nicht mehr automatisch durch den Inhalt vorgegeben, sondern explizit über die Eigenschaften Width und Height gesetzt.

Das `Canvas` ist das kleinste und auch das kompakteste Panel innerhalb der WPF. Da es keinerlei Logik implementieren muss, um bei einer Veränderung der Größe die Ausrichtung der Steuerelemente neu zu berechnen, verbraucht es daher auch weniger Leistung als die übrigen Panels.

3.8 VirtualizingStackPanel

Das `VirtualizingStackPanel` ist eine besondere Ausprägung des `StackPanel`, das man in der Regel aber nicht in der Anwendung selbst, sondern meist nur innerhalb von Templates für Steuerelemente (wie ListBox) benötigt. Es sollte immer dann eingesetzt werden, wenn ein `StackPanel` besonders viele Einträge enthalten kann – also Tausende oder Millionen von Einträgen. In solchen Fällen kann die Leistung der Anwendung rapide nachlassen, da das Layout-System für jeden einzelnen Eintrag die Größe und die Position berechnen müsste – und das, obwohl die meiste Einträge gar nicht sichtbar wären.

Dieser Umstand brachte das `VirtualizingStackPanel` auf den Plan: Es rendert nur die Elemente, die gerade unbedingt für die Darstellung notwendig sind. Das Verhalten kann über die Eigenschaft `IsVirtualizing` gesteuert werden, die standardmäßig auf „True“ gesetzt ist.

3.9 Inhalte ausrichten und positionieren

Um die Steuerelemente und ihre Inhalte in adäquater Weise auszurichten, gibt es gleich eine Handvoll an Eigenschaften, die hierfür verwendet werden können. Dies funktioniert meist unabhängig vom verwendeten Panel, das für die Ausrichtung und die Positionierung der Steuerelemente verantwortlich ist. Welche Abstände oder welche Ausrichtungen die Steuerelemente im Einzelnen haben sollen, wissen die Panels daher in der Regel nicht, und es wird daher von den Steuerelementen selbst vorgegeben.

Die meisten dieser Einstellungen sind aber nur dann sinnvoll, wenn das Panel die Steuerelemente automatisch ausrichtet. Bei einer absoluten Positionierung, wie es beim `Canvas` der Fall ist, sind die meisten Einstellungen dagegen oft wirkungslos.

Mit den beiden Eigenschaften `HorizontalAlignment` und `VerticalAlignment` kann das Steuerelement auf der Gesamtfläche ausgerichtet werden. Da diese beiden Eigenschaften standardmäßig auf dem Wert „Stretch“ stehen, werden die Steuerelemente in der Regel über die gesamte zur Verfügung stehende Fläche ausgerichtet. Die Ausrichtung lässt sich aber beeinflussen: Setzt man beispielsweise für die Eigenschaft `HorizontalAlignment` den Wert „Left“, so wird das Steuerelement am linken Rand ausgerichtet. Die Größe des Steuerelements wird dann entweder von der Eigenschaft `Width` vorgegeben, oder sie wird durch den Platzbedarf seines Inhaltes errechnet.

Tabelle 3.2 Horizontale Ausrichtung

Wert	Erklärung
`Left`	Ausrichtung am linken Rand
`Center`	Zentriert
`Right`	Ausrichtung am rechten Rand
`Stretch`	Ausrichtung über die gesamte Breite

Tabelle 3.3 Vertikale Ausrichtung

Wert	Erklärung
`Top`	Ausrichtung am oberen Rand
`Center`	Zentriert
`Bottom`	Ausrichtung am unteren Rand
`Stretch`	Ausrichtung über die gesamte Höhe

Wie sich diese Einstellungen in Kombination auf das Steuerelement auswirken, zeigt das folgende Beispiel. Hier wurden die Schaltflächen der Reihe nach mit den unterschiedlichen Einstellungen erstellt.

Listing 3.24 Inhalte horizontal und vertikal ausrichten

```
<Grid ShowGridLines="True">
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
  </Grid.RowDefinitions>

  <Button Content="Left / Top"
      Grid.Column="0" Grid.Row="0"
      HorizontalAlignment="Left" VerticalAlignment="Top" />
  <Button Content="Center / Top"
      Grid.Column="1" Grid.Row="0"
      HorizontalAlignment="Center" VerticalAlignment="Top" />
  <Button Content="Right / Top"
      Grid.Column="2" Grid.Row="0"
      HorizontalAlignment="Right" VerticalAlignment="Top" />
  <Button Content="Stretch / Top"
      Grid.Column="3" Grid.Row="0"
      HorizontalAlignment="Stretch" VerticalAlignment="Top" />

  <Button Content="Left / Center"
      Grid.Column="0" Grid.Row="1"
      HorizontalAlignment="Left" VerticalAlignment="Center" />
```

```
    <Button Content="Center / Center"
        Grid.Column="1" Grid.Row="1"
        HorizontalAlignment="Center" VerticalAlignment="Center" />
    <Button Content="Right / Center"
        Grid.Column="2" Grid.Row="1"
        HorizontalAlignment="Right" VerticalAlignment="Center" />
    <Button Content="Stretch / Center"
        Grid.Column="3" Grid.Row="1"
        HorizontalAlignment="Stretch" VerticalAlignment="Center" />

    <Button Content="Left / Bottom"
        Grid.Column="0" Grid.Row="2"
        HorizontalAlignment="Left" VerticalAlignment="Bottom" />
    <Button Content="Center / Bottom"
        Grid.Column="1" Grid.Row="2"
        HorizontalAlignment="Center" VerticalAlignment="Bottom" />
    <Button Content="Right / Bottom"
        Grid.Column="2" Grid.Row="2"
        HorizontalAlignment="Right" VerticalAlignment="Bottom" />
    <Button Content="Stretch / Bottom"
        Grid.Column="3" Grid.Row="2"
        HorizontalAlignment="Stretch" VerticalAlignment="Bottom" />

    <Button Content="Left / Stretch"
        Grid.Column="0" Grid.Row="3"
        HorizontalAlignment="Left" VerticalAlignment="Stretch" />
    <Button Content="Center / Stretch"
        Grid.Column="1" Grid.Row="3"
        HorizontalAlignment="Center" VerticalAlignment="Stretch" />
    <Button Content="Right / Stretch"
        Grid.Column="2" Grid.Row="3"
        HorizontalAlignment="Right" VerticalAlignment="Stretch" />
    <Button Content="Stretch / Stretch"
        Grid.Column="3" Grid.Row="3"
        HorizontalAlignment="Stretch" VerticalAlignment="Stretch" />
</Grid>
```

In Bild 3.16 sind die Schaltflächen zu sehen, die mit den unterschiedlichen Angaben aus `HorizontalAlignment` und `VerticalAlignment` ausgerichtet worden sind.

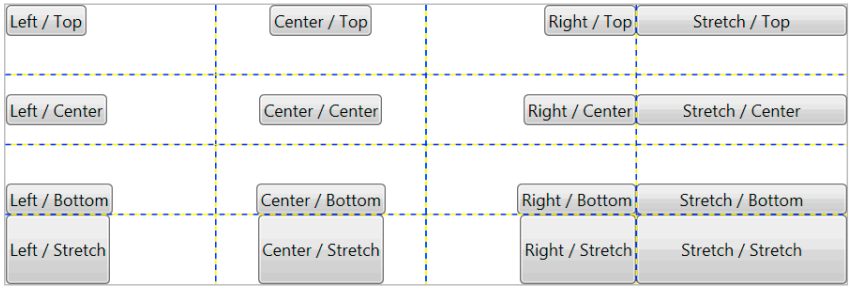

Bild 3.16 Die horizontale und vertikale Ausrichtung im Vergleich

Neben den beiden Eigenschaften `HorizontalAlignment` und `VerticalAlignment` gibt es außerdem noch `Margin` und `Padding`, die für den äußeren und inneren Abstand der Steuerelemente verantwortlich sind. Der Abstand zwischen zwei Steuerelementen kann am

einfachsten mit der Angabe von `Margin` gesteuert werden. Diese Angabe ist besonders hilfreich, um dem verwendeten Panel mitzuteilen, dass zwischen zwei Steuerelementen ein Abstand eingehalten werden soll. Ansonsten würden die Steuerelemente der Reihe nach dicht an dicht gedrängt auf dem Bildschirm ausgegeben werden.

Listing 3.25 Margin und Padding für Abstände verwenden

```
<DockPanel>
  <StackPanel DockPanel.Dock="Bottom"
      Orientation="Horizontal" HorizontalAlignment="Right"
      Margin="6">
    <Button Content="Speichern"
        Margin="0 0 6 0"
        Padding="8 1 8 1" />
    <Button Content="Abbrechen"
        Padding="8 1 8 1" />
  </StackPanel>
  <Grid Background="#DEDEDE" Margin="6" />
</DockPanel>
```

In diesem Beispiel werden zwei Schaltflächen mithilfe eines `StackPanel` am rechten, unteren Ende des Fensters platziert. Damit die Schaltflächen nicht direkt an den Rand des Fensters positioniert werden, wurde bereits im `StackPanel` ein Abstand von sechs Pixeln angegeben. Außerdem wurde bei der ersten Schaltfläche der rechte Abstand zur zweiten Schaltfläche auf sechs Pixel gesetzt, damit diese nicht dicht gedrängt ausgegeben werden.

Neben dem Außenabstand, der über die Eigenschaft `Margin` festgelegt wird, gibt es außerdem noch das `Padding`. Hierüber wird der Abstand vom Inhalt zur Außenkante eines Steuerelementes festgelegt. Diese Angabe ist besonders hilfreich, wenn die Größe des Steuerelementes vom Layout automatisch anhand seines Inhaltes errechnet wird, wie es beispielsweise bei `HorizontalAlignment` oder `VerticalAlignment` der Fall ist.

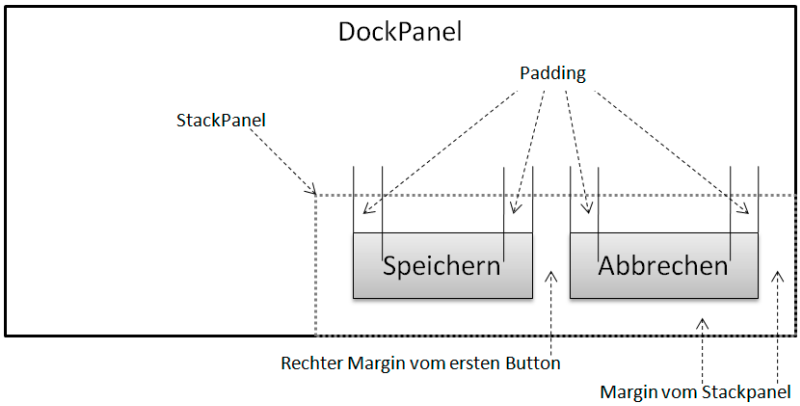

Bild 3.17 Das Zusammenspiel von Margin und Padding bei zwei Schaltflächen

Wird bei `Margin` und bei `Padding` ein einzelner Wert angegeben, so wird der Abstand in alle Richtungen gleichmäßig angewendet. Der Abstand kann aber auch einzeln für jede Richtung angegeben werden. Hierbei müssen lediglich vier einzelne Werte angegeben werden, die jeweils für die Richtungen links, oben, rechts und unten stehen.

Interessant ist nun das Zusammenspiel der verschiedenen Angaben und wie diese sich auf die Größe des Steuerelementes auswirken. Neben `Padding` und `Margin` existiert bei den meisten Steuerelementen zusätzlich ein Außenrahmen, deren Stärke über die Eigenschaft `BorderThickness` gesetzt wird. Sowohl `BorderThickness` als auch `Padding` zählen bei einem Steuerelement zur Gesamtgröße und werden beispielsweise durch die Eigenschaften `Width` und `Height` mitberücksichtigt, wie in Bild 3.18 zu sehen ist. Zum Vergleich: Bei HTML werden diese Werte nicht für die Größe berücksichtigt.

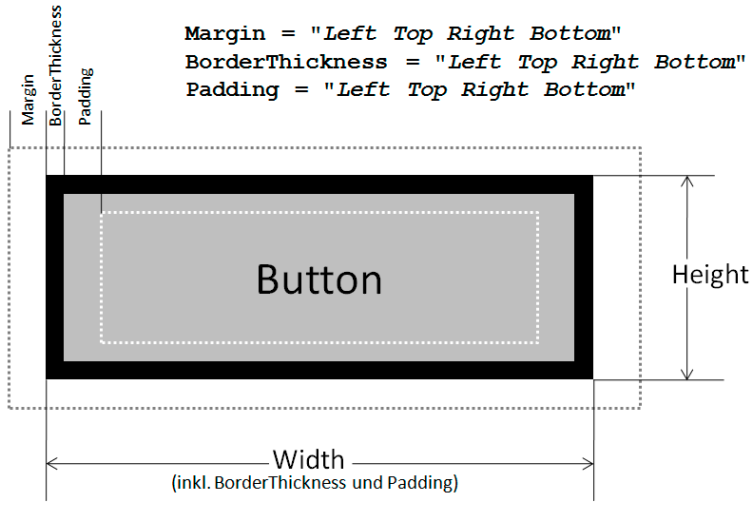

Bild 3.18 Der Einfluss von Padding und BorderThickness auf die Größe eines Steuerelementes

Leider funktioniert das Positionieren und Platzieren in den visuellen Designern weniger elegant. Platziert man beispielsweise eine Schaltfläche in einem Grid, so wird dieser über die Angaben von `HorizontalAlignment`, `VerticalAlignment` und `Margin` auf eine absolute Position gebracht – obwohl das `Grid` ursprünglich nicht für diese Aufgabe entwickelt worden ist. Dabei entsteht außerdem eine Menge an unnötigem Code, der meist nur durch viel Handarbeit einigermaßen optimiert werden kann. Die folgenden Zeilen (die bereits für eine bessere Darstellung optimiert wurden) sind das Ergebnis des Designers.

Listing 3.26 Die visuellen Designer erstellen oft unnötigen Code.

```
<Button
    HoriziontalAlignment="Right"
    VerticalAlignment="Bottom"
    Margin="0,0,26.083,20.677"
    Width="112.056" Height="23">
  Button
</Button>
```

In Bild 3.19 ist das Ergebnis dieses Beispiels zu sehen. Die Pfeile auf der rechten Seite des Bildes geben die Richtung an, mit denen die Ausrichtung erfolgt. Leider werden die Steuerelemente beim Platzieren oft auch mit festen Größenangaben belegt, wie im vorherigen Beispiel mit `Width` und `Height` zu sehen war. Das führt dazu, dass das Layout-System die Steuerelemente nicht mehr optimal nach ihren Inhalten ausrichten kann.

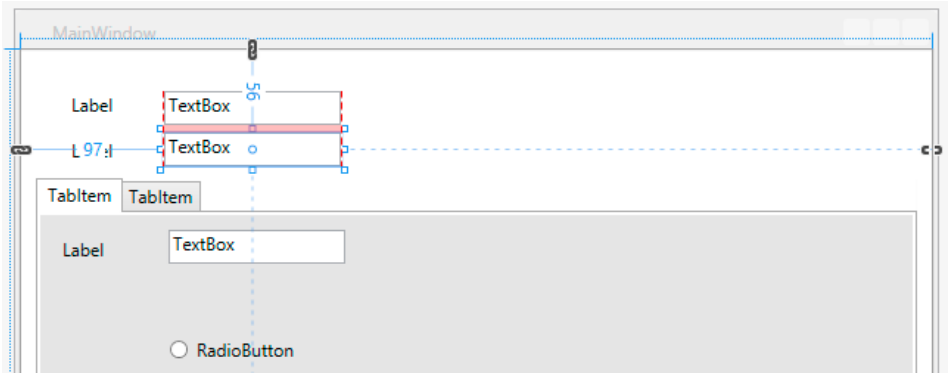

Bild 3.19 Steuerelemente können mit den visuellen Designern schnell und bequem ausgerichtet werden – leider auch mit entsprechenden Nachteilen zulasten des Layouts.

Bei der Arbeit mit visuellen Designern, wie in Blend oder Visual Studio, ist daher Vorsicht geboten. Diese Tücken sind leider oft systembedingt: Ein Steuerelement muss eine bestimmte Anfangsgröße haben, damit es mit der Maus ausgewählt werden kann.

Zur weiteren Verunsicherung eignen sich aber auch gut die unterschiedlichen Angaben, die sich auf die Größe eines Steuerelements beziehen. Gab es unter Windows Forms lediglich die Eigenschaften `Width` und `Height`, um die Größe einer Schaltfläche festzulegen, gibt es in der WPF hierfür scheinbar eine Vielzahl an Eigenschaften.

Jedes Steuerelement enthält auch in der WPF für die Größenangaben die Eigenschaften `Width` und `Height`. Anders als gewohnt, haben die beiden aber eine andere Bedeutung. Da die Größe eines Steuerelements automatisch über den Platzbedarf seines Inhaltes bestimmt wird, ist eine explizite Angabe der Größe nicht mehr notwendig. Um dem Steuerelement anzugeben, dass es sich allein nach seinem Inhalt richten soll, dürfen die Eigenschaften `Width` und `Height` keinen Wert besitzen. Diese sind daher in der Regel auf `NaN` (Not a Number bzw. keine Zahl) gesetzt.

Durch das explizite Setzen einer Höhe oder einer Breite kann das Standardverhalten des Layouts deaktiviert werden. Vergibt man beispielsweise für eine Schaltfläche eine feste Breite, so gibt nicht mehr länger sein Inhalt die Breite vor. Die Eigenschaften `Width` oder `Height` sollten daher ausschließlich zum Setzen und zum manuellen Überschreiben der Breite oder der Höhe verwendet werden. Zum Auslesen der Breite eignen sich beide jedoch nicht.

Bleibt nun die Frage, wie man die Größe eines Steuerelementes auslesen kann, wenn sein Inhalt die Größe vorgibt? Die aktuelle Größe, die vom Layout errechnet worden ist, kann über die beiden Eigenschaften `ActualWidth` und `ActualHeight` ausgelesen werden. Diese geben immer die aktuelle Größe des Steuerelements zurück – unabhängig davon, ob die Größe nun automatisch durch seinen Inhalt errechnet oder manuell über die Eigenschaften `Width` und `Height` gesetzt worden ist. Zum Auslesen der Größe sollten daher ausschließlich diese beiden Eigenschaften verwendet werden; da sie aber schreibgeschützt sind, kann die Größe hier nicht gesetzt werden.

Zur Vollständigkeit kann die Größe mit den Eigenschaften `MinWidth`, `MaxWidth`, `MinHeight` und `MaxHeight` in einem festen Rahmen vorgegeben werden. Das Layout-System orientiert sich dann beim Berechnen der Größe an diesen Werten.

Last, but not least gibt es mit `DesiredSize` und `RenderSize` noch weitere Ausprägungen, in denen Größeninformationen gespeichert sind. Diese werden jedoch nur intern vom Layout-System verwendet, wobei `DesiredSize` die gewünschte Größe und `RenderSize` die tatsächliche Größe des Steuerelements wiedergibt. Beide Angaben sind nach dem Ausrichten und Rendern allerdings mit den aktuellen Werten belegt, die in `ActualWidth` und `ActualHeight` stehen.

3.10 Sichtbarkeit

Die Sichtbarkeit von Steuerelementen wird über die Eigenschaft `Visibility` gesteuert. Diese ist vom gleichnamigen Enum `Visibility` und nimmt dementsprechend nicht einfach nur „True“ oder „False“ an, sondern kann etwas detaillierter auf das Erscheinungsbild Einfluss nehmen.

Tabelle 3.4 Werte für die Sichtbarkeit von Steuerelementen

Wert	Erklärung
`Visible`	Das Steuerelement wird dargestellt.
`Hidden`	Das Steuerelement wird nicht dargestellt, wobei die Fläche auch nicht für andere Steuerelemente freigegeben wird.
`Collapsed`	Das Steuerelement wird nicht dargestellt, die freiwerdende Fläche kann von anderen Steuerelementen genutzt werden.

Dieses Verhalten ähnelt zum Teil der Eigenschaft `Opacity`, die jedoch vom Typ `Double` ist und mit der man die Deckkraft der Steuerelemente wesentlich feiner justieren kann.