Name	Description
Margin	Adds a bit of breathing room around an element. The Margin property is an instance of the System.Windows.Thickness structure, with separate components for the top, bottom, left, and right edges.
MinWidth and MinHeight	Sets the minimum dimensions of an element. If an element is too large for its layout container, it will be cropped to fit.
MaxWidth and MaxHeight	Sets the maximum dimensions of an element. If the container has more room available, the element won't be enlarged beyond these bounds, even if the HorizontalAlignment and VerticalAlignment properties are set to Stretch.
Width and Height	Explicitly sets the size of an element. This setting overrides a Stretch value for the HorizontalAlignment or VerticalAlignment properties. However, this size won't be honored if it's outside of the bounds set by the MinWidth, MinHeight, MaxWidth, and MaxHeight.

All of these properties are inherited from the base FrameworkElement class and are therefore supported by all the graphical widgets you can use in a WPF window.

■ **Note** As you learned in Chapter 2, different layout containers can provide *attached properties* to their children. For example, all the children of a Grid object gain Row and Column properties that allow them to choose the cell where they're placed. Attached properties allow you to set information that's specific to a particular layout container. However, the layout properties in Table 3-3 are generic enough that they apply to many layout panels. Thus, these properties are defined as part of the base FrameworkElement class.

This list of properties is just as notable for what it *doesn't* contain. If you're looking for familiar position properties, such as Top, Right, and Location, you won't find them. That's because most layout containers (all except for the Canvas) use automatic layout and don't give you the ability to explicitly position elements.

Alignment

To understand how these properties work, take another look at the simple StackPanel shown in Figure 3-2. In this example—a StackPanel with vertical orientation—the VerticalAlignment property has no effect because each element is given as much height as it needs and no more. However, the HorizontalAlignment *is* important. It determines where each element is placed in its row.

Ordinarily, the default HorizontalAlignment is Left for a label and Stretch for a Button. That's why every button takes the full column width. However, you can change these details:

```
<StackPanel>
  <Label HorizontalAlignment="Center">A Button Stack</Label>
  <Button HorizontalAlignment="Left">Button 1</Button>
  <Button HorizontalAlignment="Right">Button 2</Button>
  <Button>Button 3</Button>
  <Button>Button 4</Button>
</StackPanel>
```

Figure 3-4 shows the result. The first two buttons are given their minimum sizes and aligned accordingly, while the bottom two buttons are stretched over the entire StackPanel. If you resize the window, you'll see that the label remains in the middle and the first two buttons stay stuck to either side.

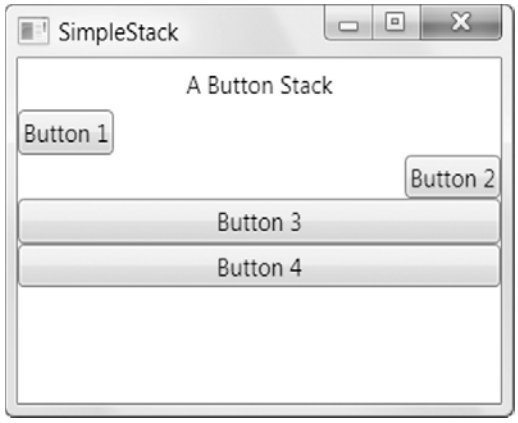

Figure 3-4. *A StackPanel with aligned buttons*

■ **Note** The StackPanel also has its own HorizontalAlignment and VerticalAlignment properties. By default, both of these are set to Stretch, and so the StackPanel fills its container completely. In this example, that means the StackPanel fills the window. If you use different settings, the StackPanel will be made just large enough to fit the widest control.

Margin

There's an obvious problem with the StackPanel example in its current form. A well-designed window doesn't just contain elements—it also includes a bit of extra space in between the elements. To introduce this extra space and make the StackPanel example less cramped, you can set control margins.

When setting margins, you can set a single width for all sides, like this:

```
<Button Margin="5">Button 3</Button>
```

Alternatively, you can set different margins for each side of a control in the order *left, top, right, bottom*:

```
<Button Margin="5,10,5,10">Button 3</Button>
```

In code, margins are set using the Thickness structure:

```
cmd.Margin = new Thickness(5);
```

Getting the right control margins is a bit of an art because you need to consider how the margin settings of adjacent controls influence one another. For example, if you have two buttons stacked on top of each other, and the topmost button has a bottom margin of 5 and the bottommost button has a top margin of 5, you have a total of 10 units of space between the two buttons.

Ideally, you'll be able to keep different margin settings as consistent as possible and avoid setting distinct values for the different margin sides. For instance, in the StackPanel example it makes sense to use the same margins on the buttons and on the panel itself, as shown here:

```
<StackPanel Margin="3">
  <Label Margin="3" HorizontalAlignment="Center">
   A Button Stack</Label>
  <Button Margin="3" HorizontalAlignment="Left">Button 1</Button>
  <Button Margin="3" HorizontalAlignment="Right">Button 2</Button>
  <Button Margin="3">Button 3</Button>
  <Button Margin="3">Button 4</Button>
</StackPanel>
```

This way, the total space between two buttons (the sum of the two button margins) is the same as the total space between the button at the edge of the window (the sum of the button margin and the StackPanel margin). Figure 3-5 shows this more respectable window, and Figure 3-6 shows how the margin settings break down.

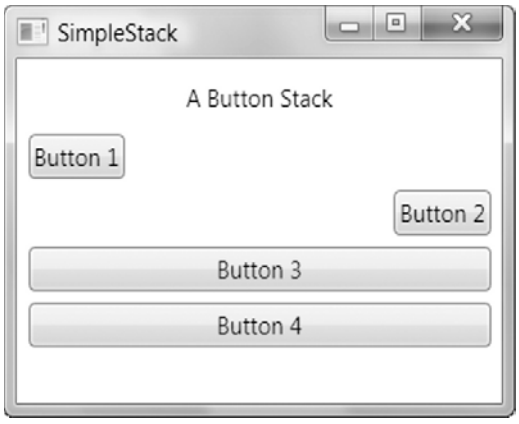

Figure 3-5. *Adding margins between elements*

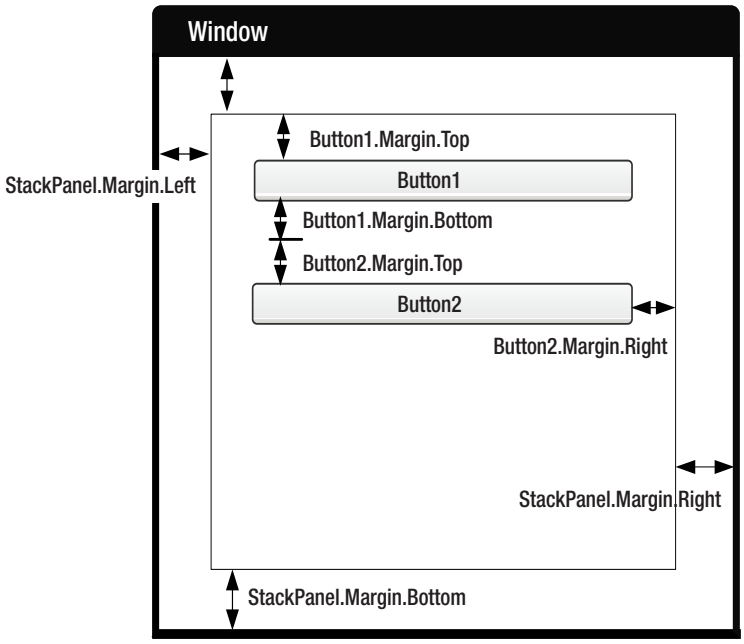

Figure 3-6. *How margins are combined*

Minimum, Maximum, and Explicit Sizes

Finally, every element includes Height and Width properties that allow you to give it an explicit size. However, it's rarely a good idea to take this step. Instead, use the maximum and minimum size properties to lock your control into the right range, if necessary.

■ **Tip** Think twice before setting an explicit size in WPF. In a well-designed layout, it shouldn't be necessary. If you do add size information, you risk creating a more brittle layout that can't adapt to changes (such as different languages and window sizes) and truncates your content.

For example, you might decide that the buttons in your StackPanel should stretch to fit the StackPanel but be made no larger than 200 units wide and no smaller than 100 units wide. (By default, buttons start with a minimum width of 75 units.) Here's the markup you need:

```
<StackPanel Margin="3">
  <Label Margin="3" HorizontalAlignment="Center">
   A Button Stack</Label>
  <Button Margin="3" MaxWidth="200" MinWidth="100">Button 1</Button>
```

```
  <Button Margin="3" MaxWidth="200" MinWidth="100">Button 2</Button>
  <Button Margin="3" MaxWidth="200" MinWidth="100">Button 3</Button>
  <Button Margin="3" MaxWidth="200" MinWidth="100">Button 4</Button>
</StackPanel>
```

■ **Tip** At this point, you might be wondering if there's an easier way to set properties that are standardized across several elements, such as the button margins in this example. The answer is *styles*—a feature that allows you to reuse property settings and even apply them automatically. You'll learn about styles in Chapter 11.

When the StackPanel sizes a button, it considers several pieces of information:

- **The minimum size.** Each button will always be at least as large as the minimum size.
- **The maximum size.** Each button will always be smaller than the maximum size (unless you've incorrectly set the maximum size to be smaller than the minimum size).
- **The content.** If the content inside the button requires a greater width, the StackPanel will attempt to enlarge the button. (You can find out the size that the button wants by examining the DesiredSize property, which returns the minimum width or the content width, whichever is greater.)
- **The size of the container.** If the minimum width is larger than the width of the StackPanel, a portion of the button will be cut off. Otherwise, the button will not be allowed to grow wider than the StackPanel, even if it can't fit all its text on the button surface.
- **The horizontal alignment.** Because the button uses a HorizontalAlignment of Stretch (the default), the StackPanel will attempt to enlarge the button to fill the full width of the StackPanel.

The trick to understanding this process is to realize that the minimum and maximum size set the absolute bounds. Within those bounds, the StackPanel tries to respect the button's desired size (to fit its content) and its alignment settings.

Figure 3-7 sheds some light on how this works with the StackPanel. On the left is the window at its minimum size. The buttons are 100 units each, and the window cannot be resized to be narrower. If you shrink the window from this point, the right side of each button will be clipped off. (You can prevent this possibility by applying the MinWidth property to the window itself, so the window can't go below a minimum width.)

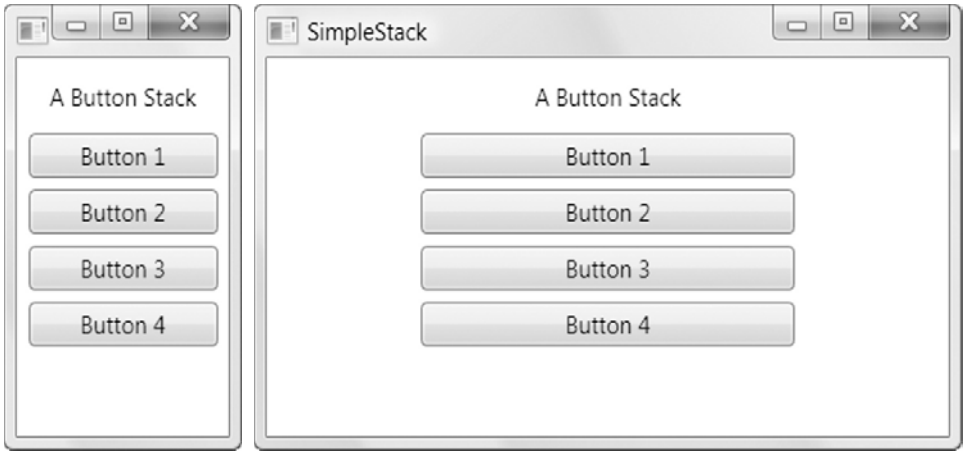

Figure 3-7. *Constrained button sizing*

As you enlarge the window, the buttons grow with it until they reach their maximum of 200 units. From this point on, if you make the window any larger the extra space is added to either side of the button (as shown on the right).

■ **Note** In some situations, you might want to use code that checks how large an element is in a window. The Height and Width properties are no help because they indicate your desired size settings, which might not correspond to the actual rendered size. In an ideal scenario, you'll let your elements size to fit their content, and the Height and Width properties won't be set at all. However, you can find out the actual size used to render an element by reading the ActualHeight and ActualWidth properties. But remember, these values may change when the window is resized or the content inside it changes.

Automatically Sized Windows

In this example, there's still one element that has hard-coded sizes: the top-level window that contains the StackPanel (and everything else inside). For a number of reasons, it still makes sense to hard-code window sizes:

- In many cases, you want to make a window *smaller* than the desired size of its child elements. For example, if your window includes a container of scrollable text, you'll want to constrain the size of that container so that scrolling is possible. You *don't* want to make the window ridiculously large so that no scrolling is necessary, which is what the container will request. (You'll learn more about scrolling in Chapter 6.)

- The minimum window size may be usable, but it might not give you the most attractive proportions. Some window dimensions just look better.
- Automatic window sizing isn't constrained by the display size of your monitor. So, an automatically sized window might be too large to view.

However, automatically sized windows are possible, and they do make sense if you are constructing a simple window with dynamic content. To enable automatic window sizing, remove the Height and Width properties and set the Window.SizeToContent property to WidthAndHeight. The window will make itself just large enough to accommodate all its content. You can also allow a window to resize itself in just one dimension by using a SizeToContent value of Width or Height.

The Border

The Border isn't one of the layout panels, but it's a handy element that you'll often use alongside them. For that reason, it makes sense to introduce it now, before you go any further.

The Border class is pure simplicity. It takes a single piece of nested content (which is often a layout panel) and adds a background or border around it. To master the Border, you need nothing more than the properties listed in Table 3-4.

Table 3-4. *Properties of the Border Class*

Name	Description
Background	Sets a background that appears behind all the content in the border using a Brush object. You can use a solid color or something more exotic.
BorderBrush and BorderThickness	Set the color of the border that appears at the edge of the Border object, using a Brush object, and set the width of the border, respectively. To show a border, you must set both properties.
CornerRadius	Allows you to gracefully round the corners of your border. The greater the CornerRadius, the more dramatic the rounding effect is.
Padding	Adds spacing between the border and the content inside. (By contrast, margin adds spacing outside the border.)

Here's a straightforward, slightly rounded border around a group of buttons in a StackPanel:

```
<Border Margin="5" Padding="5" Background="LightYellow"
 BorderBrush="SteelBlue" BorderThickness="3,5,3,5" CornerRadius="3"
 VerticalAlignment="Top">
  <StackPanel>
    <Button Margin="3">One</Button>
```

```
    <Button Margin="3">Two</Button>
    <Button Margin="3">Three</Button>
  </StackPanel>
</Border>
```

Figure 3-8 shows the result.

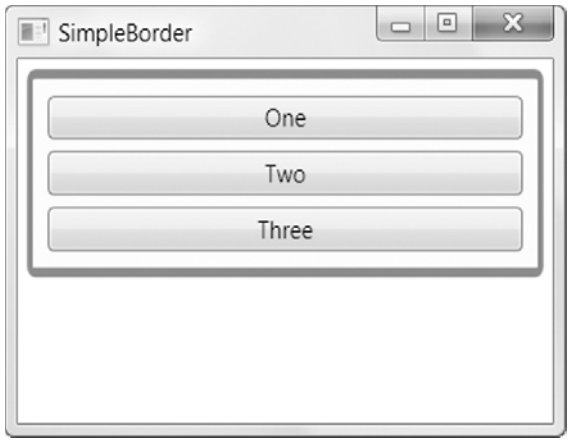

Figure 3-8. *A basic border*

Chapter 6 has more details about brushes and the colors you can use to set BorderBrush and Background.

■ **Note** Technically, the Border is a *decorator*, which is a type of element that's typically used to add some sort of graphical embellishment around an object. All decorators derive from the System.Windows.Controls.Decorator class. Most decorators are designed for use with specific controls. For example, the Button uses a ButtonChrome decorator to get its trademark rounded corner and shaded background, while the ListBox uses the ListBoxChrome decorator. There are also two more general decorators that are useful when composing user interfaces: the Border discussed here and the Viewbox you'll explore in Chapter 12.

The WrapPanel and DockPanel

Obviously, the StackPanel alone can't help you create a realistic user interface. To complete the picture, the StackPanel needs to work with other, more capable layout containers. Only then can you assemble a complete window.

The most sophisticated layout container is the Grid, which you'll consider later in this chapter. But first, it's worth looking at the WrapPanel and DockPanel, which are two more of the simple layout containers provided by WPF. They complement the StackPanel by offering different layout behavior.

The WrapPanel

The WrapPanel lays out controls in the available space, one line or column at a time. By default, the WrapPanel.Orientation property is set to Horizontal; controls are arranged from left to right and then on subsequent rows. However, you can use Vertical to place elements in multiple columns.

■ **Tip** Like the StackPanel, the WrapPanel is really intended for control over small-scale details in a user interface, not complete window layouts. For example, you might use a WrapPanel to keep together the buttons in a toolbar-like control.

Here's an example that defines a series of buttons with different alignments and places them into the WrapPanel:

```
<WrapPanel Margin="3">
  <Button VerticalAlignment="Top">Top Button</Button>
  <Button MinHeight="60">Tall Button 2</Button>
  <Button VerticalAlignment="Bottom">Bottom Button</Button>
  <Button>Stretch Button</Button>
  <Button VerticalAlignment="Center">Centered Button</Button>
</WrapPanel>
```

Figure 3-9 shows how the buttons are wrapped to fit the current size of the WrapPanel (which is determined by the size of the window that contains it). As this example demonstrates, a WrapPanel in horizontal mode creates a series of imaginary rows, each of which is given the height of the tallest contained element. Other controls may be stretched to fit or aligned according to the VerticalAlignment property. In the example on the left of Figure 3-9, all the buttons fit into one tall row and are stretched or aligned to fit. In the example on the right, several buttons have been bumped to the second row. Because the second row does not include an unusually tall button, the row height is kept at the minimum button height. As a result, it doesn't matter what VerticalAlignment setting the various buttons in this row use.

Figure 3-9. *Wrapped buttons*

■ **Note** The WrapPanel is the only panel that can't be duplicated with a crafty use of the Grid.

The DockPanel

The DockPanel is a more interesting layout option. It stretches controls against one of its outside edges. The easiest way to visualize this is to think of the toolbars that sit at the top of many Windows applications. These toolbars are docked to the top of the window. As with the StackPanel, docked elements get to choose one aspect of their layout. For example, if you dock a button to the top of a DockPanel, it's stretched across the entire width of the DockPanel but given whatever height it requires (based on the content and the MinHeight property). On the other hand, if you dock a button to the left side of a container, its height is stretched to fit the container, but its width is free to grow as needed.

The obvious question is, How do child elements choose the side where they want to dock? The answer is through an attached property named Dock, which can be set to Left, Right, Top, or Bottom. Every element that's placed inside a DockPanel automatically acquires this property.

Here's an example that puts one button on every side of a DockPanel:

```
<DockPanel LastChildFill="True">
  <Button DockPanel.Dock="Top">Top Button</Button>
  <Button DockPanel.Dock="Bottom">Bottom Button</Button>
  <Button DockPanel.Dock="Left">Left Button</Button>
  <Button DockPanel.Dock="Right">Right Button</Button>
  <Button>Remaining Space</Button>
</DockPanel>
```

This example also sets the LastChildFill to true, which tells the DockPanel to give the remaining space to the last element. Figure 3-10 shows the result.

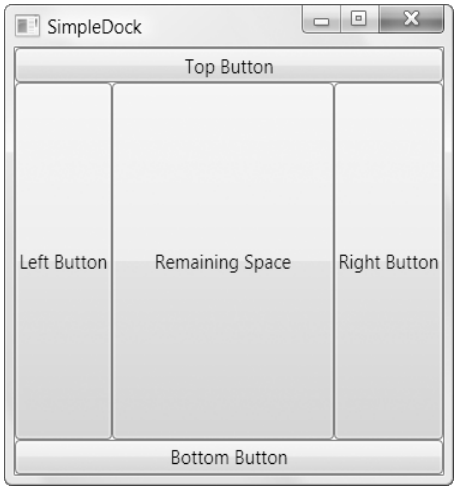

Figure 3-10. *Docking to every side*

Clearly, when docking controls, the order is important. In this example, the top and bottom buttons get the full edge of the DockPanel because they're docked first. When the left and right buttons are docked next, they fit between these two buttons. If you reversed this order, the left and right buttons would get the full sides, and the top and bottom buttons would become narrower because they'd be docked between the two side buttons.

You can dock several elements against the same side. In this case, the elements simply stack up against the side in the order they're declared in your markup. And, if you don't like the spacing or the stretch behavior, you can tweak the Margin, HorizontalAlignment, and VerticalAlignment properties, just as you did with the StackPanel. Here's a modified version of the previous example that demonstrates:

```
<DockPanel LastChildFill="True">
  <Button DockPanel.Dock="Top">A Stretched Top Button</Button>
  <Button DockPanel.Dock="Top" HorizontalAlignment="Center">
   A Centered Top Button</Button>
  <Button DockPanel.Dock="Top" HorizontalAlignment="Left">
   A Left-Aligned Top Button</Button>
  <Button DockPanel.Dock="Bottom">Bottom Button</Button>
  <Button DockPanel.Dock="Left">Left Button</Button>
  <Button DockPanel.Dock="Right">Right Button</Button>
  <Button>Remaining Space</Button>
</DockPanel>
```

The docking behavior is still the same. First the top buttons are docked, then the bottom button is docked, and finally the remaining space is divided between the side buttons and a final button in the middle. Figure 3-11 shows the resulting window.

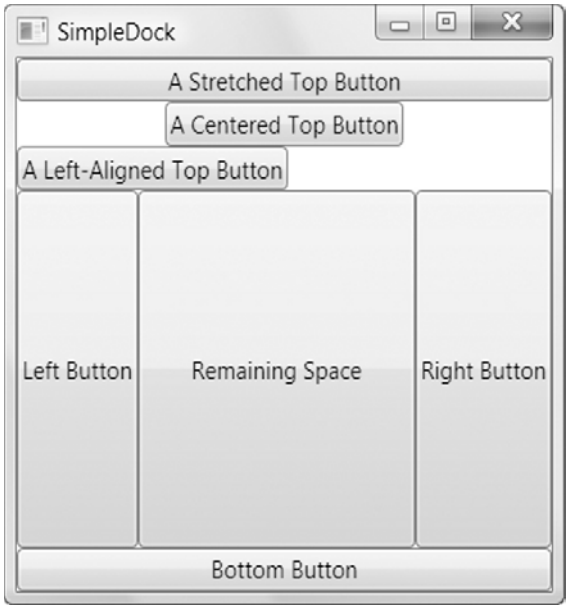

Figure 3-11. *Docking multiple elements to the top*

Nesting Layout Containers

The StackPanel, WrapPanel, and DockPanel are rarely used on their own. Instead, they're used to shape portions of your interface. For example, you could use a DockPanel to place different StackPanel and WrapPanel containers in the appropriate regions of a window.

For example, imagine you want to create a standard dialog box with an OK and Cancel button in the bottom-right corner and a large content region in the rest of the window. You can model this interface with WPF in several ways, but the easiest option that uses the panels you've seen so far is as follows:

1. Create a horizontal StackPanel to wrap the OK and Cancel buttons together.
2. Place the StackPanel in a DockPanel and use that to dock it to the bottom of the window.
3. Set DockPanel.LastChildFill to true so you can use the rest of the window to fill in other content. You can add another layout control here or just an ordinary TextBox control (as in this example).
4. Set the margin properties to give the right amount of whitespace.

Here's the final markup:

```
<DockPanel LastChildFill="True">
  <StackPanel DockPanel.Dock="Bottom" HorizontalAlignment="Right"
   Orientation="Horizontal">
    <Button Margin="10,10,2,10" Padding="3">OK</Button>
    <Button Margin="2,10,10,10" Padding="3">Cancel</Button>
  </StackPanel>
  <TextBox DockPanel.Dock="Top" Margin="10">This is a test.</TextBox>
</DockPanel>
```

In this example, the Padding adds some minimum space between the button border and the content inside (the word "OK" or "Cancel"). Figure 3-12 shows the rather pedestrian dialog box this creates.

Figure 3-12. *A basic dialog box*

At first glance, this seems like a fair bit more work than placing controls in precise positions using coordinates in a traditional Windows Forms application. And in many cases, it is. However, the longer setup time is compensated by the ease with which you can change the user interface in the future. For example, if you decide you want the OK and Cancel buttons to be centered at the bottom of the window, you simply need to change the alignment of the StackPanel that contains them:

```
<StackPanel DockPanel.Dock="Bottom" HorizontalAlignment="Center" ... >
```

This design—a simple window with centered buttons—already demonstrates an end result that wasn't possible with Windows Forms in .NET 1.*x* (at least not without writing code) and required the specialized layout containers with Windows Forms in .NET 2.0. And if you've ever looked at the designer code generated by the Windows Forms serialization process, you'll realize that the markup used here is cleaner, simpler, and more compact. If you add a dash of styles to this window (Chapter 11), you can improve it even further and remove other extraneous details (such as the margin settings) to create a truly adaptable user interface.

■ **Tip** If you have a densely nested tree of elements, it's easy to lose sight of the overall structure. Visual Studio provides a handy feature that shows you a tree representation of your elements and allows you to click your way down to the element you want to look at (or modify). This feature is the Document Outline window, and you can show it by choosing View ➤ Other Windows ➤ Document Outline from the menu.

The Grid

The Grid is the most powerful layout container in WPF. Much of what you can accomplish with the other layout controls is also possible with the Grid. The Grid is also an ideal tool for carving your window into smaller regions that you can manage with other panels. In fact, the Grid is so useful that when you add a new XAML document for a window in Visual Studio, it automatically adds the Grid tags as the first-level container, nested inside the root Window element.

The Grid separates elements into an invisible grid of rows and columns. Although more than one element can be placed in a single cell (in which case they overlap), it generally makes sense to place just a single element per cell. Of course, that element may itself be another layout container that organizes its own group of contained controls.

■ **Tip** Although the Grid is designed to be invisible, you can set the Grid.ShowGridLines property to true to take a closer look. This feature isn't really intended for prettying up a window. Instead, it's a debugging convenience that's designed to help you understand how the Grid has subdivided itself into smaller regions. This feature is important because you have the ability to control exactly how the Grid chooses column widths and row heights.

Creating a Grid-based layout is a two-step process. First, you choose the number of columns and rows that you want. Next, you assign the appropriate row and column to each contained element, thereby placing it in just the right spot.

You create grids and rows by filling the Grid.ColumnDefinitions and Grid.RowDefinitions collections with objects. For example, if you decide you need two rows and three columns, you'd add the following tags:

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

  ...
</Grid>
```

As this example shows, it's not necessary to supply any information in a RowDefinition or ColumnDefinition element. If you leave them empty (as shown here), the Grid will share the space evenly between all rows and columns. In this example, each cell will be exactly the same size, depending on the size of the containing window.

To place individual elements into a cell, you use the attached Row and Column properties. Both these properties take 0-based index numbers. For example, here's how you could create a partially filled grid of buttons:

```
<Grid ShowGridLines="True">
  ...

  <Button Grid.Row="0" Grid.Column="0">Top Left</Button>
  <Button Grid.Row="0" Grid.Column="1">Middle Left</Button>
  <Button Grid.Row="1" Grid.Column="2">Bottom Right</Button>
  <Button Grid.Row="1" Grid.Column="1">Bottom Middle</Button>
</Grid>
```

Each element must be placed into its cell explicitly. This allows you to place more than one element into a cell (which rarely makes sense) or leave certain cells blank (which is often useful). It also means you can declare your elements out of order, as with the final two buttons in this example. However, it makes for clearer markup if you define your controls row by row and from right to left in each row.

There is one exception. If you don't specify the Grid.Row property, the Grid assumes that it's 0. The same behavior applies to the Grid.Column property. Thus, you leave both attributes off of an element to place it in the first cell of the Grid.

■ **Note** The Grid fits elements into predefined rows and columns. This is different from layout containers such as the WrapPanel and StackPanel that create implicit rows or columns as they lay out their children. If you want to create a grid that has more than one row and one column, you must define your rows and columns explicitly using RowDefinition and ColumnDefinition objects.

Figure 3-13 shows how this simple grid appears at two different sizes. Notice that the ShowGridLines property is set to true so that you can see the separation between each column and row.

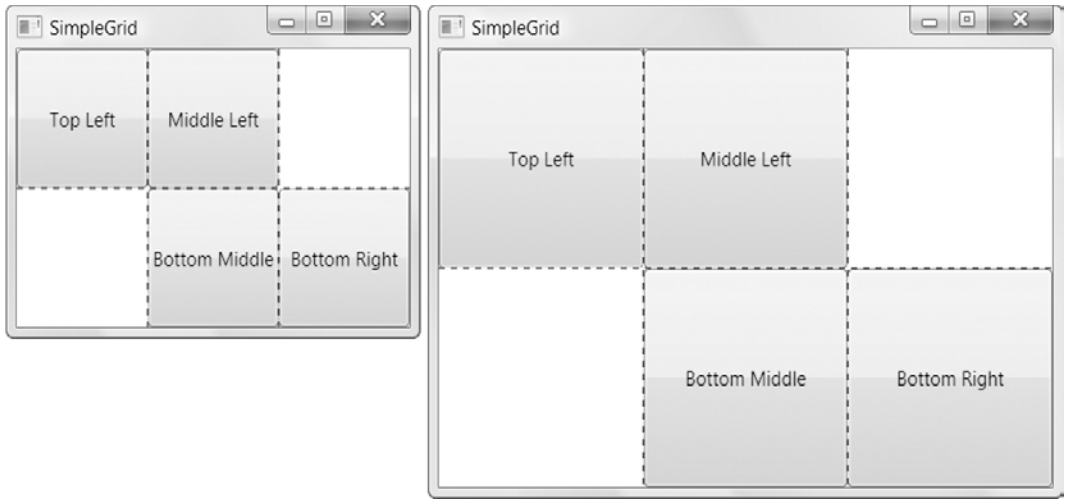

Figure 3-13. *A simple grid*

As you would expect, the Grid honors the basic set of layout properties listed in Table 3-3. That means you can add margins around the content in a cell, you can change the sizing mode so an element doesn't grow to fill the entire cell, and you can align an item along one of the edges of a cell. If you force an element to have a size that's larger than the cell can accommodate, part of the content will be chopped off.

Using the Grid in Visual Studio

When you use a Grid on the Visual Studio design surface, you'll find that it works a bit differently than other layout containers. As you drag an element into a Grid, Visual Studio allows you to place it in a precise position. Visual Studio works this magic by setting the Margin property of your element.

When setting margins, Visual Studio uses the closest corner. For example, if your element is nearest to the top-left corner of the Grid, Visual Studio pads the top and left margins to position the element (and leaves

the right and bottom margins at 0). If you drag your element down closer to the bottom-left corner, Visual Studio sets the bottom and left margins instead and sets the VerticalAlignment property to Bottom. This obviously affects how the element will move when the Grid is resized.

Visual Studio's margin-setting process seems straightforward enough, but most of the time it won't create the results you want. Usually, you'll want a more flexible flow layout that allows some elements to expand dynamically and push others out of the way. In this scenario, you'll find that hard-coding position with the Margin property is extremely inflexible. The problems get worse when you add multiple elements, because Visual Studio won't automatically add new cells. As a result, all the elements will be placed in the same cell. Different elements may be aligned to different corners of the Grid, which will cause them to move with respect to one another (and even overlap each other) as the window is resized.

Once you understand how the Grid works, you can correct these problems. The first trick is to configure your Grid before you begin adding elements by defining its rows and columns. (You can edit the RowDefinitions and ColumnDefinitions collections using the Properties window.) Once you've set up the Grid, you can drag and drop the elements you want into the Grid and configure their margin and alignment settings in the Properties window or by editing the XAML by hand.

Fine-Tuning Rows and Columns

If the Grid were simply a proportionately sized collection of rows and columns, it wouldn't be much help. Fortunately, it's not. To unlock the full potential of the Grid, you can change the way each row and column is sized.

The Grid supports three sizing strategies:

- **Absolute sizes.** You choose the exact size using device-independent units. This is the least useful strategy because it's not flexible enough to deal with changing content size, changing container size, or localization.
- **Automatic sizes.** Each row or column is given exactly the amount of space it needs, and no more. This is one of the most useful sizing modes.
- **Proportional sizes.** Space is divided between a group of rows or columns. This is the standard setting for all rows and columns. For example, in Figure 3-13 you'll see that all cells increase in size proportionately as the Grid expands.

For maximum flexibility, you can mix and match these different sizing modes. For example, it's often useful to create several automatically sized rows and then let one or two remaining rows get the leftover space through proportional sizing.

You set the sizing mode using the Width property of the ColumnDefinition object or the Height property of the RowDefinition object to a number. For example, here's how you set an absolute width of 100 device-independent units:

```
<ColumnDefinition Width="100"></ColumnDefinition>
```

To use automatic sizing, you use a value of Auto:

```
<ColumnDefinition Width="Auto"></ColumnDefinition>
```

Finally, to use proportional sizing, you use an asterisk (*):

```
<ColumnDefinition Width="*"></ColumnDefinition>
```

This syntax stems from the world of the Web, where it's used with HTML frames pages. If you use a mix of proportional sizing and other sizing modes, the proportionally sized rows or columns get whatever space is left over.

If you want to divide the remaining space unequally, you can assign a *weight*, which you must place before the asterisk. For example, if you have two proportionately sized rows and you want the first to be half as high as the second, you could share the remaining space like this:

```
<RowDefinition Height="*"></RowDefinition>
<RowDefinition Height="2*"></RowDefinition>
```

This tells the Grid that the height of the second row should be twice the height of the first row. You can use whatever numbers you like to portion out the extra space.

■ **Note** It's easy to interact with ColumnDefinition and RowDefinition objects programmatically. You simply need to know that the Width and Height properties are GridLength objects. To create a GridLength that represents a specific size, just pass the appropriate value to the GridLength constructor. To create a GridLength that represents a proportional (*) size, pass the number to the GridLength constructor, and pass GridUnitType.Star as the second constructor argument. To indicate automatic sizing, use the static property GridLength.Auto.

Using these size modes, you can duplicate the simple dialog box example shown in Figure 3-12 using a top-level Grid container to split the window into two rows, rather than a DockPanel. Here's the markup you'd need:

```
<Grid ShowGridLines="True">
  <Grid.RowDefinitions>
    <RowDefinition Height="*"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
  </Grid.RowDefinitions>
  <TextBox Margin="10" Grid.Row="0">This is a test.</TextBox>
  <StackPanel Grid.Row="1" HorizontalAlignment="Right" Orientation="Horizontal">
    <Button Margin="10,10,2,10" Padding="3">OK</Button>
    <Button Margin="2,10,10,10" Padding="3">Cancel</Button>
  </StackPanel>
</Grid>
```

■ **Tip** This Grid doesn't declare any columns. This is a shortcut you can take if your Grid uses just one column and that column is proportionately sized (so it fills the entire width of the Grid).

This markup is slightly longer, but it has the advantage of declaring the controls in the order they appear, which makes it easier to understand. In this case, the approach you take is simply a matter of preference. And if you want, you could replace the nested StackPanel with a one-row, two-column Grid.

■ **Note** You can create almost any interface using nested Grid containers. (One exception is wrapped rows or columns that use the WrapPanel.) However, when you're dealing with small sections of user interface or laying out a small number of elements, it's often simpler to use the more specialized StackPanel and DockPanel containers.

Layout Rounding

As you learned in Chapter 1, WPF uses a resolution-independent system of measurement. Although this gives it the flexibility to work on a variety of different hardware, it also sometimes introduces a few quirks. One of these is that elements can be aligned on subpixel boundaries—in other words, positioned with fractional coordinates that don't exactly line up with physical pixels. You can force this to happen by giving adjacent layout containers nonintegral sizes. But this quirk also crops up in some situations when you might not expect it, such as when creating a proportionately sized Grid.

For example, imagine a two-column Grid has 200 pixels to work with. If you've split it evenly into two proportional columns, that means each gets 100 pixels. But if you have 175 pixels, the division isn't as clean, and each column gets 87.5 pixels. That means the second column is slightly displaced from the ordinary pixel boundaries. Ordinarily, this isn't a problem, but if that column contains one of the shape elements, a border, or an image, that content may appear blurry because WPF uses anti-aliasing to "blend" what would otherwise be sharp edges over pixel boundaries. Figure 3-14 shows the problem in action. It magnifies a portion of a window that contains two Grid containers. The topmost Grid does not use layout rounding, and as a result, the sharp edge of the rectangle inside becomes blurry at certain window sizes.

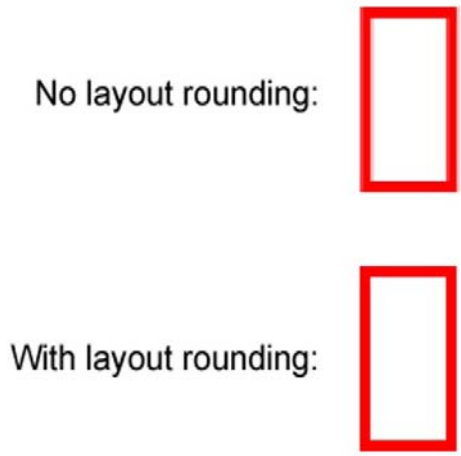

Figure 3-14. *Blur from proportionate sizing*

If this problem affects your layout, there's an easy fix. Just set the UseLayoutRounding property to true on your layout container:

```
<Grid UseLayoutRounding="True">
```

Now WPF will ensure that all the content in that layout container is snapped to the nearest pixel boundary, removing any blurriness.

Spanning Rows and Columns

You've already seen how to place elements in cells using the Row and Column attached properties. You can also use two more attached properties to make an element stretch over several cells: RowSpan and ColumnSpan. These properties take the number of rows or columns that the element should occupy.

For example, this button will take all the space that's available in the first and second cell of the first row:

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

And this button will stretch over four cells in total by spanning two columns and two rows:

```
<Button Grid.Row="0" Grid.Column="0" Grid.RowSpan="2" Grid.ColumnSpan="2">
  Span Button</Button>
```

Row and column spanning can achieve some interesting effects and is particularly handy when you need to fit elements in a tabular structure that's broken up by dividers or longer sections of content.

Using column spanning, you could rewrite the simple dialog box example from Figure 3-12 using just a single Grid. This Grid divides the window into three columns, spreads the text box over all three, and uses the last two columns to align the OK and Cancel buttons.

```
<Grid ShowGridLines="True">
  <Grid.RowDefinitions>
    <RowDefinition Height="*"></RowDefinition>
    <RowDefinition Height="Auto"></RowDefinition>
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="*"></ColumnDefinition>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
    <ColumnDefinition Width="Auto"></ColumnDefinition>
  </Grid.ColumnDefinitions>
  <TextBox Margin="10" Grid.Row="0" Grid.Column="0" Grid.ColumnSpan="3">
    This is a test.</TextBox>
  <Button Margin="10,10,2,10" Padding="3"
    Grid.Row="1" Grid.Column="1">OK</Button>
  <Button Margin="2,10,10,10" Padding="3"
    Grid.Row="1" Grid.Column="2">Cancel</Button>
</Grid>
```

Most developers will agree that this layout isn't clear or sensible. The column widths are determined by the size of the two buttons at the bottom of the window, which makes it difficult to add new content into the existing Grid structure. If you make even a minor addition to this window, you'll probably be forced to create a new set of columns.

As this shows, when you choose the layout containers for a window, you aren't simply interested in getting the correct layout behavior—you also want to build a layout structure that's easy to maintain and enhance in the future. A good rule of thumb is to use smaller layout containers such as the StackPanel for one-off layout tasks, such as arranging a group of buttons. On the other hand, if you need to apply a consistent structure to more than one area of your window (as with the text box column shown later in Figure 3-22), the Grid is an indispensable tool for standardizing your layout.

Split Windows

Every Windows user has seen *splitter bars*—draggable dividers that separate one section of a window from another. For example, when you use Windows Explorer, you're presented with a list of folders (on the left) and a list of files (on the right). You can drag the splitter bar in between to determine what proportion of the window is given to each pane.

In WPF, splitter bars are represented by the GridSplitter class and are a feature of the Grid. By adding a GridSplitter to a Grid, you give the user the ability to resize rows or columns. Figure 3-15 shows a window where a GridSplitter sits between two columns. By dragging the splitter bar, the user can change the relative widths of both columns.

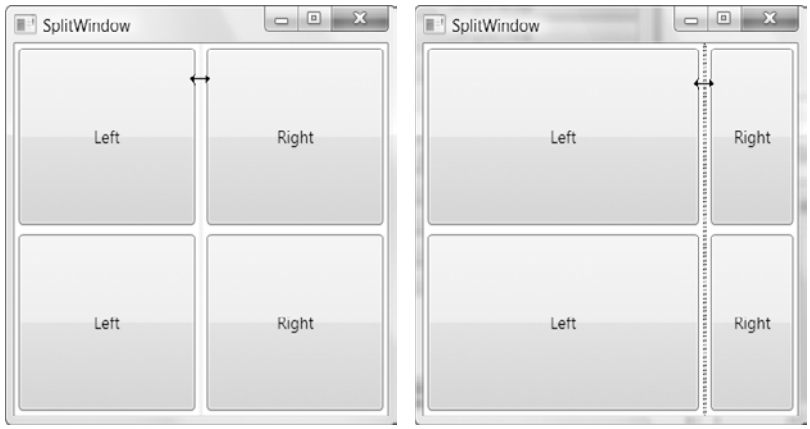

Figure 3-15. *Moving a splitter bar*

Most programmers find that the GridSplitter isn't the most intuitive part of WPF. Understanding how to use it to get the effect you want takes a little experimentation. Here are a few guidelines:

- The GridSplitter must be placed in a Grid cell. You can place the GridSplitter in a cell with existing content, in which case you need to adjust the margin settings so it doesn't overlap. A better approach is to reserve a dedicated column or row for the GridSplitter, with a Height or Width value of Auto.
- The GridSplitter always resizes entire rows or columns (not single cells). To make the appearance of the GridSplitter consistent with this behavior, you should stretch the GridSplitter across an entire row or column, rather than limit it to a single cell. To accomplish this, you use the RowSpan or ColumnSpan properties