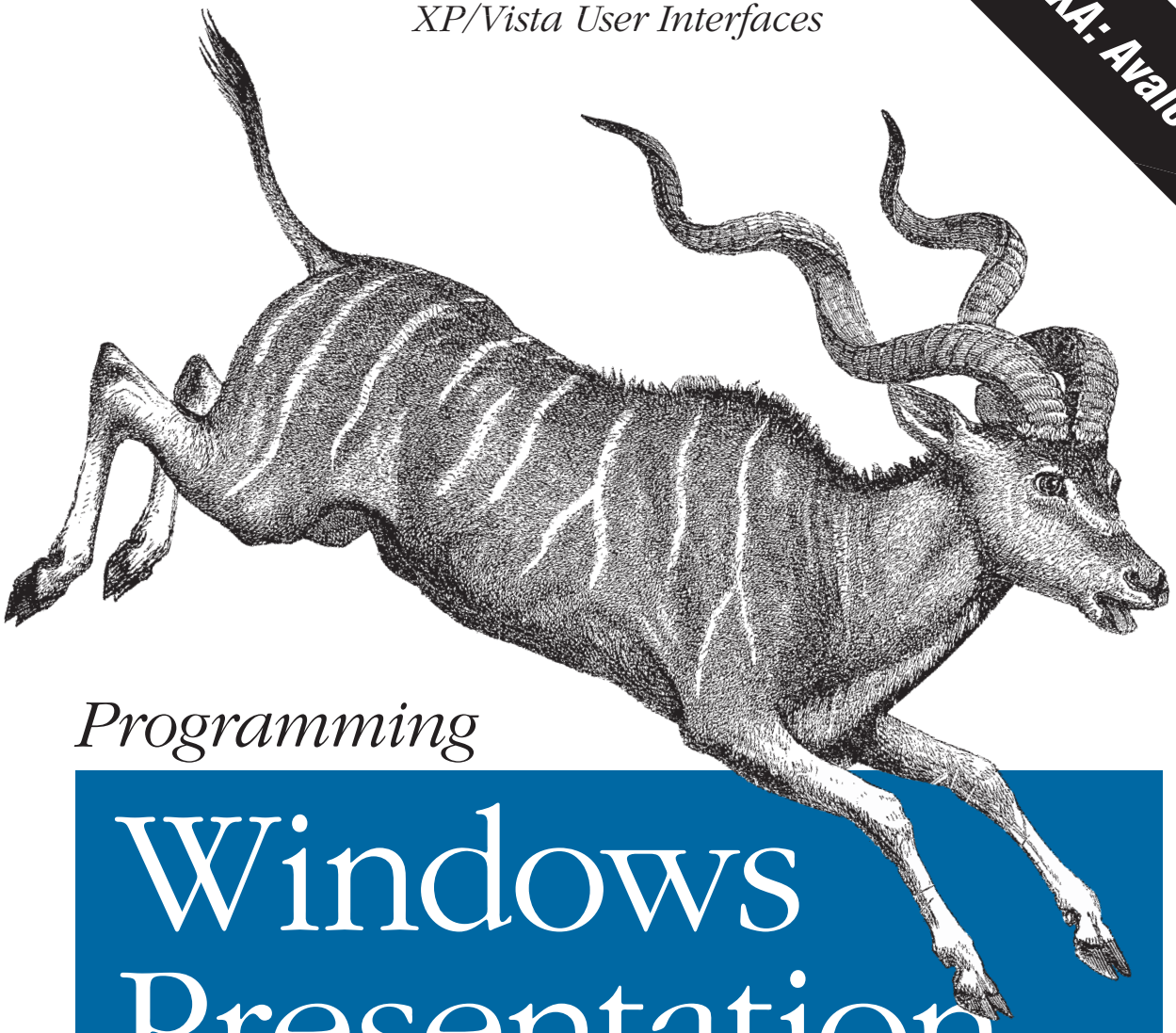
Building Windows XP/Vista User Interfaces

AKA: Avalon



Programming

Windows Presentation Foundation

O'REILLY®

Chris Sells & Ian Griffiths

CHAPTER 5

Styles and Control Templates

In a word processing document, a "style" is a set of properties to be applied to ranges of content—e.g., text, images, etc. For example, the name of the style I'm using now is called "Normal,Body,b" and for this document in pre-publication, that means a font family of Times, a size of 10, and full justification. Later on in the document, I'll be using a style called "Code,x,s" that will use a font family of Courier New, a size of 9, and left justification. Styles are applied to content to produce a certain look when the content is rendered.

In WPF, a *style* is also a set of properties applied to content used for visual rendering. A style can be used to set properties on an existing visual element, such as setting the font weight of a `Button` control, or it can be used to define the way an object looks, such as showing the name and age from a `Person` object. In addition to the features in word processing styles, WPF styles have specific features for building applications, including the ability to associate different visual effects based on user events, provide entirely new looks for existing controls, and even designate rendering behavior for non-visual objects. All of these features come without the need to build a custom control (although that's still a useful thing to be able to do, as discussed in Chapter 9).

Without Styles

As an example of how styles can make themselves useful in WPF, let's take a look at a simple implementation of tic-tac-toe in Example 5-1.

Example 5-1. A simple tic-tac-toe layout

```
<!-- Window1.xaml -->
<Window
    x:Class="TicTacToe.Window1"
    xmlns="http://schemas.microsoft.com/winfx/avalon/2005"
    xmlns:x="http://schemas.microsoft.com/winfx/xaml/2005"
    Text="TicTacToe">
```

Example 5-1. A simple tic-tac-toe layout (continued)

```
  <!-- the black background lets the tic-tac-toe -->
  <!-- crosshatch come through on the margins -->
  <Grid Background="Black">
    <Grid.RowDefinitions>
      <RowDefinition />
      <RowDefinition />
      <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
      <ColumnDefinition />
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Button Margin="0,0,2,2" Grid.Row="0" Grid.Column="0" x:Name="cell00" />
    <Button Margin="2,0,2,2" Grid.Row="0" Grid.Column="1" x:Name="cell01" />
    <Button Margin="2,0,0,2" Grid.Row="0" Grid.Column="2" x:Name="cell02" />
    <Button Margin="0,2,2,2" Grid.Row="1" Grid.Column="0" x:Name="cell10" />
    <Button Margin="2,2,2,2" Grid.Row="1" Grid.Column="1" x:Name="cell11" />
    <Button Margin="2,2,0,2" Grid.Row="1" Grid.Column="2" x:Name="cell12" />
    <Button Margin="0,2,2,0" Grid.Row="2" Grid.Column="0" x:Name="cell20" />
    <Button Margin="2,2,2,0" Grid.Row="2" Grid.Column="1" x:Name="cell21" />
    <Button Margin="2,2,0,0" Grid.Row="2" Grid.Column="2" x:Name="cell22" />
  </Grid>
</Window>
```

This grid layout arranges a set of nine buttons in a 3×3 grid of tic-tac-toe cells, using the margins on the button for the tic-tac-toe crosshatch. A simple implementation of the game logic in the XAML code-behind file looks like Example 5-2.

Example 5-2. A simple tic-tac-toe implementation

```
// Window1.xaml.cs
...
namespace TicTacToe {
  public partial class Window1 : Window {
    // Track the current player (X or O)
    string currentPlayer;

    // Track the list of cells for finding a winner etc.
    Button[] cells;

    public Window1( ) {
      InitializeComponent( );

      // Cache the list of buttons and handle their clicks
      this.cells = new Button[] { this.cell00, this.cell01, ... };
      foreach( Button cell in this.cells ) {
        cell.Click += cell_Click;
      }

      // Initialize a new game
      NewGame( );
```

Example 5-2. A simple tic-tac-toe implementation (continued)

```
    }

    // Wrapper around the current player for future expansion,
    // e.g. updating status text with the current player
    string CurrentPlayer {
      get { return this.currentPlayer; }
      set { this.currentPlayer = value; }
    }

    // Use the buttons to track game state
    void NewGame( ) {
      foreach( Button cell in this.cells ) {
        cell.Content = null;
      }
      CurrentPlayer = "X";
    }

    void cell_Click(object sender, RoutedEventArgs e) {
      Button button = (Button)sender;

      // Don't let multiple clicks change the player for a cell
      if( button.Content != null ) { return; }

      // Set button content
      button.Content = CurrentPlayer;

      // Check for winner or a tie
      if( HasWon(this.currentPlayer) ) {
        MessageBox.Show("Winner!", "Game Over");
        NewGame( );
        return;
      }
      else if( TieGame( ) ) {
        MessageBox.Show("No Winner!", "Game Over");
        NewGame( );
        return;
      }

      // Switch player
      if( CurrentPlayer == "X" ) {
        CurrentPlayer = "O";
      }
      else {
        CurrentPlayer = "X";
      }
    }

    // Use this.cells to find a winner or a tie
    bool HasWon(string player) {...}
    bool TieGame( ) {...}
  }
}
```

Our simple tic-tac-toe logic uses strings to represent the players and uses the buttons themselves to keep track of the game state. As each button is clicked, we set the content to the string indicating the current player and switch players. When the game is over, the content for each button is cleared. The middle of a game looks like Figure 5-1.

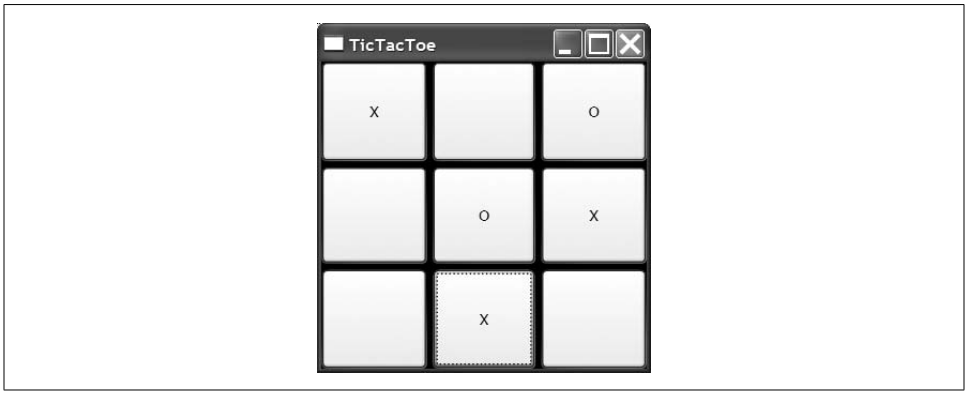


Figure 5-1. A simple tic-tac-toe game

Notice in Figure 5-1 how the grid background comes through from the margin. These spacers almost make the grid look like a drawn tic-tac-toe board (although we'll do better later). However, if we're really looking to simulate a hand-drawn game, we've got to do something about the size of the font used on the buttons; it doesn't match the thickness of the lines.

One way to fix this problem is by setting the size and weight for each of the Button objects, as in Example 5-3.

Example 5-3. Setting control properties individually

```
<Button FontSize="32" FontWeight="Bold" ... x:Name="cell00" />
<Button FontSize="32" FontWeight="Bold"... x:Name="cell01" />
<Button FontSize="32" FontWeight="Bold"... x:Name="cell02" />
<Button FontSize="32" FontWeight="Bold"... x:Name="cell10" />
<Button FontSize="32" FontWeight="Bold"... x:Name="cell11" />
<Button FontSize="32" FontWeight="Bold"... x:Name="cell12" />
<Button FontSize="32" FontWeight="Bold"... x:Name="cell20" />
<Button FontSize="32" FontWeight="Bold"... x:Name="cell21" />
<Button FontSize="32" FontWeight="Bold"... x:Name="cell22" />
```

While this will make the X's and O's look better according to my visual sensibilities today, if I want to change it later, I've now committed myself to changing both properties in nine separate places, which is a duplication of effort that offends my coding sensibilities. I'd much prefer to refactor my decisions about the look of my tic-tac-toe cells into a common place for future maintenance. That's where styles come in handy.

Inline Styles

Each "style-able" element in WPF has a `Style` property, which can be set inline using standard XAML property-element syntax (discussed in Chapter 1), as in Example 5-4.

Example 5-4. Setting an inline style

```
<Button ... x:Name="cell00" />
  <Button.Style>
    <Style>
      <Setter Property="Button.FontSize" Value="32" />
      <Setter Property="Button.FontWeight" Value="Bold" />
    </Style>
  </Button.Style>
</Button>
```

Because we want to bundle two property values into our style, we have a `Style` element with two `Setter` sub-elements, one for each property we want to set—i.e., `FontSize` and `FontWeight`—both with the `Button` prefix to indicate the class that contains the property. Properties suitable for styling are dependency properties, which are described in Chapter 9.

Due to the extra style syntax and because inline styles can't be shared across elements, inline styles actually involve more typing than just setting the properties. For this reason, inline styles aren't used nearly as often as named styles.

Named Styles

By hoisting the same inline style into a resource (as introduced in Chapter 1), we can award it a name and use it by name in our button instances, as shown in Example 5-5.

Example 5-5. Setting a named style

```
<!-- Window1.xaml -->
<Window ...>
  <Window.Resources>
    <Style x:Key="CellTextStyle">
      <Setter Property="Control.FontSize" Value="32" />
      <Setter Property="Control.FontWeight" Value="Bold" />
    </Style>
  </Window.Resources>
  ...
  <Button Style="{StaticResource CellTextStyle}" ... x:Name="cell00" />
  ...
</Window>
```

In Example 5-5, we've used the `Control` prefix on our properties instead of the `Button` prefix to allow the style to be used more broadly, as we'll soon see.

The TargetType Attribute

As a convenience, if all of the properties can be set on a shared class, like `Control` in our example, we can promote the class prefix into the `TargetType` attribute and remove it from the name of the property, as in Example 5-6.

Example 5-6. A target-typed style

```
<Style x:Key="CellTextStyle" TargetType="{x:Type Control}">
  <Setter Property="FontSize" Value="32" />
  <Setter Property="FontWeight" Value="Bold" />
</Style>
```

When providing a `TargetType` attribute, you can only set properties available on that type. If you'd like to expand to a greater set of properties down the inheritance tree, you can do so by using a more derived type, as in Example 5-7.

Example 5-7. A more derived target-typed style

```
<Style x:Key="CellTextStyle" TargetType="{x:Type Button}">
  <!-- IsCancel is a Button-specific property -->
  <Setter Property="IsCancel" Value="False" />
  <Setter Property="FontSize" Value="32" />
  <Setter Property="FontWeight" Value="Bold" />
</Style>
```

In this case, the `IsCancel` property is only available on `Button`, so to set it, we need to switch the `TargetType` attribute for the style.

You may be wondering why I'm setting the `FontSize` to `"32"` instead of `"32pt"` when the latter is more in line with how font sizes are specified and the two representations are definitely not equivalent (the former is pixels, while the latter is points). I'm using pixels because as of this writing, WPF styles using a non-prefixed property allow `"32pt"` to be specified for `FontSize`, while prefixed properties do not. For example, the following works (assuming a `TargetType` is set):

```
<Setter Property="FontSize" Value="32pt" />
```

whereas the following does not (regardless of whether a `TargetType` is set or not):

```
<Setter Property="Control.FontSize" Value="32pt" />
```

Hopefully this problem will have been fixed by the time you read this (and not replaced with others).

Reusing Styles

In addition to saving you from typing out the name of the class prefix for every property name, the `TargetType` attribute will also check that all classes that have the style applied are an instance of that type (or derived type). What that means is that if we

leave TargetType set to Control, we can apply it to a Button element, but not to a TextBlock element, as the former derives ultimately from Control but the latter does not.

On the other hand, while Control and TextBlock both share the common ancestor FrameworkElement, the FrameworkElement class doesn't define a FontSize dependency property, so a style with a TargetType of FrameworkElement won't let us set the FontSize property because its not there, despite the fact that both Control and TextBlock have a FontSize property.

Even with the TargetType set to Control, we gain a measure of reuse of our style across classes that derive from Control—e.g., Button, Label, Window, etc. However, if we drop the TargetType attribute from the style altogether, we gain a measure of reuse of styles across controls that don't have a common base but share a dependency-property implementation. In my experimentation, I've found that dependency properties that share the same name across classes, such as Control.FontSize and TextBlock.FontSize, also share an implementation. What that means is that even though Control and TextBlock each define their own FontSize property, at runtime they share the implementation of this property, so I can write code like Example 5-8.

Example 5-8. Reusing a style between different element types

```
...
<Style x:Key="CellTextStyle">
  <Setter Property="Control.FontSize" Value="32" />
</Style>
...
<!-- derives from Control -->
<Button Style="{StaticResource CellTextStyle}" ... />

<!-- does *not* derive from Control -->
<TextBlock Style="{StaticResource CellTextStyle}" ... />
...
```

In Example 5-8, I've dropped the TargetType attribute from the style definition, using instead the class prefix on each property the style sets. This style can be applied to a Button element, as you'd expect, but can also be applied to a TextBlock control, with the FontSize set as specified by the style. The reason this works is that both the Button, which gets its FontSize dependency property *definition* from the Control class, and the TextBlock, which provides it's own FontSize dependency property *definition*, share the FontSize dependency property *implementation* with the TextElement class. Figure 5-2 shows the relationship of elements to their dependency-property implementations.

As Figure 5-2 shows, if we wanted to, we could redefine our style in terms of the TextElement class, even though it falls into the inheritance tree of neither Control nor TextBlock, as in Example 5-9.

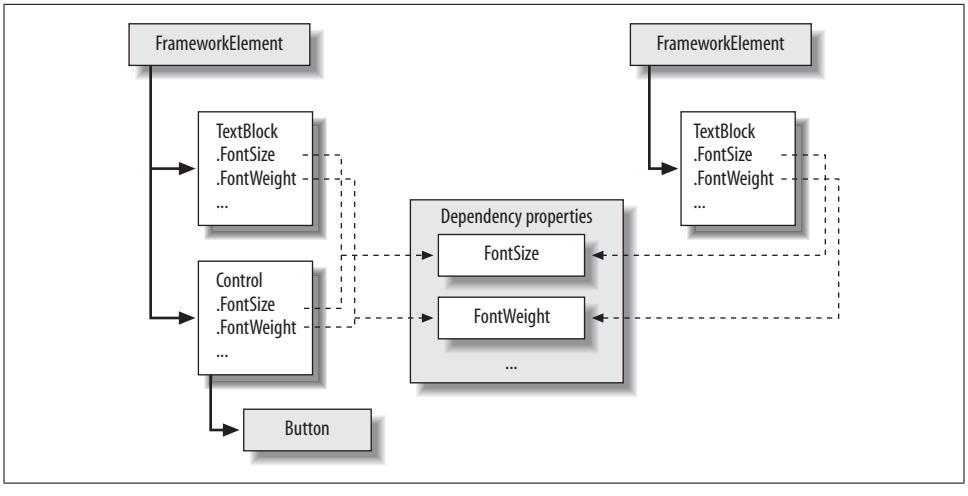


Figure 5-2. Elements and dependency properties

Example 5-9. Depending on the implementation of dependency properties

```
<Style x:Key="CellTextStyle">
  <Setter Property="TextElement.FontSize" Value="32" />
</Style>
...
<Button Style="{StaticResource CellTextStyle}" ... />
<TextBlock Style="{StaticResource CellTextStyle}" ... />
```

Taking this further, if we'd like to define a style that contains properties not shared by every element we apply them to, we can do that, too, as in Example 5-10.

Example 5-10. Styles can have properties that targets don't have

```
<Style x:Key="CellTextStyle">
  <Setter Property="TextElement.FontSize" Value="32" />
  <Setter Property="Button.IsCancel" Value="False" />
</Style>
...
<!-- has an IsCancel property -->
<Button Style="{StaticResource CellTextStyle}" ... />

<!-- does *not* have an IsCancel property -->
<TextBlock Style="{StaticResource CellTextStyle}" ... />
```

In Example 5-10, we've added the `Button.IsCancel` property to the `CellTextStyle` and applied it to the `Button` element, which has this property, and the `TextBlock` element, which doesn't. This is OK. At runtime, WPF will apply the dependency properties that exist on the elements that have them and silently swallow the ones that aren't present.

WPF's ability to apply styles to objects that don't have all of the properties defined in the style is analogous to applying the Word Normal style, which includes a font size property of its own, to both a range of text and an image. Even though Word knows that images don't have a font size, it applies the portions of the Normal style that do make sense (such as the justification property), ignoring the rest.

Getting back to our sample, we can use the `CellTextStyle` on a `TextBlock` in a new row to show whose turn it is, as in Example 5-11.

Example 5-11. Applying a style to Button and TextBlock elements

```
<Window.Resources>
  <Style x:Key="CellTextStyle">
    <Setter Property="TextElement.FontSize" Value="32" />
    <Setter Property="TextElement.FontWeight" Value="Bold" />
  </Style>
</Window.Resources>
<Grid Background="Black">
  <Grid.RowDefinitions>
    <RowDefinition />
    <RowDefinition />
    <RowDefinition />
    <RowDefinition Height="Auto" />
  </Grid.RowDefinitions>
  <Grid.ColumnDefinitions>
    <ColumnDefinition />
    <ColumnDefinition />
    <ColumnDefinition />
  </Grid.ColumnDefinitions>
  <Button Style="{StaticResource CellTextStyle}" ... />
  ...
  <TextBlock
    Style="{StaticResource CellTextStyle}"
    Foreground="White"
    Grid.Row="3"
    Grid.ColumnSpan="3"
    x:Name="statusTextBlock" />
</Grid>
</Window>
```

This reuse of the style across controls of different types gives me a consistent look in my application, as shown in Figure 5-3.

One thing you'll notice is that the status text in Figure 5-3 is white, while the text in the buttons is black. Since black is the default text color, if we want the status text to show up against a black background, we have to change the color to something else, hence the need to set the `Foreground` property to white on the `TextBlock`. Setting per-instance properties works just fine in combination with the style, and you can combine the two techniques of setting property values as you see fit.

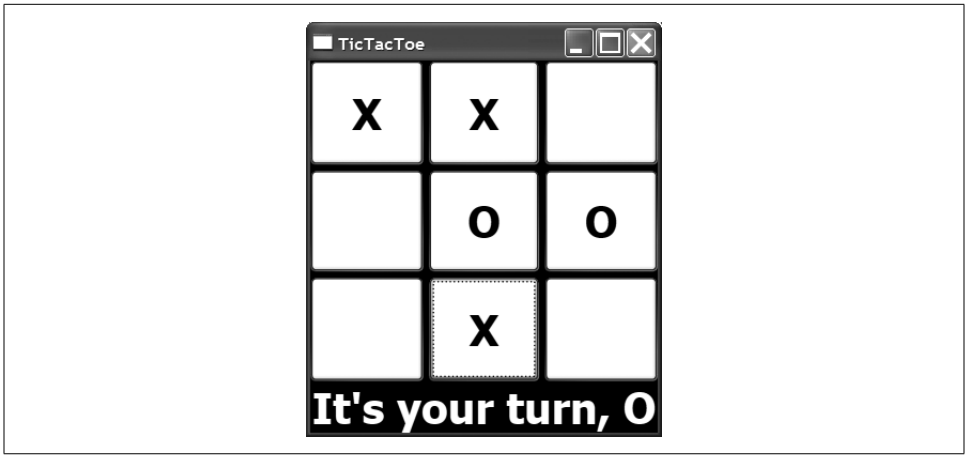


Figure 5-3. A tic-tac-toe game with style

Overriding Style Properties

Further, if we want to *override* a style property on a specific instance, we can do so by setting the property on the instance, as in Example 5-12.

Example 5-12. Overriding the FontWeight property from the style

```
<Style x:Key="CellTextStyle">
  <Setter Property="TextElement.FontSize" Value="32" />
  <Setter Property="TextElement.FontWeight" Value="Bold" />
</Style>
...
<TextBlock
  Style="{StaticResource CellTextStyle}"
  FontWeight="Thin" ... />
```

In Example 5-12, the `TextBlock` instance property setting of `FontWeight` take precedence over the style property settings of `FontWeight`.

Inheriting Style Properties

To complete the object-oriented triumvirate of reuse, override, and inheritance, you can *inherit* a style from a base style, adding new properties or overriding existing ones, as in Example 5-13.

Example 5-13. Style inheritance

```
<Style x:Key="CellTextStyle">
  <Setter Property="TextElement.FontSize" Value="32" />
  <Setter Property="TextElement.FontWeight" Value="Bold" />
</Style>
<Style x:Key="StatusTextStyle" BasedOn="{StaticResource CellTextStyle}">
```

Example 5-13. Style inheritance (continued)

```
  <Setter Property="TextElement.FontWeight" Value="Thin" />
  <Setter Property="TextElement.Foreground" Value="White" />
  <Setter Property="TextBlock.HorizontalAlignment" Value="Center" />
</Style>
```

The `BasedOn` style attribute is used to designate the base style. In Example 5-13, the `StatusTextStyle` style inherits all of the `CellTextStyle` property setters, overrides the `FontWeight`, and adds setters for `Foreground` and `HorizontalAlignment`. Notice that the `HorizontalAlignment` property uses a `TextBlock` prefix; this is because `TextElement` doesn't have a `HorizontalAlignment` property.

Our current use of styles causes our tic-tac-toe game to look like Figure 5-4.

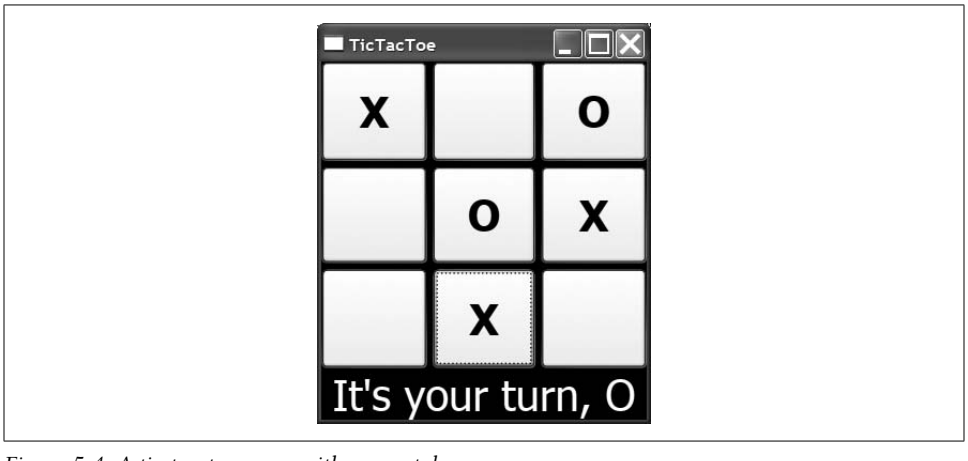


Figure 5-4. A tic-tac-toe game with more style

Our application so far is pretty good, especially with the thin font weight on the status text, but we can do better.

Setting Styles Programmatically

Once a style has a name, it's easily available from our code. For example, we might decide that we'd like each player to have their own style. In this case, using named styles in XAML at compile time won't do the trick, since we want to set the style based on the content, which isn't known until runtime. However, there's nothing that requires us to set the `Style` property of a control statically; we can set it programmatically as well, as we do in Example 5-14.

Example 5-14. Setting styles programmatically

```
public partial class Window1 : Window {
  void cell_Click(object sender, RoutedEventArgs e) {
    Button button = (Button)sender;
```

Example 5-14. Setting styles programmatically (continued)

```
    ...
    // Set button content
    button.Content = this.CurrentPlayer;
    ...
    if( this.CurrentPlayer == "X" ) {
      button.Style = (Style)FindResource("XStyle");
      this.CurrentPlayer == "O";
    }
    else {
      button.Style = (Style)FindResource("OStyle");
      this.CurrentPlayer == "X";
    }
    ...
  }
  ...
}
```

In Example 5-14, whenever the player clicks, in addition to setting the button's content, we pull a named style out of the window's resources and use that to set the button's style. This assumes a pair of named styles defined in the window's scope, as in Example 5-15.

Example 5-15. Styles pulled out via FindResource

```
<Window.Resources>
  <Style x:Key="CellTextStyle">
    <Setter Property="TextElement.FontSize" Value="32" />
    <Setter Property="TextElement.FontWeight" Value="Bold" />
  </Style>
  <Style x:Key="XStyle" BasedOn="{StaticResource CellTextStyle}">
    <Setter Property="TextElement.Foreground" Value="Red" />
  </Style>
  <Style x:Key="OStyle" BasedOn="{StaticResource CellTextStyle}">
    <Setter Property="TextElement.Foreground" Value="Green" />
  </Style>
</Window.Resources>
```

With these styles in place and the code to set the button style along with content, we get Figure 5-5.

Notice that the Xs and Os are colored according to the named player styles. In this particular case (and in many other cases, too), data triggers (discussed in "Data Triggers," later in this chapter) should be preferred to setting styles programmatically, but you never know when you're going to have to jam.

As with all XAML constructs, you are free to create styles themselves programmatically. Appendix A is a good introduction on how to think about going back and forth between XAML and code.

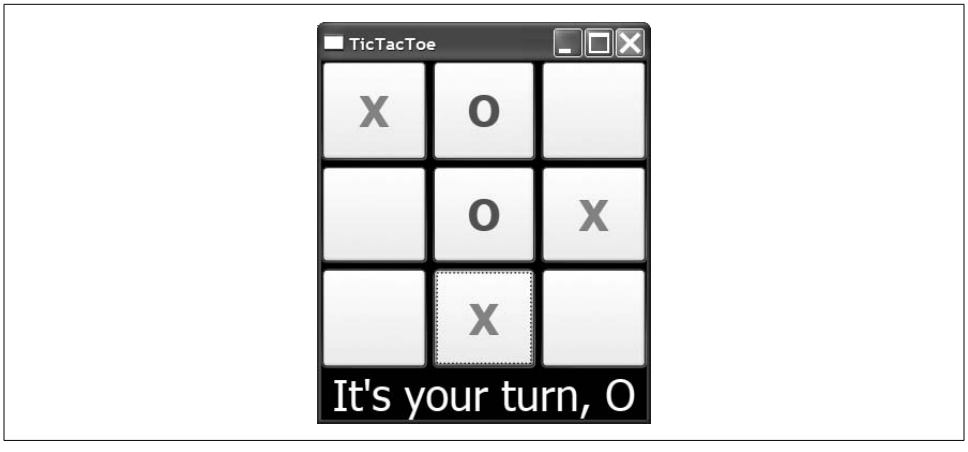


Figure 5-5. Setting styles programmatically based on an object's content (Color Plate 9)

Element-Typed Styles

Named styles are useful when you've got a set of properties to be applied to specific elements. However, if you'd like to apply a style uniformly to all instances of a certain type of element, set the `TargetType` without a `Key`, as in Example 5-16.

Example 5-16. Element-typed styles

```
...
<!-- no Key -->
<Style TargetType="{x:Type Button}">
  <Setter Property="FontSize" Value="32" />
  <Setter Property="FontWeight" Value="Bold" />
</Style>
<!-- no Key -->
<Style TargetType="{x:Type TextBlock}">
  <Setter Property="FontSize" Value="32" />
  <Setter Property="FontWeight" Value="Thin" />
  <Setter Property="Foreground" Value="White" />
  <Setter Property="HorizontalAlignment" Value="Center" />
</Style>
...
<Button Grid.Row="0" Grid.Column="0" x:ID="cell00" />
...
<TextBlock Grid.Row="5" Grid.ColumnSpan="5" x:ID="statusTextBlock" />
...
```

In Example 5-16, we've got two styles, one with a `TargetType` of `Button` and no `Key` and another with a `TargetType` of `TextBlock` and no `Key`. Both work in the same way; when an instance of `Button` or `TextBlock` is created without an explicit `Style` attribute setting, it uses the style that matches the target type of the style to the type of the control. Our element-typed styles return our game to looking like Figure 5-4 again.

Element-typed styles are handy whenever you'd like all instances of a certain element to share a look, depending on the scope. For example, we've scoped the styles in our sample thus far at the top-level Window in Example 5-17.

Example 5-17. Styles scoped to the Window

```
<!-- Window1.xaml -->
<Window ...>
  <!-- every Button or TextBlock in the Window is affected -->
  <Window.Resources>
    <Style TargetType="{x:Type Button}">...</Style>
    <Style TargetType="{x:Type TextBlock}">...</Style>
  </Window.Resources>
  ...
</Window>
```

However, you may want to reduce the scope of an element-typed style. In our sample, it would work just as well to scope the styles inside the grid so that only buttons and text blocks in the grid are affected, as in Example 5-18.

Example 5-18. Styles scoped below the Window

```
<!-- Window1.xaml -->
<Window ...>
  <Grid ...>
    <!-- only Buttons or TextBlocks in the Grid are affected -->
    <Grid.Resources>
      <Style TargetType="{x:Type Button}">...</Style>
      <Style TargetType="{x:Type TextBlock}">...</Style>
    </Grid.Resources>
    ...
  </Grid>
  <!-- Buttons and TextBlocks outside the Grid are unaffected -->
  ...
</Window>
```

Or, if you want to make your styles have greater reach in your project, you can put them into the application scope, as in Example 5-19.

Example 5-19. Styles scoped to the application

```
<!-- MyApp.xaml -->
<Application ...>
  <!-- every Button or TextBlock in the Application is affected -->
  <Application.Resources>
    <Style TargetType="{x:Type Button}">...</Style>
    <Style TargetType="{x:Type TextBlock}">...</Style>
  </Application.Resources>
</Application>
```

In general it's useful to understand the scoping rules of element-typed styles so you can judge their effect on the various pieces of your WPF object model. Chapter 6 discusses resource scoping of all kinds, including styles, in more detail.

Named versus element-typed styles

When choosing between styles set by style name or by element type, one of our reviewers said that in his experience, once you get beyond 10 styles based on element type, it was too hard to keep track of where a particular control was getting its style from. This is one reason that I'm a big fan of named styles.

To me, a style is a semantic tag that will be applied to content in one place and awarded a visual representation in another. As simple as our tic-tac-toe sample is, we've already got two styles, one for the status text and one for the move cell; before we're done, we're going to have more. The major differentiating factor is going to be the kind of data we'll be showing in these elements, not the type of the element holding the data. In fact, we'll have several styles assigned to `TextBox` controls, which negates the use of type-based styles anyway, even for this simple application.

Data Templates and Styles

Let's imagine that we wanted to implement a version of tic-tac-toe that's more fun to play (that's an important feature in most games). For example, one variant of tic-tac-toe allows players to have only three of their pieces on at any one time, dropping the first move off when the fourth move is played, dropping the second move when the fifth is played, and so on. To implement this variant, we need to keep track of the sequence of moves, which we can do with a `PlayerMove` class, as in Example 5-20.

Example 5-20. A custom type suitable for tracking tic-tac-toe moves

```
namespace TicTacToe {
  public class PlayerMove {
    private string playerName;
    public string PlayerName {
      get { return playerName; }
      set { playerName = value; }
    }

    private int moveNumber;
    public int MoveNumber {
      get { return moveNumber; }
      set { moveNumber = value; }
    }

    public PlayerMove(string playerName, int moveNumber) {
      this.playerName = playerName;
      this.moveNumber = moveNumber;
    }
  }
}
```

Now, instead of using a simple string for each of the button object's content, we'll use an instance of PlayerMove in Example 5-21. Figure 5-6 shows the brilliance of such a change.

Example 5-21. Adding the PlayerMove as Button content

```
namespace TicTacToe {
  public partial class Window1 : Window {
    ...
    int moveNumber;

    void NewGame( ) {
      ...
      this.moveNumber = 0;
    }

    void cell_Click(object sender, RoutedEventArgs e) {
      ...
      // Set button content
      //button.Content = this.CurrentPlayer;
      button.Content =
        new PlayerMove(this.CurrentPlayer, ++this.moveNumber);
      ...
    }
    ...
  }
}
```

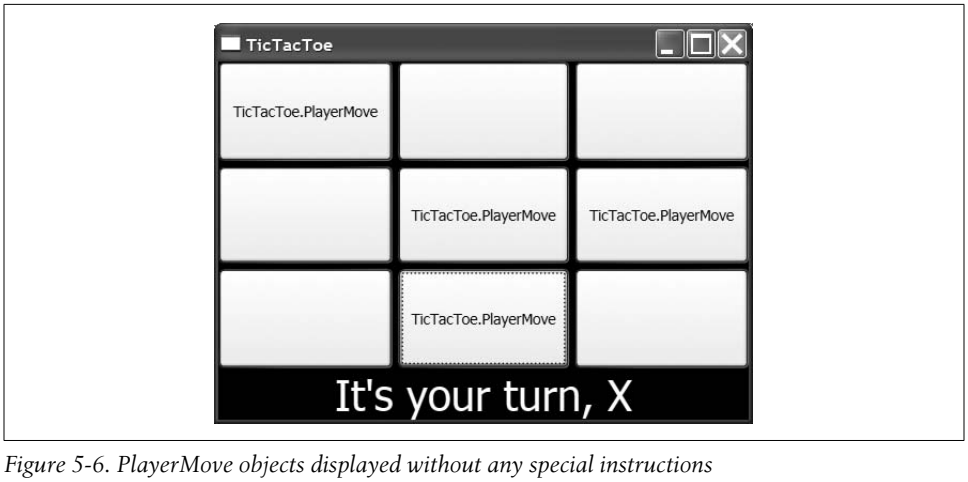


Figure 5-6. PlayerMove objects displayed without any special instructions

As you'll recall from Chapter 4, what's happening in Figure 5-6 is that the button doesn't have enough information to render a PlayerMove object, but we can fix that with a data template.

Data Templates

Recall from Chapter 4 that WPF allows you to define a data template, which is a tree of elements to expand in a particular context. Data templates are used to provide an application with the ability to render non-visual objects, as shown in Example 5-22.

Example 5-22. Setting a PlayerMove data template without styles

```
<?Mapping XmlNamespace="l" ClrNamespace="TicTacToe" ?>
<Window ... xmlns:local="local">
  <Window.Resources>
    <DataTemplate DataType="{x:Type local:PlayerMove}">
      <Grid>
        <TextBlock
          TextContent="{Binding Path=PlayerName}"
          FontSize ="32"
          FontWeight="Bold"
          VerticalAlignment="Center"
          HorizontalAlignment="Center" />
        <TextBlock
          TextContent="{Binding Path=MoveNumber}"
          FontSize="16"
          FontStyle="Italic"
          VerticalAlignment="Bottom"
          HorizontalAlignment="Right" />
      </Grid>
    </DataTemplate>
    ...
  </Window.Resources>
  ...
</Window>
```

Using the XAML mapping syntax introduced in Chapter 1, we've mapped the `PlayerMover` type into the XAML with the mapping directive and the `xmlns` attribute, which we've used as the data type of the data template. Now, whenever WPF sees a `PlayerMove` object, such as the content of all of our buttons, the data template will be expanded. In our case, the template consists of a grid to arrange two text blocks, one showing the player name in the middle of the button and one showing the move number in the bottom right, along with some other settings to make things pretty.

Data Templates with Style

However, these property settings are buried inside a data template several layers deep. Just as it's a good idea to take "magic numbers" out of your code, pulling them out and giving them names for easy maintenance, it's a good idea to move groups of settings into styles,* as in Example 5-23.

* Moving groups of settings into styles also allows for easier skinning and theming, as described in Chapter 6.

Example 5-23. Setting a PlayerMove data template with styles

```
<Window.Resources>
  <Style x:Key="CellTextStyle" TargetType="{x:Type TextBlock}">
    <Setter Property="FontSize" Value="32" />
    <Setter Property="FontWeight" Value="Bold" />
    <Setter Property="VerticalAlignment" Value="Center" />
    <Setter Property="HorizontalAlignment" Value="Center" />
  </Style>
  <Style x:Key="MoveNumberStyle" TargetType="{x:Type TextBlock}">
    <Setter Property="FontSize" Value="16" />
    <Setter Property="FontStyle" Value="Italic" />
    <Setter Property="VerticalAlignment" Value="Bottom" />
    <Setter Property="HorizontalAlignment" Value="Right" />
  </Style>
  <DataTemplate DataType="{x:Type l:PlayerMove}">
    <Grid>
      <TextBlock
        TextContent="{Binding Path=PlayerName}"
        Style="{StaticResource CellTextStyle}" />
      <TextBlock
        TextContent="{Binding Path=MoveNumber}"
        Style="{StaticResource MoveNumberStyle}" />
    </Grid>
  </DataTemplate>
</Window.Resources>
```

It's common to use styles, which set groups of properties, with data templates, which create groups of elements that have groups of properties. Figure 5-7 shows the result.

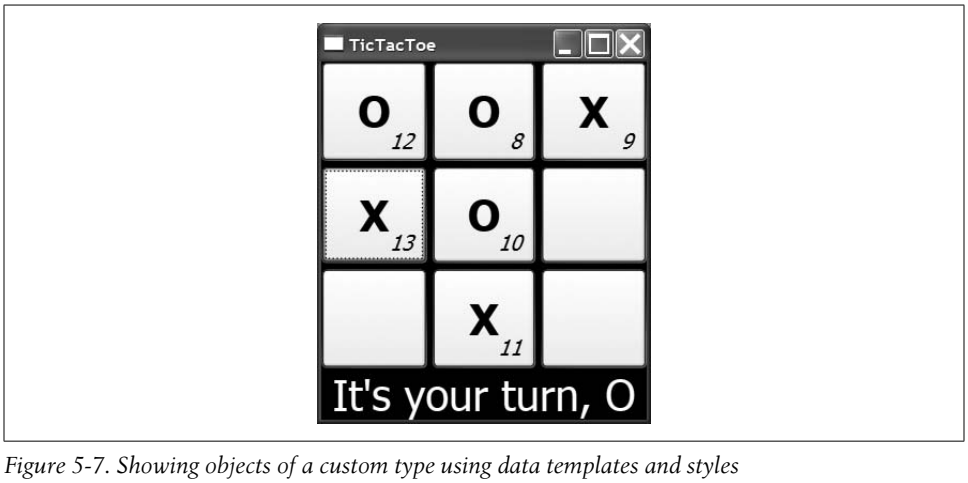


Figure 5-7. Showing objects of a custom type using data templates and styles

Still, as nice as Figure 5-7 is, the interaction is kind of boring given the capabilities of WPF. Let's see what we can do with style properties as the application is used.

Triggers

So far, we've seen styles as a collection of `Setter` elements. When a style is applied, the settings described in the `Setter` elements are applied unconditionally (unless overridden by per-instance settings). On the other hand, *triggers* are a way to wrap one or more `Setter` elements in a condition so that, if the condition is true, the corresponding `Setter` elements are executed, and when the condition becomes false, the property value reverts to its pre-trigger value.

WPF comes with three kinds of things that you can check in a trigger condition: a dependency property, a .NET property, and an event. The first two directly change values based on a condition, as I described, while the last, an event trigger, is activated when an event happens and then starts (or stops) an animation that causes properties to change.

Property Triggers

The simplest form of a trigger is a property trigger, which watches for a dependency property to have a certain value. For example, if we wanted to light up a button in yellow as the user moves the mouse over it, we can do so by watching for the `IsMouseOver` property to have a value of `True`, as in Example 5-24.

Example 5-24. A simple property trigger

```
<Style TargetType="{x:Type Button}">
  ...
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True" >
      <Setter Property="Background" Value="Yellow" />
    </Trigger>
  </Style.Triggers>
</Style>
```

Triggers are grouped together under the `Style.Triggers` element. In this case, we've added a `Trigger` element to the button style. When the `IsMouseOver` property of our button is true, the `Background` value of the button will be set to yellow, as shown in Figure 5-8.

You'll notice in Figure 5-8 that only the button where the mouse is currently hovering has its background set to yellow, even though other buttons have clearly been under the mouse. There's no need to worry about setting a property back when the trigger is no longer true—e.g., watching for `IsMouseOver` to be `False`. The WPF dependency-property system watches for the property trigger to become inactive and reverts to the previous value.

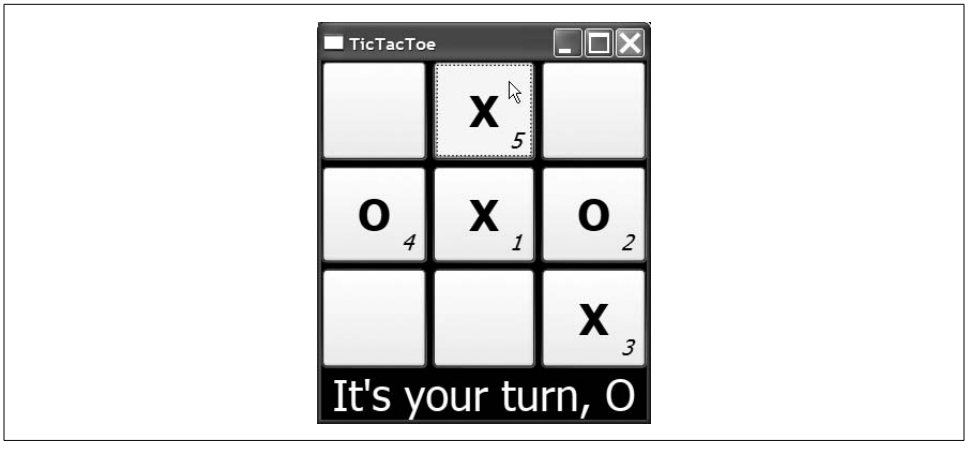


Figure 5-8. A property trigger in action

Property triggers can be set to watch any of the dependency properties on the control to which your style is targeted and to set any of the dependency properties on the control while the condition is true. In fact, you can use a single trigger to set multiple properties if you like, as in Example 5-25.

Example 5-25. Setting multiple properties with a single trigger

```
<Style TargetType="{x:Type Button}">
  ...
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True" >
      <Setter Property="Background" Value="Yellow" />
      <Setter Property="FontStyle" Value="Italic" />
    </Trigger>
  </Style.Triggers>
</Style>
```

In Example 5-25, we're setting the background to yellow and the font style to italic when the mouse is over a button.

One property that you're not allowed to set in a trigger is the `Style` property itself. If you try, you'll get the following error:

> A `Style` object is not allowed to affect the `Style` property of the object to which it applies.

This makes sense. Since it's the `Style` that's setting the property, and that participates in "unsetting" it when the trigger is no longer true, what sense would it make to change the very `Style` that's providing this orchestration? This would be somewhat like switching out your skis after you've launched yourself off of a jump but before you've landed.

Multiple Triggers

While you can set as many properties as you like in a property trigger, there can be more than one trigger in a style. When grouped together under the `Style.Triggers` element, multiple triggers act independently of each other.

For example, we can update our code so that if the mouse is hovering over one of our buttons, it'll be colored yellow and if the button has focus (the tab and arrow keys move focus around), it'll be colored green, as in Example 5-26. Figure 5-9 shows the result of one cell having focus and another with the mouse hovering.

Example 5-26. Multiple property triggers

```
<Style TargetType="{x:Type Button}">
  ...
  <Style.Triggers>
    <Trigger Property="IsMouseOver" Value="True" >
      <Setter Property="Background" Value="Yellow" />
    </Trigger>
    <Trigger Property="IsFocused" Value="True" >
      <Setter Property="Background" Value="LightGreen" />
    </Trigger>
  </Style.Triggers>
</Style>
```

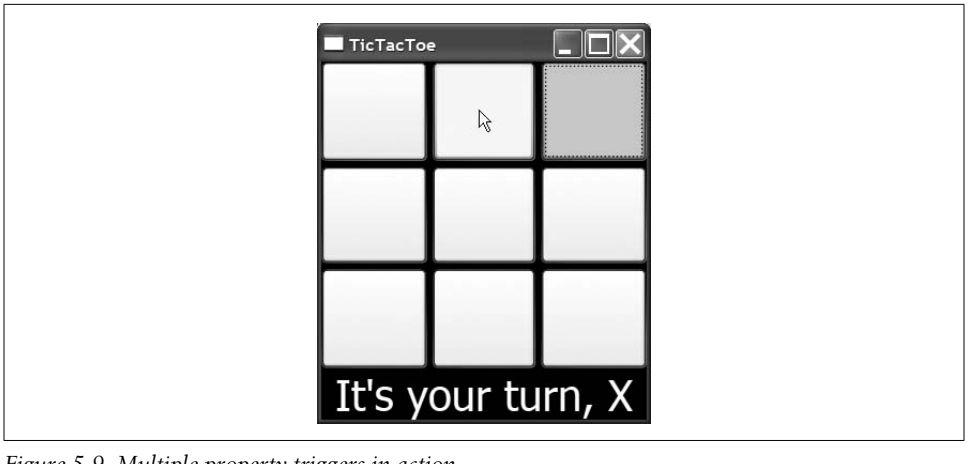


Figure 5-9. Multiple property triggers in action

If multiple triggers set the same property, the last one wins. For example, in Figure 5-9, if a button has focus and the mouse is over it, the background will be light green because the trigger for the `IsFocused` trigger is last in the list of triggers.

Multi-Condition Property Trigger

If you'd like to check more than one property before a trigger condition is activated—e.g., the mouse is hovering over a button *and* the button content is empty—you can combine multiple conditions with a multiple-condition property trigger, as in Example 5-27.

Example 5-27. A multi-property trigger

```
<Style TargetType="{x:Type Button}">
  ...
  <Style.Triggers>
    <MultiTrigger>
      <MultiTrigger.Conditions>
        <Condition Property="IsMouseOver" Value="True" />
        <Condition Property="Content" Value="{x:Null}" />
      </MultiTrigger.Conditions>
      <Setter Property="Background" Value="Yellow" />
    </MultiTrigger>
  </Style.Triggers>
</Style>
```

Multi-condition property triggers check all of the properties' values to be set as specified, not just one of them. Here, we're watching for both a mouse hover and for the content to be null,* reflecting the game logic that only clicking on an empty cell will result in a move.

Figure 5-10 shows the yellow highlight on an empty cell when the mouse hovers, and Figure 5-11 shows the yellow highlight absent when the mouse hovers over a full cell.

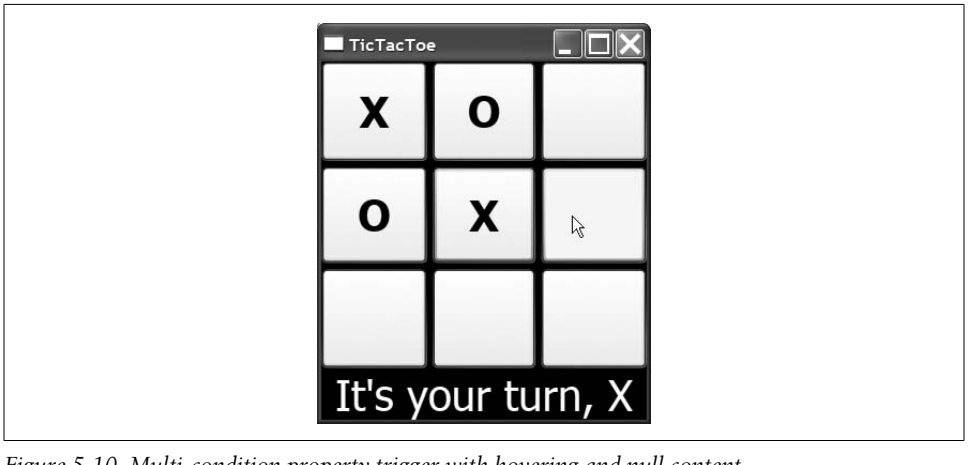


Figure 5-10. Multi-condition property trigger with hovering and null content

* The null value is set via a XAML markup extension, which you can read more about in Appendix A.

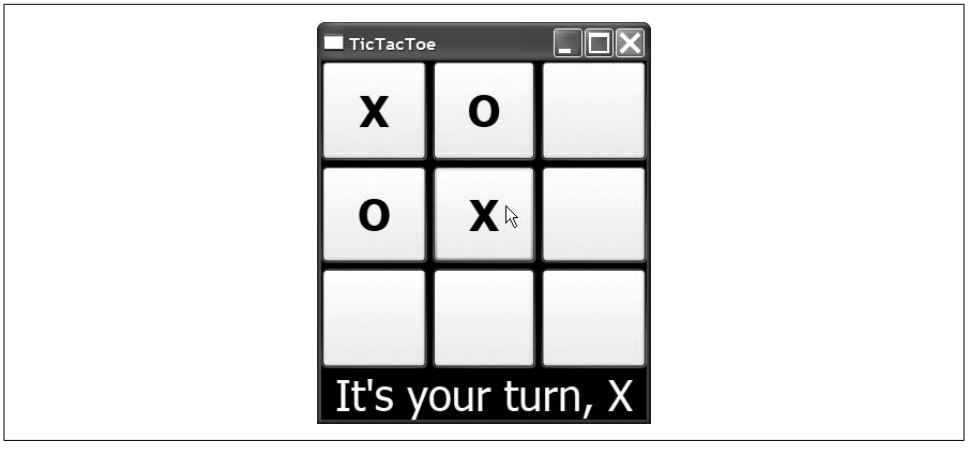


Figure 5-11. Multi-condition property trigger not triggering as content is not null

Property triggers are great for noticing when the user is interacting with a control displaying your program's state. However, we'd also like to be able to notice when the program's state itself changes, such as when a particular player makes a move, and update our style settings accordingly. For that, we have data triggers.

Data Triggers

Unlike property triggers, which check only WPF dependency properties, data triggers can check any old .NET object property. While property triggers are generally used to check WPF visual-element properties, data triggers are normally used to check the properties of non-visual objects used as content, such as our `PlayerMove` objects in Example 5-28.

Example 5-28. Two data triggers

```
<Window.Resources>
  <Style TargetType="{x:Type Button}">
    ...
  </Style>
  <Style x:Key="CellTextStyle" TargetType="{x:Type TextBlock}">
    ...
    <Style.Triggers>
      <DataTrigger Binding="{Binding Path=PlayerName}" Value="X">
        <Setter Property="Foreground" Value="Red" />
      </DataTrigger>
      <DataTrigger Binding="{Binding Path=PlayerName}" Value="O">
        <Setter Property="Foreground" Value="Green" />
      </DataTrigger>
    </Style.Triggers>
  </Style>
  <Style x:Key="MoveNumberStyle" TargetType="{x:Type TextBlock}">
    ...
```

Example 5-28. Two data triggers (continued)

```
  </Style>
  ...
  <DataTemplate DataType="{x:Type l:PlayerMove}">
    <Grid>
      <TextBlock
        TextContent="{Binding Path=PlayerName}"
        Style="{StaticResource CellTextStyle}" />
      <TextBlock
        TextContent="{Binding Path=MoveNumber}"
        Style="{StaticResource MoveNumberStyle}" />
    </Grid>
  </DataTemplate>
</Window.Resources>
```

`DataTrigger` elements go under the `Style.Triggers` element just like property triggers and, just like property triggers, there can be more than one of them active at any one time. While a property trigger operates on the properties of the visual elements displaying the content, a data trigger operates on the content itself. In our case, the content of each of the cells is a `PlayerMove` object. In both of our data triggers, we're binding to the `PlayerName` property. If the value is `X`, we're setting the foreground to red and if it's `O`, we're setting it to green.

Take care where you put the data trigger. In our example, we've got the `Button`-type style and the named `CellTextStyle` style as potential choices. I've written this chapter twice now and both times I've initially put the data trigger on the button style instead of on the content in the data template. Data triggers are based on content, so make sure you put them into your content styles, not your control styles.

We haven't had per-player colors since we moved to data templates after setting styles programmatically in Figure 5-5, but data triggers bring us that feature right back, along with all of the other features we've been building up, as shown in Figure 5-12.

Unlike property triggers, which rely on the change notification of dependency properties, data triggers rely on an implementation of the standard property-change notification patterns that are built into .NET and are discussed in Chapter 4—e.g., `INotifyPropertyChanged`. Since each `PlayerMove` object is constant, we don't need to implement this pattern, but if you're using data triggers, chances are that you will need to implement it on your custom content classes.

One other especially handy feature of data triggers is that there's no need for an explicit check for null content. If the content is null, the trigger condition is automatically false, which is why the application isn't crashing trying to dereference a null `PlayerMove` to get to the `PlayerName` property.

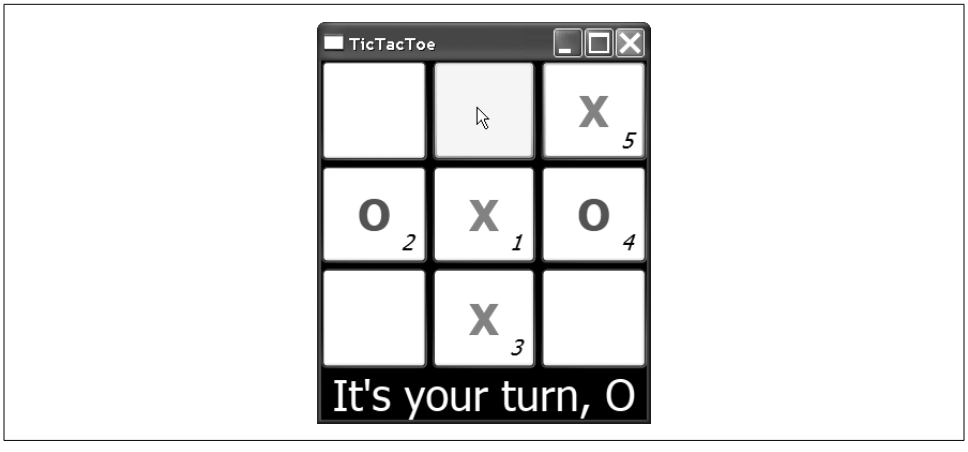


Figure 5-12. Data triggers in action

Multi-Condition Data Triggers

Just as property triggers can be combined into "and" conditions using the `MultiTrigger` element, data triggers can be combined using the `MultiDataTrigger` element. For example, if we wanted to watch for the 10th move of the game, make sure it's player "O," and do something special, we can do so as shown in Example 5-29.

Example 5-29. A multi-data trigger

```
<?Mapping XmlNamespace="sys" ClrNamespace="System" Assembly="mscorlib" ?>
...
<Window ... xmlns:sys="sys">
  <Style x:Key="MoveNumberStyle" TargetType="{x:Type TextBlock}">
    ...
    <Style.Triggers>
      <MultiDataTrigger>
        <MultiDataTrigger.Conditions>
          <Condition Binding="{Binding Path=PlayerName}" Value="O" />
          <Condition Binding="{Binding Path=MoveNumber}">
            <Condition.Value>
              <sys:Int32>10</sys:Int32>
            </Condition.Value>
          </Condition>
        </MultiDataTrigger.Conditions>
        <Setter Property="Background" Value="Yellow" />
      </MultiDataTrigger>
    </Style.Triggers>
  </Style>
  ...
</Window>
```

The only thing about Example 5-29 that may seem a little strange is the use of the mapping syntax to bring in the `System` namespace from the mscorlib .NET assembly. We do this so that we can pass `10` as an `Int32` instead of as a string; otherwise, the multi-condition data trigger won't match our `MoveNumber` property correctly. The multi-condition data trigger in Example 5-29 sets the background of the move number to yellow to connote a cause for celebration for this special move that regular tic-tac-toe doesn't have, but you can use multi-condition data triggers for celebrations of your own kinds.

Event Triggers

While property triggers check for values on dependency properties and data triggers check for values on CLR properties, event triggers watch for events. When an event happens, such as a `Click` event, an event trigger responds by raising an animation-related action. While animation is challenging enough to deserve its own chapter (Chapter 8), Example 5-30 illustrates a simple animation that will transition a cell from solid yellow to white over five seconds when an empty cell is clicked.

Example 5-30. An event trigger

```
<Style TargetType="{x:Type Button}">
  ...
  <Setter Property="Background" Value="White" />
  <Style.Storyboards>
    <ParallelTimeline Name="CellClickedTimeline" BeginTime="{x:Null}">
      <SetterTimeline Path="(Button.Background).(SolidColorBrush.Color)">
        <ColorAnimation From="Yellow" To="White" Duration="0:0:5" />
      </SetterTimeline>
    </ParallelTimeline>
  </Style.Storyboards>
  <Style.Triggers>
    <EventTrigger RoutedEvent="Click">
      <EventTrigger.Actions>
        <BeginAction TargetName="CellClickedTimeline" />
      </EventTrigger.Actions>
    </EventTrigger>
  </Style.Triggers>
</Style>
```

Adding an animation to a style requires two things. The first is a storyboard with a named timeline that describes what you want to happen. In our case, we're animating the button's background brush color from yellow to white over five seconds.

For any property being animated with a nested path, there needs to be an explicit property setting that creates the top level of the nesting. In Example 5-30, this means that we need a `Setter` element for the `Background` property. If the top level of the nesting isn't created, there won't be anything to animate at runtime.

The second thing you need is an event trigger to start the timeline. In our case, when the user clicks on a button with the `CellButtonStyle` style applied (all of them, in our case), we begin the action described by the named timeline in the storyboard.

As of this writing, if you have an event trigger and a multi-condition property trigger animating the same property—e.g., the `Background` of a `Button`, make sure you put the multi-data trigger in the XAML file before the event trigger; otherwise, you'll get a nonsensical error at runtime.

The results of the animation, showing various shades of yellow, through past clicks can be seen in Figure 5-13.

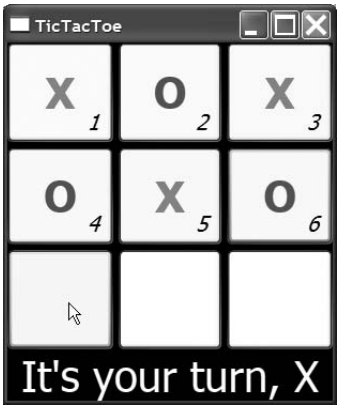


Figure 5-13. The event trigger and our fading yellow animation (Color Plate 10)

Property and data triggers let you set properties when properties change. Event triggers let you trigger events when events happen. While both are pretty different—e.g., you can't set a property with an event trigger or raise an event with a property or data trigger—both let you add a degree of interactivity to your applications in a wonderfully declarative way with little or no code.

For the full scoop on event triggers, you'll want to read Chapter 8.

Control Templates

If we take a closer look at our current tic-tac-toe game, we'll see that the `Button` objects aren't quite doing the job for us. What tic-tac-toe board has rounded inset corners (Figure 5-14)?

What we really want here is to be able to keep the behavior of the button—i.e., holding content and firing click events—but we want to take over the look of the

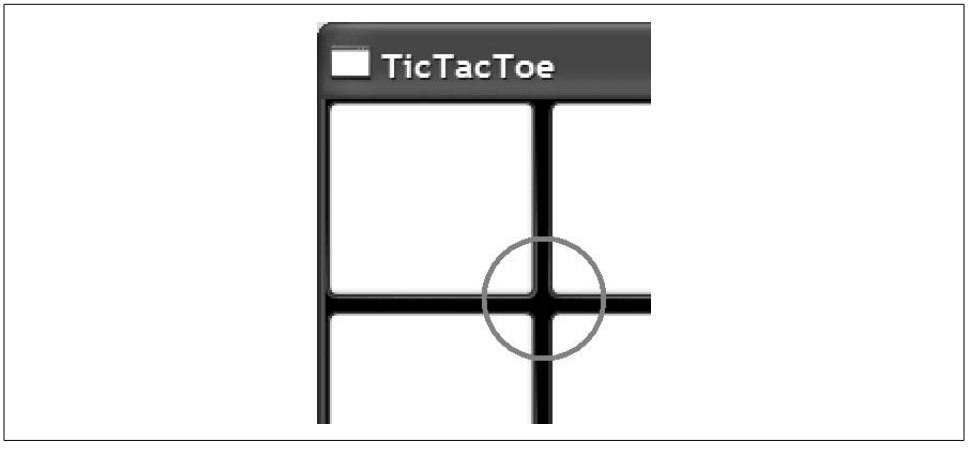


Figure 5-14. Tic-tac-toe boards don't have rounded insets!

button. WPF allows this kind of thing because the intrinsic controls are built to be *lookless*—i.e., they provide behavior, but the look can be swapped out completely by the client of the control.

Remember how we used data templates to provide the look of a non-visual object? We can do the same to a control using a *control template*, which is a set of storyboards, triggers, and, most importantly, elements that provide the look of a control.

To fix our buttons' looks, we'll build ourselves a control-template resource. Let's start things off in Example 5-31 with a simple rectangle and worry about showing the actual button content later.

Example 5-31. A minimal control template

```
<Window.Resources>
  <ControlTemplate x:Key="ButtonTemplate">
    <Rectangle />
  </ControlTemplate>
  ...
  <!-- let's just try one button for now... -->
  <Button Template="{StaticResource ButtonTemplate}" ... />
  ...
</Window.Resources>
```

Figure 5-15 shows the results of setting a single button's `Template` property.

Notice that no vestiges of what the button used to look like remain in Figure 5-15. Unfortunately, no vestige of our rectangles can be seen, either. The problem is that, without a fill explicitly set, the rectangle's fill defaults to transparent, showing the grid's black background. Let's set it to our other favorite Halloween color instead:

```
<ControlTemplate x:Key="ButtonTemplate">
  <Rectangle Fill="Orange" />
</ControlTemplate>
```

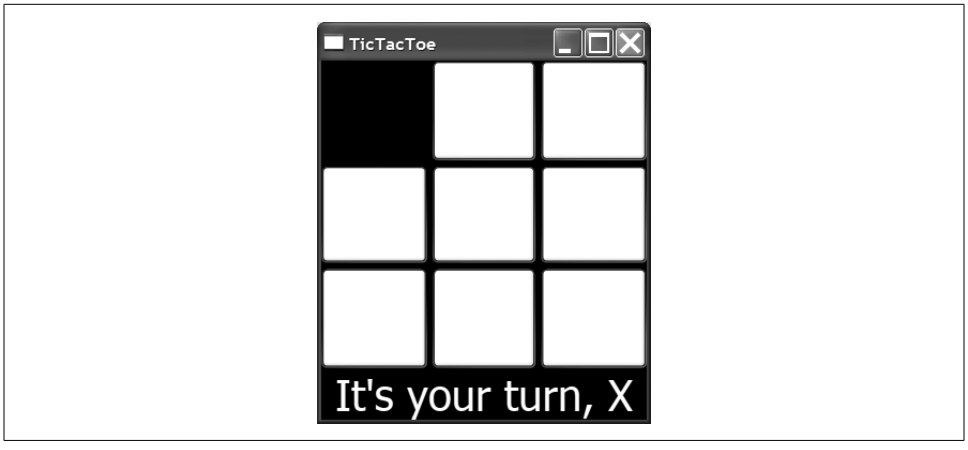


Figure 5-15. Replacing the control template with something less visual than we'd like…

Now we're getting somewhere, as Figure 5-16 shows.

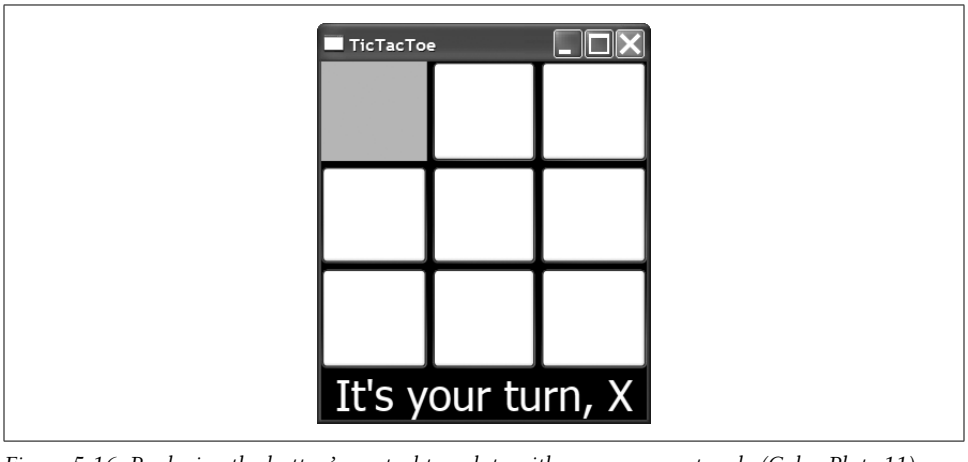


Figure 5-16. Replacing the button's control template with an orange rectangle (Color Plate 11)

Notice how square the corners are? Also, if you click, you won't get the depression that normally happens with a button (and I don't mean "a sad feeling").

Control Templates and Styles

Now that we're making some progress on the control template, let's replicate it to the other buttons. We can do so by setting each button's `Template` property by hand or, as is most common, we can bundle the control template with the button's style, as in Example 5-32.

Example 5-32. Putting a control template into a style

```
<Window.Resources>
  <Style TargetType="{x:Type Button}">
    ...
    <Setter Property="Template">
      <Setter.Value>
        <ControlTemplate>
          <Rectangle Fill="Orange" />
        </ControlTemplate>
      </Setter.Value>
    </Setter>
  </Style>
  ...
</Window.Resources>
...
<!-- No need to set the Template property for each button -->
<Button ... x:Name="cell00" />
...
```

As Example 5-32 shows, the `Template` property is the same as any other and can be set with a style. Figure 5-17 shows the results.

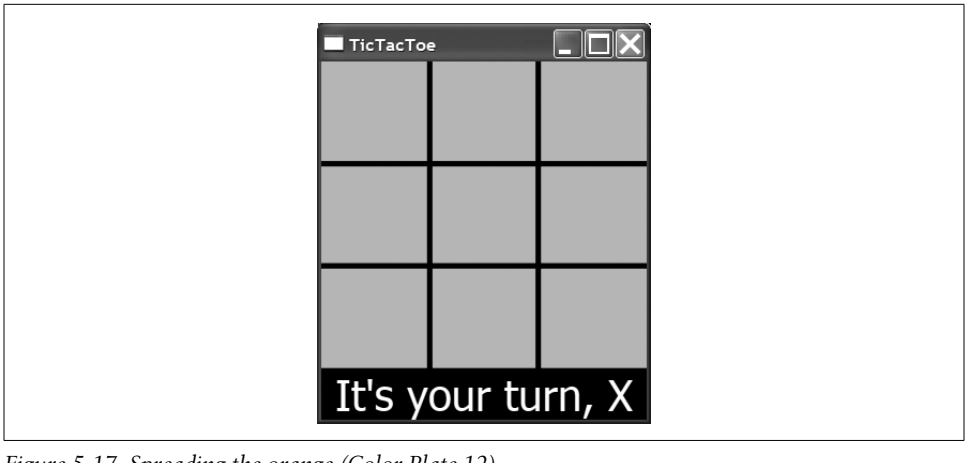


Figure 5-17. Spreading the orange (Color Plate 12)

Still, the orange is kind of jarring, especially since the settings on the style call for a white background. We can solve this problem with template bindings.

Template Binding

To get back to our white buttons, we could hardcode the rectangle's fill to be white, but what happens when a style wants to change it (such as in the animation we've now broken)? Instead of hardcoding the fill of the rectangle, we can reach out of the template into the properties of the control using template binding, as in Example 5-33.

Example 5-33. Template binding to the Background property

```
<Style TargetType="{x:Type Button}">
  <Setter Property="Background" Value="White" />
  ...
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate x:Key="ButtonTemplate">
        <Rectangle Fill="{TemplateBinding Property=Background}" />
      </ControlTemplate>
    </Setter.Value>
  </Setter>
  ...
</Style>
```

A *template binding* is like a data binding, except that the properties to bind come from the control whose template you're replacing (called the *templated parent*). In our case, things like `Background`, `HorizontalContentAlignment`, and so on, are fair game for template binding from the parent. And, like data binding, template bindings are smart enough to keep the properties of the items inside the template up to date with changing properties on the outside, as set by styles, animations, etc. For example, Figure 5-18 shows the effect of aliasing the rectangle's `Fill` property to the button's `Background` property with our click animation and mouse-over behavior still in place.

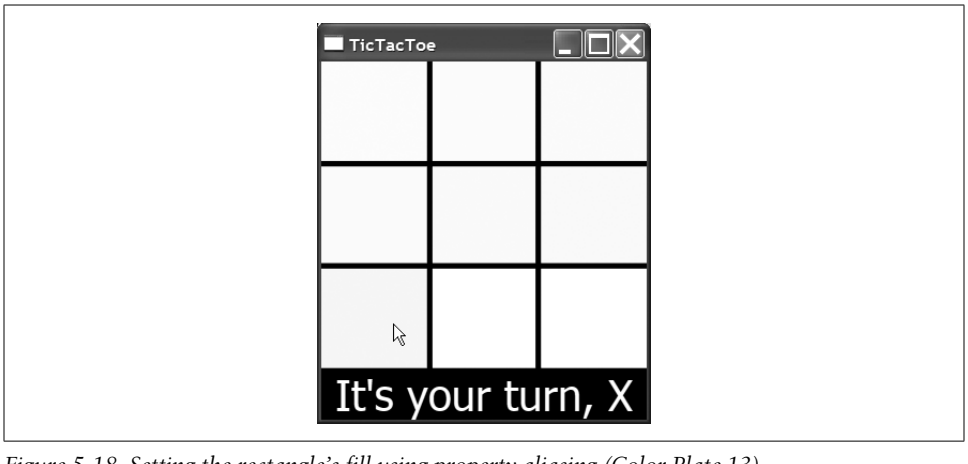


Figure 5-18. Setting the rectangle's fill using property aliasing (Color Plate 13)

We're not quite through yet, however. If we're going to change the paint swatch that Figure 5-18 has become into a playable game, we have to show the moves. To do so, we'll need a content presenter.

Content Presenters

If you've ever driven by a billboard or a bench at a bus stop that says "Your advertisement here!" then that's all you need to know to understand content presenters. A

content presenter is the WPF equivalent of "your content here" that allows content held by a `ContentContainer` control to be plugged in at runtime.

In our case, the content is the visualization of our `PlayerMove` object. Instead of reproducing all of that work inside of the button's new control template, we'd just like to drop it in at the right spot. The job of the content presenter is to take the content provided by the templated parent and do all of the things necessary to get it to show up properly, including styles, triggers, etc. The content presenter itself can be dropped into your template wherever you'd like to see it (including multiple times, if it tickles your fancy—e.g., to produce a drop shadow). In our case, we'll compose a content presenter in Example 5-34 with the rectangle inside a grid using techniques from Chapter 2.

Example 5-34. A content presenter

```
<Style TargetType="{x:Type Button}">
  <Setter Property="Background" Value="White" />
  ...
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate>
        <Grid>
          <Rectangle Fill="{TemplateBinding Property=Background}" />
          <ContentPresenter
            Content="{TemplateBinding Property=ContentControl.Content}" />
        </Grid>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
  ...
</Style>
```

In Example 5-34, the content presenter's `Content` property is bound to the `ContentControl.Content` property so that content comes through. As with styles, we can avoid prefixing template binding property names with classes by setting the `TargetType` attribute on the `ContentTemplate` element:

```
<ControlTemplate TargetType="{x:Type Button}">
  <Grid>
    <Rectangle Fill="{TemplateBinding Property=Background}" />
    <ContentPresenter
      Content="{TemplateBinding Property=Content}" />
  </Grid>
</ControlTemplate>
```

Further, with the `TargetType` property in place, you can drop the explicit template binding on the `Content` property altogether, as it's now set automatically:

```
<ControlTemplate TargetType="{x:Type Button}">
  <Grid>
    <Rectangle Fill="{TemplateBinding Property=Background}" />
    <!-- with TargetType set, the template binding for the -->
```

```
      <!-- Content property is no longer required -->
      <ContentPresenter />
    </Grid>
  </ControlTemplate>
```

The content presenter is all we need to get our game back to being functional, as shown in Figure 5-19.

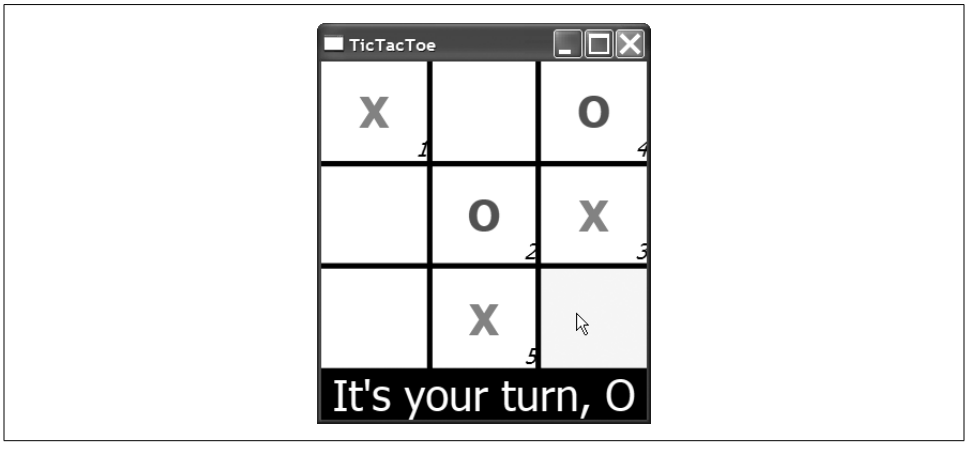


Figure 5-19. Adding a content presenter to our control template (Color Plate 14)

The Real Work

The last little bit of work is getting the padding right. Since the content presenter doesn't have its own `Padding` property, we can't bind the `Padding` property directly (it doesn't have a `Background` property, either, which is why we used `Rectangle` and its `Fill` property). For properties that don't have a match on the content presenter, you have to find mappings or compose the elements that provide the functionality that you're looking for. For example, `Padding` is an amount of space inside of a control. `Margin`, on the other hand, is the amount of space around the outside of a control. Since they're both of the same type, `System.Windows.Thickness`, if we could map the `Padding` from the inside of our button to the outside of the content control, our game would look very nice:

```
<Style TargetType="{x:Type Button}">
  <Setter Property="Background" Value="White" />
  <Setter Property="Padding" Value="10,5" />
  ...
  <Setter Property="Template">
    <Setter.Value>
      <ControlTemplate TargetType="{x:Type Button}">
        <Grid>
          <Rectangle Fill="{TemplateBinding Property=Background}" />
          <ContentPresenter
            Content="{TemplateBinding Property=Content}"
```

```
            Margin="{TemplateBinding Property=Padding}" />
        </Grid>
      </ControlTemplate>
    </Setter.Value>
  </Setter>
  ...
</Style>
```

Figure 5-20 shows our completed tic-tac-toe variation.

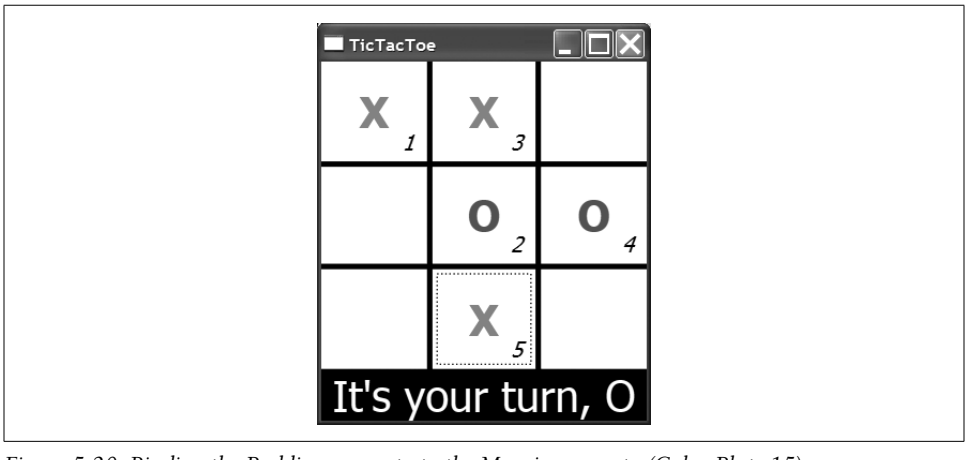


Figure 5-20. Binding the Padding property to the Margin property (Color Plate 15)

Like the mapping between `Padding` and `Margin`, building up the elements that give you the look you want and binding the appropriate properties from the templated parent is going to be a lot of the work of creating your own control templates.

Where Are We?

Styles enable you to define a policy for setting the dependency properties of visual elements. The sets of properties can be named and applied manually or programmatically by name, or applied automatically using element-typed styles. In addition to providing constant dependency-property values, styles can contain condition-based property values based on other dependency properties, data properties, or events. And, if setting properties isn't enough to get the look you're after, you can replace a lookless control's entire rendering behavior using a control template.

But that's not all there is to styles. For information about how animations work, you'll want to read Chapter 8, and for information about styles as related to resources, themes, and skins, you'll want to read Chapter 6.