1

Overview of Windows Presentation Foundation

For those of us who have been developing applications to run on the Windows platform, the topic of this book presents a compelling and exciting wave of change to how such applications can be built. In addition, for those of us who have been developing web-based applications, an even more exciting shift is approaching.

Windows Presentation Foundation, also known as *WPF*, is the next-generation graphics platform on which both Windows- and web-based applications can be built to run on Windows Vista, the latest evolutionary release of the Windows operating system. WPF provides the foundation for introducing an elegant and high fidelity User Experience (UX) by juxtaposition of user interface, application logic, documents, and media content.

> *Although originally targeted solely for Windows Vista, WPF will be made available for Windows XP and Windows Server 2003 as part of the .NET Framework 3.0 (formerly WinFX) developer platform.*

This coalescence of form and function is further empowered by tools such as XAML and the Microsoft Expression Designers, which allow designers and developers to work in parallel on the user interface and the application logic, coming together to provide a seamless UX.

This chapter provides an overview of WPF, including the following key topics:

- Evolution of the Windows API
- .NET Framework 3.0, the next-generation APIs for Windows-based development
- WPF architecture and development model
- XAML, the new declarative language backing WPF
- Tools to develop WPF applications

A Brief History of the Windows API

The Windows API exposes the core set of functionality provided by the Windows operating system for use in applications. Primarily designed for C/C++ development, the Windows API has been the most direct mechanism with which an application can interact with Windows.

The Windows API comprises the following functional groups:

- **Base Services** — Provides access to the core resources of the computer, such as memory, filesystems, devices, processes, and threads
- **Common Control Library** — A collection of common control windows used throughout the Windows operating system, providing the distinctive Windows look and feel
- **Common Dialog Box Library** — A collection of dialog boxes used to execute common tasks, including file opening, saving, and printing
- **Graphics Device Interface (GDI)** — Provides the facilities for an application to generate graphical output to displays, printers, and other devices
- **Network Services** — Provides access to various networking capabilities of the Windows operating system, including RPC and NetBIOS
- **User Interface (UI)** — Provides the mechanism for managing windows and controls in an application and input from devices such as the mouse and keyboard
- **Windows Shell** — The container that organizes and presents the entire Windows UI, including the desktop, taskbar, and Windows Explorer

Through these services, developers have had significant flexibility in creating powerful applications for the Windows operating system. However, this flexibility also bore the responsibility of handling low-level and often tedious operations.

With each new release of the Windows operating system, additions and updates to the Windows API were almost always included. Yet with each release, Microsoft strived to support backwards compatibility. Thus, many functions included in the original Windows API still exist today in Windows XP and Windows Server 2003.

The following list includes major versions of the Windows API:

- **Win16** — The API for the first 16-bit versions of Windows
- **Win32** — Introduced in Windows NT, the API for 32-bit versions of Windows
- **Win32 for 64-bit Windows** — Formerly known as Win64, the API for 64-bit versions of Windows XP and Windows Server 2003

Platform Evolution

Since the release of Windows 1.0 more than 20 years ago, the GDI and the UI services of the Windows API have provided a reliable graphics platform for Windows applications. Many applications we use on a day-to-day basis, including the Microsoft Office suite and Internet Explorer, are built on this foundation.

Although this graphics platform has seen many successes, fundamental changes have occurred in technology since its initial appearance more than two decades ago. With the proliferation of the personal computer and the increasing availability of broadband Internet access, the demand for rich visual experiences has dramatically increased. Moreover, advancements in graphics hardware technology have paved the way for astounding digital media and interactive entertainment products. Facing this new demand for a rich visual experience, Microsoft has invested heavily in providing such an experience as part of the next-generation Windows API — .NET Framework 3.0.

Introducing .NET Framework 3.0

.NET Framework 3.0 is a revolutionary milestone for developing applications on the Windows operating system. Built atop the .NET Framework 2.0, .NET Framework 3.0 is a set of managed APIs that provide enhanced functionality for messaging, workflow, and presentation.

Key components of .NET Framework 3.0 include:

- **Windows Presentation Foundation (WPF)** — The graphical subsystem for all things related to the UI
- **Windows Communication Foundation (WCF)** — The messaging subsystem of .NET Framework 3.0, securing program communication through a single API
- **Windows Workflow Foundation (WF)** — Provides workflow services for applications built to run on Windows

As the new programming model for Windows Vista, .NET Framework 3.0 fuses the Windows platform with applications developed using Visual Studio 2005. With direct access to low-level services of the operating system and hardware surface, .NET Framework 3.0 provides a compelling solution for creating applications with a rich UX.

> *Although WCF and WF are equally important components of .NET Framework 3.0, they are beyond the scope of this book. For more information on WCF and WF, visit the .NET Framework Developer Center on MSDN (*`http://msdn2.microsoft.com/en-us/netframework`*).*

Meet Windows Presentation Foundation

Formerly known as *Avalon*, Windows Presentation Foundation (WPF) is the new graphical subsystem in Windows Vista that provides a holistic means for combining user interface, 2D and 3D graphics, documents, and digital media. Built on the .NET Framework, WPF provides a managed environment for development with the Windows operating system. This takes advantage of the existing investment made by Microsoft in the .NET Framework, and allows developers familiar with .NET technologies to rapidly begin developing applications that leverage WPF.

Guiding Design Principles

To enhance UX and empower both designers and developers, WPF has been built under the following design principles:

- Integration
- Vector graphics
- Declarative programming
- Simplified deployment
- Document portability

Integration

In today's world, developing a Windows application may require the use of any number of different technologies, ranging from GDI/GDI+ for 2D graphics, UI services (User32 or WinForms), or Direct3D or OpenGL for 3D graphics. On the contrary, WPF was designed as a single model for application development, providing seamless integration between such services within an application. Similar constructs can be used for developing storyboard animation, data bound forms, and 3D models.

Vector Graphics

To take advantage of new powerful graphics hardware, WPF implements a vector-based composition engine. This allows for graphics to scale based on screen-specific resolution without loss of quality, something nearly impossible with fixed-size raster graphics. WPF leverages Direct3D for vector-based rendering, and will utilize the graphics processing unit (GPU) on any video card implementing DirectX 7 or later in hardware. In anticipation of future technology, such as high-resolution displays and unknown form factors, WPF implements a floating-point logical pixel system and supports 32-bit ARGB colors.

Declarative Programming

WPF introduces a new XML-based language to represent UI and user interaction, known as *XAML* (eXtensible Application Markup Language — pronounced "zammel"). Similar to Macromedia's MXML specification, within XAML elements from the UI are represented as XML tags. Thus, XAML allows applications to dynamically parse and manipulate UI elements at either compile-time or runtime, providing a flexible model for UI composition.

Following the success of ASP.NET, XAML follows the code-behind model, allowing designers and developers to work in parallel and seamlessly combine their work to create a compelling UX. With the aid of design-time tools such as the Visual Designer for Windows Presentation Foundation add-in for Visual Studio 2005, the experience of developing XAML-based applications resembles that of WinForms development. Moreover, designers accustomed to visual tools such as Macromedia Flash 8 Professional can quickly ramp-up to building XAML-based solutions using visual design tools such as Microsoft Expression Blend. These tools are covered later in this chapter and throughout this book.

Simplified Deployment

WPF applications can be deployed as standalone applications or as web-based applications hosted in Internet Explorer. As with smart client applications, web-based WPF applications operate in a partial-trust sandbox, which protects the client computer against applications with malicious purpose.

Furthermore, WPF applications hosted in Internet Explorer can exploit the capabilities of local client hardware, providing a rich web experience with 3D, digital media, and more, which is the best argument for web-based applications available today.

Document Portability

Included in WPF is an exciting new set of document and printing technologies. In conjunction with the release of Microsoft Office 12, WPF utilizes Open Packaging Conventions, which supports compression, custom metadata, digital signatures, and rights management. Similar to the Portable Document Format (PDF), the XML Paper Specification (XPS), which allows for documents to be shared across computers without requiring that the originating application be installed, is included in WPF.

Architecture

The anatomy of WPF consists of unmanaged services, managed subsystems, and a managed API available for consumption by WPF applications, known as the *presentation framework*.

Figure 1-1 outlines the general underlying architecture of WPF, with each major component detailed in the sections that follow.

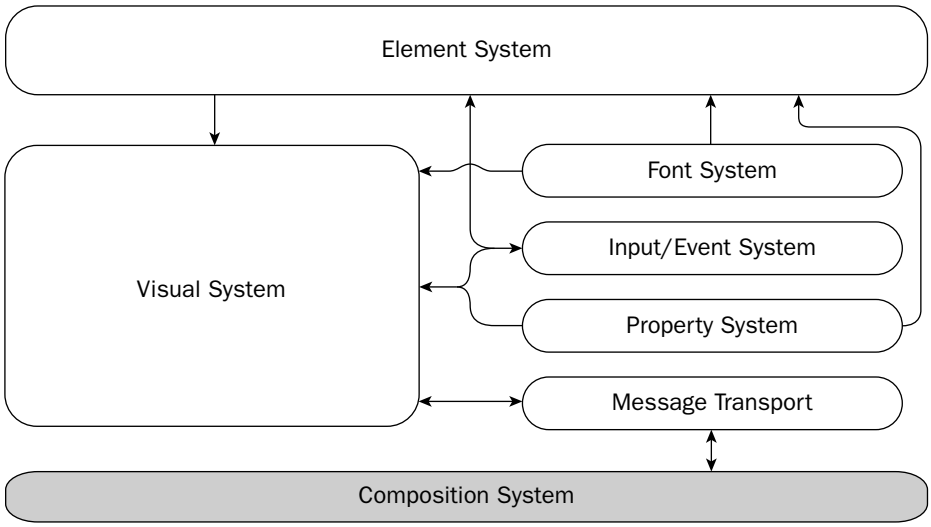

Figure 1-1

Element System

The *element system* represents the surface of WPF with which developers will interact. Contained within the element system are the core components making up the UI, such as styles, layout, controls, binding, and text layout.

Almost all the components in the element system derive from the all-important `System.Windows.FrameworkElement` class. This class provides the base functionality required to interact with WPF core presentation service, and implements key members of its parent class, `System.Windows.UIElement`,

which provides functionality for all visual elements within WPF for layout, eventing, and input. The `UIElement` class can be compared to `HWND` in Win32, and is the starting point for input-specific functionality within the WPF inheritance hierarchy. The `FrameworkElement` and related classes are discussed further in Chapter 2.

> *Most of the topics covered within this book related to WPF application development pertain to components and features of the element system. However, the underlying subsystems are indeed used extensively by WPF applications, and will be noted where appropriate.*

Element Trees

An important concept to grasp in WPF is the notion of element trees, which represent the visual elements of which a WPF application is comprised. Two element trees exist within a WPF application: the logical tree and the visual tree.

Logical Tree

The *logical tree* is the hierarchical structure containing the exact elements of your WPF application as defined either declaratively within a XAML file, or imperatively in code. Consider the following XAML code snippet:

```
<Window x:Class="ElementTrees.Sample"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="ElementTrees" Height="300" Width="300"
  >
  <StackPanel Name="drawingCanvas">
    <Label Name="nameLabel">Please Enter Your Name:</Label>
    <TextBox Name="nameTextBox" Margin="2px"></TextBox>
    <Button Name="submitButton" Margin="2px"
            Click="submitButton_Click">Submit</Button>
  </StackPanel>
</Window>
```

When compiled for or parsed under WPF, this code snippet would yield a logical tree like that depicted in Figure 1-2.

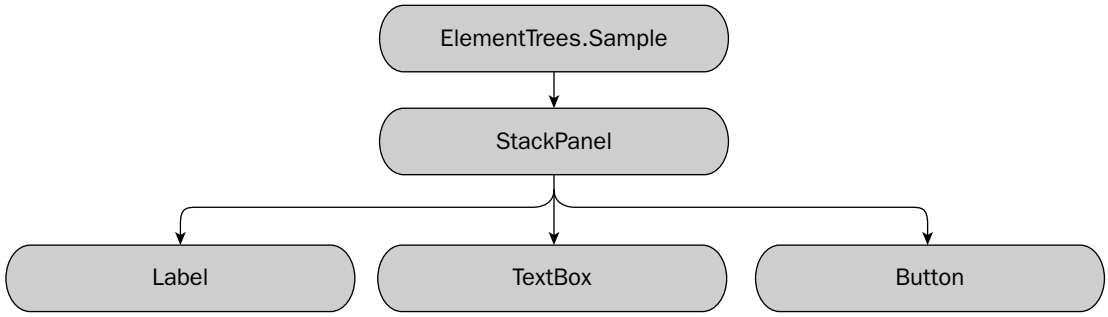

Figure 1-2

The logical tree outlines the 1:1 mapping of the nested XAML elements declared in the code snippet to their appropriate classes in the WPF API. The top-level element is `ElementTrees.Sample`, which derives from `System.Windows.Window`, the top-most, non-navigable container for a WPF application.

The `Window` class itself derives from `System.Windows.Controls.ContentControl`, which is the base class for all controls containing a single piece of content within WPF. To provide support for context-aware layout, the `System.Windows.Controls.Panel` class is available and includes functionality to arrange and lay out elements in WPF. The next element in the logical tree for the preceding code snippet is `System.Windows.Controls.StackPanel`, which derives from `Panel`. `StackPanel` arranges all client elements of the control in a single line, oriented either horizontally or vertically. This control behaves similarly to an HTML table with a single column or row, depending on orientation.

Contained within the `StackPanel` instance are the controls visible to the user: `System.Windows.Controls.TextBox`, `System.Windows.Controls.Button`, and `System.Windows.Controls.Label`. Although ubiquitous, these controls are unique implementations created from the ground up for WPF. Although they may behave in much the same way as their counterparts in WinForms from a user's perspective, their composition and behavior within the programming model and the visual tree are quite different.

Visual Tree

As its name implies, the logical tree makes sense — each element in the application is represented by a corresponding instance created by WPF. Under the hood, however, much more is taking place.

For each element in the logical tree, additional elements may be created to represent visual aspects required by the element. The combination of application elements and the elements created for visualization support is known as the *visual tree*. Considering the previous code snippet, the visual tree shown in Figure 1-3 would result.

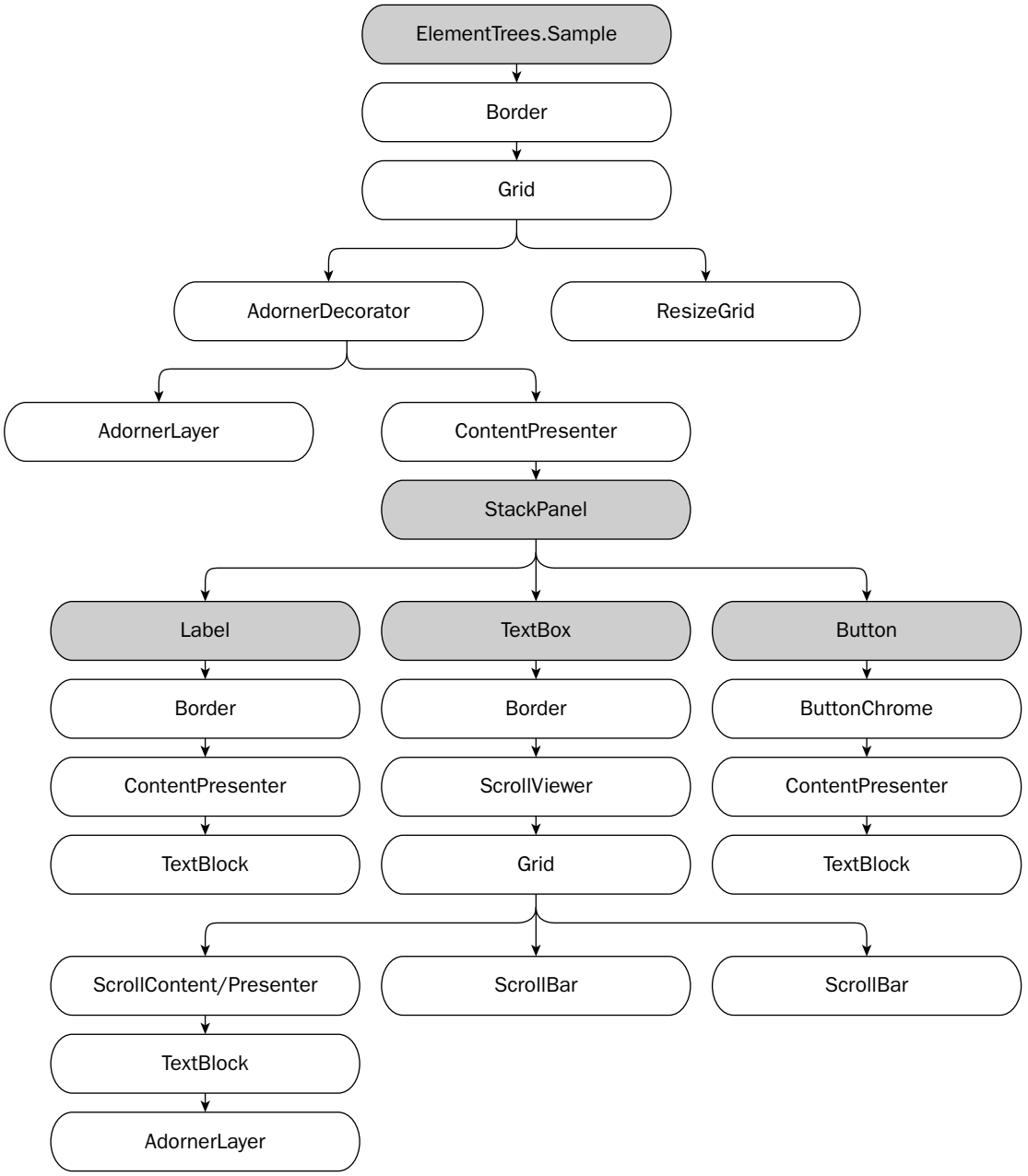
ElementTrees.Sample
Border
Grid
AdornerDecorator
ResizeGrid
AdornerLayer
ContentPresenter
StackPanel
Label
TextBox
Button
Border
Border
ButtonChrome
ContentPresenter
ScrollViewer
ContentPresenter
TextBlock
Grid
TextBlock
ScrollContent/Presenter
ScrollBar
ScrollBar
TextBlock
AdornerLayer

Figure 1-3

Okay, take a moment to catch your breath. WPF does a lot of work under the hood to support the flexible visual model implemented therein. In addition to the elements of the application included in the logical tree, 20 additional elements are created to support visualization. The following are some of these elements created for the `ElementTrees.Sample` element:

- `System.Windows.Controls.Border` — This control will draw a border and/or background surrounding its nested elements. The Border control is commonly found in many default controls, such as `System.Windows.Controls.Canvas`, `System.Windows.Controls.Label`, and `System.Windows.Controls.TextBox`.
- `System.Windows.Controls.Grid` — Similar to an HTML table, the Grid control provides a flexible area for content layout and positioning using rows and columns. Note, however, that there is a distinct Table control, `System.Windows.Documents.Table`, which provides block-level content flow also based on rows and columns and derives from a different class than `Grid`, which derives from `System.Windows.Controls.Panel`.
- `System.Windows.Documents.AdornerDecorator` — This control is used for styling of the control to which the element belongs. Adorners are discussed further in Chapter 10.
    - `System.Windows.Documents.AdornerLayer` — The AdornerLayer control represents the surface for rendering adorners.
    - `System.Windows.Controls.ContentPresenter` — This control is used to specify the location in the visual tree wherein content should be loaded for a specific control. In this case, the ContentPresenter control will specify where the contents of `ElementTrees.Sample` will be located.
- `System.Windows.Controls.Primitives.ResizeGrip` — This control enables the `Window` to have a resize grip within the visual tree. Thus, interaction with the host window of the WPF application will be captured by ResizeGrip such that the application can respond accordingly (for example, layout changes with a new surface area caused by the expansion or contraction of a host window).

Although many more elements have been created in the visual tree for this example, the preceding list should provide a basic understanding of their purpose and importance.

> *Although you can access both the logical and visual trees from code, you will spend most of your time developing WPF applications using the logical tree to interact with controls, their properties, methods, and events.*

Visual System

The *visual system* is the subsystem through which applications access the core presentation services available through WPF. This subsystem examines the components within an application (labels, buttons, text, 2D and 3D graphics, animations) and will communicate with the underlying composition system (via the message transport) to generate the rendered result to the screen.

Although the concept of the visual system is key to the architecture of WPF, much of the heavy lifting is done in the underlying composition system.

Font System

The font system was completely rewritten for WPF to provide a superior font and text engine over that of the systems previously available in Windows. The two font engines available in Windows today, GDI and Uniscribe, have significant drawbacks that do not make them suitable for WPF.

The new font system provides a unique mechanism for creating and caching information about fonts in WPF, including TrueType and Adobe OpenType fonts. The metrics, glyphs, and paths that make up a particular typeface are calculated by the font system, cached, and made available for use by WPF. This process is expensive, and using this caching technique significantly improves the performance of text layout and rendering in WPF.

> *Note that the font system is the only managed component outlined in Figure 1-1 that runs out-of-process, and communicates directly with WPF through interprocess communication to share font data from the cache.*

As outlined in Figure 1-1, the font system interacts with two main subsystems — the element system and the visual system — each for a different purpose:

- The font system interacts with the element system's Page and Table Service (PTS). PTS is responsible for the organization and layout of text within the UI. This includes paragraphs, tables, and blocks of text.
- The visual system leverages the font system for text layout services within a single line of text, such as kerning, leading, and spacing.

Input/Event System

The input/event system in WPF introduces significant advancements for input and user interaction over that of previous systems available in Windows, such as Win32. Messages evoked by Win32 for device-based user input, such as `WM_*` messages, provide a verbose mechanism for input and lack enhanced support for modern devices, such as a stylus.

As such, WPF sports a succinct input/event system that provides streamlined integration of user input with the visual tree. Through commands, WPF provides an advanced mechanism with which an application can discover and respond to user input.

Consider the following chain of events:

1. The user clicks his or her mouse.
2. User32 receives the device input message.
3. WPF receives the raw message and converts it into an input report (a WPF wrapper surrounding device input).
4. The application distinguishes the type of input report received from WPF and performs structural and geometric hit tests to determine the click target.
5. After validating the click target, the input report is converted into one or more events. For each event discovered, an event route is created, based on the composition of the hit area.
6. Each discovered event is raised by the input/event system.

As just outlined, WPF bears the responsibility of receiving the device input gesture and determining the appropriate course of action based on the type of input received and events exposed by elements within an application. Furthermore, WPF supports routed events, which allow for elements within the visual tree to listen and respond to events of parent and nested elements through tunneling and bubbling. Event routing is an important concept to understand when developing WPF applications. To learn more, see Chapter 2.

When implementing a pure delegate model for user input and device interaction, consider a number of issues. Given the complexity of the visual tree, implementing instance-based delegation through routed events can incur serious performance and storage overhead. As such, WPF implements class handlers as part of the input/event system. Class handlers provide static event listeners for any instance to respond to user input, such as text being typed on a keyboard.

Property System

The property system is integral to core data-related functions within WPF. It comprises the following three main components:

- Change notification
- Storage
- Expressions

Change Notification

The change notification system is the basis for many aspects of WPF. All data within WPF is exposed through properties; as such, changes to these properties are integral to interaction with the elements within a WPF application.

In .NET 2.0, change notification is achieved by implementing the `System.ComponentModel.INotifyPropertyChanged` interface. This interface defines only one member, the `PropertyChanged` event, which will notify listeners when a property of the implementing class is changed. Although this mechanism works well, it leaves much to be desired. When implementing this interface, it is the responsibility of the derived class to set up the necessary plumbing required to raise the `PropertyChanged` event for each property to be monitored. The following code snippet depicts a typical implementation of this interface:

```
public class MyTextBox : System.ComponentModel.INotifyPropertyChanged
{
  //
  // Private storage for Text property
  //
  private string _text;

  //
  // Change-aware property
  //
  public string Text
  {
    get { return _text; }
    set
    {
```

```
        _text = value;
        OnPropertyChanged("Text");
      }
    }

    //
    // INotifyPropertyChanged.PropertyChanged
    //
    public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

    //
    // Method to raise PropertyChanged event
    //
    private void OnPropertyChanged(string propertyName)
    {
      if (PropertyChanged != null)
      {
        PropertyChanged(
            this,
            new System.ComponentModel.PropertyChangedEventArgs(propertyName));
      }
    }
  }
```

As evident in this snippet, there are several common areas of plumbing to be inserted into any class wanting to implement this change notification mechanism. Although this example may seem trivial, consider the scale of change notification if this object implemented 40 change-aware methods. Moreover, consider the situation in which a property's value is set to the same value currently held. If your requirements distinguished this to be a non-change, then additional logic would need to be included in each property to first test the equality of the property's current and specified changed values and raise the `PropertyChanged` event only when these values were different.

Enter the notion of dependency properties in WPF. *Dependency properties* represent properties registered with the WPF property system that implement value expressions, styling, data binding, change notification, and more. Considering the preceding code snippet, the following revised snippet depicts the same functionality using a dependency property:

```
  public class MyTextBox : DependencyObject
  {
    //
    // Dependency property for Text
    //
    public static readonly DependencyProperty TextProperty =
      DependencyProperty.Register(
          "Text",
           typeof(string),
           typeof(MyTextBox));

    //
    // Change-aware property
    //
    public string Text
    {
```

```
      get { return base.GetValue(TextProperty) as string; }
      set { base.SetValue(TextProperty, value); }
    }
  }
```

Immediately apparent is the sheer reduction in code required to implement a dependency property. Let's dissect this to understand the difference between this snippet and the previous listing.

For the `Text` property of the `MyTextBox` class, we've removed the private `_text` variable used to store the value for that instance, and we've replaced the `get` and `set` functions with calls to methods included in the base class, `DependencyObject`. The `GetValue` and `SetValue` methods work within the dependency property system to store and retrieve property values, and trigger notification when such properties change. In order for this class to participate in the dependency property system, it must define static `DependencyProperty` members for each property to be included. In the case of the `Text` property, a `TextProperty` member was created and registered to the `Text` property using the `DependencyProperty.Register` method. This method creates an entry in the dependency property system for the `MyTextBox` class and will be responsible for change notification.

Storage

As noted earlier, properties are the main data element of WPF. Properties provide the foundation upon which a declarative model can be supported, and enable many of the key features included in WPF, such as styling, data binding, animation, and more.

This extensive coupling to properties is not without drawbacks, however. Supporting extensive properties within WPF elements can create a massive storage overhead because each element can contain many properties. Using instance-based property storage, WPF elements would be too bloated to support the ambitious goals for runtime performance and visual appeal. Thus, the dependency property system, as mentioned previously, provides support for streamlined property storage. Leveraging helper methods, classes can easily access properties and partake in change notification, data binding, styling, and other features available in this system.

Expressions

Expressions within WPF provide the extensibility required to successfully implement a declarative model. Features of WPF, such as styling, inheritance, and data binding, rely on expressions to dynamically evaluate and move data around the system.

Consider the following code snippet:

```
<Window x:Class="XamlExpressions.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title="XamlExpressions" Height="300" Width="300"

  <Window.Resources>
    <Style TargetType="{x:Type Button}">
      <Setter Property="FontFamily" Value="Segoe Black" />
      <Setter Property="FontSize" Value="12pt" />
      <Setter Property="Foreground" Value="#777777" />
      <Setter Property="HorizontalAlignment" Value="Center" />
    </Style>
```

```
    </Window.Resources>
    <Grid>
      <Grid.RowDefinitions>
        <RowDefinition Height="50px" />
        <RowDefinition Height="*" />
      </Grid.RowDefinitions>
      <Button Grid.Column="0" Grid.Row="0">Click Me!</Button>
    </Grid>
  </Window>
```

In this example, we've created a rather simple WPF application containing a `Grid` and `Button` control. This should not be new, as we've touched on these controls and seen their declaration earlier in this chapter. However, points of interest are the elements under the `Window.Resources` element.

The `Style` element provides the mechanism for creating custom styles for controls within WPF. If you use expressions, styles can be applied to elements within a WPF application in one of two ways: through the `TargetType` attribute or through a unique style identified and stored in the `x:Key` attribute of the `Style` element. The previous listing demonstrated the former method by assigning an expression distinguishing the type of element to which this style should apply.

> *Although we cover styling later in this book, it's important to note here that you can apply styles to elements within a WPF application through the use of expressions.*

Expressions are not limited to styling; they are used extensively throughout WPF. The concept of expressions is embedded in almost every subsequent chapter of this book, and we'll dive deeper as needed.

Message Transport System

The message transport service is a key component of the WPF architecture that ties the visual system to the composition system. As mentioned previously, the visual system provides a managed interface through which all other managed subsystems within WPF provide instructions on what elements need to be represented on the screen. Although the visual system provides this hook, it doesn't perform the work itself; rather, it offloads such tasks to the composition system.

For such offloading to take place, the message transport system provides communication channels between the visual system and the composition system. These channels implement a .NET Remoting protocol that creates an efficient mechanism through which data structures can be passed. Moreover, leveraging a Remoting protocol allows WPF to transfer data structures across boundaries, thus providing the possibility of offloading the rendering of UI elements to an entirely different machine. This concept is very compelling for terminal client scenarios, such as Remote Desktop or Citrix MetaFrame, where graphics processing of applications executing remotely can be offloaded from the server to the terminal client.

Composition System

As we have alluded to throughout this section, the composition system really provides the internal combustion of the WPF architecture. The composition system is an unmanaged subsystem that receives instructions from the managed visual system and turns such instructions into graphics that appear on the screen.

The composition system provides the functionality to visualize the seemingly nebulous elements within a WPF application. This includes the functionality to calculate pixels for a particular element, and generates triangle geometries for visual elements, which, in turn, are sent directly to the GPU for output to the display. This process of hardware-based rendering — through Direct3D — frees up the CPU to perform other operations, improving the overall success of WPF applications. Moreover, the quality of onscreen graphics is significantly higher through the use of hardware-based rendering.

Although WPF has been built to leverage the GPU for hardware-based rendering, software-based rendering is available as a fallback.

XAML

Now that you have a good understanding of the underlying plumbing of WPF, let's take a closer look at XAML. As noted earlier in this chapter, XAML is the new declarative language for use in developing WPF applications. As a declarative language, XAML introduces significant flexibility in the way WPF applications can be developed. Such flexibility is examined in the following section.

Declarative vs. Imperative

The recent history of developing a UI for a typical Windows-based application encompasses the following general tasks (in no particular):

- **Layout UI** — Define and set up controls on a form, typically with the aid of a visual designer (for example, Visual Basic 6 or Visual Studio 2005).
- **Create bindings** — Hook data-bound controls to data sources, typically a database or related data access objects.
- **Wire up user interaction** — Configure control events so that user interaction triggers application logic to manipulate data and/or the UI.

Although overly simplistic, this covers most of the things that must occur to develop a UI. What's important to realize is that almost all of the plumbing required to achieve the tasks in the preceding list is imperative — that is, the definition of controls, their binding, and user interaction is included as statements within code. These imperative statements modify the state of the application, and are typically compiled directly into an executable format. For example, the WinForms Visual Design in Visual Studio 2005 generates code for every control dragged onto the design surface, which in turn is compiled with the rest of the application code into an executable .NET assembly.

So what's wrong with this approach? In fact, nothing is wrong with this approach. Although XAML provides a declarative model for developing WPF applications, you can achieve any task possible in XAML purely in code. However, there are benefits of the declarative model that make XAML such a compelling solution.

To see one benefit of leveraging the declarative model of XAML, consider the following code snippet:

```
<Window x:Class="XamlXample.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  Title=" XamlXample" Height="300" Width="300"
  >
  <Grid>
    <Grid.RowDefinitions>
      <RowDefinition Height="50px" />
      <RowDefinition Height="*" />
    </Grid.RowDefinitions>
    <Label Grid.Column="0" Grid.Row="0">Click This Button:</Label>
    <Button Grid.Column="0" Grid.Row="1"
        DockPanel.Dock="Top" Name="clickButton">Click Me!</Button>
  </Grid>
</Window>
```

Although this is yet another simple example, this code snippet depicts a very simple example of the straightforward approach implemented by XAML. In this example, we've defined a `Window` element containing a Grid control for content containment. The Grid defines some properties about itself and contains two simple controls: a Label and TextBox. Seems simple enough, right?

Now, let's examine the code necessary to generate the same elements as outlined in the preceding XAML code snippet:

```
class ImperativeWindow : Window
{
  public ImperativeWindow()
  {
    InitializeComponent();
  }

  private void InitializeComponent()
  {
    this.Title = "XamlXample";
    this.Height = 300;
    this.Width = 300;

    Grid grid = new Grid();

    RowDefinition row1 = new RowDefinition();
    row1.Height = new GridLength(50);
    grid.RowDefinitions.Add(row1);

    RowDefinition row2 = new RowDefinition();
    row2.Height = GridLength.Auto;
    grid.RowDefinitions.Add(row2);

    Label label = new Label();
    label.Content = "Click This Button:";

    grid.Children.Add(label);
```

```
        Grid.SetColumn(label, 0);
        Grid.SetRow(label, 0);

        Button button = new Button();
        button.Name = "clickButton";
        button.Content = "Click Me!";

        DockPanel.SetDock(button, Dock.Top);

        grid.Children.Add(button);
        Grid.SetColumn(button, 0);
        Grid.SetRow(button, 1);
      }
    }
```

As you can see, creating the exact same application using code requires a significantly greater amount of statements than its XAML counterpart. It is important to note that under the hood, WPF will actually create a source file at compile time for each XAML file in your application. The code that's generated will be very similar to the preceding code; however, the source of your application remains as a XAML file.

XAML Runtime Support

The benefit we've covered thus far pertains simply to preference and convenience. We have not yet dived into some of the additional features of a declarative model. So far, we've discussed XAML as a language that's interpreted at compile time and, when combined with code-behind files and other resources, is included in a distributable executable. In addition to this method, WPF provides runtime support for dynamic parsing of XAML files. This exciting feature of WPF creates many new possibilities for applications.

To further appreciate the power of a dynamic declarative model, consider the following scenario.

> **You've been tasked with developing a kiosk-style application for your customer, a global record store chain. This application will allow their customers to browse the entire catalog of music available throughout all stores, listen to song previews, view album artwork and artist photographs, and even watch video clips. The look and feel for this application must be avant-garde, and the application must include animation and sync up with the visual identity of this edgy corporate brand. Moreover, the style and layout of the application must be dynamic so that each store can present a localized interface on a seasonal basis.**

This scenario provides a not-so-extreme example of what many companies might seek within a consumer-facing application. While many of the requirements make for a great case to use WPF (animation, images, audio, and video integration), a few stand out.

Dynamic styling and layout for localized applications is a very non-trivial feat accomplished only by the most skilled developers. To achieve this using current technology, developers have limited support out-of-the-box, and have to rely on third-party and custom solutions to create dynamic interfaces. On the

contrary, with WPF, developers are empowered with foundational support to create declarative UIs that can be interpreted dynamically. Moreover, enhanced support for localization in WPF makes developing global applications much easier to develop.

Designer and Developer Collaboration

In the example scenario, the customer desires a visually aesthetic application that will wow customers and represent the avant-garde visual identity of their edgy corporate brand. Although some developers might consider themselves astute graphic designers, odds are you're more focused on function than form.

With current technologies, developing such an application would require a large team of developers to build out services available out-of-the-box with WPF, such as digital media integration and animation. Moreover, the visual design of the application would be limited to such developed services and would most likely be an adaptation of design compositions provided by a graphic designer. This doesn't provide the customer with a best-of-breed solution and is not only quite limited to the services available, but is also constrained by time to integrate the visual design and the translation of the designers' vision into the application by the development team.

Thankfully, this problem has, in fact, been solved with WPF. Through the use of XAML, designers and developers have a unified platform on which visual design and application logic can be seamlessly integrated. Visual designers available for XAML, including Microsoft Expression Blend, further support this notion by providing designers with a familiar interface for design, typesetting, and illustration, all while generating XAML under the hood. The resultant XAML can then be handed off to a developer for integration with business logic and other application-specific features. Never has the designer and developer experience been so tightly integrated.

Visual Design Tools

As mentioned throughout this chapter, several visual design tools are available for WPF. These tools can provide a familiar interface for both designers and developers to create WPF applications. Moreover, these tools support the notion of designer and developer collaboration.

XamlPad

Provided with the Windows SDK, XamlPad is a basic visual editor for XAML. Similar to the ubiquitous Notepad, XamlPad will be used by almost all WPF developers for quick and easy editing of XAML files. XamlPad provides real-time visualization of XAML code, allowing developers to author, test, and validate XAML code on-the-fly. Furthermore, XamlPad features XAML syntax validation, making it easy to troubleshoot errors in a XAML file.

Figure 1-4 illustrates the XamlPad user interface.

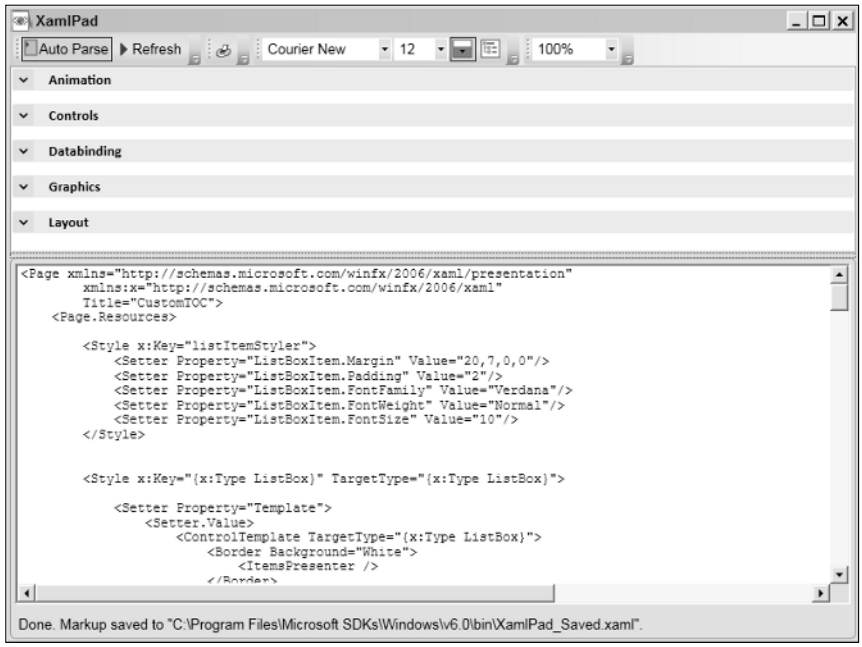

Figure 1-4

Microsoft Expression Blend

As illustrated in Figure 1-5, Microsoft Expression Blend (formerly "Expression Interactive Designer") is Microsoft's flagship tool for designing WPF applications. A full-featured design tool, Expression Blend provides a truly usable authoring environment for designers to create sophisticated designs for WPF applications.

Some of the key features in Expression Blend include:

- Vector drawing tools
- Timeline-based animation support, media integration, and 3D authoring
- Robust integration with data sources and seamless integration with Visual Studio 2005

For a detailed discussion of Expression Blend, please see Chapters 4 and 5.

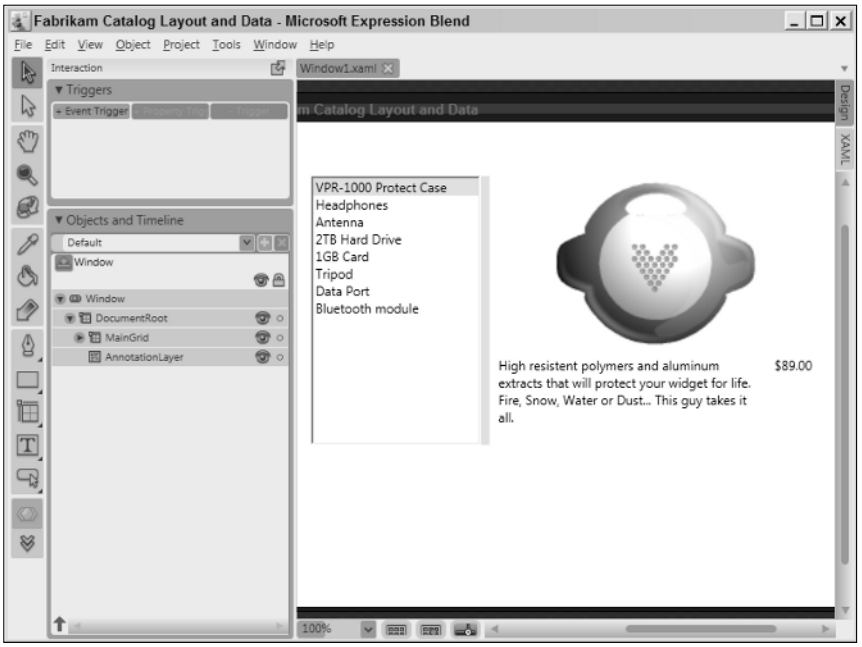

Figure 1-5

Visual Designer for Windows Presentation Foundation

Code-named *Cider*, the visual designer for WPF provides a visual authoring environment for WPF applications in Visual Studio 2005. As illustrated in Figure 1-6, Cider provides a design experience familiar to those who have developed WinForms applications using Visual Studio 2005.

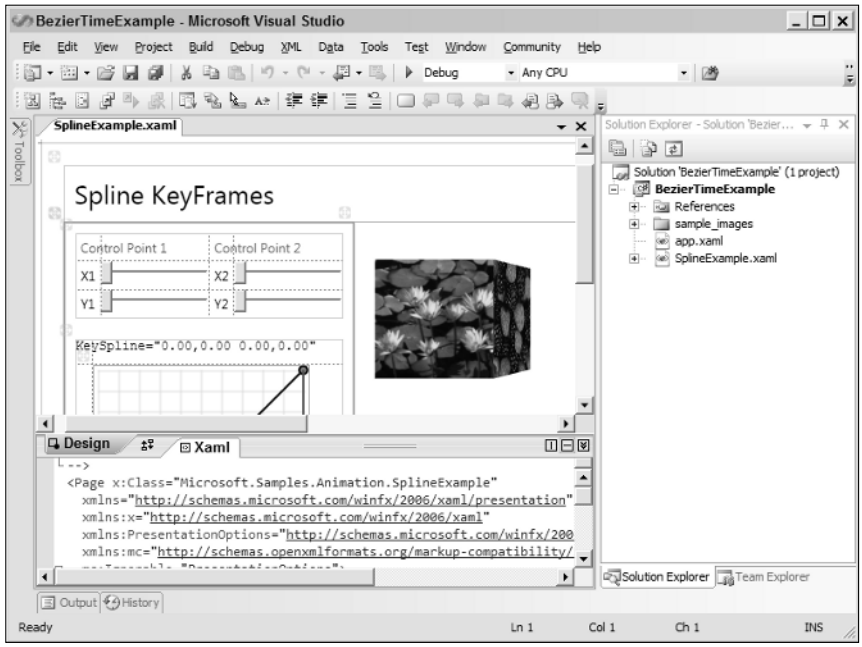

Figure 1-6

Electric Rain ZAM 3D

From Electric Rain (`www.erain.com`), the creators of the excellent Flash 3D plug-in Swift3D, comes ZAM 3D, a 3D modeling and animation application based on WPF. Touted as a developer productivity tool, ZAM 3D offers a robust environment for modeling and animating 3D graphics, while automatically generating the XAML markup representing such models.

If have experience with Swift3D (or any other 3D authoring environment, such as 3D Studio Max), you'll find the UI very familiar. It features dual viewports, which provide instant gratification for animations, as well as model and environment effects. Under the hood, ZAM 3D creates XAML files containing the `ViewPort3D` element to draw and animate models and scenes created within the tool. One really nifty feature is the Copy XAML command, which will generate the appropriate XAML for your current model and place this text on the clipboard. This is extremely handy if you want to test your model in XamlPad.

Figure 1-7 depicts the ZAM 3D user interface.

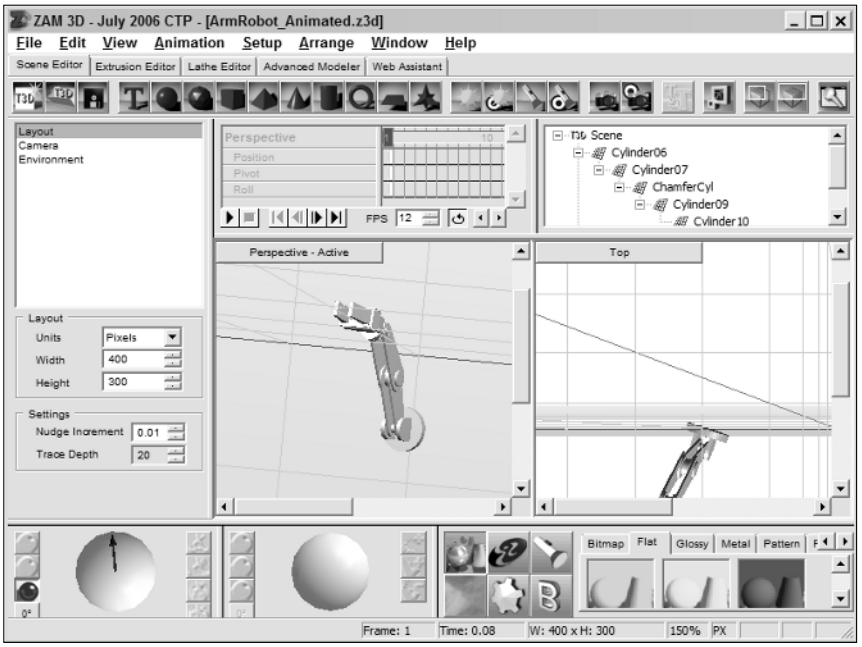

Figure 1-7

Mobiform Aurora

At the forefront of WPF, Mobiform Software, Inc. (`www.mobiform.com`) has created a very compelling visual designer: Aurora. Aurora is itself a WPF application that provides a design experience similar to EID, generating XAML documents. Documents created in Aurora are pure XAML and can be used interchangeably with other designers, including Visual Studio 2005 and EID.

In addition to providing a design experience for functionality available in WPF, Aurora includes custom controls developed by Mobiform for use your applications, such as Pie Chart and File Control.

Figure 1-8 illustrates Aurora's clean WPF-based user interface.

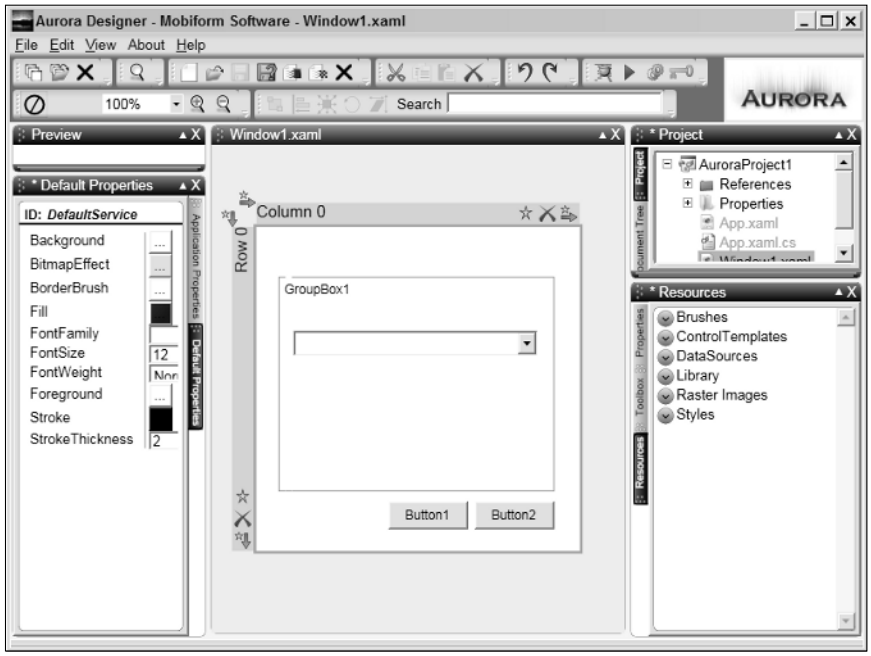

Figure 1-8

Summary

In this chapter, we've merely peeked into the exciting new world of application development with WPF. We started by reviewing the evolution of the Windows platform and how WPF empowers developers to create Windows- and web-based applications with a superior User Experience. This chapter also outlined some key areas of the WPF architecture, including:

- Element system and element trees (visual and logical)
- Visual system
- Font system
- Input/event system
- Property system (change notification, storage, and expressions)
- Message transport system
- Composition system

You learned about the new declarative language of WPF and XAML, and how this language introduces greater flexibility in application development and deployment. You also learned that the XAML provides a unified platform for design and development collaboration.

Finally, you caught a glimpse of the visual design tools with which WPF applications can be developed, including XamlPad and Microsoft Expression Blend.

The following chapters expand on what you've learned and provide a detailed look at .NET programming with WPF.