THE EXPERT'S VOICE® IN .NET

FOURTH EDITION

Pro WPF 4.5 in C#

Windows Presentation Foundation in .NET 4.5

Matthew MacDonald

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

PART I

Fundamentals

CHAPTER 1

Introducing WPF

The Windows Presentation Foundation (WPF) is a modern graphical display system for Windows. It's a radical change from the technologies that came before it, with innovative features such as built-in hardware acceleration and resolution independence, both of which you'll explore in this chapter.

WPF is the best toolkit to use if you want to build a rich desktop application that runs on Windows Vista, Windows 7, and Windows 8 in desktop mode (as well as the corresponding versions of Windows Server). In fact, it's the *only* general-purpose toolkit that targets these versions of Windows. By comparison, Microsoft's new Metro toolkit—although exciting—is limited to Windows 8 systems only. (WPF applications can even be made to run on ancient Windows XP computers, which are still found in many businesses. The only limitation is that you must configure Visual Studio to target the slightly older .NET 4.0 Framework, rather than .NET 4.5.)

In this chapter, you'll take your first look at the architecture of WPF. You'll learn how it deals with varying screen resolutions, and you'll get a high-level survey of its core assemblies and classes. You'll also consider how WPF has evolved from its initial release to version 4.5.

The Evolution of Windows Graphics

Before WPF, Windows developers spent nearly 15 years using essentially the same display technology. That's because every traditional, pre-WPF Windows application relies on two well-worn parts of the Windows operating system to create its user interface:

- *User32*: This provides the traditional Windows look and feel for elements such as windows, buttons, text boxes, and so on.
- *GDI/GDI+*: This provides drawing support for rendering shapes, text, and images at the cost of additional complexity (and often lackluster performance).

Over the years, both technologies have been refined, and the APIs that developers use to interact with them have changed dramatically. But whether you're crafting an application with .NET and Windows Forms or even Visual Basic 6 or MFC-based C++ code, behind the scenes the same parts of the Windows operating system are at work. Different frameworks simply provide different wrappers for interacting with User32 and GDI/GDI+. They can provide improvements in efficiency, reduce complexity, and add prebaked features so you don't have to code them yourself; but they can't remove the fundamental limitations of a system component that was designed more than a decade ago.

■ **Note** The basic division of labor between User32 and GDI/GDI+ was introduced more than 15 years ago and was well established in Windows 3.0. Of course, User32 was simply User at that point, because software hadn't yet entered the 32-bit world.

DirectX: The New Graphics Engine

Microsoft created one way around the limitations of the User32 and GDI/GDI+ libraries: *DirectX*. DirectX began as a cobbled-together, error-prone toolkit for creating games on the Windows platform. Its design mandate was speed, and so Microsoft worked closely with video card vendors to give DirectX the hardware acceleration needed for complex textures, special effects such as partial transparency, and three-dimensional graphics.

Over the years since it was first introduced (shortly after Windows 95), DirectX has matured. It's now an integral part of Windows, with support for all modern video cards. However, the programming API for DirectX still reflects its roots as a game developer's toolkit. Because of its raw complexity, DirectX is almost never used in traditional types of Windows applications (such as business software).

WPF changes all this. In WPF, the underlying graphics technology isn't GDI/GDI+. Instead, it's DirectX. In fact, WPF applications use DirectX no matter what type of user interface you create. That means that whether you're designing complex three-dimensional graphics (DirectX's forte) or just drawing buttons and plain text, all the drawing work travels through the DirectX pipeline. As a result, even the most mundane business applications can use rich effects such as transparency and anti-aliasing. You also benefit from hardware acceleration, which simply means DirectX hands off as much work as possible to the graphics processing unit (GPU), which is the dedicated processor on the video card.

■ **Note** DirectX is more efficient because it understands higher-level ingredients such as textures and gradients that can be rendered directly by the video card. GDI/GDI+ doesn't, so it needs to convert them to pixel-by-pixel instructions, which are rendered much more slowly by modern video cards.

One component that's still in the picture (to a limited extent) is User32. That's because WPF still relies on User32 for certain services, such as handling and routing input and sorting out which application owns which portion of screen real estate. However, all the drawing is funneled through DirectX.

Hardware Acceleration and WPF

Video cards differ in their support for specialized rendering features and optimizations. Fortunately, this isn't a problem, for two reasons. First, most modern computers have video hardware that's more than powerful enough for WPF features such as 3-D drawing and animation. This is true even of laptops and desktop computers with integrated graphics (graphics processors that are built in to the motherboard, rather than on a separate card). Second, WPF has a software fallback for everything it does. That means WPF is intelligent enough to use hardware optimizations where possible, but can perform the same work using software calculations if necessary. So if you run a WPF application on a computer with a legacy video card, the interface will still appear the way you designed it. Of course, the software alternative may be much slower, so you'll find that computers with older video cards won't run rich WPF applications very well, especially ones that incorporate complex animations or other intense graphical effects.

WPF: A Higher-Level API

If the only thing WPF offered was hardware acceleration through DirectX, it would be a compelling improvement but a limited one. But WPF includes a basket of high-level services designed for application programmers.

The following are some of the most dramatic changes that WPF ushers into the Windows programming world:

- *A web-like layout model:* Rather than fix controls in place with specific coordinates, WPF emphasizes flexible flow layout that arranges controls based on their content. The result is a user interface that can adapt to show highly dynamic content or different languages.
- *A rich drawing model:* Rather than painting pixels, in WPF you deal with *primitives*—basic shapes, blocks of text, and other graphical ingredients. You also have new features, such as true transparent controls, the ability to stack multiple layers with different opacities, and native 3-D support.
- *A rich text model:* WPF gives Windows applications the ability to display rich, styled text anywhere in a user interface. You can even combine text with lists, floating figures, and other user interface elements. And if you need to display large amounts of text, you can use advanced document display features such as wrapping, columns, and justification to improve readability.
- *Animation as a first-class programming concept*: In WPF, there's no need to use a timer to force a form to repaint itself. Instead, animation is an intrinsic part of the framework. You define animations with declarative tags, and WPF puts them into action automatically.
- *Support for audio and video media:* Previous user interface toolkits, such as Windows Forms, were surprisingly limited when dealing with multimedia. But WPF includes support for playing any audio or video file supported by Windows Media Player, and it allows you to play more than one media file at once. Even more impressively, it gives you the tools to integrate video content into the rest of your user interface, allowing you to pull off exotic tricks such as placing a video window on a spinning 3-D cube.
- *Styles and templates:* Styles allow you to standardize formatting and reuse it throughout your application. Templates allow you to change the way any element is rendered, even a core control such as the button. It has never been easier to build modern skinned interfaces.
- *Commands:* Most users realize that it doesn't matter whether they trigger the Open command through a menu or through a toolbar; the end result is the same. Now that abstraction is available to your code, you can define an application command in one place and link it to multiple controls.
- *Declarative user interface:* Although you can construct a WPF window with code, Visual Studio takes a different approach. It serializes each window's content to a set of XML tags in a XAML document. The advantage is that your user interface is completely separated from your code, and graphic designers can use professional tools to edit your XAML files and refine your application's front end. (*XAML* is short for Extensible Application Markup Language, and it's described in detail in Chapter 2.)

- *Page-based applications:* Using WPF, you can build a browser-like application that lets you move through a collection of pages, complete with forward and back navigation buttons. WPF handles the messy details such as the page history. You can even deploy your project as a browser-based application that runs right inside Internet Explorer.

Resolution Independence

Traditional Windows applications are bound by certain assumptions about resolution. Developers usually assume a standard monitor resolution (such as 1366 × 768 pixels), design their windows with that in mind, and try to ensure reasonable resizing behavior for smaller and larger dimensions.

The problem is that the user interface in traditional Windows applications isn't scalable. As a result, if you use a high monitor resolution that crams in pixels more densely, your application windows become smaller and more difficult to read. This is particularly a problem with newer monitors that have high pixel densities and run at correspondingly high resolutions. For example, it's common to find consumer monitors (particularly on laptops) that have pixel densities of 120 dpi or 144 dpi (dots per inch), rather than the more traditional 96 dpi. At their native resolution, these displays pack the pixels in much more tightly, creating eye-squintingly small controls and text.

Ideally, applications would use higher pixel densities to show more detail. For example, a high-resolution monitor could display similarly sized toolbar icons but use the extra pixels to render sharper graphics. That way, you could keep the same basic layout but offer increased clarity and detail. For a variety of reasons, this solution hasn't been possible in the past. Although you can resize graphical content that's drawn with GDI/GDI+, User32 (which generates the visuals for common controls) doesn't support true scaling.

WPF doesn't suffer from this problem because it renders all user interface elements itself, from simple shapes to common controls such as buttons. As a result, if you create a button that's 1 inch wide on your computer monitor, it can remain 1 inch wide on a high-resolution monitor—WPF will simply render it in greater detail and with more pixels.

This is the big picture, but it glosses over a few details. Most importantly, you need to realize that WPF bases its scaling on the *system* DPI setting, not the DPI of your physical display device. This makes perfect sense—after all, if you're displaying your application on a 100-inch projector, you're probably standing several feet back and expecting to see a jumbo-size version of your windows. You don't want WPF to suddenly scale down your application to "normal" size. Similarly, if you're using a laptop with a high-resolution display, you probably expect to have slightly smaller windows—it's the price you pay to fit all your information onto a smaller screen. Furthermore, different users have different preferences. Some want richer detail, while others prefer to cram in more content.

So, how does WPF determine how big an application window *should* be? The short answer is that WPF uses the system DPI setting when it calculates sizes. But to understand how this really works, it helps to take a closer look at the WPF measurement system.

WPF Units

A WPF window and all the elements inside it are measured using *device-independent units*. A single device-independent unit is defined as 1/96 of an inch. To understand what this means in practice, you'll need to consider an example.

Imagine that you create a small button in WPF that's 96 by 96 units in size. If you're using the standard Windows DPI setting (96 dpi), each device-independent unit corresponds to one real, physical pixel. That's because WPF uses this calculation:

```
[Physical Unit Size] = [Device-Independent Unit Size] x [System DPI]
                     = 1/96 inch x 96 dpi
                     = 1 pixel
```

Essentially, WPF assumes it takes 96 pixels to make an inch because Windows tells it that through the system DPI setting. However, the reality depends on your display device.

For example, consider a 19-inch LCD monitor with a maximum resolution of 1600 by 1200 pixels. Using a dash of Pythagoras, you can calculate the pixel density for this monitor, as shown here:

$$[\text{Screen DPI}] = \frac{\sqrt{1600^2 + 1200^2}\ \text{Pixels}}{19\ \text{inches}}$$

```
= 100 dpi
```

In this case, the pixel density works out to 100 dpi, which is slightly higher than what Windows assumes. As a result, on this monitor a 96-by-96-pixel button will be slightly smaller than 1 inch.

On the other hand, consider a 15-inch LCD monitor with a resolution of 1024 by 768. Here, the pixel density drops to about 85 dpi, so the 96-by-96-pixel button appears slightly *larger* than 1 inch.

In both these cases, if you reduce the screen size (say, by switching to 800 by 600 resolution), the button (and every other screen element) will appear proportionately larger. That's because the system DPI setting remains at 96 dpi. In other words, Windows continues to assume it takes 96 pixels to make an inch, even though at a lower resolution it takes far fewer pixels.

■ **Tip** As you no doubt know, LCD monitors are designed to work best at a specific resolution, which is called the native resolution. If you lower the resolution, the monitor must use interpolation to fill in the extra pixels, which can cause blurriness. To get the best display, it's always best to use the native resolution. If you want larger windows, buttons, and text, consider modifying the system DPI setting instead (as described next).

System DPI

So far, the WPF button example works exactly the same as any other user interface element in any other type of Windows application. The difference is the result if you change the system DPI setting. In the previous generation of Windows, this feature was sometimes called *large fonts*. That's because the system DPI affects the system font size but often leaves other details unchanged.

■ **Note** Many Windows applications don't fully support higher DPI settings. At worst, increasing the system DPI can result in windows that have some content that's scaled up and other content that isn't, which can lead to obscured content and even unusable windows.

This is where WPF is different. WPF respects the system DPI setting natively and effortlessly. For example, if you change the system DPI setting to 120 dpi (a common choice for users of large high-resolution screens), WPF assumes that it needs 120 pixels to fill an inch of space. WPF uses the following calculation to figure out how it should translate its logical units to physical device pixels:

```
[Physical Unit Size] = [Device-Independent Unit Size] x [System DPI]
                     = 1/96 inch x 120 dpi
                     = 1.25 pixels
```

In other words, when you set the system DPI to 120 dpi, the WPF rendering engine assumes one device-independent unit equals 1.25 pixels. If you show a 96-by-96 button, the physical size will actually be 120 by 120 pixels (because 96 x 1.25 = 120). This is the result you expect—a button that's 1 inch on a standard monitor remains 1 inch in size on a monitor with a higher pixel density.

This automatic scaling wouldn't help much if it applied only to buttons. But WPF uses device-independent units for everything it displays, including shapes, controls, text, and any other ingredient you put in a window. As a result, you can change the system DPI to whatever you want, and WPF adjusts the size of your application seamlessly.

■ **Note** Depending on the system DPI, the calculated pixel size may be a fractional value. You might assume that WPF simply rounds off your measurements to the nearest pixel. However, by default, WPF does something different. If an edge of an element falls between pixels, it uses anti-aliasing to blend that edge into the adjacent pixels. This might seem like an odd choice, but it actually makes a fair bit of sense. Your controls won't necessarily have straight, clearly defined edges if you use custom-drawn graphics to skin them; so some level of anti-aliasing is already necessary.

The steps for adjusting the system DPI depend on the operating system. The following sections explain what to do, depending on your operating system.

Windows Vista

1. Right-click your desktop and choose Personalize.
2. In the list of links on the left, choose Adjust Font Size (DPI).
3. Choose between 96 or 120 dpi. Or click Custom DPI to use a custom DPI setting. You can then specify a percentage value, as shown in Figure 1-1. (For example, 175% scales the standard 96 dpi to 168 dpi.) In addition, when using a custom DPI setting, you have an option named Use Windows XP Style DPI Scaling, which is described in the sidebar "DPI Scaling."

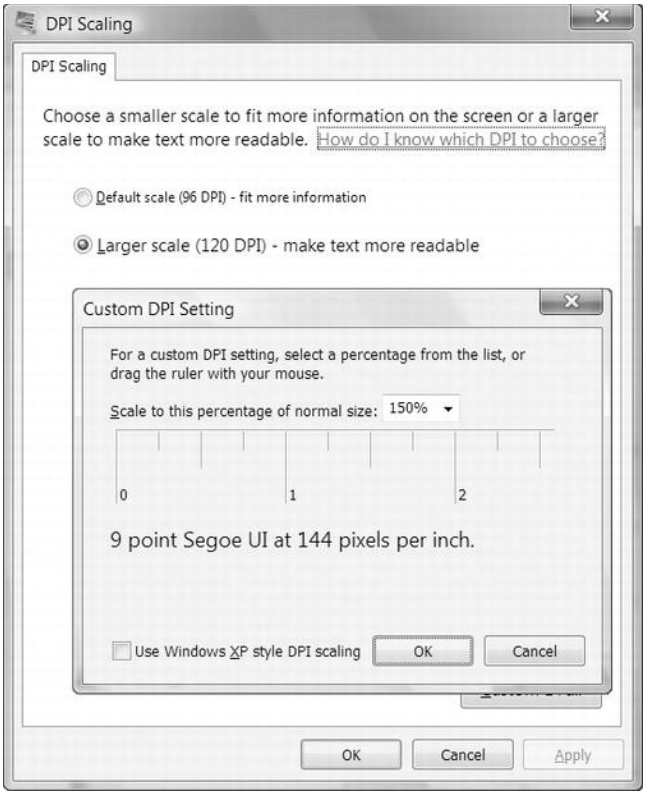

Figure 1-1. *Changing the system DPI*

Windows 7 and Windows 8

1. Right-click your desktop and choose Personalize.
2. In the list of links at the bottom-left of the window, choose Display.
3. Choose between Smaller (the default option), Medium, or Larger. Although these options are described by scaling percentages (100%, 125%, or 150%), they actually correspond to the DPI values 96, 120, and 144. You'll notice that the first two are the same standards found in Windows Vista and Windows XP, while the third one is larger still. Alternatively, you can click Set Custom Text Size to use a custom DPI percentage, as shown in Figure 1-1. (For example, 175% scales the standard 96 dpi to 168 dpi.) When using a custom DPI setting, you have an option named Use Windows XP Style DPI Scaling, which is described in the following sidebar.

DPI SCALING

Because older applications are notoriously lacking in their support for high DPI settings, Windows Vista introduced a technique called *bitmap scaling*. Later versions of Windows also support this feature.

With bitmap scaling, when you run an application that doesn't appear to support high DPI settings, Windows resizes it as though it were an image. The advantage of this approach is that the application still believes it's running at the standard 96 dpi. Windows seamlessly translates input (such as mouse clicks) and routes them to the right place in the application's "real" coordinate system.

The scaling algorithm that Windows uses is a fairly good one—it respects pixel boundaries to avoid blurry edges and uses the video card hardware where possible to increase speed—but it inevitably leads to a fuzzier display. It also has a serious limitation in that Windows can't recognize older applications that *do* support high DPI settings. That's because applications need to include a manifest or call SetProcessDPIAware (in User32) to advertise their high DPI support. Although WPF applications handle this step correctly, applications created prior to Windows Vista won't use either approach and will be stuck with bitmap scaling even when they support higher DPIs.

There are two possible solutions. If you have a few specific applications that support high DPI settings but don't indicate it, you can configure that detail manually. To do so, right-click the shortcut that starts the application (in the Start menu) and choose Properties. On the Compatibility tab, enable the option named Disable Display Scaling on High DPI Settings. If you have a lot of applications to configure, this gets tiring fast.

The other possible solution is to disable bitmap scaling altogether. To do so, choose the Use Windows XP Style DPI Scaling option in the Custom DPI Setting dialog box shown in Figure 1-1. The only limitation of this approach is there may be some applications that won't display properly (and possibly won't be usable) at high DPI settings. By default, Use Windows XP Style DPI Scaling is checked for DPI sizes of 120 or less but unchecked for DPI sizes that are greater.

Bitmap and Vector Graphics

When you work with ordinary controls, you can take WPF's resolution independence for granted. WPF takes care of making sure that everything has the right size automatically. However, if you plan to incorporate images into your application, you can't be quite as casual. For example, in traditional Windows applications, developers use tiny bitmaps for toolbar commands. In a WPF application, this approach is not ideal because the bitmap may display artifacts (becoming blurry) as it's scaled up or down according to the system DPI. Instead, when designing a WPF user interface, even the smallest icon is generally implemented as a vector graphic. *Vector graphics* are defined as a set of shapes, and as such they can be easily scaled to any size.

■ **Note** Of course, drawing a vector graphic takes more time than painting a basic bitmap, but WPF includes optimizations that are designed to lessen the overhead to ensure that drawing performance is always reasonable.

It's difficult to overestimate the importance of resolution independence. At first glance, it seems like a straightforward, elegant solution to a time-honored problem (which it is). However, in order to design interfaces that are fully scalable, developers need to embrace a new way of thinking.

The Architecture of WPF

WPF uses a multilayered architecture. At the top, your application interacts with a high-level set of services that are completely written in managed C# code. The actual work of translating .NET objects into Direct3D textures and triangles happens behind the scenes, using a lower-level unmanaged component called milcore.dll. milcore.dll is implemented in unmanaged code because it needs tight integration with Direct3D and because it's extremely performance-sensitive.

Figure 1-2 shows the layers at work in a WPF application.

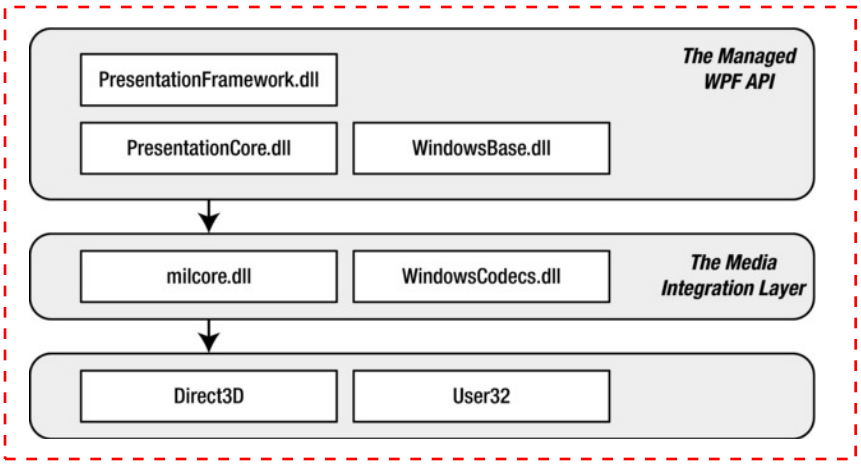

Figure 1-2. *The architecture of WPF*

Figure 1-2 includes these key components:

- *PresentationFramework.dll:* This holds the top-level WPF types, including those that represent windows, panels, and other types of controls. It also implements higher-level programming abstractions such as styles. Most of the classes you'll use directly come from this assembly.
- *PresentationCore.dll:* This holds base types, such as UIElement and Visual, from which all shapes and controls derive. If you don't need the full window and control abstraction layer, you can drop down to this level and still take advantage of WPF's rendering engine.
- *WindowsBase.dll:* This holds even more basic ingredients that have the potential to be reused outside of WPF, such as DispatcherObject and DependencyObject, which introduces the plumbing for dependency properties (a topic you'll explore in detail in Chapter 4).
- *milcore.dll:* This is the core of the WPF rendering system and the foundation of the Media Integration Layer (MIL). Its composition engine translates visual elements into the triangle and textures that Direct3D expects. Although milcore.dll is considered part of WPF, it's also an essential system component for Windows Vista and Windows 7. In fact, the Desktop Window Manager (DWM) uses milcore.dll to render the desktop.

■ **Note** milcore.dll is sometimes referred to as the engine for "managed graphics." Much as the common language runtime (CLR) manages the lifetime of a .NET application, milcore.dll manages the display state. And just as the CLR saves you from worrying about releasing objects and reclaiming memory, milcore.dll saves you from thinking about invalidating and repainting a window. You simply create the objects with the content you want to show, and milcore.dll paints the appropriate portions of the window as it is dragged around, covered and uncovered, minimized and restored, and so on.

- *WindowsCodecs.dll:* This is a low-level API that provides imaging support (for example, processing, displaying, and scaling bitmaps and JPEGs).
- *Direct3D:* This is the low-level API through which all the graphics in a WPF application are rendered.
- *User32:* This is used to determine what program gets what real estate. As a result, it's still involved in WPF, but it plays no part in rendering common controls.

The most important fact that you should realize is Direct3D renders *all* the drawing in WPF. It doesn't matter whether you have a modest video card or a much more powerful one, whether you're using basic controls or drawing more complex content, or whether you're running your application on Windows XP, Windows Vista, or Windows 7. Even two-dimensional shapes and ordinary text are transformed into triangles and passed through the 3-D pipeline. There is no fallback to GDI+ or User32.

The Class Hierarchy

Throughout this book, you'll spend most of your time exploring the WPF namespaces and classes. But before you begin, it's helpful to take a first look at the hierarchy of classes that leads to the basic set of WPF controls.

Figure 1-3 shows a basic overview with some of the key branches of the class hierarchy. As you continue through this book, you'll dig into these classes (and their relatives) in more detail.

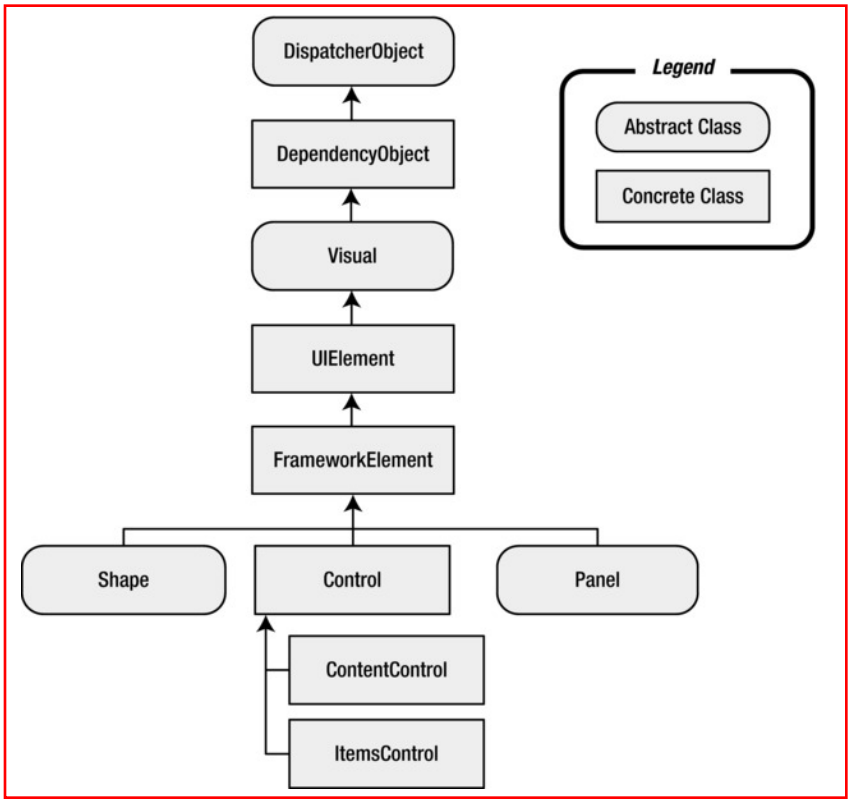

Figure 1-3. *The fundamental classes of WPF*

The following sections describe the core classes in this diagram. Many of these classes lead to whole branches of elements (such as shapes, panels, and controls).

■ **Note** The core WPF namespaces begin with System.Windows (for example, System.Windows, System.Windows.Controls, and System.Windows.Media). The sole exception is namespaces that begin with System.Windows.Forms, which are part of the Windows Forms toolkit.

System.Threading.DispatcherObject

WPF applications use the familiar single-thread affinity (STA) model, which means the entire user interface is owned by a single thread. It's not safe to interact with user interface elements from another thread. To facilitate this model, each WPF application is governed by a *dispatcher* that coordinates messages (which result from keyboard input, mouse movements, and framework processes such as layout). By deriving from DispatcherObject, every element in your user interface can verify whether code is running on the correct thread and access the dispatcher to marshal code to the user interface thread. You'll learn more about the WPF threading model in Chapter 31.

System.Windows.DependencyObject

In WPF, the central way of interacting with onscreen elements is through properties. Early on in the design cycle, the WPF architects decided to create a more powerful property model that baked in features such as change notification, inherited default values, and more economical property storage. The ultimate result is the *dependency property* feature, which you'll explore in Chapter 4. By deriving from DependencyObject, WPF classes get support for dependency properties.

System.Windows.Media.Visual

Every element that appears in a WPF window is, at heart, a Visual. You can think of the Visual class as a single drawing object that encapsulates drawing instructions, additional details about how the drawing should be performed (such as clipping, opacity, and transformation settings), and basic functionality (such as hit testing). The Visual class also provides the link between the managed WPF libraries and the milcore.dll that renders your display. Any class that derives from Visual has the ability to be displayed on a window. If you prefer to create your user interface using a lightweight API that doesn't have the higher-level framework features of WPF, you can program directly with Visual objects, as described in Chapter 14.

System.Windows.UIElement

UIElement adds support for WPF essentials such as layout, input, focus, and events (which the WPF team refers to by the acronym *LIFE*). For example, it's here that the two-step measure and arrange layout process is defined, which you'll learn about in Chapter 18. It's also here that raw mouse clicks and key presses are transformed to more useful events such as MouseEnter. As with properties, WPF implements an enhanced event-passing system called *routed events*. You'll learn how it works in Chapter 5. Finally, UIElement adds supports for commands (Chapter 9).

System.Windows.FrameworkElement

FrameworkElement is the final stop in the core WPF inheritance tree. It implements some of the members that are merely defined by UIElement. For example, UIElement sets the foundation for the WPF layout system, but FrameworkElement includes the key properties (such as HorizontalAlignment and Margin) that support it. UIElement also adds support for data binding, animation, and styles, all of which are core features.

System.Windows.Shapes.Shape

Basic shapes classes, such as Rectangle, Polygon, Ellipse, Line, and Path, derive from this class. These shapes can be used alongside more traditional Windows widgets such as buttons and text boxes. You'll start building shapes in Chapter 12.

System.Windows.Controls.Control

A *control* is an element that can interact with the user. It obviously includes classes such as TextBox, Button, and ListBox. The Control class adds additional properties for setting the font and the foreground and background colors. But the most interesting detail it provides is template support, which allows you to

replace the standard appearance of a control with your own stylish drawing. You'll learn about control templates in Chapter 17.

■ **Note** In Windows Forms programming, every visual item in a form is referred to as a control. In WPF, this isn't the case. Visual items are called elements, and only some elements are actually controls (those that can receive focus and interact with the user). To make this system even more confusing, many elements are defined in the System.Windows.Controls namespace, even though they don't derive from System.Windows.Controls.Control and aren't considered controls. One example is the Panel class.

System.Windows.Controls.ContentControl

This is the base class for all controls that have a single piece of content. This includes everything from the humble Label to the Window. The most impressive part of this model (which is described in more detail in Chapter 6) is the fact that this single piece of content can be anything from an ordinary string to a layout panel with a combination of other shapes and controls.

System.Windows.Controls.ItemsControl

This is the base class for all controls that show a collection of items, such as the ListBox and TreeView. List controls are remarkably flexible—for example, using the features that are built into the ItemsControl class, you can transform the lowly ListBox into a list of radio buttons, a list of check boxes, a tiled display of images, or a combination of completely different elements that you've chosen. In fact, in WPF, menus, toolbars, and status bars are actually specialized lists, and the classes that implement them all derive from ItemsControl. You'll start using lists in Chapter 19 when you consider data binding. You'll learn to enhance them in Chapter 20, and you'll consider the most specialized list controls in Chapter 22.

System.Windows.Controls.Panel

This is the base class for all layout containers—elements that can contain one or more children and arrange them according to specific layout rules. These containers are the foundation of the WPF layout system, and using them is the key to arranging your content in the most attractive, flexible way possible. Chapter 3 explores the WPF layout system in more detail.

WPF 4.5

WPF is a mature technology. It's been part of several releases of .NET, with steady enhancements along the way:

- *WPF 3.0:* The first version of WPF was released with two other new technologies: Windows Communication Foundation (WCF) and Windows Workflow Foundation (WF). Together, these three technologies were called the .NET Framework 3.0.
- *WPF 3.5:* A year later, a new version of WPF was released as part of the .NET Framework 3.5. The new features in WPF are mostly minor refinements, including bug fixes and performance improvements.

- *WPF 3.5 SP1:* When the .NET Framework Service Pack 1 (SP1) was released, the designers of WPF had a chance to slip in a few new features, such as slick graphical effects (courtesy of pixel shaders) and the sophisticated DataGrid control.
- *WPF 4:* This release added a number of refinements, including better text rendering, more natural animation, and support for multitouch.
- *WPF 4.5:* The latest version of WPF has the fewest changes yet, which reflects its status as a mature technology. Along with the usual bug fixes and performance tweaks, WPF 4.5 adds a number of refinements to that data binding system, including improvements to data binding expressions, virtualization, support for the INotifyDataError interface, and data view synchronization. You'll see these new features in Chapter 8, Chapter 19, and Chapter 22.

The WPF Toolkit

Before a new control makes its way into the WPF libraries of the .NET platform, it often begins in a separate Microsoft download known as the WPF Toolkit. But the WPF Toolkit isn't just a place to preview the future direction of WPF—it's also a great source of practical components and controls that are made available outside the normal WPF release cycle. For example, WPF doesn't include any sort of charting tools, but the WPF Toolkit includes a set of controls for creating bar, pie, bubble, scatter, and line graphs.

This book occasionally references the WPF Toolkit to point out a useful piece of functionality that's not available in the core .NET runtime. To download the WPF Toolkit, review its code, or read its documentation, surf to `http://wpf.codeplex.com`. There, you'll also find links to other Microsoft-managed WPF projects, including WPF Futures (which provides more experimental WPF features) and WPF testing tools.

Visual Studio 2012

Although you can craft WPF user interfaces by hand or using the graphic-design-oriented tool Expression Blend, most developers will start in Visual Studio and spend most (or all) of their time there. This book assumes you're using Visual Studio and occasionally explains how to use the Visual Studio interface to perform an important task, such as adding a resource, configuring project properties, or creating a control library assembly. However, you won't spend much time exploring Visual Studio's design-time frills. Instead, you'll focus on the underlying markup and code you need to create professional applications.

■ **Note** You probably already know how to create a WPF project in Visual Studio, but here's a quick recap. First, select File ➤ New ➤ TRA Project. Then, pick the Visual C# ➤ Windows group (in the tree on the left), and choose the WPF Application template (in the list on the right). You'll learn about the more specialized WPF Browser Application template in Chapter 24. After you pick a directory, enter a project name, and click OK, you'll end up with the basic skeleton of a WPF application.

Multitargeting

In the past, each version of Visual Studio was tightly coupled to a specific version of .NET. Visual Studio 2012 doesn't have this restriction—it allows you to design an application that targets any version of .NET from 2.0 to 4.5.

Although it's obviously not possible to create a WPF application with .NET 2.0, all later versions have WPF support. You may choose to target an older version, such as .NET 3.5 or .NET 4 to get the broadest possible compatibility. For example, a .NET 3.5 application can run on the .NET 3.5, 4, and 4.5 runtimes. Or, you may choose to target .NET 4.5 to get access to newer features in WPF or in the .NET platform. However, if you need to support legacy Windows XP computers, you can't target a version part of .NET 4, because this is the last .NET release that supports Windows XP.

When you create a new project in Visual Studio, you can choose the version of the .NET Framework that you're targeting from a drop-down list at the top of the New Project dialog box, just above the list of project templates (see Figure 1-4).

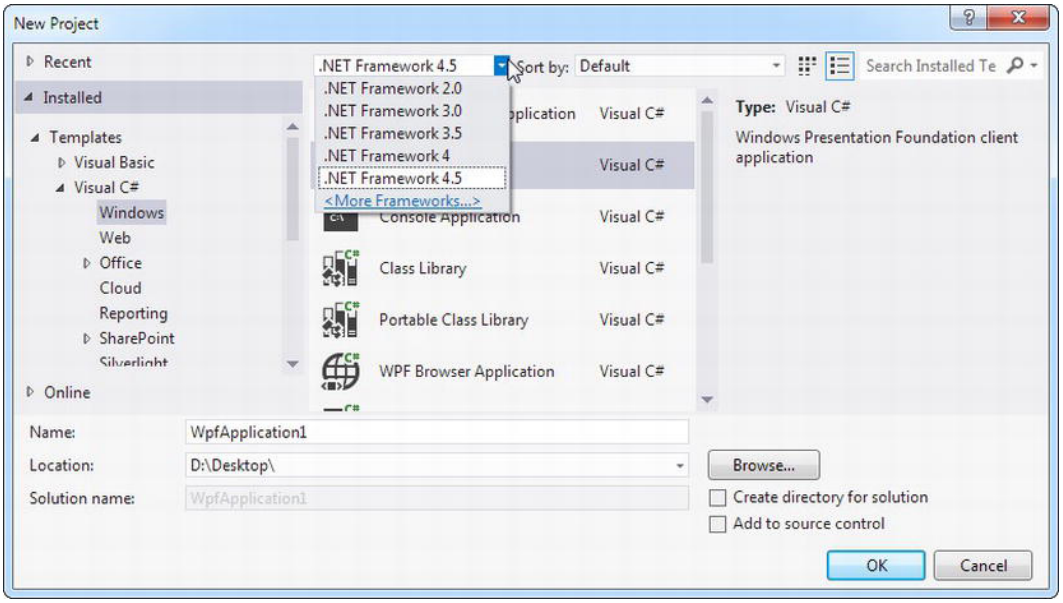

Figure 1-4. *Choosing the target version of the .NET Framework*

You can also change the version you're targeting at any point afterward by double-clicking the Properties node in the Solution Explorer and changing the selection in the Target Framework list.

To provide accurate multitargeting, Visual Studio includes *reference assemblies* for each version of .NET. These assemblies include the metadata of every type but none of the code that's required to implement it. That means Visual Studio can use the reference assembly to tailor its IntelliSense and error checking, ensuring that you aren't able to use controls, classes, or members that aren't available in the version of .NET that you're targeting. It also uses this metadata to determine what controls should appear in the Toolbox, what members should appear in the Properties window and Object Browser, and so on, ensuring that the entire IDE is limited to the version you've chosen.

The Visual Studio Designer

Visual Studio includes a rich designer for creating WPF user interfaces. But just because Visual Studio 2012 allows you to drag and drop WPF windows into existence doesn't mean you should start doing that right now—or at all. As you'll learn in Chapter 3, WPF uses a flexible and nuanced layout model that allows you to use different strategies for sizing and positioning the elements in your user interface. To get the result you need, you'll need to choose the right combination of layout containers, arrange them appropriately, and configure their properties. Visual Studio can help you out in this task, but it's far easier if you learn the basics of XAML markup and WPF layout *first*. Then, you'll be able to watch as Visual Studio's visual designer generates your markup, and you can modify it by hand as needed.

After you've mastered the syntax of XAML (Chapter 2) and you've learned about the family of WPF layout controls (Chapter 3), it's up to you to choose how you want to create your windows. There are professional developers who use Visual Studio, those who use Expression Blend, those who write XAML by hand, and those who use a combination of both methods (for example, creating the basic layout structure by hand and then configuring it with the Visual Studio designer).

The Last Word

In this chapter, you took your first look at WPF and the promise it holds. You considered the underlying architecture and briefly considered the core classes.

Clearly, WPF introduces many significant changes. However, there are five key principles that immediately stand out because they are so different from previous Windows user interface toolkits such as Windows Forms. These principles are the following:

- *Hardware acceleration:* All WPF drawing is performed through DirectX, which allows it to take advantage of the latest in modern video cards.
- *Resolution independence:* WPF is flexible enough to scale up or down to suit your monitor and display preferences, depending on the system DPI setting.
- *No fixed control appearance:* In traditional Windows development, there's a wide chasm between controls that can be tailored to suit your needs (which are known as *owner-drawn* controls) and those that are rendered by the operating system and essentially fixed in appearance. In WPF, everything from a basic Rectangle to a standard Button or more complex Toolbar is drawn using the same rendering engine and completely customizable. For this reason, WPF controls are often called *lookless controls*—they define the functionality of a control, but they don't have a hardwired "look."
- *Declarative user interfaces:* In the next chapter, you'll consider XAML, the markup standard you use to define WPF user interfaces. XAML allows you to build a window without using code. Impressively, XAML doesn't limit you to fixed, unchanging user interfaces. You can use tools such as data binding and triggers to automate basic user interface behavior (such as text boxes that update themselves when you page through a record source, or labels that glow when you hover overtop with the mouse), all without writing a single line of C#.
- *Object-based drawing*: Even if you plan to work at the lower-level visual layer (rather than the higher-level element layer), you won't work in terms of painting and pixels. Instead, you'll create shape objects and let WPF maintain the display in the most optimized manner possible.

You'll see these principles at work throughout this book. But before you go any further, it's time to learn about a complementary standard. The next chapter introduces XAML, the markup language used to define WPF user interfaces.