

Introducing Windows Presentation  
 Foundation

 David Chappell, Chappell & Associates

September 2007

© Copyright Microsoft Corporation 2007. All rights reserved.

Contents

[Describing Windows Presentation Foundation 3](#_Toc178740662)

[Illustrating the Problem 3](#_Toc178740663)

[Addressing the Problem: What Windows Presentation Foundation Provides 4](#_Toc178740664)

[A Unified Platform for Windows-Based User Interfaces 4](#_Toc178740665)

[The Ability for Developers and Designers to Work Together 9](#_Toc178740666)

[Interoperability with Existing User Interface Technologies 10](#_Toc178740667)

[Using Windows Presentation Foundation 11](#_Toc178740668)

[The Technology of Windows Presentation Foundation 12](#_Toc178740669)

[Application Model 12](#_Toc178740670)

[Layout and Controls 13](#_Toc178740671)

[Styles and Templates 15](#_Toc178740672)

[Text 16](#_Toc178740673)

[Documents 16](#_Toc178740674)

[Images 18](#_Toc178740675)

[Video and Audio 18](#_Toc178740676)

[Two-Dimensional Graphics 19](#_Toc178740677)

[Three-Dimensional Graphics 20](#_Toc178740678)

[Transformation and Effects 21](#_Toc178740679)

[Animation 22](#_Toc178740680)

[Data Binding 23](#_Toc178740681)

[User Interface Automation 23](#_Toc178740682)

[Interfaces for Add-ins 24](#_Toc178740683)

[Applying Windows Presentation Foundation 24](#_Toc178740684)

[Standalone WPF Applications 24](#_Toc178740685)

[XAML Browser Applications: XBAPs 25](#_Toc178740686)

[XPS Documents 25](#_Toc178740687)

[Tools for Windows Presentation Foundation 27](#_Toc178740688)

[For Developers: Visual Studio’s WPF Designer 28](#_Toc178740689)

[For Designers: Expression Blend 29](#_Toc178740690)

[Choosing an Interface Technology 31](#_Toc178740691)

[Interfaces for Windows Applications: WPF and Windows Forms 31](#_Toc178740692)

[Standards-Based Web Interfaces: ASP.NET and ASP.NET AJAX 32](#_Toc178740693)

[Rich Internet Applications: Silverlight 33](#_Toc178740694)

[Conclusion 35](#_Toc178740695)

[For Further Reading 35](#_Toc178740696)

[About the Author 35](#_Toc178740697)

# Describing Windows Presentation Foundation

Nothing is more important than a user’s experience of an application. While many software professionals are more interested in how an application works, its users care deeply about its user interface. An application’s interface is a major part of the complete user experience with that software, and to many of its users, the experience *is* the application. Providing a better experience through a better interface can improve productivity, help create loyal customers, increase sales on a Web site, and more.

Once happy with purely character-based interfaces, users have now become accustomed to graphical interfaces. Yet the requirements for user interfaces continue to advance. Graphics and media have become more widely used, and the Web has conditioned a generation of people to expect easy interaction with software. The more time people spend interacting with applications, the more important the interfaces to those applications become. To keep up with increasing expectations, the technology used to create user interfaces must also advance.

The goal of Windows Presentation Foundation (WPF) is to provide these advances for Windows. First included in version 3.0 of Microsoft’s .NET Framework, an enhanced version of the technology is now part of the .NET Framework 3.5. Using WPF, developers and designers can create interfaces that incorporate documents, media, two- and three-dimensional graphics, animations, and much more. Like everything else in the .NET Framework 3.5, WPF is available for Windows Vista, Windows XP, Windows Server 2003, and Windows Server 2008. This paper introduces WPF, describing its various parts. The intent is to make clear the problems this technology addresses, then survey the solutions that WPF provides.

## Illustrating the Problem

Suppose a hospital wants to create a new application for examining and monitoring patients. The requirements for this new application’s user interface might include the following:

Displaying images and text about the patient.

1. Displaying and updating two-dimensional graphics showing the patient’s vital signs, such as heart rate and blood pressure.
2. Providing three-dimensional views and overlays of patient information.
3. Presenting video of ultrasounds and other diagnostics, perhaps allowing physicians and nurses to add annotations.
4. Allowing hospital staff to read and make notations on documents describing the patient and her condition.

These requirements are ambitious, but they’re not unreasonable. User interfaces that present the right information in the right way at the right time can have significant business value. In situations such as the health care example described here, they can actually save lives. In less critical scenarios, such as on-line merchants or other consumer-oriented applications, providing a powerful user experience can help differentiate a company’s offerings from its competitors, increasing both sales and the value of the firm’s brand. The point is that many modern applications can benefit from providing interfaces that integrate graphics, media, documents, and the other elements of a modern user experience.

Building this kind of interface on Windows is possible with the technologies that preceded WPF, but it’s remarkably challenging. Two of the major hurdles are:

1. Many different technologies have been used for working with graphics, images, and video. Finding developers who are competent to work with these diverse technologies can be difficult and expensive, as is maintaining the applications they create.
2. Designing an interface that effectively presents all of this functionality to users is hard. Professional designers are required—software developers commonly don’t have the right skills—but designers and developers face significant challenges in working together, especially with full-featured interfaces like the one described here.

There’s no inherent reason why creating powerful, modern user interfaces should be so complex. A common foundation could address all of these challenges, offering a unified approach to developers while letting designers play an important role. As described next, this is exactly the intent of WPF.

## Addressing the Problem: What Windows Presentation Foundation Provides

Three aspects of what WPF provides stand out as most important. They are:

1. A unified platform for Windows-based user interfaces;
2. The ability for developers and designers to work together;
3. Interoperability with existing user interface technologies.

This section describes each of these three.

### A Unified Platform for Windows-Based User Interfaces

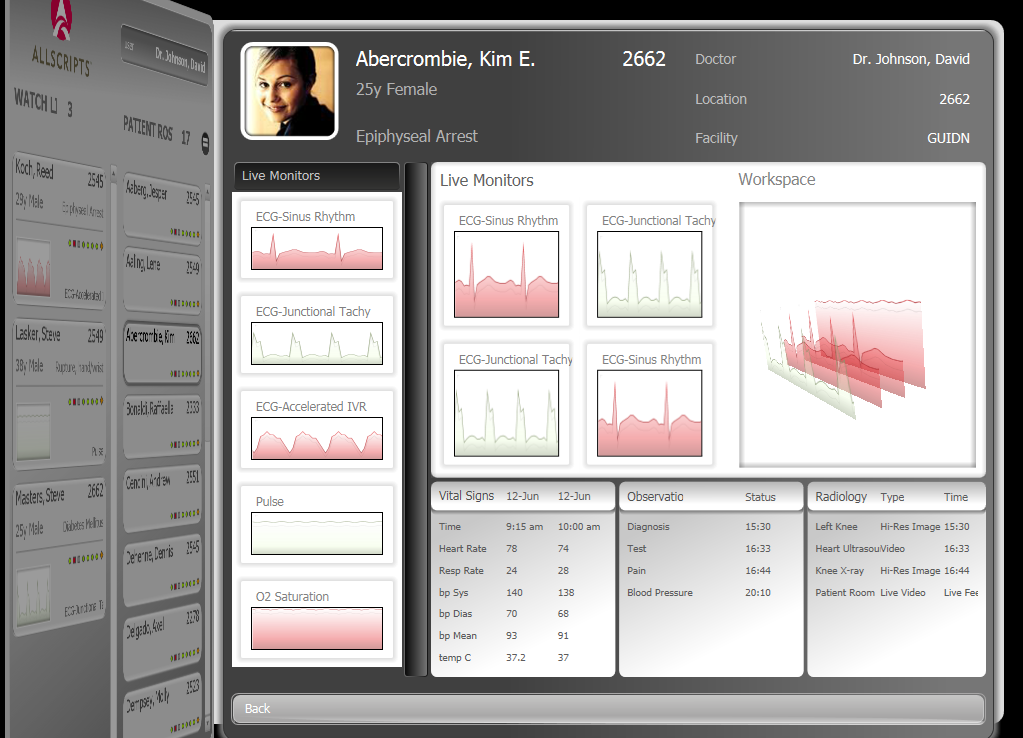
In the world before WPF, creating a Windows user interface like the one described earlier would require using several different technologies. The figure below summarizes the situation.

To create the forms, controls, and other typical aspects of a Windows graphical user interface, a developer would most likely choose Windows Forms. If the interface needs to display documents, Windows Forms has some support for on-screen documents, while fixed-format documents might use Adobe’s PDF. For images and two-dimensional graphics, that developer will use GDI+, a distinct programming model which is also accessible via Windows Forms. To display video and audio, he might rely on Windows Media Player, and for three-dimensional graphics, he’ll use Direct3D, a standard part of Windows.

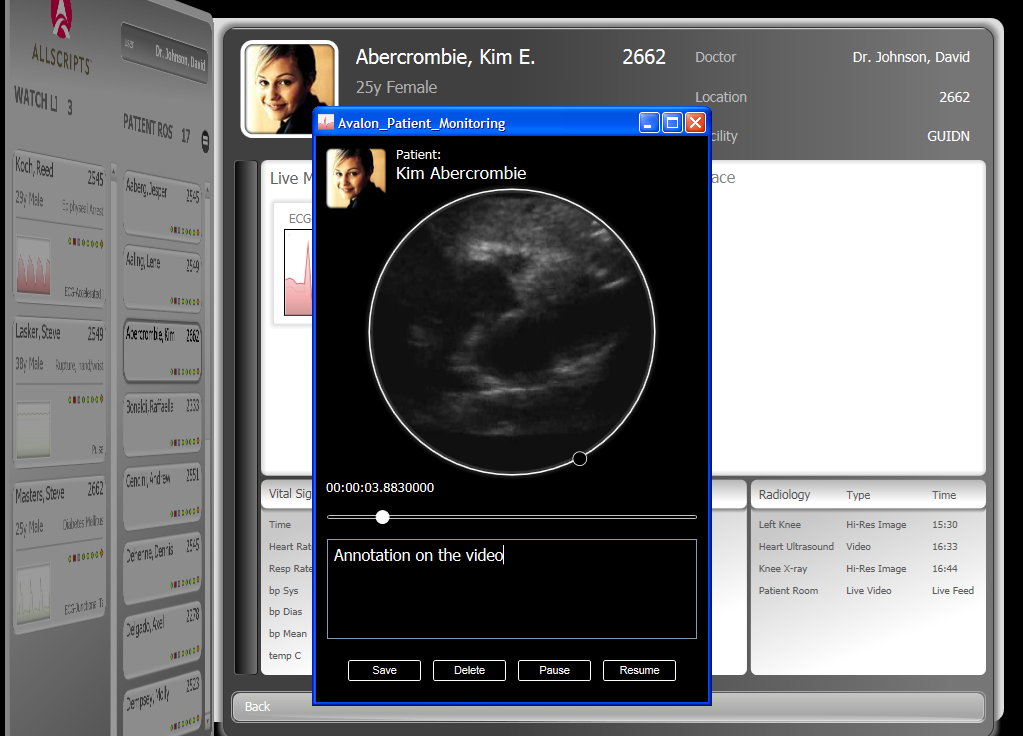
This complicated situation exists solely for historical reasons. No one would argue that it makes much sense. What does make sense is providing a single unified solution: WPF. Developers creating applications with WPF will likely use it to address all of the areas listed above. After all, why not use one coherent foundation for creating user interfaces rather than a diverse collection of independent technologies?

WPF doesn’t replace everything on this list, of course. Windows Forms applications will continue to have value, and even in a WPF world, some new applications will continue to use Windows Forms. It’s also important to note that WPF can interoperate with Windows Forms, something that’s described in more detail later in this section. Windows Media Player continues to have an independent role to play, and PDF documents will continue to be used. Direct3D also remains an important technology for games and some other kinds of applications. (In fact, WPF itself relies on Direct3D for all rendering.)

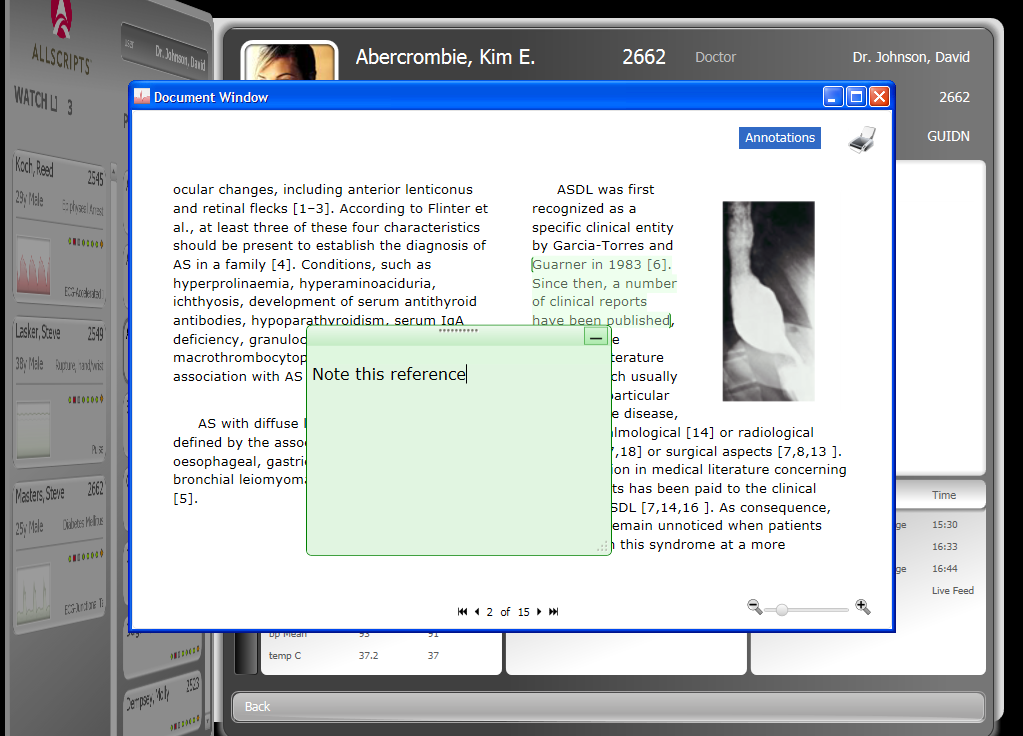
Yet by providing a broad range of functionality in a single technology, WPF can make creating modern user interfaces significantly easier. To get a sense of what this unified approach allows, here’s a typical screen that a WPF-based version of the health care application described above might present to a user:



This screen contains text and images along with both two- and three-dimensional graphics. All of this was produced using WPF—the developer doesn’t need to write code that uses specialized graphics technologies such as GDI+ or Direct3D. Similarly, WPF allows displaying and perhaps annotating video, such as the ultrasound feed shown below.



WPF also allows displaying documents in a readable way. In the hospital application, for instance, a physician might be able to look up notes about a patient’s treatment or access current medical research on a relevant topic. As with video, the physician might be able to add annotations as the screen below shows.



Notice that the document is displayed in readable columns and that the user can move through it a page at a time rather than by scrolling. Improving on-screen readability is a worthy aim, and it’s an important goal of WPF. Useful as on-screen documents are, however, fixed-format documents can sometimes be the right choice. Because they look the same on screen and on a printer, fixed-format documents provide a standard look in any situation. To define this type of document, Microsoft has created the XML Paper Specification (XPS). WPF also provides a group of application programming interfaces (APIs) that developers can use to create and work with XPS documents.

Yet creating modern user interfaces means more than just unifying what were once diverse technologies. It also means taking advantage of modern graphics cards, and so WPF exploits whatever graphics processing unit (GPU) is available on a system by offloading as much work as possible to it. Furthermore, modern interfaces shouldn’t be constrained by the limitations of bit-mapped graphics. Accordingly, WPF relies entirely on vector graphics, allowing an image to be automatically resized to fit the size and resolution of the screen it’s displayed on. Rather than creating different graphics for display on a small monitor and a big-screen television, the developer can let WPF handle this.

By unifying all of the technologies required to create a user interface into a single foundation, WPF can make life significantly simpler for the people who create those interfaces. By requiring those people to learn only a single environment, WPF can make creating and maintaining applications less expensive. And by making it straightforward to build interfaces that incorporate graphics, video, and more, WPF can improve the quality—and business value—of how users interact with Windows applications.

### The Ability for Developers and Designers to Work Together

Providing a unified technology foundation for creating full-featured user interfaces is a good thing. Yet expecting the average developer to use this power wisely, creating comprehensible, easy-to-use interfaces, is probably asking too much. Creating good user interfaces, especially when they’re as comprehensive as the hospital example just described, often requires skills that most software professionals just don’t have. Even though many applications are built without them, the truth is that building great user interfaces requires working with professional interface designers.

But how can designers and developers work together? The way the two disciplines interact today is problematic. Most commonly, a designer uses a graphical tool to create static images of the screen layouts that an application should display. He then gives these images to the developer, whose job is to create the code that makes them real. Something that’s easy for a designer to draw, however, might be difficult or impossible for a developer to implement. Technology limitations, schedule pressures, lack of skill, misunderstandings, or simple disagreement might prevent the developer from fully realizing the designer’s vision. What’s needed is a better way for members of these two interdependent disciplines to work together without compromising the quality of the interface.

To allow this, WPF introduces the eXtensible Application Markup Language (XAML). XAML defines a set of XML elements such as Button, TextBox, Label, and many more to define exactly how a user interface looks. XAML elements typically have attributes as well, allowing various options to be set. For example, this simple XAML snippet creates a red button containing the word “No”:

<Button Background="Red“>

No

</Button>

Each XAML element corresponds to a WPF class, and each of that element’s attributes has a corresponding property or event in the class. For example, the same red button could be produced with this C# code:

Button btn = new Button();  
btn.Background = Brushes.Red;  
btn.Content = “No";

If everything expressible in XAML is also expressible in code—and it is—what’s the value of XAML? The answer is that building tools that generate and consume XML-based descriptions is much easier than doing the same thing with code. Because XAML offers a tool-friendly way to describe a user interface, it provides a better way for developers and designers to work together. The figure below illustrates the process.

A designer can specify how a user interface should look and interact using a tool such as Microsoft Expression Blend. Oriented entirely toward defining the look and feel of a WPF interface, this tool generates a description of that interface expressed in XAML. (While it might include a simple button like the example shown here, this description is in fact much more complex than the snippet above might suggest.) A developer then opens that XAML description with a tool such as Microsoft Visual Studio. Rather than recreating the interface from scratch based on static images produced by a designer, the interface definition itself is adopted wholesale. The developer then writes the code for the interface, such as event handlers, along with any other functionality the application requires. It’s also possible to create styles that can be globally applied to an application’s interface, allowing it to be customized as needed for different situations.

Letting designers and developers work together like this reduces the translation errors that tend to occur when developers implement interfaces from designer-created images. It can also allow people in these two roles to work in parallel, with quicker iteration and better feedback. And because both environments use the same build system, a WPF application can be passed back and forth between the two development environments. More specialized tools for designing XAML-defined interfaces are also available, such as Electric Rain’s ZAM 3D for creating three-dimensional interface elements.

Better user interfaces can increase productivity—they have measurable business value. Yet to create truly effective interfaces, especially in the multi-faceted world that WPF provides, designers must become first-class citizens in the process. A primary goal of XAML and the tools that support it is to make this possible.

### Interoperability with Existing User Interface Technologies

Since the initial release of the .NET Framework, many applications have been created using Windows Forms. Many applications were also created in the pre-.NET era with technologies such as Microsoft Foundation Classes (MFC) and Win32. Still other applications rely on DirectX for creating 3D graphics. While WPF can potentially address all of these areas, it must still work well with what’s already there; applications should be able to build on what they’re using today. Given this reality, interoperating with existing user interface technologies is a fundamentally important goal for WPF.

In fact, even with the arrival of WPF, some applications will continue to use other interface technologies. Windows Forms, for example, is clearly the right choice for anything that must run on systems where WPF isn’t available, such as older versions of Windows. New applications might also choose Windows Forms over WPF for other reasons. It’s important to understand that while Windows Forms has proven to be an effective interface technology for line-of-business (LOB) applications, WPF (today, at least) targets a slightly different problem. A developer creating an immersive user interface, such as the health care application shown earlier, would be happier with WPF than with Windows Forms. Someone creating a more conventional LOB application, however, might prefer Windows Forms. Both the current feature set and the tool support for Windows Forms are oriented toward LOB applications, while WPF has a somewhat different focus.

Another option is to combine these technologies, using aspects of WPF and Windows Forms in a single application. This is certainly possible, since each technology is capable of hosting user interface elements, known as *controls*, defined by the other. For example, Windows Forms provides a useful DataGridView control that has no analog in WPF, while WPF offers many things that Windows Forms does not, such as 3D graphics and animations. Although using WPF and Windows Forms together does have some restrictions, a large percentage of Windows applications can potentially use both to create their user interface.

WPF can also be used in concert with MFC or Win32. Once again, WPF controls can be hosted within existing Win32/MFC code, and existing Win32/MFC controls can be hosted within WPF. As with Windows Forms, there are some limitations in mixing these two worlds. Yet once again, the key point is that Windows applications can use WPF and Win32/MFC together. Using WPF doesn’t require throwing away all of an application’s existing user interface code.

One more interface technology is also important to mention here: Direct3D. This API, part of Microsoft’s DirectX family, has been a mainstay for Windows developers who create three-dimensional graphics. The advent of WPF in no way obsoletes Direct3D. In fact, as described earlier, WPF relies entirely on Direct3D for rendering. Yet since WPF also allows developers to create 3D graphics, developers working in 3D must decide between the two. The decision isn’t especially difficult, however. Direct3D is still the best choice for intensive 3D development, such as games and 3D-centered technical applications, e.g., high-end scientific visualization. WPF does make 3D graphics available to a much wider and less specialized audience, however, and it also allows integrating 3D graphics naturally with two-dimensional graphics, documents, and other aspects of an application’s user interface. Both WPF and Direct3D have distinct roles, and both have a good future as part of the Windows platform.

# Using Windows Presentation Foundation

Knowing what problems WPF addresses is useful, but having some understanding of how it addresses those problems is also useful. This section surveys the WPF technology itself, then looks at the different ways it’s applied in Windows desktop applications, Web browsers, and XPS documents.

## The Technology of Windows Presentation Foundation

Even though WPF offers a unified foundation for creating user interfaces, the technologies it contains can be examined in discrete, understandable parts. These parts include documents, images, graphics, animation, and more. All of them depend on WPF’s basic application model, however, which is described next.

### Application Model

Like other parts of the .NET Framework, WPF organizes its functionality into a group of namespaces, all contained in the System.Windows namespace. Whatever parts of this functionality it uses, the basic structure of every WPF application is much the same: a set of XAML pages and code associated with those pages.

As its root, every application inherits from WPF’s standard Application class. This class provides common services that are useful to every application. These include holding state that needs to be available to the entire application and providing standard methods such as Run, which starts the application, and Shutdown, which terminates it.

An Application object can be created with either XAML, via the Application element, or code, using the Application class. (This is true for virtually everything in WPF, but for simplicity, this paper always shows the XAML option.) Here’s a simple XAML illustration:

<Application xmlns=

"http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

StartupUri="Page1.xaml"

x:Class="Example.SimpleApp">

…

</Application>

This definition first specifies WPF’s namespace as the default for this element, then defines a prefix for the XAML namespace. (XAML is used for more than WPF, so these two namespaces aren’t synonymous.) It next uses the StartupUri attribute to indicate the name of the XAML page that should be loaded and displayed when the application is started. The final attribute, Class, is used to identify the class that contains the code associated with this Application. As mentioned earlier, WPF applications typically contain both XAML and code written in C# or Visual Basic, and so a *code-behind* file is used to contain code for this class. Following this opening Application tag appears the rest of the XAML used to define this application, all of which is omitted here, followed by the closing tag for the Application element.

Even though all WPF applications derive from the same root class, there are still plenty of choices that a developer needs to make. A big one is deciding whether an application should provide a traditional dialog-driven interface or a navigational interface. A dialog-driven interface provides the buttons and other elements that every Windows user is familiar with. A navigational interface, by contrast, acts much like a browser. Rather than opening a new window for a dialog, for instance, it commonly loads a new page. Interfaces like this are implemented as a group of pages, each consisting of a user interface defined in XAML together with logic expressed in a programming language. Like HTML-defined browser pages, XAML provides a Hyperlink element that can be used to link pages together. A user navigates through these pages much as she would through the pages of a Web-based application, relying on a History list to move back and forth. Don’t be confused, however—this is still a Windows application, with no requirement to run inside a browser.

Whatever interface style an application uses, it usually displays one or more windows. WPF provides a few choices for doing this. The simple Window class provides basic windowing functions, such as displaying, hiding, and closing a window, and it’s typically used by a WPF application that’s not using a navigational interface. NavigationWindow, used by applications that do have a navigational interface, extends the basic Window class with support for navigation. This support includes a Navigate method that allows the application to move to a new page, a journal that keeps track of the user’s navigation history, and various navigation-related events.

### Layout and Controls

To organize the various parts of an interface, a WPF application uses *panels* for layout. Each panel can contain children, including controls such as buttons and text boxes, and other panels. Different kinds of panels provide different layout options. A DockPanel, for example, allows its child elements to be positioned along the edges of the panel, while a Grid allows positioning its children precisely on a grid, just as its name suggests. The developer defines the number of rows and columns in the grid, then specifies exactly where any children should be placed. A Canvas lets a developer position its children freely anywhere within the panel’s boundaries.

Like any user interface technology, WPF provides a large set of controls, and developers are free to create custom controls as well. The standard set includes Button, Label, TextBox, ListBox, Menu, Slider, and other traditional atoms of user interface design. More complex controls are also provided, such as SpellCheck, PasswordBox, controls for working with ink (as with a Tablet PC), and more.

As usual in a graphical interface, events generated by the user, such as mouse movements and key presses, can be caught and handled by the controls in a WPF application. While controls and other user interface elements can be fully specified using XAML, events must be handled in code. For example, here’s a XAML definition of a simple Button on a Canvas:

<Canvas xmlns=

"http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

x:Class="Example.CodeForCanvas">

<Button Click="Button\_Click">

Click Here

</Button>

</Canvas>

The opening Canvas tag starts by defining the usual WPF and XAML namespaces. It then specifies that the code associated with this XAML can be found in a class named CodeForCanvas, which is contained in the .NET namespace Example. Next comes the definition of the Button itself, specifying “Click Here” as its on-screen text. The Click attribute on the opening Button tag indicates that this button relies on a method called Button\_Click to handle the click event. The C# code for that method might look like this:

namespace Example {

public partial class CodeForCanvas : Canvas {

void Button\_Click(object sender, RoutedEventArgs e) {

Button btn = e.Source as Button;

btn.Background = Brushes.Purple;

}

}

}

The namespace and class name match that specified in the Canvas tag just shown. The class CodeForCanvas inherits from the base Canvas class provided by WPF, and it’s defined as a partial class. Partial classes were a new addition in version 2.0 of the .NET Framework, and they allow combining code defined separately into a single class. In this case, the XAML-defined Canvas generates a partial class that gets combined with the partial class shown here. The result is a complete class capable of both displaying a canvas with a button and handling its event.

The Button\_Click method to handle that event is provided within the CodeForCanvas class. It follows the usual .NET Framework conventions for an event, although the event’s arguments are conveyed using the WPF-defined RoutedEventArgs class. This class’s Source property contains a reference to the Button that generated the event, which the method uses to change the button’s color to purple.

As this simple example suggests, the elements in a WPF user interface are organized into a visual tree. Here, the tree consists of just a Canvas with a single child Button, but in a real WPF application, this tree is typically much more complex. To actually create the on-screen interface, this visual tree must be rendered. Whenever possible, WPF relies on hardware rendering, letting the graphics card installed on the application’s machine handle the work. If the machine’s graphics hardware isn’t up to the job, however, WPF will render the interface using its own software. The decision is made at runtime by WPF—developers don’t need to do anything special.

Whether rendering is done in hardware or software, WPF always relies on an approach known as *retained mode* graphics. The creators of an application define what the visual tree looks like, typically using a combination of XAML and code. WPF itself then retains the information in this tree. Rather than requiring the application to repaint all or part of a window when the user uncovers it, for example, WPF handles this on its own. The elements that comprise the tree are stored as objects, not as pixels on the screen, and so WPF has enough information to handle this kind of rendering. Even if a window and the controls it contains are resized, WPF can re-render everything on its own. Because it understands the form of the graphics—lines, ellipses, and so on—and because it relies on vector graphics rather than maps of pixels, WPF can recreate the interface at the new size.

### Styles and Templates

It’s often useful to be able to define how some user interface element looks once, then apply that look over and over. Cascading Style Sheets (CSS) allow doing this in HTML pages, for example. WPF provides something similar with *styles*. The ability to define styles can be quite useful, as the popularity of CSS stylesheets suggests. They allow better separation between designers and developers, for instance, allowing a designer to create a uniform look for an interface while letting the developer ignore these details.

Using XAML’s Style element, the creator of a WPF application can define one or more aspects of how something should look, then apply that style over and over. For example, a style named ButtonStyle might be defined like this:

<Style x:Key="ButtonStyle">

<Setter Property="Control.Background" Value="Red"/>

<Setter Property="Control.FontSize" Value="16"/>

</Style>

Any Button defined using this style would be given a red background and use a font size of 16. Here’s how a Button can specify that it wishes to use this style:

<Button Style="{StaticResource ButtonStyle}">

Click Here

</Button>

As the appearance of “StaticResource” in this example suggests, WPF styles are typically defined as a resource, which is just data defined separately from an application’s code.

Styles allow more than the simple example shown here might suggest. A style can be derived from another style, for instance, inheriting and perhaps overriding its settings. A style can also define *triggers* that specify common aspects of interactive behavior. For example, a style might specify that hovering the mouse over a Button should cause the button’s background to turn yellow.

WPF also supports the use of *templates*. A template is similar to a style, and two different kinds are available:

1. Data templates: allow using XAML’s DataTemplate element to specify a group of characteristics for how data should be displayed. Colors, alignment, and more can be defined once in a data template, then used elsewhere in an application’s user interface.
2. Control templates: allow using XAML’s ControlTemplate element to define the appearance of a control.

Providing a straightforward way for an application’s creators to define the appearance of its interface makes sense. In WPF, styles and templates are primary mechanisms for doing this.

### Text

Most user interfaces display at least some text, and some display little else. Yet for most people, reading text on a screen can’t compare with reading a printed page. We’ve become accustomed to the high-quality depictions of letters and the relationships between them typically found in books and magazines. When we read on-screen text, things just aren’t the same—the text somehow doesn’t feel as readable.

WPF aims at closing this gap, making on-screen text as readable as a printed page. Toward this end, WPF supports industry-standard OpenType fonts, allowing existing font libraries to be used. It also supports the more recently defined ClearType technology. Through sub-pixel positioning, a technique for individually lighting up the sub-elements comprising each pixel on modern display screens, ClearType allows text to look smoother to the human eye. WPF also provides low-level support for rendering text via the Glyphs class. As described later, this class is used by XPS documents to represent characters.

To improve readability further, WPF also allows extras such as ligatures, where a group of characters are replaced by a single connected image. For instance, the group “ffi” will typically be replaced in a printed page by a single connected ligature containing those three characters. Adding this to on-screen text makes the reader feel more at home, even if she doesn’t consciously perceive the details that create that feeling.

### Documents

Making text more readable is a good thing, since text appears in buttons and lists and many other places in a user interface. Yet we care most about text when we’re reading longer chunks of it, such as in a document. Accordingly, improving on-screen readability also requires improving how documents are displayed. Toward this end, WPF supports two kinds of documents: *fixed* documents and *flow* documents.

Fixed documents look exactly the same whether they’re rendered on a screen or a printer. Knowing that a document will always look the same is important for some forms, legal documents, and other kinds of publications, and so fixed-format documents are important in a number of areas. The fixed-format documents supported by WPF are defined by XPS, which is described later in this paper. A fixed document’s contents can be specified using XAML’s FixedDocument element. This simple element contains just a list of PageContent elements, each containing the name of a page in the fixed document. To display a fixed document, WPF provides the DocumentViewer control. This control provides read-only display of an XPS document, letting the reader move backward and forward in the document, search for specific text, and more.

While fixed documents are meant to be used both on a screen and on paper, flow documents are intended solely for on-screen display. To make its contents as readable as possible, a flow document can adjust how text and graphics are displayed based on the window size and other factors. Unsurprisingly, flow documents are defined using a XAML element called FlowDocument. Here’s a simple example:

<FlowDocument

ColumnWidth="300"

IsColumnWidthFlexible="True"

IsHyphenationEnabled="True">

<Paragraph FontSize="12">

<Bold>Describing WPF</Bold>

</Paragraph>

<Paragraph FontSize="10">

WPF is the user interface technology for the .NET

Framework 3.5. It provides a unified foundation for modern

user interfaces, including support for documents, two- and

three-dimensional graphics, media, and more.

</Paragraph>

</FlowDocument>

This document asks to be displayed in a column with a width no less than 300. (The width is measured in *device-independent pixels*, each of which is defined to be 1/96th of an inch.) In the very next line, however, the document’s creator says that this width is flexible by setting the IsColumnWidthFlexible property to true. This authorizes WPF to change the width and number of columns that will be used to display this document. If the user changes the width of the window in which this document is displayed, for example, WPF can increase or decrease the number and the width of columns used to display the document’s text.

Next, the document requests hyphenation by setting the IsHyphenationEnabled property to true. Following this are the two paragraphs this document contains. The text inside each one is contained within a Paragraph element, each setting a different font size. The text in the first paragraph also indicates that it should be displayed in bold.

WPF defines several more FlowDocument options for improved readability. For instance, if the IsOptimalParagraphEnabled property is set to true, WPF will distribute white space as evenly as possible throughout a paragraph. This can prevent the “rivers” of white space that hurt readability, something that’s commonly done with printed documents. Flow documents also allow annotations, such as adding notes in ordinary text or, on Tablet PCs, in ink. Each annotation consists of an *anchor* that identifies what content in the document an annotation is associated with and *cargo* that contains the content of the annotation itself.

To display a FlowDocument, WPF includes a few different controls. They are the following:

1. FlowDocumentPageViewer: allows moving through a document one page at a time. This control provides a forward and back button along with a zoom control that allows the user to resize the document she’s reading.
2. FlowDocumentScrollViewer: provides a more traditional scrolling view of a FlowDocument, complete with a scrollbar on the right side of the page.
3. FlowDocumentReader: combines the functionality of both FlowDocumentPageViewer and FlowDocumentScrollViewer. This control allows the user to switch between a page-oriented view of a flow document (including seeing two pages at a time) and a scrolling view.

As more and more information is delivered digitally, the quality of the on-screen reading experience becomes more important. By providing adaptive display of information through flow documents, WPF attempts to improve this experience for Windows users.

### Images

Whether they represent company logos, pictures of sunsets, or something else, images are a fundamental part of many user interfaces. In WPF, images are typically displayed using the Image control. To show a JPEG file, for example, the following XAML could be used:

<Image

Width="200"

Source="C:\Documents and Settings\All Users\Documents\

My Pictures\Ava.jpg" />

The image’s width is set to 200, and once again, the units here are device-independent pixels. The file that contains the image is identified using the Source attribute.

An image file can contain information about the image—metadata—such as keywords and ratings applied by users, and WPF applications can read and write this information. An image can also be used in more interesting ways, such as painting it onto one face of a revolving three-dimensional object. Although the simple example shown here illustrates a common case, WPF allows images to be used in a significantly broader way.

WPF’s Image control can display images stored in various formats, including JPEG, BMP, TIFF, GIF, and PNG. It can also display images stored using Microsoft’s *Windows Media Photo* (also known as *HD Photo*) format, new with Windows Vista. Whatever format is used, WPF relies on the Windows Imaging Component (WIC) to produce the image. Along with coder/decoders (commonly known as *codecs*) for all of the image formats just listed, WIC also provides a framework for adding third-party codecs.

### Video and Audio

As both network and processor speeds have increased, video has become a larger part of how people interact with software. People also spend a good deal of time listening to music and other audio on their computers. Accordingly, WPF provides built-in support for both.

That support depends on the MediaElement control. Here’s a simple XAML example of how this control might be used:

<MediaElement

Source="C:\Documents and Settings\All Users\Documents\

My Videos\Ruby.wmv" />

This control can play WMV, MPEG, and AVI video, along with various audio formats.

### Two-Dimensional Graphics

For the last twenty years, creators of two-dimensional graphics in Windows have relied on the Graphics Device Interface (GDI) and its successor, GDI+. Yet even Windows Forms applications must access this functionality through a distinctly different namespace—2D graphics aren’t integrated into the user interface technology itself. The situation was even worse for three-dimensional graphics, since an entirely separate technology, Direct3D, was required. With WPF, this complexity goes away for a large share of applications. Both 2D and 3D graphics can be created directly in XAML or in code using the WPF libraries. Like everything else in WPF, the elements they use are just another part of an application’s visual tree.

For 2D graphics, WPF defines a group of *shapes* that applications can use to create images. They are:

1. Line: draws a straight line between two points.
2. Elllipse: draws an ellipse.
3. Rectangle: draws a rectangle.
4. Polygon: draws a closed shape defined by a group of connected straight lines.
5. Polyline: draws an open shape defined by a group of connected straight lines.
6. Path: draws shapes described by an arbitrary path. The shapes can be open or closed, and the lines in the path can be straight or curved. In fact, all of the other shapes exist solely for convenience, since Path can be used to draw lines, ellipses, rectangles, polygons, polylines, and more.

Using these classes to create simple graphics is straightforward. For example, the following XAML draws a red ellipse:

<Ellipse Width=”30” Height=”10” Fill="Red" />

Filling a shape relies on a *brush*. The example above uses the default, which is a solid color brush, but WPF provides several other options. For example, a rectangle filled with a color gradient changing horizontally from red to yellow can be defined with:

<Rectangle Width=”30” Height=”10”

Fill="HorizontalGradient Red Yellow" />

Several other brushes are also available, including a vertical gradient, a radial gradiant, and brushes that paint with images, bitmaps, and more. Although it’s not shown here, shapes can also use *pens* to specify the color, width, and style of their outline.

A key thing to understand about WPF is that because everything is built on a common foundation, combining different aspects of the user interface is straightforward. An application can display an Image inside a Rectangle, place an Ellipse within a Button, and much more. Because of this, combining 2D graphics with 3D graphics and other parts of an interface is straightforward.

Along with shapes, WPF also provides another group of classes for working with two-dimensional graphics. Known as *geometries*, these classes are similar in many ways to shapes. Like shapes, which include choices such as Line, Rectangle, Ellipse, and Path, geometries provide options such as LineGeometry, RectangleGeometry, EllipseGeometry, and PathGeometry. The most important difference between the two kinds of classes is that while shapes are typically used to draw visible images, geometries are more often used to define regions. If a square image needs to be cropped to fit inside a circle, for example, the EllipseGeometry class can be used to specify the circle’s boundaries. Similarly, if an application wishes to define a hit-testing region, such as an area in which mouse clicks will be detected, it can do this by specifying a geometry for that region.

Finally, it’s worth mentioning that everything described in this section is actually implemented on top of a lower-level interface called the *visual layer*. It’s possible to create graphics, images, and text using this layer directly. While doing this can be useful in some situations, such as for creating simple, high-performance graphics, the great majority of applications will use shapes and the other higher-level abstractions that WPF provides.

### Three-Dimensional Graphics

Two-dimensional graphics are a common part of Windows interfaces, and so WPF provides quite a bit of technology in this area. Three-dimensional graphics are less commonly used today, however, even though they can provide substantial value through better data visualization, 3D charts, product renderings, and more. Working in 3D has traditionally required a distinct skill set, one that’s not commonly found outside of game developers and other specialized groups. By making support for 3D graphics part of the standard environment, WPF aims at changing this.

Without WPF, 3D development on Windows typically relies on the Direct3D API. Like everything else in WPF, its support for 3D graphics uses Direct3D under the covers, but developers are presented with a significantly simpler world. While there are still cases where it makes sense to use Direct3D rather than WPF, as described earlier, Microsoft’s intent is that mainstream 3D development for Windows interfaces use WPF.

To display 3D graphics in WPF, an application uses the Viewport3D control. This control essentially provides a window into the three-dimensional world the application describes. A Viewport3D control can be used anywhere in a WPF interface, allowing 3D graphics to appear wherever they’re needed.

To create a 3D scene, a developer describes one or more *models*, then specifies how those models should be lit and viewed. As usual, all of these things can be specified using XAML, code, or a mix of the two. To describe a model, WPF provides a GeometryModel3D class that allows defining the model’s shape. Once a model is defined, its appearance can be controlled by applying different kinds of *material*. The SpecularMaterial class, for instance, makes a surface look shiny, while the DiffuseMaterial class does not.

Regardless of the materials it uses, a model can be lit in various ways. DirectionalLight provides light that comes from a specific direction, while AmbientLight provides uniform lighting for everything in a scene. Finally, to define how the model should be viewed, the developer specifies a *camera*. A PerspectiveCamera, for instance, allows specifying the distance and perspective from which a model is viewed, while an OrthographicCamera does the same thing, except without perspective: objects further from the camera don’t appear smaller.

Creating complex 3D scenes directly in either XAML or code isn’t simple. It’s safe to assume that for the great majority of WPF applications that use 3D, developers will rely on graphical tools to generate the necessary definitions. However it’s accomplished, the ability to use 3D graphics in a standard user interface has the potential to improve significantly the quality of what users see on their screens.

### Transformation and Effects

Along with providing a way to define shapes and other elements, WPF also offers developers the ability to transform these elements by rotating them, changing their size, and more. In XAML, elements such as RotateTransform and ScaleTransform are used to do this. These transformations can be applied to any user interface element. Here’s a simple example:

</Button>

<Button Content=”Click Here”>

<Button.RenderTransform>

<RotateTransform Angle=“45” />

</Button.RenderTransform>

</Button>

The RotateTransform element rotates the button by 45 degrees. While rotating a button like this isn’t especially useful, the fact that it’s possible indicates the generality of WPF’s design. Because the various aspects of a user interface don’t rely on different underlying technologies, they can be combined in diverse ways.

WPF also includes a few pre-defined effects. Like transformations, these effects can be applied to various aspects of a user interface, such as Buttons, ComboBoxes, and others. They include a blur effect that makes the interface element appear fuzzy, an outer glow effect that makes an element appear to glow, and a drop shadow effect that adds a shadow behind an interface element.

### Animation

The ability to make the elements in an interface move—to animate them—can be very useful. Clicking on a button might cause the button to appear to move down, then up, for instance, giving better feedback to the user. More complex animations can help create interfaces that engage their users by directing their attention and telling stories. WPF’s animation support makes this possible.

As with transformations, animations can be applied to many different aspects of the interface, including buttons, shapes, images, and more. Animation is accomplished by changing the value of one or more of an object’s properties over time. For example, an Ellipse might appear to be slowly squashed by incrementally decreasing its Height property over a period of two seconds.

It’s often useful to define a group of related animations. To allow this, WPF provides the Storyboard class. Each Storyboard can contain one or more *timelines*, and each of these can contain one or more animations. Various kinds of timelines are provided, allowing animations to run sequentially or in parallel. Here’s a simple (although slightly incomplete) XAML example that illustrates squashing an Ellipse:

<Ellipse Width="100" Height="50" Fill="Blue"

Name="EllipseForSquashing">

…

<Storyboard>

<DoubleAnimation

Storyboard.TargetName="EllipseForSquashing"

Storyboard.TargetProperty="Height"

From="50" To="25" Duration="0:0:2" />

</Storyboard>

…

</Ellipse>

The example begins with the definition of an Ellipse, as seen earlier in this paper. Here, however, the Name property is also used, assigning an identifier that allows this Ellipse to be referenced later. Some details are omitted, but to define the animation in XAML, a Storyboard element must appear. Because Ellipse’s Height property is of the type double, the Storyboard contains a DoubleAnimation element. This element specifies the name of the Ellipse being animated, the property that will be changed, and exactly what those changes should be. Here, the value of Height is being changed from 50 to 25 over a period of two seconds.

Animations can be much more complex than this. They can be triggered by events, such as mouse clicks, be paused and then resumed, be set to repeat some number of times (or forever), and more. The goal is to allow developers to create user interfaces that provide better feedback, offer more functionality, and are all-around easier to use than they otherwise might be.

### Data Binding

Most user interfaces display some kind of data. To make life simpler for the developers who create those interfaces, data binding can be used to make displaying data easier. Data binding allows directly connecting what a WPF control displays with data that lives outside that control. For example, the value of the Text property in a WPF TextBox control might be bound to a property called Name in an Employee object that’s part of this application’s business logic. A change to either property could then be reflected in the other. If a user updated the value in the TextBox, the Employee object’s Name property would also change, and vice-versa.

Creating this kind of connection between properties in two objects requires using WPF’s Binding class. Here’s a slightly simplified XAML illustration of how this might look:

<TextBox …>

<TextBox.Text>

<Binding Path=”Name” />

</TextBox.Text>

</TextBox>

In this example, the Binding element’s Path attribute is used to identify the property to which the TextBox’s Text property should be bound. Path is used when the object this property is part of (which will be specified at runtime) is a Common Language Runtime (CLR) object, defined in a language such as C# or Visual Basic. Along with CLR objects, WPF’s data binding can also connect to XML data directly using Binding’s XPath property. This option creates an XPath query that selects one or more nodes in an XML document referencing the specified data. And beginning with the version of WPF in the .NET Framework 3.5, it’s also possible to use Language Integrated Query (LINQ) to access XML-defined data.

More complex data binding options are also possible. For example, list bindings allow the contents of a ListBox control to be populated from any CLR object that implements the standard IEnumerable interface. If necessary, data can also be filtered or sorted before it’s displayed. Whatever data binding option is used, the intent is to make a common requirement—displaying data in a user interface—as straightforward as possible.

### User Interface Automation

The most common user of a WPF interface is, of course, a person. But there are times when a user interface needs to be driven not by a human being, but instead by other software. WPF’s user interface (UI) automation makes this possible.

Suppose, for example, that a developer wishes to create automated test scripts for an interface. Using the programmatic access that UI automation provides, she can create scripts that drive the interface just as a human user would. UI automation is also useful for creating accessibility aids, such as a tool that reads aloud the various elements of the interface. Because it allows programmatically walking through the tree that contains those elements, UI automation makes building these kinds of tools possible.

To allow this, WPF creates a UI automation tree. This tree consists of AutomationElement objects, each representing something in the interface. The root of the tree is the Desktop, and each open application is a child of this root. The tree continues into each of these applications, with each WPF control represented as one (or sometimes more than one) AutomationElement objects. To allow complete programmatic access to the interface, everything that a user can interact with is represented as a distinct AutomationElement. For example, a control with multiple buttons will have both the control itself and each button represented as a distinct AutomationElement object. Building this level of granularity into the UI Automation tree allows a client application, whether it’s a test script, an accessibility aid, or something else, to access each component of the interface just as would a human user.

UI automation isn’t the most mainstream aspect of WPF. Most people will probably never use it. Yet those who need it, such as software testers and users with disabilities, *really* need it. Something doesn’t have to be widely used to be important.

### Interfaces for Add-ins

Many developers would like to let others add functionality to the applications they create. To do this safely, the .NET Framework 3.5 provides support for creating isolated *add-ins*. Implemented largely in a new System.AddIn namespace, developers can use add-ins for things such as allowing extensions to a social networking application, providing the ability to display advertising in a news reader, and more. Beginning with the .NET Framework 3.5, WPF allows creating add-in user interfaces that integrate with and extend the interface of the host application.

## Applying Windows Presentation Foundation

WPF contains a remarkable amount of technology. While all of it relates to interacting with people, the technology is applied today in three related ways: standalone WPF applications, XAML Browser Applications (XBAPs), and XPS documents. This section looks at each of these three.

### Standalone WPF Applications

The most general—and by far the most common—way to use WPF is in a standalone application. Virtually any .NET Framework application that runs on Windows can have a WPF interface. That application might run entirely on Windows or, more likely, it might at least occasionally communicate with other software via the Internet or some other network. Although it’s not required, WPF applications commonly use Windows Communication Foundation (WCF) to accomplish this.

Like other Windows applications, a standalone WPF application can be installed from a local disk or from a network server. It can also be installed using the .NET Framework’s ClickOnce facility. ClickOnce provides a straightforward way for Internet Explorer users to download and install Windows applications, including WPF applications, and to have those applications automatically updated when they change.

### XAML Browser Applications: XBAPs

While standalone WPF applications offer the most capability, they’re not always the right choice. Some situations make more sense with a client that runs in a Web browser rather than as a Windows application. To let these clients present modern user interfaces, especially in enterprise scenarios, WPF provides XBAPs.

XBAPs let developers use most of WPF’s capabilities in a browser-hosted application. They also allow a common programming model, using mostly the same code, for standalone applications and browser applications. XBAPs can run inside either Internet Explorer or Firefox, and they can act as a client for Web applications built using ASP.NET, JavaServer Pages (JSP), or other Web technologies. To communicate back to this Web application, an XBAP can use HTTP or SOAP. Whatever server platform is used, an XBAP is always loaded via ClickOnce. It presents no dialogs or prompts to the user during this process, however; an XBAP loads just like a Web page. Because of this, XBAPs don’t appear on the Start menu or in Add/Remove Programs.

While it’s not strictly required, XBAPs typically present a navigational interface to the user. This lets the application behave like a Web client, which is probably what the user expects. In Internet Explorer 7, an XBAP uses the forward and back buttons of the browser itself, and the XAML pages a user accesses will appear in the browser’s history list. In Internet Explorer 6, the XBAP displays its own forward and back buttons, along with maintaining its own history list.

Because it’s loaded from the Web and runs inside a browser, an XBAP is given only partial trust by the .NET Framework’s code access security. Accordingly, there are a number of things that a standalone WPF application can do that an XBAP cannot. For example, an XBAP deployed from the Internet zone can’t create standalone windows, display application-defined dialogs, or access the file system beyond a limited Isolated Storage area. It also can’t use any code created with Windows Forms, MFC, or direct Win32 calls, nor can it use unmanaged code. While it’s fair to think of XBAPs as providing a niche solution, that niche—browser-hosted WPF clients for enterprise applications—can be important in some situations.

### XPS Documents

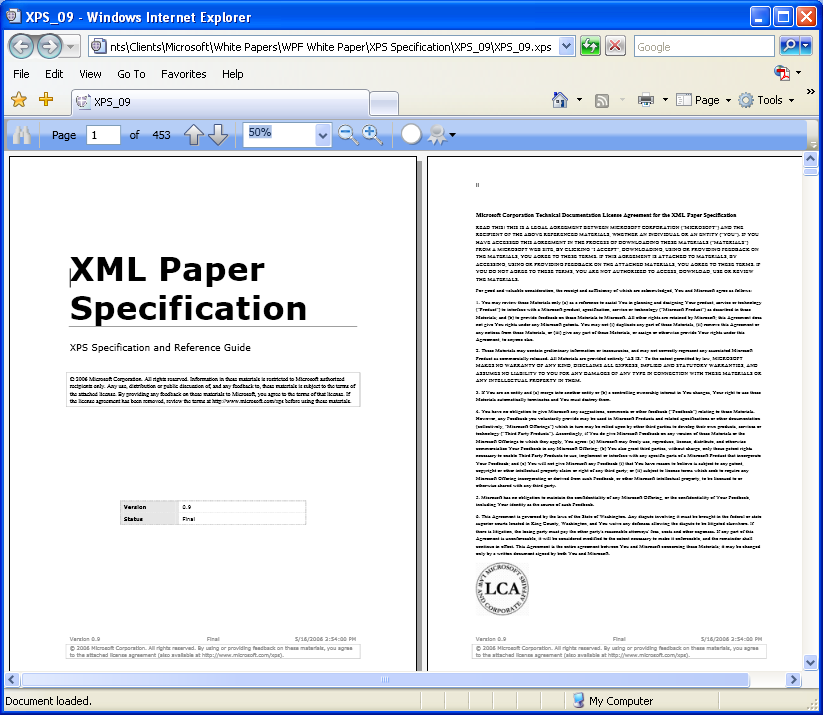
Fixed-format documents, which in the WPF world means XPS documents, clearly have a role in user interfaces. As described earlier, WPF provides the DocumentViewer control for displaying XPS documents. Yet while it certainly makes sense to include this control in WPF, it’s less obvious why XPS itself should be considered part of WPF. After all, the XPS specification provides a highly detailed way to define fixed-format documents, and the documents themselves can be used in different ways. Everything else in WPF is focused solely on creating a user interface. Given its broader purview, why include XPS under the WPF umbrella?

One big reason is that XPS documents are defined using XAML. Only a small subset of XAML is used, including the Canvas element for layout, the Glyphs element for representing text, and the Path element for creating two-dimensional graphics, but every XPS document is really a XAML document. Given this, viewing XPS as part of WPF is plausible.

Still, one of XPS’s most important applications isn’t about on-screen user interfaces. Beginning with Windows Vista, XPS becomes a native print format for Windows. XPS acts as a page description language, and so XPS documents can be rendered directly by XPS-aware printers. This allows using a single description format—XAML—all the way from the screen to the printer. It also improves on existing GDI-based print mechanism in Windows, providing better print support for complex graphic effects such as transparency and gradients.

Along with XAML, an XPS document can contain binary data such as images in various formats (including JPEG, PNG, TIFF, and HD Photo), font data, information about document structure, and more. If necessary, XPS documents can also be digitally signed using the W3C XML Signature definitions and X.509 certificates. Whatever it contains, every XPS document is stored in a format defined by the Open Packaging Conventions (OPC). OPC specifies how the various parts of an XML document (not just an XPS or XAML document) are related, how they’re stored in a standard ZIP format, and more. Microsoft Office 2007 also uses OPC for its XML formats, providing some commonality between the two kinds of documents.

Users of a WPF application can view XPS documents via WPF’s DocumentViewer control, as mentioned earlier. Microsoft also provides an XPS viewer application, built on the DocumentViewer control, as shown below. Like the control, this application lets users move through documents page by page, search for text, and more. XPS documents are not Windows-specific, and so Microsoft plans to provide XPS viewers for other platforms as well, such as the Apple Macintosh.



To let developers work with XPS documents, WPF provides a set of APIs to create, load, and manipulate them. WPF applications can also work with documents at the OPC level, allowing generalized access to XPS documents, Office 2007 documents, and others. Applications built using Microsoft’s Windows Workflow Foundation can also use these APIs to create workflows that use XPS documents.

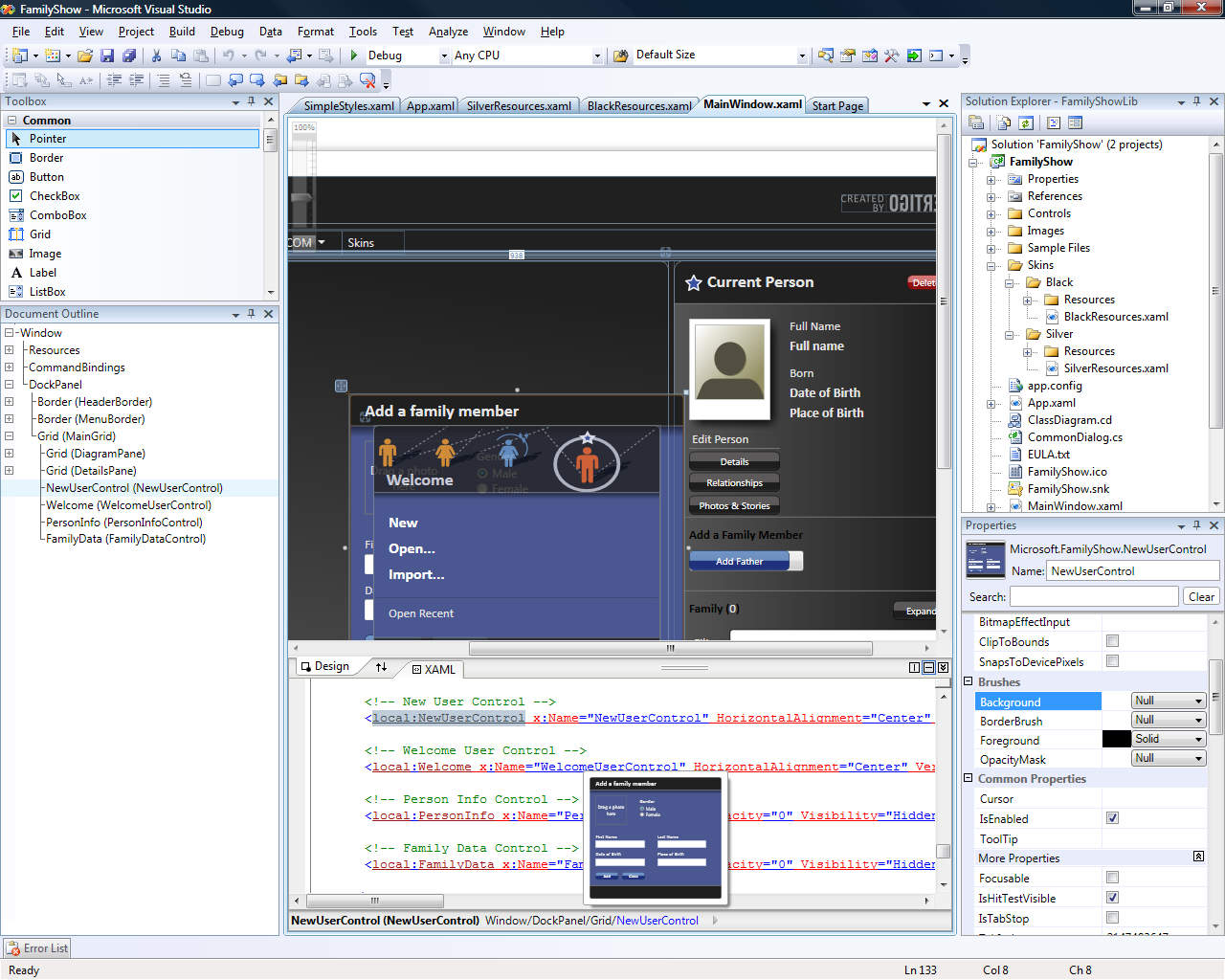
By allowing applications to display and work with fixed format documents, WPF integrates this component of modern user interfaces into its consistent approach. By using this same format to print documents, Windows Vista allows a better match between what people see on the screen and what they see on paper. While this type of document probably isn’t the first thing people expect from a user interface technology, the broad use of XPS illustrates the range that a technology like WPF can cover.

# Tools for Windows Presentation Foundation

WPF provides lots of functionality for developers, which is a good thing. No matter how powerful it is, though, a technology can be made more useful by good tools. For WPF, Microsoft provides one tool aimed specifically at developers and another aimed at designers. This section takes a brief look at both.

## For Developers: Visual Studio’s WPF Designer

Visual Studio 2008, the most recent release of Microsoft’s flagship tool for software developers, includes the WPF designer. Developers can use this tool to create WPF interfaces graphically. The figure below shows an example.



As this screen shot illustrates, the WPF designer lets developers work with both a graphical view of the interface and directly with XAML. (And, of course, the developer can also work directly with C# or VB code.) As usual with Visual Studio support for user interface development, available controls are listed on a palette (shown in the upper left of this screen), and the developer can drag and drop them onto the design surface to create her interface. The properties window in the lower right allows setting the properties of each control, customizing its appearance and behavior for this application.

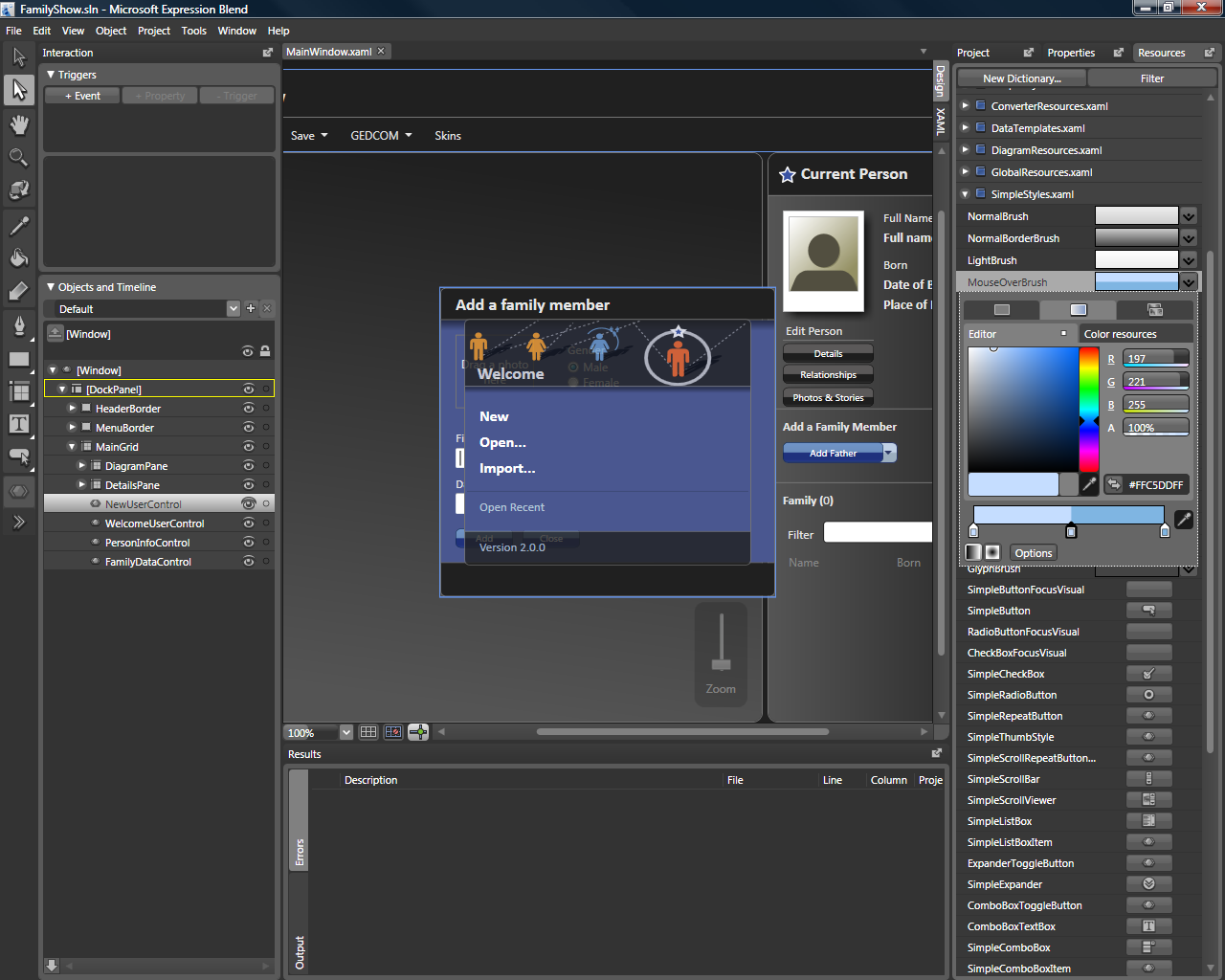
The WPF designer is similar in many ways to the graphical designer for Windows Forms. It’s important to understand, however, that just as WPF and Windows Forms target slightly different problems, the developer tools for each one are also somewhat different. While the Windows Forms designer aims at providing full-featured support for developers of LOB applications, the Visual Designer for WPF is instead intended to boost the productivity of today’s WPF developers. Just as WPF by design doesn’t offer everything an LOB developer might need today, the Visual Designer for WPF by design doesn’t offer every capability that’s provided by the Windows Forms designer.

Still, this tool provides a variety of useful things for WPF developers. These include Intellisense for XAML, allowing statement completion in its XAML editor, along with Live Thumbnaills, a real-time illustration of how the current visual element will look. In the example shown above, for instance, the small blue box in the lower center of the screen shows how the *Add a family member* element will be displayed on-screen. The designer also provides support for debugging WPF applications.

Graphical design tools are an important part of the modern developer arsenal. The WPF designer makes life easier for software professionals who build user interfaces with this interface technology. Yet developers aren’t the only people involved in creating good user interfaces, and so a Visual Studio-hosted tool isn’t all that’s needed. To let designers participate more fully in the process, Microsoft also provides Expression Blend, as described next.

## For Designers: Expression Blend

A primary goal of WPF is to make designers first-class citizens in the creation of user interfaces. XAML makes this possible, but only if tools are provided that let designers work in this new world. Toward this end, Microsoft has created Expression Blend. The screen shot below shows how the Visual Studio example just described looks in this tool.



As this example suggests, Expression Blend looks more like a design tool than a software development environment, allowing its user to work in familiar ways. Yet Expression Blend is exclusively focused on creating interfaces for WPF applications. The icons on the far left, for instance, provide a designer-friendly way to work with commonly used WPF controls (although the controls are also available directly, as shown in the list on the lower right). The tool also allows graphically creating animations, transformations, effects, and more. The result of the designer’s work is expressed in a generated XAML file, which can then be read by Visual Studio.

Expression Blend is one of four members of Microsoft’s Expression family. The others are Expression Web, a tool for creating standards-based Web interfaces, Expression Design, a tool for creating vector and/or bitmap images, and Expression Media, a tool for managing digital assets such as images. Of the four, only Expression Blend is focused on creating user interfaces for WPF applications. A designer might use the others to create parts of a user interface—maybe the interface’s GIF images are created with Expression Design, for instance—but these tools aren’t specific to WPF.

# Choosing an Interface Technology

There’s no shortage of approaches to building user interfaces today. Deciding which one is right for a particular application can be challenging. To help make sense of the options, it’s useful to group them into three areas:

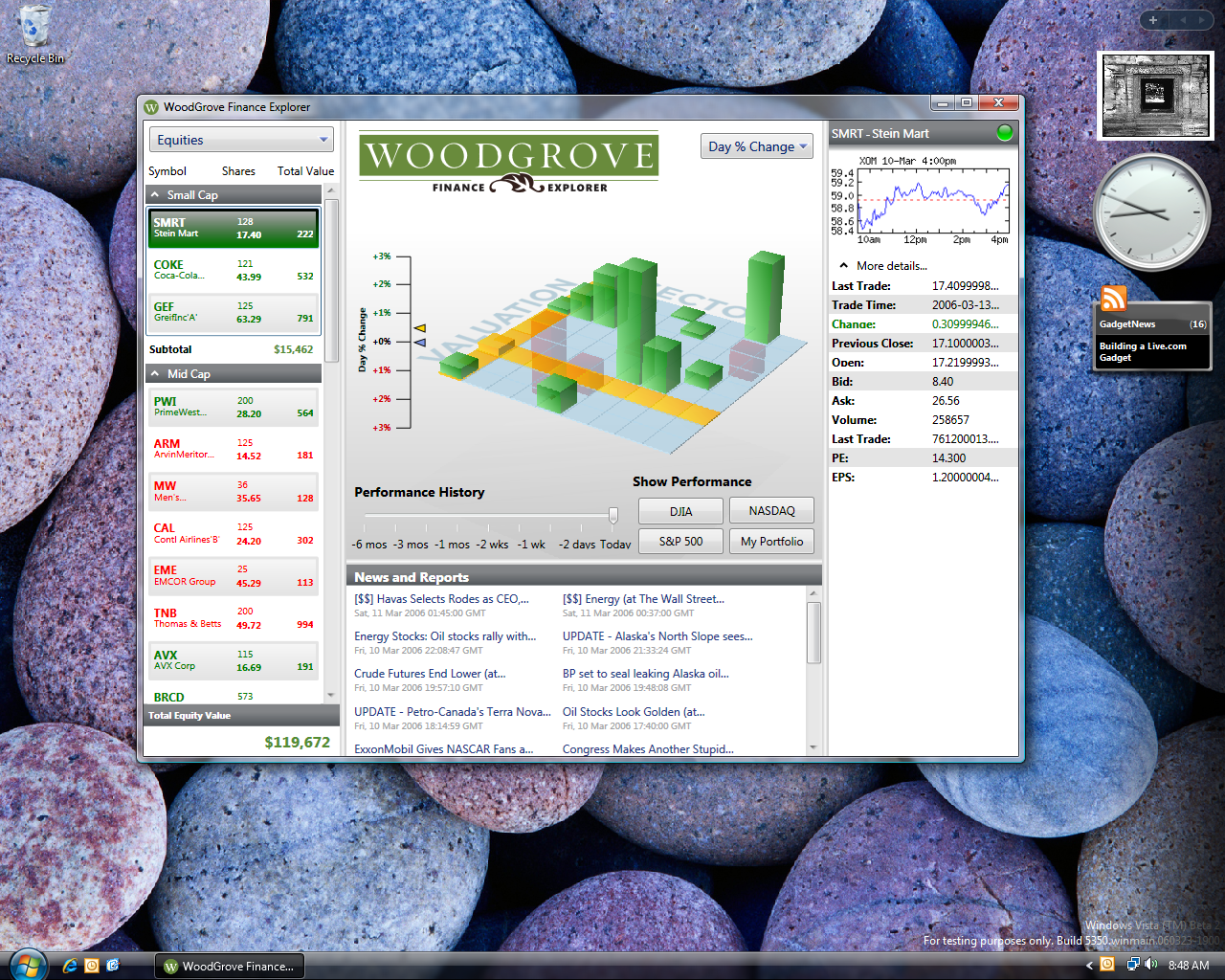
1. Interfaces for Windows applications;
2. Standards-based Web interfaces;
3. Rich Internet applications (RIAs).

It’s possible to view these as a continuum, with interfaces for Windows applications at one end, standards-based Web interfaces at the other, and RIAs somewhere in between. This section looks at each of these three choices.

## Interfaces for Windows Applications: WPF and Windows Forms

If an application will run solely on Windows, choosing a user interface technology is relatively simple. Applications that must present a more modern, immersive interface should use WPF, while LOB applications (today, at least) should use Windows Forms. Microsoft sometimes uses the phrase “Connected Desktop Experiences” to describe this interface style, illustrating both its online/offline capabilities—these applications can be at least occasionally connected to the Internet—and the essentially desktop nature of this choice.

Examples of how WPF can be used to create this kind of application were shown earlier, but as with most visual topics, more pictures are better. Here’s another example of a Windows application with a WPF-created user interface.

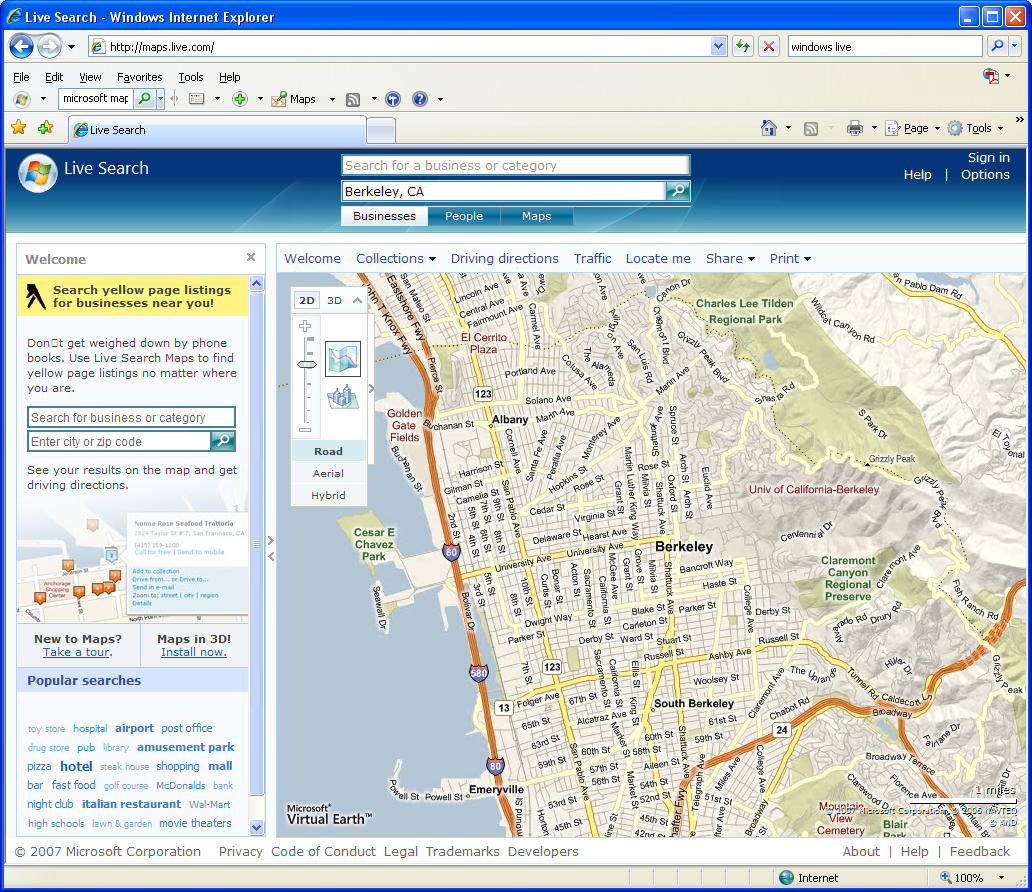


In this case, a WPF application running on Windows Vista provides financial data, much of it retrieved from remote databases via the Web. This example uses various kinds of text display, 2D and 3D graphics, and more. Creating this interface using pre-WPF technologies would be challenging; using WPF can make it significantly simpler.

## Standards-Based Web Interfaces: ASP.NET and ASP.NET AJAX

Windows applications are certainly useful, but so are Web applications. The standard technology for creating Web user interfaces is, of course, HTML, perhaps with some JavaScript. Today, the more powerful approach known as Asynchronous JavaScript and XML (AJAX) has also become popular. AJAX lets applications be much more responsive while still requiring only a browser on the client system.

Microsoft provides ASP.NET for creating Web interfaces and Internet Explorer for displaying them. Microsoft also provides ASP.NET AJAX, pre-packaged functionality that helps developers create ASP.NET applications displaying AJAX-style interfaces in Internet Explorer and other Web browsers. The screen shot below illustrates an application that provides a standards-based Web interface using AJAX.



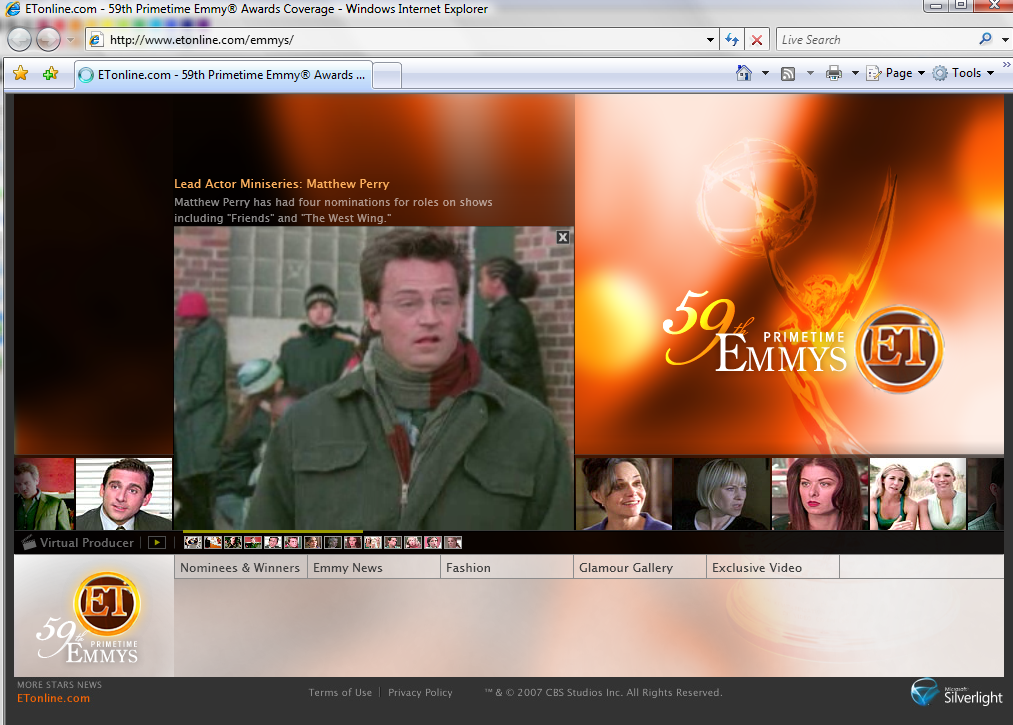
This example shows maps.live.com, Microsoft’s mapping service. It relies on AJAX to give users a responsive interface into a very large amount of data. No special download is required—all that’s needed is a standard browser such as Internet Explorer or Firefox.

## Rich Internet Applications: Silverlight

WPF lets Windows applications display modern, full-featured user interfaces. Standards-based Web interfaces let ordinary browsers display useful but simpler interfaces, perhaps made more interactive with AJAX. These two approaches represent the two ends of the user interface continuum. In between lie rich Internet applications.

Unlike desktop-focused technologies such as WPF, RIAs provide a browser-based user interface. Unlike standards-based Web technologies, RIAs offer a more powerful approach that includes animation, sound, video, and more. To support this style of interface, Microsoft provides Silverlight.

The goal of Silverlight is to provide a subset of WPF’s capabilities on diverse platforms. (In fact, the technology’s codename was “WPF/Everywhere”.) Toward this end, Silverlight supports 2D graphics, images, sound, video, animation, and text, although more advanced capabilities such as 3D graphics aren’t provided. It’s available for Windows, the Macintosh, and (eventually through Novell) Linux, and it runs in diverse Web browsers, including Internet Explorer, Firefox, and Netscape. Developers can write code for Silverlight using JavaScript, today’s most popular language for browser-hosted logic. And because Silverlight is based on WPF, it also relies on the .NET Framework and XAML. This lets developers and designers use the same knowledge and tools to create desktop applications using WPF and RIAs using Silverlight. Here’s an example of an RIA created with Silverlight:



Unlike a standards-based Web interface, a Silverlight interface requires the user to download a plug-in. This isn’t an especially onerous requirement, however, as the download is relatively small, and the user need only do it once. Once the Silverlight plug-in is installed, interfaces like the entertainment site shown above are possible. It’s also possible to combine AJAX with an RIA technology such as Silverlight. This is a good example of why it’s useful to think of user interfaces as a continuum, letting a developer choose the point that’s best for a particular application.

Describing Silverlight in the context of WPF raises an obvious question: How does Silverlight compare to WPF’s XBAPs? Both run in browsers, and both can provide audio, video, and other modern interface features. The answer is simple: XBAPs are Windows-only applications that require WPF (and thus the .NET Framework) to be installed. Given this, the primary scenario for which they’re useful is enterprise applications, situations where an application developer knows that the application will run only on Windows and that the .NET Framework will be present. Silverlight, by contrast, is cross-platform—it’s available for all widely used clients. Put simply, XBAPs allow building Windows applications that run in a browser, while Silverlight is intended for creating RIAs for all of today’s popular systems.

# Conclusion

User interfaces are a fundamentally important part of most applications. Making those interfaces as effective has possible can have measurable benefits to the people and organizations that rely on them. The primary goal of WPF is to help developers provide these benefits, and so for anybody who creates or uses Windows applications, WPF is big news.

By providing a unified platform for modern user interfaces, helping make designers more active participants in creating those interfaces, and interoperating with earlier interface technologies, WPF aims at significantly improving the Windows user experience. Some of the technologies it supplants had a twenty-year run as the foundation for Windows user interfaces. The intent of WPF is to lay the foundation for the next twenty years.

# For Further Reading

Windows Presentation Foundation:

1. http://msdn2.microsoft.com/en-us/netframework/aa663326.aspx

Microsoft Expression:

1. http://www.microsoft.com/expression

Microsoft Silverlight:

1. http://silverlight.net

Electric Rain ZAM 3D:

1. http://www.erain.com/Products/ZAM3D/DefaultPDC.asp

# About the Author

David Chappell is Principal of Chappell & Associates (www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps technology professionals around the world understand, use, and make better decisions about enterprise software.