

# WPF

**in Action**

with

**VISUAL STUDIO  
2008**

Arlen Feldman  
Maxx Daymon

MEAP

Unedited Draft





**MEAP Edition**  
**Manning Early Access Program**

Copyright 2007 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

## Table of Contents

- 1. The Road to Avalon (WPF)**
- 2. Getting started with WPF and VS 2008**
- 3. WPF from 723 feet**
- 4. Working with layouts**
- 5. The Grid panel**
- 6. Resources, Styles, Control Templates and Themes**
- 7. Events**
- 8. Oooh, Shiny!**
- 9. Laying out a more complex application**
- 10. Commands**
- 11. Data Binding with WPF**
- 12. Data Templates**
- 13. Custom Controls**
- 14. Drawing**
- 15. Drawing in 3D**
- 16. Building a Navigation Application**
- 17. WPF and Browsers – XBAP, ClickOnce & Silverlight**
- 18. Printing, Documents and XPS**
- 19. Transition Effects**
- 20. Interoperability**
- 21. Threading**

# 1

## *The road to Avalon (WPF)*

When the development team at Microsoft started to work on their brand new framework for developing user interfaces, they used the codename *Avalon*. Avalon, in British mythology, is the island where King Arthur was taken under the care of the lady of the lake--until the time when he will return. The name conjures up images of user interfaces with glimmering water and misty backgrounds.

The *marketing* department at Microsoft, whose job it is to make technology appealing to the masses, decided that a better, more appealing name would be “Windows Presentation Foundation”. Ah well. If the name is not particularly appealing, the technology certainly is.

Building user interfaces is an often under-appreciated facet of development. Both of the authors of this book have architected systems, large and small, dealing with everything from the database, security, communication, etc., all the way to the user interface. It is hard to say that one part of the infrastructure of a system is more or less important than any other. However, to the *user*, the interface *is* the application. It doesn't matter how brilliantly you build stored procedures or how carefully you have made sure your communications are secure. If the UI is poor, the application is poor. Period.

That is where Windows Presentation Foundation comes in. WPF is the latest Microsoft technology for building “rich” Windows applications. *Rich* is one of the terms used to differentiate Windows applications from browser applications. They are also sometimes called smart applications or, (usually if you are a web developer), fat applications. In this respect, WPF can be seen as the latest in the line of technologies including the Windows SDK, MFC, and Windows Forms. WPF *does* include several other technologies, which we will discuss in more detail in chapter 3, but, when you get right down to it, WPF is mostly about building Windows applications.

This book is not just about how to use WPF—it is also about how to use WPF *well*. Throughout the book we provide suggestions on best practices and good UI design.

This first chapter explains some of the motivations for building WPF in the first place, and provides an extremely high-level view of how WPF works. Before we get to that though, we want to provide some historical context, explaining some existing technologies and comparing them to WPF. This is partially to help bridge the gap between how you currently go about building UI and

how it is done in WPF. It is also because we believe strongly in the maxim that those who cannot remember the past are condemned to repeat it.<sup>1</sup>

## *1.1 The Past and the Present*

Up until now, developing for Windows and for the Web required a completely different set of tools and technologies. This is hardly surprising considering the target and genesis of each, but as times have changed, there has been a huge demand for Windows-like tools for the Web and Web-like tools for the desktop.

The results have been, shall we say, mixed.

Part of the reason for this is that it has generally involved tacking additional functionality onto existing tools and technologies. WPF, on the other hand, has the advantage of being built from the ground up with this problem in mind. It can address the needs of its target domain, while learning the lessons from all of the other frameworks and technologies that have grown up in the last few years.

Windows Presentation Foundation is primarily a technology for building Windows applications, but it also has a Web story and a document format story. These all fall under the aegis of “presentation” – presenting content to a user, whether via a rich application, a browser or a piece of paper.

The Foundation part of the name comes in, we suppose, because WPF is the base for presentation-based applications, just as WCF, Windows Communication Foundation, is the base for communication between applications. The names may be a tad on the pretentious side, but for those of us who survived the alphabet soup of Microsoft DNA, it is really not too bad.

As we said, WPF had the opportunity to start from scratch, while learning lessons from earlier technologies. Two of the strongest influences on WPF were existing Windows development methods and Web development. Influence, is the word to use if you include it to mean “lets not do *that* ever again.”

Nonetheless, in order to understand how revolutionary WPF is, it is extremely helpful to look at how Windows development and Web development came into being, and how they exist today. As a reader of this book, you are probably already somewhat familiar with the details of one or both technologies, but we try to highlight their genesis and some of the specific issues that WPF addresses.

### *1.1.1 Why Windows drawing is the way it is*

Time passes strangely in the computer world. We talk about last year’s technology being obsolete, and only fit for the rubbish heap. At the same time, particularly for programmers, we end up having to do things in certain ways because of decisions made decades ago. Windows first came out in 1985 and Windows 3.0 (the first popular version) came out in 1990. Despite all of the

---

<sup>1</sup> And it was painful enough the first time through.

various enhancements and new versions, some of that early Windows code is still floating around behind the scenes and, more scarily, the *patterns* of that code are still around, like some sort of design virus, even when the code itself has been replaced. Figure 1.1 shows a screen shot from Window 3.x. Even though it looks a lot different than Windows XP or Vista, it is quite recognizable as a forebear.

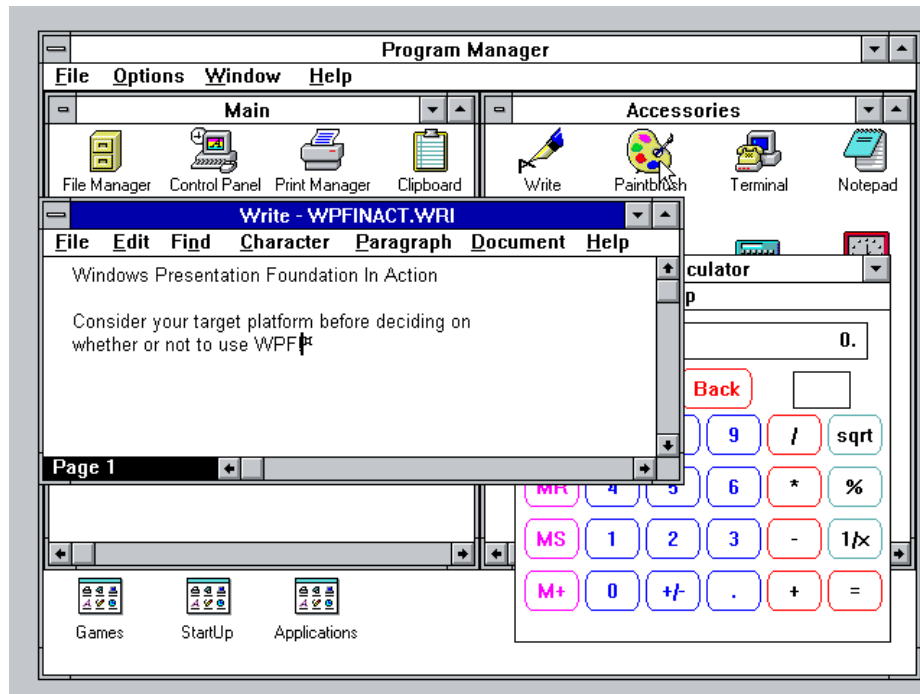


Figure 1.1 Even though Windows 3.x came out more than 15 years ago, it has an influence on the UIs of today.

Drawing/painting in Windows is one area where those original decisions have a really strong influence. Think back to the computer you were using in 1985. In the fledgling PC world, 4.77 megahertz (note the *m*) machines were all the rage and 640k was more memory than any one would ever need<sup>2</sup>. The machine on which this text is being written is about twenty-thousand times faster, and has around two-thousand times more memory (although, sadly, it takes much longer to boot than our machines from 1985). Important to our story, those fancy 640x480 256-glorious-color VGA cards didn't come out until 1988.

Even with all of those limitations, the Windows designers attempted to think ahead by making things as abstract as possible. You didn't code directly to the screen's memory, but to a *device context*, which might be the screen or might be a printer. Instead of plotting everything directly to the screen, you created brushes, fonts and pens, and worked with handles that abstracted them slightly (although woe betide anyone who had more than five declared at one time). And windows – and the controls within windows – were even represented by object-

<sup>2</sup> Bill Gates supposedly made this claim in 1981, although he denies it!

orientedish structures called classes, and referenced by pointers – HWNDs (Handles to WiNDows).

This matters today because, up until WPF, just about every drawing technology on Windows has sat on top of this design. MFC, for example, was a thin wrapper. .NET Windows Forms, which is much more robust, does a lot to hide the complexity and the arcane rules of working with the low-level libraries. Even with Windows Forms, though, the original design occasionally leaks through. Why, when you draw a line, does it end one pixel shy of where you said? Because HP was a big customer of Microsoft, and their plotters needed the pens to stop short to avoid getting a noticeable blob at corners. Why do the rules for disabled text differ from the rules for regular text? Because the developers working on the original user interface library didn't want to wait around for the GDI people to add disabled text support to their TextOut() function, so they created their own DrawText() function. It goes on and on.

The biggest legacy of all this, however, is the *philosophy* of drawing. Each window is responsible for drawing itself, and refreshing itself when asked. Drawing is done by using various methods that set the value of different pixels on a pixel-by-pixel basis. And all of the drawing is done by *your computer's processor*. This may seem like an obvious point, but it is not. In this day and age, graphics cards are extremely powerful. In fact, in an average gamer's PC, the graphics card may have more computing power than the computer itself. Yet, when you write a Windows Forms application, as Mark Boulter<sup>3</sup> says, no matter how complicated the graphics in your application, you are barely lighting up one diode on your graphics card.

Windows Presentation Foundation is an almost complete departure from this legacy. *Almost* complete because WPF still has to interact in some ways with existing technologies, and there *is* still a single HWND lurking below the surface of WPF applications. This has some implications for WPF development, particularly when interacting with non-WPF code. But, as you will see throughout the rest of this book, WPF is really a new beast, built from the ground up. It takes the best ideas from Windows drawing, Web presentation, DirectX, and modern graphics theory, with only a minimal thread tying it to the limitations of the technology and ideas that have ruled GUI development for the last 20 years.

To see exactly how far we have come, we should look at how existing Windows applications really work.

### *1.1.2 How we currently create Windows UIs*

When you look at a Windows Forms application (or an MFC or ATL application, or even one written using C and doing low-level message handling) you are looking at some number of *windows*. If you see a dialog with some text, a textbox, and a couple of buttons you are probably looking at 5 windows – one for the dialog, one for the text, one for the textbox and one for each of the buttons.

Each of those windows is responsible for painting itself, and responding to *messages*. Messages might be things like “the mouse has moved over me” or “I just got focus.” For some

---

<sup>3</sup> The Program Manager/Technical Lead on the .NET Client Team.

windows, such as buttons, Windows (capital W) knows what to do, and can do a lot of the basic handling automatically. For others that do their own thing, or do something special (for example, a button that looks like glass) the application is responsible for handling everything itself.

The fact that each window is responsible for painting itself is important. If you drag something over the top of the dialog, then move it, Windows doesn't remember what that dialog looked like. Instead it sends a message to the dialog (and to each window within the dialog) telling them each to repaint themselves. There are a number of reasons why Windows works this way, but the major one is that there was simply not enough memory to store the bits representing each separate pixel on all of the possible overlapping windows.

To be consistent with this approach, when a window wants to change the way it looks, it doesn't just repaint the bit of the screen that it occupies. For example, consider what happens when you click on a button. When the mouse is pressed, the button has to be drawn in a depressed state (or happy but pushed). Instead of just painting over that bit of the screen, this is more-or-less what happens (figure 1.2):

1. The user clicks their mouse over the button.
2. The button detects the mouse-down.
3. The button *Invalidates* the bit of the screen it occupies, telling Windows that it needs to be repainted.
4. Windows (at some point in the future) sends a *Paint* message to the application telling it to repaint part of its self.
5. The application passes the message to the button.
6. The button draws a depressed version of itself.



**Figure 1.2** To have a control change state, you have to force it to redraw itself, as with these buttons shown pre-and-during a click.

There are two important points to remember about the way Windows UI works:

- Each window is constantly redrawing itself – when it is first created, when it is covered and then re-exposed, or when something about the look and feel needs to change.
- Controls are responsible for receiving messages from Windows and handling them appropriately. These are pretty low-level – mouse moved over me, focus has changed from me, etc. Windows Forms does some wrapping to make this as painless as possible, but rest assured, it is going on, and if you want to customize behavior or look-and-feel, even in Windows Forms, you need to know a lot about it.

Finally, there is the drawing itself. When the application is told to paint something, it works with a *device context* (wrapped in a *Graphics* object in Windows Forms). This is an abstraction so that the same code can paint to a printer, to different screens, to a bitmap, etc. A good way to think of the device context is a surface on which you can draw<sup>4</sup>.

Drawing is a matter of calling various methods for things like rectangles, shapes, text, etc. However, this is much like painting in a drawing program. Once you draw a circle on a device context, it is no longer a circle, but a bunch of dots with color values. Same thing with drawing lines, dots or even text, although text is a bit special because graphics cards and printers work better if they know that they are printing text instead of dots. However, for all practical purposes, the text is just dots as far as any interface that you can get to is concerned.

If you have used fancy layout programs like Corel Draw or Visio, you know that you can click on circles, for example, and move them around. However, the drawing program is doing all the work – including determining whether your click was inside of the circle or outside of it (which can get quite complicated with more complex shapes), telling Windows to redraw the bit of the screen where the circle was and the bit of screen where the circle has been moved to.

This brings us to the final important point:

- In classic Windows applications, everything you see, as far as Windows is concerned, is a bunch of colored dots.

This is a ridiculously high-level overview of how classic Windows UIs are created, but when we talk about the way in which WPF handles drawing, it will be good to remember the three “important points” to see how different WPF is.

A lot of what programming Windows UI is about is figuring out *how* to do things. Parallel to the later development of Windows, however, was the creation of the World Wide Web. On the Web, everything was about *what* you wanted to say, with the details of presentation left to the browser. Over time, though, as the Web developed, more and more effort went into controlling *how* that content was presented.

### *1.1.3 Why the Web is the way it is*

Around the time Windows 3.0 was being released, in 1990, Tim Berners-Lee was busy creating the World Wide Web. Originally designed to author and disseminate documents, The Web has grown into a multi-purpose platform far beyond its original roots. Through many incremental advances, the web has become an application platform, although it is still fundamentally document-centric. The fact that the web *has* evolved into an application platform is a testament both to the flexibility of the system, and to the creativity of the developers who write applications for it.

---

<sup>4</sup> Although not entirely accurate. A device context has a lot more going on.

HTML, the fundamental building block of a web application, is the means by which web content is created and displayed. Early HTML was mostly *semantic*. Semantic HTML is HTML in which the tags describe the structure and meaning of the content, *not* the way in which it is presented. For example, rather than declaring the font, size, and style of text, as you might do in a word processor, you declare the text as being a header, paragraph, or a citation, and so on. The web client software then determines the appropriate font, size, and style to render. This is particularly relevant because control of the presentation of documents by the document authors was not a primary concern, and even something to be avoided.

Then something happened that turned all of this on its head. War was declared!

### *The first Great Browser War*

In the mid-1990s, seeing the potential of the web, Marc Andreessen and Jim Clark formed *Mosaic Communications Corporation* (later to become Netscape). When excitement around the web grew, it eventually caught the roving eye (Sauron-like) of Microsoft, who then entered with their own web browser, *Internet Explorer*. The increasingly tense competition resulted in a number of design decisions that would simultaneously advance and drag down web development for years.

The first casualty was the erratic and uncontrolled expansion of HTML. In order to gain favor, Netscape and Microsoft both added tags to HTML that would describe not only what a given block of text was for, but would also describe how to format the text. The most egregious, shark-jumping example of this would have to be Netscape's inclusion of the `<blink>` tag<sup>5</sup>.

At the same time all of this was playing out, developers were piling onto the HTML bandwagon. Wild-west style, people were staking claims, and figuring out what worked and using it, even if it only worked because of an accident or side-effect of that week's browser release.

### *Too late for Conformance*

By the time standards were really starting to get nailed down, it was already too late. There were too many people relying on the side-effects. The solution? Make "conformance" optional. An HTML document could violate the rules<sup>6</sup> of HTML, and browsers would simply do their best to display the document. The ability to render invalid HTML even became a selling point. Because of this, a great deal of energy today goes into browser development to make invalid documents display correctly (and why browsers to this day have things like "quirks-mode" and, we kid you not, "almost-standards-mode").

In the last few years, there has been some improvement here with the introduction of Cascading Style Sheets (CSS). With CSS, the content to be rendered (in the HTML) is separated from the instructions as to how it should be rendered (in the CSS). In addition, to making things

---

<sup>5</sup> And shame on you if you ever used it.

<sup>6</sup>The rules around the HTML document structure are defined by a meta-language called SGML, but even that wasn't true until HTML 2.0.

simpler, this approach provides significantly *more* control over how content is rendered<sup>7</sup>. This is an example of the concept of *separation of concerns*, which will be touched on throughout this book.

As with existing Windows technology, the WPF team looked thoroughly at how UI works on the Web, so it is worth spending a little bit of time talking about this ourselves.

### 1.1.4 How UI is created on the Web

The basis for any true<sup>8</sup> web application is HTML. HTML itself provides for user interfaces indirectly through a subset of native platform controls. This control support was originally provided to enable form-based documents with fillable fields. There is a limited subset of controls exposed by HTML for this purpose (figure 1.3), and this is by design. One important goal of HTML is for it to be usable across a wide variety of platforms and devices, and that goal tends to gravitate towards the lowest-common-denominator of the platforms of interest. The lack of controls can be problematic for developers though. In particular, we have seen the lack of tree controls, combo boxes and calendar controls cause many Windows developers confusion and grief when first introduced to web development.

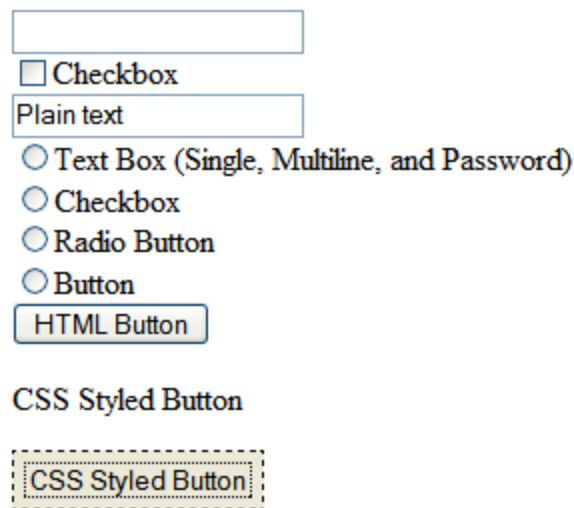


Figure 1.3 The set of available native controls in HTML is very limited. This image shows them all.

Though not essential, JavaScript is the second tool in the web UI designer's toolbox. JavaScript is an object-oriented<sup>9</sup> scripting language that enables use of the events on HTML elements and controls, and allows behavior to be overridden to an extent—providing a much richer user experience. Without JavaScript, web user interfaces are extremely limited, and largely

<sup>7</sup>For a CSS tour-de-force, visit [www.csszengarden.com](http://www.csszengarden.com). We have not seen a better site for demonstrating the power of CSS as a theming device.

<sup>8</sup>For the purposes of this discussion, a web application is one that does not rely on any platform-specific technologies such as ActiveX or Flash, which should be considered to be Windows or Flash applications delivered via HTTP.

<sup>9</sup>Few comments in the book have generated more comments than this. Yes, Javascript *really* is an object-oriented language, even though it is rarely used as such.

only support the form submission model from HTML, where controls may either contain information (textbox, radio button, checkbox), send information to a server (via form button), or abandon a view entirely (by leaving the page).

The third major tool, CSS, provides the developer with the ability to fine tune the look and behavior of the UI. CSS, or *Cascading Style Sheets*, are used to define the way content should be presented, by providing styles that are applied to various elements. The degree to which the presentation can be altered lies within the constraints of CSS itself. Prior to widespread support of CSS, web-based interfaces (and documents for that matter) tended to include an obscene number of tables to influence the layout of the UI. Like HTML, CSS is oriented around the formatting of documents, and tends to center around page layout instructions.

To summarize, there are two important points for Web UI development:

- Web UI is described using HTML and CSS. The browser then follows a set of internal rules to create the actual user interface from the descriptions given to it. This is the basis of declarative programming. This can be extremely powerful, as it greatly simplifies UI design, and allows for very dynamic user interfaces.
- A web developer does not have direct control over the user interface. Through use of HTML, CSS, and JavaScript, the web developer *influences* the user interface, but ultimately the browser has the final word. If CSS does not support a text style you want, you cannot add it to CSS. If, for example, CSS didn't support ~~strike-through~~, you are pretty much out of luck<sup>10</sup>. Contrast this with native UI development in which the developer may choose to take over virtually any aspect of presentation and behavior of a UI element.

There are web application frameworks that overcome, to some degree, many of the limitations discussed—albeit with considerable effort. The best of these frameworks typically create an entire presentation layer based on JavaScript, many generic HTML div and span tags, and extensive use of CSS. While the results can be impressive, the downside of this approach is that a tremendous amount of power is dedicated to providing a user experience on par with Windows 3.1.

Now, imagine a markup language with the simplicity and declarative style of HTML, but expressly designed for describing applications rather than documents. Imagine a framework that uses the massive power of modern GPUs (Graphics Processing Units) to provide the *next* generation user experience. Enter Windows Presentation Foundation.

## 1.2 Why Avalon / WPF

So, why did Microsoft decide that it was time to completely recreate the way in which user interfaces were built? In many ways, the last two sections provide a lot of reasons – the technology behind Windows UI is creaking. The technology behind web UI is being tortured into

---

<sup>10</sup> Baring some ugly image-based hack.

something that can be used for building applications. Both have some very powerful capabilities and concepts, but the two certainly don't play well together.

Microsoft had very big goals for Windows Vista, their new flagship operating system. Sadly, a lot of these goals have been missed, such as WinFS, the SQL Server-based replacement for the NTFS file system. As far as presentation is concerned, however, WPF delivers on most of its promises (and doesn't even require Windows Vista).

Obviously some of the impetus for a new graphical system is market-driven. Anyone who has any familiarity with Macintosh OS X knows that it is extremely slick, both to use and to code. While Apple's market-share is pretty small<sup>11</sup> in comparison, Microsoft knows a good idea when they see it. However, if keeping up with the Jobses was one of the driving factors behind the decision to create WPF, there were also a lot of specific technical goals:

- Modern Hardware – Hardware has changed a lot in the last decade or two, but taking advantage of the hardware required extremely specialized coding. WPF should make use of the underlying hardware by default.
- Modern Software Design – When the graphic subsystem of Windows was first created, things like Object-Oriented development, Patterns and garbage collection were either non-existent or bleeding edge. WPF should be built using modern software design *and* make it easy to access by programmers who *use* modern software design.
- Separating presentation from presentation logic – A goal of WPF is to be able to develop the look-and-feel of an application independently from the logic that makes it work.
- Simpler to code – These days, doing simple things is pretty easy, but a goal of WPF is to make doing complex, formerly really painful things, relatively easy as well.

We will dig a little deeper into each of these.

### *1.2.1 Taking advantage of modern hardware*

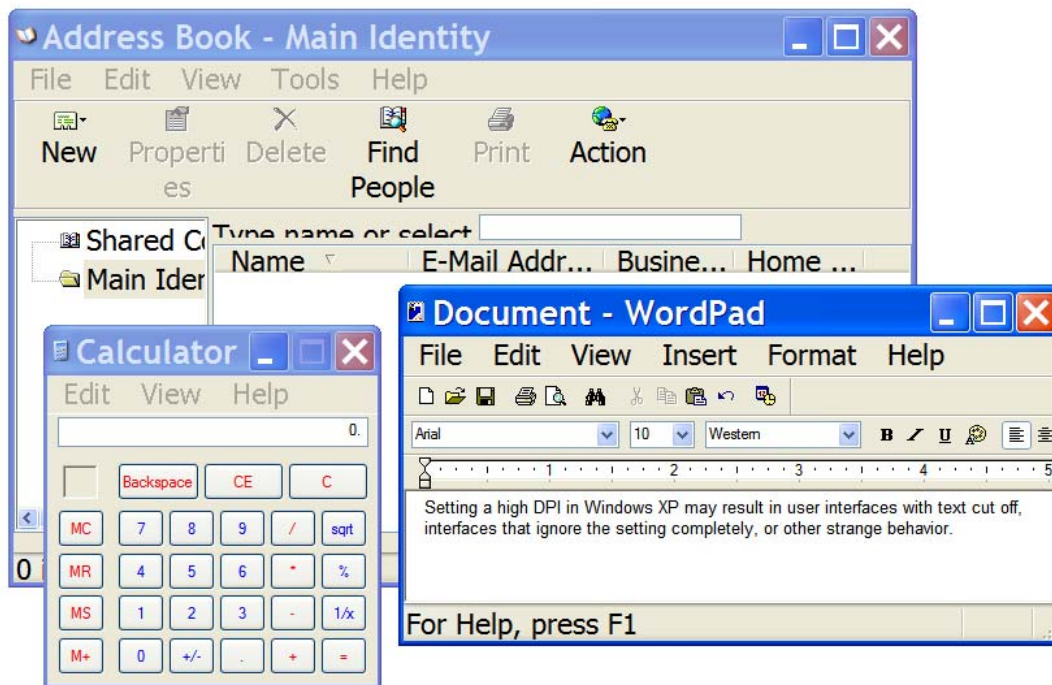
Earlier, we talked about how little advantage most Windows application take of the super-powered graphic cards in most of today's PCs. Prior to WPF, to do any sort of serious graphical UI, you were basically required to use DirectX or OpenGL. It is somewhat ironic that programming games required doing some of the most unpleasant types of programming. Making the standard library for Windows UI take advantage of current and future graphics cards was very important.

It is more than the graphics card, however. With tablet PCs becoming popular (well *more* popular), and with hand-held devices in general, handling their specialized input easily was important, which is where *Ink* comes in. Ink is the technology that provides support for writing directly on screens and converting that writing into text. WPF applications can get input from Ink, and WPF can merge Ink input with standard mouse and keyboard input so that your applications work reasonably, even if not built for tablets.

---

<sup>11</sup> Apple's handicap is its size, while Microsoft's is its size.

Modern display devices are also something that needed to be addressed. Multiple monitors are now much more common, and high-definition displays will be the norm in the near future. Even today, many machines now ship with their DPI set to 120, instead of the standard 96 DPI to which most applications have been developed. Windows Forms and other technologies play tricks to try to make things look the same when changing DPI, but there are a couple of issues with this. First, it doesn't work all that well. It is not uncommon for applications to wrap text strangely or have oddly-sized text when running at an alternative DPI (figure 1.4). Second, it doesn't take advantage of the better equipment – instead of having a sharper UI, it generally just gets smaller.



**Figure 1.4** Most applications don't handle DPI changes elegantly. These are *Windows* programs running at a higher DPI, with various poor side effects. Vista handles this better than XP, but still has issues—for example, old toolbars tend to be tiny.

If Microsoft's own applications don't handle DPI changes well – what chance do the rest of us have? To date, the most common solution has been to request users to not use the fancy new modes of their equipment – not a very popular work-around.

WPF is built on top of Direct3D, which can take advantage of the features of current and new graphic cards as they come out. As you will see, it also has a clean approach for the DPI problem. WPF uses *device independent pixels* (dips). There are always 96 pixels to an inch. If the DPI setting of the target device changes, everything is automatically scaled up or down. The main reason that 96 was used is that most current hardware uses 96 dots per inch. Also, it is easy to scale from 96ths of an inch to 72nds of an inch, which is what most fonts use.

## 1.2.2 Using modern software design

It is a little odd to describe object-oriented programming as “modern” given that it has been around since the 70s. However, it was not until much later that the concepts and technology caught up with the promise of the early days. Of course, Windows Forms is object-oriented and MFC is, well, um, well MFC has things called classes. However, both MFC and Windows Forms are wrappers on low-level technology, and the underlying mechanisms have a lot of influence on the higher-level design. Also, the non-OO stuff underneath peeks it head out rather more than is desirable. WPF was built OO from the ground-up.

The WPF API is also completely *managed*, and almost all of WPF itself is written with managed code. This is a major step-change for Windows – for the first time there is no underlying C interface that you can call directly – the managed code *is* the code. There are a number of advantages to this:

- Managed WPF code will operate extremely smoothly with applications that are also managed.
- Having a model that relies on garbage collection means that the design of the framework is not driven by the need to clean-up after resources.
- Being able to use reflection to discover behavior means powerful tools that can pick up new capabilities automatically.
- Possibly the most important benefit of being managed is avoiding the serious security issues of the older C APIs. This means that it is harder to exploit vulnerabilities, but it also means that you can safely run WPF over the web (picture this on Amazon.com) and know that the code is limited to a properly secure sandbox.

### Managed Code

Managed code is Microsoft’s term for code designed to operate with the .NET Common Language Runtime (CLR). Before managed code, a program was compiled directly to a machine-understandable format, and did what it liked. Now, programs are compiled into an intermediate language (called, cleverly, Intermediate Language) that is processed by the CLR at runtime. It is beyond the scope of this book to go into a detailed explanation of why this is a good idea, and how it works in detail, but it provides a huge number of advantages, including security, garbage collection, interoperability between languages, reflection, better multiple target support and extra dessert on Tuesdays.

## 1.2.3 Separating presentation logic from presentation

Hard as it is to admit, most programmers are not artists. That is not to say we don’t try – given six or seven hours, we can come up with a 16x16 toolbar button that is almost (but not quite) recognizable. This has become harder, now, as resolutions and user expectations have increased (figure 1.5). Microsoft has recognized this “difficulty,” and has built WPF with the explicit idea that a developer will make things work, while a UI designer will make things look nice. In WPF, the graphic designer can take the description of the UI and make it pretty without (hopefully) breaking the behavior.

Of course, the downside to this is that many companies don't bother with a UI designer, so developers will still be responsible for the look and feel of many applications<sup>12</sup>. We think it will be a long time before most companies have the resources to create the desired separation of responsibilities suggested by Microsoft. Fortunately, the default behavior for UI is *reasonably* sane with Visual Studio. The problem is that, while Windows Forms was flexible, there was a limit to how horrible a UI could be developed. With WPF, the opportunities for crimes against good taste have expanded exponentially.

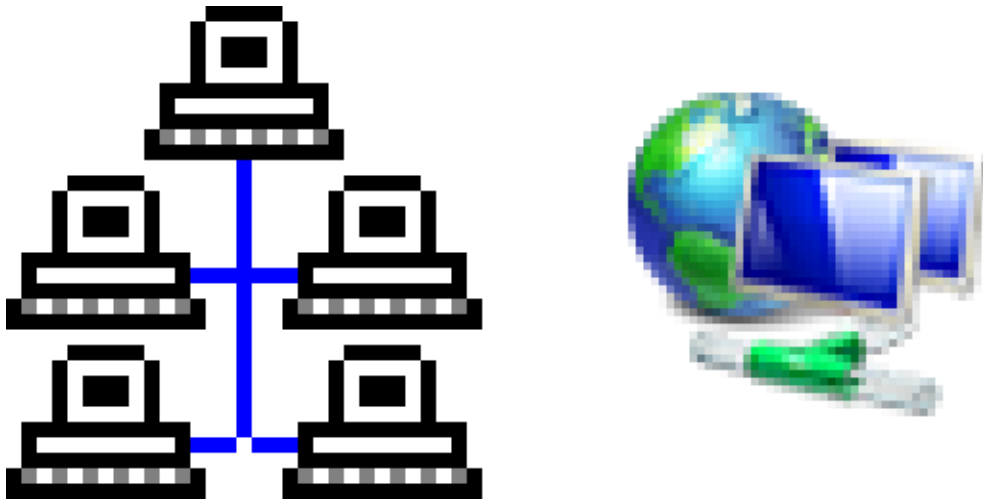


Figure 1.5 Which one requires an artist? Users have a much greater expectation for “pretty” UIs these days.

One major benefit of separating the look-and-feel from the behavior is in prototyping user-interfaces. Often, if you *do* have a graphic artist, he will create mock-ups in tools like Photoshop or Flash. They may be pretty, but they have two big problems. First, Photoshop can create pictures of *anything*. WPF is flexible, but replicating an artist's painted vision can be, shall we say, difficult. Second, the prototype is a throw-away – it has nothing to do with the real application. With WPF and XAML, however, the graphic designer can build his mock-up using tools that create real WPF UI elements. The developer can take that UI and just make it work. If things need to change, the mock-up just has to be updated, which you would do anyway because it is now part of the application.

### 1.2.4 Making it simpler to code GUIs

This is one of those sections that can get you into a lot of trouble. WPF *does* simplify the development of user interfaces. In particular it makes it easier to do things that would have previously been extremely difficult, and would have required an extensive knowledge of underlying APIs. However, in some respects, programming with WPF will make some things *harder!*

---

<sup>12</sup> Although the very design of WPF and Windows Vista may cause this to change.

The reason for this is not so much WPF itself, but because of the broader target of applications. It is still possible to build a dialog by dragging a bunch of controls onto a form, positioning them in a way that looks nice, and then going forward. But if you want that form to adapt properly when the display device is at a different DPI setting, or automatically adjust when terminology changes, or be set up properly for your graphic artist to work out the ideal look-and-feel, you will have to spend more time up front planning and setting up your UI elements.

In addition, WPF and Windows Vista will really raise the bar on what is considered “acceptable” UI. The tools keep improving, but so do the targets. For example, a few years ago (okay a decade or so ago), features such as toolbars, context-menus, and drag-and-drop, were not expected. Now they are considered basic functionality. An application that doesn’t take advantage of the richness of WPF will, in a few years, stand out starkly. This all means that we will have to do more work to provide the “basics.”

Even so, WPF *does* make it easier to do most things. There is also a great deal of tool support, both within Visual Studio and with tools like Expression Blend for graphic designers. The tools will also improve with time, and 3<sup>rd</sup> party tools are already available.

Overall, WPF has done a good job of addressing all of these goals, and a host of lesser goals. A very incomplete list of these include animation support, 3D drawing, style-support and a consistent printing model. As you will see as we move forward, there are literally dozens of other advantages to WPF.

So, what is involved in building a WPF application? In the next chapter, we will show a more complete example, but we first want to talk about what the building blocks and tools involved in creating a WPF application.

## *1.3 Creating UI using WPF*

In many respects, developing WPF UI is much more like building web UI than native Windows development. WPF development is more about “what you want” than “how do I make it work.” You start by defining the elements that make up your user-interface, and go from there. There are also two (and-one-half) different ways in which you can specify what you want. One way is by writing code to create the various elements and appropriately associate them. The other way is by using XAML. The remaining *one-half* a way is to use the designers and tools, such as those in Visual Studio or Blend.

In the next sections we will talk about how to *define* UI in WPF, then we will talk a little bit about WPF’s approach to rendering that UI.

### 1.3.1 Defining WPF UI with XAML

XAML (pronounced *zammel*) is an acronym for eXtensible Application Markup Language, and is an XML-based specification for defining user-interfaces<sup>13</sup>. Although XAML was created specifically for WPF, it is possible that, in the future, it might be used for defining UI for other things. It wouldn't be too far-fetched, for example, to see some version of XAML replace HTML!<sup>14</sup>

Using XAML, you can describe what your user interface should look like. This is technically called a *declarative* programming model. WPF will take that definition and convert it into real elements on the screen. For example, let's look at the canonical (albeit somewhat dull) "Hello, World!" example (listing 1.1):

#### Listing 1.1 – Hello, World in XAML

```
<Page
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation">
  <StackPanel>
    <Label>Hello, World!</Label>
  </StackPanel>
</Page>
```

If you would like to see what it looks like (it will be a big surprise, we assure you), there is a great utility program that comes with the .Windows SDK, called XAMLPad. It should be on the SDK menu, under Tools. Just run XAMLPad and type in the above example into the window at the bottom. You should see something like figure 1.6.

---

<sup>13</sup> *Technically*, XAML can also be used for other technologies, such as defining workflows, but it's *raison d'être* is for designing UI via WPF. In fact, XAML used to stand for eXtensible *Avalon* Markup Language, (where Avalon was the code-name for WPF) but they changed it to Application because Avalon was not going to be the public name.

<sup>14</sup> Of course, what *would* be far-fetched would be the W3C accepting a standard patented by Microsoft, and using it. . .

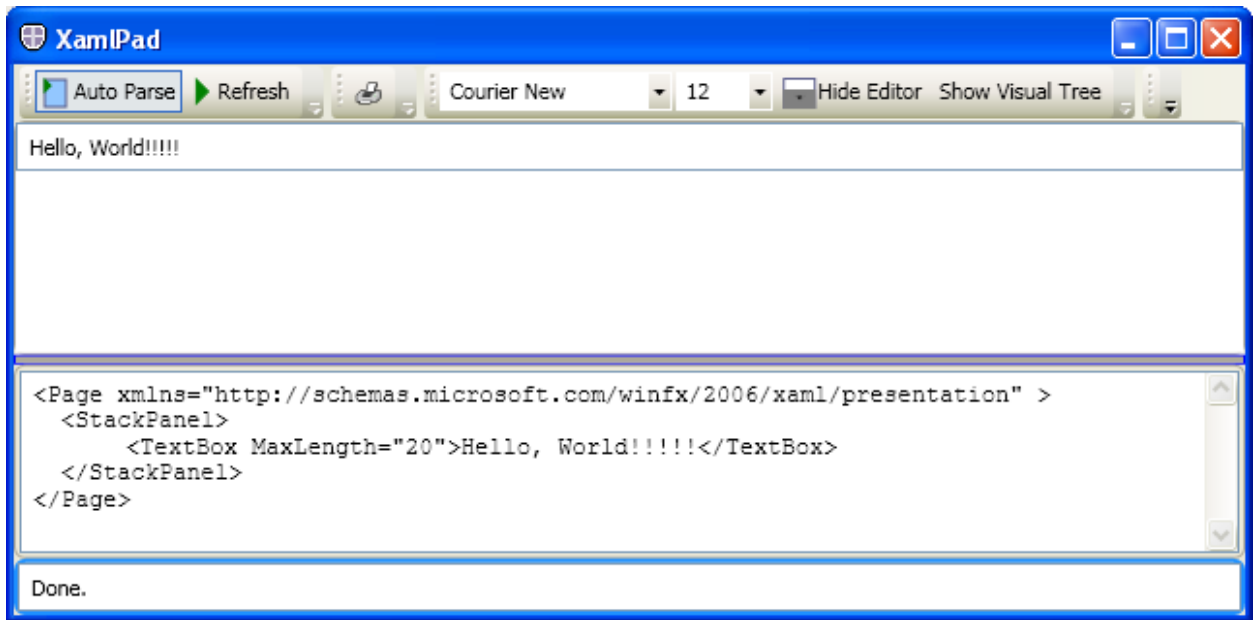


Figure 1.6 Hello, World! In XAMLPad. XAMLPad is a utility that can immediately render XAML as you type.

The XML here is validated by a schema, which is referenced in the `xmlns` attribute in the first line. The schema enforces the correctness of the XAML. It goes deeper than that, however. XAML documents are *strict* XML. Every tag (`StackPanel`, `Label`, etc.) and attribute (ex: `Margin` or `FontSize`) must correspond to a valid .NET type or property. If a tag has the wrong case, or an unknown attribute is used, the resulting XAML will not work. The benefit of this is that XAML will never fall into the black hole of quirks, incompatibilities, and haphazard development of HTML.

There are a fair number of other markup languages for designing user interfaces, such as MXML (Adobe Flex), XUL (Mozilla/Firefox), and GladeXML (GNOME/Linux). However, given the ubiquity of Windows, it is likely that XAML will quickly become the most widely deployed of these languages.

### 1.3.2 Defining WPF UI through code

You do not have to use XAML to define UI elements. You can write code to define your UI, much as you did with Windows Forms. This is the classic *imperative* programming model that we all know and love. Alternatively, you can mix-and-match – define the basics in XAML, but have some elements added in code. The following code does exactly the same thing as the XAML above (listing 1.2).

#### Listing 1.2 - Hello, World in code

```
Window window1;
Page page1;
StackPanel stackPanel1;
```

Please post comments or corrections to the Author Online forum at <http://www.manning-sandbox.com/forum.jspa?forumID=312>

```

Label label1;
TextBox textBox1;

public Procedural()
{
    window1 = new Window();
    page1 = new Page();
    stackPanel1 = new StackPanel();

    label1 = new Label();
    label1.Content = "Hello, World!";

    stackPanel1.Children.Add(label1);
    page1.Content = stackPanel1;
    window1.Content = page1;
}

```

One thing should be immediately obvious; the declarative model (XAML) is *much* more concise and easier to read. Declarative programming recognizes that domain-specific problems (such as creating a UI) generally operate in a well-known and prescribed fashion. Think about creating a form using Windows Forms. The designer creates *procedural* code that always does the exact same thing: it declares a set of controls, sets the relevant properties on them, and adds them to each other as necessary. Using XML to create a declarative UI language, the parenting of controls can be implied based on the hierarchy of the XML, and each control is declared with the relevant attributes set.

Of course, you can't do everything in XAML. For example, if our Hello, World had put a button on the screen, we could, using XAML, completely change the way the button looks and even make it do things like change color when the mouse moves over it. But if we want the button to *do* something useful when the user clicks on it, we have to add code somewhere.

Also, there are some things that can be done in either code or XAML, but are actually easier to read in code. You will see some of this as we go through various examples. The nice thing is that we have the choice.

In this day and age, however, we don't necessarily expect to have to write presentation code from scratch – rather, we rely on tools.

### 1.3.3 Defining WPF UI with Tools

We said that there are two *and-a-half* ways to build user interfaces in WPF – The first two are Declarative (XAML) or Imperative (Coding). In all probability, however, much of your work will be done using the *half* – tools such as Visual Studio 2008 and Microsoft Expression Blend – Microsoft's nifty tool for graphic designers.

Visual Studio has a WPF form designer, similar to the one for Windows Forms. However, just by switching to a declarative model, the tools can become much better and more reliable. Prior to XAML, typical UI development involved a delicate editing dance between the developer and IDE. Unfortunately, things could get out-of-hand<sup>15</sup> if the design view and code view got out

---

<sup>15</sup> Out-of-hand is shorthand for “the bloody designer ate my form again.”

of sync. Partial classes were added to .NET 2.0 largely to support the IDE writing UI and web code. The core problem is simply that the imperative model doesn't fit well with the UI designer concept. Declarative models work extremely well for this. So well that working on the UI and code independently is now not only possible, but a reasonable and recommended approach.

Now that the look-and-feel of the UI can be defined in XAML, linked only by references to the code, it becomes much easier to have different tools (such as Expression Blend) for graphic designers that let them play with look-and-feel without messing up the underlying code, and vice-versa.

Unfortunately, there is a reason why we called using tools only *half* a method. WPF is so flexible, and the tools so new, that there are severe limits to what they can do. You will probably find yourself dropping into XAML quite often. Hopefully, as the tools mature, this will become less necessary, but it unlikely that the tools will ever be able to handle *everything* that WPF can do.

No matter how you choose to build your user interface – via XAML, code or with the use of the provided tools, you still end up with a description of how your UI should look. It is then up to WPF to figure out how to present that UI and make it behave appropriately.

### *1.3.4 Who does the drawing*

As you may have noticed, XAML is a lot like HTML. Rather than specifically turning on dots on the screen when you are told that you need to repaint, you describe what you want and get out of the way. Unlike HTML, however, you have extreme control over the way in which everything is rendered.

With WPF, you describe the look-and-feel and the behavior of the UI. WPF then takes care of making all that work. Then you just have to worry about dealing with *application behavior*. If you, via XAML, say that you want a video to show up on a button whenever the user moves the mouse over it, and then have the button change color, WPF takes care of it for you. You don't have to watch for mouse move events to start and stop the video, manage the state of the button, etc.

You *can*, by the way, look for and handle the low-level messages about mouse-moves, etc., but the situations where you *have to* are rarer. WPF has an extremely powerful event model for dealing with the types of events that you *do* care about, which we will discuss later in great detail.

Another thing that WPF does, under the hood, is to work with your graphic card, offloading the heavy-lifting of drawing. This means that you can have a significantly more complex UI that, nonetheless, runs much more smoothly than a relatively simple Windows Forms application that has to do the drawing, handle input, and do the dishes, *as well as* all of the application-specific work.

We said that you describe the UI to WPF. This goes all the way from complex control-trees, right down to low-level drawing.

### 1.3.5 Pixels vs. Vectors

We haven't really got into straight *drawing* much, yet. However, when talking about classic Windows, we pointed out that you are drawing dots on a surface. If you draw a circle, that gets turned into a set of dots. Nothing in the system is aware that those dots make up a circle. This is called "immediate mode" drawing.

WPF, on the other hand, remembers what you have drawn. If you describe a circle, to WPF it *is* a circle, and can receive events and be scaled as a circle. This is part of how WPF can do what it does – it doesn't have to store each separate pixel and ask for more information when sizes change. It just has to know a center point and a radius. This is called "retained mode" drawing. Conveniently, modern graphic cards know how to draw circles too, so WPF can just pass that information to the card to do the work.

Of course, screens these days *don't* know how to draw circles. Everything eventually does get turned into dots, but it is done at the last point of contact, not the first, and that makes a huge difference. Interestingly, monitors used to be able to draw vector-based-images (although circles were pushing it). If you ever played some of the old video games like Asteroids or Battle Zone, those games did everything by constantly redrawing vectors to the screen.

## 1.4 Summary

In the section about Windows and Web UI, we brought up a number of "important points" about the ways in which each work. Now that we have also talked about how WPF does things, we'd like to revisit those points, and compare the old and new worlds:

- In classic Windows, the application is responsible for drawing itself whenever it is told to do so. In WPF, the application just describes how the UI should look, and then lets WPF do all of the drawing work. Even if you do decide, for a specialized control, to do the rendering yourself, you don't have to keep doing it. WPF will ask your custom control to render (It will literally call the *OnRender* method on your control), you do the drawing once, and then WPF handles it from there, unless you specifically indicate that something has changed, and that you want to render the control differently. This is referred to as *retained-mode* drawing, and we will go into much more detail about this when we talk about drawing.
- In classic Windows, the application receives low-level messages from the operating system. The application must appropriately handle the messages to change appearance, etc., and to determine that an event that relates to the application logic has taken place (ex: the mouse was pushed down, then released, and was still over the button at the time, so that is a click). In WPF, the low-level stuff is taken care of. You just have to worry about events that relate to application logic, and WPF provides lots of support for making that even easier.
- In classic Windows, you draw dots on a surface. The dots are just dots, and have no semantic meaning. In WPF, you draw shapes, and WPF intrinsically understands that they are shapes.

- On the web, UI is described by HTML, just as WPF UI is described by XAML. However, unlike HTML, XAML is strongly-typed and validated, so the description is reliable and consistent.
- On the web, you are extremely limited as to what you can control as far as the UI is concerned. In WPF, you have control over *everything*.
- Falling between the two sections, for both classic Windows and the web, the look-and-feel, and the behavior of the UI is tightly coupled. In WPF, you can completely divorce the two, such that a graphics designer can build the look-and-feel while a developer makes the application operate.

Any one of these points would be enough to make WPF significantly superior. In addition, there are *dozens* of other smaller advantages to WPF which you will discover throughout the book. As we go through different facets of WPF, we will revisit these points to highlight advantages (and potential pitfalls) of the WPF approach.

In the next chapter we will provide a simple demonstration of using WPF with Visual Studio, and we will also have something of a guided tour of Visual Studio 2008, and its WPF-specific features. After that, in chapter 3, we will have a rundown of all of the various technologies that make up and surround WPF.