Dedication

To my wife, Marie Séguin,

my children, Philippe Mathieu

and Michèle Audrey, and my

parents, George and Irma Jo.

Acknowledgments

There is no experience quite like writing a book. The closest thing to it I've ever done is run a marathon when I wasn't fully prepared. In both cases, although it's up to you to get the job done, you need the kindness of strangers to help you along the way.

I must first thank Ron Burk, editor of *Windows Developer's Journal*, who inspired me to write this book, although that inspiration was unbeknownst to him. I submitted to him a proposal for an article that would give a simple procedure for testing the user interfaces of Microsoft Windows programs. Let's just say that he lacked enthusiasm for my approach; he felt the presentation would be too abstract to be helpful. My first reaction was that it would take a long, boring book to cover the subject at the level of detail he suggested. My realization that I could present specific, practical Windows user interface development information in a fairly short and (hopefully) interesting book led to what you are holding in your hands. Ron was also helpful enough to give explicit instructions in "How to Write a Good Windows 95 UI Design Book" in the August 1997 issue of WDJ. I have tried to follow his advice as best I could.

I wish I could claim that every idea presented in this book is original. That would make me a pretty smart guy. Instead, the material in this book is a combination of my personal experience as a Windows programmer, the writings of many user interface experts, and the combined wisdom of all the talented user interface designers and programmers who have created the excellent programs I use every day. I would like to thank those experts whose insight into user interface design has helped me the most: Alan Cooper of Cooper Interaction Design, Tandy Trower of Microsoft Corporation, Donald Norman and Jakob Nielsen of Nielsen Norman.
Introduction

Good user interfaces matter. In fact, they matter a great deal to the success of a program. While a program’s design and technology affects its overall capability and performance, as far as the user is concerned, the user interface is the program. If the user interface isn’t good, the program isn’t good. Don’t expect users to look behind a bad user interface to see what lies beneath. They won’t.

How many Microsoft Windows software products on the market that have bad user interfaces have been commercial successes? I am familiar with only one: a popular high-end image-processing program (which shall remain anonymous). Ask anyone who uses this program how they like it and you will get the same response: love the program, hate the interface. But this program is fairly unusual. It provides complex image manipulation that is unavailable in other programs, and its awkward interface allows for batch processing that its users love. As a good rule of thumb, if there are several programs competing in a particular market segment, the winner will be the one with the best user interface.

Since the ultimate goal of software development is to make successful software products that make lots of money (this is certainly my goal), it would stand to reason that most software developers would be obsessed with obtaining as much knowledge about user interface design as possible. Surprisingly, this is not happening. I have several explanations for this:

• The software development process and those who manage it are usually focused mostly on schedules and less on software quality. For every manager clamoring for better user interfaces, there are probably a hundred who are clamoring to get the product out the door. For programmers, this means that success lies in understanding the tools and technologies that help deliver software quickly. Focusing on quality user interfaces tends to have the opposite effect.
Programmers are often encouraged not to do user interfaces. Programmers are often told that they aren’t capable of being good interface designers. In large software companies, interface designs are often done by specialists, usually user interface designers or graphic designers. Consequently, programmers do not have much motivation for improving their user interface design skills. This is in spite of the fact that the user interface constitutes a significant portion of most modern programs.

User interface design is not part of most computer science curricula, nor is it a prominent topic in most programmers’ magazines. Some programmers’ magazines have never run an article about user interface design.

Many discussions about user interface design are deemed by programmers to be irrelevant or too academic to be useful. Often they are right. Many user interface books are focused on the theoretical, not the practical. They are often jam-packed with useless mumbo jumbo.

Programmers might simply be unaware of how poor their user interfaces are. They might be aware of problems in other programs’ user interfaces but not of problems in their own.

Since a significant portion of most modern software development is the user interface, a skilled programmer needs to have a solid understanding of user interface design. Very few programming jobs today require no user interface work at all. Even if a programmer works in an organization where specialists design the user interfaces, how can such a programmer do quality work without understanding what the interface is trying to accomplish? How can such a programmer identify problems, find bugs, or make appropriate changes?

Understanding the fundamentals of Windows user interface design is essential for all programmers to create quality software.

The Audience

The target audience for this book is Windows programmers, specifically those programming Windows applications and utilities using MFC, the Windows API, or Microsoft Visual Basic. Many of the ideas I present also apply to other types of programming, such as programs designed for other operating systems, Web pages, and thin-client Windows DNA (Distributed interNet Applications architecture) programs, but I have not gone out of my way to address these specific subjects. Likewise, many of the ideas I present should be useful to software development team members other than programmers, such as managers, quality assurance testers, and technical writers, but again I haven’t gone out of my way to address these specific audiences.

The Goals

To best present the subject of user interface design to Windows programmers, I have established several goals for myself:

- To focus on specific, practical, experience-based information. I’ve tried to focus on ways to avoid mistakes that I have seen myself and other programmers make.

- To present the information in short, self-contained chapters.

- To avoid vague generalities, user interface clichés, academic theories, and other forms of mumbo jumbo.

- To not try to explain everything about user interface design, but to focus on the key concepts that every programmer should know. You are referred to other sources for additional information when appropriate.
To try to avoid "exercises" for the reader (at least the target reader). I try to provide specific examples in the text whenever practical.

This book is essentially a summary of what I believe a programmer needs to understand to create great user interfaces. These ideas and techniques have helped me tremendously in my work, and my ultimate goal is to share them in the hope that they will help you achieve great results.

What Is a User Interface?

So just what is a user interface? Interestingly, none of my favorite user interface books bother to define it. I use a fairly broad notion of user interface. To me, a user interface is much more than just the windows and dialog boxes—it is the entire user experience of using a program. In addition to the standard user interface elements, I believe the user interface also includes details like how the program installs, how the program integrates with Windows, how fast the program loads, how responsive the program is, how helpful the error messages and documentation are, and how well the program prints. Furthermore, developing a user interface is very much a team effort. I find it important to understand the roles of the other team members and how to help them help you make the best user interfaces you can.

All these factors play a role in how the user perceives a program. Consider how users react to a slow program. They don't say, "Oh, looks like this program has an inefficient internal algorithm." Rather, they say, "This program has a pokey user interface." Whether the slowness is caused directly by the user interface or not, the user interface gets the blame. Consequently, this book covers a number of subjects that relate to this total user experience that are not covered in most user interface books.

More of a Cliché than a Science

Alfred Hitchcock often said that one of his primary goals in filmmaking was to avoid the cliché. While I'm not exactly working at his level, I have made a sincere effort to avoid the standard user interface clichés and vague generalities. I have tried to avoid the common practice of describing user interface concepts in terms of analogies. There is no discussion of the KISS Principle. (And if you don't know what that is, I'm not going to tell you.) I'm not going to say that anything is more of an art than a science. Nor am I going to discuss being "user-friendly." I do include a "Keep It Simple" chapter, which is clearly titled after a vague generality, but the chapter mostly describes specific, practical techniques that a Windows programmer can use to simplify a user interface.

I've made an effort to focus on user interface techniques that are practical. The user interface techniques that I discuss are either standard techniques, techniques that I use, techniques that I used to use, or techniques that I could easily use when necessary. I don't suggest doing things that I would never do myself. I've also made an effort not to overstate the importance of things. Many things are important, but they are not all the most important. Many good user interface techniques can actually harm an interface when poorly done or overdone. I try to discuss both the right way and the wrong way and how to make sure you haven't overdone it.

Many of the user interface suggestions I make are easier to deal with early in the development process. Often this means that they are easy to implement in a new project but difficult to implement in an existing program. This is an unfortunate fact of life. If you're working on an existing program and you want to maintain a schedule, you simply cannot fix all user interface problems even if you want to. You have to choose your battles.

User interface development is a team effort, and it's important to have a realistic idea of when and how to involve the various members of the team in the process. I could say that you involve absolutely everybody in absolutely everything as early as possible, but this is clearly impractical. No one can work that way. So, for example, when I say you need to involve a technical writer early in the process, I try to give guidelines on just how early.
Finally, all rules have exceptions, so I try to explore exceptional cases whenever I can. To be practical, you can't just make blanket statements without exceptions.

**User Interface Design Is No Longer a Mystery**

I believe user interface design is no longer the mystery it once was. Today's typical Windows 98 user interface is far better than yesterday's typical Windows 3.1 interface. The behavior is much better and more powerful. The look is much cleaner and more professional. On the whole, Windows programs are more consistent because the Windows API and development tools like MFC have much higher-level user interface support built in. Today, you can create a good program framework in a matter of seconds by using the Microsoft Visual C++ AppWizard. An MFC program can provide advanced features like a print preview without any additional programming effort. These tools help make good user interfaces easier to implement, but they also raise the bar for the level of quality that users expect.

Most important, we now have a much better understanding of what a good Windows program looks like. The Windows 95 user interface really set the standard. We now understand the basic user interface components better: menus, toolbars, status bars, dialog boxes, property sheets, and message boxes. And we have a much better understanding of how user interface components fit together and their limitations. For example, we know that MDI isn't the right solution for all programs.

Since we are now creating user interfaces at a much higher level, your job should be easier. You can now create a good user interface by understanding the basic concepts of user interface design, understanding the Windows standards, understanding the basic user interface options, and making the right choices to help your users get their work done. There are many good Windows user interfaces that you can use as a model. What you don't have to do is design everything from scratch—there's little you have to invent. You don't need to have elaborate user testing facilities with two-way mirrors and videotaping. You don't need to perform elaborate user testing to determine the best way for a user to set the properties of an object. If you use the standard controls and follow the Windows standards for user interface design, most of this hard work has already been done for you. Many user interfaces have withstood the test of time. We already know they work.

One of my important goals in this book is to help you fully understand the Windows user interface components and how to make the right choices.

**User Interfaces for Users**

When you make user interface decisions, you need to remember that the ultimate goal is to help users get their work done. In fact, the best user interfaces go out of their way to help users get their work done. This is why they are called "user interfaces" and not "program interfaces."

While this observation might not seem especially profound, programmers often fail to make decisions based on what is best for the user. Rather, they often make user interface decisions based on what is best for themselves. Programmers choose to add inappropriate features because they are cool, because they are fun to program, or because they are easy to implement. Programmers rarely add inappropriate features because they are trying to help users get their work done.

To create good user interfaces, you have to become an advocate for users and put their goals ahead of your goals.

**Programmers Can Create Good User Interfaces**

In his otherwise excellent book *About Face: The Essentials of User Interface Design*, Alan Cooper flatly states that programmers are incapable of creating good user interfaces because they are too concerned about things like efficient algorithms. It is true that programmers have created some truly terrible user interfaces.
However, I reject the notion that programmers are incapable of creating good interfaces because of some kind of innate inability. I'm just as concerned about efficient algorithms as the next guy, but it hasn't impaired my ability to create usable software.

Rather, I believe the problem is much more fundamental. Many programmers are simply not trained in the fundamentals of good user interface design. User interface design is not a standard part of the typical computer science curriculum (at least it wasn't for me), and it's not a popular topic in programmers magazines. Programmers are fed a steady diet of "how-to" information that they need to get their work done, and interface design has not traditionally been presented in this manner. Without a solid understanding of the fundamentals, neither programmers nor anyone else can create good user interfaces. You cannot expect someone to have expertise in a skill they have never been trained in.

I believe it is important to understand that, like it or not, programmers are the ones doing most of the user interface work. Few organizations have specialized user interface designers. (For me, most of the projects I have worked on have been user interfaces with very little true core code. If you removed the user interface work from my projects, there wouldn't be much left.) Thus, there's a significant need for programmers to improve their user interface skills. And I believe programmers, with the right information, can create good user interfaces.

**Controversial Ideas**

Developing user interfaces is a highly personal subject. There is no one way to do it, nor is there one right way. That said, most of the advice I give in this book shouldn't be especially surprising since similar ideas have been described in other software development sources. However, in addition to believing that programmers can create good user interfaces, I have some other ideas that are not widely accepted, specifically:

- Users are not designers, so you cannot expect them to design your interfaces for you. Users can give you valuable information and they can test your program to tell you about its usability problems, but you will have to do the designing. Users can't turn a poorly designed program into a well-designed program. At best, they can turn a poorly designed program into a usable, poorly designed program. I believe the best way to design software is to work with users and other team members to create a program vision. You then use this vision as a decision-making framework.

- Prototyping can be very effective when done right, but it is easier to do wrong than most people think. Most prototyping efforts I've seen have been a waste of time. A couple of them were catastrophic disasters. To prototype effectively, you need to have a specific problem that needs to be solved; you need clear, realistic goals; and you need to know how to do proper user testing to get good feedback. Otherwise, don't waste your time. Finally, there are many types of design problems that prototyping cannot find.

- Programmers are primarily responsible for quality. Programmers can and should test their own user interfaces. The biggest hurdle is testing the program as a user by performing real tasks that users perform. Performing the same tests you used during debugging won't do the job. I've presented many practical techniques for testing user interfaces that any programmer should be able to use.

- User testing can be valuable, but you need to do programmer testing and QA testing first. User testing is more realistic than these other types of testing, but you can find many usability problems by using more cost-effective techniques.

Well, that's it. That is as controversial as I get. You might not agree with these ideas, but I'll do my best to talk you into them. These ideas reflect what I have experienced personally, and they have worked well for me. I have to call it as I see it.
A Word About Microsoft Software

I feel strongly about presenting practical, experience-based information in this book. Since it is important to understand why good programs are good and bad programs are bad, it is necessary to present real-world examples of both good and bad user interfaces. To make such examples practical and experience-based, I thought it would be a good idea to focus on products that I have extensive experience with and that most programmers are likely to have experience with as well. These would have to be programs I have used for years, not hours.

In my original proposal for this book, I chose several Microsoft products as examples of programs with good user interfaces and several non-Microsoft products as examples of programs with bad user interfaces. But when Microsoft Press agreed to publish this book, I immediately realized that I had a problem. It would be a little too cute for a book about user interface design from Microsoft Press to use Microsoft products for good examples and non-Microsoft products for bad examples. This would give my book a bias that I did not intentionally set out to give it.

My solution to this problem was to feature Microsoft products for all examples, both good and bad (with only a couple of exceptions). This approach has balance, and I am happy to report that Microsoft Press made no effort to censor any of my criticism of Microsoft products. I realize that there are many other companies out there creating excellent user interfaces, but unfortunately these are not products I use on a daily basis.

Chapter 1

Know the Standards

Knowing, understanding, and applying the standards for Microsoft Windows user interface design allows you to create programs that the user already knows how to use. It also makes your programs more familiar and comfortable and ensures that they integrate well with the Windows environment. If you are doing user interface development, there is simply no excuse for not knowing the standards. Your knowledge of the standards is the absolute minimum requirement for programming user interfaces.

What Are the Standards?

I consider the following sources to be the standards for Windows user interface design.

Designing for the User Experience

These guidelines (formerly called The Windows Interface Guidelines for Software Design) cover user interface design principles, basic design concepts, visual design, the Windows environment, mouse and keyboard input, interaction techniques, windows, menus, toolbars, dialog boxes, controls, message boxes, online help, and Windows environment integration. This book is the single best source of Windows user interface information there is.

Microsoft Manual of Style for Technical Publications

This manual is a valuable reference that can help you make sure you’re using the right terminology in your interfaces. While this manual is geared primarily for technical writers and not programmers, much of its
information applies to the text in user interface elements such as dialog boxes, message boxes, menus, tooltips, status bars, and online help. Since text is a vital part of every user interface, using the right terms is critical to the clarity and consistency of the interface.

While I discuss this subject in detail in Chapter 3, note that the terminology used to describe many common Windows elements is different for programmers than for users. (Don't ask me how this happened.) For example, while programmer documentation refers to combo boxes, radio buttons, and subdirectories, user documentation refers to boxes ("combo" is never used), option buttons, and folders.

The Designed for Microsoft Windows Logo Requirements

The Designed for Microsoft Windows logo requirements are a set of requirements your program must satisfy to receive the Designed for Microsoft Windows logo. The specific requirements vary depending upon the type of program, but they are all geared toward making sure that your program integrates well with the Windows environment. They also help you provide optimum usability and ensure a consistent, accessible user interface.

Among the requirements are that your program must

- Use the Win32 application programming interface (API) and be in the Portable Executable (PE) format.

- Run in all current versions of Windows and Microsoft Windows.

TIP

You should review the Designed for Microsoft Windows logo requirements, even if you could care less about obtaining the logo.

Take the Quiz

While I clearly believe in the importance of knowing the standards, I want to convince you as well. But rather than trying to persuade you by begging and pleading, I want to give you a small quiz to test your user interface knowledge. These questions should be a snap if you have read Designing for the User Experience and the Designed for Microsoft Windows logo requirements.

Question 1—Ellipsis

The menu command to display a program's About box should have an ellipsis, as with "About My Program…"

a) True
b) False

Question 2—Static Text

Static text controls are the best way to display uneditable text in a dialog box.

a) True
b) False
Question 3—Message Boxes

When asking the user a question with a message box, the best approach is to ask the question, provide Yes and No buttons (and perhaps a Cancel or Help button, if appropriate), and the question mark symbol.

a) True  
b) False

Question 4—Gray Backgrounds

A gray background indicates that a control is disabled.

a) True  
b) False

Question 5—Determining Disk Space

The best way to report to the user the amount of free hard disk space is to use the GetDiskFreeSpace API function to determine the available disk space.

a) True  
b) False

Question 6—Invisible Caret

There is never a need for a window to change the position of an invisible caret.

a) True  
b) False

The Quiz Answers

The answer to all of these questions is false. Here are the details:

Question 1—Ellipsis

Contrary to common belief, an ellipsis does not mean that a dialog box follows. Rather, it means that more information other than a simple confirmation is required to carry out the command. Since no additional information is needed from the user to display the About box, this command should not have an ellipsis. Interestingly, the Microsoft Foundation Classes (MFC) application AppWizard incorrectly provides an ellipsis with the About box menu command text. Even Charles Petzold himself incorrectly states: "The programmer indicates that a menu item invokes a dialog box by adding an ellipsis (...) to the menu item." Other common commands that should not have an ellipsis are Properties, Settings, Preferences, and Help Topics.

Question 2—Static Text

While static text is the best control for dialog box labels, read-only edit controls are much more useful for displaying other types of text. For example, the Windows Explorer Properties dialog box uses read-only edit controls for all the property text. Using read-only edit controls is more useful than static text since it allows the user to select the text, copy the text to the clipboard, and scroll the text if it is wider than the edit control.
Question 3—Message Boxes

The question mark symbol (MB_ICONQUESTION) is no longer recommended for message boxes because it is now used consistently within Windows 98 to signify context-sensitive help. In its place, the warning and critical message types are recommended. Note that using Yes and No buttons is much better than using the OK and Cancel buttons for questions since it is easier for the user to figure out what to do no matter how the question is phrased.

Question 4—Gray Backgrounds

Gray backgrounds are used to indicate that an object is read-only. Gray text is used to indicate that an object is disabled. By the way, this question was supposed to be the easy one.

Question 5—Determining Disk Space

The GetDiskFreeSpace API function is designed for hard disk volumes smaller than 2 GB, which is the limit with FAT16. With the new FAT32 file system, programs must use GetDiskFreeSpaceEx to handle hard disk volumes that are greater than 2 GB. (See the Designed for Microsoft Windows logo requirements, "Do Not Assume a Hard-Drive Size Limit of 2 GB.")

Question 6—Invisible Caret

To allow accessibility aids such as the Windows Magnifier to work properly, programs must provide Windows a notification of the current keyboard focus. While this notification is performed automatically with the standard Windows controls, custom controls that move the keyboard focus within themselves (such as a custom list control) need to notify Windows of the current keyboard focus. While there are several ways of doing this, often the easiest way is to create an invisible caret and move the position of the caret within the control. (See the Designed for Microsoft Windows logo requirements, "Provide Notifications of the Keyboard Focus Location.")

So What?

What's the big deal? After all, if everything else in your program is correct, having an ellipsis in the About box menu command isn't going to make your program difficult to use, is it? While that is certainly a valid conclusion, the premise is questionable. If you don't know the standards, what are the chances that everything else in your program is going to be correct? Bad user interfaces are usually not bad because they have one or two glaring problems. They're bad because they have dozens of small mistakes that can contribute to or directly result in the following problems:

- Inconsistency with other Windows programs and with the user's expectations
- Nonstandard mouse and keyboard input
- Nonstandard windows or dialog boxes that require nonstandard responses from the user
- Nonstandard controls or standard controls that behave in nonstandard ways
- Poor integration with the Windows environment
When to Violate the Standards

- Confusing and hard-to-understand error messages
- Inadequate accessibility
- Inadequate user assistance

Helping you make sure that your program conforms to what the user expects is exactly what the standards are for. Complying with the standards allows you to create programs that the user already knows how to use. For example, if you use a standard list box in a standard way, there's no need to document how to use it. And best of all, you don't have to do any of the really hard work—all of the design work and user testing has already been done for you. When you create programs that don't conform to the standards, you're pretty much on your own.

TIP

Conforming to the standards makes your job easier, not harder.

I appreciate the fact that Designing for the User Experience is a fairly large book and can be difficult to read at times. However, it is effective as a reference and you really don't have to memorize it. Just have a copy within easy reach, know what's in it, know what you already know, know what you're less familiar with, and know when you need to refer to it. That's all!

When to Violate the Standards

While the guidelines specifically state that compliance is optional, I believe they say this just to be nice. You should conform to the standards unless you have an amazingly good reason to break them. Here are some good reasons for not conforming to the standards:

- You are trying to advance state-of-the-art user interface design, and you really know what you are doing. In this case, you should also have a significant budget for user testing.
- You are creating a radically new kind of program.
- Your program has extraordinary requirements.
- You are creating a game or multimedia program that is designed primarily to entertain, and you feel that conforming to the standard appearance is too boring.
- You have decided that achieving another goal is more important, and you are making a well-thought-out trade-off.
- You made an honest mistake and will fix the problem as soon as possible.

I consider the following to be really bad reasons for not conforming to the standards:

- You don't know what the standards are.
- You prefer to make up your own standards as you go.
- You find conforming to the standards too much trouble.
- You found a really cool ActiveX control on the Internet and want to use it in your program somehow, even though it doesn't conform to the standards.
People often note that Microsoft itself breaks the standards all the time. However, note that they do this to advance the state-of-the-art of Windows user interface design. Also note that they have a substantial budget to design and test new user interface designs. Microsoft typically unveils these new designs with their Office and Internet Explorer products and eventually incorporates them into the standards and other products. If you want to follow Microsoft's latest user interface designs and you have a good reason to, that is fine by me. Typically I plan to adopt them too, but I'm usually not in a big hurry to do so. I prefer to give myself, my users, and my tools (especially MFC) a chance to catch up.

**TIP**

Violate the standards to go forward, not backward.

**Recommended Reading**


  Presents a good discussion of the importance of standards and when to break them.

- Microsoft Corporation. The Designed for Microsoft Windows logo requirements.

  You can find the logo requirements on the CD-ROM included with this book and in the MSDN Library.


  While you don't need to read this book from cover to cover, you should read the first two chapters and use the remainder as a reference. The first two chapters give a concise overview of the fundamental user interface design concepts. Definitely worthwhile.


**CD-ROM Resources**

The CD-ROM included with this book contains the following resources related to this chapter:


- The Designed for Microsoft Windows logo requirements.

**Chapter 2**

**Read Other User Interface Design Books**

I know it's a bit odd to start off a book about user interface development by saying that you should read other books about user interface design. What I mean, of course, is that you should read other books because it is simply not possible for an author, including me, to present in a single book everything that one should know about user interface design. I clearly do not intend to try. In fact, I believe in this idea so strongly that I've included a Recommended Reading section at the end of each chapter. To further improve your interface designs, make sure you take a look at these other sources of relevant information.
In this book, I present user interface design from the Microsoft Windows programmer's point of view. Reading this book is an excellent start, but you should try to get other points of view as well. It's important to understand user interface design from the point of view of users, user interface designers, visual and graphic designers, and designers in other fields. I also find it useful to look at user interface design from the point of view of other platforms. Even though the fundamentals of good user interface design are largely platform-independent, looking at other platforms will give you a fresh perspective. Don't avoid a user interface book just because it isn't Windows-specific. Looking at other platforms can give you ideas that otherwise might never have occurred to you. In fact, I find books that discuss the Macintosh user interface to be far more relevant to my work than books that try to apply to all platforms.

In the remainder of this chapter, I will discuss some of my favorite books on user interface design. These are the books that I have found to be the most useful, and I will present them in order of their usefulness to me. I find these books to be practical, relevant, insightful, and worthwhile. Of course, your needs might be different than mine, so you might draw other conclusions. The Recommended Reading section in this chapter presents user interface-related books that I believe are worthy of honorable mention.

Alan Cooper's *About Face*


This book takes a comprehensive look at user interface design. It covers the goals of user interface design, how to deal with various implementation problems, the behavior of programs, interaction techniques, and the major user interface elements, such as windows, menus, dialog boxes, message boxes, toolbars, and controls. While the treatment of the subject matter ranges from practical to philosophical, Cooper discusses all of these subjects with incredible insight. His book includes more good ideas and observations about user interfaces and user interface design than any other book I have read. Best of all, the book is enjoyable to read and full of excellent examples.

A number of Cooper's chapters are masterpieces of user interface analysis. For example, Chapter 5, "Idioms and Affordance," gives an excellent analysis of metaphors vs. idioms. Chapter 11, "Orchestration and Flow," includes a great discussion on how to keep users focused on their work and not on the program. Chapter 12, "Posture and State," defines the different types of programs (postures) based on how they are used and how their type influences the appearance on the screen (state). Chapter 13, "Overhead and Idiocy," discusses how to recognize and eliminate unnecessary tasks. Chapter 21, "Dialog Boxes" discusses when and when not to use dialog boxes and compares the different types of dialog boxes. And Chapter 23, "Toolbars" compares menus to toolbars and makes it clear that menus are for teaching and toolbars are for convenience. Each of these chapters is a must-read.

Cooper is not shy about taking controversial views on many user interface subjects. Happily, I agree with most of them. For example, he thinks that user interface metaphors are overrated: "There is an infinity of idioms waiting to be invented, but only a limited set of metaphors waiting to be discovered. Metaphors give first-timers a penny's worth of value but cost them many dollars' worth of problems as they continue to use the software." He believes that dialog boxes should be avoided: "Dialogs, for good or ill, interrupt the interaction and make the user react to the program instead of driving it." "A dialog box is another room. Have a reason to go there." "Dialogs break flow." And Cooper knows that usability testing is not a substitute for design: "The chief drawback of usability is that it sidesteps actual design. The process of testing is very different from the process of design. Design springs directly from the knowledge of goals. Usability derives from specific objects." Good stuff.

This is not to say that I agree with all his ideas. I find some of them downright loopy. For example, Cooper suggests that companies should establish a pronounced proprietary look for their programs: "As a software publisher, you can personalize your entire application. By putting identifying marks on all of the components
of the program, you help in creating a branded product." He suggests that a good alternative to negative audible feedback (that is, beeping) is positive audible feedback: "Our programs would be much friendlier and easier to use if they issued barely audible but easily identifiable sounds when user actions were correct. The program could issue a soft 'coo' every time the user entered valid input to a field." Cooper also argues that it is better to allow users to enter questionable data into a database than to bother users with error messages when they enter questionable data: "Data integrity is a good concept on paper, but it dumps the burden of entering correct data in the user's lap rather than when—and if—the correct data is actually needed." He also proposes that programs should indicate that a drag operation has been cancelled by displaying a giant red "Drag Cancelled" stamp in the middle of the screen and by making a "thump" sound effect. Sorry, I don't think so.

I also think there are problems in the book's presentation. There is a bit too much attitude, too much ranting, and too much harsh language. I also believe there are too many unsupported opinions and too many analogies. Despite its problems, however, what this book has is understanding and insight—and plenty of it. Even Cooper's worst ideas are worthwhile reading in that they are thought-provoking. The goal of reading this book shouldn't be to adopt Cooper's ideas without question but to let Cooper widen your thinking about user interface design and help you develop a better understanding of the fundamental user interface components. In my case, this book has helped me understand user interface design more than any other.

This is a must-have book.

**Microsoft's *Designing for the User Experience***


I discussed this book in Chapter 1, so I'll keep my comments short here. I believe that this book is the single best source of Windows user interface information, and I refer to it often. Then why do I rank it second? While *Designing for the User Experience* does an excellent job of defining standard Windows appearance and behavior, it doesn't give you the understanding and insight that Cooper's book does. *Designing for the User Experience* explains what and how, and *About Face* explains why. I believe mastering why is ultimately more important.

This is a must-have book.

**Virginia Howlett's *Visual Interface Design***


This book presents user interface design from a visual designer's point of view. Virginia Howlett is a former director of visual interface design at Microsoft and was responsible for the team that designed the visual interfaces of Windows 3.1, Windows 95, and Microsoft Windows NT. Covering the fundamentals of graphic design and teaching you how to recognize common user interface mistakes, the book presents many important design principles, such as simplicity, balance, restraint, scale, contrast, focus, emphasis, grouping, and grid, as well as valuable information about using 3D, fonts, color, and icons.

Howlett's book is especially important if you are creating highly visual, consumer-oriented programs, such as home, multimedia, or children's programs. While you most likely will be working with a graphic designer on such projects, it is still important to understand what this type of interface is trying to accomplish. Even if you are not working on multimedia-style programs, this book gives several excellent examples of common user interface mistakes that programmers are likely to make, such as poor use of color and contrast, poor use of 3-D, and screen design that lacks focus and balance.
The chapters on graphic design principles (Chapter 4, "Universal Design Principles," and Chapter 5, "Graphic Information Design Principles") as well as the chapters on visual design elements (Chapter 10, "Color"; Chapter 11, "Icons and Imagery"; and Chapter 12, "Fonts") are must-reads. The book also presents two user interface makeover examples that are outstanding (Chapter 13, "Interface Makeovers"). Short and to the point, they feature problems that programmers are likely to create. One of the examples is the makeover of Microsoft Encarta from the original version (apparently designed by programmers) to the current version, which is visually vastly superior. Finally, the last chapter (Chapter 15, "Common Pitfalls and How to Avoid Them") presents several useful tips on how to avoid common user interface problems.

One of the reasons this book is so effective is that it is a quick read. The ideas are well presented, and you're always given the essential information you need, typically with an excellent example screen shot to make the point. The subjects are not beaten to death, and there is only a tinge of mumbo-jumbo—you might be put off by the discussion of contextual inquiry, for example. This book would not be nearly as effective if it were larger. Page per page, it is possibly the best user interface book out there.

If you don't like the look of your programs, this is the first place to turn. In the author's own words: "Obviously, all of these topics could be treated in more depth, but my goal is to give you just enough information to inspire you to make better, more beautiful, Windows products." Enough said.

This is a must-have book.

**Donald A. Norman's The Design of Everyday Things**


This is an excellent user interface design book, but it does not directly discuss software user interfaces. By "everyday things," Norman means very common objects such as doors, telephones, refrigerators, cars, VCRs, and faucets. He only occasionally discusses computers. While at first this book might seem irrelevant to software user interface design, in fact it is nothing but relevant. Nearly every key idea applies directly to software user interface design with surprisingly little effort. One could argue that since the time this book was written, the computer has become an everyday thing. With today's sub-$1000 computers and with the Internet seemingly everywhere, formerly "nontechnical" people are using computers in their daily routine, and software user interface design needs to reflect this fact. Even my two-year-old son Philippe is able to turn his computer on and off, insert CD-ROMs (using caddies!), and start and use programs all on his own.

You need to read this book for two reasons: first, to understand basic user interface design concepts, which are explained extraordinarily well in this book; and second, to understand how to make complex software work as simply as an "everyday thing" by eliminating unnecessary complexity. Whereas Howlett's book helps you understand how a Windows program should look, this book helps you understand how a Windows program can be made easy to learn and use. This book has incredible insight.

Originally titled The Psychology of Everyday Things—the title was changed to make the book more appealing to a broader audience—Norman's book could be accurately titled How Users Learn to Use Everyday Things. For software user interfaces, human psychology forms the vital link between what the user sees on the screen and how the user decides what to do (ultimately determining how the user learns to interact with the user interface). It is therefore valuable to understand this psychology—how users understand visual elements, explain them, remember them, form relationships between them, become confused, and make mistakes. Knowing this information is fundamental to good user interface design. Since the book focuses on simple, practical examples of everyday objects and avoids academic psychobabble, it is enjoyable reading for anyone.
Many design principles are examined throughout the book, the most important being visibility, affordance, natural mapping, constraints, conceptual models, and feedback. User interfaces that implement these principles well are easy for the user to learn, understand, and use. Visibility gives the user the ability to figure out how to use something just by looking at it. Affordance relates to the perceived and actual properties of an object that suggest how the object is to be used. Natural mapping creates a clear relationship between what the user wants to do and the mechanism for doing it. Constraints reduce the number of ways to perform a task and the amount of knowledge necessary to perform a task, making it easier to figure out. A good conceptual model is one in which the user's understanding of how something works corresponds to the way it actually works. This way the user can confidently predict the effect of his actions. Lastly, feedback indicates to the user that a task is being done and being done correctly.

Here's another interesting principle that Norman discusses: If a control needs a label, the design of the control has failed. While labels are normally fine in software user interfaces, clearly a control that makes sense without a label is better than a control that requires a label. Specifically, edit boxes, which usually require labels to make sense, are not as effective as more constrained controls, such as combo boxes and slider bars, which are easier to understand without labels than edits boxes are. In terms of everyday things, a door handle is an excellent example. If a door looks like it should be pushed, people are going to push it. Only after a door fails to open after being pushed a couple of times do users bother to read the label that instructs them to pull.

I had many revelations while reading this book; I'll share just one. Many products are advertised as not requiring programming. What exactly does this mean? This distinction is somewhat arbitrary since you could argue that just about everything is "programmed" in some way. Doors are effectively programmed to open and close. Washing machines are effectively programmed to handle differing load sizes, colors, and fabrics. Certainly, using any sort of computer software is a form of programming. However, a task can be said to be performed without programming if there is a simple, visible, and direct mechanism that maps from what the user wants to do to how the object does it. Each control on the object has a single obvious function and provides some sort of feedback that the task is being accomplished correctly. On the other hand, a task requires programming when there is no direct relationship between what the user wants to do and how the object does it. For example, a digital watch requires programming and an analog watch does not. While it is possible to design a digital watch that doesn't require programming, its controls would support only a few functions. The reason digital watches require programming is to provide many features with a small number of controls. In the end, if given a choice, most users prefer to perform their work without programming.

This is a must-have book.

Bruce Tognazzini's *Tog on Interface*


Bruce Tognazzini wrote this book while he was the human interface evangelist at Apple Computer. The book consists of many letters from developers and his responses (taken from *Apple Directions*, Apple's developer news magazine), techniques for the user interface design process, and basic design principles. The specific subjects covered include visibility, user-centered design, perceived stability, managing menus, managing dialog boxes, working with other team members, balloon help, agents, feedback, and dealing with user errors. Its coverage of user testing ("on the cheap") is the most concise treatment of the subject.

While this information is mostly Macintosh-specific, much of it applies to Windows with little translation. But for a modern Windows programmer, the details are mostly irrelevant. A Windows programmer should not read this book for specific user interface techniques. Rather, the reason you should read this book is to gain insight on how to think about user interfaces, including the underlying principles of user interface design.

This book is especially dear to me because it was the first real user interface design book I ever read. It played a major role in rewiring my brain in just the right way so that I began to develop some understanding and
insight about user interfaces. I would have ranked this book higher were it not for Cooper's book, which covers much of the same ground and is more relevant to modern Windows programmers.

Jakob Nielsen's *Usability Engineering*


User interface development is a difficult subject to write about. User interface books written by practitioners are typically the most relevant, since they discuss current technology and describe lessons learned from practical experience. However, such books are based largely on the opinion of the author and are rarely backed up by research. On the other hand, academic user interface books often have little practical information for programmers and are just plain boring. (How's that for an opinion!) Jakob Nielsen's *Usability Engineering* strikes a remarkable compromise. Nielsen's information is practical, interesting, well researched, well written, and very readable. And, because the subject of usability engineering is not based on any particular technology, the information is timely.

So, just what is usability engineering? The goal of usability engineering is to create software with the following attributes: learnability, efficiency, memorability, reduction of errors, and overall user satisfaction. Usability engineering is a collection of techniques that strive to obtain these attributes throughout the entire software development process. Since it is difficult for developers to predict what users want and how users will react to a program, the primary focus of usability engineering is working with users to establish program goals and evaluate results.

Usability engineering is similar in concept to user-centered design. One of the biggest problems I have with user-centered design is that it is based on the faulty assumption that developers have no clue how to design user interfaces and that the only way to develop a good user interface is through user testing. Nielsen knows better. Chapter 5, "Usability Heuristics" (which in any other book would be called "Basic Design Principles"), outlines many usability techniques that can be taken for granted in software design. Furthermore, Nielsen presents a realistic view of user-centered design and prototyping. He understands that user testing doesn't have all the answers, that users aren't designers, and that their feedback must be carefully interpreted. He states, "There are two major reasons for alternating between heuristic evaluation and user testing as suggested here. First, a heuristic evaluation pass can eliminate a number of usability problems without the need to 'waste users,' who sometimes can be difficult to find and schedule in large numbers. Second, these two categories of usability assessment methods have been shown to find fairly distinct sets of usability problems." This is great practical advice.

I have two minor complaints about this book. The first is that it expends a significant amount of effort describing formal usability measurement techniques. While such information is useful for researchers, it has little value for programmers. The second is that some of the research cited is fairly old. There's no particular reason to think any of the conclusions are wrong, but research based on character-mode user interfaces is clearly dated.

I find what Nielsen calls "discount usability engineering" to be the most useful and interesting information in this book. He understands that developers don't have unlimited budgets or time, so they need to use techniques that are quick and cost-effective. The inability to perform all of the processes described in this book should not be used as an excuse to avoid usability altogether. As Nielsen puts it, "Unfortunately, it seems that 'Le mieux est l'ennemi du bien' (the best is the enemy of the good) to the extent that insisting on using only the best methods may result in using no methods at all." Clearly, some usability engineering is better than none. Discount usability engineering consists primarily of user and task observation, scenarios (a minimal prototyping technique), user testing that employs thinking out loud, and heuristic evaluation. You shouldn't feel guilty about not using the ideal usability techniques. (I know I don't.) Instead, try to do the best you can.

This book is definitely a worthwhile read.
Laura Arlov's GUI Design for Dummies


I know—it's hard to believe. The saying that you can't judge a book by its cover definitely applies here. This is a solid book, definitely not for dummies. This book would probably sell much better if it had a different title.

Arlov's book has a balanced presentation, covering user interface design in both practical and theoretical terms. The issues are covered from many points of view, including programming, graphic design, and user psychology. Arlov also includes quite a lot of information about the design process and task analysis. This book would be an especially good choice if you are interested in creating a design process. While all the chapters are solid, I found that Chapter 6, "How Users Get Around: Navigation Models"; Chapter 11, "Making Your GUI Easy to Understand"; Chapter 13, "The ABCs of Visual Design"; Chapter 14, "Color Is Communication"; and Chapter 15, "Icons and Graphics," are especially good. My only complaint is that while this book covers most of the user interface subjects that I feel are important, the coverage isn't especially deep and the presentation is a bit too breezy for my taste.

If you have read Cooper's *About Face*, Howlett's *Visual Interface Design*, Norman's *The Design of Everyday Things*, and Horton's *The Icon Book*, much of Arlov's book will be review. On the other hand, if you haven't read these other books and don't plan to, this book would be a good choice for a summary of much important user interface design information.

Peter Bickford's Interface Design


This book is based on a user interface design column that Peter Bickford wrote for *Apple Directions*. (He took over the column after Bruce Tognazzini moved on.) It consists of 38 chapters, each covering a single subject. Among the subjects that I find the most interesting are transparency, error messages, preferences, icons, speed and feedback, usability testing, complexity, and providing intelligence. Though fairly short, each chapter typically has worthwhile insight that can be quite helpful. This book also has the advantage of being a relatively quick read.

Windows programmers should note that although Bickford made a token effort to expand the coverage to include Windows, this book is clearly written from the Macintosh point of view. While this in itself isn't a problem, the fact is that Bickford is a serious Mac-head with an agenda. You know what I mean: The Macintosh is ten years ahead of Windows. (He doesn't literally make this statement, but that's what comes across since every reference to Windows is negative.) All the user interface innovations developed in Windows are terrible. And so on. Try to ignore that stuff, and you'll find some really good ideas in this book.

Of course, these are my personal opinions. I can only guarantee that these books have helped me.

Recommended Reading

I have presented the items in this list in order of their usefulness to me.

While you could never learn how to design a user interface from this help information, it can be very helpful in addressing specific problems you may have when implementing your interface. I recommend reviewing this information just so you know what is out there. For some reason, this information is easy to overlook.


This book is the ultimate resource for icon design information. This is possibly the most narrowly focused book on a user interface subject ever written. However, while this information is intended to help you design icons, note that the same design principles apply to other visual symbols, such as toolbar buttons, cursors, and control indicators (such as the Windows close, minimize, maximize, and restore button graphics). Chapter 2, "How Icons Work"; Chapter 3, "Representing Ideas Graphically"; and Chapter 7, "Color in Icons," all contain valuable information general enough to be useful to those who never plan to create icons but who want to improve their visual design skills. This book will help you learn how to communicate visually. If you are having trouble expressing your ideas visually, this is the first place to turn.


Presents everything you need to know about internationalizing software. This isn't a user interface design book per se, but it's a fact that much of the effort in internationalizing a program involves the user interface. This book makes it clear that creating international software is much easier to do at the beginning of the development process than at the end, so this book is a must-read if you're planning to create international software. Don't be intimidated by the heft of this book—it's a quick read.


Provides useful guidelines for all aspects of documentation. As I discuss in the next chapter, I believe it is important to establish consistent terminology in your interface, and this book is the best source I know of to help you accomplish this goal.


This book is a good tool for making sure you are using the right term. Again, consistent terminology is crucial in interface design, and this book also helps accomplish that goal. Of course, this book is less helpful than the *Microsoft Manual of Style for Technical Publications* because the manual of style helps you understand which term to use and the dictionary just gives definitions.


This book is a good source of information on how to integrate context-sensitive help into a program. Third-party tools are available to help this process, but this book has useful information if you plan on doing it the hard way.


Discusses everything you ever wanted to know about error messages. While I think this book does an excellent job of identifying error message problems, I wouldn't use the proposed solutions presented here. Nevertheless, I find this book useful when considering error messages and error message
problems.


Despite its title, this book is an excellent resource for basic Windows graphics information, even if you never plan on doing any animation. The first three chapters provide a topnotch presentation on device-dependent bitmaps (DDBs), device-independent bitmaps (DIBs), video modes, colors, and palettes. It is essential reading if you need to do any bitmap or palette programming in your user interfaces.


While many user interface books have bibliographies, the one presented in Appendix B of this book is possibly the best. It has excellent coverage, is well annotated, and is not Mac-specific. Unfortunately, it is getting a bit old. Check here first if you are doing advanced research.

**CD-ROM Resources**

The CD-ROM included with this book contains the following resources related to this chapter:


**Chapter 3**

**Establish Consistent Terminology**

Carmel, California, is world famous for its small-town charm and character. In addition to having Clint Eastwood as a former mayor, it has many other interesting characteristics, such as beautiful beaches, distinctive shops, fabulous art galleries, no franchise businesses, no mail boxes anywhere but at the post office, and most interestingly, no street numbers. Since I used to live two towns over in Monterey, I knew about these distinctions for years, but it wasn't until someone in Carmel gave me directions to their house that I fully appreciated the significance of not having street numbers. Without street numbers, residents can't just say they live at 123 Seventh Street. Rather, they must give directions such as "Our house is on Seventh Street, between San Carlos and Junipero. It is the light-gray house with white trim, on the north side next to the large oak tree...."

While not having street numbers might give Carmel character, not being able to easily find your way around does nothing for software. Yet programmers often seem reluctant to establish terminology to describe their programs. Often someone other than the programmers is responsible for deciding the names. But regardless of who is responsible, it is important to establish consistent terminology early in the software development process for all the significant elements of a program that will appear in the user interface. Because the terms you choose will often show up in the actual interface through menus, dialog boxes, status bar text, help systems, documentation, and so on, consistent terminology will both clarify the interface and make it more consistent. If you find yourself saying such things as "the second button on the window on the left-hand side of the screen—the Workspace window or whatever we end up calling it—needs to be larger....," you know you need to work on your terminology.