Part I

Keeping It Simple

CHAPTER 1

The UUI: The Useable User Interface

And to think, the day had been going so well for me. I got up on time, I ate a healthy breakfast, the mail came early, the birds were singing, and the day was perfect. That is, it *was* perfect, until *it* happened.

What happened? The software that had long been my friend decided to try my patience one more time, and my head came very close to exploding. In desperation I looked to the sky and saw a few sprinklings of clouds as they started to turn gray, bragging of the looming thunderstorm they were about to bring.

You've probably had days like this. Think about the software you use on a daily basis. Can you think of anything that you don't like about it? Can you think of things that just go wrong when you use it? Or maybe you just have a list of annoyances about the software.

Without naming names, here's a list of my complaints about the software program I was using:

- I clicked the wrong menu item and a dialog box I wasn't interested in opened. That's fine (my mistake), but when I pressed Esc, the dialog box didn't go away. It just stayed there, grinning at me, taunting me, daring me to click the Cancel button.
- The buttons have a strange look about them and the program ignores the colors and themes that I have chosen for my Windows XP computer. While this doesn't really change the useability of the software, it does make it a tad bit more annoying.
- *Some* of the buttons do not respond to the keyboard. You must click them with the mouse. (Others do respond to the keyboard.)
- Various child windows (that is, smaller windows that are part of the program) open unexpectedly, without me manually opening them. And these child windows have buttons. If I happen to be typing into a different child window, and just by chance I hit the spacebar, that window that opened will receive the spacebar and interpret me as clicking its default button. (Wouldn't you know, these buttons *do* respond to the keyboard.)
- When I start up the software, I have to wait…and wait…and wait before it does anything. Worse, during this time, my whole computer seems to slow way down.
- This software has the ability to communicate over the Internet. I have a cable modem, but once I accidentally chose the modem option. I don't have a phone cable hooked to my computer. And so I had to wait…and wait…and wait until the software figured out the dial tone was not present.
- While using a dialog box, I pressed Tab to get to another control in the dialog box. The focus, however, switched not to the control I expected. Pressing Tab over and over, I saw the focus go in a very strange order throughout the dialog box.
- At one point the program let me type in a few paragraphs. After doing so, I wanted to edit the text a bit. I clicked on some text in the middle. Then, as I tried to use keyboard commands to

select the text, I held down the Shift key and pressed the right-arrow key while holding down Ctrl to select some text word-by-word. (So far so good.) I went too far, so, while still holding down Ctrl, I pressed left-arrow. But instead of unselecting the rightmost word, the selection expanded to the left from where I began. This is not standard.

All this was just one program. Now here are some things I ran into with some other programs on that very same fateful day:

- I clicked the close button, and a small dialog box opened asking if I was sure I wanted to quit. I accidentally clicked back on the main window…and the small dialog box vanished. I could continue using the program as if I had not clicked the close button. But when I moved the main window, I could see that the dialog box was still there. It had just gone in back and was hiding.
- One program I use occasionally decides it has to do some figuring or calculating or something (I honestly don't know what it is doing), and it slows my whole computer to a crawl. It's not a good software neighbor. (It probably does what is called a *tight loop*, which is a topic I mention in Chapter 8, "Under the Hood."
- And then consider this example: A friend of mine was typing a long e-mail message using one of the free Internet e-mail servers. She clicked Send, and something went awry on the Internet and she got the infamous "Cannot find server" message. She clicked back. The message "Page has expired" came up. Her e-mail message? Gone forever. (I encouraged her to always press Ctrl+A and then Ctrl+C to select all the text and copy it to the clipboard… just in case.)

These are just a few items for you to think about. I'm sure you've created in your mind your own list of frustrations. You are, after all, a user of a computer as much as you are a programmer. In the sections that follow I talk about ways you can keep your software from being *frustrating*.

So first, let me present my Golden Rule of Software:

RULE

Make it invisible. The user's mind can focus on either the work the user is trying to accomplish or the software itself. The moment the user focuses on the software itself is the moment she stops thinking about her work and starts to become *frustrated with your software*. Suppose you're expecting a phone call regarding a possible six-figure job. The phone rings, and you go answer it. Easy. But what if when you answer the phone, you are startled to hear a voice say, "This is your telephone speaking. I have encountered a software error. Please retype your password before accepting the incoming call." Can you say *blood pressure medicine?* I think you get the point. You want the phone to be invisible so you can just use it and focus on the real task of talking on the phone. *Same with your software*. The user has a job to perform and doesn't want to have to worry about your software because it functions poorly.

It's Intuitive! Trust Me!

Maybe I'll just quit using a computer. My life would be so much easier. But considering computers are supposed to simplify our lives, instead of quitting, I'm going to help you learn to create software that is not plagued with these problems so that we can all enjoy our computers (for hours on end, of course, unable to peel ourselves away). And to get the ball rolling, I want you to consider the following statement:

"Our software is intuitive."

This is not a direct quote by some marketing guy creating advertising copy for a circa-1985 computer magazine (although it might as well be). Instead, I made it up, but it's indicative of a mindset common among software developers, even to this day. We want to believe that our software is *intuitive*, whatever that means. Early in the days of personal computers, people bragged that their software was intuitive. People said the Macintosh was *intuitive*. The Amiga computer created by Commodore had a graphical user interface system that was named *Intuition*.

Various dictionaries give different definitions of intuition, but a common thread is that if you have to use your intuition, you are using a part of your brain where you don't need your cognitive and reasoning abilities. *Yee-ikes!* After reading that, I'd be terribly hesitant to suggest that my software is intuitive, because being intuitive would be a major thing to live up to. I'd hate to see my software put to the test: Send it off to a deserted island and hand it to Mr. Robinson, who has been stranded there since 1975, living, quite happily and healthily, off of the local flora, without any computers. Hand him a computer equipped with Microsoft Windows XP, and put him to work using my (ahem) *intuitive* software program, *without me giving him any instructions whatsoever*.

I can see it now. Assuming Mr. Robinson doesn't smash open the computer and tear out the parts and build a two-way radio (not intuitive, but definitely ingenious), I seriously doubt he would be able to figure out my program. In fact, I can see him getting frustrated and yelling at it the same way I did when I started my day. (While knowing how to use computers might not be hardwired into our brain, I strongly suspect *getting mad* at computers *is* hardwired.)

Okay, I think that pretty much drives home the point that we're deceiving ourselves to think we're building intuitive software. However, I would argue that we are constantly encountering things in our software experience that is *counterintuitive*, that is, stuff that simply doesn't make sense.

Consider the scrollbars, for example. Right now, as I type, I have a big Word document open, with several pages. I look over and I see a scrollbar. I am about to use that little work of wonder to scroll back to the previous page. Now in my mind, I can imagine what it will look like as I move the scrollbar's thumb *down*: The text will all move downward. The entire document will fall closer to the ground and soon the top of it will be at my eye level. And so I go over to the scrollbar and move the scrollbar down.

FIGURE 1.1
When the scrollbar's thumb is at the top, the document is at the top. When the thumb is in the middle, the middle of the document is in view.

Right. In reality, if I move the scrollbar *down*, the text moves *up*—the text goes in the opposite direction. Does *that* make sense? Not particularly *at first*. But what does make sense is when you use a bit more cognition and logic and instead realize that the thumb of the scrollbar represents the *position in the document*. To go to the top of the document, I move the thumb to the top of the scrollbar. To go to the bottom of the document, I move the thumb to the bottom. At first my intuition might have told me one thing (or maybe not), and what really happened was quite different and required a bit of logic mixed with experience. Figure 1.1 shows the relative positions of the scrollbar within two identical documents.

And now when I press Enter at the end of a document, due to the current formatting selection, my insertion point automatically gets indented. That way, each paragraph has a nice half-inch indentation. Well that's fine and dandy, but what happens if I press the Up-arrow key? As I might expect, the insertion point goes pretty much straight up so it's sitting between two characters about a half inch into the last line of the previous paragraph.

But now I press the Home key so the insertion point goes to the left edge of the paragraph. And then I press the Down-arrow key, so I'm back where I originally started, indented in, ready to type a brand new paragraph. And *again* I press the Up-arrow key just as I did a few moments

ago. But *this* time, the insertion point doesn't go straight up! It goes up to the next line up as expected, but instead moves all the way over to the left.

Again this might at first seem counterintuitive. Sometimes the Up-arrow moves the insertion point straight up, and sometimes it moves the insertion point up and to the leftmost edge of the paragraph. But logic and reasoning come in, and I realize that the reason is that Microsoft Word is smart and decides whether to move the insertion point straight up or up and to the left based on my *previous keystrokes*.

Clearly, intuitive computer software exists only in a fantasyland. But fortunately, you, as a software designer, can at least expect that certain idioms exist. In other words, you can be assured that the people using your program—for better or for worse—have certain expectations when it comes to using computers. And you, as a software designer, would have the happiest customers (and fewest angry e-mails) if you stick to these idioms. In the next section, I talk all about what exactly an idiom is and how, if you recognize idioms, you will have a software package that people will love.

Imagine Changing the Scrollbar's Direction

Want to really upset your users? Why bother trying to make your users happy when you could have some *fun* with them? Imagine the amused looks on their faces if you wrote a hack that reverses the functionality of all the scrollbars on a single computer. To scroll the text up, these new scrollbars require that you move the thumb up—not down as is the norm. To scroll the text down, move the thumb down. To get to the beginning of the document, move the scrollbar all the way to the bottom. And so on.

I'm sure the people you did this to would laugh and get a big kick out of it. They would want to meet you because you're so clever and they would invite you out to dinner and even pay for your meal!

Doubtful. We humans have a strange behavior when it comes to computers. The most relaxed, happy-go-lucky Homo sapiens will quickly turn into a *ragingus maniacus* given just the smallest computer problem. You don't want to be in the room when this happens, and you certainly don't want to be accused of writing the software that caused this extreme flow of adrenaline. Therefore, I suggest for your safety and for the mental well-being of your fellow humans that you stick to the norms when it comes to designing software. The psychiatrists of the world will thank you in the end.

Idioms and the Software Experience

Now that I've pretty much made the case that software cannot be intuitive, I want to show you what takes the place of intuition: the idioms. The term *idiom* is often used in reference to spoken languages. People learning foreign languages often study idioms before traveling to the countries using those languages. An idiom is a common phrase that, when taken for its pure word-for-word meaning, has nothing to do with the way people use it. I found a website that has a great list of common American idioms; you can visit it at `http://www.englishdaily626.com/idioms.html`. Here are some idioms this page mentions:

- Spill the beans
- Jump the gun
- Hang on

The first of these has nothing to do with spilling the beans. To "spill the beans" means that you are revealing something that you were not supposed to, like, "She spilled the beans that she's taking me to the mountains for a vacation."

The second, "jump the gun," has nothing to do with guns. It means somebody started something too early (although the root of this phrase has to do with races and a gun firing to signify the beginning of the race).

And finally, "hang on" has nothing to do with hanging on. You might use it like, "Hang on, I'm on the telephone," or "Hang on, I'll be with you in a minute." It means "wait." (Although I suppose it could be used as in, "Hang on, Dad! Grandpa is bringing the other ladder that's still in one piece! Don't let go!")

Software is filled with idioms, which are simply cues that are visual or otherwise that we know to mean something. One idiom is the Taskbar in Windows. I know what the Taskbar is, and so do you. It wouldn't take long to teach somebody what the Taskbar does. Now the Taskbar doesn't have much of a literal meaning: It's just a bunch of rectangles at the bottom of the screen, each containing an icon and some words. But we know that if you click one of the rectangles, the program by that name will come to the front. There's not much intuitive about the Taskbar, but it's simple and it works. Here's a sample Taskbar:

The notion of a window *coming to the front* is another idiom. Realistically, the screen isn't layered, and you don't have pixels behind the ones that you see containing the windows that are in back and obscured from view. But we do have an idiom: We all know that the windows have a layered look.

NOTE However, Microsoft has made a bit of a mess, because in Windows XP, you can set up your Taskbar to group windows. This is a good thing, because instead of seeing 14 Internet Explorer rectangles all squeezed onto your Taskbar, you see only one, and clicking that one gives you a small menu of all the windows in the group. But the downside is that you have no way of knowing whether the window is minimized or not. If you accidentally click the icon for the window you're looking at, then the current window will minimize, showing what was behind it. That can be confusing.

Idioms also take the form of icons and symbols. We have all learned what the mouse arrow represents, what the hourglass symbol is, and what a folder icon is, even when it looks a little different between versions and operating systems.

These are by no means intuitive. Consider the blinking caret that sits in the middle of the text in a Word document. Does it represent some magical division line, where text to the left of the caret is important, while text to the right is unimportant? Or maybe it simply signifies the exact center of the document. Yes, I'm being silly, but you get the point: The blinking caret is an idiom, and now you and I both know what it does.

RULE When you design your software, keep the idioms simple enough so that they don't get in the way of the real purpose of the program.

If I were to build a word processor, I might consider this idiom:

The speed at which the caret blinks denotes which characters you are allowed to type. If the caret blinks five times a second, you may type any character. If the caret blinks two times a second, you can type only vowels. If the caret blinks once a second, you may type only consonants. And if the caret blinks only once every four seconds, you may type only the letter *s*. Anything slower, and you may not type at all. Further, you can control the speed of the caret by moving the mouse….

You can see this is getting a bit out of hand. Yes, it's an idiom. No, it's not a good idiom. It's too complicated, and it doesn't provide a value to the user. What is the goal of the word processor? To type pages of text, not to mess around with some stupid idiom about the speed of the blinking caret. Soon you'd be more involved in trying to understand the blink speed rather than focusing on what you're trying to type. Remember the Golden Rule of Software: Software should be invisible. The moment the user's mind leaves the work that the software is serving and begins to focus on the software itself is the moment you're about to have a frustrated user. And to aid in this Golden Rule, keep your idioms few, and keep them simple.

How do you design good idioms?

RULE Good idioms are the result of a combination of established idioms mixed with common sense and careful design.

That Stupid Desktop Metaphor

My desk is a mess. Now, if my mother, Pat, were sitting here beside me, she'd use the term she always used to describe my bedroom when I was a child: *It's a disaster*. (And trust me, she wasn't laughing at the time.)

Let me describe my desk for a moment. I have a monitor on it, surrounded by junk—a digital camera, several cards for the camera, two speakers for the computer that are turned off, numerous business cards, two empty glasses, a measuring tape for an item I recently sold on eBay, a digital thermometer from when I was recently sick, a dental appointment card, a pair of fingernail clippers, several pens (half of which don't work), a pile of papers with notes scribbled on them, a couple of books…and so on. It's a mess. Trust me; I'm not including a photo.

So why in the world would I want to base my computer organization on this thing that the gurus of the 1980s called *the Desktop Metaphor*? And tell me, where on my desk is an icon?

The desktop metaphor is a joke. But it was a good attempt. The idea was to give the people of the world something they could relate to: A desktop, nice and simple. And for the most part, it worked early on, because the computer trainer teaching the frightened secretary could say, "Think of this as your desktop."

Other metaphors have also worked their way into the computing world. The file cabinet is one. (And Microsoft even changed terminology back in 1995, when they deemed directories as *folders*. Apple, on the other hand, always had *folders*.)

Here's the basic problem with the folder metaphor: In my computer, I can put a folder in a folder in a folder in a folder. Try doing that with your real file cabinet, the one of the metallic persuasion. You probably could do it, but it would make for a bit of a mess. And where does the file cabinet itself fit in? Figure 1.2 is an example of a folder containing folders.

Now I admit, the folder metaphor has, for the most part, worked. People seem to understand the metaphor, *to a point*. However, when a user sits at the computer, and he opens a folder he has never seen before, why does he know what that is? Does he mentally make a connection to the metal file cabinet beside his desk? Of course not. Instead, the reason he understands it is that *yesterday* he knew what it was. And tomorrow he'll still remember what it is. In other words, his knowledge and understanding have nothing whatsoever to do with the metaphor of a file cabinet. His knowledge and understanding come from experience and from *recognizing the idioms*. That is, once he *learned* what a folder is, he could recognize folders in the future and continue using them. *The metaphor portion is gone*.

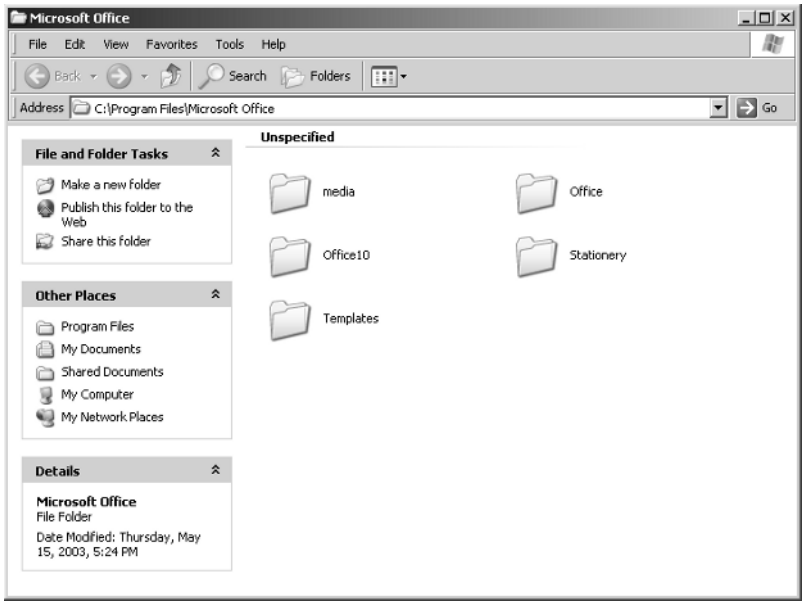


FIGURE 1.2
This window represents an open folder, and you can see that it contains other folders.

TIP Metaphors are (sometimes) acceptable in training situations and have little use beyond that. Do not design your software with a metaphor in mind because, frankly, people will *no longer need the metaphor* once they understand how to use your software. Otherwise you'll be using a metaphor as a replacement for good idioms. Instead, focus on the idioms: few and simple.

When the newbie learned to use the folder, somebody probably said, "This is called a folder, and think of it like a folder in a file cabinet." But after that, the trainer had to explain how to double-click the folder and show him that a window would open, and that this window contains icons, and that he can drag them to another folder, and so on. And notice what's happening? The metaphor is gone! And is the newbie making a mental connection to the metaphor? Doubtful.

WARNING Even in training situations, metaphors can become a problem. If the student is too smart, she might try to relate every process to the metaphor, which will invariably cause confusion. Consider these metaphors: creating a folder in a folder (does that really make sense?), creating a shortcut to a program (I've never seen a shortcut in a folder inside a metal file cabinet). And instead of accepting everything at face value, the thinking student will start asking questions the trainer is unable to answer.

REAL WORLD SCENARIO

The Vintage Doorknob That Trapped Them Inside the Room

I had something scary happen. And I mean really *scary*. I noticed that the doorknob in one of the bedrooms of the old house I live in was loose. It had *play* on it. If I turned it just a bit, I would feel nothing happen, and only after I turned it just a little bit more would I feel it engage and begin to do its job. And so I decided to fix it. I didn't think a doorknob could be all that complicated, even though these are vintage doorknobs. I've replaced newer doorknobs before, and how different could this one be?

And so I fiddled a bit and figured out how to unscrew the outer mechanism, and soon I had the whole contraption sliding out of the door. The two knobs connected in the middle and had a little bolt attached to one that would tighten up into the other. All I had to do was tighten the bolt. And that's when it hit me: *If this thing ever so much as even thought about loosening up any more than it already had, the inside door handle would be rendered completely useless!*

I can see it now. I have my door closed in my office as I'm working hard doing all that computer stuff that we computer people do (like seeing how many web pages it's humanly possible to read in a single day), while the family is outside enjoying life. And finally at dinnertime I'm ready to emerge and return to the living. I walk up to the doorknob…closer…closer…and I finally grab it…I turn it…and nothing happens!

For years the neighbors would hear the screams of "Help!" emanating from that secret dark room. They would see my silhouette in the window as I peer out at them passing by, frightening them while I try to flag them down. Trapped. Unable to get out, for years upon years.

Now how is that for good design (or not)? Engineers can certainly make mistakes, and you, as a software engineer, can easily put yourself in the consumer's shoes when you experience objects and items outside the computer. Think about how they work, and if they frustrated you, think about how they could be designed differently. Become the consumer. And then you can relate to the consumers of *your* products.

Let's face it, the metaphor of the desktop and the file cabinet are pretty much here to stay, like it or not. But let's leave good enough alone and keep additional metaphors out of the business.

Besides, I don't keep my file cabinet on my desktop. I've heard horror stories of file cabinets tipping over and crushing innocent 19-year-old administrative assistants.

The Visual Basic Programmer Has Been Let Out of the Cage!

One of my biggest gripes about programming today is that it is unnecessarily difficult. For example, in one language I have to learn how to use standard templates if I want to create a list of strings. Another language has a type called TStringList. One is certainly easier to use than the other.

(However, please don't think that I dislike C++, which is obviously what I'm referring to here. C++ is one of my favorite languages. I'm just always looking for a simpler way to do things.)

And so I was very excited when Visual Basic came along. Visual Basic actually made programming *easy*. But some people might argue it made programming too easy. Alan Cooper (himself a guru in the useability world) created the original version of a development tool that soon became known as Visual Basic. What Cooper did in his proverbial garage was a breakthrough in the programming world. And today we see his original idea in everything from Borland C++Builder, Borland Delphi, Microsoft C#, various Java designers, and so on.

When Microsoft purchased Visual Basic and introduced version 1.0, they included an amazing feature that let people create their own *custom controls*. A custom control was basically a specialized control that could perform actions that the standard controls (buttons, listboxes, comboboxes, and so on) were not capable of.

Unfortunately, that's exactly the major flaw in this thinking. While some useful controls came out (such as various spreadsheet controls), the world was suddenly overcome with the most horrifying sight to the eyeballs. People made controls that were simply buttons but had the most unreadable 3D text that appeared to be carved out of gray rock. Other people made controls that used every single color that the graphics card at the time was capable of. These controls could spin, dance, whirl, sing, change your oil, and cook your dinner. And they were the ugliest things ever to grace a computer screen.

But to add horror to horrors, other programmers were actually *using* these things in their programs. Users would download these cool new programs and find out that they needed this file and that file to use the program. *This* file and *that* file were, of course, extra Visual Basic controls. And then when people would start up the programs, they would be treated to the crayon equivalent of an explosion in a spaghetti factory.

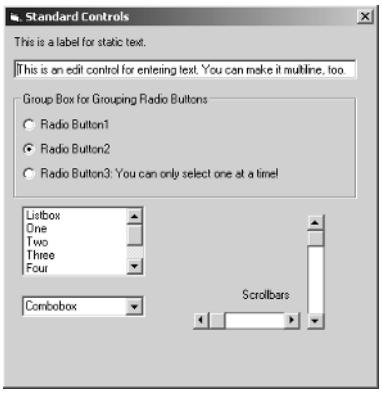
Come on now. This wasn't necessary, was it? Alan Cooper would be rolling in his grave, except he's still with us, thankfully.

A friend of mine summed it up well when he said, "There's nothing worse than a Visual Basic programmer let loose."

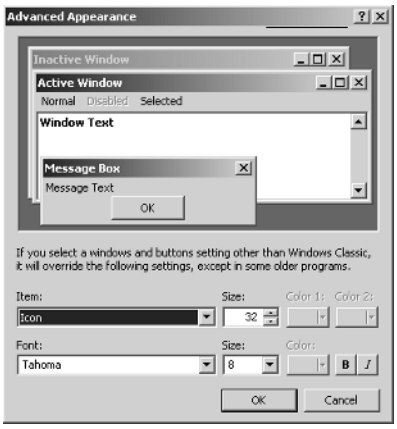
Fortunately, the days of outrageous custom controls have passed. People don't tolerate such abominations. However, creating user interfaces that look like such abominations is still possible, whether you use custom controls or not. Here, then, are some tips to help you avoid creating screens and dialog boxes that look like an expensive, original Matisse painting sent through a meat grinder:

- Use only the standard controls the operating system provides you. The built-in controls in Windows XP, for example, are rich with features and *are well-established idioms*. If you think you need a custom control, think again. Yes, it's true that occasionally people really do need a

custom control. But these moments are *extremely* rare. Go about your work assuming you don't need one, and only if you get stuck while trying to program with the existing ones should you consider writing a new custom control. Following is a window containing the basic controls on all Windows systems; you can find these controls on other systems, too. Starting with Windows 95, you have at your disposal other controls, too, such as the ListView and the TreeView.



- Don't set *any* colors. I'm serious. First, simply call into the operating system, and let the operating system assign colors to elements such as buttons and listboxes. The operating system in turn chooses its color based on *user preferences*, which are set in the Advanced Appearance dialog box shown below (for Windows). But if you must assign colors for whatever reason, pick from the list of colors chosen by the operating system, since these colors come from the user preferences. These colors have names like ButtonFace (depending on the language you're using) and ActiveWindowColor, and they often match the names in the display preferences. (Of course, you can see the obvious exceptions: If you're writing a program for manipulating graphics, then you'll be setting the colors of the image. But I'm talking about the user interface: For the window title bar, the buttons, and so on, don't set the colors.)



- Don't use 3D and all that other crazy stuff to "enhance" the look of your interface. (The one exception where this seems to be fine is with games. With games, people usually make their own entire user interface.) Okay, I admit, the following is pretty cool and I had fun making it. But I would never *dream* of releasing this in a software package!

- Avoid messing with text. There is a practical reason for this: If you obscure your text in any way whatsoever, you might make reading it difficult for visually impaired people. And a less practical reason is that it will annoy people, whether they are visually impaired or not.

Giving Users What They Want (Including Respect)

I'm going to make a bold statement here. Bear with me:

RULE Don't assume your users understand the full inner workings of the computer (and don't hold that against them). But don't assume they're stupid, either.

Here's an example of where Microsoft messed up and assumed people are of a slightly lesser intelligence than they really are—and ultimately damaged many hard drives. The key sentence is "Hide extensions for known file types." Does this sentence ring a bell? Most likely, since you're a programmer, you *have* seen this sentence before.

When you first install Windows on a computer (any version these days), by default, the icons in the folders do not have their filename extension showing. So instead of `ThisIsCool.exe`, you will simply see `ThisIsCool`. The idea is that the icon itself should be a clue as to the type of program. If the file is `ThisIsCool.doc`, the icon will be that of Microsoft Word, since `.doc` files are normally associated with Microsoft Word. The users will then (supposedly) know what type of program the file goes with.

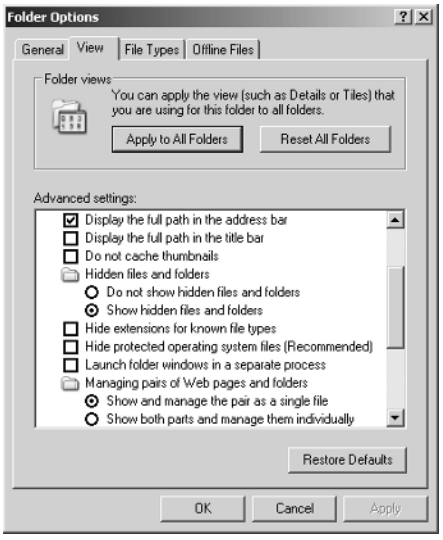


FIGURE 1.3
The users can choose whether to hide extensions for known file types.

But the problem is icons are easy to fake. I could create a file called `ThisIsCool.exe` and embed a Microsoft Word icon inside it, making it look like a simple Word document, when in fact the file could be a virus. Very bad. Or, the icon might be meaningless to a user, and the user has no idea that he is opening an executable file. Not good at all.

And so we more experienced computer users always open the Folder Options dialog box on a new system (or on a friend's computer), click the View tab, and uncheck the Hide Extensions For Known File Types option, as shown in Figure 1.3. (It's checked by default!)

The idea was a noble one: Microsoft felt that users shouldn't have to worry about the file-name extensions, since that's more of an *internal* issue. Who cares what the file type is? What matters is what's inside the file! Right? Wrong. And now lots of users have been hit with viruses and inadvertently ran them without realizing they were executable files.

You can see that you have to find a balance: Don't assume your users are stupid and can't handle things, but don't require your users to possess a complete knowledge of the intricate workings of the computer. Microsoft felt it was unnecessary to make the users understand what a `.dll` or `.exe` extension is. A little knowledge is important, and knowing what a `.dll` or `.exe` file is *is* important.

Software Libraries and Required Knowledge

These assertions about assuming that your users aren't stupid but not making them understand the internals apply even if you're creating software for other programmers. Suppose you're

creating a library of C++ classes. With this library you will ship a set of header files, possibly the source code, and likely a library file containing the linkable object code, either in the form of a static library with a `.lib` extension or a dynamic library with a `.dll` extension.

Now with such a library, don't force your users (the programmers) to learn completely how the library works *from the inside*. They need to know how to use the library. If the library uses Newton's Method to calculate a root of a polynomial, then definitely let the users know the algorithm used. That way they can decide whether it's right for them. But don't require them to understand that you used a stack structure and to know the names of the internal variables you used in your function. Do you see where the line is?

Of course, some users might *want* to know (or need to know!) such information. For those users, you can give them access to the source code of your Newton's Method function. (However, you might not want to; it's up to you.)

Just like with user-oriented software, keep your software libraries simple to use without requiring an advanced knowledge, but don't make the advanced users suffer by not giving them the access they need.

The Private versus Protected Debate in Software Libraries

When you design a class library in a language such as C++, you have the option of declaring members of your class private rather than protected. (Remember, the only difference is that derived classes can access protected members but not private members.) This has been a great source of debate over the years. If you are designing a library, always give a lot of thought to whether you really want your class members, especially member functions, to be private. The reason is that people using your library might want to derive their own newer, refined classes from your classes. By making some of the members private, you might prevent these new classes from calling some important functions. By making them protected, the functions are still secure from being called by outside routines; however, the derived classes can call them when necessary.

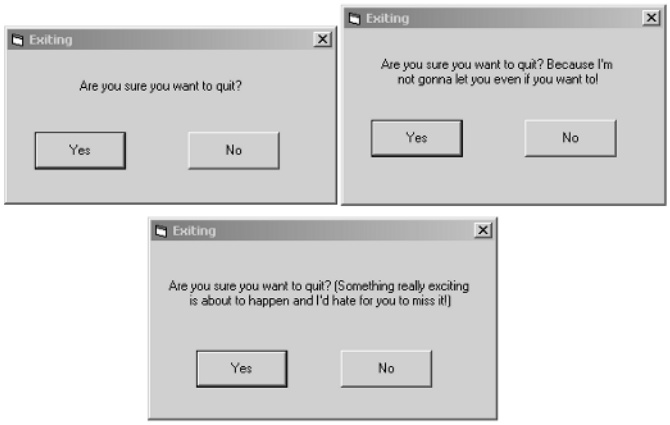
Asking Too Many Questions

One sure-fire way to annoy your users is to program your software to ask too many questions. Remember, your users are intelligent people, and you're safe making certain assumptions. And for those assumptions you're not sure about, choose some answers that seem safe, and place these answers in a user preferences section. Then the users can change the preferences if they're not happy with your answers.

In this section, I provide you with some questions that are annoying and you should avoid.

Are You Sure You Want to Quit?

Before you include a question such as "Are you sure you want to quit?" in your program, think about what would happen if you *don't* include such a question. Here's an example:



For fun I included two extra windows that answer my question, "Why is it such a big deal?"

Suppose Happy User is using your program, and then she decides to quit. She saves the file she was working on and chooses File ➢ Close. The program shuts down.

But then she realizes she forgot to do something before quitting. Would the "Are you sure…" question have helped here? Not really, if you look at it from a practical standpoint. What did she lose? Nothing. No data was lost. No problems ensued. All she has to do is restart the program and reload the previous document. No big deal. By having an "Are you sure you want to quit?" message, you are not giving any help to your users. If they mess up and really didn't want to quit, let them go back in. Better that than frustrate the users by always having to click the "Yes I'm Sure!" button.

One popular online messaging program opens a dialog when you quit, warning you that the program will no longer function if you quit. Not only is that annoying, it is, frankly, *insulting*. If you're going to choose to annoy your users, fine, but don't insult them on top of it all. Annoyed users don't like to be insulted. (And remember the *ragingus maniacus* hiding inside.)

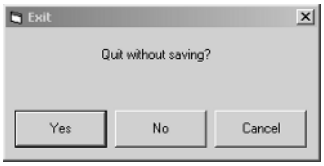
Do You Want to Save Your Changes?

Although a question about whether you *really* want to quit is annoying, a question about whether to save the changes before quitting is important, *unless* you follow some important tips that I provide later on in this section. If a user has been working on a file and then quits the program and forgets to save, you want your program to warn the user and give him a chance to save the work.

However, a dialog box asking whether to save before exiting has the potential to be extremely confusing. Look at this:

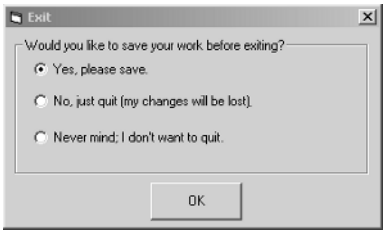
This dialog box has three buttons: Yes, No, Cancel. To us initiated folks, we know exactly what this means. But the reason I have a problem with it is that the beginner has to stop for a while and contemplate what this means. "Let's see…" he thinks. "Yes, I want to save changes." Or "No, I don't want to save them." But what happens if you click No? Does that mean the program will *not* save but *will* quit? To us, we know that's exactly what will happen. But to the beginner this isn't necessarily clear. And what about Cancel? What does that mean?

Now you might argue that this is just an idiom; deal with it, and move on. But it's not. Here's why. I've fallen victim to the following dialog:

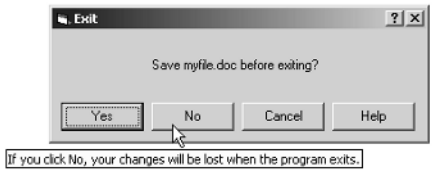


This dialog box is diametrically opposed to the previous example and, therefore, not just a little bit evil but 100 percent evil. If I click Yes, this means the program will not save my work, whereas in the previous example, the program *will* save my work. The programmer who created this program was doing his duty by ensuring that work will not be lost. But the person messed up by flipping the logic from what we're accustomed to.

Here's a better dialog box:



Unfortunately, this one is not standard. People who use Microsoft products all day long will have trouble with it, even though it's a perfectly good solution. Oh, what to do? Instead, try this:



This is a slightly modified version from what Microsoft Word uses, for example. This one conforms to the Windows way of doing things but gives the beginner a bit more help. If the user is confused, she has three ways to proceed:

- She can click the Help button.
- She can hold the mouse over the different buttons and see a *tooltip*.
- She can click the little question-mark button and then click a button.

Now the tooltip and the question-mark button might seem a bit unintuitive to the beginning user. So these two choices are for the users who have explored Windows a bit and are familiar with these two conventions. The Help button, however, is a safe place for all users.

However, I personally feel this is not the best option. I think the previous option (with the radio buttons) was clearer. This is a case where you have to choose between clarity and conforming to the standard. What would I personally do? I'd prefer to take the Fifth on this, although I have a feeling I'd probably choose the radio button version.

But you have another option, as I alluded to at the beginning of this section. Instead, you can *automatically save the file* when the user quits the program. To a lot of programmers, this might seem like a shocking thing to suggest. But I would encourage you to explore it, because some programs already do it with success. Microsoft Outlook is one example. When you exit Outlook, you don't have to save your changes. Instead, all your e-mail is saved automatically, and when you exit, you just exit.

But in the case of a word processor, this option might seem a bit strange. Here's how you can make this work. First, remember that when the user is working on a document, most likely the work he's doing is intentional. (He doesn't just randomly hit a bunch of keys.) After the user works for about 10 minutes (or 30 seconds if the user presses Ctrl+S obsessively as I do!), you can assume that the work the user did was *intentional*, and if the user exits, you can simply save his work automatically for him, no questions asked.

But what about the .01 percent of the time where the user didn't want to save his work? You can provide two approaches for these users:

- Allow the users to open a *copy of a file*. Then, when the user exits your program, the program will save the changes to a new (possibly unnamed) file. (Of course, the very first time, you might want the user to type in the name of the file.)
- Save a backup file before saving the new file.
- Possibly include the undo list with the file itself. That way the user can undo any changes that were made prior to exiting the program. (I like this idea the best, because it really allows you to *continue working* the next time you come back to the document.)

Remember, your goal is to make the majority of the users happy. Occasionally a user might have a strange situation. She might have wanted to do all this work without saving it. In that case, include in your online help instructions for getting the old version back. But for the other thousands of times, the users really did want to save the work, and there was no reason for you to ask before doing so, each and every single time.

If you don't believe me, think about this: How often do you save your work just before exiting? And if you don't, how many times in your life have you seen the dialog box asking if you would like to save before exiting? And now think how much time you wasted by having to click that dialog box or by manually requesting a save just before exiting.

Questions That Yield an "I Don't Know!" Answer

A question that yields an "I don't know!" answer is typical during installation programs. Here's an example of a question that most of us, even a skilled programmer, might sit and stare at for several moments, wondering what to do:

Although I made this window myself, I based it on a dialog box that I actually saw the other day. First of all, the choice of "Yes" and "No" is rather arbitrary. Better buttons would be "Install New" and "Keep Old", because "Yes" and "No" could easily result in a mistake.

But worse is the question itself. How should I know if I should install a newer version? A typical end user would have no clue and would just randomly pick one option. And for me, a

programmer, I know that some other program might need the version that's currently on the disk and won't be able to run with the newer version (but I don't know what program that is), and the program I'm installing might need the newer version and won't be able run with the older version!

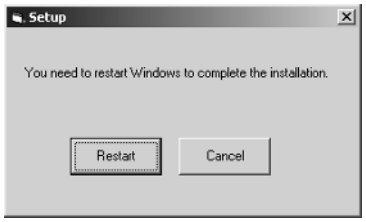
And if I'm feeling particularly ornery, I might even ask, "Why are you installing files in my System32 directory? Keep out!"

Don't ask questions like this when you write an installation program. Put the files in your own directory, and the issue won't even come up. And if you're working with Windows and have to register an OLE component, either make your own version of the OLE component or know what the differences between versions are so you can be more specific in your questions.

Would You Like to Restart Windows Now...or Later?

After installing a software package, if the program tells me I need to restart the computer, I usually laugh and say "No" and run the program anyway *just to see what happens*. Maybe that's a bit too impish, but I've reached a point in my computer life where I'm a bit fearless. I mean really now, what could possibly go wrong? Will I see an image of a little beastie go across the screen eating up all the bits on my hard drive because I forgot to restart Windows? Will my monitor start leaking poisonous fumes? What could go wrong?

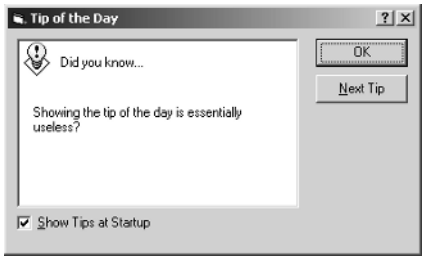
Yet time and again, I'm told I need to reboot Windows. Here's a typical dialog box:



In addition to the fact that the beginning user might wonder what the Cancel button does (does it undo the installation?), there's no reason for this dialog. Don't make your users reboot Windows. If you make changes to the Registry, those changes take effect automatically. If you register an OLE component, those changes take place immediately. Don't make them reboot. It's not nice. (I will concede that there might be some exceptions to this, and that's primarily when you're installing lower-level software such as device drivers. However, make sure you absolutely have to, because these days even device drivers can be loaded and unloaded without having to reboot Windows!)

Tip of the Day

Although this isn't a question, a Tip of the Day box is just as annoying as most unnecessary questions. Since the mid-1990s, the Tip of the Day dialog box has become popular. Here's an example of one:



Don't use these. Period. Although in theory the idea is great (what better way to convey important information?), the reality is the Tip of the Day dialog box breaks every imaginable rule. First, very few people read them. They either always click OK or the first time the thing opens they click the Don't Show This Again box. The information in the box is wasted, as is the time and energy you (the programmer) spent creating the box.

But worse, just what are the chances that your Tip of the Day dialog box is going to give the users some information that actually pertains to what they are going to do today? If I'm using a word processor to write a letter to a client, why would I need to see a Tip of the Day that explains how to import clip art? The Tip of the Day is wasted time and wasted space. Don't bother with it.

Saving User Preferences

One way to minimize the use of annoying questions is by implementing a user preferences system, as I mentioned earlier. In the user preferences section, you can allow your users to customize your software through the use of a User Preferences dialog box.

But don't go overboard; a User Preferences dialog box with 20 tabs and a million controls is frustrating, too, if the user is searching for a particular preference. Keep your User Preferences dialog box simple and easy to use.

Here are the steps you use for User Preferences:

1. When the software opens, read the user preferences, either from a file or from the Registry, a database, or wherever you stored them.
2. If the preferences are not found, set some defaults.

3. When the user opens the User Preferences dialog box and clicks OK, save the preferences internally. (As for whether to save them to the file or the Registry at this point is up to you.)
4. When the user shuts down your program, save the preferences to the file or the Registry.

You have some choices in implementing this kind of a system. For one, you might save only those preferences that are not defaults. If you're writing software for the PalmOS, for example, where space is tight, this might be a good idea. Also, you have different ways in which you can allow access to the user preferences. Microsoft products typically have a Tools menu, with an Options dialog box underneath it. This is pretty much standard, although some other big software packages (most notably Netscape Navigator) use other means, such as putting Preferences under the Edit or File menu. Since Microsoft has created many of the standards we see on Windows, then if you're writing a Windows program, I recommend following the Microsoft approach.

TIP If you are programming for Microsoft Windows, make use of the Registry. You can save your preferences in the `HKEY_CURRENT_USER` area. In this area you will find a key called `Software`. Under this key you can create a key for the name of your company, and under the company key you can create a key for your product name. Standard practice is to have version numbers under the product name key. Finally, under the version numbers you can place your user preferences.

Another issue that comes up is that of multiple people using your program. You have various ways to handle this, too. If you're storing your preferences to a file, you can save the file in an area specifically designated for the particular user. However, be careful with this approach. I don't like it when a program dumps a file in the `C:\documents and settings\jeff` directory on my Windows computer. (I have a bad habit of purposely deleting such files!) Instead, under this same directory is another directory called `Application Data`. Under that directory, you're free to create your own directory (again, don't put the file right in the `Application Data` directory), and inside your own directory, you can put your file. The advantage there is then you know what the file is if you go poking through these directories.

For example, right now I see this file on my computer: `C:\Documents and Settings\jeff\Application Data\dmqr.ini`. I have no idea what this file is, but it's the only file in this directory. Everything else in this directory is a subdirectory with the name of a company or product. I think I'll go delete it (but if you put it there, please expect a support call from me when your program doesn't work; but don't worry, I'm friendly and don't bite).

TIP Remember that multiple people may well use your program. Therefore, if you have a user preferences system, save the preferences on a per-user basis, not just on a system-wide basis. When programming for Microsoft Windows, you can save on a per-user basis by saving the preferences in the `HKEY_CURRENT_USER` area of the system Registry and *not* the `HKEY_LOCAL_MACHINE` area.

Remember the Keyboard?

A long time ago when Windows 3.1 was the great new thing, I wrote a memo to several coworkers explaining that the notion of a keyboard shortcut was stupid, and it's safe to assume at that day and age (roughly 1993) that every computer running Microsoft Windows had a mouse attached to it.

What was I thinking!

Hey, I had 10 years less experience than I do now, and I had a lot to learn about life. (Sounds like a song.) Needless to say, I no longer agree with the notion that keyboard shortcuts are not needed. In fact, I use them all the time. Lots of people do. Why? Because:

- They're faster if you're a fast typist (like me and all the piano players of the world).
- They're more convenient than reaching over to grab the mouse, moving it to where it needs to go, and clicking it.
- They're configurable; I can make Ctrl+Alt+S do what I want it to do, which is something you can't say about clicking with the mouse.

However, these points are true only when the software application includes a decent keyboard shortcut interface. What makes for a decent keyboard shortcut interface? Here I list what I personally expect out of one. (And trust me, I use keyboard shortcuts a *lot*, so for now, consider me your favorite user.)

NOTE People used to use the term *hot key* for keyboard shortcut. Somewhere along the lines, Microsoft instilled the term *keyboard shortcut* into our vocabulary, and these days, that's what most people say. I still prefer hot key, because I like shorter words, and the word *shortcut* is overused in Microsoft products: A shortcut on a desktop is a symbolic link to another program; a shortcut in Internet Explorer is a hypertext link.

The Shortcut System Is Implemented.

You want to make sure you actually implemented the keyboard shortcut system. For starters, most better programs include shortcut keys on the menu items. The Mac often uses the Apple key for its shortcut keys. Windows often uses the Ctrl key. On both the Mac and Windows you

can see the shortcut key for a menu item by clicking on the menu; the shortcut key is shown to the right of the menu item name.

Further, on Windows, you can add an additional shortcut key for the menu bar. Put an ampersand, &, inside the text for the menu, and the letter following the ampersand will show up underlined. For example, if you call your first menu &File, the menu will show up as File. Then, when the user holds down Alt and presses F, the File menu will open.

For menu items under the menu, you again use an ampersand to specify which letter is underlined. But for these shortcuts, the user first must open the menu (such as by pressing Alt+F) and then type the underlined letter (with or without the Alt key).

The Dialog Boxes Make Use of the Tab Ordering.

As a self-professed keyboard shortcut whiz, I find the lack of tab ordering always frustrating. Yet, it's amazing how many software designers and testers completely overlook this. Again and again I find software where the tab ordering is all messed up.

When a dialog box opens, typically you will want the focus to be on the control closest to the upper-left corner of the dialog box. For example, if that's an edit control, then the user doesn't have to first click that edit control to type text into it. The user simply starts typing and the text appears, because that's the control with the focus.

Then, if the user presses Tab, the focus should move to the next control to the right, if there is one, and down if there's not one to the right. The focus should move from right to left and top to bottom, just like the words in your code editor.

However, the Enter key is special, and so is the Esc key. When you design a dialog box, you can specify which button is the active one. The active button shows up with a dark border around it, like the OK button shown here:

Regardless of which control has focus, if the user presses Enter, the OK button should click. Pressing Esc should be the same as clicking the Cancel button. (Doing so is in line with the standard idioms that people expect of your software.)

Standard Keyboard Shortcuts Mean What They Usually Do.

Prior to version 2000 of Microsoft Outlook, many of us had a problem with sending e-mail midway through the composition of the e-mail. Why was that? Because Microsoft made a blunder. For quite some time, Microsoft has always used Ctrl+S as the keyboard shortcut for Save. And so those of us who had "Save Early Save Often" beat into our heads would press Ctrl+S almost obsessively. (I do to this day. In fact, I better go do it right now. There, I feel

much better. Never know when a power outage might come and I'll lose the last three words.)

But not with Outlook 97. In Outlook 97, Ctrl+S was the shortcut for none other than *Send*. And so during the composition of an e-mail, many of us would out of sheer reflex, about halfway through the e-mail, press Ctrl+S. And much to our shock, the message window would disappear and we'd see the little words "Sending Message 1 of 1" appear at the bottom of our Outlook screen.

See, Microsoft violated a standard idiom. We all knew that Ctrl+S meant Save. And guess what? In Outlook 2000 and beyond, Ctrl+S now *saves* the e-mail into your Drafts folder. It doesn't send the message. *Whew!*

Remember that most people expect various keyboard shortcuts to mean something. Please don't rearrange these, because you'll frustrate users like me who operate out of 90 percent reflex. The big ones are:

Ctrl+S = Save

Ctrl+C = Copy

Ctrl+V = Paste

Ctrl+X = Cut

Ctrl+F = Find

Ctrl+R = Replace, although Microsoft these days usually insists on Ctrl+H. Should we listen to them or not? This one is your call.

And some people like to use Ctrl+Q to quit. That's fine, if you like that one. We all have slightly different opinions on these. Just don't make Ctrl+S send e-mail. Please?

Keyboard Shortcuts Should Be Configurable.

Even though I just listed the "big ones," some people still get frustrated at these. But that shouldn't be a problem, because you're about to make your keyboard shortcuts configurable. Include some sort of dialog box that allows me (the user) to choose what I want the keyboard shortcuts to do.

The common way to do this is by developing a set of *use cases* for your product, which are basically the elemental things that people can do with your product. Some people call these *commands*. If you start one of the Microsoft Office products, and right-click on a toolbar or menu and choose Customize, you'll see a dialog box listing all the commands. For example, in Microsoft Word, you'll find a command for pretty much everything you can do, such as Toggle Italic; Toggle Bold; Insert Column Break; and so on.

When you design your software, if you divide the functionality into use cases or commands and then break each of these into its own function (or member function if you put it in a class),

then you can easily create a configurable system. All you have to do is list all the names of the commands in a dialog box and for each one let the user choose a keyboard sequence. The sequence can include the Ctrl key, the Alt key, or the Shift key, or any combination thereof. For example, you might want to assign Ctrl+Shift+I to the command Toggle Italic.

And how do you implement this? Through function pointers, of course. Use a common function prototype for each function, and it's a snap: Store the function pointers in a table of keyboard shortcuts (or some other list or array), and then when the user presses a key sequence such as Ctrl+Shift+I, look up the function that goes with that key (such as the address of `toggleItalic`) and call the function. Since all the functions have the same function prototype, you will know exactly what to pass to the function.

This is actually easier than it sounds. Although this part of the book doesn't focus so much on coding, here's a sample design pattern in C++ that does the trick using the C++ standard library's `map` class. I wrote the code from scratch and compiled it using the gnu gcc compiler:

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

typedef void (*command)();

// Here are the commands
void toggleItalic() {
    cout << "italic toggled" << endl;
}

void toggleBold() {
    cout << "bold toggled" << endl;
}

// Here's the class for the shortcuts
class Shortcuts {
protected:
  map<string, command> shortcuts;
public:
  void CallCommand(string sequence);
  void ReplaceCommand(string oldsequence,
    string newsequence, command cmd);
};

void Shortcuts::CallCommand(string sequence) {
  if (shortcuts[sequence] != NULL) {
    shortcuts[sequence]();
```

```
  }
}

void Shortcuts::ReplaceCommand(string oldsequence,
  string newsequence, command cmd) {
  if (oldsequence != "")
    shortcuts[oldsequence] = NULL;
  shortcuts[newsequence] = cmd;
}
```

Then, using this code is easy. To assign the keystroke Ctrl+I to the function called `toggleItalic`, call `keys.ReplaceCommand("", "cI", &toggleItalic)` (where `keys` is the name of the object). Then when the user presses Ctrl+I, simply call `keys.CallCommand("cI")`. Here's a sample `main` demonstrating this:

```
int main(int argc, char *argv[])
{
  Shortcuts keys;
  keys.ReplaceCommand("", "cI", &toggleItalic);
  keys.ReplaceCommand("", "cB", &toggleBold);
  // Assume user pressed Ctrl+I. Then call this:
  keys.CallCommand("cI");
  // Assume user pressed Ctrl+B. Then call this:
  keys.CallCommand("cB");
  // Assume user pressed something unassigned:
  keys.CallCommand("aQ");
  // User reconfigures italic to be Alt+I:
  keys.ReplaceCommand("cI", "aI", &toggleItalic);
  // Now user presses Alt+I
  keys.CallCommand("aI");
  return 0;
}
```

Once you implement a design pattern such as this, keyboard shortcuts (and menus and toolbar buttons) all become a snap. You can use the same interface for a menu; instead of mapping keyboard shortcut names such as "caI" to a function, you map menu item IDs to a function. And similarly with toolbars, you would map toolbar button IDs to a function.

Such a system is incredibly easy to implement, provided you know how to program various design patterns and that you fully understand your programming language.

The Real Risk of Repetitive Motion Injury

Let's face it and be up front here. Not only do you not want to annoy your users, you probably don't want to actually *injure* them, either. (If you've simply annoyed your users, you might know what it's like to receive e-mail from them, where they threaten injury.)

Now I know we're computer people and many of us are more likely to perform an exorcism than to actually *exercise*. But whether you work out daily or only during major holidays on every tenth year, you're probably familiar with that sore-muscle feeling the day after you work out.

The feeling of sore muscles is actually due to the muscles tearing at a microscopic level. When this happens, with the proper nutrition, your muscles heal themselves and become stronger. The key, however, in addition to proper nutrition, is in allowing time for recovery.

But the one exercise many of us do on a daily basis is type at the computer and use the mouse. While this might not seem like much, if you type and work the mouse for eight hours straight, you're likely to feel it a bit. My personal problem is that my elbow will often ache from reaching over to the mouse. I fixed this by getting a Logitech trackball instead, but I still have to reach over to the trackball. And sometimes my elbow still hurts, although the trackball has definitely helped a great deal. Other people feel pain in their fingers and hands from all the typing.

And, unfortunately, rarely do we give ourselves a few days off to recover from these aches and pains, which are, in fact, torn muscles, tendons, and ligaments. (Now isn't that a pleasant thought!)

NOTE Right through the middle of your wrist runs a nerve called the *median nerve*. This nerve is especially susceptible to trauma from using a computer keyboard. Damage to this nerve is known as carpal tunnel syndrome (CTS), which most of us have heard of. CTS is an example of a repetitive motion injury (RMI) or repetitive stress injury (RSI).

What can you do to help prevent repetitive motion injury and carpal tunnel syndrome? Since it would be a serious bummer to think that your software was responsible for somebody's bodily injury, here are some things that you can do:

- Don't require overuse of the mouse. In other words, include keyboard shortcuts for everything that might otherwise require the mouse. This includes menu items, controls in a dialog box, buttons in a dialog box, the works. And if the mouse is required, don't make the user bounce back and forth between using the mouse and the keyboard. For example, if you're writing a graphics program whereby the user can use the mouse to do drawings, set up the software so that the user can work the keyboard with one hand and the mouse with the other. Make this configurable, and don't forget about the left-handed people, either, who prefer to put the mouse on the left side of the keyboard.

But also don't require that the user constantly click in the upper-right part of the screen, then the lower-left, then the upper-right, and so on. For that matter, a lot of very small mouse movements can be bad as well. Further:

- Allow the user to do most of the keyboard work without bouncing the right hand back and forth between the J-position and the cursor and page keys and numeric keypad.
- Create a macro system, whereby users can program repeated keystrokes and activate the keystrokes with a single keystroke or click of the mouse.
- Design your software such that users can take a break from the keyboard without loss of data. This might seem like a strange request, but there are situations where this could be a factor. (Games come to mind.)

TIP A lot of people who suffer from RSIs find they're most comfortable with the Dvorak layout of the keyboard rather than the standard QWERTY style. However, if you're writing software that responds to the keyboard in any way (as most software does), don't build in Dvorak capabilities. Instead, Windows allows users to configure their own keyboards. This means that if your program receives the letter *j* as input, you can be assured the user typed *j*, regardless of where *j* is on the keyboard.

The common factor here is minimizing the movements of your users. Don't force them to go through jumps and dances just to get the task done.

Finally, if you want to learn more about the various repetitive motion and stress injuries, I have found a great place to start is by going to Yahoo! and searching on the phrase "repetitive stress injury."

Moving Forward

In this chapter I presented you with a set of introductory material on designing highly useable software. If you follow the principles I develop in this chapter, your software will already be more useable than that of the competition.

In the next chapter I move to the next step and look at the real problem of modeling the real world. Life can get touchy then; remember what happened with the "desktop metaphor." Ahh, that was an attempt to model the real world, wasn't it? Sure, it kind of worked, sort of, but let's leave good enough alone and skip the metaphors. Still, sometimes you have to model a business process. I take that up in the next chapter.

And finally, before you move on, here's one more quick story of a timely nature.

REAL WORLD SCENARIO

The Atomic Clock that Wouldn't

They called it an atomic clock. But really, it's a clock with some kind of radio-controlled device that picks up the exact time over the airwaves broadcast from some station in Colorado that has an atomic clock attached to it. My mother gave it to me as a gift a couple of years ago. She thought it was cool, and I had to admit, it was....

Until today (no kidding—the very day I was finishing writing this chapter). The clock has almost no user interface to speak of; you don't set it. Instead, you pick your time zone from a switch on the back and let it go. When you turn it on (by inserting a battery), you just wait. The second hand ticks a little sporadically, and then suddenly the second hand starts flying, going very fast, and the clock appears to be in fast motion. The minute hand moves at about the rate of a second hand. The clock spins like this for maybe 45 minutes or however long it takes until it gets to the current time. Then it slows instantly and ticks normally. After that you never have to set it.

The clock understands daylight saving time. At least it's supposed to. And it's always been correct until this evening when I looked at it and it was an hour *ahead*. Instead of saying 6:35 it said 7:35. I figured I bumped the time zone setting on the back when I moved the clock earlier today. So I flipped the clock over and looked. It's been about three years since I've had to do anything with the clock, and the instructions are long gone. But it should be straightforward. I saw that the user interface consisted of a little dial where you choose your time zone. (Only continental U.S. time zones are included, since those are the only ones within range of the electromagnetic waves shooting out of Colorado throughout our bodies). But here's the catch: The dial has a notch for every time zone, plus a *final* notch for daylight saving time. How can that be? Daylight saving time is not a time zone! But the selector lets you choose Eastern, Central, Mountain, Pacific, or Daylight Saving Time. That doesn't make sense.

Now I know the clock automatically adjusts itself in the fall and the spring, if you somehow tell it that you live in one of the areas that practices daylight saving time (that would be nearly everywhere in the U.S. except Arizona, Hawaii, and most of Indiana). And so somehow I'm supposed to tell it which time zone I live in *and* (not *or!*) whether or not I want it to automatically adjust for daylight saving time. But how can you do that when all you have is a single dial with separate notches for each time zone plus one for daylight saving time?

I don't know. I'm at a loss. I hope I don't have to throw out the clock because it got messed up and I can't figure out how to reconfigure it.

It's just another user interface gone bad, I guess.