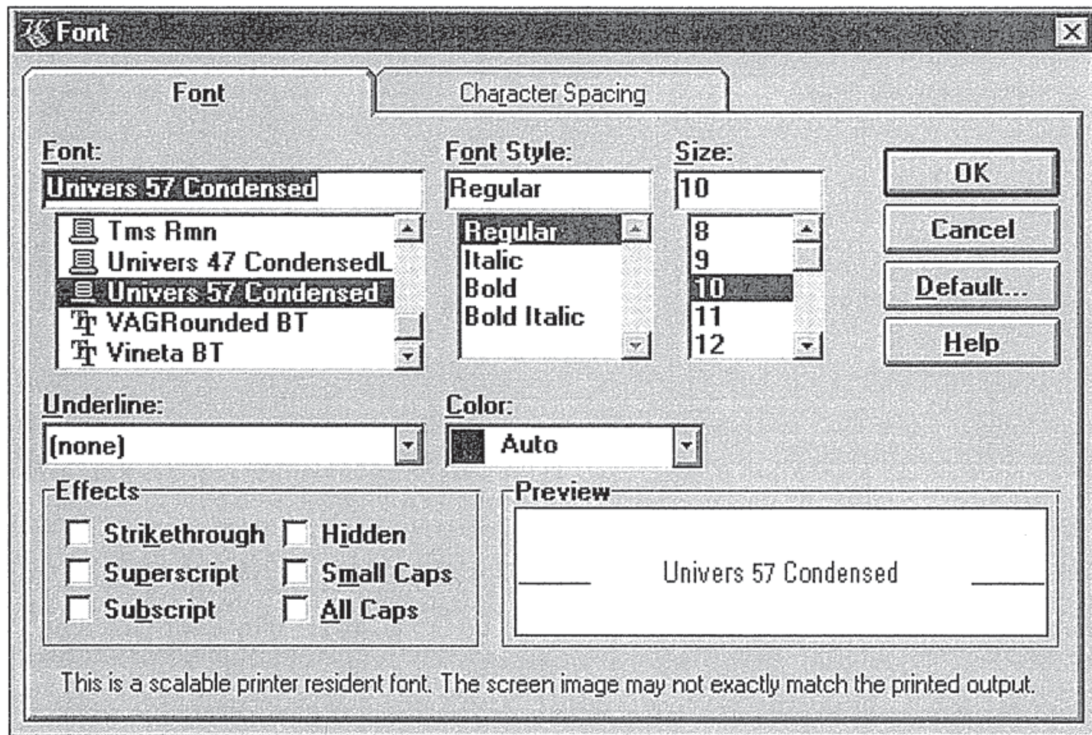


Figure 21-3

The Font dialog box in Word is a properties dialog. It reflects all of the characteristics of the current selection, as they relate to typography. When the user changes something on this dialog, the text qualities of the selection will change, but no functions are executed. The process is essentially a passive, configuring one, rather than an active, process-oriented one. This dialog reflects the best and worst of contemporary dialog design. The preview box is great, but why can't the font combobox in the upper left corner use the actual fonts, too? The OK and CANCEL buttons are in the upper right corner, an emerging Microsoft standard. Upper right corner?! English-speaking people and many others read from upper left to lower right, so the terminating command buttons should be in the lower right corner. Another big mistake is that the terminating buttons are on the panes rather than on the common dialog background. This arrangement is ambiguous. Does the OK button mean to accept this pane or accept the entire dialog box? All terminating buttons should be placed outside any tabbed panes.

It is easy to think of "properties" as an artifact of object-oriented programming because, in that world, that is how we refer to the characteristics of things. But properties are just the aspects of any artifact in a program—the characteristics of a document or chunk of data.

A properties dialog box generally controls the current selection. This follows the object-verb form: The user selects the object and then, via the property dialog, selects new settings for the selection.

Function dialog boxes

Function dialog boxes are usually summoned from the menu. They are most frequently modal dialog boxes, and they control a single function like printing, inserting or spell checking.

Function dialog boxes not only allow the user to launch an action, but they often also enable the user to configure the details of the action's behavior. In many programs, for example, when the user requests printing, the user uses the print dialog to specify which pages to print, the number of copies to print, which printer to output to and other settings directly relating to the print function. The terminating OK button on the dialog not only closes the dialog but also initiates the print operation.

This technique, though common, combines two functions into one: configuring the function and invoking it. Just because a function *can* be configured, however, doesn't necessarily mean that a user will *want* to configure it before every invocation. I prefer to see these two functions accessible separately.

Many functions available from modern software are quite complicated and have many configurable options. Their controlling dialog boxes are correspondingly complicated, too.

The example shown in Figure 21-4 is from PowerPoint. The user first configures the operation by choosing a file, then executes the configured command by pressing the terminating command button: OK. It is very tempting to make that terminating button say PRINT instead. Fight the urge. It may seem more logical, but the loss of a consistently captioned terminating command button is too great a price to pay. If the dialog's caption bar text is appropriate, it will read like an English phrase, telling the user exactly what will happen: "Print the document".....OK.

Bulletin dialog boxes

The **bulletin dialog box** is a devilishly simple little artifact that is arguably the most abused part of the graphic user interface.

The bulletin is best characterized by the ubiquitous error message box. There are well-defined conventions for how these dialogs should look and work, primarily because the MessageBox call has been present in the Windows API since Version 1.0. Normally, the issuing program's name is shown in the caption bar, and a very brief text description of the problem is displayed in the body. A

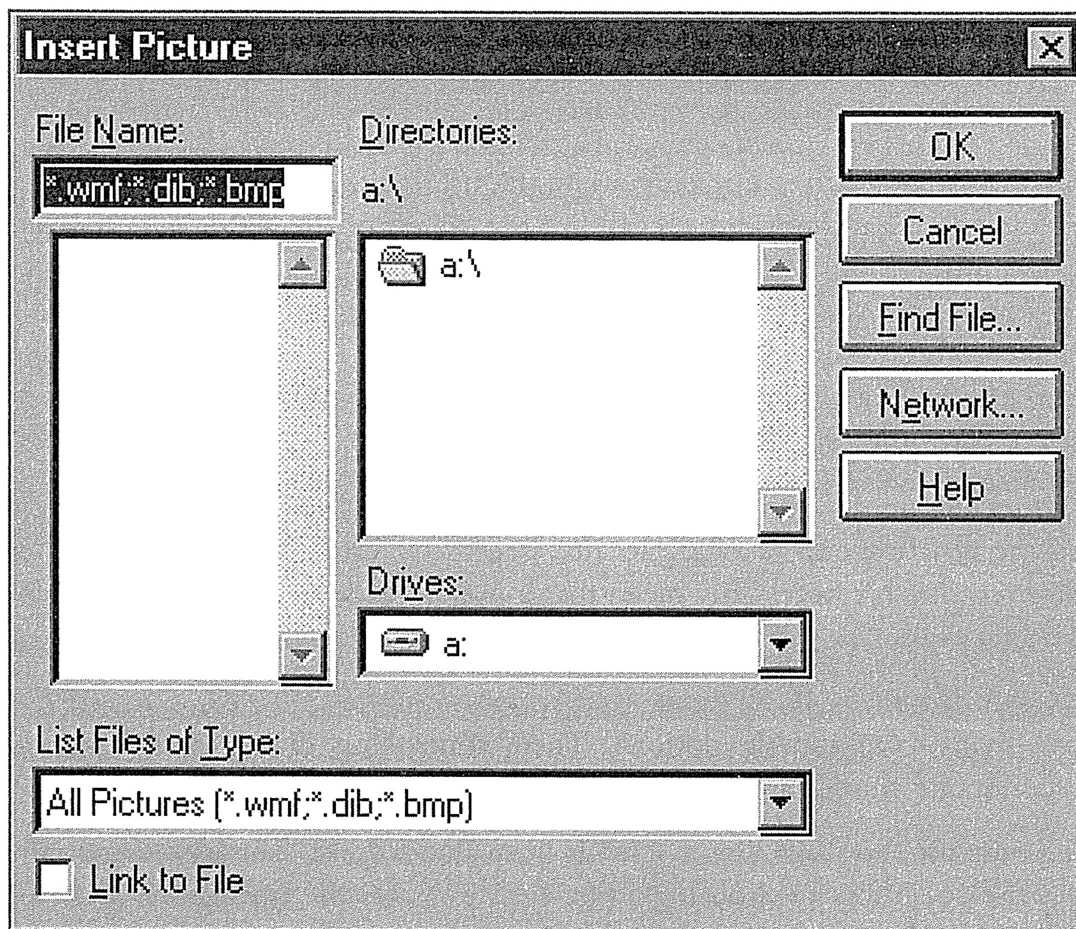


Figure 21-4

The Insert Picture dialog box from PowerPoint is a function dialog box. It is quintessentially modal, allowing the user to first configure the function by choosing a file. Nothing happens, however, until the OK button is pressed. The dialog does not have an effect on an object but rather performs an operation.

graphic icon that indicates the class or severity of the problem along with an OK button usually completes the ensemble. Sometimes a button to summon online help is added. An example from Word is shown in Figure 21-5.

Both property and function dialog boxes are always intentionally requested by the user—they serve the user. Bulletins, on the other hand, are always issued unilaterally by the program—they serve the program. Both error and confirmation messages are bulletins, and we will cover both variants in detail in Part VII, "The Guardian."

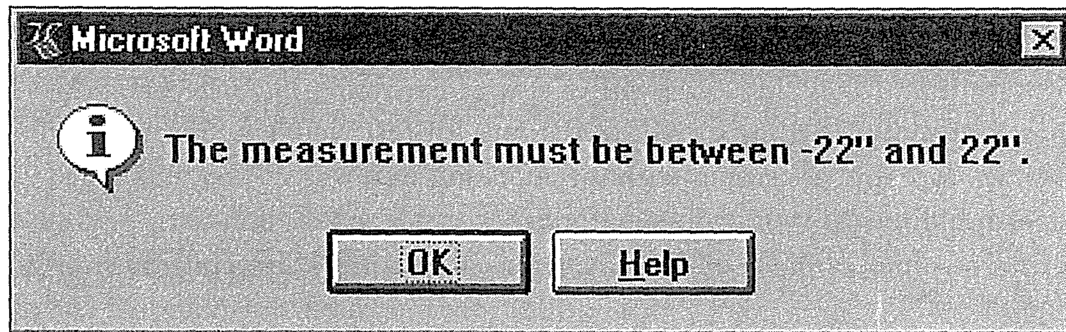


Figure 21-5

Here's a typical bulletin dialog box. It is never requested by the user but is always issued unilaterally by the program when it fails to do its job. The program simply decides that it is easier to blame the user than it is to go ahead and solve the problem. Users interpret this as saying "The measurement must be between -22 inches and 22 inches, and you are an incredible buffoon for not knowing that basic, fundamental fact. You are so stupid, in fact, that I'm not even going to change it for you!"

Process dialog boxes

Process dialog boxes, like bulletins, are erected at the program's discretion rather than at the user's request. They indicate to the user that the program is busy with some internal function and that it has become stupid.

The process dialog box alerts the user to the program's inability to respond normally. It also warns the user not to be overcome with impatience and to resist banging on the keyboard to get the program's attention.

All of today's desktop computers have a single-microprocessor central processing unit, or CPU. CPUs can only do one thing at a time, although through concurrency—where the CPU does a tiny bit of work on several programs in a kind of round-robin—they can seemingly execute multiple software threads at the same time. A problem arises when hardware becomes part of the equation. The CPU cannot use concurrent programming techniques if a chunk of hardware ties down the system for a long time. What this means is that when the computer must access the disk or the network, it cannot continue with other processing until the hardware responds. If the CPU requests something big from the disk—something that takes ten seconds, say—the entire computer comes to a grinding halt for the entire ten seconds; the computer gets stupid. This is true even in a preemptive multi-tasking environment like Windows 95.

The CPU can preempt software threads, but it still cannot preempt a hardware process.

Software that makes significant use of slower hardware, like networks, disks or tapes, will always become stupid, sometimes for relatively long periods of time. Software can also get stupid without accessing hardware. Programs that must perform billions of instructions before they can interact with users—anachronistically named "corebound" programs—frequently get stupid during their calculations.

In any case, when a program begins a process that will take perceptible quantities of time, as measured by the human user, the program must make it clear that it is busy and not just being rude. If the program does not indicate this, the user will interpret it as rudeness at best, or at worst, will assume the program has crashed and take drastic action.

> ***Design tip:*** The program must inform the user when it gets stupid.

As we discussed in Chapter 15, many programs rely on active wait cursor hinting, turning the cursor into an hourglass. This solution springs big leaks in the multi-threaded world of Windows 95, and a better solution is a process dialog box (better yet, an equivalent progress meter built directly into the program's main window).

Each process dialog box has four tasks:

- Make clear to the user that a time-consuming process is happening
- Make clear to the user that things are completely normal
- Make clear to the user how much more time the process will take
- Provide a way for the user to cancel the operation

The mere presence of the process dialog box satisfies the first requirement, alerting the user to the fact that some process is occurring. Satisfying the third requirement can be accomplished with a **progress meter** of some sort, showing the relative percentage of work performed and how much is yet to go. Satisfying the second requirement is the tough one. The program can crash and leave the dialog box up, lying mutely to the user about the status of the

operation. The process dialog box must continually show, via time-related movement, that things are progressing normally. The meter should show the progress relative to the total time the process will consume rather than the total size of the process. Fifty percent of one process may be radically different in time than 50% of the next process.

The user's mental model of the computer executing a time-consuming process will quite reasonably be that of a machine turning or reciprocating. A static dialog box that merely announces that the computer is "Reading disk" may *tell* the user that a time-consuming process is happening, but it doesn't *show* that this is true. The best way to show the process is with some animation on the dialog box. In the Explorer in Windows 95, when files are moved, copied or deleted, a process dialog box shows a small animated cartoon of papers flying from one folder to another folder or the wastebasket (see Figure 21-6). The effect is remarkable: the user gets the sense that the computer is really *doing* something. The sensation that things are working normally is visceral rather than cerebral, and users, even expert users, are reassured. The progress bar of blue squares satisfies—barely—the third requirement by hinting at the amount of time remaining in the process. Although this is one of the best designed features in Windows 95, it can still use some improvement. There is one dialog box per operation, but the operation can affect many files. The dialog should also show an animated countdown of the number of files in the operation. Right now, the blue squares in the progress bar just show the progress of the single file currently being transferred. Regardless, I'm tickled that Microsoft got this one so right.

Notice that the copy dialog in Figure 21-6 also has a CANCEL button. Ostensibly, this satisfies requirement number four, that there be a way to cancel the operation. The user may have second thoughts about the amount of time the operation will take and decide to postpone it, so the CANCEL button allows him to do so. However, if the user realizes that he issued the wrong command and wishes to cancel the operation, he will not only want the operation to stop but will want all trace of the operation to be obliterated.

If the user drags 25 files from directory Alpha to directory Bravo, and halfway through the move realizes that he really wanted them placed in directory Charlie, he can push the CANCEL button. Unfortunately, all that does is *stop* the move at its current state and abandons the remainder of the moves. In other words, if the user presses the CANCEL button after 10 files have been copied, the remaining 15 files are still in directory Alpha, but the first 10 are now in

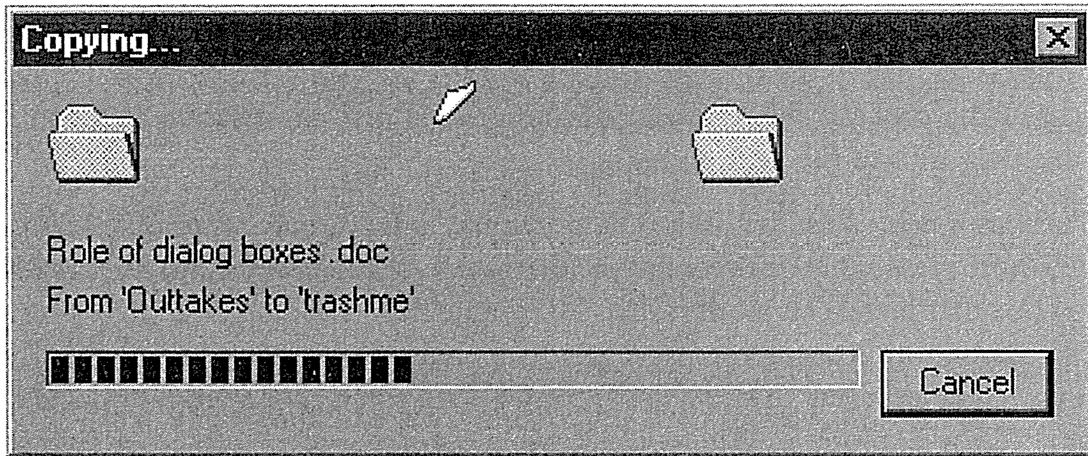


Figure 21-6

Hooray, Microsoft! They really got this one right. For any move, copy or delete operation in the Explorer, they show a well-designed process dialog box. The dialog uses animation to show paper documents flying out of the folder on the left into the folder (or wastebasket) on the right. The user's mental model is one of the things moving inside the computer, and this little gem actually shows things moving. It is refreshing to see the outside of the computer reflect the inside of the computer in users' terms for once. The only thing that worries me is whether Microsoft merely spawns an animation thread or actually ties the animation to the copy—in other words, if the program crashes, does the animation stop, too? Or will pages just keep on flying from one folder to another forever?

directory Bravo. This is *not* what the user wants. If the button says CANCEL, it should mean *cancel*, and that means "I don't want any of this to have any effect." If the button were to accurately represent its current action, it would say ABANDON, ABORT or STOP. Instead, it says CANCEL, so cancel is what it should do.

If the user pressed the CANCEL button, the program should really *cancel* the effects of the operation by undoing the already-done part. This may mean some significant buffering is needed, and the cancel operation could easily take more time than the original move, copy or delete. But isn't this rare event one when the time required is easily justified? In the Explorer, the program can completely undo a copy, move or delete (attaboy!), so there is no reason why the CANCEL button couldn't also undo the portion that had already been performed.

A good alternative would be to have two buttons on the dialog, one labeled CANCEL and the other labeled ABANDON. The user could then choose the one he really wants.

Dialog Box Etiquette

In the last chapter, we discussed the larger design issues concerning dialog boxes. In this chapter, we zoom in closer to examine the way well-behaved dialogs should act. Even an appropriate dialog box can exhibit behavior that is unexpected or irritating. By attending to the details, we can change them from rude interrupters to polite and helpful attendants.

You rang?

If you recall, we divided dialog boxes into four types: property, function, bulletin and process. One of the most important differences between these types is the way they are summoned. The first two are shown only at the user's explicit request, while the latter two are issued unilaterally by the program. When you say, "Jeeves, come in here," you expect the butler to step smartly into the room and plainly and immediately offer his services. On the other hand, when Jeeves wants to ask for a raise, you want him to wait

obsequiously until you are relaxing in a pleasant mood before interrupting your reverie to impose his own needs. In this butlerian spirit, bulletin and process dialogs should show much more deference than property or function dialogs. Unfortunately, the opposite is often true.

A user-requested dialog may be large and place itself front-and-center on the screen. No unrequested dialog should be so brassy, however. It should be smaller, more compact in its use of space, and should appear off to one side of the screen so as not to obstruct the user's view of things.

Who's processing?

Actually, the need for bulletin and process dialog boxes is unclear. They are as common as weeds in contemporary software, and about as useful, too. In Chapters 28 and 29, we'll discuss ways of eliminating bulletin dialog boxes, but what can we do with process dialogs?

The answer to that question is found by asking who is doing the processing. Because a dialog is a separate room, we must ask whether the process reported by the dialog is a function separate from that on the main window. If the function is an integral part of what is shown on the main window, then the status of that function should be shown on the main window instead. For example, the Windows 95 flying pages dialog that was shown in Figure 21-6 is attractive and appropriate, but isn't copying a file fundamental to what the Explorer does? The animation in this case could have been built right into the main Explorer window. The little pages could fly across the status bar, or they could fly directly across the main window from directory to directory.

Process dialogs are, of course, much easier to program than building animation right into the main window of a program. They also provide a convenient place for the CANCEL button, so it is a very reasonable compromise to fling up a process dialog for the duration of a time-consuming task. But don't lose sight of the fact that by doing this we are still going to another room for a this-room function. It is an easy solution, but not the correct solution.

The caption bar

If a dialog box doesn't have a caption bar, it cannot be moved. All dialog boxes should be movable so they don't obscure the contents of the windows they overlap. Therefore, all dialog boxes should have caption bars. Is that clear? Even the Windows style guide almost agrees on this point, saying, "In general, an application should use only movable dialog boxes."

Design tip: All dialog boxes should have caption bars.

There seems to be some belief that system modal messages (which, of course, you will *never* create) don't have to have caption bars, because they are often used to report fatal errors. I guess the programmer's reasoning goes: "Well, the system is crashed, so why bother to let them move the dialog around?" Of course, when your system crashes is precisely the time you might need to get a good look at what was on your screen before you reboot. After all, you will probably lose whatever was there.

There also seems to be widespread confusion about what text string to put in the caption bar of a dialog box. Some people think it should be the name of the function, while others think it should be the name of the program. The belt-and-suspenders crew tends to use both. The correct answer is very simple: neither of these.

If the dialog box is a function dialog, the caption bar should have the name of the function—the verb, if you will. For example, if you request "Break" from the "Insert" menu, the caption bar of the dialog should say "Insert Break." What are we doing? We are *inserting a break*! We are not "breaking," so the caption bar should not say "Break." A word like that could easily scare or confuse somebody.

Design tip: Use verbs in function dialog caption bars.

I would go so far as to say that when the function will operate on some selection, the caption bar should indicate what is selected to the best of its ability. For example, if you select a sentence "Smilin' Ed is dead," and invoke the "Font" item from the "Format" menu, the dialog's caption bar should say "Format font for 'Smilin' Ed is dead.'" If you've selected text that's too big to fit on the caption bar, it should show the first and last couple of words of the selection separated by ellipses. If nothing is selected, the caption should say "Format font for future text."

Design tip: Use object names in property dialog caption bars.

If the dialog box is a property dialog, the caption bar should have the name or description of the object whose properties we are setting. The properties dialogs in Windows 95 work this way. When I request the Properties dialog for a directory named "Backup," the caption bar says "Properties for Backup."

Transient posture

If dialog boxes were independent programs, they would be transient-posture programs. As you might expect, dialog boxes should then look and behave like transient programs, with bold, visual idioms, bright colors and large buttons. On the other hand, transient programs borrow their pixels from sovereign applications, so they must never be wasteful of pixels. The imperative to be large is constantly at war with the imperative to be small. One solution is to make each of the individual gizmos slightly larger, but to make sure that the dialog itself wastes no additional space.

Design tip: Dialogs should be as small as possible, but no smaller.

A few years ago, Borland International popularized a standard by creating extra-large buttcons with bitmapped symbols on their faces: a large red "X" for CANCEL, a large green checkmark for OK, and a big blue question mark for HELP. They were cleverly designed and very attractive—at first. Most people now find them wasteful of space and with good reason. The icons on the buttcons worked well to visually identify themselves, well enough that the extra size wasn't necessary. Borland now uses the same bitmaps on buttons of a more conventional size, which is a much better solution. The visual images accomplished the job just fine without the need to waste precious pixels.

Obscuring the parent window can be avoided by always being conservative of space. Dialog boxes should never take more room than they need. Pixels remain the most limited resource in modern desktop computers, and dialog boxes can easily overstep the boundaries of good taste by sprawling across the screen. Compare the space efficiency of the CompuServe Navigator dialog in Figure 22-1 to the one from Word in Figure 22-2.

Checkboxes are a relatively space-inefficient gizmo: the accompanying text requires a lot of dedicated space. Compared to the text of checkboxes, buttcons can be crammed together like sardines.

Reduce excise

Dialog boxes can be a burden on the user if they require a lot of excise—unnecessary overhead, which we discussed in Chapter 13. The user will soon tire of having to always reposition or reconfigure a dialog box every time it appears.

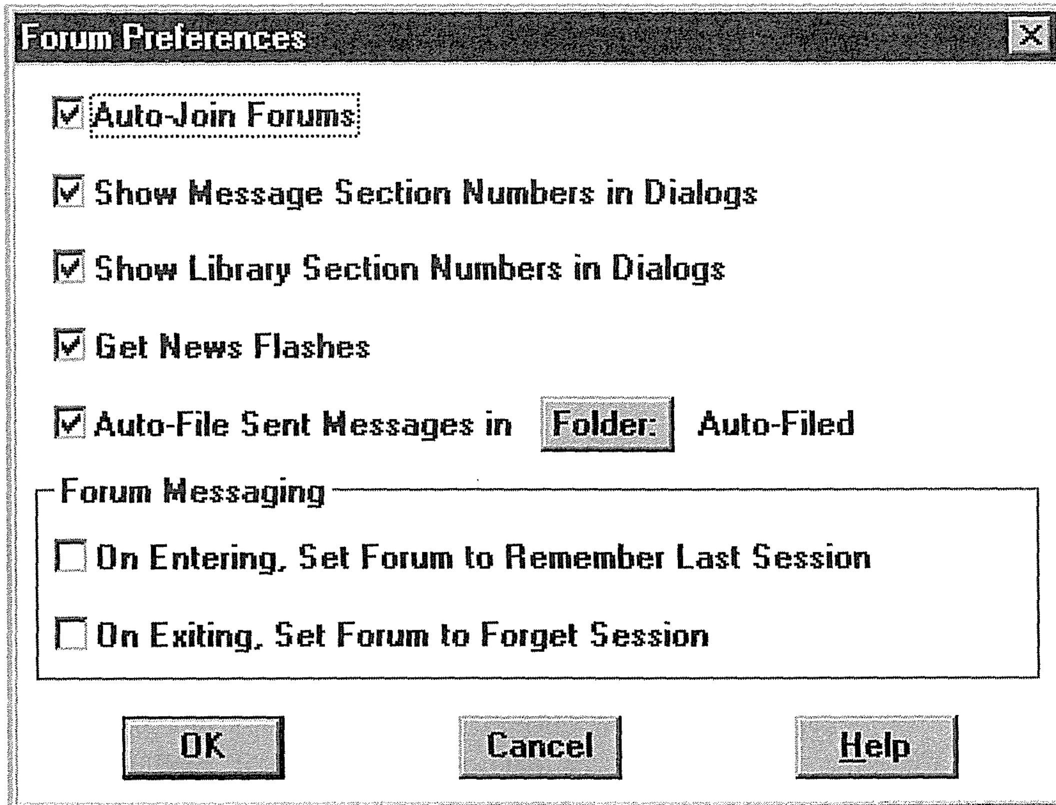


Figure 22-1

Here is a properties dialog box from CompuServe Navigator for Windows (Version 1.0). The sprawling checkboxes consume a lot of space. At least it has a caption bar, so you can move it out of the way.

The duty of the dialog box designer is to assure that the excise is kept to a bare minimum, particularly because dialog boxes are only supporting actors in the interactive drama.

The most usual areas where dialog boxes fail to reduce excise are in their geographical placement and their state. Dialogs should always remember where they were placed the last time, and they should return to that place automatically. Most dialogs also start out fresh each time they are invoked, remembering nothing from their last run. This is an artifact of the way they are implemented: as subroutines with dynamic storage. We should not let these implementation details so deeply affect the way our programs behave. Dialogs should always remember what state they were in the last time they were invoked and return to that same state. If the dialog was expanded or a certain tab was

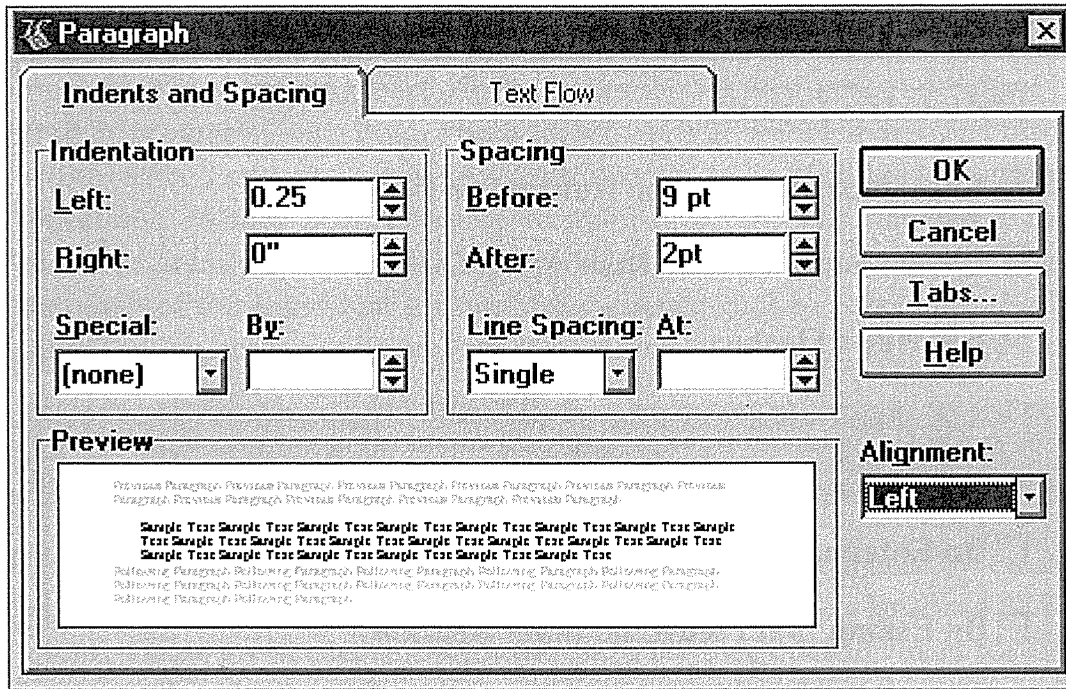


Figure 22-2

A typical function dialog box from Microsoft Word shows an excellent use of space. The controls are compact and very conservative of space. Compare this with the previous figure (Figure 22-1). Notice, also, their willingness to use graphic objects instead of just canned, text-based gizmos like edit fields, checkboxes and push-buttons.

selected, the dialog should return the exact same way on subsequent visits. In Chapter 14, I talked in more detail about how to apply memory to this type of problem.

The same idea can be applied to the contents of input fields. If a checkbox was checked last time, the dialog box should remember and come up with the box checked next time. Chances are good that the settings used the last time will be used the next time, too.

Know if you are needed

The most effective way that a dialog box can reduce excise is to not even bother appearing if it is not needed. If there is some way for the dialog box to be smart enough to know whether it is really necessary, the program should—by all means—determine this and prevent the user from having to merely dismiss the unneeded box: an action that is pure excise.

For example, in Word, I always save my document just before I print it, and I often print it just before closing it. In other words, I frequently want to SAVE, PRINT and CLOSE a document. Unfortunately the repagination involved in printing inadvertently marks the document as changed. This means that the program always asks me if I want to save it when I issue the CLOSE command, even though I just did! The program should pay attention! Of course, I want to save the document before closing. Not only should it not ask this question at all, it should be able to see from my actions that *I* didn't change it, the *program* did. The entire invocation of this dialog box is excise.

The same thing is true of bulletin dialogs that tell me that the program has completed some function normally. If it was so normal, the program shouldn't need to resort to the excise of a dialog box that stops the proceedings with idiocy.

If a program uses a dialog box to offer me a selection of options every time I ask for a certain function, and I always use the same options, the program shouldn't bother to even put up the dialog box. It should be able to recognize the pattern and remove the unnecessary step. Of course, it would have to inform me first, so I am not surprised, and it should give me the option to override its decision.

Terminating commands for modal dialog boxes

Every modal dialog box has one or more terminating commands. Most modal dialog boxes have three: the OK and CANCEL buttons and the close box on the caption bar. The OK button means "accept any changes I have made, then close the dialog and go away." The CANCEL button means "reject any changes I have made, then close the dialog and go away." This is such a simple and obvious formula, such a well-established standard, that it is inconceivable that anyone would vary from its familiar, trustworthy, well-trod path. Yet, for inexplicable reasons, many user interface designers do diverge from this simple formula, always to the detriment of their product and the despair of their users.

The modal dialog box makes a contract with the user that it will offer services on approval—the OK button—and a bold and simple way to get out without hurting anything—the CANCEL button. These two buttons cannot be omitted without violating the contract, and doing so deflates any trust the user might

have had in the program. It is extremely expensive in terms of stretching the user's tolerance. Never omit these two buttons or change their legends.

Design tip: Offer OK and CANCEL buttons on all modal dialog boxes.

A colleague countered this tip by suggesting that a dialog box asking if the user wants to "Cancel Reservation?" would cause problems when it appears with an OK and CANCEL button. What does it mean to say CANCEL to Cancel? Good question, and the solution to the problem is to never ask questions like that. The example is a particularly ugly one for several reasons, notably because it is a confirmation dialog. Besides, if you ever need to ask a question like that—and you shouldn't—don't express it using the same words that are in the termination keys. With "Cancel Reservation?" the user must respond with the word CANCEL to avoid canceling. Confusing? You bet! Instead, the question should be stated like this: "Discard the Reservation?" Better yet, we'll talk about how to eliminate confirmation dialogs entirely in Part VII.

Design tip: Never use terminating words in dialogs.

The design tip "Offer OK and CANCEL buttons on all modal dialog boxes" applies to function and property types. Bulletin dialogs reporting errors—those hateful things—can get away with just an OK button (as if the user wants to collude in the program's failure!). Process dialogs only need a CANCEL button so the user can end a time-consuming process.

The OK and CANCEL buttons are the most important controls on any dialog box. These two buttons must be immediately identifiable visually, standing out from the other controls on the dialog box, and particularly from other action buttons. This means that lining up several, visually-identical buttons, including OK and CANCEL is *not* the right thing to do, regardless of how frequently it is done (id est: the stack of buttons in Figure 21-4). Even from companies who should know better, the OK and CANCEL buttons are buried in groups of other, unrelated buttons, and their familiar legends change with depressing frequency.

The CANCEL button, in particular, is crucial to the dialog box's ability to serve its pedagogic purpose. As the new user browses the program, he will want to examine the dialogs to learn their scope and purpose, then CANCEL them so as not to get into any trouble. For the more experienced user, the OK button

begins to assume greater import than the CANCEL button. The user calls the dialog box, makes his changes, and exits with a confirming push of the OK button.

Lately, Microsoft has shown off a new standard for terminating buttons. They demand that the OK button be in the upper right corner of the dialog and that the CANCEL button be positioned immediately below it, with the HELP button below that. Unfortunately, Microsoft's style troopers have chosen poorly: The majority of users read from upper left to lower right, so the terminating buttons make more sense in the lower right of the dialog box. Microsoft has also gone for the executive gray look, and the terminating buttons are not visually identified by any unique color, bitmaps or even a unique font or typesize. They just blend right in with the other buttons on the dialog—too bad.

I'm much more concerned with consistency in placement of these buttons than I am in the particular location they occupy. However, I'm not indifferent to their placement. The OK button should be placed in the lower right corner of the dialog box, and the CANCEL button should be placed immediately to its left (or immediately above it). The user can then dependably know that an affirmative ending of the dialog can be had by going to the extreme lower-right corner.

The close box

Because dialog boxes are windows with caption bars, they have another terminating idiom. Clicking in the closebox in the upper right corner (or double-clicking the system menu box in the upper left corner in Windows 3.x) terminates the dialog box. The problem with this idiom is that the disposition of the user's changes is unclear: Were the changes accepted or rejected? Was it the equivalent of an OK or a CANCEL? Because of the potential for confusion, there is only one possible way for programs to interpret the idiom: as CANCEL. Unfortunately, this conflicts with its meaning on a modeless dialog, where it is the same as a CLOSE command. The close box is needed on a modeless dialog but not on a modal dialog box. So, to avoid confusion, the close box should *not* be included in modal dialogs.

Design tip: Don't put close boxes on modal dialogs.

If the user expects an OK and gets a CANCEL, he will be surprised and will have to do the work over—and he will learn. On the other hand, if the user expects

a CANCEL and gets an OK, he will still have to do the work over, but this time he will be angry. Don't let this situation arise.

The HELP button requests context-sensitive help but doesn't terminate the dialog, so it isn't a terminating button. It is so often grouped with the terminating buttons that it has assumed the same importance by association. Online help, however, is not as important as the terminating commands. Putting help adjacent to them is weak, but not harmful, and it has the power of a familiar standard. In Windows 95, Microsoft is showing that they understand this problem. As you can see in Figure 22-3, they are beginning to move help away from the OK and CANCEL terminating buttons, putting it on the caption bar. Up there, it is on an area common to all dialogs but clearly separated from the very special terminating commands.

Keyboard shortcuts

Many dialogs offer services that are frequently used or used repetitively, like those for REPLACE or FIND. As users gain experience with the program, they will appreciate the presence of keyboard shortcuts for these frequently used dialogs. There are usually enough keys to go around, and there is no reason why a given function should have just a single keyboard shortcut. A function like FIND should be callable with a CTRL+F keystroke as well as a special function key, like F2. REPLACE could be CTRL+R and F3.

Users learn these shortcuts either from the help system or from the menus. Usually, these shortcuts go unnoticed until they are desired. New users go directly to the menus, and it is only after they find themselves actively searching for faster ways to operate that they discover them. And they will then be grateful that you had the foresight to put those shortcuts in for them. It can really please the power-user crowd, and this crowd will have a big influence over new users.

Tabbed dialogs

The latest user interface idiom to take the world of commercial software by storm is the **tabbed dialog**, sometimes called a "multi-pane dialog."In less than two years, tabbed dialogs, as shown in Figure 22-4, have gone from a virtually unknown idiom to a well-established standard. When an idiom has merit, it is widely copied, and the tab gizmo has been such a blessing to dialog box

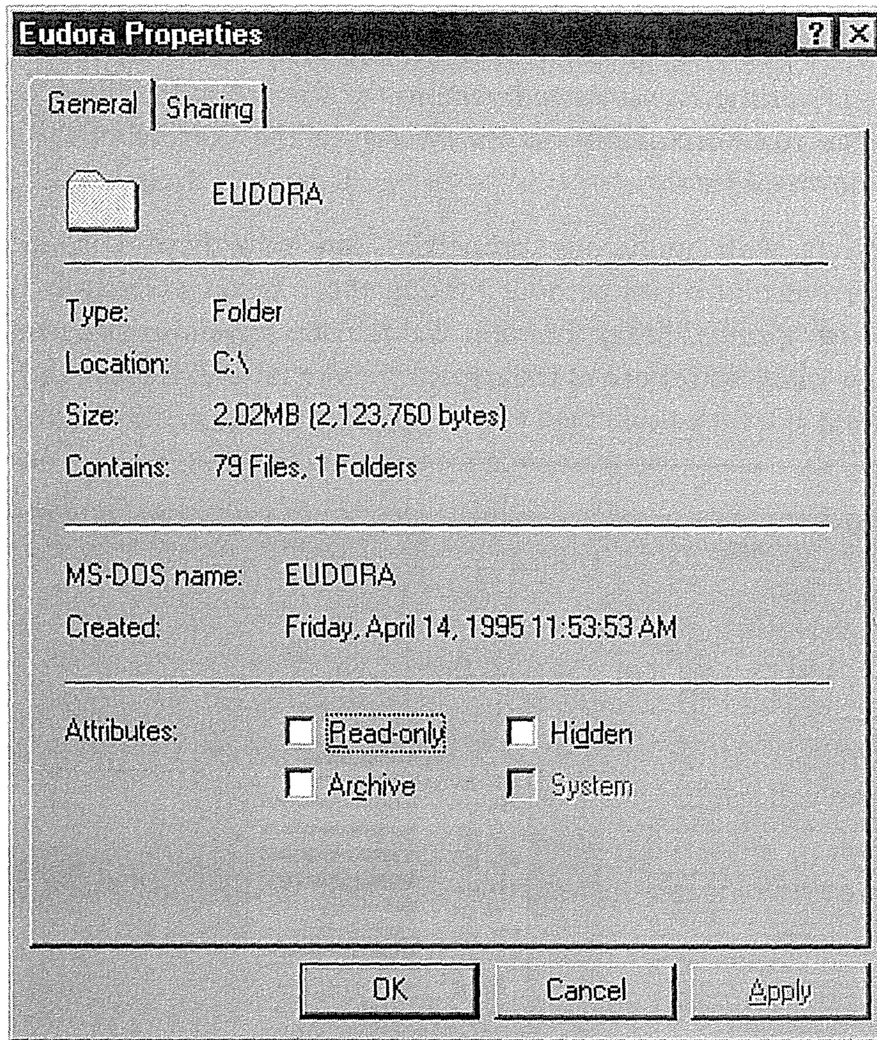


Figure 22-3

This properties modal dialog box from Windows 95 shows how Microsoft has finally realized that Help is not a terminating command. They removed it from the suite of terminating buttons and put it on the caption bar near the close box. This is certainly an improvement, but then they went ahead and added a newcomer to the terminating-button row: Apply. There is no use for an apply function on this pane, but it is applicable on the other pane, Sharing. Why not put the Apply button on the pane where it means something and keep it out of the way of the terminating buttons?

designers that it has become a standard part of Windows 95. We can expect that developers will soon embrace it with even more vigor than they already have.

Tabbed dialogs allow all or part of a dialog to be set aside in a series of fully overlapping **panes**, each one with a protruding, identifying **tab**.

Pressing a tab brings its associated pane to the foreground, hiding the others. The tabs can run horizontally across the top or the bottom of the panes or vertically down either side.

Many objects with numerous properties can now have correspondingly rich property dialog boxes without making those boxes excessively large and crowded with gizmos. Many function dialogs that were also jam-packed with gizmos now make better use of their space. Before tabbed dialogs, the problem

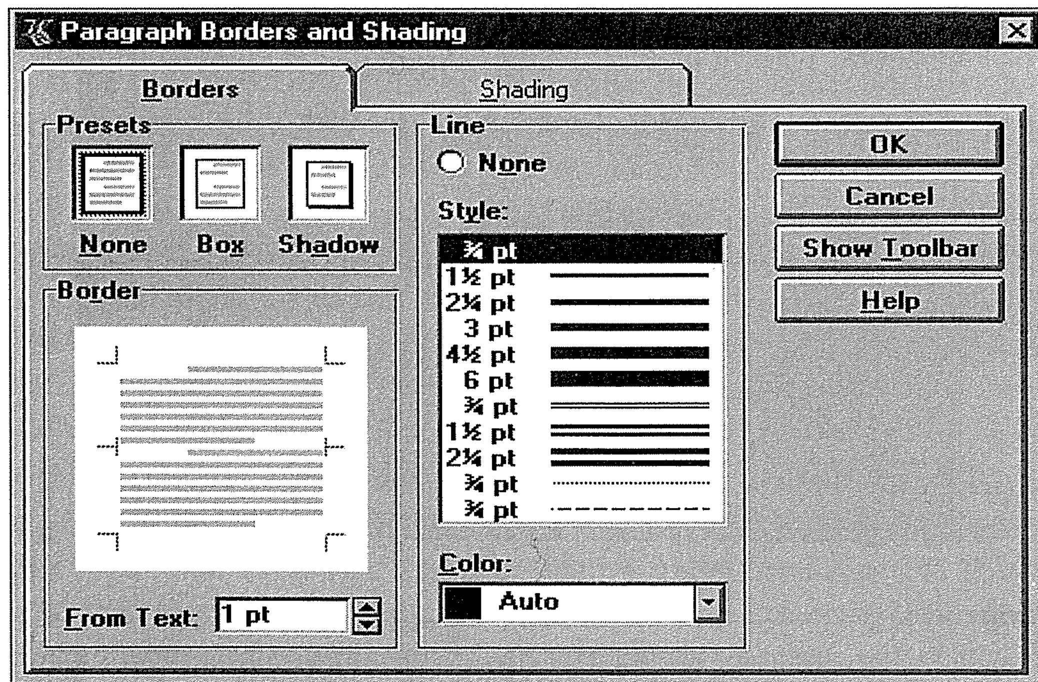


Figure 22-4

This is a tabbed dialog box from Microsoft Word. Combining borders and shading on one dialog box makes sense, if you have a convenient way to do it. Tabbing provides that way. Notice that Microsoft hasn't yet learned to put the terminating command buttons on the background instead of the pane. Putting them on the pane can confuse the user over whether he is canceling the pane or the entire dialog.

was clumsily solved with expanding and cascading dialogs, which I'll discuss shortly.

I believe that the tabbed dialog box is having such success because it follows the user's mental model of how things are normally stored: in a monocline grouping. The various gizmos are grouped in several parallel panes, one level deep.

A tabbed dialog allows you to cram more gizmos onto a single dialog box, but more gizmos doesn't necessarily mean that the user will find it better. The contents of the various panes on the dialog must have a meaningful rationale for being together; otherwise, this capability just degrades to what is good for the programmer, rather than what is good for the user.

The various panes on a dialog can be organized to manage increased depth or increased breadth. For more breadth, each pane covers additional aspects of the main topic, the way borders and shading, in Figure 22-4, both address ways that text is enhanced. For more depth, each pane probes the same aspect of one topic in greater depth. For example, the cascading dialog in Figure 22-7 could be implemented as a tabbed dialog with the custom color factory as a separate pane.

Every tabbed dialog box is divided into two parts, the stack of panes, which I call the **tabbed area**, and the remainder of the dialog outside the panes, which I call the **untabbed area**.

The terminating command buttons must be placed on the untabbed area. If the terminating buttons are placed directly on the tabbed area, even if they don't change position from pane to pane, their meaning is ambiguous. The user may well ask "if I press the CANCEL button, am I canceling just the changes made on *this* pane or all of the changes made on *all* of the panes?" By removing the buttons from the panes and placing them on the untabbed area, their scope becomes visually clear. Microsoft's Office suite has many terminating buttons incorrectly placed in the tabbed area, but the new Windows 95 has them correctly placed in the untabbed area. Expect to see them migrate off the panes in subsequent releases of Office.

Design tip: Put terminating buttons on untabbed area.

Multi-pane dialogs have been around for a while, implemented with a row of push-buttons, radio buttons or other common idioms for switching panes. The new tab gizmo that comes standard with Windows 95 has powerful visual affordances that the other idioms lack. The fact that Microsoft is supporting the standard with code will make it the idiom of choice for dialog box designers.

Because you can cram so many gizmos into a tabbed dialog, the temptation is great to add more and more panes to a dialog. The Options dialog in Microsoft Word, shown in Figure 22-5, is a clear example of this problem. The twelve tabs

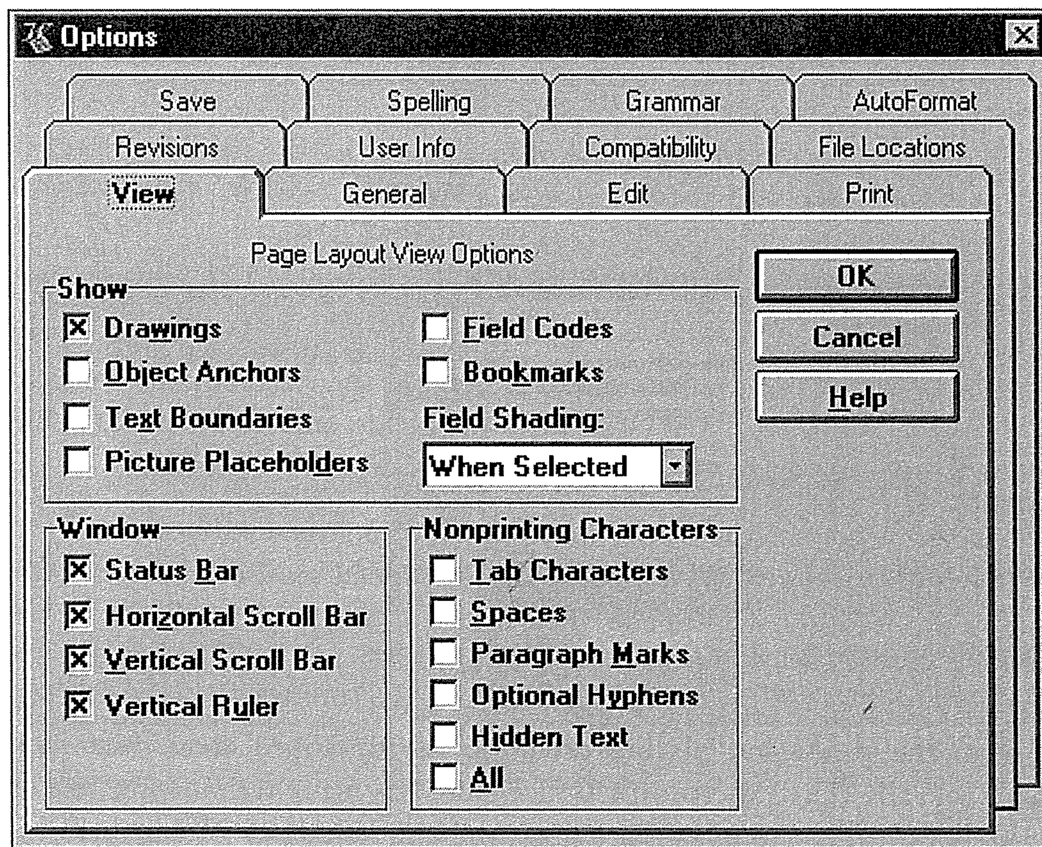
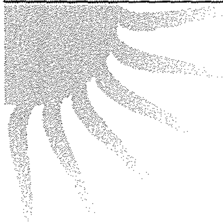


Figure 22-5

The Options dialog in Word is an extreme example of what can be done with tabs. There is certainly a lot of stuff crammed into this one dialog, which is good. The problem is that the tabs move around! The active tab must be on the bottom row, so if you clicked on "Grammar," for example, that row rolls down to the bottom and the other two rows bubble up one level. Everybody hates it when the tabs move underneath the cursor. It's better just to break this up into smaller dialogs.

are far too numerous to show in a single line, so they are stacked three deep. The problem with this implementation, which I call **stacked tabs**, is that, if you click on a tab in the back row, the entire row of tabs moves forward, shunting the other two rows to the back. Very few users seem to be happy with this because it is disconcerting to press on a tab and then have it move out from under the mouse. It works, true, but at what cost?

Stacked tabs illustrate an axiom of user interface design: That all idioms, regardless of their merits, have practical limits. A group of five radio buttons may be excellent, but a group of fifty of them is ridiculous. Five or six tabs in a row are fine, but adding enough tabs to require stacking destroys the usefulness of the idiom. Like overloading your Gremlin with 17 passengers, your performance edge decreases.



All idioms have practical limits.

So far, whenever I ask my colleagues for a viable alternative to the stacked tabs, they can't give me a good answer. They all see the advantage of twelve panes of options grouped in a common place, without incurring all of the other potential nasty problems with that many gizmos: There are no cascading dialogs; the dialog isn't too large; the gizmos are logically grouped; the implementation is simple. Although they accept its advantages, they all recognize its shortcomings, stemming almost completely from the dynamic rearranging of the tabs.

Design tip: Don't stack tabs.

A better alternative would be to just use three separate dialogs with four tabs each. There is little connection between the twelve panes, so there is little need to move between them. The solution lacks a certain programming elegance, but it is much easier for the user.

Expanding dialogs

Expanding dialog boxes were big around 1990 but have declined in popularity since then, largely due to the omnipresence of toolbars and tabbed dialogs. You can still find them in many mainstream applications, although Microsoft has been working hard to eliminate them from both Windows 95 and its applications.

Expanding dialogs "unfold" to expose more controls. The dialog shows a button marked "More" or "Expand," and when the user presses it, the dialog box grows to occupy more screen space. The newly added portion of the dialog box contains added functionality, usually for advanced users. The Color section ofthe Windows 3.x Control Panel is a familiar example of the expanding dialog box, as shown in Figure 22-6.

Usually, expanding dialog boxes allow infrequent or first-time users the luxury of not having to confront the complex facilities that more frequent users don't find upsetting. You can think of the dialog as being in either beginner or advanced mode, which is the cause of one of its more debilitating flaws. When a program has one dialog for beginners and one for experts, it both insults the beginners and hassles the experts.

As implemented, most expanding dialogs always come up in beginner mode. This forces the advanced user to always have to promote the dialog. Why can't the dialog come up in the *appropriate* mode instead? It is easy enough to know which mode is appropriate: it's usually the mode it was left in. If a user expands the dialog, then closes it, it should come up expanded next time it is summoned. If it was put away in its shrunken state last time, it should come up in its shrunken state next time. This simple trait could make the expanding dialog automatically choose the mode of the user, rather than forcing the user to select the mode of the dialog box.

For this to happen, of course, there has to be a "Shrink" button as well as an "Expand" button. The most common way this is done is to have only one button but to make its legend change between "Expand" and "Shrink" as it is pressed. Notice that the Color dialog in Figure 22-6 does not do this: Once the dialog has been expanded, it cannot be shrunk. Normally, changing the legend on a button is weak, because it gives no clue as to the current state, only indicating the opposite state. In the case of expanding dialogs, though, the visual

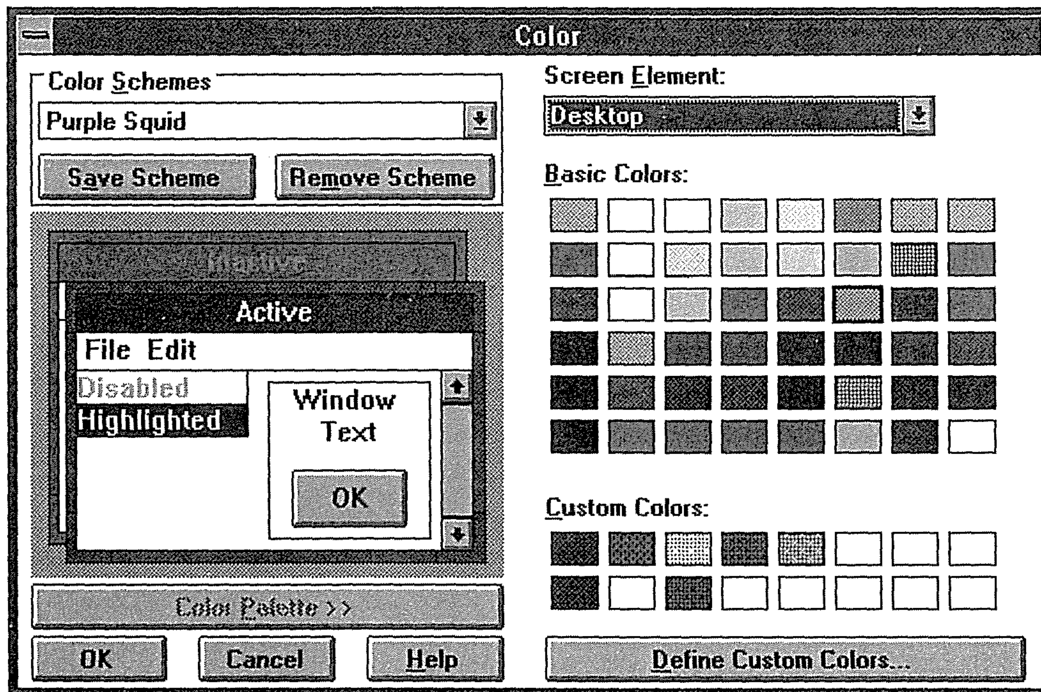


Figure 22-6

The Color tool in the Windows 3.x Control Panel is an expanding dialog box, shown here in its expanded state. When it first comes up, only the left half is visible. By pressing the "Color Palette>>" button, the right side of the dialog becomes visible. The left half is for normal users. Both halves are used by those fussy people who want to configure their own, private color schemes rather than selecting one from the standard palette. It sure would be nice if the dialog were smart enough to remember how I last left it!

nature of the expanded dialog itself is clear enough about the state the dialog is in.

A more mundane reason for its demise is the difficulty of coding an expanding dialog. The Windows API offers little help, whereas Windows 95 comes with the new tab control class pre-written.

Cascading dialogs

Cascading dialogs are a diabolically simple technique whereby gizmos, usually push-buttons, on one dialog box summon up another dialog box in a hierarchical nesting.

The second dialog box usually covers up the first one. Sometimes the second dialog can summon up yet a third one. What a mess! Thankfully, cascading dialogs have been falling from grace, but examples can still be found. Figure 22-7 shows an example taken from Windows 95.

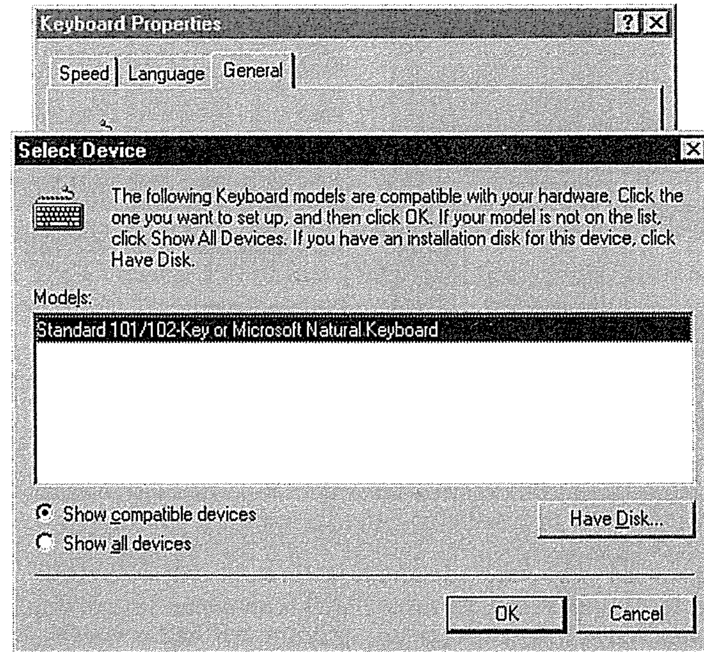


Figure 22-7

You can still find cascading dialogs in Windows 95. Double-click on the "Keyboard" icon in the Control Panel, then select the "General" pane of the dialog. You are greeted with a vast expanse of unused dialog pane space, yet if you press the "Change . . . " button, you get a second, cascading dialog that covers most of the first one. Why wasn't it part of the first dialog instead? The second dialog needlessly obscures the first one, and each dialog now offers up its own pair of terminating buttons resulting in a very unhelpful ambiguity.

It is simply hard to understand what is going on with cascading dialogs. Part of the problem is that the second dialog covers up the first. That isn't the big issue—after all, comboboxes and popup menus do that. The real confusion comes from the presence of a second set of terminating buttons. What is the scope of each CANCEL? What am I OKing?

The strength of tabbed dialogs is handling breadth of complexity, while cascading dialogs are better suited for depth. The problem is that excessive depth is a prime symptom of a too-complex interface. If you find your program

requiring cascading dialogs for anything other than really obscure stuff that your users won't generally need, you should take another look at the overall complexity of your interface.

Examples of cascading are common. Most print dialogs allow print-setup dialogs to be called, and most print-setup dialogs allow print-driver-configuration dialogs to be called. Each layer of dialog box is another layer deeper into the process, and as the user terminates the uppermost dialog, the system returns control to the next lower dialog, and so on.

Cascading dialogs exist because they seem natural to programmers and because they mirror the physical processes underneath them. But this is about as backward a motivation as one can have—it ignores the user's goals and the user's mental model of the process. I'm not saying that cascading dialog boxes should be avoided entirely, but they represent a very weak idiom. Sometimes the situation demands them, as in the print dialog example described above. However, even in that case, I would combine them into one dialog with three tabs, or, at the least, I would combine the first two dialogs into a single one and only maintain the printer driver dialog separately from the main print dialog. Three dialog boxes in cascade is excessive for almost any purpose.

Directed dialogs

Most dialogs are pretty static, presenting a fixed array of gizmos. A variant that I call **directed dialogs** changes and adapts its suite of gizmos based on some user input.

A typical example of a directed dialog can be found in the "Customize" dialog of Windows Word, as shown in Figure 22-8. The gizmos on the face of the dialog change dynamically to adapt to the user's input to other gizmos on the same dialog. Depending on the selection the user makes in the "Categories:" gizmo, not only do the contents of the "Buttons" group control change, but sometimes the group gizmo itself is replaced by a different type of gizmo, usually a listbox. If the selection in the left-hand gizmo calls for buttons to be displayed, we see buttons in a groupbox, but if the selection in the left-hand gizmo calls for a list of fonts or macros, we see a listbox filled with text items. As the figure shows, the directed dialog technique can easily be combined with tabbing.

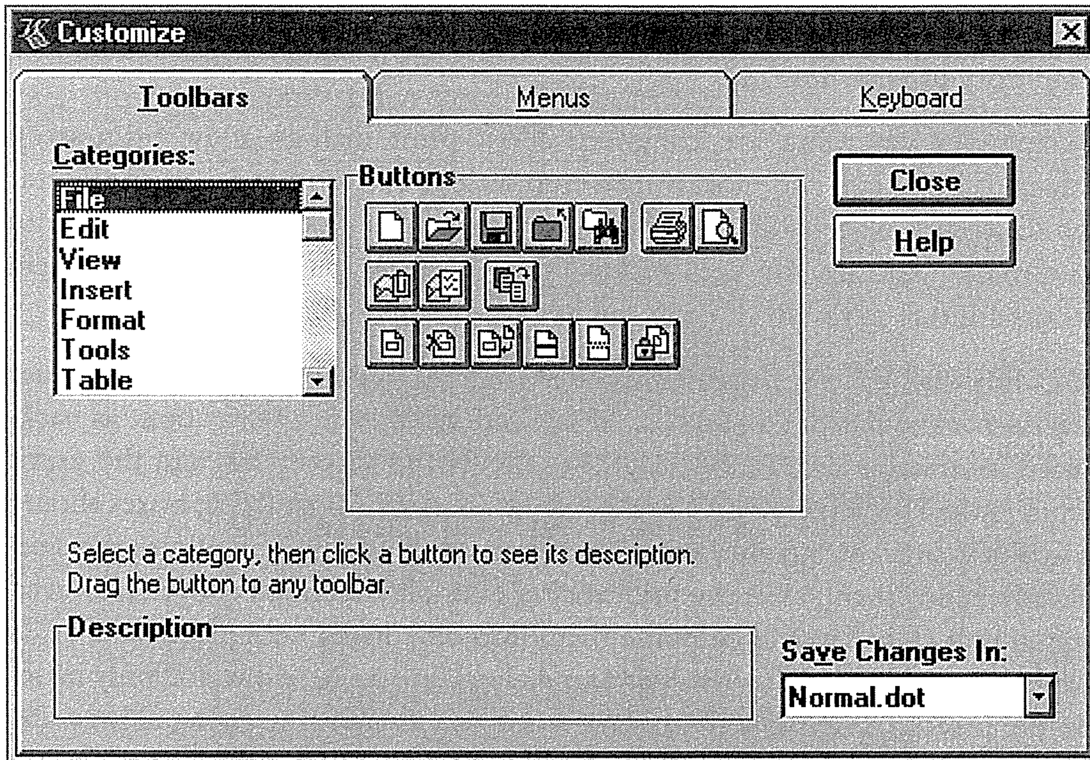


Figure 22-8

Word for Windows' Customize dialog box is an example of a directed dialog box. Depending on what you select in the "Categories" listbox gizmo, the groupbox to its right will either be a collection of buttons (as shown) or a listbox filled with macros, font names or other items. The gizmos on the dialog box configure themselves in real-time to the user's actions.

Programming a directed dialog can get complicated, so it is not done with great frequency. The new nested dialog feature in Windows 95, however, may make it easier to implement. It also may be confusing to the user, as he wonders where certain gizmos went. But I think that it is particularly effective when the user is entering settings in a clearly sequenced manner. For example, in a database access application where the user must select a server, then a database on that server, then a table within that database, a directed dialog would be very appropriate. The structure of the problem at hand calls for the server to be chosen first, so the user would select one from a list. As soon as the selection is made, the dialog would configure itself to include a field for a password if the server required it. If the server wasn't password-protected, the field would be

omitted. As soon as the user selects the database, one or more fields would appear as necessary to allow the user to select the table, its owner and other required information in sequence.

Toolbars

Toolbars are the new kid on the idiom block. Although not an exclusive feature of Windows, they were first popularized on this platform, not the Macintosh, like so many other GUI idioms. The toolbar has great strengths and weaknesses, but they are complementary to those of its partner, the menu. Where menus are complete toolsets with the main purpose of teaching, toolbars are only for frequently used commands and offer little help to the new user.

Visible and immediate

The typical toolbar is a collection of buttcons, usually with images instead of text captions, in a horizontal bar positioned adjacent to and below the menu bar. Essentially, the toolbar is a single, horizontal row of immediate, always visible menu items.

The toolbar really gave birth to the buttcon; a happy marriage between a button and an icon. As a visual mnemonic of

a function, buttcons are excellent. They can be hard for newcomers to interpret, but then, they're not *for* newcomers.

Great ideas in user interface design often seem to spring from many sources simultaneously. The toolbar is no exception. It appeared on many programs at about the same time, and nobody can say who invented it first (if *you* did, thanks!). What is clear, is that its advantages were immediately apparent to all. In a stroke, the invention of the toolbar solved the problems of the pulldown menu. Toolbar functions are always plainly visible, and the user can trigger them with a single mouse click. The user doesn't have to pull down a menu to get to a frequently used function.

Toolbars are not menus

Toolbars are often thought of as just a speedy version of the menu. The similarities are hard to avoid: They offer access to the program's functions, and they form a horizontal row across the top of the screen. Designers imagine that toolbars, beyond being a command vector in parallel to menus, are an *identical* command vector to those on menus. They think that the functions available on toolbars are supposed to be the same as those available on menus.

But the purpose of toolbars is actually quite different from the purpose of menus, and their composition shouldn't necessarily be the same. The purpose of toolbars and their controls is to provide fast access to functions used frequently by those who have already mastered the program's basics. Toolbars offer nothing to beginners and are not supposed to. The menu is where the beginner must turn for help.

> ***Design tip:*** Toolbars provide experienced users with fast access to frequently used functions.

The great strength of menus is their completeness. Everything the user needs can be found somewhere on the program's menus. Of course, this very richness means that they get big and cumbersome. To keep these big menus from consuming too many pixels, they have to be folded away most of the time and only "popped-up" on request. The act of popping up excludes menus from the ranks of visible and immediate commands. The tradeoff with menus is thoroughness and power in exchange for a small but uniform dose of clunkiness applied at every step.

The buttcons on toolbars, on the other hand, are incomplete and inscrutable, but they are undeniably visible and immediate. They are very space-efficient compared to menus. A simple, single click of the mouse on a toolbar buttcon generates instant action. The user doesn't have to search for the function a layer deep in menus—it's right there in plain sight, and one click is all it takes, unlike the mouse-dragging required by menus.

Why not text?

If the buttcons on a toolbar act the same as the items on a pulldown menu, why are the menu items almost always shown with text and the toolbar buttons almost always shown with little images? Why is the sky blue? No, wait, really, there *are* good reasons for the difference, although we almost certainly stumbled on them accidentally.

Text labels, like those on menus, can be very precise and clear—they aren't always, but precision and clarity are their basic purpose. To achieve this, they demand that the user take the time to focus on them and read them. As we discussed in Chapter 4, reading is slower and more difficult than recognizing images. In their pedagogic role, menus must offer precision and clarity—a teacher who isn't precise and clear is a bad teacher. Taking the extra time and effort is a reasonable tradeoff in order to teach.

On the other hand, pictorial symbols are easy for humans to recognize, but they often lack the precision and clarity of text. Pictographs can be ambiguous until you actually learn their meaning. However, once you've learned their meaning, you don't easily forget it, and your recognition remains lightning fast, whereas you still have to read the text every time. In their role of providing quick access to frequently used tools, familiar recognition by experienced users has the highest priority. The pictorial imagery of symbols suits that role better than text does.

Buttcons have all of the immediacy and visibility of buttons, along with the fast-recognition capability of images. They pack a lot of power into a very small space. As usual, their great strength is also their great weakness: the image part.

Relying on pictographs to communicate is all right as long as the parties have agreed in advance what the image means. They must do this because the meaning cannot be guaranteed to be unambiguous.

Many designers think that they must invent visual metaphors for buttcons that adequately convey meaning to first-time users. This is a quixotic quest that not only reflects a misunderstanding of the purpose of toolbars, but reflects the futile hope for magical powers in metaphors, which we discussed in Chapter 5.

The image on the buttcon *doesn't* need to teach the user its purpose; it merely needs to have a bold and visual identity. He will have already learned its purpose through other means. This is not to say that the designer shouldn't strive to achieve both ends, but don't fool yourself: it can't be done very often. It's a lot easier to find images that represent *things* than it is to find images that represent actions or relationships. A picture of a trash can, printer or chart is pretty easy to interpret, but what icon do you draw to represent *apply style* or *cancel* or *connect* or *merge* or *convert* or *measurement* or *adjust*? Then again, perhaps the user will find himself wondering what a picture of a printer means. It could mean find a printer, change the printer's settings or report on the status of the printer. Of course, once he learns that the little printer means "print one copy of the current document on the active printer now," he won't have trouble with it again.

Modern programs like Microsoft Word offer a small library of predesigned icons to select from when customizing the toolbars. I wanted a buttcon that would insert today's date into a document and none of the predesigned ones particularly communicated "date" to me, so I just used a big, yellow smiley face. Its sole virtue is that once you know what it does, you don't forget it or confuse it with anything else.

The problem with both

It might seem like a good idea to label buttcons with both text and images. There is not only logic to this argument, but precedent, too. I've seen many programs that do this. The original icons on the Macintosh desktop had text subtitles. Icons are really useful for allowing quick classification, but beyond that, we need text to tell us *exactly* what the object is for.

The problem is that using both text and images is very expensive in terms of pixels. Besides, toolbar functions are often dangerous or dislocating, and offering too easy access to them can be like leaving a loaded pistol on the coffee table. The toolbar is for users who know what they are doing. The menu is for the rest.

Some user interface designers have gone ahead and added text to buttcons, either right on them or just below them, and left the images in place. This strikes me as a worst-of-both-worlds solution. After all, the space is far too valuable to waste this way. They are trying to satisfy two groups of users with two different goals: one wants to learn in a gentle, forgiving environment. The other knows where the sharp edges are but sometimes needs a brief reminder. Certainly, there must be a way to bridge the gap between these two classes of users. Later in this chapter, we'll discuss some methods that don't dedicate lots of precious video real estate to solving the problem.

Immediate behavior

Unlike menus, we don't depend on toolbar buttcons to teach us how they are used. Although we depend on buttcons primarily for speed and convenience, their behavior should not mislead us. Toolbar buttcons should become disabled if they are no longer applicable due to the current selection. They may or may not gray out—this is up to you—but if a buttcon becomes moot, it must not offer the pliant response: The buttcon must not depress.

I've seen programs that make moot buttcons disappear altogether, and the effect of this is ghastly. The supposedly Rock-of-Gibraltar-like toolbar becomes this skittish, tentative idiom that scares the daylights out of new users and disorients even those more experienced (me!). The path to modeless operation does not lie in becoming more ephemeral but rather in becoming more solid, permanent and dependable.

The toolbar freed the menu to teach

It was the toolbar's invention that finally allowed the pedagogical purpose of the menu to emerge. Once the frequently used functions were put into toolbar buttcons, the pulldown menus immediately ceased to be the primary function idiom. For users with even slight experience, if a buttcon existed, it was much faster and easier to use than pulling down a menu and selecting an item—a task requiring significantly more dexterity and time than merely pointing-and-clicking in one stationary spot. Before the advent of the toolbar, the pulldown menu was home to both pedagogy and daily-use functionality. Although the two purposes were intermixed, software designers didn't segregate them into different idioms until the toolbar demonstrated its potency. However, once the toolbar became widespread, the menu fell into the background as a supporting character.

The only programs where the menu is still used for daily-use functions are programs with poorly designed or non-existent toolbars.

ToolTips

The big problem with toolbar buttcons is that although they are fast and memorable, they are not decipherable. How is the new user supposed to learn what buttcons do?

Macintosh was the first to attempt a solution by inventing a facility called **balloon help**.

Balloon help is one of those frustrating things that everyone can clearly see is good, yet nobody actually uses, like no-fat cheese. Balloon help is a **flyover** facility (sometimes called **rollover**). This means that it appears as the mouse cursor passes over something without the user pressing a mouse button, similar to active visual hinting.

When balloon help is active, little speech bubbles like those in comic strips appear next to the object that the mouse points to. Inside the speech bubble is a brief sentence or two explaining that object's function.

Balloon help doesn't work for a couple of good reasons. Primarily, it is founded on the misconception that it is acceptable to discomfit daily users for the benefit of first-timers. The balloons are too big, too long, too obtrusive and too condescending. They are very much in the way. Most users find them so annoyingly in-your-face that they keep them turned off. Then, when they have forgotten what some object is, they have to go up to the menu, pull it down, turn balloon help on, point to the unknown object, read the balloon, go back to the menu, and turn balloon help off. Whew, what a pain!

Microsoft, on the other hand, is never one to make things easy for the beginner at the expense of the more frequent user. They have invented a variant of balloon help called ToolTips that is one of the cleverest and most-effective user interface idioms I've ever seen.

From a distance, ToolTips seem the same as balloon help, but on closer inspection you can see the minor physical differences that have a huge effect from the user's point of view. Unlike balloon help, ToolTips only explain the purpose of gizmos on the toolbar. They don't try to explain other stuff on the screen like scroll-bars, menus and status bars. Microsoft obviously understands that the user isn't a complete idiot and doesn't need to have the most basic stuff

explained to him. It also shows an understanding that, although we are all beginners once, we all evolve into more-experienced daily users.

ToolTips contain a single word or a very short phrase. They don't attempt to explain in prose how the object is used; they assume that you will get the rest from context. This is probably the single most-important advance that ToolTips have over balloon help, illustrating the difference in design intent of Microsoft versus Apple. Apple wanted their bubbles to *teach* things to first-time users. Microsoft figured that first-timers would just have to learn the hard way how things work, and ToolTips would merely act as a memory jogger for frequent users.

By making the gizmos on the toolbar so much more accessible for normal users, they have allowed the toolbar to evolve from simply supporting menus. ToolTips have freed the toolbar to take the lead as the main idiom for issuing commands to sovereign applications. This also allows the menu to quietly recede into the background as a command vector for beginners and for invoking occasionally used functions. The natural order of buttcons as the primary idiom, with menus as a backup, makes sovereign applications much easier to use. For transient programs, though, most users qualify as first-time or infrequent users, so the need for buttcons—shortcuts—is much less.

ToolTip windows are very small, and they have the presence of mind to not obscure important parts of the screen. As you can see in Figure 23-1, they appear underneath the buttcon they are explaining and label it without consuming the space needed for dedicated labels. There is a critical time delay, about a half a second between placing the cursor on a buttcon and having the ToolTip appear. This is just enough time to point to and select the function without getting the ToolTip. This means that in normal use, when you know full well what function you want and which buttcon to use to get it, you can request it without ever seeing a ToolTip window. It also means that if you forget what a rarely used buttcon is for, you only need to invest a half-second to find out.

That little picture of a printer may be ambiguous until I see the word "Print" next to it. There is now no confusion in my mind. If the buttcon were used to configure the printer, it would say "Configure Printer" or even just "Printer," referring to the peripheral rather than to its function. The context tells me the rest. The economy of pixels is superb.

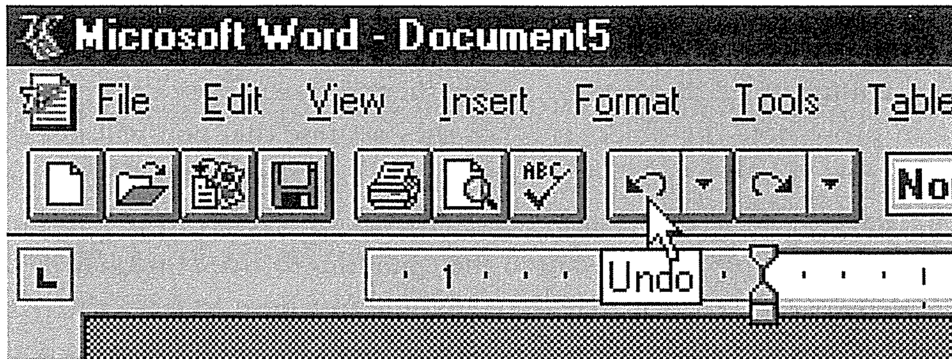


Figure 23-1

Microsoft's ToolTips were the solution to the toolbar problem. Although toolbars are for experienced users, sometimes these users forget the purpose of a less-frequently used command. The little text box that pops up as the cursor rests for a second is all that is needed to remind the user of the buttcon's function. The ToolTip succeeds because it respects the user by not being pedantic and by having a very strongly developed respect for the value of pixels. The idiom was the gate that allowed the toolbar to develop as the primary control mechanism in sovereign applications, while letting the menu fall quietly into the background as a purely pedagogic and occasional-use command vector.

I'm a very experienced user, and I leave ToolTips on all of the time. Balloon help on my Mac is never on except in rare cases in which I turn it on for just one balloon's worth of help. Microsoft's solution is a quantum leap beyond balloon help, and yet it is exactly the same. It just goes to prove that the devil is in the details.

Design tip: ToolTips are indispensable to toolbars.

ToolTips have completely spoiled me for anything else. I now get upset with any program that doesn't offer them. Toolbars without ToolTips force me to read the documentation or, worse, to learn their function by experimentation. And because toolbars contain immediate versions of commands that should be used by moderately experienced users, they inevitably contain some that are dislocating or dangerous. Explaining the purpose of buttcons with a line of text on the status line at the bottom of the screen just isn't as good as ToolTips that appear right there where I'm looking. That cheerful little yellow box with a terse word or two tells me all I need, where I need it, when I need it.

Do not create toolbars without ToolTips. In fact, ToolTips should be used on all pictographic buttcons, even those on dialog boxes.

Beyond the buttcon

Once people started to regard the toolbar as something more than just an accelerator for the menu, its growth potential became more apparent. Designers began to see that there was no reason other than habit to restrict the gizmos on toolbars to buttcons.

Opening the door to other popular gizmos was just the beginning. Soon designers began to invent new idioms expressly for the toolbar. With the advent of these new constructions, the toolbar truly came into its own as a primary control device, separate from—and in many cases superior to—pulldown menus.

After the buttcon, the next gizmo to find a home on the toolbar was the combobox, as in Word's style, font and fontsize controls. It is perfectly natural that these selectors be on the toolbar. They offer the same functionality as those on the pulldown menu, but they also offer a more object-oriented presentation by showing the current style, font and font size as a property of the current selection. The idiom delivers more information in return for less effort by the user.

Once comboboxes were admitted onto the toolbar, the precedent was set, the dam was broken, and all kinds of idioms appeared and were quite effective. The original buttcon was a **momentary buttcon**—one that stays pressed only while the mouse button is pressed. This is fine for invoking functions but poor for indicating a setting. In order to indicate the state of selected data, new varieties of buttcons had to evolve from the original.

The first variant was a **latching buttcon**—one that stays depressed after the mouse button is released.

For example, the four alignment buttcons shown in Figure 23-2 "latch" down to reflect the current status of the selected text. We'll talk more about these gizmos in Part VI.

Indicating state

This variety of gizmos contributed to a broadening in the use of the toolbar. When it first appeared, it was merely a place for fast access to frequently used *functions*. As it developed, gizmos on it began to reflect the *state* of the program's data. Instead of a buttcon that simply changed a word from plain to italic text, the buttcon now began to indicate—by its state—whether the

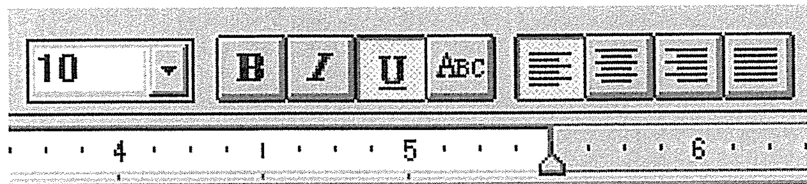


Figure 23-2

The development of the toolbar soon led to an extension of its purpose. It evolved from a mere repository of imperative command buttons to a place where gizmos could indicate the state of the currently selected item. This is a more object-oriented concept, and it makes our software more powerful. The toolbar has become the place for gizmo innovation, far beyond what we have come to expect from dialog boxes. This image shows the alignment buttcons from Microsoft Word. They are latching buttcons, staying down long after the user releases the mouse button. The buttcons indicate state in addition to allowing control of it. Good user interface idioms characteristically offer such richness.

currently selected text was already italicized. The buttcon not only controlled the application of the style, but it represented the status of the selection with respect to the style. This is a significant move towards a more object-oriented presentation of data, where the system tunes itself to the object that you have selected.

As the variety of gizmos on the toolbar grows, we find ourselves in the ironic position of adding pop-ups to it. The Word toolbar shown in Figure 23-3 shows the UNDO pop-up. It is ironic that such a very menu-like idiom should migrate onto the toolbar. Immediate UNDO certainly belongs on the toolbar, but does the associated pop-up that shows the history of past actions belong there, too? There isn't a clear answer. It is good that the historical list is positioned next to the UNDO buttcon because it makes it easy to find, but the toolbar is the place for frequently used functions. How frequently would one need to access the list? Ultimately, it comes down to pixels. As Microsoft implemented it, the pixel consumption is small enough to justify the idiom. What should be evident, though, is that the modern toolbar is increasingly pushing the old-fashioned menu bar into the background as a secondary command vector.

Toolbar morphing

Microsoft has done more to develop the toolbar as a user interface idiom than any other software publisher. This is reflected in the quality of their products.

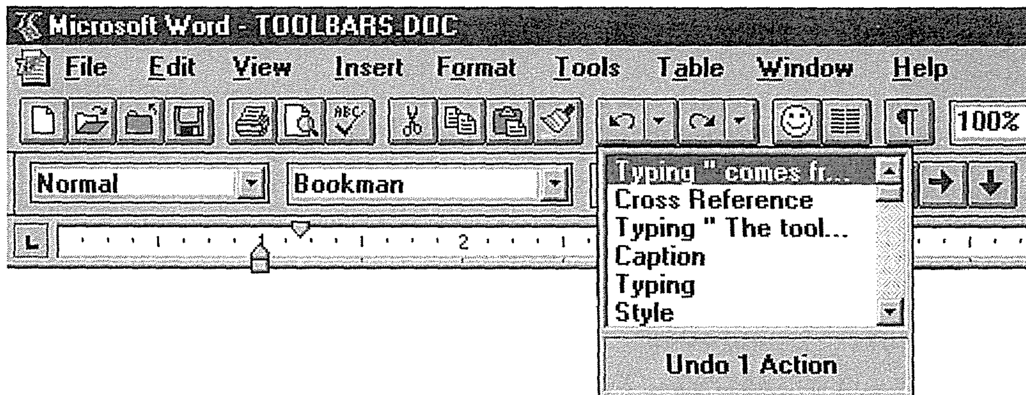


Figure 23-3

A typical Microsoft toolbar showing the half-button, half-icon gizmo I call a buttcon. Notice the evolution of the buttcon in the undo and redo functions on the right. They now have drop-down lists associated with them. Irony of ironies, the pulldown menu is migrating onto the toolbar, which began as a refutation of the old-fashioned pulldown!

In their Office suite, all of the toolbars are very customizable. Each program has a standard battery of toolbars that the user can choose to be visible or invisible. If they are visible, they can be dynamically positioned in one of five locations. They can be attached—referred to as "docked"—to any of the four sides of the program's main window. You click the mouse anywhere in the interstices between buttcons on the toolbar and drag it to any point near an edge and release. The toolbar attaches itself permanently to that side, top or bottom. If you drag the toolbar away from the edge, it configures itself as a floating toolbar, complete with mini-caption bar. Very clever, but not as clever as the customizability of the individual toolbars.

Customizing toolbars

Microsoft has clearly seen the dilemma that toolbars represent the frequently used functions for all users, but that those functions are different for each user. This conundrum is solved by shipping the program with their best guess of what an average person's daily-use gizmos will be and letting others customize things. This solution has been diluted somewhat, however, by the addition of non-daily-use functions. Clearly, amateurs (probably from marketing) got their hands on the Word toolbar. Its default buttcon suite contains functions that certainly are not frequently used. Things like "Insert Autotext" or "Insert Excel Spreadsheet" sound to me more like marketing features than practical

daily options for a majority of users. While they may be useful at times, they are not used *frequently* by throngs of users.

The program gives the more advanced user the ability to customize and configure the toolbars to his heart's content. There is a certain danger in providing this level of customizability to the toolbars, as it is quite possible for a reckless user to create a really unrecognizable and unusable toolbar.

Mitigating this is that it takes some effort to totally wreck things. People generally won't invest "some effort" into creating something that is ugly and hard to use. More likely, they will do what I have done: make just a few custom changes and enter them one at a time over the course of months or years. The toolbars on my personalized copy of Word look just about the same as the toolbars on anyone else's, except for a couple of exceptions. I've added a smiley face buttcon that inserts the date in my favorite format. I've added a buttcon from the format library that specifies SMALL CAPS, a format I seem to use a lot more than most people. If you were to use my word processor, you might be thrown by the smiley face and the small caps, but the overall aspect would remain familiar and workable.

Of course, Microsoft has extended the idiom so that you can create your own completely new, completely custom toolbars. The feature is certainly overkill for normal users, but corporate MIS managers might like it a lot for creating that "corporate" look (see Chapter 32 for a discussion of such configuration needs).

My favorite part of the Microsoft toolbar facility is the attention to detail. You have the ability to drag buttcons sideways a fraction of an inch to create a small gap between them. This allows you to create "groups" of buttcons with nice visual separations. Some buttcons are mutually exclusive, so grouping them is very appropriate. You can also select whether the buttcons are large or small in size. This is a nice compensation for the disparity between common screen resolutions, ranging from 640×480 to 1280×1024. Fixed-size buttcons can be either unreadably small or obnoxiously large if their size is not adjustable. You have the option to force buttcons to be rendered in monochrome instead of color, though I don't really understand why you would want to. Finally, you can turn ToolTips off, though, again, I can't imagine why anyone would do this, and I don't know anyone who does.

One of the criticisms I have of the Microsoft toolbar facility is its scattered presence on the menus. There is a "Toolbars..." item on the "View" menu that

brings up a small dialog box for selecting which toolbars are visible, creating new toolbars, turning ToolTips on or off, turning color on or off, and selecting large or small buttcons. However, if you want to change the selection of buttcons on a toolbar, you have to go to the "Customize..." item on the "Tools" menu, which brings up a dialog box that allows you to configure the toolbars, the keyboard and the menu (yes, there is a button on the Toolbars dialog that takes you directly to the Customize dialog, but that is a hack compared to a simple, unified view). I have the distinct impression that this design is the result of either accident or user testing rather than from the judgment of a skilled designer. Splitting the toolbar stuff into these two separate dialogs seems irrational to me. It is not at all clear how to find the various toolbar settings, nor do I think that the neophyte is effectively protected from self-defeating actions this way.

I'd much prefer a single dialog box, like the one in Figure 23-4, that enables the user to configure all aspects of the toolbars. A two-paned dialog would do the job well—one pane for the basic selections and a second pane for advanced stuff. This would be a classic rendition of increasing depth on additional panes.

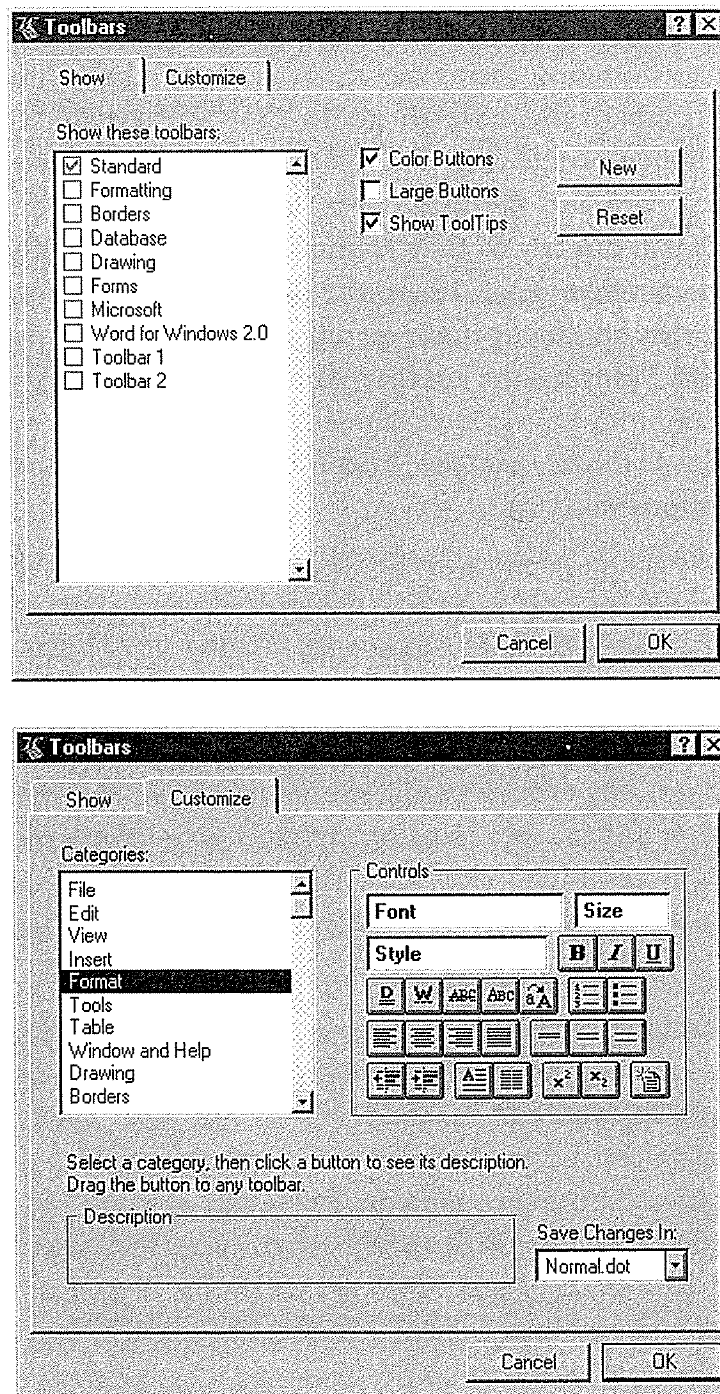


Figure 23-4

Here is a proposed solution to Microsoft's scattered toolbar dialogs. Instead of two dialogs available from two different menus, all toolbar operations are combined on a single dialog box with two panes. The "Show" pane contains the basics that most users may occasionally need to use. The "Customize" pane contains the more-sophisticated stuff that the experienced users might want.

Roll the Credits, Please

The modern desktop computer is getting quite crowded. A typical user has a half-dozen programs running concurrently, and each program must assert its identity. The user needs to recognize your application when he has relevant work to be done, and you should get the credit you deserve for the program you have created. There are several conventions for asserting your identity in software.

Your program's name

The most fundamental element of any program's identity is its name. By convention, this name is spelled out in the caption bar of the program's main window. I call this text value the program's **title string** because it is a single text value within the program that is usually owned by the main window.

The title string is used in several other places, too. In Windows 3.x, it is displayed beneath the program's icon. In

Windows 3.x, it is acceptable to have a very long title string because most programs have main windows that are at least eight to ten centimeters wide. Their correspondingly wide caption bars can easily render long program names.

Windows 95, though, introduces some complications. In Windows' latest version, the title string plays a greater role in the operating system's shell interface. Particularly, the title string is displayed on the program's **launch button** on the Startbar.

The launch buttons on the Startbar automatically reduce their size as more buttons are added, which happens each time the user launches more programs. As the buttons get shorter, their title strings are truncated to fit. If you add your company's name to your program's name, like, say "Microsoft Word," you will find that it only takes seven or eight running programs or open folders to truncate your program's launch-button string to "Microsoft...." If you are also running "Microsoft Excel," you will find two adjacent buttons with identical, useless title strings. The discriminating portion of their names—"Word" and "Excel"—are hidden. Yes, the launch buttons also contain your program's icon in the always-visible left end, but it sure would be nice if the programs were named "Word by Microsoft" and "Excel by Microsoft" instead.

You can rightfully say that this is a design fault of the Startbar and not your problem. You'd be right, of course, but this doesn't help the poor user. In the past, your program's title string was the place to show off your company or brand. It is now becoming a more-functional part of the interface.

The title string has, over the years, acquired another purpose. Many programs use it to display the name of the currently active document. Microsoft's Office suite of programs does this. They append the name of the active document to the right end of the title string using a hyphen to separate it from the program name. The File Manager shows the current pathname instead of a document name. The technique isn't a standard, but because Microsoft does it, it is often copied. It makes the title string extremely long—far too long to fit onto a launch button.

What Microsoft should have done with Windows 95 is add a new title string to the program's internal data structure. This string would be used only on the launch button, leaving the original title string for the window's caption bar. This would enable the programmer to tailor the launch-button string for its restricted space, while letting the title string languish full-length on the always-roomier caption bar.

Your program's icon

The second biggest component of your program's identity is its icon. A standard program icon is 32 pixels square. In Windows 3.x, the icon is usually shown—by convention—in the program's "About..." box. It is also displayed in the Program Manager and on the desktop when the program is minimized. The icon didn't do a lot in Windows 3.x.

In Windows 95, each program's icon is used much more widely than it is in Windows 3.x. First, there are now two icons: the standard one at 32 pixels square and a new, miniature one that is 16 pixels square.

The regular size is used on the desktop, but the miniature one is used on the caption bar, the Startbar, the Explorer and other locations in the Windows 95 interface. Because of this increased importance, you must pay greater attention to the quality of your program icon. In particular, you want your program's icon to be readily identifiable from a distance—especially the miniature version. The user doesn't necessarily have to be able to recognize it outright—although that would be nice—but he should be able to readily see that it is different from other icons. Bold color is especially helpful in accomplishing this, unlike Microsoft's white and pale-blue icons that are very hard to distinguish.

Dependencies

A nineteenth-century farm was a complex facility supporting many functions and, often, multiple families. The architecture reflected this, and the main house was always supported by a gathering of small outbuildings such as the stable, servants' quarters, the hen house, the stone house, the privy, the kitchen and the blacksmith's shop. These supporting structures were called "dependencies." Because modern software frequently consists of a single main screen supported by a family of smaller windows, I have adopted the old term. I call the many smaller, supporting windows **dependencies**.

I include in this category those windows that are not strictly needed to perform the main function of the program, particularly splash screens, About boxes and Easter eggs.

Dependencies are either available only on request or are offered up by the program only once. Those that are offered unilaterally by the program are erected when the program is used for the *very* first time or each time the program is initiated.

About boxes

The **About box** is a single dialog box that—by convention—identifies the program to the user.

The About box is also used as the program's credit screen, identifying the people who created it. Ironically, the About box rarely tells the user much "about" the program.

On the Macintosh, the About box can be summoned from the top of the "Apple" popup menu. In early versions of Windows, it could be called from the bottom of the "File" menu, but in modern Windows (3.x and 95), it is almost always found at the bottom of the "Help" menu.

Microsoft has been consistent with About boxes in their programs, and they have taken a simple approach to its design, as you can see in Figure 24-1. Microsoft sets the pace by using the About box almost exclusively as a place for identification; a sort of driver's license of software. I find this unfortunate, as it is a good place to give the curious user an overview of the program in a way that doesn't intrude on those users who don't need it. Programmers everywhere are following Microsoft's lead and making identity-only About boxes. It is often, but not always, a good thing to follow in Microsoft's design footsteps. This is one place where diverging from Microsoft can be a big advantage for you.

The main problem with Microsoft's approach is that the About box doesn't "tell me about" the program. In reality it is an "identification box." It identifies the program by name and version number. It identifies various copyrights in the program. It identifies the user and the user's company. These are certainly useful functions, but are more useful for Microsoft than for the user.

Because this facility just presents the program's fine print instead of telling the user *about* the program, I think it should be called an **identity box** instead of an About box.

The identity box identifies the program to the user, and the dialog in Figure 24-1 fulfills this definition admirably. It tells us all the stuff the lawyers require and the tech support people need to know. Clearly, Microsoft has made the decision that an identity box is important while an About box is expendable. Personally, I'd like to see the two combined into one, really helpful, dialog.

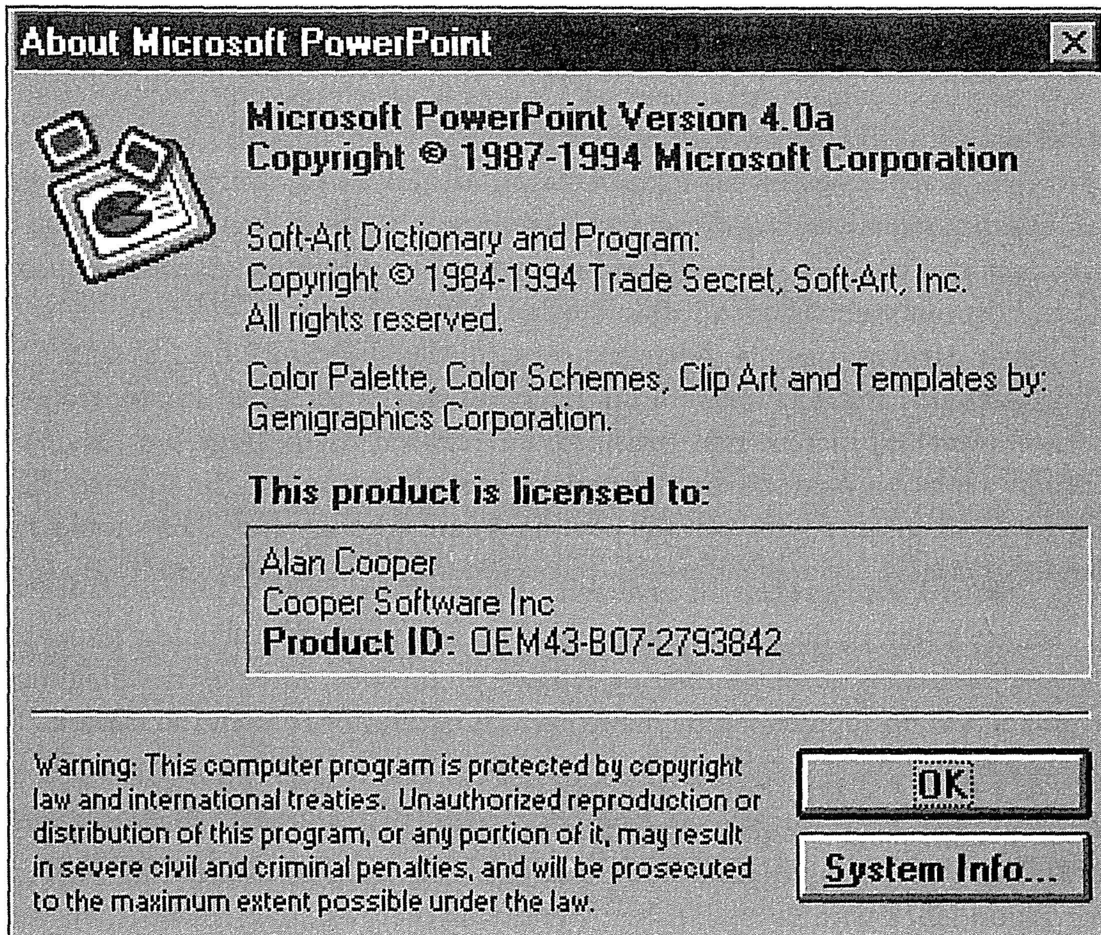


Figure 24-1

This About box from PowerPoint is a typical example of Microsoft's approach. It tells you the exact name and version of the program, states relevant copyrights, issues legal warnings (ugh!) and displays the user's name and company. The program's icon is traditionally shown in the upper left corner. The problem is, if I asked you to "tell me about PowerPoint," you probably would not bother to recite the relevant copyrights to me but would instead say something to me regarding what the program is about. What is wrong with this picture?

As we've seen, the About box must offer the basics of identification, including the publisher's name, the program's icons, the program's version number and the names of its authors. Another item that could profitably belong here is the publisher's technical support telephone number.

Many software publishers don't identify their programs with sufficient discrimination to tie them to a specific software build. Some vendors even go so far as to issue the same version number to significantly different programs for

marketing reasons. Get a clue! The version number in the identity—or about—box is mainly used for technical support. A misleading version number will cost the publisher a significant amount of phone-support time just figuring out precisely which version of the program the user has. It doesn't matter what scheme you use, as long as this number is very specific.

An important part of reporting the version number is telling the user which previous version it replaces. Knowing that this is Version 3.2 isn't tremendously meaningful. Knowing that Version 3.2 fixes bugs in Version 3.1 and supersedes all Versions 2.x, however, *is* useful. Vendors work hard to improve their software, and each version is usually intended to replace some previous version. Smaller, incremental revisions are released to fix bugs but may not entirely replace a predecessor. Similarly, a special version may be shipped that allows compatibility with certain new hardware or software. This should be stated, as well.

If you are going to display an informative version number, it wouldn't hurt to explain the details of the numbering scheme on this box. Most users will ignore it, but it will be appreciated by many curious users, not to mention thousands of professional users and installers.

Many programs are uniquely identified by their serial number. This, of course, is the place to display that number. The user may need to use that number in correspondence with the publisher, or for his own company records, so the program should let the user view it and select it for copying.

The About box is absolutely the right place to state the authors' names. Most modern programs are built by teams of technical experts that can range from three or four to thirty or forty individuals. As a former software author, I'm very bullish on the idea of giving credit where credit is due in the design and development of software. Programmers, designers, managers and testers all like to see their names in lights. Like the credits in a movie, the people who labored over the product deserve their day in the sun. The documentation writers often get to put their names in the manual, but the others only have the program itself. The About box is one of the few dialogs that has no functional overlap with the main program, so there is no reason why it can't be oversized. Take the space to mention everyone who contributed. Although some programmers are indifferent to seeing their names on the screen, many programmers are powerfully motivated by it and really appreciate managers who make it happen.

What possible reason could there be for *not* naming the smart, hard-working people who built the program?

That last question was directed at Bill Gates, who has a corporate-wide policy that individual programmers *never* get to put their names in the About boxes of programs. Having sold him some software, I can tell you that it is his personal belief that no programmers should ever be identified, although I was able to convince him to bend sufficiently to include the name of my company in the Visual Basic About box. He feels that it would be difficult to know where to draw the line with individuals. I understand his plight, but as I watch the credits for modern movies scroll for ten minutes or so, I'm not terribly sympathetic.

Microsoft's policy in this area bothers me because their conventions are so widely copied across the industry. As a result, their no-programmer-names policy is also widely copied by companies who have no real reason for it other than wanting to be like Microsoft.

System information

Some vendors put system information on the About box, particularly information about the amount of memory used and available in the system. Why do they do this? I know that some Microsoft programs did it, so maybe vendors are just copying Microsoft. (An undocumented function in Windows 3.x enables this for all of the utilities in that version.) If you put a memory-meter in the About box, the only effect it can have is to bother new users by implying that they need to know about memory consumption. What possible connection could be made between asking "about" the program and learning how much memory is left in the system? It might make sense if the dialog told how much memory were used by the particular program, especially if it were expressed in terms of the percentage of memory available. At least then the user could see that the program in question is using up, say, 20% of the memory available to applications.

The About box in Figure 24-1 has a push-button that launches the Microsoft system utility that reports on the capabilities of the entire system. This is a nice feature, but it would make more sense as part of the online help facility rather than as part of the About box facility.

The desire to make About boxes more useful is clearly a strong one. Otherwise, we wouldn't see memory usage and system-information buttons on them. This

is admirable, but, by taking a more goal-directed approach, we can add information to the About box that can really help the user. The single most important thing that the About box can convey to the user is the scope of the program. It should tell, in the broadest terms, what the program can and can't do. It should also state succinctly what the program's purpose is. Most program authors forget that many users don't know what the InfoMeister 2000 Version 3.0 program actually does. This is the place to gently clue them in.

The About box is also a great place to give the one lesson that might launch a new user successfully. For example, if there is one new idiom—like a direct-manipulation method—that is critical to the user interaction, this is a good place to briefly tell him about it. Additionally, the About box can direct the new user to other sources of information that will help him get his bearings in the program. Pointing him to online help or to other informational facilities built in to the interface can give a nascent user a real boost.

Splash screens

A **splash screen** is an identity dialog box displayed when a program first loads into memory.

Sometimes, it is just the About box that is displayed unilaterally, but often a publisher creates a separate splash screen that is more engaging and visually exciting than a boring, old About box. Not every program will have a splash screen, whereas almost every Windows application will have an About box.

The splash screen must be placed on the screen immediately when the user begins loading the program so that he can read it while the bulk of the program loads and prepares itself for running. It isn't really fair for the program to finish loading and prepping before it erects the splash screen. The user is then penalized for loading it, and will sense this and be irked by it. The program must show the utmost respect for the user's time, even if it is measured in milliseconds.

Shareware splash screens

If your program is shareware, the splash screen can be your most important dialog. It is the mechanism whereby you inform the user of the terms for using the product and the appropriate way for him to pay for the product. I've heard the shareware splash screen referred to as the **guilt screen**. Of course, this information will also be embedded in the program where the user can request it, but

by forcing it in the user's face every time the program loads, you can reinforce the concept that the program *should* be paid for. Some shareware splash screens go so far as to include the text of a license agreement and buttons labeled "I Agree" and "I Don't Agree." The program only runs if the user presses the "Agree" button.

The splash screen should appear as soon as possible after the program is invoked. As soon as a decent interval has passed, it should disappear and the program should go about its business. If, during the splash screen's tenure, the user presses any key or mouse button, it also should disappear. If the program is still loading or deploying, it will then have to use some other idiom to indicate this to the user.

The splash screen is an excellent opportunity to create a good impression in the user's mind about your program. It can be used to reinforce the idea in the user's mind that he made a good choice by purchasing your product. It also helps to establish a visual brand by displaying the company logo, the product logo, the product icon and other appropriate visual symbols.

Help the first-time user

Splash screens are also excellent tools for directing first-time users to training resources that are not used in the normal course of daily usage. If the program has built-in tutorials or wizards, the splash screen can provide push-buttons that move the user directly to these facilities.

Microsoft has begun including a variant of the splash screen in their Office suite of programs. These little dialogs offer up a handy "Tip of the Day," shown in Figure 24-2, that helps the beginner become a more sophisticated user.

Because splash screens are going to be seen by every user, it means that even first-timers will see it, so if you have something to say to them, this is a good place to do it. On the other hand, the message you offer to those first-timers will probably get pretty boring for more-experienced users who must see it over and over. So whatever you say, say it clearly and tersely and without ornamentation or cuteness. An irritating message on the splash screen is like a pebble in your shoe, rapidly creating a sore spot if it isn't removed promptly. Although the Tip of the Day dialog is well-liked by many beginners, experienced computerphiles tend to hate them. The "Show Tips at Startup" checkbox gizmo allows the latter group to make the annoying idiom go away (but it might be a good idea to mention how to get it back next to the checkbox).

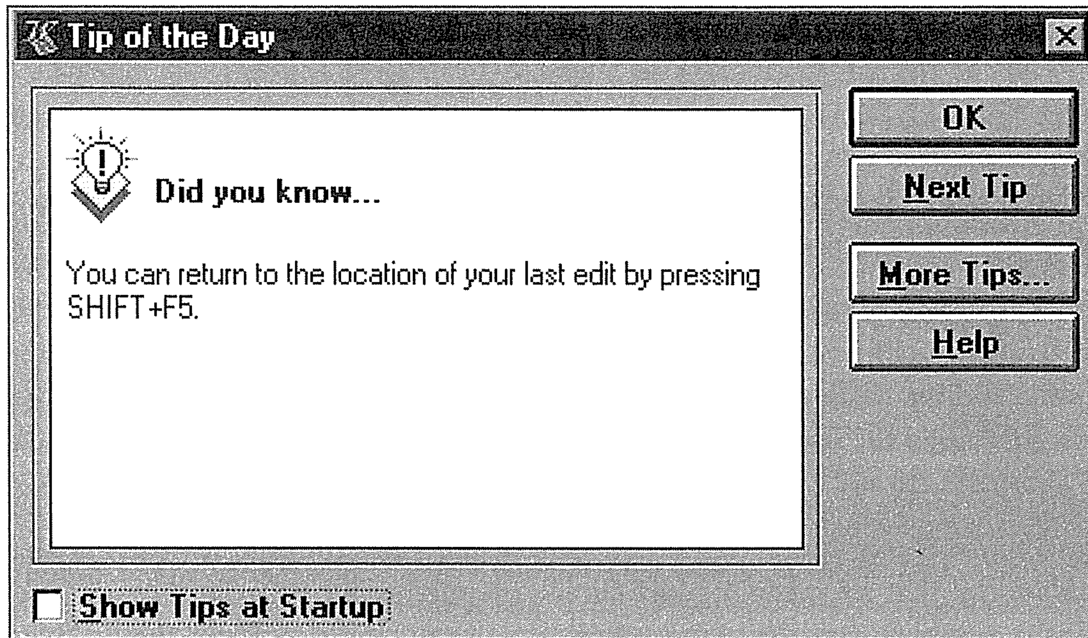


Figure 24-2

The Tip of the Day dialog from Microsoft Word is a variant of the splash screen. It appears once every time the program is invoked and offers the user a handy hint on how to use the program. Its intention is to make the user a better, more sophisticated user of the program. Don't overlook the potential of such artifacts; users like this kind of doting.

Easter eggs

If you call up the PowerPoint About box shown in Figure 24-1 and then click on it with the mouse *outside* either of the push-buttons, the box quickly flashes to black, and the names of all of the programmers who built it scroll smoothly upwards like the credits after a motion picture. Remember, this is a Microsoft program, and Bill doesn't allow the programmers to put their names in the program. But the philosophy in Redmond is "What Bill doesn't know, won't hurt him," and programmers have gotten their names into just about every program Microsoft ships. They do it by sneaking them into the interface with clever—but hidden—command idioms like this one.

Hidden surprises in a program are called **easter eggs**, and they are wonderful, engaging bonuses that contribute to the likableness of their host products.

Easter eggs don't have to be useful; in fact, practicality detracts from their appeal. The wonder of easter eggs is that they show off the astonishing power of the computer by doing something for pure entertainment. Like their name implies, the user gets the thrill of stumbling upon a secret: a flower in the forest; a pearl in the oyster; a prize in the Cracker Jack box; a winner in the Lotto. Users will cherish the knowledge and share it eagerly with their colleagues the way a good joke makes the rounds at the office. Whatever they say, they are talking about *your* program, and that is good for sales.

Easter eggs should be visually attractive. They are worth some time and effort in animating them or adding eye-catching artwork. I know one product whose About box, when you press the appropriately obscure key combination, changes to display a photograph of the company president in his cups at the office Christmas party. Nobody except employees know about this easter egg (maybe it should be called a Christmas egg?), but it has become part of the company lore. Good customers are shown this treat when they buy lots of copies, and then they feel like part of the inner circle. Just like any interface idiom, easter eggs are very memorable. It's harmless fun, and harmless fun is something that binds people tightly together.

In Windows 3.1, go into the Program Manager and hold the CTRL and SHIFT keys down and don't let go. Now pull down the Help menu and select About. When the about box appears, double-click on the Windows flag icon. Close the box with the OK button. Repeat, then repeat once again. On the second repetition, you see a waving flag. On the third repetition, you get a caricature of Bill Gates, Steve Ballmer, Brad Silverberg or a bear hosting a scrolling list of credits, naming every individual person who worked on the product. Stuff like this unites the team, creates legends and builds the corporate culture.

The About box is a common location for easter eggs, and whenever I get a new program, I try clicking in unlikely spots on the About box looking for them.

Easter eggs can be big or small, active or static. There are even a few of them in this book. Check out the fine print in Figure 7-1, for example.

Probably the most frequent use for easter eggs is the hidden credit box like the one described above, but they have other manifestations, too. The Tip of the Day issues random, helpful tips about the host program. In Word, though, you occasionally stumble on a real non sequitur of a tip as shown in Figure 24-3.

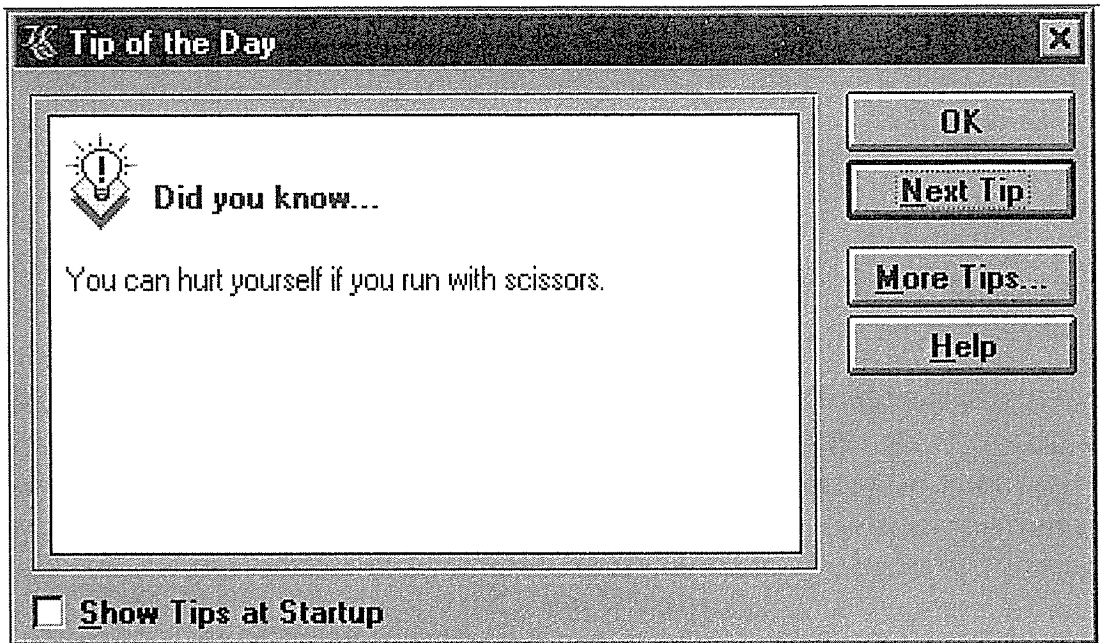


Figure 24-3

The Tip of the Day dialog from Microsoft Word also contains easter eggs. This one is pretty tame, but it is still worth calling over your buddy at the office and showing it to her. This curious message can't be argued with, but its placement is incongruous. Users really like easter eggs, and I think every program should have a few. The fuddy-duddies in your company probably disagree, as they do in most companies, which is why easter eggs are usually so well hidden.

Part VI: The Gizmos
Canned Visual Design

Gizmos are concentrations of interface design as much as they are modules of code. They offer the great advantage of shortening the development cycle while simultaneously presenting familiar affordances to the user. They also offer the great disadvantage of locking designers into old ways of thinking, instead of leaving them free to create methods of interaction that are more appropriate for the situation. They also give designers a false sense of security: using gizmos doesn't automatically make interfaces good. We need to look critically at these creations, understanding that they are as much a result of accident as they are of conscious design.

Imperative and Selection Gizmos

Gizmos are directly manipulable, self-contained, visual screen idioms. Gizmos, controls, widgets or gadgets—whatever you choose to call them, they are a primary building block for creating graphical user interfaces. They are closely identified with GUI programming, being as fundamental to GUI construction as windows, menus and dialog boxes.

This use of the word **gizmo** was coined by Mike Geary, a Windows programmer of legendary skill who worked with me to create the visual programming front-end for Visual Basic.

Microsoft calls gizmos "controls," but I avoid this term because it has so many other meanings in the same context: "How much control do I have over the code that controls that control that is busy controlling the modem controller?"

Gizmo-laden dialog boxes

Dialog boxes are the easiest things to build in Windows. The dialog box facility offers automatic tools for specifying how and where gizmos will be placed. The de facto definition of a dialog box is a modal window covered with gizmos. The ease with which programmers can create user interfaces based on one gizmo-laden dialog box after another is significant. Equally significant is the difficulty involved in creating Windows interfaces using any other visual, directly manipulable idioms. Essentially, Windows divides the universe of interaction into two worlds: the extremely easy to implement world of canned gizmos and the extremely difficult to implement world of direct visual interaction. Consistent with this, most existing literature—from Microsoft and elsewhere—covers the canned-gizmo world reasonably well, while utterly ignoring any other approach. Just for the record, let me say that canned-gizmo-laden dialog boxes are *not* the key to successful user interface design.

A multitude of gizmo-laden dialog boxes doth not a good user interface make

I'm not saying that we should toss out gizmos. I just want to make clear that while their inclusion may guarantee ease of implementation, it doesn't guarantee ease of use.

Most of the gizmos we are familiar with are those that come standard with the Windows system. This set of canned gizmos has always been very limited in scope and power.

Gizmo liberation

Using gizmos has always been pretty easy, but writing them is a lot tougher. Any journeyman C programmer can create a new gizmo, although it takes a considerable amount of attention to detail to code it fully. To comply with the de facto standards of gizmohood, it must offer many niceties, such as recognizing all keyboard commands, showing a dynamic gray rectangle to indicate it has the focus and having the ability to gray out when inactive. The difference

in effort between writing and merely employing gizmos, though, has historically kept most programmers out of the business of creating gizmos.

Microsoft Windows has always come with a standard suite of gizmos, and these form the basis of most interaction with applications. These standard gizmos are an integral part of the operating system. Technically, they are in USER.EXE, the portion of Windows that defines the GUI, or most of what we see on the screen. Although the programming interfaces to these gizmos bear a vague family resemblance, each one is quite different. There has never been a common application programming interface (API) for them, and Microsoft has failed to define an acceptable standard for third-party vendors. This missing interface has really restricted the development of new gizmos.

Brave Windows developers have always invented new gizmos, but almost all of them remained proprietary. The lack of either a market or standards simply made it too difficult to distribute them, although a few aggressive companies developed one or two trademark gizmos to use in their own products. In the past, several companies tried to sell their gizmos but never seemed to make much headway. The only gizmos that had any universality were those that came as part of Windows itself.

The advent of Visual Basic in 1992 changed all that. VB has an interface that allows third-party gizmos to be installed dynamically. The gizmos are coded into dynamic link libraries (DLLs) with a commonly defined interface called VBX. The VBX interface isn't particularly pretty or powerful (I should know, I invented it), but it *is* standard, and it defines a method whereby a program can use a gizmo that isn't an integral part of Windows. Putting it another way, a programmer now has an easy way to use gizmos not made by Microsoft. Since the VBX interface gained popularity along with VB, there has been a veritable explosion in the gizmo market.

The VBX interface is giving way to OCX, and thence to OLE Custom controls—superior APIs all—but it was VBX that broke the dam and showed the way to decoupling gizmos from the operating system.

As the software market shook out in the last few years, many second- and third-tier software publishers saw the VBX phenomenon as an escape route from a one-way trip to Chapter 11. They took their powerful-but-money-losing spreadsheets, graphing programs and word processors and made them VBX-compliant. These products found a new lease on life in the thriving third-party gizmo marketplace, and programmers can now, for nominal fees, include in

their software gizmos that provide functionality comparable to more mainstream products. This market shift has greatly expanded our old notions about gizmos and their capabilities.

The gizmos that Mother gives you

In USER.EXE of Windows 3.x, there were only six classes of gizmos: buttons, edit fields, static fields, listboxes, scrollbars and comboboxes. All of the other familiar, traditional gizmos like labels, group boxes, radio buttons, checkboxes, frames, rectangles and icons were derived from one or the other of these classes.

With Windows 95, the set of available gizmos has grown considerably, but even better, Microsoft is delivering these new gizmos to the developer by way of DLLs rather than as part of USER.EXE. This decoupling of user code from operating system code is a very progressive step that will further encourage the development of third-party gizmos.

In spite of this, a huge amount of software already exists that was built with the notion that a gizmo can only be one of those six original ones that came with Windows 3.x. Much of what passes for interface design is really an artifact of the limited palette of gizmos available to the Windows programmers in their formative years. We are only slowly breaking out of that trap.

Although gizmos can be categorized by many factors, when you examine them in light of the user's goals, we find that they come in four basic flavors. They can be used to initiate a function, which I call an *imperative gizmo.* They can be used to select some option or data, which I call a *selection gizmo.* They can be used to enter some data, which I call an *entry gizmo.* And they can be used to directly manipulate the program visually, which I call a *display gizmo.* Of course, some gizmos combine one or more of these flavors. We'll now look at each type in more detail.

Imperative gizmos

In the interaction between humans and computers, there is a language of nouns (sometimes called objects), verbs, adjectives and adverbs. When we issue a command, we are specifying the verb—the action of the statement. When we describe what the action will affect, we are specifying the noun of the sentence. Sometimes we choose a noun from an existing list and sometimes we enter a new one. We can modify both the noun and the verb with adjectives and adverbs, respectively.

I call the gizmo type that corresponds to a verb the **imperative gizmo** because it commands immediate action. Imperative gizmos take action, and they take it immediately. Menu items, which I discussed in Part V, are also immediate idioms. In the world of gizmos, the quintessential imperative idiom is the push-button; in fact, it is the only one, although it comes in numerous guises. Press the button and the associated action—the verb—executes immediately.

Push-buttons used to be identified by their unique outline, but since Windows went 3D (with Windows 3.0), buttons are now identified by their raised aspect. If the gizmo is rectangular and appears raised (due to its shadow on the right and bottom and highlight on the top and left), then it has the visual affordance of an imperative. It will execute as soon as the user presses and releases it with the mouse cursor.

The push-button is arguably the most visually compelling gizmo in the designers bag 'o gizmos. It isn't surprising that it has evolved with such diversity across the user interface. The manipulation affordances of contemporary faux-three-dimensional push-buttons have prompted their widespread use. It's a good thing—so why *not* use it a lot?

Part of the affordance of a push-button is its "pressability," which indicates its pliancy. When the user points to it and presses the mouse button, the push-button on screen visually changes from raised to sunken, indicating that it is pressed. This is an example of dynamic visual hinting, like I discussed in Part IV. I've seen programs where buttons are painted on the screen but don't actually move when pressed. This is cheap and easy for the programmer to do, but it is very disconcerting for the user, because it generates a mental question: "Did that actually do something?" The user expects to see the button move—the pliant response—and you must satisfy his expectations.

This is increasingly important in multimedia applications, many of which draw beautiful pictures on the screen and set aside portions of them as **hotspots** that are sensitive to clicking.

Cursor hinting isn't enough, though it's a desirable supplement. Even if the entire screen is consumed by a collage of, say, baseball collectibles, when the user clicks on a Louisville Slugger, the bat should move to visually confirm to the user that it is an imperative push-button—or, in this case, a push-bat!

The development of the push-button

In the early versions of Windows, push-buttons were stodgy beasts, largely used for terminating dialog boxes, which did the bulk of the heavy work of interaction.

On modal dialog boxes, the push-button is usually used only for terminating commands. This means that it is really an excise control; managing the window rather than directly affecting the user's information.

Concurrent with the release of Windows 3.0 came a surge of activity in user interface innovation across the industry. The toolbar was one of the great advances of that period, and it has quickly grown into a de facto standard as familiar as the menu bar. To populate the toolbar, the push-button was adapted from its traditional home on the dialog box. On its way, it changed significantly in function, role and visual aspect.

On dialog boxes, the push-button was rectangular and exclusively labeled with text, but when it moved to the toolbar, it became square, lost its text and acquired a pictograph, an iconic legend. Thus was born the buttcon: half button, half icon.

The invention of the toolbar qualitatively changed the role of the push-button. Actually, its role expanded rather than changed, as it is still a fixture of dialog box management.

Buttcons

Buttcons are easy: they are always visible and don't demand as much time or dexterity as a pulldown menu does. Because they are constantly visible, they are easy to memorize, particularly in sovereign applications. The advantages of the buttcon are hard to separate from the advantages of the toolbar—the two are inextricably linked. The consistently annoying problem with the buttcon derives not from its button part, but from its icon part. We instantly decipher the visual affordance—it screams "press me." The problem is that the image on the face of the buttcon never gets that clear.

Icons in general are hard to decipher with certainty, and icons of verbs are much harder to decipher than icons of nouns. Because buttcons are imperative, they are verbs, and thus remain problematic. It isn't really a matter of coming up with better visual metaphors or finding a better graphic artist. The problem

is that visual symbols that convey actions and relationships are difficult, if not impossible, to find. If you do find an appropriate image, it may have good mnemonic qualities but will usually be inadequate to teach newcomers its purpose.

The dilemma arises because images do have such good mnemonic qualities. These qualities are good enough that the visual image is more than enough to remind the daily user of the command represented by the buttcon. The image buttcon is very space-efficient compared to the older, text-legend button. As long as there is a way to learn it initially, and it is part of the user's working set of commands, he will remember the image idiomatically and have no problem with the lack of innate learnability. The distinguishing quality of the buttcon's image is that it is visually distinct and memorable.

Without a mechanism for explaining their purpose, however, buttcons and toolbars are badly afflicted and significantly less useful than they could be. The rapid spread of buttcons on toolbars caused a widespread grumbling about incomprehensible icons. In response, some companies pumped up their buttcons until they were big enough to hold text legends in addition to icons. Yet others made it the user's choice, adding another annoying layer of excise to the interface. Then, as I discussed in Chapter 23, Microsoft's ToolTips neatly solved the inscrutable buttcon problem once and for all. ToolTips provide initial learning without intruding on the view of the frequent user. They have spoiled me. I now find myself getting impatient and frustrated with programs whose buttcons lack them. The groundswell of grumbling over inscrutable buttcons has subsided to inaudibility due to ToolTips. I am in constant wonder over the efficacy of the idiom. They seem so clunky, so much like a quick-fix solution, and yet they solve the problem very adroitly.

Selection gizmos

Since the imperative gizmo is a verb, it needs a noun upon which to operate. Selection and entry gizmos are the two types used to select nouns. A **selection gizmo** allows the user to choose an operand from a group of valid choices.

No action is associated with selection gizmos. Selection gizmos can either present a single choice (to which the user can only say "yes" or "no"), or it can present a group of choices (from which the user can select one or more choices, depending on how the gizmo is configured). The listbox and the checkbox are good examples of selection gizmos.

Checkbox

The **checkbox** was one of the earliest visual gizmo idioms invented, and it remains the favorite for presenting a single, binary choice. The checkbox has a strong visual affordance for clicking; it appears as a pliant area, either because of its little square or, in Windows 95, because of its 3D recessing. Once the user clicks on it and sees the checkmark appear, he has learned all he needs to know to make it work at will: click to check, click again to uncheck. The checkbox is simple, visual and elegant.

The checkbox is, however, primarily a text-based gizmo. The checkable box acts as a visually recognizable icon next to its discriminating text. This works in just the way that icons to the left of text items in a listbox help the user visually discriminate their type. Like those listbox entries, however, the graphic supports the text, rather than the other way around. The checkbox is a familiar, effective idiom, but it has the same strengths and weaknesses as menus. The exacting text makes checkboxes unambiguous. The exacting text forces the user to slow down to read it, and takes a considerable amount of real estate.

Traditionally, checkboxes are square. Users recognize visual objects by their shape, and the square checkbox is an important standard. There is nothing inherently good or bad about squareness; it just happens to have been the shape originally chosen, and many users have already learned to recognize this shape. There is no good reason to deviate from this pattern. Don't make them diamond shaped or round, regardless of what the marketing or graphic arts people say.

Perhaps we could do to the checkbox what the buttcon did to the menu. Perhaps we could develop a checkbox gizmo that dispensed with text and used an icon instead. Well, sort of. We won't get far trying to iconize the checkbox, but we can replace the checkbox function with another evolving idiom: the buttcon.

The push-button evolved into the buttcon by replacing its text with an icon, then migrating onto the toolbar. Once there, the metamorphosis of the button continued by the simple expedient of allowing it to stay in the recessed—or pushed-in—state when clicked, then returning to the raised aspect when it is clicked again. The character of the gizmo changed sufficiently to move it into an entirely different category, from imperative to selection gizmo! The state of the buttcon is no longer momentary, but rather locks in place until it is clicked again. I call this idiom, shown in Figure 25–1, a **latching buttcon**.

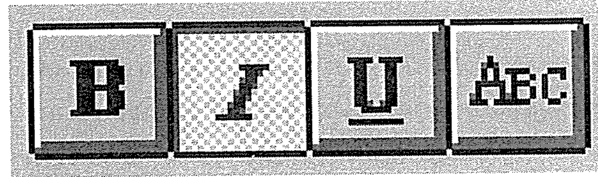


Figure 25-1

The latching buttcon was invented by applying the simple expedient of not letting the button pop back out after it has been clicked. What is remarkable about this idiom is that it moves the buttcon idiom from the imperative category—a verb—into the selection category—a noun. It has all of the idiomatic and space-saving advantages of the buttcon, except that it doesn't issue an immediate command.

The default toolbar configuration on Microsoft's Office suite of programs seems tacitly to separate the momentary, imperative buttcons from the latching, selection buttcons. Generally, they only put imperative buttcons on the top bar and put mostly selection buttcons on the others. Other than that, there are no visible differences between the two. There are no differences in their respective ToolTips, either. I don't believe that this sleight of hand has been noticed by anyone—Microsoft included—and it certainly doesn't bother me, though it is really a gross inconsistency. It is another example demonstrating how consistency is not a user interface design principle and is something that can often be flouted with impunity if the situation calls for it.

Consistency is not necessarily a virtue

The latching buttcon is widely superseding the checkbox as a single-selection idiom. Latching buttcons devote a smaller portion of their pixels to excise than checkboxes do, as you can see in Figure 25-2. They are smaller because they can rely on pattern recognition instead of text reading to indicate their purpose. Of course, this means that they exhibit the same problem as imperative buttcons: the inscrutability of the icon. We are saved once again by ToolTips.

Those tiny, yellow popup windows give us just enough text to disambiguate the buttcon, without permanently consuming too many pixels.

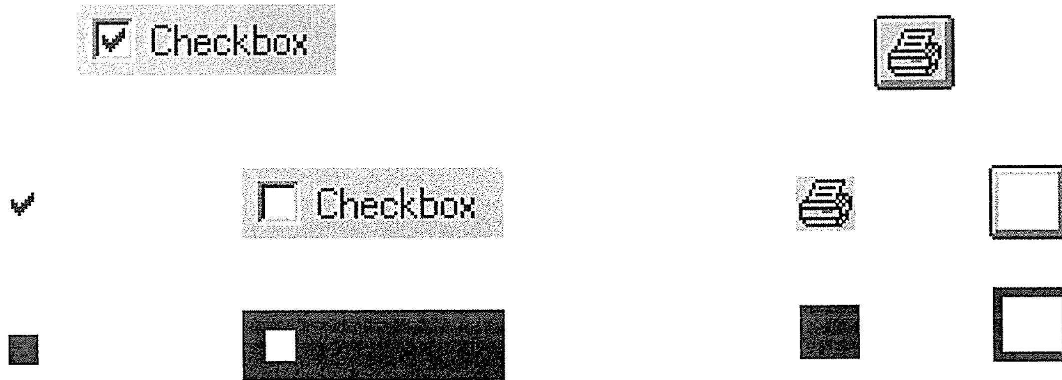


Figure 25-2

The venerable checkbox is an idiom that has been around since the beginning of GUIs. The buttcon is only a few years old. The latching variant of the buttcon is widely displacing the checkbox, and here is one reason why. If you compare the amount of video real estate devoted to excise (unproductive overhead) to the amount of real estate doing useful work as an operating gizmo, you see that buttcons are significantly more efficient. Not only are they more conservative of pixels overall, but the percentage of pixels devoted to useful work is much higher for latching buttcons than for checkboxes. The disadvantage of the latching buttcon is that it lacks the discriminating text of the checkbox. Ta da! Our old friend Mr. ToolTips to the rescue.

Menu items and momentary buttons often do the **flip-flop** thing.

If a flip-flop gizmo controls print resolution, for example, it will say "draft mode" until you click it then it will say "presentation mode." The control affords that you can click it, so when it says "presentation mode," it intends to mean that by pressing it you will get into presentation mode. Of course, then the gizmo changes to say "draft mode," to indicate that pressing it will get you there. This technique means that the control serves double-duty as an indicator of which state you are in. Unfortunately, it always shows "draft mode" when you are in presentation mode and vice versa. The gizmo can either serve as a state indicator or as a working control, but not both (see Figure 25-3).

The solution to this one is to either spell it out—"Change to presentation mode"—or to use some other technique entirely. Replacing it with two radio buttons is a popular choice. See Word's standard page setup dialog, which has

mutually exclusive radio buttons for portrait and landscape orientation. They allow control and indicate current state. The downside is that they consume a lot of real estate.

Another approach is pictorial. Draw a picture of the page in portrait. When the user clicks on it, it rolls over onto its side to show that you are in landscape orientation. This is very memorable and engaging, but it is not necessarily very discoverable. That depends on how well the rest of your program has influenced the user to expect that a small picture is pliant and will have some effect. Cursor hinting will help. However, it's important not to put the image on top of a button. If you do, you will have just created a pictographic flip-flop with the same conflicting messages as those with a text label.

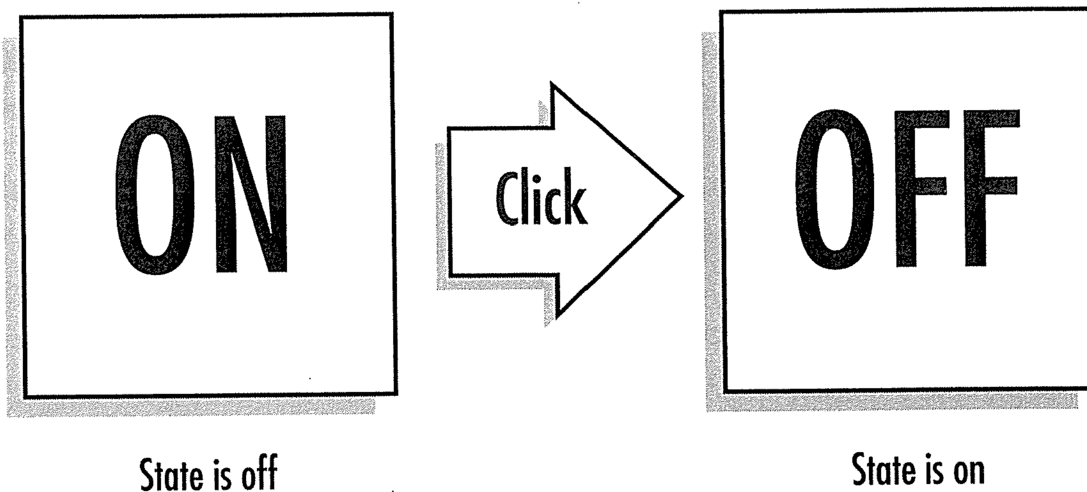


Figure 25-3

Flip-flop controls are very efficient. They save space by controlling two mutually exclusive options with a single gizmo. The problem with flip-flop controls is that they fail to fulfill the second duty of every gizmo—to inform the user of their current state. If the button says "ON" when the state is off, it is clear as mud what the setting is. If it says "OFF" when the state is off, though, where is the "ON" button? Don't use 'em. Not on buttons, and not on menus!

Radio buttons

A hoary variant of the checkbox is the **radio button**. The name says it all: When radios were first put in automobiles, it was discovered that manually tuning an analog radio with a rotating knob while driving was dangerous to your

health, so automotive radios were offered with a newfangled panel consisting of a half-dozen chrome-plated push-buttons, each of which would twist the tuner to a pre-set station. Now you could tune to your favorite station, without taking your eyes off the road, just by pushing a button. The idiom is a powerful one, and it still has many practical uses in interaction design, but I get a spooky feeling when we use interface idioms in our programs that are copied from a tube-based, AM radio from a 1963 Studebaker.

The behavior of radio buttons is **mutually exclusive**, which means that when one option is selected, the previously selected option automatically deselects. Only one button can be selected at a time. We techno-geeks frequently use the word **mux** as a convenient contraction of the phrase "mutually exclusive," and we often use it in reference to these gizmos (some geeks use a slightly different contraction: **mutex**).

In consequence of mux, radio buttons always come in groups of two or more, and one radio button in each group is always selected. (Technically speaking, there is no enforcement of this mutual exclusion, nor is there enforcement of the always-one-selected rule. The individual programmer is quite free to break the rules.) A single radio button is undefined—it must act like a checkbox instead.

Radio buttons are even more wasteful of video space than checkboxes. They waste the same amount of space as checkboxes, and for the same reasons, but radio buttons are only meaningful in groups, so their waste is always multiplied. In some cases the waste is justified, particularly where it is important to show the user the full set of available choices at all times. This should sound vaguely pedagogic, and it is. Radio buttons are well suited to a teaching role, which means they can be justified on infrequently used dialog boxes but should not be visible on the surface of a sovereign application where we must cater to daily users.

For the same reason that checkboxes are traditionally square—that's how we've always done it—radio buttons are round. There are no reasons to change this shape other than aesthetic or marketing ones, and these reasons take back seat to the established tradition. Motif's radio buttons are diamonds so, I assume, this isn't hard-and-fast. Even Microsoft is susceptible to this silly and counterproductive thinking: One of the beta versions of Windows 95 was shipped with diamond-shaped radio buttons. Cooler heads prevailed, though, and the final version shipped with good 'ol round ones.

Radio buttons are one of the oldest GUI idioms, and consequently many designers see them as somehow better than other, newer idioms, but this isn't so. In some cases, radio buttons are being supplanted by more modern idioms.

As you might imagine, the buttcon has also done to the radio button what it did to the checkbox: replaced it on the surface of an application. If two or more latching buttcons are grouped together and mux linked—so that only one of them at a time can be latched—they behave in exactly the same way as radio buttons. They form what I call a **radio buttcon**.

They work just like radio buttons: One is always selected—latched down—and whenever another one is pressed, the first one returns to its normal—raised—position. The alignment gizmos on Word's toolbar are an excellent example of a radio buttcon, as shown in Figure 25–4.

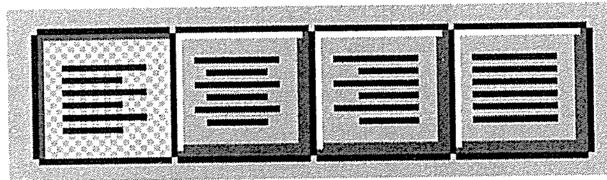


Figure 25-4

Word's alignment gizmos are a mux buttcon group, acting like radio buttons. One is always selected, and when another is clicked, the first one returns to its normal, raised position. This variant is a very space-conservative idiom that is well-suited for frequently used options.

Just as in all of the buttcon idioms, these are very efficient consumers of space, letting experienced users rely on pattern recognition to identify them and letting infrequent users rely on ToolTips to remind them of their purpose. First-time users will either be clever enough to learn from the ToolTips or will learn more slowly, but just as reliably, from other, parallel, pedagogic command vectors.

The combuttcon

A variant of the radio buttcon is a dropdown version. Because of its similarity to the combobox gizmo, I call this a **combuttcon**. It is shown in Figure 25–5.

Normally, it looks like a single latched buttcon, but if you click and hold on it, it drops down a menu of several latching buttcons. You slide the cursor down the same way you do on a pull-down menu to select one of the buttcon items. When you release the mouse button, the selection is made and the one you select now appears as the single buttcon on the toolbar. (Like menus, the menu of buttcons should also deploy if the user clicks once and releases. A second click makes the selection.)

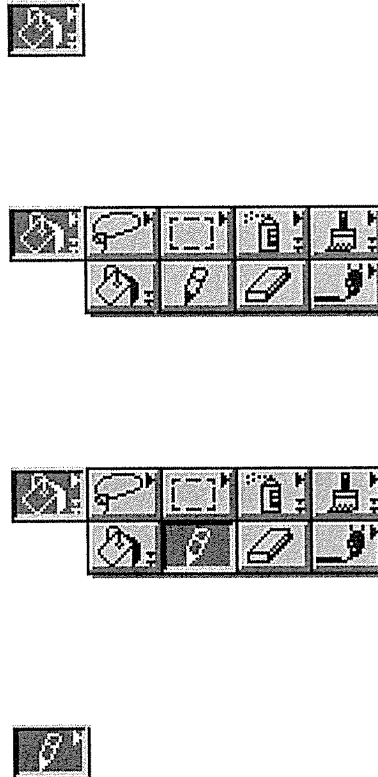


Figure 25-5

This is what I call a "combuttcon." It is a mux-linked group of latching buttcons that behave like a dropdown combobox. Pressing the mouse button while over the combuttcon drops down a menu of buttcons. Slide the cursor down to the desired one and release. The newly selected buttcon shows on the toolbar as the selected option. Think of this idiom as a way to cram several related buttcons into the space of a single one. It is less powerful than just putting up four latching buttcons, but useful when space is at a real premium.

Drawing a small, inverted triangle in the lower right corner of the combuttcon can serve as a visual hint that this buttcon is different. You can vary this idiom quite a bit, and creative software designers are doing just that in the never-ending bid to cram more functions onto screens that are always too small.

You can see a Microsoft variant in PowerPoint, where the buttcons for specifying the colors of fills, lines, text and shadows show combuttcon menus that

look more like little palettes than stacks of buttcons. As you can see from Figure 25–6, these menus pack a lot of power and information into a very compact package. This facility is definitely for frequent users, particularly minnies, and not at all for first-timers. However, for the user who has at least a basic familiarity with the available tools, the idiom is instantly clear once it is discovered or demonstrated. This is an excellent gizmo idiom for sovereign-posture programs, with which users can be expected to spend long hours interacting. It demands sufficient manual dexterity to work a menu with relatively small targets but is much faster than going to the menu bar, pulling down a menu, selecting an item, waiting for the dialog box to deploy, selecting a color on the dialog box and then pressing the OK button.

Figure 25-6

This combuttcon is taken from PowerPoint. The buttcon in the upper left corner controls the "fill color" of the selected object. Clicking or pressing the buttcon causes the drop-down menu to appear. Colors are selected from the eight swatches at the top, or from one of the text items, most of which bring up selection dialogs. This dense packing of information, both input and output, is indicative of the direction in which good user interfaces are moving.

Listbox

Selection gizmos that present lists of text strings are often called **picklists** because they offer lists of items from which the user can pick a selection.

The picklist is a powerful tool for simplifying interaction because it eliminates the possibility of making an incorrect selection. Essentially, a picklist is a gizmo for selecting from a finite set of text strings. The first picklist was the **listbox gizmo**. In Windows 95, it has been made largely obsolete by the listview gizmo.

Listboxes were one of the original six gizmo classes that came with Windows 1.0. They are little text windows with a vertical scrollbar on the right-hand edge. You can add lines of text to the box, and the scrollbar will move them up or down. The user cannot select text as in a word processor—on a character-by-character basis—but he can select a single line of text at a time. A listbox variant allows multiple selection, where the user can have more than one line selected at one time.

The original listbox gizmo was for text only—it wasn't until Windows 3.0 that non-text items could be inserted—and that strong influence permeates its operation to this day. A listbox filled with line after line of text unrelieved by visual symbols is a dry desert indeed. Some adventurous programmers have adapted them for graphics, but let's face it, most programmers aren't that masochistic.

In Chapter 15, I talked about the problems with scrollbars, and later in this chapter I'll talk about them some more. For now, suffice it to say that the problems with scrollbars are numerous. The traditional listbox gizmo joins a brain-dead, text-only box with a clumsy, annoying scrollbar in a particularly fetid combination of idioms.

I've often felt that the listbox control is the one whose potential was exploited the least by Microsoft. In the early days of Windows, I, like many other developers, was forced to write my own listbox gizmo class. Succeeding versions of Windows obsoleted my work, but my aspirations grew, too. Now that more robust gizmo interfaces are here, I hope some ambitious gizmo builder creates a really great listbox.

Actually, Microsoft seems to have done a pretty decent job in Windows 95 with the new **listview gizmo**. Among other features, it allows each line of text to automatically be preceded with an icon.

This is excellent news, because I believe that every item in every list should show an identifying visual icon next to each text entry. Because of this, I will now forget that listboxes ever existed and deal only with the listview gizmo for all future design. I suggest you do the same. Too bad they didn't improve the scrollbar while they were at it.

> ***Design tip***: Every text item in a list should have an identifying graphic icon next to it.

What listboxes, er, I mean, listviews are good for is displaying lists of items and allowing the user to select one or more of them. They are also good idioms for providing a source of draggable items. If the items are draggable within the listview itself, it makes a fine tool for enabling the user to put items in a specific order. For example, you might want to rearrange a list of the people in your department in descending order by how frequently you work with them. There is no automatic function that will do this; you just have to drag them until it's right.

Many lists are not static but are modified by users adding, deleting and changing the text of entries. Supporting this requires the ability to key, directly into the listview, a new text item or a modification to an existing one.

Much to my surprise and pleasure, Microsoft did a pretty good job with the listview gizmo. In addition to icons, it supports drag-and-drop and edit-in-place. Both of these necessities were lacking from the older, Windows 3.x listbox gizmo. Until Windows 95, Microsoft handed rank and file programmers some pretty dull tools. It should come as no surprise that much of what they built with them is coarse and difficult to use.

The weight of history and habit still put the listbox gizmo onto a lot of dialog boxes. Thankfully, most of those dialogs appear quickly, allow the user to select a single item, and then go away. To provide better interaction, the programmer must use the listview gizmo exclusively (although I'll use the terms "listview" and "listbox" interchangeably).

Earmarking

Generally, the user selects an item in a listview as input to some function, like selecting the name of a desired font from a list of several available fonts. Selection in a listbox is discrete rather than concrete, and it is entirely conventional, with keyboard equivalents, focus rectangles and items shown with COLOR_HIGHLIGHT.

Occasionally, the listbox is used to select multiple items, and this can introduce complications. There is nothing conceptually wrong about using multiple selection in listboxes, but in practice it can be problematic. The selection idiom is very well suited for single selection but much weaker for multiple selection. In concrete data, multiple selection is contiguous, so it is, above all, visible. In discrete data, multiple selection works adequately if the entire playing field is

visible at once, like the icons on a desktop. If two or more icons are selected at the same time, you can clearly see this because all of the icons are visible.

If the pool of available discrete items is too large to fit in a single view and some of it must be scrolled off screen, the selection idiom immediately becomes unwieldy. This is the normal state for listboxes and listviews. Their normal state of selection is mux; that is, when you select one thing, the previous selected thing deselects. If you are expecting different behavior, it is far too easy to select an item, then scroll it into invisibility and select a second item, forgetting that you have now *deselected* the first item because you can't see it anymore.

The alternative is equally unpalatable: the multiple selection option in the standard listbox merely disables the mux linking in the selection algorithm. Things now work absolutely perfectly: the user selects one item after another and each one stays selected. The fly in the ointment is that there is no visual indication that things are behaving differently from the norm. It is just as likely that a user will select an item, scroll it into invisibility, then spot a more desirable second item and select it *expecting the first—unseen—item to automatically deselect* by way of the mux standard. You get to choose between offending the first half of your users or the second half. Bad news.

The correct action, of course, is to use a completely different idiom from selection, one that is visually distinct. You only need to do this in the case where you have multiple selection within a list that may scroll off screen, but that case arises frequently. Please understand that what we are doing is nothing more or less than multiple selection; we are just doing it with a more appropriate visual idiom. The selection idiom is heavily used in GUIs and, in this case, it is overused. When things can scroll off the screen, multiple selection requires a better, more distinct idiom.

It just so happens we already have a well-established idiom to indicate that something is, well, let's say "chosen" instead of selected. Of course, that idiom is the checkbox. Checkboxes communicate their purpose and their settings quite clearly and, like all good idioms, are extremely easy to learn. Checkboxes are also very clearly *dis*associated from any hint of mutual exclusion. If we were to add a checkbox to every item in our problematic listbox, the user would not only clearly see which items were selected and which were not, he would also clearly see that the items were not mux linked, solving both of our problems in one stroke. I call this checkbox alternative to multiple selection **earmarking**.

An example of earmarking is shown in Figure 25–7.

Figure 25-7

Selection is normally a mutually exclusive (mux) operation. When the need arises to discard mux in order to provide multiple selection, things can become confusing if some of the items can be scrolled out of sight. Earmarking is a solution to this. Put checkboxes next to each text item and use them instead of selection to indicate the user's choices. Checkboxes are a clearly non-mux idiom and a very familiar GUI idiom. Users grasp the workings of this idiom right away.

Earmarking also solves another niggling problem with multiple selection. Multiple selection list gizmos, when they are created, have no selected items. However, in some variants, once the user selects an item, there is no way to return to a state where nothing is selected. In other words, there is no idiom for selecting nothing. If the listbox is used in the sense of an operand selector for a function dialog box, the CANCEL button provides the escape route if the user changes his mind; but if the listbox isn't on a dialog box, he may be stuck. Earmarking doesn't operate under the same rules as selection, and each item in the list is independent. One click checks the box; a second click unchecks the box.

The inability to deselect all items in a listbox has given rise to a kludge that can be seen on Word's Modify Style dialog box, among many others. There is an entry in the listbox for "(none)" or "(no selection)." What a hack! Don't stoop to this level of programming. If you need a way to turn off all of the selections, use an earmarking idiom to make choosing items different from selecting items.

Dragging-and-dropping

Listboxes can be imagined as little palettes of goodies to use in a direct-manipulation idiom. If the list in Figure 25-7 were on an email program, for

example, you could click on an entry and drag it to a message to select a device for output. It's not really selection, because it is a completely captive operation. Without a doubt, many programs would benefit if their listboxes and listviews supported dragging-and-dropping.

Such listviews with draggable items are commonly used to gather items in a desired set for the user. Having two adjacent listboxes, one showing available items and the other showing chosen items, is a common GUI idiom. A pair of push-buttons placed between them allow items to be selected and transferred from one box to the other. It is so much more pleasant when the idiom is buttressed with the ability to just click-and-drag the desired item from one box to another without having to go through the intermediate steps of selection and function invocation.

Ordering listboxes

Sometimes, the need arises to drag an item from one position to another position in the same listbox. Actually, this need arises far more often than most designers seem to think. At least, I find myself frequently wishing that I could. I want to order the items in a list according to how often I use them or how important they are, for example, instead of just alphabetically. Many programs offer automatic sort facilities for important lists. The Explorer, for example, lets me sort my files by name, by type, by modification date and by size. That's nice, but what I really want is to order them by importance. Algorithmically, the program could order them by frequency of access, but that won't always get the results I want. For example, I may have to access Word's system files a lot, but they are important to Word, not to me. I'd like to press a sort button to get a good head start, then click-and-drag the Word files down to the bottom of the list. (Of course, this is the kind of thing that an experienced user wants to do after long hours of familiarization. It takes a lot of effort to fine-tune a directory like this, and the program *must* remember my exact settings from run-to-run—otherwise, the ability to reorder things is worthless.)

Being able to drag items from one place to another in a listbox is powerful, but it demands that autoscrolling be implemented. I discussed autoscrolling in Chapter 18. If you pick up an item in the list but the place you need to drop it is currently scrolled out of view, you must be able to scroll the listbox without putting down the dragged object.

Horizontal scrolling

Listboxes normally have a vertical scrollbar for moving up and down through the list, the way Santa scans his long strip of paper with the names of good boys and girls. Listboxes can also be made to scroll horizontally. This feature allows the programmer to put extra long text into the listbox with a minimum of effort. It offers nothing to the user.

Scrolling a list of text horizontally is a terrible thing, and it should never, ever need to be done. When a listbox scrolls up and down, entire lines come and go from view, but the text inside the box remains completely readable. However, when a text list is scrolled horizontally, it hides from view one or more of the first letters of every single line of text showing. This makes *none* of the lines readable and utterly destroys the continuity of the text. To see what I mean, take your bookmark and cover up just the first two characters of each line in this paragraph. See how hard it becomes to read? Yes, it is decipherable, but you have to strain at it. The purpose of computers is to eliminate strain from the lives of humans.

Design tip: Never scroll text horizontally.

The horizontal scrollbar provided for this action is the exact same gizmo as the vertical one (aside from its orientation, of course), and it was designed for a relatively permanent setting—not a temporary one. The whole action of horizontally scrolling a listbox is inappropriate: the user clicks on the scrollbar to read the right half of the entry in question, then must move the cursor back to the left and click again to restore the listbox to its normal, left-justified position. If the scroll took more than one click on the scrollbar, or worse yet, mixed an arrow click with a bar click, getting back to square one is extremely difficult and far more complex than the operation justifies.

If you find a situation that seems to call for the horizontal scrolling of text, search for alternative solutions. Begin by asking yourself why the text in the listbox is so long. Can you shorten the entries? Can you use more than one line per entry to avoid that horizontal length? Can you wrap the text onto the next line? Can you allow the user to enter aliases for the longer entries? Can you use graphical entries instead? Can you use a smaller typeface? You should alternatively be asking yourself if there is some way to widen the listbox. Can you rearrange things on the window or dialog to expand horizontally?

The best answer will usually be to wrap the text onto the next line, indenting it so it is visually different from other entries. This, of course, means that you now have a listbox with entries of variable height. The listbox gizmo from Microsoft lets you handle this with the ownerdraw option, but it demands lots of work by the programmer. Rats! The listbox gizmo just keeps giving us fits when we try to do the right thing. What a market opportunity for some clever entrepreneur.

Microsoft provides a multi-column option for their listbox gizmo, but I would still never use it. This option lets you organize listbox items like a newspaper, with snaking columns. This works for newspapers, because everything is laid out for us to read at once. However, once some of it is hidden off screen, the user must simultaneously scroll vertically with his eyes while scrolling horizontally with a mouse and scrollbar. The mental management is far too difficult. Multi-column display may help programmers, but it offers nothing to users.

Remember, I'm just talking about lists of text. For graphics, there is nothing wrong with horizontal scrollbars or horizontally scrollable windows in general. I am just saying that providing a text-based listbox with a required horizontal scrollbar is like providing a computer with a required pedal-powered electrical generator—bad news.

Entering data to a listbox

An enormous area where little work has historically been done is in enabling the user to make direct text entry into an item in a listbox. The old listbox gizmo merely punted on this, and it takes some pretty nifty coding to implement it by yourself. Of course, the need to enter text where text is output is widespread, and much of the hacky-kludgy nature of dialog box design can be directly attributed to its programmer trying to dodge the bullet of having to write edit-in-place code.

Finally, the listview and treeview gizmos in Windows 95 offer an edit-in-place facility. The Explorer in Windows 95 uses both of these gizmos, and you can see how they work by renaming a file or directory. Excellent! To rename a file, you just click twice on the desired name and enter whatever changes are desired. (I don't want to look a gift horse in the mouth, so I won't whine about the clunky way Microsoft implemented this idiom, except to say that you have to carefully delay that second click to shift into edit-in-place mode; otherwise, it will be interpreted as a double-click and launch the file. I think Microsoft

could have done better, but, like I said, this is a tradeoff I'm happy to make to get free edit-in-place.) Many—no, most—of the items displayed in listboxes could benefit by being able to be edited by the user.

The edge case that makes edit-in-place a real problem is adding a new entry to the list. Most designers use other idioms to add list items: Press a button or select a menu item and a new, blank entry is added to the list, and the user can then edit-in-place its name. It would be more sensible if you could, say, double-click in the space between existing entries to create a new, blank one right there. Ahhh, wishful thinking...

The real-world solution to this problem that has actually emerged over the past few years is the combobox, which we'll talk about next.

Combobox

Windows 3.0 introduced a new gizmo called the **combobox**. It is—as its name suggests—a combination of a listbox and an edit field. It provides an unambiguous method of data entry into a listbox. This solution is something of a scatter-gun approach, but it is effective. The other attribute of the combobox that makes it a winner is its popup variant that is extremely conservative of video real estate.

With the combobox gizmo, there is a clear separation between the text-entry part and the list-selection part. The user's confusion is minimized, and you can bet that the programming was significantly easier. For single selection, the combobox is a superb gizmo. The edit field can be used to enter new items, and it also shows the current selection in the list. When the current selection is showing in the edit field, the user can edit it there—sort of a poor man's edit-in-place.

Because the edit field of the combobox shows the current selection, the combobox is by nature a single-selection gizmo. There is no such thing as a multiple-selection combobox. Single selection implies mux, which is one of the reasons why the combobox is fast replacing groups of radio buttons for mux-linked options. The other reasons include its space efficiency and its ability to add items dynamically, something that radio buttons cannot do.

When the dropdown variants of the combobox are used, the gizmo shows the current selection without consuming space to show the list of choices.

Essentially, it becomes a list-on-demand, sort of like a menu provides a list of immediate commands on demand. A combobox is a popup listbox.

The video efficiency of the combobox allows it to do something remarkable for a gizmo of such depth and complexity: it can reasonably reside permanently on a program's main screen. No listbox in a supporting role could ever do that. It can even fit comfortably on a toolbar. It is a very effective gizmo for deployment on a sovereign-posture application. There are currently four comboboxes visible on my word processor's toolbars, for example. This effectively crams a huge amount of information and usefulness into a very small space. Using comboboxes on the toolbar is more effective than putting the equivalent functions on menus, because the comboboxes show me their current selection without requiring any action on my part, such as pulling down a menu to see the current status. Once again, the gizmo that delivers the goods with the smallest permanent video footprint wins the Darwinian battle for pixels.

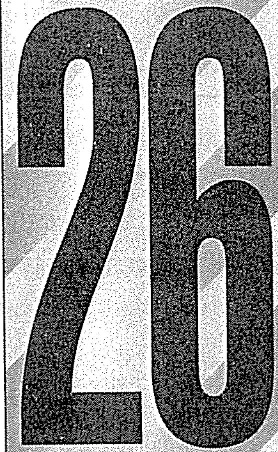
If drag-and-drop is implemented in listboxes, it should also be implemented in comboboxes. For example, being able to open a combobox, scroll to a choice and then drag the choice onto a document under construction is a very powerful idiom. Because comboboxes fit so well on toolbars, the idiom has real appeal for adding direct manipulation to sovereign applications. Drag-and-drop functionality should be a standard part of comboboxes.

The utility of the combobox collapses if the situation calls for multiple selection; the idiom just can't handle it, and you must return to the plain listbox. The listbox consumes significant space on-screen—enough so that it should probably never be considered practical for permanent deployment. Instead, it should be relegated to transient dialog boxes.

Treeview gizmo

Windows 95 brings us this new gizmo. It is a listview that can present hierarchical data. It shows a sideways tree, with icons for each entry. The entries can be expanded or compressed the way many outline processors work. As a programmer, I like this presentation. It is used for the left half of the Explorer, and I find the format of the display to be effective—certainly more effective than scattering icons around on my desktop. Unfortunately, it is problematic for users because of the trouble many people have with hierarchical data structures. If the treeview contents are restricted to no more than two levels, however, it can nicely show a monocline grouping of data.

Entry and Display Gizmos



In the last chapter, I discussed imperative and selection gizmos. In this chapter, I will examine the remaining two types of gizmos, those for entry and display. Display gizmos enable the user to configure the form and appearance of their windows. The job of entry gizmos is to accept the user's unstructured data, which puts them squarely in between error-prone, fallible humans and rigid, deterministic data management software.

Entry gizmos

Entry gizmos enable the user to enter new information into the program, rather than merely selecting information from an existing list.

The most basic entry gizmo is a text-edit field. Like selection gizmos, entry gizmos represent nouns to the program. Because one half of a combobox is an edit field, some combobox variants qualify as entry gizmos, too. Also, any gizmo

that lets the user enter a numeric value is an entry gizmo. Many of the new gizmos from third-party developers—like spinners, gauges, sliders and knobs—fit in this segment. Microsoft has not shipped any other standard entry gizmos, so there isn't a single dominant form, but in a vacuum standards form from the bottom up, so stay tuned.

Bounding

I call any gizmo that restricts the available set of values that the user can enter a **bounded-entry gizmo**. A slider that moves from 1 to 100, for example, is bounded (but don't confuse this with a "bound" gizmo used for database access).

Regardless of the user's actions, no number outside those specified by the program can be entered with a bounded gizmo. The essential fact about bounded gizmos is that it is impossible to enter an invalid value with one.

Conversely, an edit field can accept any data the user keys into it. I call an open-ended entry idiom like this an **unbounded-entry gizmo**.

With an unbounded-entry gizmo, it is easy to enter an invalid value. The program may subsequently reject it, of course, but the user can still enter it.

Simply put, bounded gizmos should be used wherever bounded values are needed. If the program needs a number between 7 and 35, presenting the user with a gizmo that will accept any numeric value from –1,000,000 to +1,000,000 is not doing him any favors. He would much rather be presented with a gizmo that embodies 7 as its bottom limit and 35 as its upper limit. Users are smart, and they will immediately comprehend and respect the limits of their sandbox.

Design tip: Offer bounded gizmos for bounded input.

It is important to understand that I am talking here about a quality of the entry gizmo and not of the data. To be a bounded gizmo, it needs to clearly communicate, preferably visually, the acceptable data boundaries to the user. An edit field that rejects the user's input *after* he has entered it is *not* a bounded control. Edit fields are never bounded entry fields.

Most quantitative values needed by software are bounded, yet many programs allow unbounded entry with edit fields. When the user inadvertently enters a

value that the program cannot accept, the program issues an error message box. This is cruelly teasing the user with possibilities that aren't. "What would you like for dessert? We've got everything," we say. "Ice cream," you respond. "Sorry, we don't have any," we say. "How about pie?" you innocently ask. "Nope," we say. "Cookies?" "Nope." "Candy?" "Nope." "Chocolate?" "Nope." "What, then?" you scream in anger and frustration. "Don't get mad," we say indignantly, "we have plenty of fruit compote." This is how the user feels when we put up a dialog box with an unbounded edit field and ask for the number of desired veeblefetzers. He enters "17" and we reward this innocent entry with an error message box that says "You can only have between 4 and 8 veeblefetzers." This is extremely bad user interface design, and don't ever let me catch you doing it. You should use a bounded gizmo that automatically limits the input to 4, 5, 6, 7 or 8.

If the bounded set of choices is composed of text rather than numbers, you can still use a slider of some type, or a combobox or listbox. Figure 26-1 shows a bounded slider used by Microsoft in the Display Settings dialog box of Windows 95. It works like a slider or scrollbar but has four discrete positions that represent distinct resolution settings. They could easily have used a combobox in its place, but isn't the slider nicer to look at and friendlier? Part of its appeal comes from the innate visibility of the gizmo: you can see the scope of the control just by looking. A combobox isn't much smaller, but it keeps its cards hidden—a less friendly stance.

Figure 26–1

A bounded gizmo only lets you enter valid values. It does not let you enter invalid values, only to reject them when you try to move on. This figure shows a bounded slider gizmo from the Display Settings dialog in Windows 95. The little slider has four discrete positions. As you drag the slider from left to right, the legend underneath it changes from "640 by 480 pixels" to "800 by 600 pixels" to "1024 by 768 pixels" to "1280 by 1024 pixels." Why didn't they use a combobox? Which would you prefer? I rest my case.

If the program requires a numeric value that must remain within specific boundaries, give the user a control that intrinsically communicates those limits and prevents him from entering a value outside of the boundaries. The scrollbar control class does this; it is a bounded gizmo—the only one that comes with Windows. Although scrollbars have significant drawbacks, they are exemplary in one area: they allow the user to enter quantitative information by analogy. Scrollbars allow the user to specify numeric values in relative terms, rather than by directly keying in a number. That is, the user moves the sliding thumb to indicate, by its relative position, a proportional value for use inside the program. They are less useful for entering precise numbers, though many programs use them for that purpose. Newer gizmos, like spinners, are better for entering exact numbers.

Spinners

A new gizmo type, commonly called a **spinner**, is rapidly gaining currency, especially in Microsoft's Office suite.
It grays the difference between bounded and unbounded gizmos to a certain extent. The spinner gizmo is a small edit field with two half-height buttons attached, as shown in Figure 26-2.

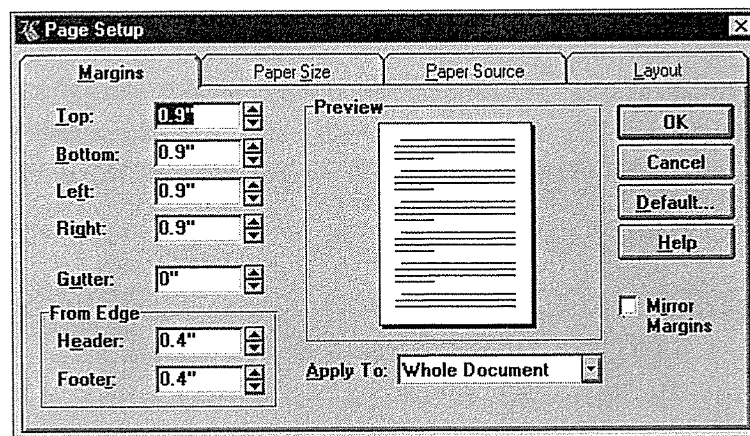


Figure 26-2

The Page Setup dialog from Word for Windows makes heavy use of the spinner gizmo. On the left side of the dialog, you see a stack of seven of these new controls, whose popularity is growing fast. By clicking on either of the small, arrowed buttons, the specific numeric value is made to increase or decrease in small, discrete steps. If the user wants to make a large change in one action, or to enter a precise setting, he can use the edit field portion for direct text entry. The arrow button portion of the gizmo embodies bounding, while the edit field portion does not. Does that make this a bounded gizmo?

Using either of the two small arrow buttons enables the user to change the value in the edit window in small, discrete steps. These steps are bounded—the value won't go above the upper limit set by the program or below the lower limit. If the user wants to make a large change in one action, or to enter a specific number, he can do so by clicking in the edit window portion and directly entering keystrokes into it, just like entering text into any other edit field. Unfortunately, the edit window portion of this gizmo is unbounded, leaving the user free to enter values that are out of bounds, or even unintelligible garbage. In the Page Setup dialog box in the figure, if the user enters a bad value, the program behaves like most other rude programs, issuing an error message box explaining the upper and lower boundaries and requiring the user to press the OK button to continue.

Overall, the spinner is an excellent idiom and can be used in place of plain edit fields for most bounded entry. In Part VII, we will discuss ways to improve gizmo error handling.

Unbounded-entry fields

The primary unbounded-entry gizmo is the **text-edit gizmo**. This simple box allows the user to key in any text value.

Edit fields are usually very simple boxes where a word or two of data can be entered by the user, but they can also be moderately sophisticated text editors in their own right. The user can edit text within them using the standard tools of concrete selection (as discussed in Chapter 16) with either the mouse or the keyboard.

Text-edit gizmos are generally used either as data-entry fields in database applications, as option-entry fields in dialog boxes, or as the entry field in a combobox. In all of these roles they are frequently called upon to do the work of a bounded-entry gizmo. However, if the desired values are finite, the text-edit gizmo should not be used. If the acceptable values are numeric, use a bounded numeric-entry gizmo, such as a slider or knob, instead. If the list of acceptable values is composed of text strings, then a picklist should be used so the user is not forced to type.

Sometimes the set of acceptable values is finite but too big to be practical for a picklist. For example, a program may require a string of any 30 alphabetic characters excluding spaces, tabs and punctuation marks. In this case, a text-edit

gizmo is probably unavoidable even though its use is bounded. If these are the only restrictions, however, the edit gizmo can be designed to reject non-alphabetic characters and characters after 30, thus making it bounded.

Validation

From the gizmo's point of view, there is really no such thing as invalid data. Data can only be adjudged invalid in the context of the program. For example, "1995" is valid in a text-entry gizmo that gathers the year but not in one that gathers the month. Physically, an unbounded-entry gizmo cannot recognize invalid data—only the program can make the actual determination of validity. From the program's point of view, a *bounded*-entry gizmo will only hand it valid input. Thus, by definition, an *unbounded* gizmo *can* return invalid input to the program.

An unbounded gizmo that is used to gather bounded data is in a moral bind. It must serve two bosses: The gizmo must blithely accept whatever data the user keys in; then, if the program judges that input to be invalid, the gizmo is forced to be the bearer of someone else's bad news. I believe that this moral bind of putting unbounded gizmos in the role of accepting bounded input is one of the most important contributors to user dissatisfaction with computers.

Accepting bounded data into unbounded gizmos causes user dissatisfaction

If the data is bounded—but not too bounded—the program must let the user enter the data, only to reject it afterwards. Although there are some mitigating steps, there really is no good way to solve this problem. Unless...

There *is* one way to solve this problem: the program should just go ahead and accept whatever the user enters. In other words, eliminate semi-bounded data. Either coerce the correct data with a bounded gizmo, or accept whatever the user gives you in an unbounded gizmo. Most programmers reject this solution. They do not feel that their programs can accept, for example, "asdf;lkj" as input to a social security number field. I'll wait until Chapter 29, "Managing Exceptions," to argue the point.

The way most programmers have historically dealt with this dilemma is by creating what I call a **validation gizmo**, or an unbounded text-entry gizmo with built-in editing.

Many edits are commonplace, covering such things as dates, phone numbers, zip codes and social security numbers, and are packaged with text-edit gizmos as a unit, particularly for database data-entry applications. You can purchase variants of the text-entry gizmo that will only allow numbers or letters or phone numbers, or that will reject spaces and tabs, for example.

Although the validation gizmo is a widespread idiom, it is a very poor one. Tactically, though, it is often necessary, so we'll ignore the bigger issues for now and look at practical ways to make it better. The key to successfully designing a validation gizmo is to give the user generous feedback. An entry gizmo that merely refuses to accept input is just plain rude, not helpful, and will guarantee an angry, resentful and upset user.

A fundamental improvement, based on the axiom that things that behave differently should look different (Chapter 21), is to make validation gizmos visually distinct from unvalidated gizmos. I recommend using a different color and line style for the gizmo's border. A dashed line in blue instead of solid black would alert the user that something was up. He would then observe it more closely than usual when he used it for the first time, and would be poised to learn about its unique behavior.

Design tip: Show validated-entry gizmos with a different border.

The main tool for validation gizmos is to provide rich status feedback to the user. Unfortunately, the edit gizmo as we know it today provides virtually no built-in support for feedback of any kind. The designer must specify such feedback mechanisms in detail, or none will be provided.

Some gizmos reject the user's keystrokes as he enters them. When a gizmo actively rejects keystrokes during the entry process, I call it a **hot validation gizmo**. A text-only entry gizmo, for example, may accept only alphabetic characters and refuse to allow numbers to be entered. Some gizmos work the opposite way, rejecting any keystrokes other than the numeric digits 0 through 9. Other gizmos reject spaces, tabs, dashes and other punctuation in real-time. Some variants can get pretty intelligent and reject some numbers based on live calculations, for example, unless they pass a checksum algorithm.

When a hot validation gizmo rejects a keystroke, it must make it clear to the user that it has done so. It should also clue the user into why it made the rejection, though that is more difficult. If an explanation is proffered, the user will be less inclined to assume the rejection is arbitrary. He will also be in a better position to give the program what it wants.

The user is expecting to be able to enter keystrokes at will; this is the nature of the keyboard. If the gizmo is going to reject some keystrokes based on their value, it must clearly communicate this to the user.

Sometimes the range of possible data is such that the program cannot validate it until the user has completed his entry, rather than at each individual keystroke. The editing step then takes place only when the gizmo loses focus, that is, when the user is done with the field and moves on to the next one. The editing step must also take place if the user closes the dialog—or invokes another function if the gizmo is not on a dialog box. If the gizmo waits until the user finishes entering data before it edits the value, I call it a **cold validation gizmo**.

The gizmo may wait until a name is fully entered, for instance, before it interrogates a database to see if it is an existing entry. Each character is valid by itself, yet the whole may not pass muster. The program could attempt to verify the name as each character is entered, but that would probably bring the network and server to their knees with the extra workload. Besides, although the program would know at any given instant whether the name was valid, the user could still move on while the name was in an invalid state.

Another way to address this is by maintaining a countdown timer in parallel with the input and reset it on each keystroke. If the countdown timer ever hits zero, do your validation processing. The timer should be set to around 400 milliseconds, although you may wish to user test this for a more precise number. The effect of this is that as long as the user is entering a keystroke faster than once every 400 ms, the system is extremely responsive. If the user pauses for more than 400 ms, the program reasonably assumes that the user has paused to think (something that takes months in CPU terms) and goes ahead and performs its analysis of the input so far.

To provide rich visual feedback, the entry field could change colors to reflect its estimate of the validity of the entered data. The field could show in shades of pink until the program judged the data valid, where it would change to white or green.

Another good solution to the validation gizmo problem is what I call a **clue box**.

This little popup window looks and behaves just like a ToolTip. By convention, ToolTips are yellow, so the clue box would be pink or some other color, and it explains the range of acceptable data for a validation gizmo, either hot or cold. Whereas a ToolTip appears when the cursor sits for a moment on a gizmo, a clue box would appear as soon as the gizmo detects an invalid character (it can also display unilaterally just like a ToolTip if the cursor sits unmoving on the field for a second or so). If the user enters, for example, a non-numeric character in a numeric-only field, the program would put up a clue box near the point of the offending entry, yet without obscuring it. It would say, for example, "0–9." Short, terse but very effective. Yes, the user is rejected, but he is not also ignored. The clue box would work for cold validation, too, as shown in Figure 26-3.

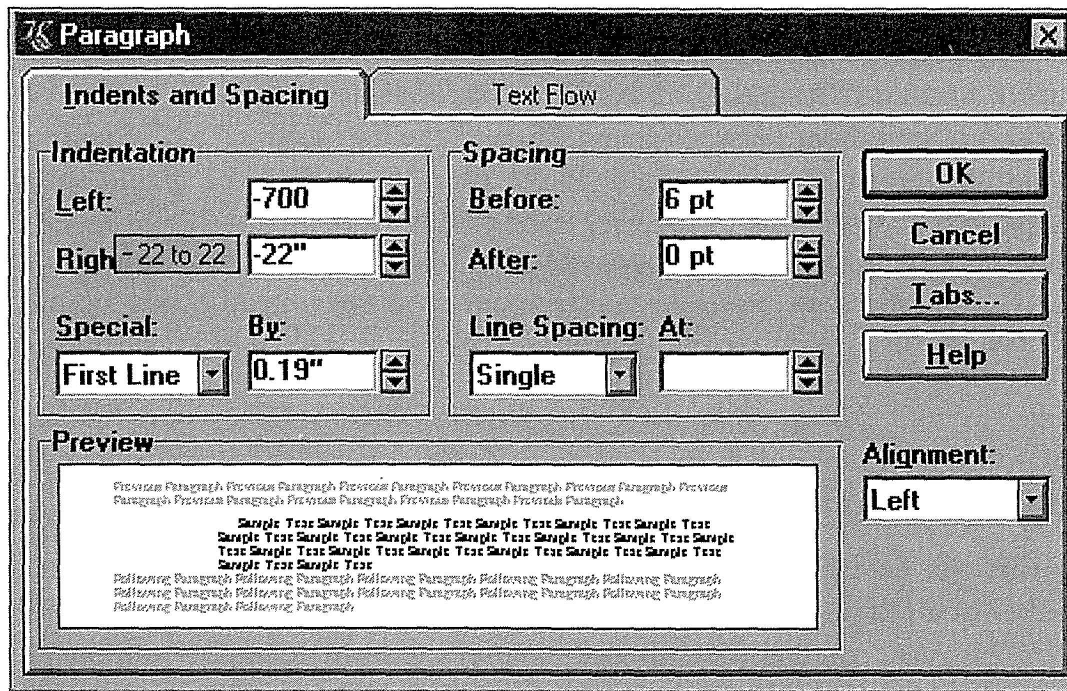


Figure 26-3

The ToolTip idiom is so effective I'm surprised that it hasn't been extended into other uses. Instead of yellow ToolTips offering flyover labels for buttcons, we could have pink ones offering flyover limits for unbounded edit fields. These could easily double as error-message-box-eliminating hint windows that I call clue boxes. In the example shown here, when the user enters some value lower than the lowest allowable value, instead of stopping the proceedings with an idiotic error message box, the program could replace the value in the edit field with the lowest allowable value (–22 in this case) and display the pink clue box that modelessly explains the reason for the substitution. The user can enter some new value or accept the minimum but in either case he can proceed without getting an error message.

Typically, an edit field is used to enter a numeric value needed by the program, like the point size of a font. The user can enter anything he wants, from "5" to "500," and the field will accept it and return the value to the owning program. If the user enters garbage, the gizmo must make some kind of decision. In Microsoft Word, for example, if I enter "asdf" as a font point size, the program issues an error message box informing me that "This is not a valid number" and then reverts the size to its previous value. I think the error message box is rather silly, but the summary rejection of my meaningless input is perfectly appropriate. But what if I had keyed in the value "nine"? The program rejects it with the same curt error message box. I believe that if the gizmo were programmed to think of itself as a numeric-entry gizmo, it might take a different approach. It doesn't bother me if the program converts the "nine" into a "9," but it certainly is incorrect when it says that "nine" is "not a valid number." Without a doubt, it is valid and the program has put its foot in its mouth.

Barring other tools, a simple rejection of input data is better than a rejection coupled with an error message box. For example, if a cold validation gizmo can only accept a number between 5 and 25, and the user enters "50," the gizmo should change to 25 and proceed. If the user enters "2," the gizmo should change to 5 and proceed. If the user enters "asdf," the gizmo should revert to the previously valid value and proceed.

It's nice when a text-edit gizmo is smart enough to recognize appropriate qualifiers. For example, if a program is requesting a measurement, and the user enters "5i" or "5in" or "5 inches," the gizmo should not only report the result as *five*, but it should report *inches* as well. If the user enters "5mm," the gizmo should report it as five *millimeters*.

Say that the field is requesting a column width. The user can enter either a number or a number and an indicator of the measurement system as described above. The user could also be allowed to enter the word "default," and the program would set the column width to the default value for the program. The user could alternately enter "best fit," and the program would measure all of the entries in the column and choose the most appropriate width for the circumstances. There is a problem with this scenario, however, because the words "default" and "best fit" must be in the user's head rather than in the program somewhere. This is easy to solve, though. All we need to do is provide the same functionality through a combobox. The user can drop down the box and find a few standard widths and the words "default" and "best fit." The dropdown would look like the one in Figure 26-4.

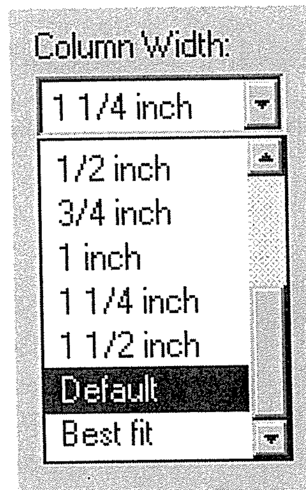


Figure 26-4

The dropdown combobox makes an excellent tool for bounded entry fields, because it can accommodate entry values other than numbers. The user doesn't have to remember or type words like "default" or "best fit" because they are there to be chosen from the drop-down list. The program interprets the words as the appropriate number, and everyone is satisfied.

The user can pull down the combobox, see the words "Default" and "Best fit" and choose the appropriate one. With this idiom, the information has migrated from the user's head into the program where it visible and choosable. This is good.

In Chapter 29, "Managing Exceptions," I'll talk about using audible feedback with validation gizmos.

Using an edit field for output

The text-edit gizmo, with its familiar system font and visually articulated white box, strongly affords data entry as well as data output. Yet software developers frequently use the text-edit gizmo for output only. The edit gizmo certainly works as an output field, but to use this gizmo for output only is to bait-and-switch your user, and he will not be amused. If you have text data to output, use a text gizmo and not a text-edit gizmo. If you want to show the amount of free space on disk, for example, don't use a text-edit field, because the user is likely to think that he can get more free space by entering a bigger number. At least that is what the gizmo is telling him with its equivalent of body language. There's a good example of this in Chapter 11 (see Figure 11-6).

On the other hand, if you are going to output changeable information, go ahead and output it in a fully editable text gizmo and wire it up internally so that it works as it appears to. For example, output the volume name in a text-edit gizmo so that the user can directly edit the visible name to change it on the disk.

Rich text gizmos

With the advent of the **rich text gizmo** in Windows 95, it will be very possible for simple text-edit gizmos to take on the excise overhead of word processors.

It is important for designers to be clear about the scope of options that should be exposed to the user when edit fields are implemented with rich text gizmos. Activating entire paragraph formatting subsystems is not appropriate for simple entry fields that are expecting a single word or number as input.

The rich text gizmo isn't really useful as a tool for entering structured data for fitting into rigidly structured databases. It is, however, handy for such tasks as composing email messages or taking notes.

Insert and overtype

The handling of insert and overtype is another example where good judgment is called for. In most text editors there is a user-settable option toggling between insert and overtype mode. These two modes are omnipresent in the world of word processors and, like FORTRAN, never seem to die. Insert and overtype are modes that would make Larry Tesler scream: they are significant changes in the behavior of an interface, with no visible indication until after the user has interacted, and there is no clear way into or out of these modes except by means of a rather obscure key.

Fifteen years ago when I regularly used a primitive, character-based text editor, I used both modes. Although I was a power user, I needed all the help I could get with that program, and overtype mode let me shave away many keystrokes. Today, with my modern GUI word processor, I can't imagine using overtype mode, and I can't imagine anyone else actually wanting to use anything other than insert mode, but I know they are out there—I had a long exchange on the Net with one of them just recently. For edit fields of one line, adding controls beyond simple unimodal entry and editing is foolish—the potential for trouble is far greater than the advantages delivered to users. Of course, if you are writing a word processor, the story is different.

Display gizmos

I call the fourth category **display gizmos**. These are the adjectives of the gizmo grammar, modifying how our screens look.

These gizmos are used to display and manage the visual presentation of information on the screen rather than the information itself. Typical examples include scrollbars and screen splitters. Gizmos that control the way things are displayed visually on the screen fall into this category, as do those that merely display static information for output only. These include paginators, rulers, guidelines, grids, groupboxes, and those 3D lines called dips and bumps. Although many of these gizmos are familiar to Windows users under various names, only groups and scrollbars actually come with Windows.

Probably the simplest display gizmo is the text gizmo. This variant of the STATIC control class merely displays a written message at some location on the screen. The management job that it performs is pretty prosaic, serving only to label other gizmos and to output data that cannot or should not be changed by the user.

Text gizmos instead of edit gizmos

The only significant problem with text gizmos is that they are often used where edit gizmos should be. Most information in a computer can be changed by the user. Why not allow the user to change it at the same point the software displays it? Why should the mechanism to input a value be different from the mechanism to output that value? The answer is that the program should not separate these related functions. In almost all cases where the program displays a value, it should do so in an editable field so the user can click on it and change it.

For example, the Windows 3.1 File Manager program shown in Figure 26-5 displays the name of the disk drive currently displayed but, in order to change its value, the user must go to the menus and request a dialog box. The user should be able to enter the new value directly where the program displays it. Of course, this is a low-frequency operation, so putting a permanent edit gizmo there may not be necessary (it would then have to be in the tab-navigation sequence and need a keyboard accelerator). Instead, if the user clicks the mouse anywhere over the name, an edit gizmo could appear, filled with the current value as a fully selected default and allow the user to enter a new name or modify the old one.

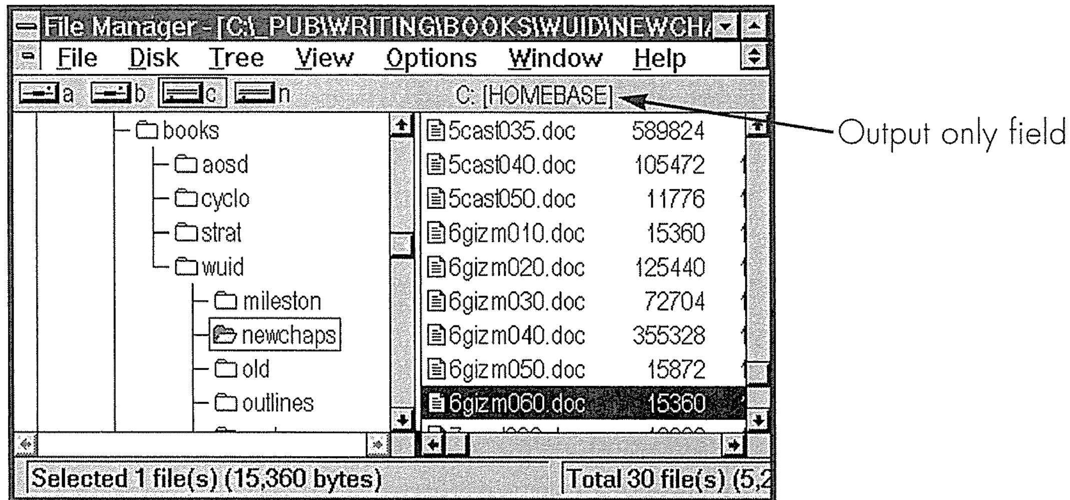


Figure 26-5

The Windows 3.1 File Manager program displays the name of the current disk volume. The user should be able to change the name by keying it in right here where the program outputs it, instead of having to go through a menu item and dialog box. Because the frequency of update is low, it doesn't need a permanent edit field, but rather, when the user clicks directly on the volume name, an edit field should appear to allow entry. Once the name has been entered, a click outside of the field, or a keypress of ENTER closes the field and records the new name.

Those darned scrollbars

Scrollbars are a very frustrating gizmo, fraught with problems, hard to manipulate and wasteful of pixels. The idiom is another of those originals that came from Xerox PARC, so it has a certain cachet that is difficult to overcome. The scrollbar is, without a doubt, both overused and under-examined. In its role as a window scroller—a display gizmo—its application is appropriate, at least. In many cases, though, it is used where it shouldn't be only because designers don't seem to have any better ideas. That's a bad rationale for any aspect of software design.

The singular advantage of the scrollbar—aside from its availability—is its proportional rendering of value. The scrollbar's **thumb** is the central, draggable box that indicates the current position.

If a scrollbar represents percentage, for example, the user can see that a scrollbar whose thumb is about equidistant between the ends represents a quantity of 50%. The fact that the scrollbar conveys no information about its terminal values detracts considerably from its usefulness as a sliding value selector. The scrollbar's proportional rendering, flawed in implementation though it may be,

is an excellent type of visual feedback. The lessons of the scrollbar should not be ignored by gizmo designers.

A big shortcoming of the scrollbar is its parsimonious doling out of information to the user. It should instead generously inform us with information about the information it is managing. The new Windows 95 scrollbar uses thumbs that are proportionally sized to show the percentage of the document that is currently visible. It could also tell us

- What page we are on
- How many pages there are in total
- The first sentence (or item) of each page as we scroll with the thumb
- The page number (record number, graphic) as we scroll with the thumb

Additionally, the scrollbar is parsimonious with functions. It manages the bulk of our navigation within documents; it should give us powerful tools for going where we want to go quickly and easily. It could

- Offer us buttons for skipping ahead by pages/chapters/sections
- Offer us buttons for jumping to the beginning and end of the document
- Give us tools for setting bookmarks that we can quickly return to

The scrollbar also demands a high degree of precision with the mouse. Scrolling down or up in a document is generally much easier than scrolling down *and* up in document. You must position the mouse cursor with great care, taking your attention away from the data you are scrolling.

The scrollbar consumes a relatively large amount of video real estate. For what it takes, it doesn't give much back to us. I'd really like to see some better scrolling idioms come into popular use.

New Gizmos

Gizmos are valuable tools for interacting with users because they encapsulate lots of complex behavior in a ready-to-use package. The ready availability of gizmos causes us to rely on them for designing user interfaces. We use them because they are available. Unfortunately, the set of gizmos that come with Windows is pallid and thin. Those available through the emerging APIs like VBX, OCX and OLE show promise but are still incomplete and not yet available to all tool platforms. Gizmos remain a significant arena for invention and entrepreneurship. Here are some ideas for future gizmo innovation.

Directly manipulable tools

The most striking area for invention is in bounded-entry gizmos. So many programs offer us dialog boxes with edit fields or spinners to gather data that could more easily be entered through direct manipulation. Using click-and-drag idioms in

place of entry gizmos not only makes input clearer and easier, but it can also automatically change a previously unbounded gizmo to a bounded one.

One of my clients had a program with some rudimentary drawing tools, including a drop shadow and a drawing grid. The drop shadow put a black shape behind and slightly offset from any graphical object to make it visually distinct. The drawing grid forced all drawing-related direct manipulations to normalize to an invisible grid, allowing drawn objects to neatly line up automatically.

Both of these facilities were adjustable by way of simple, nearly identical dialog boxes. The drop shadow offset (or grid span) was controlled by two edit gizmos, one specifying the horizontal offset, the other specifying the vertical offset. I replaced both of these with what I call a **sun gizmo**, as shown in Figure 27-1.

Because drop shadows usually appear at the bottom and to the right of an object, the sun gizmo showed a small, round sun above and to the left of a sample graphical object. The user could click and drag the sun around and, as he did so, the drop shadow under the sample object would move in response, growing or shrinking in the vertical and horizontal axes. The physical limitations of where the sun could be dragged automatically bounded the gizmo, so the user could not "enter" an invalid offset. The drop shadow could easily be turned off by dragging the sun to the center of the object, where the shadow would, according to the laws of physics, disappear.

When the user moved his cursor over the sun, cursor hinting indicated that the sun was directly manipulable.

The grid in this program was similarly controlled through a small dialog box with two edit gizmos for specifying the horizontal and vertical interval of the grid. I replaced these gizmos with a swatch of grid whose spacing the user could adjust by direct manipulation. When the cursor moved over the swatch, the cursor changed to indicate that the sample grid was pliant. The user could then just click-and-drag anywhere in the swatch to adjust the spacing. Dragging up closed the vertical interval. Dragging down opened it. Dragging right or left worked the same for the horizontal axis. In order to adjust one axis without inadvertently affecting the other, we used a drag threshold like the one described in Chapter 18.

If the resolution—the drag distance required to increment by one—is high enough, no supporting edit fields are needed to set the values at exactly some

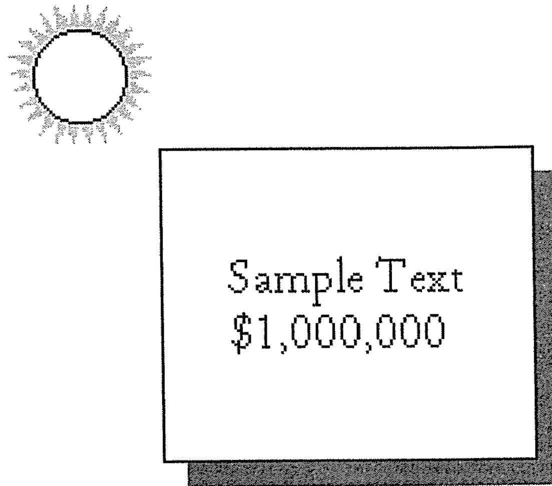


Figure 27-1

The small image of a sun in the upper left corner is a directly manipulable idiom for telling the program how to place a drop shadow on the selected object. Not only is the direct manipulation easier and less obscure than entering a numeric value, but the very nature of the sun also incorporates protective borders: the user cannot physically drag it to an invalid location; therefore, he cannot inadvertently enter an invalid value. This means that the sun gizmo is a bounded gizmo.

desired number. Of course, that number would have to be displayed either on (or alongside) the sun or swatch for this to work well. The principle of "enter where you output" demands that those numbers be editable anyway, for those who would rather keyboard than directly manipulate.

Both the sun gizmo and the grid swatch gizmo replaced ugly, inappropriate text gizmos with the direct manipulation of graphical objects that were visually appropriate to the desired result. The user could stay in context, even though the tools were used rarely enough to justify them residing on dialog boxes. Both gizmos finessed away the need for text entry and provided visual, bounded, direct manipulation of the settings.

Rubberweeks

Another client needed a calendar gizmo for a scheduling application, so we designed a new gizmo that we called **rubberweeks**. The requirements for this gizmo included the ability to adapt to any time scale from several hours to several weeks. In addition, the user needed to be able to place small icons on the calendar to indicate deadlines and other events that need to be firmly anchored in time.

The key feature of rubberweeks was its ability to stretch. When the user dragged along it with the mouse, the calendar scrolled, but when the user dragged with the ALT-shifted mouse, the calendar's scale changed. If the user dragged right, the scale stretched longer. If the user dragged left, the scale stretched shorter. If the user needed to schedule a series of events that would occur within the next three days, for example, he could ALT-stretch the rubberweeks to display a total of just three days. This allowed him to take maximum advantage of the space available on the screen.

By double-clicking on the rubberweeks, a small, triangular icon could be physically placed on the calendar to indicate a deadline. Once placed, the icon could be freely dragged to adjust its place in time.

The advantages of gizmos like the sun, the grid swatch and rubberweeks are their proportional representations of the information they represent and manage. To duplicate their functionality with existing gizmos would require descending into a world of text, where abstract symbols need interpretation. In the graphic world, the sun gizmo shows the current setting visually and immediately, and the control of it is direct and proportional. Dragging a sun gizmo to set the extent of a drop shadow much more closely approaches the ideal of a directly manipulable interface. Like the carpenter swinging his hammer, the user can place the sun just so, and clearly and immediately receive direct feedback regarding his input. He doesn't have to wonder whether three pixels are too few or four pixels are too many. He can *see* when it's just right.

Extraction gizmos

One of the most noticeable attributes of text, edit gizmos is how stupid they generally are. If the application at hand calls for entering an address, for example, there are no "address" gizmos, yet that is just what we need. Validation gizmos exist, true, but their ability to adapt to variable input, like a whole address, is nil. In fact, they are designed purposely to reject variable input. A simple zip code validation gizmo, for example, will reject anything that doesn't fit the archetype of a zip code: five (or nine) contiguous digits.

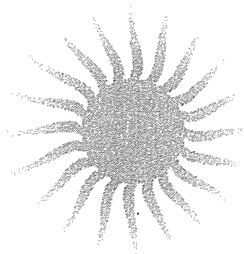
An address gizmo is an example of what I call an **extraction gizmo**.

This is a completely different approach to the problem of data entry. An extraction gizmo parses the contents of a free-form text-entry gizmo according to some rules about the general class of input. For example, instead of having one field for street address, one for apartment, suite or mail stop, one for city, one

for state, and one for zip code, there is a single text-entry gizmo, several lines tall. The user keys in the entire desired address in the single field, and the gizmo makes sense of the various parts of that address.

A normal text-edit gizmo has a method (or entry point, or value, depending on your language/coding model) to examine its "contents." The contents of a normal zip code entry field would be whatever the user entered. An extraction gizmo would have several other "content" examination methods in addition to the traditional one. They would include

- Street address line
- Street
- Number
- Geographical designation
- Second address line
- Suite number
- Building
- Apartment
- Mail stop
- Floor
- City line
- City
- State
- Province
- District
- Zip code
- Postal code
- Country



Not all of these values would be filled, only those that are relevant, depending on what the user entered. The gizmo would do its best to determine which parts of the entered text belonged in each category. There are basically three levels of discrimination in this process. The gizmo would return the text verbatim as the user entered it. Each line of the address would be separated: Street address line, Second address line, City line. Then each separate element of the address would be parsed into its appropriate category.

A gizmo like this enables users to enter addresses the same way they manually prepare an envelope: by typing the address as a block. The computer does the work of separating the fields out for efficient categorizing in a database program. A program would then be able to, for example, sort the addresses by street name or by zip code even though the address was entered as just a human-readable block.

Useful types of extraction gizmos include those for proper names, email addresses, physical descriptions and telephone numbers. An extraction gizmo could easily pull a person's first name, last name, middle name, honorific, rank and title from a single field so the user isn't forced to manually separate them out at entry time.

Yes, there will be an error rate, but it won't be high and it won't be significant. An address-parsing algorithm can easily pull apart the vast majority of addresses. If someone tried deliberately to enter garbage, the extraction gizmo would probably fail to discriminate accurately, but then again, how many of your employees deliberately enter garbage? An end user with a shrink-wrapped application who deliberately enters garbage into his own system certainly won't blame *you* for the problem.

When coded into a dialog box, a telephone-number extraction gizmo, for example, would recognize phone numbers by applying a series of simple lexical and semantic rules. The outputs of the field would consist of the raw text as entered by the user, along with an array of possible phone, fax, cellular and pager numbers. If the gizmo is unable to discern these numbers from the contents, well, it can't, but in most cases where these numbers are discernible by humans they are also discernible by software. Let's take an example: Say that I key this text into a phone number extraction gizmo:

```
415-366-2300w, Home:367-9824 (415) 367-9976 fax 508 2031 pager
```

I entered some pretty torturous stuff here: inconsistent and missing symbols and varying labels. Can you figure out what I've typed? Sure you can. I bet a computer program could, too! The first number is a well-formed number with area code, prefix and body. It has a "w" appended to it that can reasonably be interpreted as being my work phone. The comma is just a separator. The next number is prefixed by the word "Home:" so it's nature is clear. The absence of an area code is not much of a crisis. The program could easily assume it is a 415 number—the same as all of the others. If it were different, it is likely that I would have entered it. If not, it's not a hanging offense. The third number is trickier. Certainly, it a well-formed number, but what is it? The word "fax" is ambiguous. It could be referring to the third number or the fourth number. The last word in the entry, "pager," disambiguates the two because it must be referring to the fourth number, so "fax" must refer to the third. The lack of a hyphen in the fourth number should be no problem because the number is still a recognizable, well-formed phone number.

If I wanted to really tax the gizmo, I could enter something more problematic, like this:

```
4558, 1-800-555-1212 25433 976-PORN
```

Well, this would certainly put a strain on things, but it wouldn't be impossible. The first number, 4558, is not a recognizable phone number, but it is a recognizable fragment of a phone number. When you want to call someone within your company through a private PBX, you often just enter a four-digit number. If the PBX's prefix is 488—which the program is likely to already know—the number from the outside would be 488-4558. The second number is still a well-formed number; it is just more complete than many others. It includes the long distance prefix "1" and adds a five-digit extension. We guess that it is an extension because it is not delimited from the 800 number. If it were four digits, we might have trouble discriminating between its being an extension or another in-house number. The last number is, well, recognizable even though it doesn't use all-numeric digits because its form is recognizable. Software might otherwise have difficulty determining that "976-PORN" is a phone number, except that we are talking about a field that is designed to process phone numbers—that's a big hint.

If experience is any guide, many of you are probably having trouble swallowing the idea of extraction gizmos. They seem to fly in the face of our tradition of

guaranteed data integrity. Well, yes and no. We will discuss this aspect of extraction gizmos in Part VII, "The Guardian."

Visual gizmos

Most of the traditional gizmos are merely encapsulations of text. Checkboxes, radio buttons, menus, text edits, listboxes and comboboxes are mostly text with a thin veneer of graphics added. They don't exploit the full potential of the GUI medium.

Most programs, when they have options to offer to the user, describe those options with text and offer that text to the user in a text-based selection gizmo like a combobox. If the options can be rendered visually, however, we should discard the combobox and let the user point-and-click on a picture of what he wants instead of just a text description.

Figure 27-2 shows a dialog box from Word with a couple of very visual gizmos. The gizmo in the lower left corner lets the user request complex bordering options by clicking on little images of borders, instead of asking for them by name. In this situation, this gizmo rescued Microsoft from a difficult dilemma because the number of bordering options is large. Rules can be independently specified for the top, bottom, left, right and in-betweens of a paragraph, and each border can have its own weight and style. Offering a combobox filled with hundreds of options like "thin left, really thin right, thick top, dashed thin bottom" would be pathologically bad. Alternatively, a dialog box with an array of individual comboboxes for each of the five possible borders would still be a nasty morass for the user. Microsoft's solution makes a feature out of a bug.

We don't have to save visual, directly manipulable gizmos for the tough stuff, though. In the upper left corner of the dialog in Figure 27-2 is another visual gizmo that offers an extremely simple mux-linked choice of one of three options: an outline, a drop-shaded outline or no border at all. These could easily have been radio buttons, but clicking on the little pictures is soooo much better. The user can click on the image of what he wants, instead of having to click on the words that describe what he wants. It is less ambiguous, faster and more direct.

Most publishers of software use radio buttons instead of visual gizmos like these because radio buttons come free with Windows and the visual gizmos don't. If publishers want a visual gizmo, they must pay designers and programmers to create them. This is not an expensive thing to do, relative to other

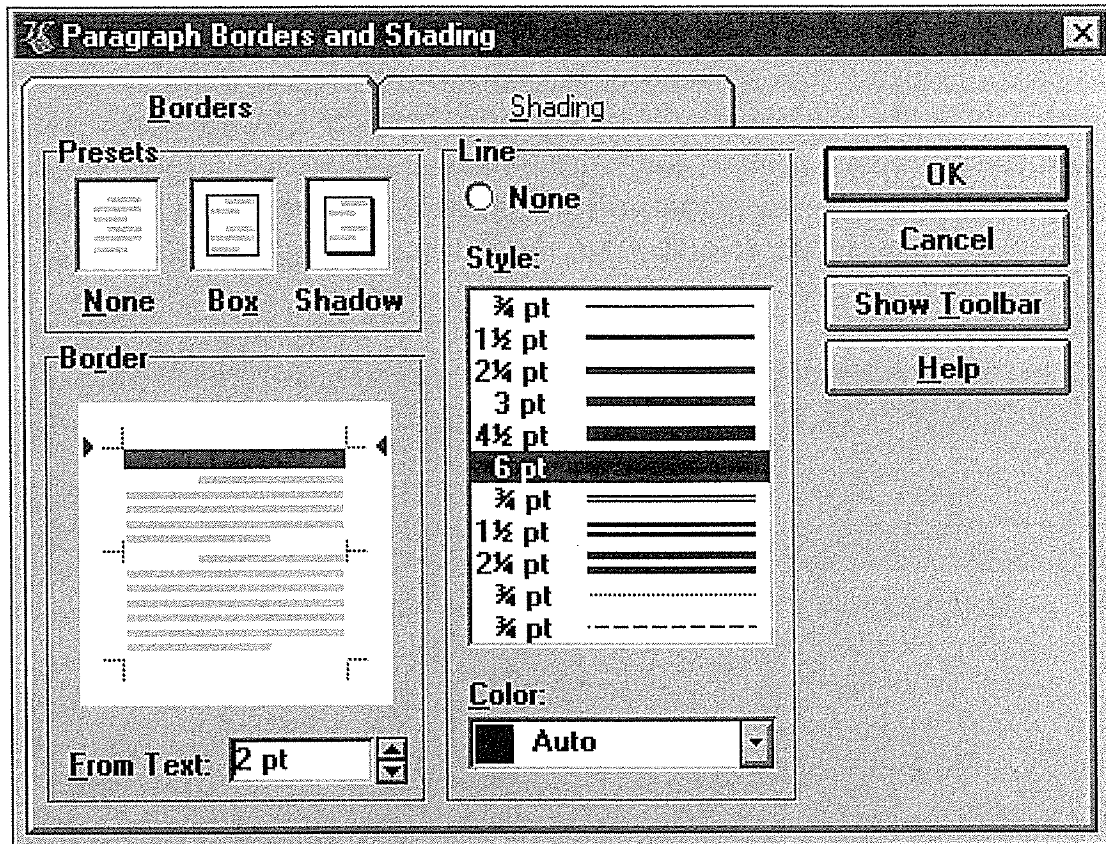


Figure 27-2

The Borders dialog box from Word shows the admirable trend toward visual, directly manipulable gizmos. On the left side of the dialog, there are two non-traditional gizmos labeled "Presets" and "Border." Preset lets the user click once on a miniature image of his desired result to quickly and easily achieve a frequently used result: a box, a drop shadow, or no border at all. Below it, the border gizmo not only acts as a preview of the current settings, but by clicking on any of the five places where lines can be specified, the user can create highly customized borders, one line at a time. Although I have some significant reservations about how Microsoft implemented borders and shading in the larger sense, this dialog box is exemplary in its use of non-text gizmos.

custom coding, but compared to a free text-based gizmo, it is very costly. Microsoft didn't put such visual gizmos into their word processor until the eighth or ninth revision, and it had generated many millions of dollars of revenue before then.

Figure 27-3 shows a beautifully crafted visual gizmo in the control panel of Windows 95. Instead of picking a time zone from a text list (although such a list exists on the dialog), you choose your time zone by clicking on a blue and green map of the world. When you select a zone, the map sensuously slides so

that your selection is centered in the window. *There was no need for Microsoft to do this,* just as there is no reason for a downtown law firm to have marble floors instead of linoleum. But as you run your hand along the teak and cherrywood trim of the lobby furnishings and slide gently into a soft, supple leather wing chair, you know true comfort and luxury. Sure, you'll pay for it; those lawyers are going to be a lot more expensive than their competitors with plywood, Formica and naugahyde. It all depends on what image you wish to convey.

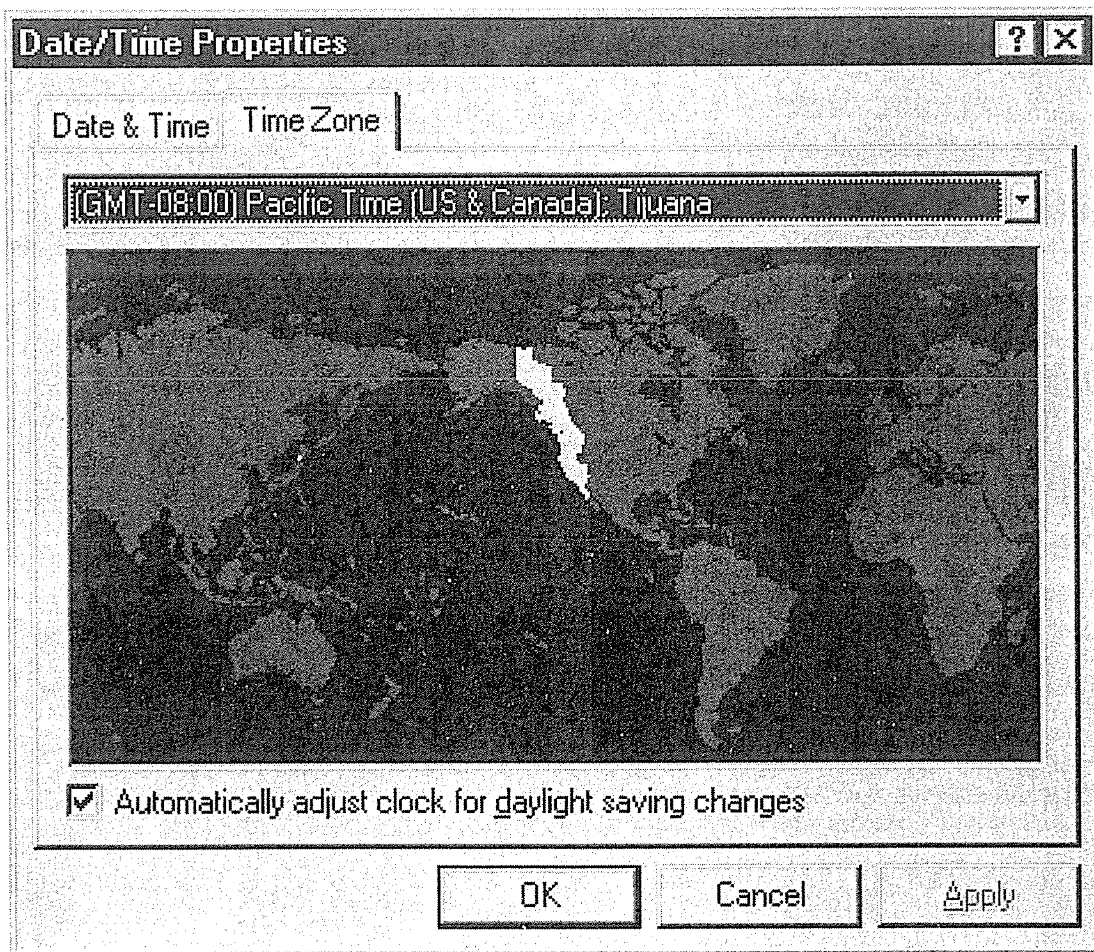


Figure 27-3

The new time zone dialog box in Windows 95 is an excellent visual gizmo. It clearly shows the selections available to the user attractively and graphically. If you live on the East Coast of the US, for example, all you need to do is click somewhere along the eastern seaboard and the map smoothly scrolls until the Eastern US is centered and highlighted. The animation speeds up and slows down so nicely that the effect is almost sensual. I've seen people spend many minutes playing with the dialog box for the sheer, tactile pleasure of it. Wow! It's like walking into the lobby of someone's office and finding marble, walnut and leather instead of stucco and plywood. If you want to add a sense of aesthetics to your program, make them tactile aesthetics.

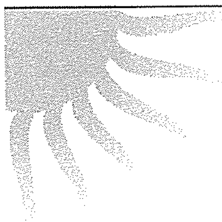
Paradoxically, gizmos are distinct objects on the surface of an application, but the path to improving them is to integrate them more intimately into the visual fabric of the program. All of the examples I've shown so far are undoubtedly hand-coded one-offs. This is an area of significant opportunity for a vendor: creating a generic visual-gizmo development kit that allows average programmers on average budgets to create visual, animated, directly manipulable gizmos for their products.

Adding visual richness

Adding visual richness to traditional gizmos, like adding icons to buttons, is an area whose potential has barely been scratched. Most gizmos can be enhanced with the addition of graphics, animation and sound.

We designed a progress meter that was functionally identical to run-of-the-mill implementations but was a lot more engaging because of its visual richness. The program downloaded a newspaper from an online service, and the progress meter reflected the status of that operation. Instead of a simple horizontal row of little rectangles that appear one at a time, we showed a dog walking from the left end of the gizmo to the right end of the gizmo, where a folded newspaper waited. When Rover got to the newspaper, the download was complete, and he gave a friendly bark before returning to sit attentively on the left again, waiting for the next download. Good doggie!

In a well-written novel, the protagonist usually doesn't come right out and state her views and opinions. Instead, she demonstrates her point of view by her actions. The novelist is showing us, instead of telling us, and this is a fundamental technique of good fiction. It is also a fundamental technique of good user interface design.



Show; don't tell

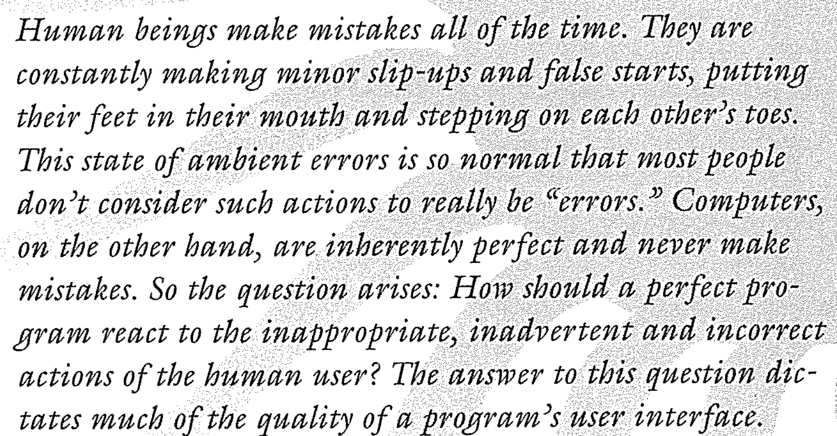
Instead of using words to tell your story, use pictures to show the user. I'm not talking about metaphoric icons here; rather, I'm saying that instead of using text to communicate some setting, draw a picture. Even though the picture

probably consumes more space, its ability to clearly communicate is well worth the pixels. In recent years Microsoft has discovered this fact, and the dialog boxes in Windows Word, for example, have begun to fairly bristle with little visualizations of their purpose instead of mere textual controls.

The Page Setup dialog box, shown in Figure 26-2, offers an image labeled "preview." This is an output-only gizmo, showing a miniature view of what the page will look like with the current margin settings on the dialog. Most users have trouble visualizing what a 1.2" left margin *looks* like. The preview gizmo shows them. You could go Microsoft one better by allowing input on the preview gizmo in addition to output. Drag the left margin of the picture and watch the numeric value in the corresponding spinner ratchet up and down.

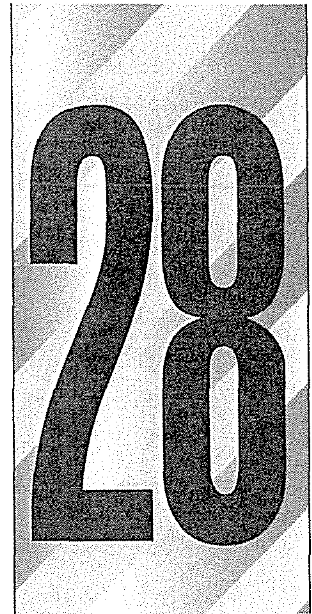
The associated text field is still important—you can't just replace it with the visual one. The text shows the result with precision, while the visual gizmo shows the result with accuracy.

Part VII: The Guardian
Protecting the User



Human beings make mistakes all of the time. They are constantly making minor slip-ups and false starts, putting their feet in their mouth and stepping on each other's toes. This state of ambient errors is so normal that most people don't consider such actions to really be "errors." Computers, on the other hand, are inherently perfect and never make mistakes. So the question arises: How should a perfect program react to the inappropriate, inadvertent and incorrect actions of the human user? The answer to this question dictates much of the quality of a program's user interface.

The End of Errors



In Part V, I discussed in detail all of the variants of the dialog box except one: the bulletin. The bulletin dialog box is issued unilaterally by the program when it is having some sort of problem or is confronting a decision that it doesn't feel capable of answering on its own. In other words, bulletin dialog boxes are used for error messages and confirmations, two of the nastiest components of modern software design. I believe that, with proper design, all error message and confirmation dialogs can be eliminated. Further, I believe that most of them *should* be. In this chapter, I'll tell you how.

Eliminating the error message box

There is probably no more-abused idiom in the GUI world than the error message box. When I lecture to groups of programmers, I make many bold assertions, but when I

assert that all error message boxes can be eliminated from all programs, it provokes them more than any other statement. Some of my listeners have come so unglued by this claim that they became apoplectic and couldn't consider any of my other thoughts or ideas. The proposal that a program doesn't have the right—no, the duty—to reject the user's input is so heretical that many practitioners dismiss it summarily. Yet, if we examine this assertion rationally and from the user's—rather than the programmer's—point of view, it is not only possible, but quite reasonable.

When I say to eliminate error messages, I don't mean to just discard the code that shows the actual error message dialog box, while still letting the program crash if a problem arises (although many programmers assume that is what I mean). Instead, I mean that we should alter our programs so they are no longer susceptible to the problem. You cannot just yank the error messages out of a program. You must replace the error-message-method of software protection with a kinder, gentler, more robust type of software that prevents error conditions from arising, rather than having the program merely complain when things aren't going precisely the way it wants. Like vaccinating it against a disease, we make the program immune to the problem, and then we can toss the message reporting it. To eliminate the error message, we must first eliminate the possibility of the user making the error.

I don't want you to crusade to eliminate all existing error messages. Instead, I want you to change your mental assumptions about all *future* error messages. Instead of assuming that error messages are normal, I want you to think of them as abnormal solutions to rare problems—as surgery instead of aspirin. Treat them as an idiom of last resort.

Users never want error messages. Believing that your users are satisfied with error messages confuses what they don't want with what they do. Users want to avoid the *consequences* of making errors, which is very different from saying that they want error messages. It's like saying people want to abstain from skiing when what they really want to do is avoid breaking their legs. Don Norman points out that people frequently blame themselves for errors in product design. Just because you aren't getting complaints from your users doesn't mean that they are happy getting error messages.

Bulletin Dialog Boxes

The familiar error message box is normally an application modal dialog that stops all further progress of the program until the user issues a terminating

command—like pressing the OK button. I call this a **blocking bulletin** because the program cannot continue until the user responds.

It is also possible for a program that has put up a dialog box to unilaterally take it down again. I call this a **sustaining bulletin** because the dialog disappears and the program continues without user intervention.

Sustaining bulletins are most frequently used as progress dialog boxes, reporting on the status of a time-consuming procedure. During the process, the dialog offers a CANCEL button so the user can terminate it if he changes his mind or grows impatient with the delay. In any case, when the program has completed the procedure, it pulls down the dialog.

Sustaining bulletins are sometimes used for error reporting. A program that erects an error message to report a problem may correct the problem itself or may detect that the problem has disappeared via some other agency. Some programmers issue an error message box merely as a warning—"Your disk is getting full"—and take it down again after it has been up for, say, 10 seconds.

An error message *must* stop the program. If it doesn't, the user may not be able to read it fully, or if he is looking away, he either won't see it or worse yet, see only a fleeting glimpse out of the corner of his eye. He will be justifiably suspicious that he has missed something important; something that will come back to haunt him later. He will now begin to worry: What did I miss? Was that an important bit of intelligence that I will regret not knowing? Is my disk full? Am I about to crash? This is true even if the problem has gone away by itself.

If a thing is worth saying with a dialog box, it's worth assuring that the user definitely gets the message. A sustaining bulletin can't make that guarantee. For this reason, the only justification for a sustaining bulletin dialog box is to report a process. It should never be used in the role of error reporting or confirmation gathering.

> ***Design tip:*** Never use sustaining dialogs as error messages or confirmations.

Stopping the proceedings

We have established that error messages must stop the proceedings with a modal dialog box. Most user interface designers—being programmers—imagine that their error message boxes are alerting the user to serious problems.

This is a widespread misconception. Most error message boxes are informing the user of the inability of the program to work flexibly. You can see an example of this back in Chapter 13 in Figure 13-1. Most error message boxes seem to the user like an admission of real stupidity on the program's part. In other words, to most users, error message boxes are seen not just as the program stopping the proceedings but, in clear violation of the axiom presented in Chapter 13, as *stopping the proceedings with idiocy.* We can significantly improve the quality of our interfaces by eliminating error message boxes.

Design tip: Error message boxes stop the proceedings with idiocy.

Why we have so many error messages

The first computers were undersized, underpowered, expensive, and didn't lend themselves easily to software sensitivity. The operators of these machines were white-lab-coated scientists who were sympathetic to the needs of the CPU and weren't offended when handed an error message. They knew how hard the computer was working. They didn't mind getting a core dump, a bomb, an "Abort, Retry, Fail?" or the infamous "FU"* message. This is how the tradition of software treating people like CPUs began. Ever since the early days of computing, programmers have accepted that the proper way for software to interact with humans was to demand input and to barf when the human failed to achieve the same perfection level as the CPU.

I call this attitude **silicon sanctimony**. Examples of silicon sanctimony exist wherever software demands that the user do things its way instead of adapting to the needs of the human. Nowhere is it more prevalent, though, than in the omnipresence of error messages. Silicon sanctimony is a negative feedback loop, ignoring users when they do what the software wants, but squawking at the slightest deviation from what they expect.

Silicon sanctimony is a requirement for actions *within* software. Every good programmer knows that if module A hands invalid data to module B, module B should clearly and immediately reject the input with a suitable error indicator. Not doing this would be a great failure in the design of the interface between the modules. But human users are not modules of code. Not only should software not reject the input with an error message, but the software designer must reevaluate the entire concept of what "invalid data" is. When it

*File Unavailable.

comes from a human, the software must assume that the input is correct, simply because the human is more important than the code. Instead of software rejecting input, it must work harder to understand and reconcile confusing input. The program may know what the state of things is inside the computer, but only the user knows what the state of things is outside in the real world. Ultimately, the real world is more relevant and important than what the computer thinks.

Humans have emotions and feelings: computers don't. When one chunk of code rejects the input of another, the sending code doesn't care; it doesn't scowl, get hurt or seek counseling. The processor doesn't even care if you flip the Big Red Switch.

On the other hand, humans—even phlegmatic programmers—have emotions, and they are raging out of control compared to anything happening in silicon. When you offer some information to a colleague and she says "Shut up, that's stupid," your feelings get hurt, your ego crushed. You search for mistaken meanings in what you said. You look in the mirror, checking your teeth for bits of spinach. You cancel your afternoon appointments so you can sulk in private and wonder what is wrong with your personality. All of these actions are part of human nature.

People hate error messages

When users see an error message box, it is like another person telling them "Fatal error, buddy. That input really sucked!" in a loud and condescending voice. Users hate this! Putting interaction like this in your program is extremely bad human interface design. (See Figure 28-1.) Despite this, most programmers just shrug their shoulders and put error message boxes in anyway. They don't know how else to create reliable software.

Many programmers and user interface designers labor under the misconception that people either like or need to be told when they are wrong. This assumption is false in several ways. The assumption that people like to know when they are wrong ignores human nature. Many people become very upset when they are informed of their mistakes and would rather not know that they did something wrong. They would be happier if the issue were never raised and the problem just got lost in the detritus of the everyday. Many people don't like to hear that they are wrong from anybody but themselves. Others are only willing to hear it from a spouse or close friend. Very few wish to hear about it from a

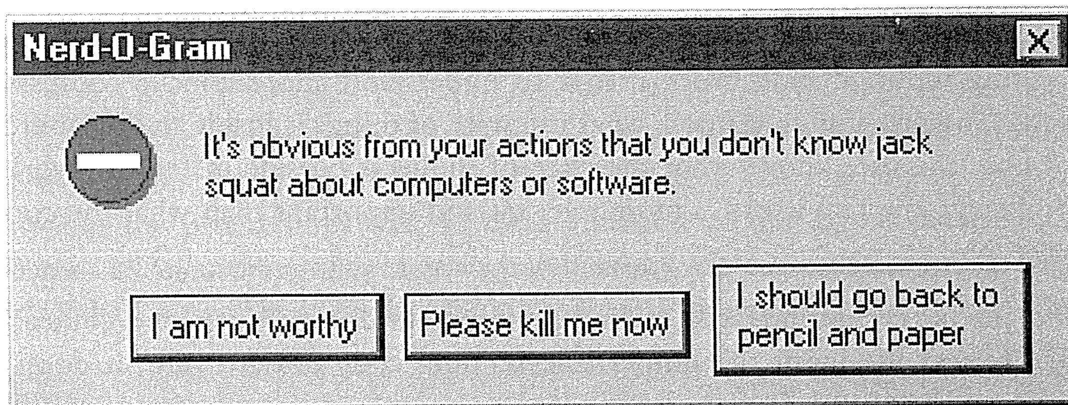


Figure 28-1

This is what all error messages feel like to users. They are not particularly sympathetic to the idiosyncrasies of the central processing unit, so it just feels like a rude rejection and personal condemnation. No matter how nicely your error messages are worded, this is how they will be interpreted.

machine. You may call it denial, but it is true, and users will blame the messenger before they blame themselves.

The assumption that users *need* to know when they are wrong is similarly false. How important is it for me to know that I requested an invalid typesize? Most programs can make a reasonable substitution. It may or may not be important, but why should the program assume that it is? We think that the program must be dependable because most programs don't offer sufficient visual cues for the user to supervise its actions. If we push our power lawnmower off the grass onto the gravel, we can *see* our error, not to mention hear the rattle of rocks and feel the sting of flung pebbles. If we choose a bad typesize, does the program show us what we have done? If it does, the user can see the results of his choice if he cares to see. If the program communicates clearly with the user, he can make up his own mind about his "need to know" when something isn't as he expected.

The assumption that it is good to tell users of their transgressions is one of the silliest canards to permeate the world of software design. It is indicative of just how socially inept most programmers are that they find such an assumption easy to swallow. No marketing person would ever fall for it. Nobody likes to be told when he makes a mistake. We consider it very impolite to tell people when they have committed some social faux pas. Telling someone he has a bit of lettuce sticking to his teeth or that his fly is open is equally embarrassing for both

parties. Sensitive people look for ways to bring the problem to the attention of the victim without letting others notice. Yet programmers assume that a big, bold box in the middle of the screen that stops all of the action and emits a bold "beep" is the appropriate way to behave.

Whose mistake is it anyway?

Another method of eliminating error messages is for the program to assume, when it receives bad input, that maybe it doesn't understand it because the program, not the user, is ill-informed. Conventional wisdom says that error messages tell the user when he has made some mistake. Actually, most error bulletins report to the user when the *program* gets confused. Most programmers, in the grip of silicon sanctimony, perpetually think about users "making mistakes," and conceive of error messages as tools for correcting the user's actions. Users make far fewer substantive mistakes than imagined. Typical "errors" consist of the user inadvertently entering an out-of-bounds number, or entering a space where the computer doesn't allow it. When the user enters something unintelligible by the computer's standards, whose fault is it? Is it the user's fault for not knowing how to use the program properly, or is it the fault of the program for not making the choices and effects clearer?

Information that is entered in an unfamiliar sequence usually is considered an error by software, but people don't have difficulty with this concept. Humans know how to wait, to bide their time until the story is complete. Software usually jumps to the erroneous conclusion that out-of-sequence input means wrong input, and issues the evil error message box.

When, for example, the user creates an invoice for an invalid customer number, most programs reject the entry. They stop the proceedings with the idiocy that the user must make the customer number valid *right now*. Alternatively, the program could accept the transaction with the expectation that a valid customer number would eventually be entered. It could, for example, make a special notation to itself indicating what it lacks. The program would then watch to make sure the user entered the necessary information to make that customer number valid before the end of the month book-closing. In fact, this is the way most humans actually work. They don't usually enter "bad" codes. Usually, they just enter codes in a sequence that the software isn't prepared to accept.

Our programs make the assumption that a customer account must be established before an invoice debited to that account can be valid, but nowhere is

this carved in stone. Why can't software accept invoices independently of account information and merely assume that things will be explained to it in due course? If the human forgets to fully explain things to the computer by month's end, the program can dump irreconcilable transactions into a suspense account. The program doesn't have to bring the proceedings to a halt with an error message. After all, the transactions in the suspense account will almost certainly amount to only a tiny fraction of the total sales, so they will not be a significant factor in the business reporting cycle. If they are significant, though, the program will remember the transactions so they can be tracked down and fixed. This is the way it worked in manual systems, so why can't computerized systems do at least this much? Why stop the entire process just because something inconsequential is missing? Why ground the airplane because the bar is short one little bottle of Scotch?

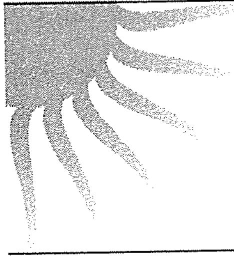
If the program were a human assistant and it staged a sit-down strike in the middle of the accounting department because we handed it an incomplete form, we'd be pretty upset. If we were the boss, we'd consider finding a replacement for this anal-retentive, petty, sanctimonious clerk. Just take the form, we'd say, and figure out the missing information yourself.

I have a name and address program that demands I enter an area code with a phone number even though I have already entered the person's address. It doesn't take a lot of intelligence to make a reasonable guess at the area code. Most of the people in my address book live nearby. If I enter a new name with an address in Menlo Park, the program can reliably assume that the area code is 415 by looking at the other 25 people in my database who also live in Menlo Park and have 415 as their area code. Sure, if I entered a new address for, say, Boise, Idaho, the program might be stumped. But how tough is it to keep a list of the 1,000 biggest cities in America along with their area codes? I can hear the protest already: "The program might be wrong. It can't be sure. Some cities have more than one area code. It can't make that assumption without approval of the user." Bullpucky!

If I asked my human assistant, Chris, to enter Joe's information, and I neglected to mention his area code, Chris would accept it anyway, expecting that the area code would arrive before its absence was critical. Alternatively, Chris could look the address up in a directory. Let's say that Joe lives in Los Angeles, so the directory is ambiguous: his area code could be either 213 or 310. If Chris rushed into my office in a panic shouting, "Stop what you're doing, Mr. Cooper! Joe's area code is ambiguous!" I'd be sorely tempted to fire poor Chris

and hire somebody with a greater-than-room-temperature IQ. Why should my software assistant be any different? Kim, Chris's replacement, just wrote "213/310?" into the area code field. The next time I need to call Joe, I'll have to determine which area code is correct, but in the meantime, life can go on.

Again, I can hear the squeals of protest: "But, but, but the area code field is only big enough for three digits! I can't fit '213/310?' into it!" Gee, that's too bad. You mean that rendering the user interface of your program in terms of the underlying implementation model—a rigidly fixed field width—forces you to reject natural human behavior in favor of obnoxious, computer-like, inflexibility supplemented with demeaning error messages? Not to put too fine a point on this, but error message boxes come from a failure of the program to behave reasonably, not from any failure of the user.



User interface is not just skin deep

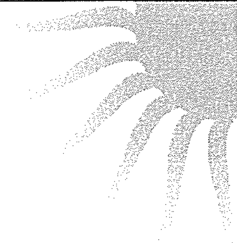
The previous example illustrates another important observation about user interface design. It is *not* only skin deep. The user just happens to be the last person in a long chain of hard-working, deadline-facing professionals. Problems that aren't solved in the code are pushed through the system until they fall into the lap of the user. There are a variety of ways to handle the exceptional situations that arise in interaction with software—and a creative programmer can probably think of a half-dozen or so off the top of her head—but most programmers just don't do it. They are compromised by their schedule and their preferences, so they tend to envision the world in the terms of perfect CPU behavior rather than in the terms of imperfect human behavior.

Make errors impossible

Making it impossible for the user to make errors is the best way to eliminate error messages. By using bounded gizmos for all data entry, users are prevented from ever being able to enter bad numbers. Instead of forcing a user to key in his selection, present him with a list of possible selections from which to choose. Instead of making the user type in a state code, for example, let him

choose from a list of valid state codes or even from a picture of a map. In other words, make it impossible for the user to enter a bad state.

Make errors impossible



Another excellent way to eliminate error messages is to make the program smart enough that it eliminates the need to ask the user questions. Many error messages say things like "Invalid input. User must type xxxx." Why can't the program, if it knows what the user must type, just enter xxxx by itself and save the user the tongue lashing? Instead of demanding that the user find a file on disk, introducing the chance that the user will select the wrong file, have the program remember which files it has accessed in the past and allow a selection from that list. Another example is designing a system that gets the date from the internal clock instead of asking for input from the user.

Undoubtedly, all of these solutions will cause more work for programmers. This doesn't bother me a bit. I don't want programmers to have to work harder just for the sake of working harder, but given the choice between programmers working harder and users working harder, I'll put the programmers to work in an instant. It is the programmer's job to satisfy the user and not vice versa. If the programmer thinks of the user as just another input device, it is easy to forget the proper pecking order in the world of software design.

Users of computers aren't sympathetic to the difficulties faced by programmers. They don't see the technical rationale behind the rejection in an error message box. All they see is the unwillingness of the program to deal with things in a human way. They see all error messages as some variant of the one shown in Figure 28-2. This is how most users perceive error message dialog boxes. They see them as Kafka-esque interrogations with each successive choice leading to a yet blacker pit of retribution and regret.

One of the big problems with error messages is that they are usually post facto reports of failure. They say "Bad things just happened and all you can do is acknowledge the catastrophe." Such reports are not helpful. And these dialog boxes always come with an OK button, requiring the user to collaborate in the mayhem. These error message boxes remind me of the scene in old war movies

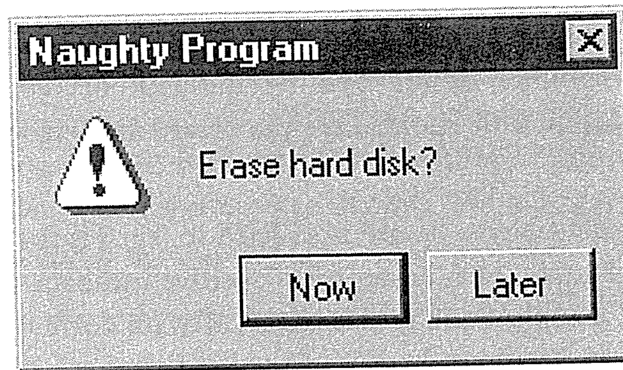


Figure 28-2

This is how most users perceive error message dialog boxes. They see them as Kafka-esque interrogations with each successive choice leading to a yet blacker pit of retribution and regret.

where an ill-fated soldier steps on a landmine while advancing across the rice paddy. He and his buddies clearly hear the click of the mine's triggering mechanism and the realization comes over the soldier that although he's safe now, as soon as he removes his foot from the mine, it will explode, taking some large and useful part of his body with it. Users get this feeling when they see most error message boxes, and they wish they were thousands of miles away, back in the real world.

Positive feedback

Humans respond more favorably to positive feedback than to the negative feedback of error messages. Instead of issuing a corrective message when things are wrong, your program could issue support messages when things are right, so the absence of a message would indicate a problem. Whenever the software can adapt to the input and accept it, the software replies with some indication of success. If the software cannot make sense of the input, it makes no reply at all—silence indicates failure. Just like Mom taught you when you were a tot: "If you can't say anything nice, don't say anything at all." Good advice, Mom, for software interaction with users.

The interaction I have with a hammer is illustrative of healthy interaction between a human user and a tool. When I use a hammer incorrectly, it doesn't give me an error message. It doesn't attempt to correct my behavior. It doesn't point out my failings as a carpenter. It just doesn't put nails in very well. The hammer rewards good use with good results and rewards poor use

with bad results. The simplicity, appropriateness and human scale of the interaction between human and hammer are proven by the lack of professional societies devoted to hammer design and by the lack of Opinion columns in carpentry magazines on how toolmakers can create more harmonious relations with hammer users.

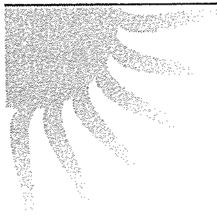
One of the big reasons why software is so hard to learn is that it so rarely gives positive feedback. People learn better from positive feedback than they do from negative feedback. People want to use their software correctly and effectively, and they are motivated to learn how to make the software work for them. They don't need to be slapped on the wrist when they fail. They do need to be rewarded, or at least acknowledged, when they succeed. They will feel better about themselves if they get approval, and that good feeling will be reflected back on the product.

Advocates of negative feedback can cite numerous examples of its effectiveness in guiding people's behavior. This evidence is true, but almost universally, the context of effective punitive feedback is getting people to refrain from doing things they want to do but shouldn't—things like not driving over 55 mph, not cheating on their spouses and not chiseling on their income taxes. But when it comes to doing what people *want* to do, positive feedback is best. Imagine a hired ski instructor who yells at you or a restaurant host who loudly announces to other patrons that your credit card was rejected.

Keep in mind that we are talking about the drawbacks of negative feedback from a computer. Negative feedback by another person, although unpleasant, can be justified in certain circumstances. One can say that the drill sergeant is at least training you how to save your life in combat, and the imperious professor is at least preparing you for the vicissitudes of the real world. But to be given negative feedback by software—any software—is an insult. The drill sergeant and professor are at least human and have bona fide experience and merit. The program is doo-doo, pond scum. It is less than zero. To be told by software that you have failed is humiliating and degrading. Users, quite justifiably, hate to be humiliated and degraded.

Users get humiliated when software tells them they failed

There is nothing that takes place inside a computer that is so important that it can justify humiliating or degrading a human user. Nothing. I don't care how important you think your precious database integrity is, it isn't worth insulting a person if you have effectiveness as your goal. If data integrity is a big deal for you, you need to work on methods of maintaining it without pissing off the user. There are plenty of good ways to do this. We only resort to the negative feedback ways of silicon sanctimony out of habit.



No crisis inside a computer is worth humiliating a human

So much effort is put into protecting the poor, fragile computer from mishandling. It's all right to protect the computer but not at the cost of bothering the user. Instead, we should put more effort into protecting the poor, fragile user from mishandling by the software.

Treat error messages like GOTOs

When I make the statement to groups of *users* that error message boxes should be eliminated, they generally agree, some with enthusiasm and some with mild reluctance. When I make that same statement to groups of programmers, they almost always protest vehemently. This reinforces my belief that the continued presence of error message boxes is due to programmers and not to users.

The debate reminds me of a similar one that began almost thirty years ago with the work of mathematician Edsgar Dijkstra, the inventor of structured programming. Dijkstra proved mathematically that the GOTO instruction could be eliminated from high-level programming languages and the result would be code whose correctness could be proven. His assertion sparked a firestorm of controversy in the industry that raged for a decade. Programmers of the '60s and '70s matured in their craft in the days of assembler language where the ability to jump randomly and without trace to anywhere in the program was considered a fundamental right and a necessary tool to achieve adequate performance. The structured programming revolution eventually proved these Luddites to be wrong, and few contemporary programmers working in languages like C, Pascal and BASIC resort to GOTOs.

It is well known that programs filled with GOTOs are nightmarishly difficult to understand and maintain although it is generally acknowledged that an occasional, well-commented GOTO harms no one. So, although programmers still use GOTOs occasionally, they treat them as a last resort. They only code GOTOs when the equivalent, structured, GOTOless code would be significantly more complex and wasteful to write. They know GOTOs are wrong, and using them is only rarely justified by extreme circumstance. They double-check their decision with their peers. They carefully add comments to the code describing the circumstances that justified the failure. They feel guilty.

I want programmers to feel the same way about error message boxes as they do about GOTOs. I want them to feel guilty when they code an error message box. I want them to know that they have better methods at their disposal for handling the situation. I want them to realize that error messages are not a fundamental right or a necessary tool to achieve adequate performance.

All error message boxes can be eliminated. If you examine the situation from the point of view that the error message box must be eliminated and that everything else is subject to change in search of this objective, you will see the truth of this assertion. You will also be surprised by how little else needs to be changed in order to achieve it. In those rare cases where the rest of program must be altered too much, that is the time to compromise with the real world, and go ahead and use an error message box. But I want the community of programmers to understand that this compromise is an admission of failure on the programmer's part. That they resorted to a low blow, a cheap shot, a GOTO in their code.

Exceptions?

I used to make a single exception to my dictum about no error message boxes. I believed that failed or missing hardware was adequate justification for an error message. I have since reconsidered this exception and decided that it doesn't hold water. As our technological powers grow, the portability and flexibility of our computer hardware grows, too. Windows 95 establishes a new standard called Plug-and-Play that allows networks and peripherals to be connected to and disconnected from your computer without having to first power it down. This means that, with Windows 95, it is now normal for hardware to appear and disappear ad hoc. Printers, modems and file servers can come and go like the tides. With the development of wireless network connectors, our computers will be attaching and detaching from networks frequently, easily, and all in the

normal course of walking from meeting to meeting with your sub-notebook. Is it an error if you print a document, only to find that no printers are connected? Is it an error if the file you are editing normally resides on a drive that is no longer reachable? Is it an error if your communications modem is no longer plugged into the computer?

The deeper we wade into the Internet ocean, the more this conundrum of here-today-gone-tomorrow becomes commonplace. The Internet can easily be thought of as an infinite hard disk—one that is out of the control of any one person, company or system administrator. A valid pointer today can be meaningless tomorrow. Is this an error?

I think that none of these occurrences should be considered as errors. If I try to print something and there is no printer available, my program should just spool the print image to disk. The print manager program should quietly indicate when it reconnects to a printer while it has unprinted documents in its queue. This should be an aspect of normal, everyday computing. It is not an error. The same is true for files. If I open a file on the server and begin editing it, then wander out to a restaurant for lunch, taking my notebook with me, the program should see that the normal home of the file is no longer available and do something intelligent. It could use the built-in digital cellular phone to log onto the server remotely, or it could just save any changes I make locally, synchronizing with the version on the server when I return to the office from lunch. In any case, it is normal. It is not an error.

The most frequent cause of error messages is in responding to the user asking for some resource that is not available. This error can be eliminated by assuring that the program doesn't offer to the user things that might not be present. If the program offers picklists instead of text-edit fields, the user will not be able to enter the name of an unavailable file.

Yes, I'd like to see an error message on my screen if the printer catches on fire, but I'd also like to see one if my colleague down the hall has a heart attack. Error messages should be reserved for emergencies, real emergencies.

Do they work?

There is a final irony to error messages: *They don't prevent the user from making errors.* We imagine that the user is staying out of trouble because our trusty error messages keep them straight, but this is a delusion. What error messages really do is prevent the program from getting into trouble. In most software,

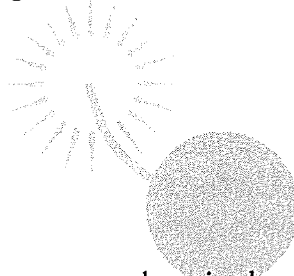
the error messages stand like sentries where the program is most sensitive, not where the user is most vulnerable, setting into concrete the idea that the program is more important than the user. Users get into plenty of trouble with our software, regardless of the quantity or quality of the error messages in it. All an error message can do is keep me from entering letters in a numeric field—it does nothing to protect me from entering the wrong numbers—which is a much more difficult design task.

What error message dialog boxes should look like

Now we will discuss some methods of improving the quality of error message boxes. In light of my attitude towards them, you can understand the reluctance I feel about doing this. Remember, use these only as a last resort. It is better just to take care of the problem behind the scenes and only bother the user with it if he asks.

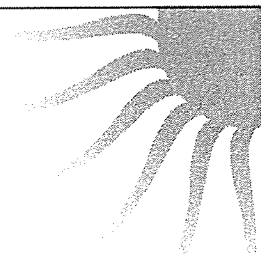
A well-formed error message box should be

- Polite
- Illuminating
- Helpful



Never forget that an error message box is the program reporting on *its* failure to do its job, and it is interrupting the user to do this. The error message box must be unfailingly polite. It must never even hint that the user caused this problem, because that is simply not true. The customer is always right.

The customer is always right



The user may indeed have entered some goofy data, but the program is in no position to argue and blame. It should do its best to deliver to the user what he

asked for, no matter how silly. Above all, the program must not, when the user finally discovers his silliness, say, in effect, "well, you did something really stupid and now you can't recover. Too bad." It is the program's responsibility to protect the user even when he takes inappropriate action. This may seem draconian, but it certainly isn't the user's responsibility to protect the computer from taking inappropriate action.

The error message box must illuminate the problem for the user. This means that it must give him the kind of information he needs to make an appropriate determination to solve the program's problem. It needs to make clear the scope of the problem; What the alternatives are; What the program will do as a default; What information was lost, if any. The program should treat this as a confession, telling the user everything.

It is wrong, though, for the program to just dump the problem on the user's lap and wipe its hands of the matter. It should directly offer to implement at least one suggested solution right there on the error message box. It should offer buttons that will take care of the problem in various ways. If a printer is missing, the message box should offer options for deferring the printout or selecting another printer. If the database is hopelessly trashed and useless, it should offer to rebuild it to a working state, including telling the user how long that process will take and what side effects it will cause.

Figure 28-3 shows an example of a reasonable error message. Notice that it is polite, illuminating and helpful. It doesn't even hint that the user's behavior is anything but impeccable.

The end of errors

Error message boxes validate the idea that the computer is the final arbiter of correctness and the user is there just to serve its digital majesty. This attitude influences both programmers and users. It tempts programmers to make bad judgments in design and to take shortcuts in implementation. These compromises necessitate the use of yet more error messages. Also, users are anesthetized by error messages so they cannot visualize the benefits of error-free computing.

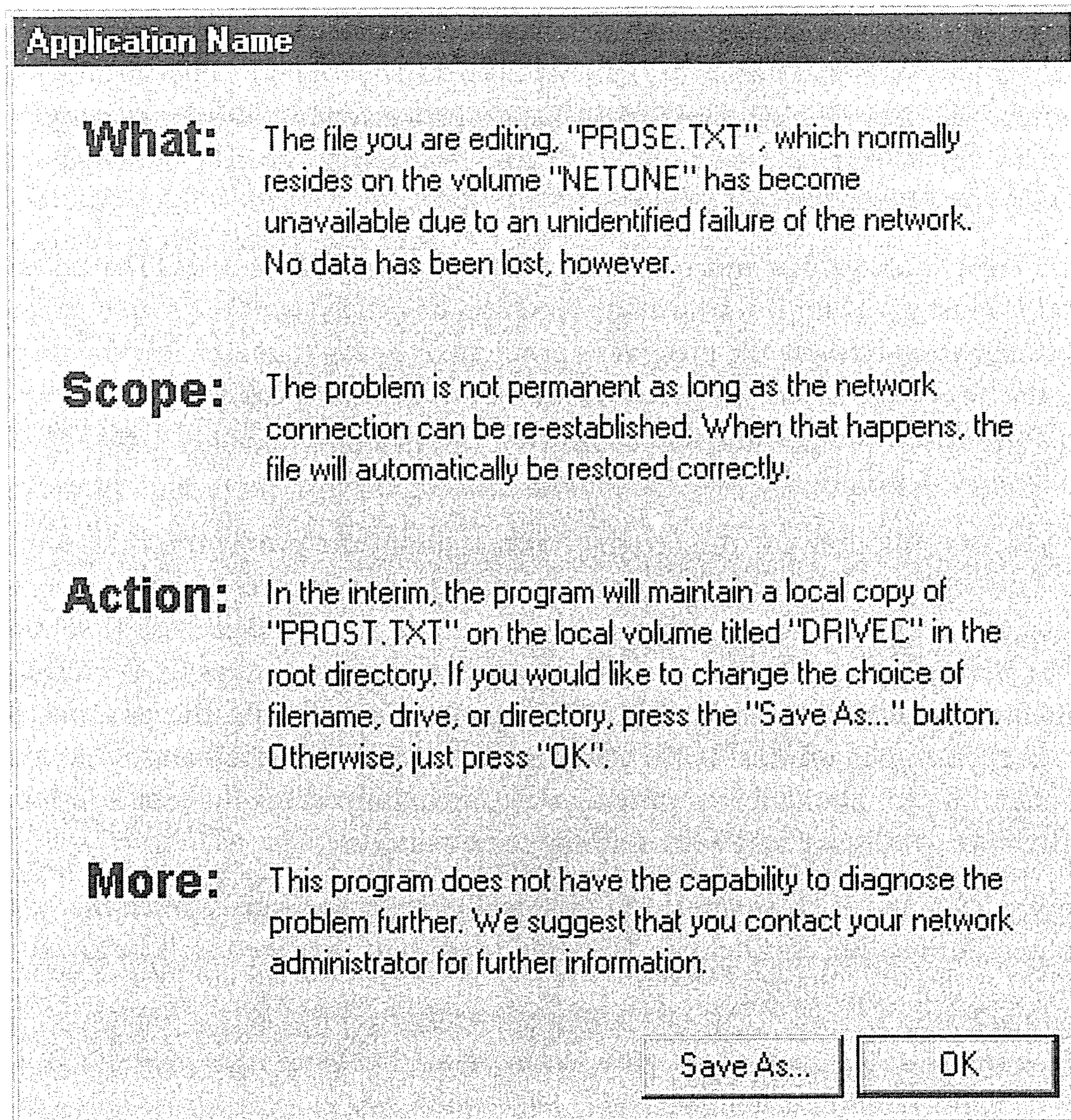
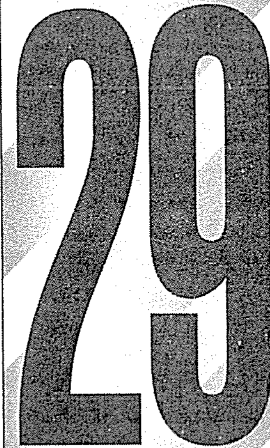


Figure 28-3

Just like there is rarely a good reason to ever use a GOTO in your code, there is rarely a good reason to issue an error message box. However, just as programmers occasionally compromise with one or two convenient GOTOs, they might occasionally issue an error message. In that case, your error message should look something like this one. It politely illuminates the problem for the user, offering him help in extricating the program from its dilemma. This error bulletin has four sections, labeled What, Scope, Action and More, that clearly help the user understand the options available and why he might choose each. The program is intelligent enough not to lose the file just because the volume became unavailable. The dialog offers an alternative action to the user by way of the Save As... button.

Managing Exceptions



Aside from errors, which we dealt with in the last chapter, there is a potpourri of exceptional user interface artifacts that we must examine. These include message and confirmation dialog boxes as well as the structure of many interactions and the underlying assumptions about them. To begin with, we will look at more ways in which poorly designed programs stop the proceedings with idiocy.

Alerts

There is another category of conditions that I call **exceptions**. Like errors, they stop the proceedings with idiocy, but they are not reporting malfunctions. Exceptions pop up like weeds in most programs, and I would like to give them the same treatment I give to errors: the old heave ho. Exceptions come in two basic varieties, *alerts* and *confirmations*. An alert notifies the user of the program's action, while a confirmation also gives the user the authority to override that action.

When the program exercises authority that it feels uncomfortable with, it often takes steps to inform the user of its actions. This is called an **alert**. Alerts violate the axiom that a dialog box is another room and you should have a good reason to go there (see Chapter 7). Even if an alert is justified (ha!), why go into another room to do it? If the program took some indefensible action, it should confess to it in the same place where the action occurred and not in a separate dialog box.

Conceptually, a program should either have the courage of its convictions or it should not take action without the user's direct guidance. If the program, for example, saves the user's file to disk automatically, it should have the confidence to know that it is doing the right thing. It should provide a means for the user to find out what the program did, but it doesn't have to stop the proceedings with idiocy to do so. If the program really isn't sure that it should save the file, then it shouldn't save the file, but should leave that operation up to the user.

Conversely, if the user directs the program to do something—dragging a file to the trash can, for example—it doesn't need to stop the proceedings with idiocy to announce that the user just dragged a file to the trash can. The program should assure that there is adequate visual feedback of the action, and if the user has actually made the gesture in error, the program should silently offer him a robust undo facility so he can backtrack.

The rationale for alerts is that they inform the user. I'm a real fan of informing the user, but not at the expense of a smooth and flowing interaction. I get a lot of high-quality information from my watch, but it doesn't need to tap me on the shoulder and interrupt me every hour to keep me informed of the time.

The alert shown in Figure 29-1 is a classic example of how alerts throw rocks at the user's feet. The Find dialog (the one at the bottom) already forces the user to press CANCEL when the search is completed, but the superimposed alert box makes it a brace of flow-breaking buttons: First the OK to the alert, then the CANCEL to the Find. If the information aspect of the alert were built into the main Find dialog, the user's burden would be reduced by half at no expense. That is good economy for user interface designers.

Alerts are so numerous because they are so easy to create. Most languages offer some form of message box facility in a single line of code. Conversely, building an animated status display into the face of a program might require a thousand or more lines of code. Programmers cannot be expected to make the right

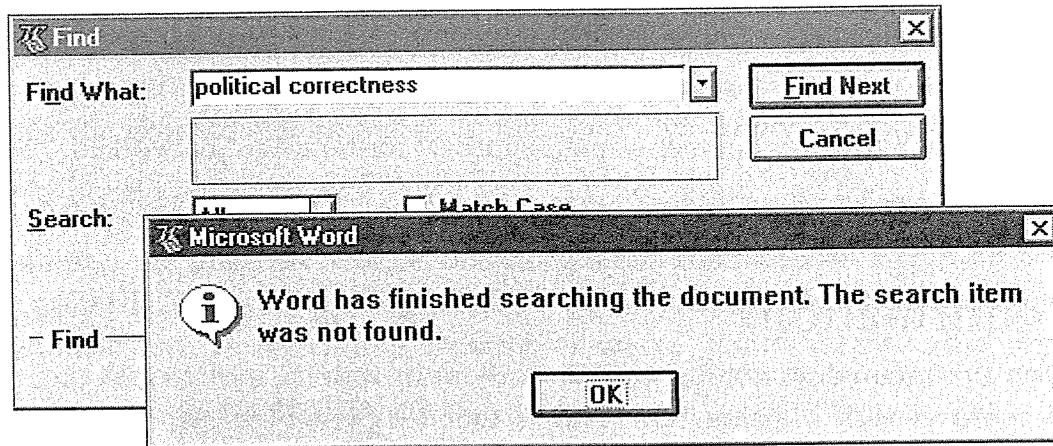


Figure 29-1

Here is a typical alert dialog box. Unnecessary, inappropriate, and it stops the proceedings with idiocy. The Find dialog in Word has finished searching the document. Is reporting that fact a different facility of Word? If not, why does it use a different dialog? It's like having to go into one dining room to use a fork and other one to use a spoon. The little "I" icon is a sure tip-off to smarmy, sanctimonious, clumsy interface design. Yes, software must constantly and effusively report its status to the user. But doing so with proceedings-stopping alert dialogs is wrong.

choice in this situation. They are too tied by conflict of interest, so designers must be sure to specify precisely where information is reported on the surface of an application, and they must follow up to be sure that the design wasn't compromised for the sake of code. Imagine if the contractor on a building site decided unilaterally not to add a bathroom because it was just too much trouble. There would be repercussions.

Announcing the obvious

Software needs to keep the user informed of its actions. It must have lights, meters or other gizmos built into its interface to make such status information available to the user, should he desire it. Putting up an alert to announce an unrequested action is bad. Putting up an alert to announce a *requested* action is pathological.

Software needs to be flexible and forgiving. It doesn't need to be fawning and obsequious. The dialog box shown back in Chapter 3 (Figure 3-2) is a classic example of an alert that should be put out of our misery. It announces that it added the entry to our phone book, immediately after we told it to add the entry to our phone book, which was mere milliseconds after we physically

added the entry to what appears to be our phone book. It stops the proceedings to announce the obvious. It wouldn't surprise me if they first popped up a dialog box to announce the dialog box that announced the addition.

It's as though the programmer wanted approval for how hard he worked: "See, dear, I've cleaned your room for you. Don't you love me?" If a person interacted with us like this, we'd suggest that he seek counseling.

Confirmations

When a program does not feel confident about its actions, it often asks the user for approval with a dialog box. This is called a **confirmation**, like the one shown in Figure 29-2. Sometimes the confirmation is offered so the user has the opportunity to second-guess one of his own actions. Sometimes the program feels that it is not competent to make a decision it faces and uses a confirmation to give the user the choice instead.

Confirmations always come from the program, and never from the user. This means that exceptions are a reflection of the implementation model, and are not representative of the user's goals. All confirmation dialog boxes can be eliminated just by changing the program's attitude. Look, for example, at the dialog in Figure 29-2.

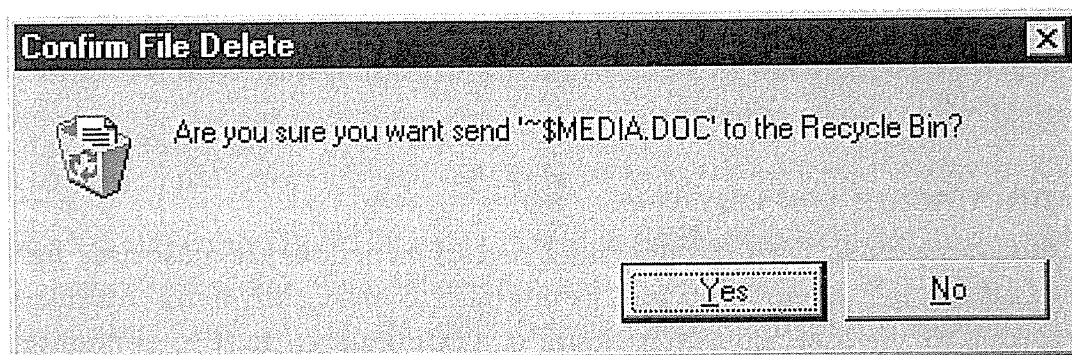


Figure 29-2

Every time I delete a file in Windows 95, I get this confirmation dialog box asking me if I'm sure. Yes, I'm sure. I'm always sure. And if I'm wrong, I expect you to be able to recover the file for me. Miracle of miracles, this version of Windows can finally live up to that expectation with its "Recycle Bin." So why does it still issue the confirmation message? When a confirmation box is issued routinely, users get used to approving it routinely. So when it eventually reports an impending disaster to the user, he goes ahead and approves it anyway, because it is routine. Confirmation boxes only work if they are unexpected. When users are performing new tasks, their senses will be alert to danger, so the only time they need unexpected confirmation boxes is when they are doing routine tasks. Deterministic algorithms can't do that. Do yourself and your users a favor, and never code another confirmation dialog box.

As I discussed in Part I, rendering the implementation model is a sure fire way to create a bad user interface. This means that the confirmation method of dealing with exceptions is wrong. Confirmations get written into software when the programmer arrives at an impasse in her coding. Typically, she realizes that she is about to take some bold action and feels that the user would want full control over this action. Sometimes the bold action is based on some condition the program detects, but more often it is based on a command the user issued. Typically, the confirmation will be erected after the user issues a command that is either irrecoverable or whose results might cause undue alarm.

In both of these circumstances, the programmer is passing the buck to the user, which is wrong. The user trusts the program to do its job, and the program should both do it and assure that it does it right. If it can't be absolutely sure it is doing it right, the program should at least be sure that it is able to backtrack on request. In other words, the program should assure that startling gizmos are clearly identified, and no actions should be irrecoverable.

Passing the buck to the user is also known as stopping the proceedings with idiocy. Yes, even if the program has found some exceptional condition, it is still idiocy from the user's point of view.

As a program's code grows during development, programmers detect numerous situations where they don't feel that they can resolve issues adequately. Programmers will unilaterally insert buck-passing code in these places, almost without noticing it. This tendency needs to be closely watched, because programmers have been known to insert dialog boxes into the code even after the user interface specification has been agreed upon. Programmers often don't consider confirmation dialogs to be part of the user interface, but they are.

Confirmations don't work

Here is a fact about confirmation messages: They only work when they are unexpected. That doesn't sound so remarkable until you examine it in context. If confirmations are offered in routine places, the user quickly becomes inured to them and routinely dismisses them without a glance. The dismissing of confirmations thus becomes as routine as the issuing of them. If—someday—a really unexpected and dangerous situation arises—one that should be brought to the user's attention, he will go ahead and dismiss the confirmation just because it has become routine. Like the fable of the boy who cried wolf, when

there is finally real danger, the confirmation box won't work because it cried too many times when there was no danger.

For confirmation dialog boxes to work, they must only appear when they are unexpected. Another way of saying this is that confirmations should only bother to appear when the user will almost definitely press the "NO" button, and they should never appear when the user is likely to press the "YES" button. Seen from this vantage, they look pretty pointless, don't they?

The confirmation dialog box shown in Figure 29-3 is a classic. It appears whenever I press the DELETE button. This means that it appears every time I want to say "YES," so I *always* push the "YES" button. If I ever want to say "NO," I probably won't even notice that there was a confirming dialog box at all. The irony of the confirmation dialog box in the figure is that I often have trouble with this dialog determining which styles I want to delete and which I want to keep. If the confirmation box appeared whenever I deleted a style that was currently in use, say, it would at least be a help because it would be less routine. But why not just put an icon next to the names of styles that are in use instead and dispense with the confirmation? It gives me a better view of what is happening so I can make a more informed decision about what to delete. Also, if the DELETE button were separated from the OK and CANCEL buttons, the chance of an inadvertent button press would be dramatically reduced.

How to eliminate confirmations

There are three axioms that tell us how to eliminate confirmation dialog boxes. The best way is to obey the simple dictum: do, don't ask. When you design your software, go ahead and give it the force of its convictions. Make sure that if it is going to do something, that it has the guts to go ahead and do it without whining and mewling about it. Users will respect its brevity and its confidence.

Do, don't ask

Of course, if the program confidently does something that the user doesn't like, it must have the ability to reverse the operation. Every aspect of the program's

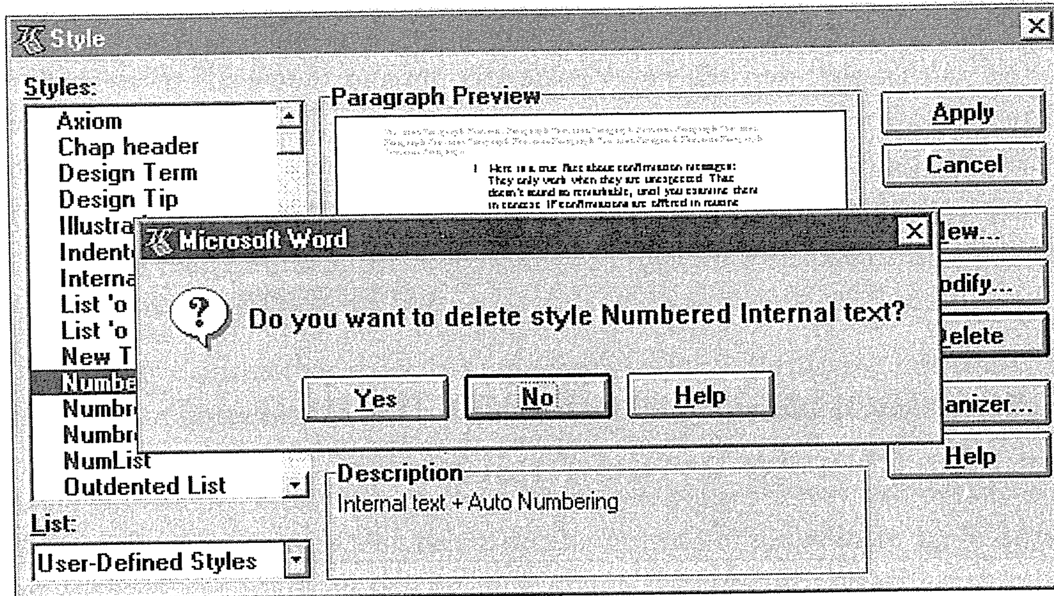


Figure 29-3

If you press the DELETE button in the Style dialog box in Word, you get this typical confirmation box. I always press YES. I never press NO. I wish, oh, how I wish that I could make this dialog go away forever. In the Style dialog, I occasionally inadvertently delete a style I really wanted to keep. This confirmation, however, doesn't help me prevent that. If its appearance were based on some criteria other than merely asking for a deletion, there is some faint chance that it would be useful. As it is, it merely irritates me. Tell me you won't ever create one of these, please?

action must be undoable. Instead of asking in advance with a confirmation dialog box though, let the user issue the stop-and-undo command on those rare occasions when the program's actions were out of turn.

Most situations that we currently consider unprotectable by undo can actually be protected fairly well. Deleting or overwriting a file is a good example. The file can be moved to a suspense directory where it is kept for a month or so before it is physically deleted. Actually, the Recycle Bin in Windows 95 uses this strategy, except for the part about automatically erasing them after a month: the user has to manually take out the garbage.

Make everything reversible

Even better than acting in haste and forcing the user to rescue the program with undo, you can make sure that the program offers the user adequate information so that the user will never issue a command that leads to an inappropriate action (or never omits a necessary command). The program should use sufficiently rich visual feedback so that the user is constantly kept informed, the same way the instruments on dashboards keep us informed of the state of our cars.

Directly offer enough information for the user to avoid mistakes

Occasionally, there arises a situation that really can't be protected by undo. I can't think of any right now, and you probably can't either, but we all know programmers who can. Is this a legitimate case for a confirmation dialog box? No. The program can't offer sufficient protection to the user, so it demands that the user waive his right to protection instead. A better approach is to provide him with protection the way we give him protection on the freeway: with consistent and clear markings. We can build really good quality, but modeless, warnings right into the interface. Isolated, brightly colored gizmos next to listboxes that offer full disclosure about the data to be messed with are a good start.

Much more common than honestly irreversible actions are those actions that are easily reversible but still uselessly protected by routine confirmation boxes. The confirmation in Figure 29-2 is an excellent specimen of this species. There is no reason whatsoever to ask for confirmation of a move to the Recycle Bin. The sole reason that the Recycle Bin exists is to implement an undo facility for deleted files. This goes beyond belt and suspenders. This confirmation box stops the proceedings with idiocy. It is more like belt, suspenders and handcuffs.

Who are we protecting, anyway?

Let's face it, most programs don't work all that hard to protect the user. However, they do work hard to protect themselves. Programs are tender, brittle souls, and a single bit in the wrong place can crash a big program.

Programmers—quite understandably—are protective of their creations. Error messages are the outward symptoms of this protection, but it is the software design imperative that shapes the program in such a way that it generates these symptoms.

This design imperative is characterized by the goal of never letting tainted, unclean data get into the software. The programmer erects barriers in the user interface so that bad data can never enter the system. This pure internal state is commonly called **data integrity**.

Data integrity posits that there is a world of chaotic information out there, and before any of it gets inside the computer it must be filtered and cleaned up. The software must have an outer barrier, a thin crust of protection, like sentries posted on the perimeter of a military base (see Figure 29-4). All data is made valid at its point of entry. Anything on the outside is assumed to be bad, or at least suspect, but anything that has penetrated the crust can be assumed to have satisfied the rigorous vetting of the barrier's best efforts. Once it has been allowed inside, it is assumed to be valid. The advantage is that once inside the database, the code doesn't have to bother with successive, repetitive checks on the validity or appropriateness of the data.

There is a completely different approach possible to protect sensitive software. Instead of keeping bad data out of the system, the programmer must make the system immune to inconsistencies and gaps in the information. This method involves writing much smarter, more sophisticated code that can robustly handle all permutations of data. I call this **data immunity**.

Programmers have traditionally used data integrity and have spurned the idea of data immunity, generally because it takes more complex code. Programmers are reluctant to accept the need to write more complex code. After all, they have deadlines, too.

Now, before you get too steamed and complain that we can't just let garbage into our systems, let me make it clear that that is not my intention.

Data integrity is a good concept on paper, but it has some severe failings in the real world. Mainly, it dumps the burden of entering correct data in the user's lap, and it demands that he do this at entry time, rather than when—and if—the correct data is actually needed. Data immunity doesn't tolerate bad data when correct data is needed. It does, though, tolerate bad data in the system when its "badness" doesn't really matter.

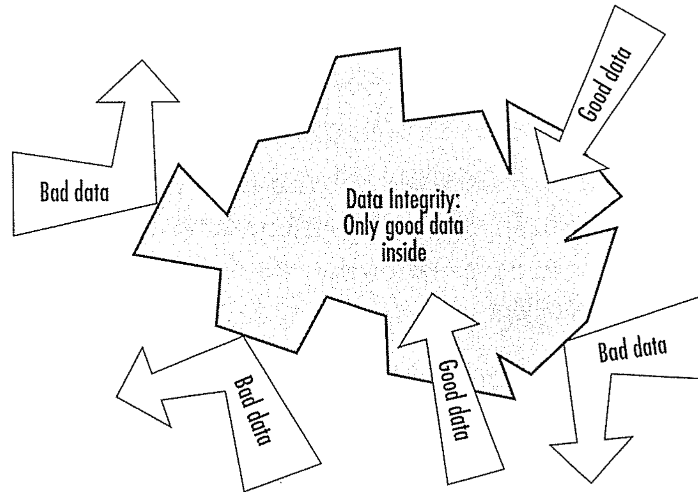


Figure 29-4

Underneath the rhetoric of data integrity—that there is an objective imperative of protecting the user and computer with sanctified data—there is a disturbing subtext. That subtext says that humans are ill-intentioned screwups; that users will, given the chance, enter the most bizarre garbage possible in a deliberate attempt to bring the system to its knees. This is not true. I'm certainly aware that users can, given the chance, enter garbage, but that is a far cry from saying that they do it intentionally. Users are very sensitive to subtext, though, and they will know that the program doesn't trust them. Data integrity not only hampers the system from serving the user for the dubious benefit of easing the programmer's burden, but it also offends the user with its accompanying attitude. It's another case of the user having to adapt to the needs of the computer, instead of vice versa. The philosophy of data integrity is based on scarcity thinking; there aren't enough precious computing resources to go around, so we must protect them from chaotic, bad data. That just isn't true anymore. It's time we start devoting some of our excess capacity to helping protect the user from chaotic, bad user interfaces.

Data integrity helps programmers, not users

Data integrity is a straitjacket placed on software design by programmers and computers and not by users. The user doesn't know or care about how much work the programmer must do to make things work correctly. The user is not concerned with the difficulties the programmer might have keeping the program from blowing up if it finds an alphabetic character in a numeric field. Yes, the user cares about having the program function reliably and about having it yield good results, but that doesn't mean he necessarily wants to have to do the scut work of correcting the details. He also wants his lawn trimmed, but it doesn't follow that he must be the one to personally wield the lawnmower.

Data integrity demands that all data be vetted at the door, and that all outliers are detected and bounced back. Once in, the data is good. This is nothing more

than a performance hack. The only reason suspect information is bounced on entry is to make things easy for the programmer and for the computer. The user doesn't count.

An invoice database, for example, may be used to generate statistical sales reports for management in addition to printing invoices to mail to customers. The absence of valid postal codes in the customer records might well hamper the invoice mailing, but it won't have any effect on the sales report. Data integrity, however, demands that the product manager can't get her report until those postal codes are perfect. From a systems point of view, getting postal codes perfect once and for all is the most efficient method. From the product manager's point of view, though, the silly program is demanding irrelevancies with an obnoxious rigidity.

The database programmer will counter that it is too difficult for the program to have to deal with possible bad data at each step of the way—that it is more efficient for the program to eliminate the bad data at entry time and assume goodness from then on. This is true, but not relevant. Efficiency is a concept that applies to machinery and central processing units. It has little or no applicability to human beings, and our task as programmers and designers is to improve the user's lot. The efficiency demanded by the database programmer is a vestige of the scarcity thinking that infects our entire generation of programmers. Today, we have plenty of computing power available to protect against bad data. But programmers drag their heels, not wanting to do the hard work. It is easier for programmers to invoke data integrity than it is for them to take the necessary effort to implement data immunity.

In particular, programmers who work with databases generally consider the integrity of their databases to be the primary, overriding concern of their work. The user's concerns rarely penetrate this deeply into the system end of the program code. Programmers may never admit it, but their decisions all seem to go in favor of data integrity at the expense of user considerations. Their sympathies are with the database rather than with the user. Users and user interface designers are told flatly "we *must* maintain the integrity of the database" with the same intensity as if the database were sustaining the life-support system on a spacecraft.

Data integrity is so widely accepted as good software design that its hegemony is rarely even questioned. Pretty much all of the art and science of database administration, management and programming is based on the assumption that

data integrity is reliably maintained. Programmers who write user-centered software that happens to rely on databases also unconsciously inculcate the data integrity principle into their work.

The database, whether residing on an aging mainframe or on an au courant client/server platform, makes louder, more immediate and more strident demands than users can, and so most applications are more sensitive to the needs of the server program than they are to the user. As a pragmatic developer of software, I know it is vital to address the needs of the platform, but I also know that someday a vendor will figure out how to keep the software happy while simultaneously giving users new and higher levels of interactive satisfaction. This vendor will cut through the marketplace like a scythe. It can be you, or it can be your competitor, but it is only a matter of time. As a designer, my allegiance is to the future, not to the database.

Data immunity

To implement data immunity, our programs must be trained to look before they leap, and they must be trained to ask for help.

Most software blindly performs arithmetic on numbers without actually examining them first. The program assumes that a number field must contain a number—data integrity tells it so. If the user entered the word "nine" instead of the number "9," the program would croak, but a human reading the form wouldn't even blink. If the program simply looked at the data before it acted, it would see that a simple math function won't do the trick.

Here's where looking for help comes in. Wait! I know what you are thinking: put up a message box asking the user. That is precisely the wrong thing to do. We must train our programs to believe that the user will enter what he means to enter, and if the user wants to correct things, he will without our paranoid insistence. But the program can look elsewhere in the computer for assistance. Is there a module that knows how to make numeric sense of alphabetic text? Is there a history of corrections that might shed some light on the user's intent?

If all else fails, the program must add annotations to the data so that when—and if—the user comes to examine the problem, he finds accurate and complete notes that describe what happened and what steps the program took.

Yes, if users enter "asdf" instead of "9.38," the program won't be able to arrive at satisfactory results. But stopping the program to resolve this *right now* is not

"satisfactory" process, either; the entry process is just as important as the end report. If the user interface is designed correctly, the program issues some visual feedback when the user enters "asdf," so the likelihood of the user entering hundreds of bad records is very low. Generally, users only act stupidly when programs treat them stupidly.

Most often, the incorrect data that the user enters is still reasonable for the situation. If the program expects a two-letter state code, the user may enter a "TY" by accident. However, that same user enters the city as "Louisville" and it doesn't take a lot of intelligence to figure out the problem. Missing postal codes can be solved by a relatively simple and small program that won't tax our modern, powerful computers. In the rare cases where the postal code locator program fails, most humans would have failed, too.

Data integrity is a privilege, not a right

From a computer's point of view, it doesn't make any difference whether garbage got into the system intentionally or not. This has been used as a justification for the autocracy of data integrity. However, from the human user's point of view, the difference between intentionally entering a bad postal code and unintentionally entering it is very great indeed. Back when computers cost millions and were slow and temperamental, the user's feelings could be justifiably snubbed for practical considerations. Those days are gone forever, hustled to the door and booted into the street by the information revolution.

This is how I justify demoting data integrity from its position as a guiding principle of software design. When our software shakes down data at the point of entry, when it strip-searches the user to assure that he isn't carrying any contraband into the high-security depths of the computer, it makes a very clear statement: that the user is insignificant and that the program is god-like; that the user works for the good of the program and not vice versa. This is not the impression that we want to give. We want the user to feel in charge; to feel that the program works for him; that the program is doing the work, while the user makes the decisions.

Data integrity helps reduce the burden on the programmer, while saying nothing about what it does for the user. Programmers who cut their teeth on mainframes with batch-processed COBOL applications (I did) learn the concept of data integrity early. Today, the gospel of data integrity is being taught to a new generation of programmers using Visual Basic to access department-level SQL

databases. The computational landscape is completely different than it was twenty years ago. While the power of the host computers has increased ten-thousand-fold, not much else has. The quantities of data typically handled haven't changed more than an order of magnitude, and the humans who use them are the same. Yet we still put data integrity at the top of our priority list, even though the demand for it is vestigial.

Audible feedback

In mass-production data-entry environments, professional data-entry clerks—touch-typists all—sit for hours in front of video screens and enter data. These users may well be examining source documents and typing by touch instead of looking at the screen. If they enter something erroneous, they need to be informed of it both audibly and visually. The clerk can then use his sense of hearing to monitor the success of his inputs while he keeps his eyes on the document.

Here, I'm absolutely *not* talking about the beep that accompanies an error message box. In fact, I'm not talking about a beep at all. When I talk about audible feedback as a problem indicator, I'm talking about silence.

With the exception of computer software, almost every object and system offers sound to indicate success rather than failure. When we close the door, we know that it is latched when we hear the click, but silence tells us that it is not yet secure. When we are talking with a group of people and they say "Yes" or "Uh-huh," we know that we have gotten through to them. When they are silent, we know our arguments have slipped off the track somehow. When we turn the key in the ignition and get silence, we know we've got a problem. When we flip the switch on the copier and it stays coldly silent instead of humming loudly, we know that there is trouble. Even things we consider silent make noise: Turning on the stovetop returns a sibilant hiss of gas and a quietly gratifying "whoomp" as the pilot ignites the burner. Electric ranges are inherently less friendly and harder to use because they lack that sound—they have to have indicator lights to tell us of their status.

When success with our tools yields a sound, that is called **positive audible feedback**.

Our software tools are mostly silent; all we hear is the quiet click of the keyboard. Hey! That's positive audible feedback. Every time you press a key, you hear a faint but positive sound. Keyboard manufacturers could easily make

perfectly silent keyboards, but they don't because we depend on audible feedback to tell us how we are doing. The feedback doesn't have to be sophisticated—those clicks don't tell us much—but they must be consistent, because if we ever detect silence, we know that we have failed to press the key. The true value of positive audible feedback is that its absence is an extremely effective problem indicator.

The effectiveness of positive audible feedback comes from its human sensitivity. Nobody—no human, that is—likes to be told that they have failed. Error message boxes are negative feedback, telling the user that he has done something wrong. Ah, but silence assures that the user knows this without actually being told of the failure. It is remarkably effective, because the software doesn't have to insult the user to accomplish its ends.

Our software should give us constant small audible cues just like our keyboards. 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. If the program didn't understand the input, it would remain silent and the user would be immediately informed of the problem and be able to correct his input without embarrassment or ego-bruising. Whenever the user starts to drag an icon, the computer would issue a short "toot-toot," then an effervescent hiss as the object was dragged. When it was dragged over pliant areas, the hiss would rise a note in pitch. When the user finally released the mouse button, he would be rewarded with a nearly silent "Yeah!" from the speakers for a success, or frigid silence if the drop wasn't meaningful.

People frequently counter this argument by telling me how users don't like audible feedback, how they are offended by the sounds that computers make and how they don't like to have their computer beeping and booping at them. To this I say "Bunk!" People are conditioned by two things about computer sound:

- Computers have always accompanied error messages with noises.
- Computer noises have always been loud, monotonous and unpleasant alarms.

Emitting noise when something bad happens is called **negative audible feedback**.

On most systems, error message boxes are normally accompanied by loud, shrill, tinny little "beeps," and audible feedback has become strongly associated them. That beep is a public stigmata of the user's failure. It coldly announces to all within earshot that you have done something execrably stupid. It is such hateful silicon sanctimony that most software developers now have an unquestioned belief that sound is bad and should never again be considered as a part of interface design. Nothing could be further from the truth. It is just the negative feedback aspect that is bad, not the audible aspect.

Negative audible feedback has several things working against it. Because the negative feedback is issued at a time when a problem is discovered, it naturally takes on the characteristics of an alarm. Alarms are designed to be purposefully loud, discordant and disturbing. They are supposed to wake sound sleepers from their slumbers when their house is on fire and their lives are at stake. They are like insurance, because we all hope that they will never be heard. Unfortunately, users are constantly doing things that programs can't handle, so these actions have become part of the normal course of interaction. Alarms have no place in this normal relationship, the same way we don't expect our car alarms to go off whenever we accidentally change lanes without using our turn indicators. Perhaps the most damning aspect of negative audible feedback is the implication that success must be greeted with silence. Humans like to know when they are doing well. They *need* to know when they are doing poorly, but that doesn't mean that they like to hear about it. Negative feedback systems are guaranteed to be appreciated less than positive feedback systems.

Given the choice of no noise versus noise for negative feedback, people will choose the former. Given the choice of no noise versus unpleasant noises for positive feedback, people will choose either based on their personal situation and taste. Given the choice of no noise versus soft and pleasant noises for positive feedback, however, my experience tells me that people will almost universally choose the audio. We have never given our users a chance by putting high-quality, positive audible feedback in our programs, so it's no wonder that people associate sound with bad interfaces.

The audible feedback must be at the right volume for the situation. Most computers don't offer volume controls, so sound is usually either too loud or too soft. Windows 95 finally offers a standard volume control, so one obstacle to beneficial audible feedback has been overcome.

Using your powers for good

Many programmers believe that it is their duty to inform the user when he has made an error. It is certainly the program's duty to inform *other programs* when they make an error, but I don't believe that this rule should extend to users. The customer is always right, so the program must accept whatever the user tells it, regardless of what the program does or doesn't know. This is similar to the concept of data immunity because whatever the user enters is acceptable, regardless of how incorrect the program believes it to be.

This doesn't mean that the program can wipe its hands and say "all right, he doesn't want to be protected, so I'll just let him crash." Just because the program must act as though the user is always right, this doesn't mean that the user actually *is* always right. Humans are always making mistakes, and your users are no exception. I can guarantee you that they will screw up. It may not be your fault, but it's your responsibility. How are you going to fix it?

It's not your fault, but it's your responsibility

The program can erect warning signs—as long as they don't stop the proceedings with idiocy—but if the user chooses to do something suspect, the program can do nothing but accept the fact and work to protect the user from harm. Like a faithful guide, it must follow its master into the jungle, making sure to bring along an elephant gun and plenty of ammo.

The warning signs must use modeless techniques on the surface of the active window to inform the user of what he has done, much like the way the speedometer silently reports our speed violations. It is not reasonable, however, for the program to stop the proceedings with modal idiocy, just like it is not right for the speedometer to cut the gas when we edge above 65 miles per hour. Instead of an error message box, for example, edit fields can have little, graphic, simulated LEDs attached to them that change from green to red depending on how the program evaluates the current input.

Once the user has gone ahead and done something that the program is sure is wrong, there is only one way to protect him. If we edit his work without telling