When a novelist writes well, the craft of the writer becomes invisible, and the reader sees the story and characters with a clarity undisturbed by the technique of the writer. Likewise, when a program interacts well with a user, the interaction mechanics precipitate out, leaving the user face-to-face with his objectives, unaware of the intervening software. The poor writer is a visible writer, and a poor user interface designer looms with a visible presence in his software: eyes wild, hair rumpled and Jolt on his breath.
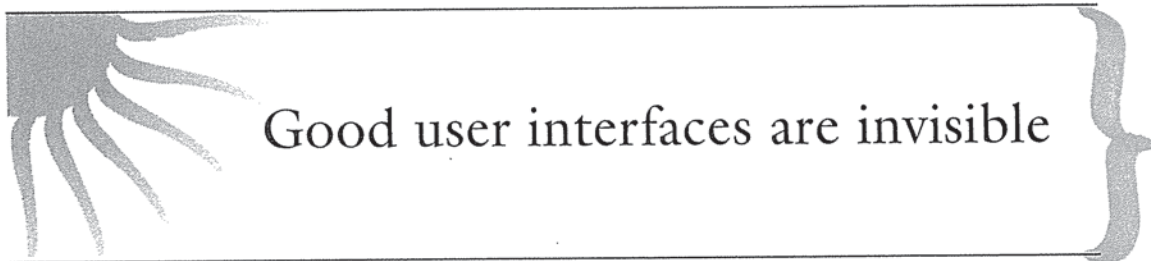
Good user interfaces are invisible

To a novelist, there is no such thing as a "good" sentence. There are no construction rules that guarantee transparent sentences. It all depends on what the protagonist is doing, or the effect the author wants to create. The writer knows to not insert an obscure word in a particularly quiet and sensitive passage, lest it sound like a sour note in a string quartet. The same goes for software. The software designer must train his ears to hear sour notes in the orchestration of software interaction.

When a program's communication with the user is well orchestrated, it becomes invisible.

Possibility versus probability

*[handwritten annotations: 1 IN 1,000,000   POSSIBILITY OF   PROBABILITY OF 999,999]*

There are many cases where interaction, usually in the form of a dialog box, slips into a user interface unnecessarily. A frequent source for such clinkers is when a program is faced with a choice. That's because programmers tend to resolve choices mathematically, and it carries over to their software design. To a mathematician, if a proposition is true 999,999 times out of a million and false one time, the proposition is false—that's the way math works. However, to the rest of us, the proposition is not only not false, it is overwhelmingly true. The proposition has a *possibility* of being false, but the *probability* of it being false is minuscule to the point of irrelevancy.

Mathematicians and programmers tend to view possibilities as being the same as probabilities. For example, a user has the choice of ending the program and saving his work or ending the program and throwing away the document he has

Work 6 hours. Throw out work — 50/50 chance
Possibility = Break Even
Probability — Low

136   PART III: THE BEHAVIOR

been working on for the last six hours. Mathematically, either of these choices is equally possible. Conversely, the probability of the user discarding his work is at least a thousand to one against, yet the typical program always includes a dialog box asking the user if he wants to save his changes, like the one shown in Figure 11-3.
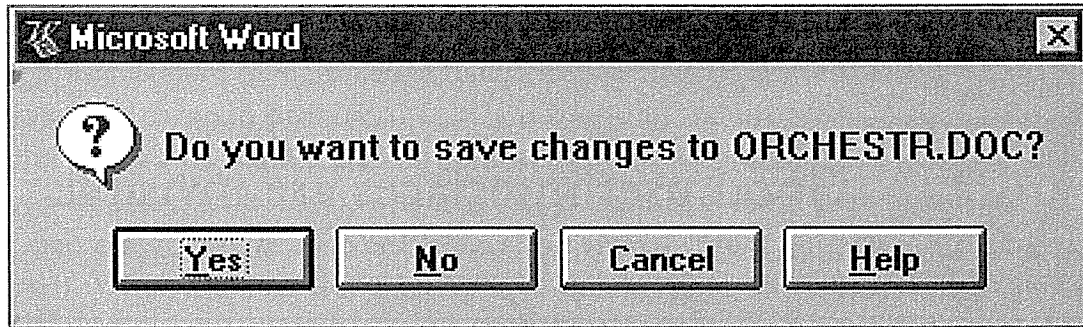
Figure 11-3

This is easily the silliest and most unnecessary dialog box in the world of GUI. Of course, I want to save my work! It is the normal course of events. Not saving it would be something out of the ordinary that should be handled by some dusty dialog box. This single dialog box does more to force the user into knowing and understanding the useless and confusing facts about RAM and disk storage than almost anything else in his entire interaction with his computer. This dialog box should never be used.

The dialog box in Figure 11-3 is completely inappropriate and should not exist. Yes, I want to save the changes to ORCHESTR.DOC. I wouldn't have bothered to name the file if I hadn't wanted to keep it. In the rare case where I change my mind and want to discard my changes and revert back to the original, the program should provide tools to do so. But these tools shouldn't be waved in my face every time I end a document. I edit a couple of hundred documents every month and I have to abandon my changes only about once a year.

Whoever designed this program confused probability with possibility, and they burden me with this irrelevancy every time I end the program. This is tantamount to my telling you not to pour your soup on your shirt every time you eat.

It is possible to argue that users have come to expect this behavior; that its absence would cause experienced users to fret that changes were being mistakenly discarded when the program ends. This rationale is like saying that a beaten dog expects to be beaten again, so we should beat it to make it happy. The time to make our programs better is now.

If a program offers a function, it ought to offer just the function and not all of its permutations. For example, when I ask to print something, I don't want a dialog box that allows me to configure the print function, I just want a simple printout of the current document. On the rare occasions when I want something special, like just a few pages, or seven copies, or sideways printing, then I should have to ask for the "Custom Print Dialog" or some such. If there is a possibility that the user might set the parameters of the function, that should be a secondary characteristic of the function itself, rather than intruding on the more probable act of invoking the function.

The print button on the Word toolbar offers immediate printing without a dialog box. This is perfect for many users, but for those with multiple printers or printers on a network, it may be offering too little information. The user may want to see which printer is selected before he either presses the button or summons the dialog to change it first. This is a good candidate for some simple modeless output placed on a toolbar or status bar.

Another good example of this confusion could be found in Microsoft Excel's older Version 4.0. When you select one or more cells and press the DELete key to clear the field, a small dialog box pops up asking what you want to delete.

The flexible little dialog box shown in Figure 11-4 conveniently allowed you the option of clearing the formats, formulas or notes from the selected cells. This dialog drove me crazy with its obtrusive uselessness. It is true that there are three types of deletion operations: format, formula and notes. However, it is also true that, although I delete formulas with great frequency, I have never in all of my spreadsheeting desired to delete formats or notes. So why does the program ignore this self-evident fact and insist on asking me through this mindless little dialog? Just because something is possible, doesn't mean that it is probable.

You *might* get hit by a bus, but you probably *will* drive safely to work this morning. Don't stay home out of fear of the killer bus. Don't let what might happen alter the way you treat what will happen. I say "Don't put might on will" to remind myself not to load up parts of a program that will get used with lots of stuff that might get used.

Programmers are judged by their ability to create software that handles the many possible but improbable conditions that crop up inside complex logical systems. This doesn't mean, however, that they should render that readiness to handle offbeat possibilities in the user interface. The presence of might-on-will
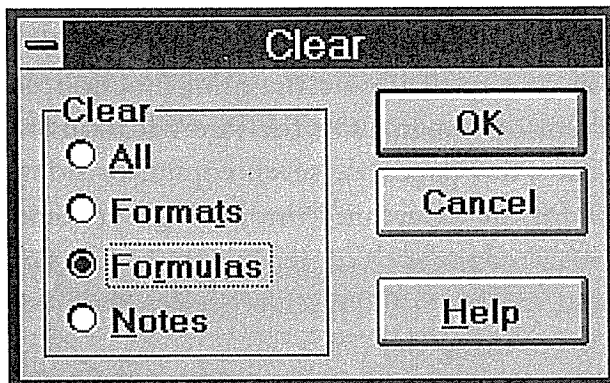
Figure 11-4

In Excel, Version 4.0, this dialog box popped up every time you pressed the DEL key. This is quite reasonable if you are a computer, but if you are a human, it means that you have to deal with the remote possibilities of deletion every time you try to do the high-probability clearing of the formula. Using Excel with this dialog was like listening to the symphony pause every time the conductor had to turn a page on the score. Thankfully, Microsoft obliterated this little gem in Excel 5.1.

Don't put might on will

is a dead giveaway for user interfaces designed by programmers. Dialogs, controls and options that are used a hundred times a day sit side-by-side with dialogs, controls and options that are used once a year or never. One of the most potent methods for better orchestrating your user interfaces is segregating the possible from the probable.

The remedy to this situation is to create user interfaces that conform to probabilities and not to possibilities. I can easily see Excel having an advanced delete command available from a menu item that gives me access to the dialog box we just condemned. I could use it for those exceedingly rare cases where I want to delete notes or formats instead of just the formula. The program could then leave the DELete key for quickly and unobtrusively doing the obvious: deleting the contents of the field: the formula. In this case, the user interface would become quieter and less obtrusive, I would notice it less, and it would be better without taking away from the power of the program.

Quantitative information

The way that a program represents information is another way that it can obtrude noisily into a user's consciousness. One area frequently abused is the representation of quantitative, or numeric, information. If an application needs to show the amount of free space on disk, it can do what the Microsoft Windows 3.x File Manager program does and come right out and give you the number of free bytes, as shown in Figure 11-5.
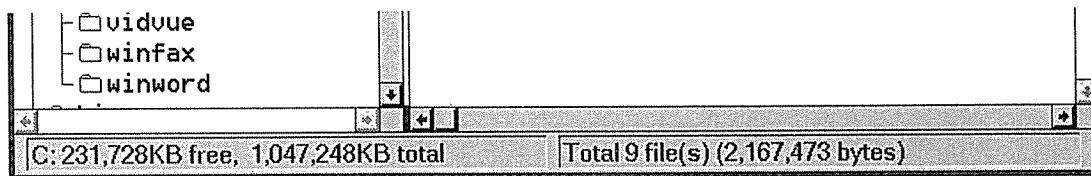

Figure 11-5

This is the bottom inch or so of the Windows 3.x File Manager program. It takes great pains to tell me how much of my disk is used and how much is free down to the billionth! Does this precision help me understand whether I need to clear out space on my disk? Certainly not. Furthermore, is a number the best way to indicate the disk's status to me? Wouldn't a graphical representation that showed the space usage in a proportional manner (like a gas gauge) be more meaningful? The way this information is rendered guarantees that my concentration will be broken if I need to know it.

In the lower left corner of the program, it tells me the number of free bytes and the total number of bytes on the disk. I find these numbers very hard to read, and extremely hard to interpret. With more than a thousand million bytes of disk storage, it ceases to be important to me just how many hundreds are left, yet the display rigorously shows me down to the kilobyte how many are used and how many are left. Even while the program is telling me the state of my disk with exaggerated precision, it is failing to communicate. What I really need to know is whether or not my disk is getting full, or whether I can add a new 20 MB program and still have sufficient breathing room. Instead, I find myself concentrating on those numbers like they were Egyptian hieroglyphics trying to make sense of them. It isn't easy because the numbers don't help me to visualize the problem.

Visual presentation expert Edward Tufte says a good numeric presentation should answer the question "compared to what?" Knowing that 231,728 KB are free on my hard disk is less useful than knowing that it is 22 percent of the disk's total capacity. Another Tufte dictum is "show the data" as opposed to telling about it. A small bar or pie chart showing the used and unused portions

in different colors would make it much easier to comprehend the scale and proportion of hard disk use. It would show me what 231,728 KB really means. This bar could easily be displayed where the numbers are currently shown. The numbers shouldn't go away, but they should be relegated to the status of captions on the data and not be the data itself. They should also be displayed with a more reasonable and consistent precision. The meaning of the information would be shown visually while the numbers would merely add support.

In Windows 95, Microsoft's right hand giveth while their left hand taketh away. The File Manager with the numbers (shown in Figure 11-5) is dead, replaced by the Explorer dialog box shown in Figure 11-6. This replacement is the properties dialog associated with a hard disk. The "Used space" is shown in blue, and the "Free space" is shown in magenta, making the pie chart an easy read. Now I can see at a glance the sad truth that "GRANFROMAGE" is packed to the gills.

Unfortunately, that nice pie chart isn't built into the Explorer's interface. Instead, I have to seek it out with a menu item. To see how full my disk is, I must first bring the program to a smoking halt, bring up a modal dialog box that, although it gives me the information I want, takes me away from the place where I want to know it. The Explorer is where I can see, copy, move and delete files, but it's not where I can see if things need to be deleted. That nice blue and magenta pie chart should have been built into the face of the Explorer. Besides, what if I didn't know how to find that nice pie chart dialog box? What warning would I have had that GRANFROMAGE was full?

Graphical input

Software frequently fails to present numerical information in a graphical way. Even rarer than this, though, is the ability of software to enable graphical input. A lot of software lets users enter numbers and then, on command, converts those numbers into a graph. Few products let the user enter a graph and, on command, convert that graph into a vector of numbers. By contrast, most modern word processors let you set tabs and indentations by dragging a marker on a ruler. The user can say, in effect, "Here is where I want the paragraph to start" and let the program calculate that it is precisely 1.347 inches in from the left margin instead of forcing the user to enter "1.347."
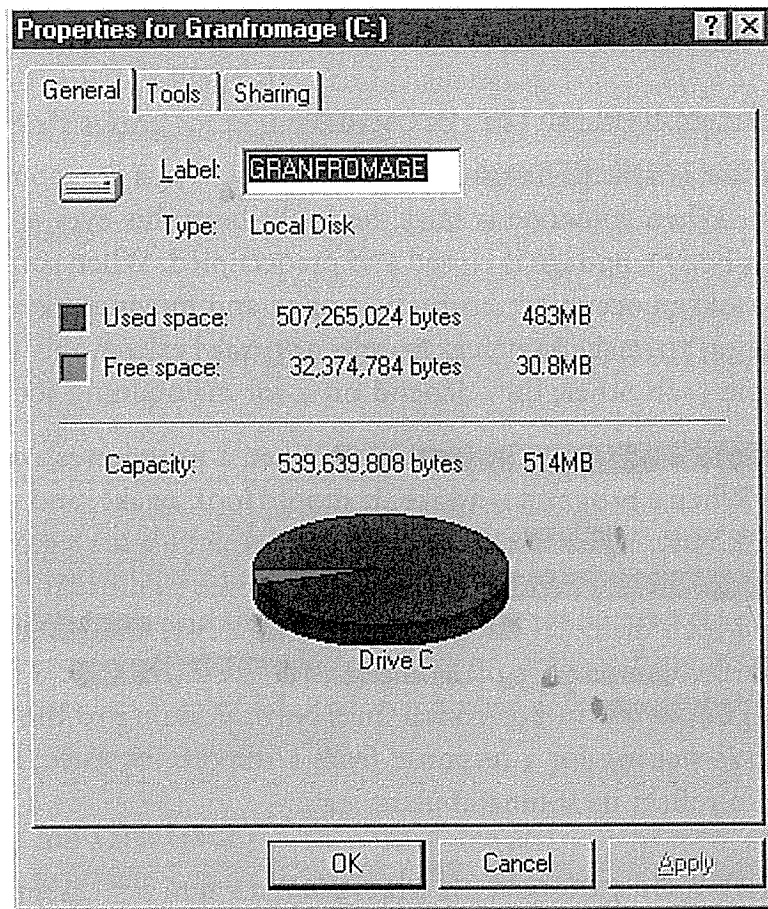
Figure 11-6

In Windows 95, Microsoft has replaced the electric chair with lethal injection. Instead of long, inscrutable numbers at the bottom of the File Manager, you can request a properties dialog box from the Explorer. The good news is that you can finally see how your disk is doing in a meaningful, graphic way with the pie chart. The bad news is that you have to stop what you're doing and open a dialog box to see fundamental information that should be readily available.

The new category of intelligent drawing programs like Shapeware's Visio are getting better at this. Each polygon that the user manipulates on screen is represented behind the scenes by a small spreadsheet, with a row for each point and a column each for the X and Y coordinates. Dragging a polygon's vertex on screen causes the values in the corresponding point in the spreadsheet, represented by the X and Y values, to change. The user can access the shape either graphically or through its spreadsheet representation.

This principle applies in a variety of situations. When items in a list need to be reordered, the user may want them ordered alphabetically, but he may also want them in order of personal preference; something no algorithm can offer.

The user should be able to drag the items into the desired order directly, without an algorithm interfering with this fundamental operation.

Reflect the status of the program

When someone is asleep, he usually looks asleep. When someone is awake, he looks awake. When someone is busy, he looks busy: his eyes are glued to his work and his body language is closed and preoccupied. When someone is unoccupied, he looks unoccupied: his body is open and moving, his eyes are questing and willing to make contact. People not only expect this kind of subtle feedback from each other, they depend on it for maintaining social order.

Our programs should work the same way. When a program is asleep, it should look asleep. When a program is awake, it should look awake, and when it's busy, it should look busy. When the computer is engaged in some significant internal action like formatting a diskette, we should see some significant external action, such as a one-inch diameter image of a diskette slowly changing from black to white. When the computer is off sending a fax, we should see a small image of the fax changing colors in horizontal lines corresponding to the bands sent. If the program is waiting for a response from a remote database, it should visually change to reflect its somnambulant state.

Lead, follow or get out of the way

I once used a program that generated tables. When I requested a table, what I got was a dialog box asking me lots of questions, but no table. I didn't ask for a big dialog. I asked for a table. Why didn't the program just give me a table? If it wasn't the table I wanted, I'd at least be motivated to see what I had to do to change it because the program clearly was willing to work with me. If it gave me a table first, then let me manipulate it to change its properties, it would be a much more effective solution.

Users—not power-users, but *normal* people—are very uncomfortable with explaining to a program what they want. Users would much rather see what the program *thinks* is right and then manipulate that to make it really right. In most cases, your program can make a fairly correct assumption based on past experience.

Don't misunderstand me. Just because I used the word "think" in conjunction with a program doesn't mean that the software should actually be intelligent and try to determine the right thing to do by reasoning. Instead, it should

simply do something that has a statistically good chance of being correct, then provide the user with powerful tools for shaping that first attempt, instead of merely giving the user a blank slate and challenging him to have at it. This way the program isn't asking for permission to act, but rather asking for forgiveness after the fact.
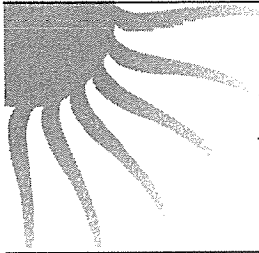
Ask forgiveness, not permission

To most people, a blank slate is a difficult starting point. It is so much easier to begin where someone has already left off. A user can easily fine-tune an approximation provided by the program into precisely what he desires with less risk of exposure and mental effort than he would have from drafting it from nothing. As we will discuss in Chapter 14, endowing your program with a good memory is the best way to accomplish this.

Reporting to the user

For programmers, it is important to know what is happening. This goes along with being able to control all of the details of the process. For users, it is disconcerting to know all of the details of what is happening. Many people are frightened to know that the database has been updated, for example. It is better for the program to just do what has to be done, issue reassuring clues when all is well, and not burden the user with the trivia of how it was accomplished.

Many programs are quick to keep users apprised of the program's progress even though the user has neither asked nor wants to know. Programs pop up dialog boxes telling us that connections have been made, that records have been posted, that users have logged on, that 274 transactions were recorded and other useless factoids. To software engineers, these messages are equivalent to the humming of the machinery, the babbling of the brook, the white noise of the waves crashing on the beach: they tell us that all is well. To the user, however, these reports can be like eerie lights beyond the horizon, like screams in the night, like unattended objects flying about the room.

As I said before, the program should make clear that it is working hard, but the detailed feedback can be offered in a more subtle way. In particular, reporting

information like this with a modal dialog box brings the interaction to a stop for no particular benefit. In Figure 11-6 we saw how Microsoft forces me to stop other things when I want to know how much space is left on my hard disk. The answer to this common question should never be relegated to a modal dialog box but should be constantly visible whenever the Explorer is running.

It is important that we not stop the proceedings to report normalcy. When some event has transpired that was supposed to have transpired, never report this fact with a dialog box. Save dialogs for events that are out of the normal course of events.

Design tip: Don't use dialogs to report normalcy.

By the same token, don't stop the proceedings and bother the user with problems that are not serious. If the program is having trouble getting through a busy signal, don't put up a dialog box to report it. Instead, build a status indicator into the program so the problem is clear to the interested user but is not obtrusive to the user who is busy elsewhere.

The key to orchestrating the user interaction is to take a goal-directed approach. You must ask yourself whether a particular interaction moves the user rapidly and directly to his goal. Contemporary programs are often reluctant to take any forward motion without the user directing it in advance. But users would rather see the program take some "good enough" first step and then adjust it to what is desired. This way, the program has moved the user closer to his goal.

Where were you on the night of the sixteenth?

Programs have a proclivity for shifting into what I call interrogation mode where they begin demanding answers from the user.

They enter interrogation mode for two reasons. Sometimes they seem to feel that it is their right to demand answers from the user. In this respect, they are totally mistaken. The program should instead offer *choices* to the user. There is a big difference between offering choices and demanding answers. The second reason is that many programs can't just do what they are told but must instead demand that you micro-manage their job.

These two reasons are often combined, resulting in obnoxious dialog boxes. You ask the program to perform a function, and it takes the opportunity to

demand that you explain to it in exacting detail precisely how you want it to do the function. The program should not use your request as an excuse to enter interrogation mode.

Dialog boxes have very little right to demand information from humans. They are merely digital scum, and they exist only at the sufferance of the user, and not vice versa. If you ask a program to perform a function, the program should perform that function and not begin to interrogate you about your precise demands. If you wanted to express your precise demands to the program, you would have requested the precision dialog instead.

For example, when I ask most programs to print a document, they respond by putting up a complex dialog box demanding that I specify how many copies to print, what the paper orientation is, what paper feeder to use, what margins to set, whether the output should be in monochrome or color, what scale to print it at, whether to use Postscript fonts or native fonts, whether to print the current page, the current selection or the entire document, and whether to print to a file and if so, what name should it get. Whew! All of those options sure are neat, but all I wanted was to print the document, and that is what I thought I asked for.

A much more reasonable design would be to have a command to PRINT and another command to CONFIGURE THE PRINT. The PRINT command would not issue any dialog but would just go ahead and print, either using previous settings or standard, vanilla settings. The CONFIGURE THE PRINT function would offer up all of those choices about paper and copies and fonts. It would also be very reasonable to be able to go directly from the configure dialog to printing.

There is a big difference between configuring and invoking a function. The former may include the latter, but the latter shouldn't include the former. In general, any user will want to invoke a command ten times for every one time he wants to configure it. It is better to make the user ask explicitly for the configurator one time in ten than it is to make the user reject the configurator *nine* times in ten.

The idea that "if you have available choices, they should be presented" is an expression of possibility thinking rather than the more user-centered probability thinking. Just because it is possible to fine-tune a function, it doesn't necessarily follow that there is a high probability that the user will want to.

Earlier versions of Word were notorious for this problem. I would finish editing a document, request PRINT from the menu, then get up from my chair, stretch, and walk down the hall for a cup of coffee. When I returned, expecting to find my printed document neatly stacked in the output tray of my printer, I would be frustrated to find no printout, just a mindless dialog idling peacefully in the middle of the screen waiting patiently till doomsday for me to confirm that, yes, I merely wanted one copy. I would pull my hair and press the redundant, mocking OK button, then stare disgustedly at the screen while the document slowly printed. The interrogation-mode dialog box wasn't usually needed; therefore it was not expected; therefore it was wrong. The latest version of Word still works this way, but the PRINT button on the toolbar immediately prints a single copy without presenting an intervening, unexpected dialog box—precisely what I expect.

Microsoft's printing solution is a reasonable rule of thumb. Put immediate access to functions on buttons in the toolbar and put access to function-configuration dialog boxes on menu items. The configuration dialogs are better pedagogic tools, while the buttons provide immediate action.

Sensible interaction

One of the essential parts of orchestrating a good user interface is interacting with the user in a sensible manner. This deceptively simple statement seems obvious, yet programs violate it constantly.

Many programs put dialog boxes up for no better reason than habit or programming ease. For example, in the File Manager program in Windows 3.x, if I ask to rename a file, the program puts up the dialog box shown in Figure 11-7.

The two outlined boxes are normal text-edit fields, but entering text in the top one has no effect on the file. My natural reaction to this dialog box is to directly edit the file name the program offers me in the top edit box. That is, if I want to change "WUIDABS.DOC" to "WUIDABS1.DOC," I would simply position the mouse between the s and the dot, click to get an edit caret and then enter the number 1 from the keyboard followed by an ENTER to close the dialog box. The tragic part of this story is that it is perfectly legal to take such a sequence of actions. However, if I do take those actions, nothing sensible happens! The dialog box closes, and the program continues blithely on its merry way with no acknowledgment of the fact that no change was made to the file's name.
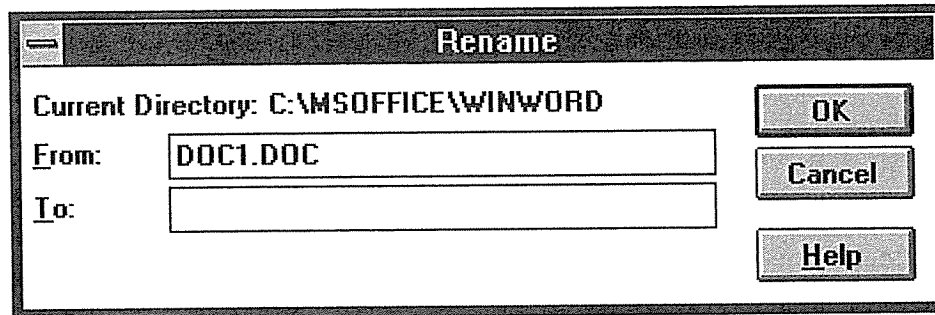
Rename

Current Directory: C:\MSOFFICE\WINWORD

OK

From: DOC1.DOC

Cancel

To:

Help

Figure 11-7

This is the File Rename dialog box from the Windows 3.x File Manager program. It violates the rule of orchestrating a good interaction with the user in numerous ways, including its very existence. Microsoft finally got rid of it in Windows 95. Are there dialogs like this one in your software?

To change the name of the file, I must enter the entire new file name in the second text edit box. I would have to enter "WUIDABS1.DOC" from the keyboard and then press ENTER to accept the change. The reaction from the program is identical to that from the first set of actions, except this time File Manager actually renames the file. Let's examine the several significant failings of the design of this dialog box.

The first field is an edit box that doesn't edit

If entering text into the topmost field is not a useful action, why did Microsoft make the field a text-entry field? The field should be for text output only, similar to the "Current Directory:" field just above it. Because the field allows text manipulation, it conveys the message to the user that text manipulation is a meaningful and effective action. If this is not true, the field should not offer such affordances. The program should not lie to the user.

I must confess that entering something into this field actually *does* have an effect. It is a nonsensical, undiscoverable and unwanted effect, but an effect nonetheless. My technical editor, Neil Rubenking, pointed it out to me, although even he can't understand what possible use it has. This is the way he explained it to me: "If you edit the text in the top line to the name of another existing file, the rename command will affect *that* file. For example, say you highlight file 'FOO' and choose File|Rename. Change 'FOO' to 'BAR' in the top box, fill in 'SNAFU' in the bottom, and you've renamed BAR to SNAFU, leaving FOO unchanged. Weird? Sure!" This begs the question: was this

intentional or an accidental side effect of using an edit gizmo where an output-only text gizmo was required? Hey, Neil, wanna bet it was an accident? Actually, this begs another question: How did Neil discover this behavior?

The program doesn't fill in the second field for you

Now let's look at the second text-edit gizmo. If the lower text-edit gizmo is the proper place to construct the new filename, why doesn't the program give the user a head start by filling the field in with the current name of the file? The lack of this obvious offering is what often tricks me into trying to edit the top field. The program should be saying "here is the field as it currently exists, edit it to what you desire." Instead, it is saying "tell me exactly what you want and I don't really care if it takes extra work on your part."

Frequently, when I want to rename a file (like the word processing file in the previous example), I will only be thinking about the eight-character file name, leaving management of the three-character file extension to the discretion of the program. When I key in the new name, I will naturally forget to add the file extension, the ".DOC." Of course, the dialog box will merrily rename the file to something that my word processor will no longer recognize. Adding insult to injury, File Manager itself will now not know what to do when I double-click on the new file name—without any memory, the File Manager depends utterly on the three-character extension to know what to do. So this sequence often occurs to me: I rename a file, omitting the extension, then try to launch it with a double-click and get an error message from File Manager for my trouble. My friend Richard Schwartz at Borland has a riddle that illustrates this type of software interaction. He says, "Ask me if I'm a fish." I dutifully reply, "Are you a fish?" and he snaps out "No. Why do you ask?" Arrrrgh!

Regardless of how I use this dialog box, I will always end up with some file name in the second box; otherwise, nothing will happen. Although the range of possible entries is infinite, it is undeniably probable that I will enter *something* in the second gizmo. If the program was better prepared for the probable case, it would be bigger help to the poor user.

> *Design tip:* Prepare for the probable case.

The appropriate way to handle this second field would be to fill it in with a copy of the current file name and then completely select the entire field. By the standard rules of behavior for selection, any single keystroke will replace the

selected portion, so if I wanted to change the name to something completely different, all I would have to do would be begin typing. The old name would instantly blink out of existence and be replaced by the newly typed name. On the other hand, I could easily point with the mouse to a single location between any two characters, and the selection would become a text-entry caret for insertion or correction. The net effect would be better than it is now, with support for either choice I might make.

The program doesn't alert you to failure

When the program leads me down the garden path, teasing me with an edit field that is pointless to edit, and I fall for it, the program doesn't even have the decency to tell me. First, it doesn't say, "Hey, you can't enter text there!" Second, it doesn't say, "Hey, you made some changes in a place where their effect is meaningless." On two counts, the dialog box fails to remain aware of what is happening. The standard dialog box code doesn't support this kind of awareness—the programmer must supply it. I consider this type of failure to maintain situational awareness one of the more frustrating aspects of poorly designed software. It is like handing your secretary a folder and asking him to file it; he says, "Okay" and then awhile later you notice the folder spilled thoughtlessly on the floor. When you ask him about it, he doesn't apologize but tells *you* to pick it all up and hand it to him again. It is very aggravating because it seems like the program isn't saying, "Sorry, I can't do that," but rather saying, "I don't care about your stupid problems."

The dialog isolates the function from context

When I ask to rename a file, I really mean something less specific, like "I need to do something with this file so I can keep track of it better." There are several ways to accomplish this, but only one of them is an actual rename. This dialog box removes the function of renaming a file from the context of managing my file system, which is the purpose of the File Manager. Often, I will consider renaming a file and then decide instead to move the file to another directory. The dialog box doesn't offer any file-movement capabilities, and in fact isolates me from the file-movement tools that are in the File Manager program. If I want to rename a file, it is likely that I don't want to have the new name collide with an already-existing name in the same directory. The dialog box doesn't offer any information about the file, like when it was created, how big it is, when it was last changed or what application would be launched if I

double-clicked on it. Not only that, it covers up what little information is offered by the File Manager on its main screen.

The dialog box doesn't need to exist at all

This dialog box begs the question "Why does it exist at all?" Why didn't Microsoft just make the filename in the directory display editable? I should be able to click on the name of the file (or on a trigger next to it) and get an edit cursor right in the filename display itself. This would instantly solve all of the above complaints: The relationship of edit boxes to things that are editable would be direct, one-to-one and unambiguous. I would only have to enter the changes I wanted, leaving untouched anything that I wanted to remain unchanged. The program would immediately alert me to any failure through the most efficient means possible: the same facility by which it shows me everything else it knows. The context of the file would not be hidden but would be as clearly shown as it normally is in the program. If I changed my mind and decided to move or copy the file instead, the various operations would be seamless, on the same plane and available by direct manipulation.

Whaddaya know? The Explorer in Windows 95 actually does exactly what I describe. It allows files to be renamed in place or dragged from one directory to another. All actions stay within this context. Just don't forget that three letter extension: It is still the only way Windows knows what to do with your file. The Rename dialog box is just a fading memory. My biggest complaint is that I can't use the Explorer as the File Open and Save As dialogs in my applications. Maybe in Windows 97...

Design tip: Have a reason for each idiom.

Don't create a dialog box without first assuring yourself that a dialog box is the appropriate idiom for this interaction. A dialog is a suspension of the normal course of events, and it is incorrect to use one during normal interaction. Conversely, when something out of the ordinary comes along, a dialog box calls the user's attention to the uniqueness of the occurrence.

The File Rename dialog box in Windows 3.x shows how the orchestration of user interaction can founder on something extremely simple. Often we concentrate on the bigger, more obvious complexities of interaction and forget about these simple functions. Inevitably, they are the ones that trip us up.

Posture and State

Most people, especially while they are working, have a predominant behavioral attitude: the teacher is imperious; the toll-taker is bored and invisible; the actor is shining and bigger than life; the butler is obsequious and servile. Programs, too, have a predominant manner of presenting themselves to the user.

A program may be bold or timid, bright or gray, but it should be so for a specific, goal-directed reason. Its manner shouldn't result merely from the personal preference of its designer or programmer. The presentation of the program affects the way the user relates to it, which strongly influences the usability of the product. Programs whose appearance or manner conflict with their purpose seem somehow clunky and inappropriate, like a loud, profane voice in a church or shouting from the audience during a stageplay.

Posture

The behavior of your program should reflect how it is used, rather than an arbitrary standard. If your program is used like Excel, then modeling its behavior after Excel is suitable. If not, your program runs the risk of ending up looking like Henry Kissinger dancing the hula.

151

I call a program's behavioral stance—the way it presents itself to the user—its posture.

The look and feel of your program is not as much an aesthetic choice as much as it is a behavioral choice. Your program's posture is its behavioral foundation.

I divide desktop applications into four categories of posture: sovereign, transient, daemonic and parasitic. Because each describes a different set of behavioral attributes, they also describe different types of user interaction. More importantly, they give the designer a point of departure for designing an interface. A sovereign posture program, for example, won't feel right unless it behaves in a "sovereign" way.

Sovereign Posture

I call a program that is the only one on the screen, monopolizing the user's attention for long periods of time, a sovereign posture application.

Sovereign applications travel in royal splendor, surrounded by their numerous courtiers. They offer a panoply of related functions and features, and users tend to keep them up and running continuously. Good examples of this type of application are word processors and spreadsheets. Many vertical applications are also sovereign applications, as they often deploy on the screen for long periods of time and interaction with them can be very complex and involved. Users working with sovereign programs often find themselves in a state of flow. Sovereign programs are usually used maximized (we'll talk more about window states later in this chapter). For example, it is hard to imagine using Word in a 10×15 cm window—at that size it's not really appropriate for its main job: creating and editing documents.

Sovereign programs are characteristically used for long, continuous stretches. I'm using Word to write this manuscript; it has been the only one on screen for the last hour and will remain so for many hours to come. Typical. A sovereign program dominates a process as its primary tool. PowerPoint, for example, is camped out full screen while I create a presentation from start to finish. Even if other programs are used for support tasks, PowerPoint remains in the royal role.

The implications of sovereign behavior are subtle but quite clear once you think about them. The most important implication is that users of sovereign programs are experienced users. Sure, each user will spend some time as a novice, but only for a short period of time *relative to the amount of time he*

will eventually spend using the product. I'm not making light of the difficulty the new user has in getting over the painful hump of first-learning, but, seen from the perspective of the entire relationship, the time the user spends getting acquainted with the program is small.

Sovereign users are experienced users
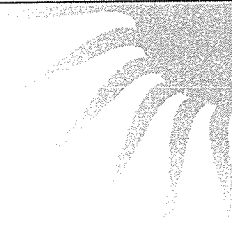
From the designer's point of view, this means that the program should be designed for optimal use by experienced users, and not primarily for first-time users. Sacrificing speed and power in favor of clumsier but easier-to-learn idioms is out of place here. Of course, if you can offer easier idioms without compromising the interaction for experienced users, that is always best.

Between first-time users and experienced users there are many people who must use sovereign applications only on occasion. These infrequent users cannot be ignored, and the quality of the interface will be reflected in the product's acceptance by its infrequent users. However, the success of a sovereign application will still be completely dependent on its experienced, frequent users until someone else satisfies both them and the inexperienced or first-time users. WordStar, an early word processing program is a good example. It dominated the word processing marketplace in the late '70s and early '80s because it served its experienced users so well, even though it was extremely difficult for infrequent and first-time users. WordStar Corporation thrived until its competition offered the same power for experienced users while simultaneously making it much less painful for infrequent users. The WordStar company rapidly shrank to insignificance.

Take the pixels

Because the user's interaction with a sovereign program dominates his session at the computer, the program shouldn't be afraid to take as much video real estate as possible. No other program will be competing with yours, so expect to take advantage of it all. Don't ever waste space, of course, but don't be shy about taking what you need to do the job. If you need four toolbars to cover the bases, use four toolbars. In a different type of program, four toolbars may

be overly complex, greedy and inappropriate, but the sovereign posture has a defensible claim on the pixels.

Generally, as I said before, you can expect that sovereign programs will be running maximized. In fact, in the absence of explicit instructions from the user, your sovereign application should always default to maximized. The program needs to be fully resizable, and must work in all manner of oddball configurations, but optimize its interface for maximization instead of the oddball stuff.

Toolbars are mostly populated with familiar, three-dimensional, rectangular push-buttons with pictographic icons on them instead of text. Naturally, I call these gizmos buttcons; a simple combination of buttons and icons. I introduce the term now because it is relevant to this discussion, but we'll discuss these gizmos in detail in Parts V and VI.

Because the user will stare at a sovereign application for long periods, you should take care to mute the colors and texture of the visual presentation. Keep the color palette narrow and conservative. That big red-striped gizmo may look really cool to newcomers, but it will seem garish after a couple of weeks of daily use. Tiny dots or accents of color will have more effect in the long run than big splashes, allowing you to pack controls together more tightly than you could otherwise.

Your user will stare at the same palettes, menus and toolbars for many hours, gaining an innate sense of where things are from sheer familiarity. This gives you, the designer, the freedom to do more with fewer pixels. Buttcons can be smaller than normal. Auxiliary controls like screen-splitters, rulers, scrollers and other manipulable items can be smaller and more closely spaced.

Sovereign applications are great platforms for creating an environment truly rich in visual feedback for the user. You can productively add extra little bits of information into the interface. The status bar at the bottom of the screen, the ends of the space normally occupied by scroll-bars, the caption bar and other dusty corners of the program's visible extents can be filled with graphs, numbers, indicators, simulated leds and many other visual indications of the program's status, the status of the data, the state of the system and hints for more productive user actions. While enriching the visual feedback, you must be careful not to create an interface that is hopelessly cluttered and busy.

The first-time user won't even notice such artifacts, let alone understand them, because of the subtle way they are shown on the screen. After a couple of

months of steady use, though, he will begin to see them, wonder about their meaning, and explore them. At this point, the user will be willing to expend a little effort to learn more, and if you provide an easy means for him to find out what the artifacts are, he will become not only a better user, but a more satisfied user, as his power over the program grows with his understanding. Adding such richness to the interface is like adding a variety of ingredients to a meat stock—it enhances the entire meal.

In the same vein, sovereign programs benefit from rich input. Every frequently used aspect of the program should be controllable in several ways. Direct manipulation, dialog boxes, buttons, keyboard mnemonics and keyboard accelerators are all appropriate. You can make more aggressive demands on the user's fine motor skills with direct-manipulation idioms. Sensitive areas on the screen can be just a couple of pixels across, because you can assume that the user will be established comfortably in his chair, arm positioned in a stable way on his desk, rolling his mouse firmly across a resilient mouse pad.

Go ahead and use all of the corners of the program's window for controls. In a jet cockpit, the most frequently used controls are situated directly in front of the pilot; those needed only occasionally or in an emergency are found on the armrests, overhead, and on the side panels. In Word, Microsoft has put the most frequently used functions on buttons on the two main toolbars (see Figure 12-1). They put the frequently used but visually dislocating functions on small buttons to the left of the horizontal scroll-bar near the bottom of the screen. These controls change the appearance of the entire visual display— NORMAL VIEW, PAGE LAYOUT VIEW and OUTLINE VIEW. They are not usually used by neophytes and, if accidentally triggered, they can be confusing. By placing them near the bottom of the screen, they become almost invisible to the new user. Their segregated positioning subtly and silently indicates that caution should be taken in their use. More experienced users, with more confidence in their understanding and control of the program will begin to notice these controls and wonder about their purpose. They will experimentally press them when they feel fully prepared for their consequence. This is a very accurate and useful mapping of control placement to usage.

Interactions that involve a delay won't be appreciated much by the user. Like a grain of sand in your shoe, a one- or two-second delay gets awfully painful when frequently repeated. It is perfectly acceptable to have procedures that take time, but they should not be ones that are frequent or repeated during the normal use of the product. If, for example, it takes more than a fraction of a
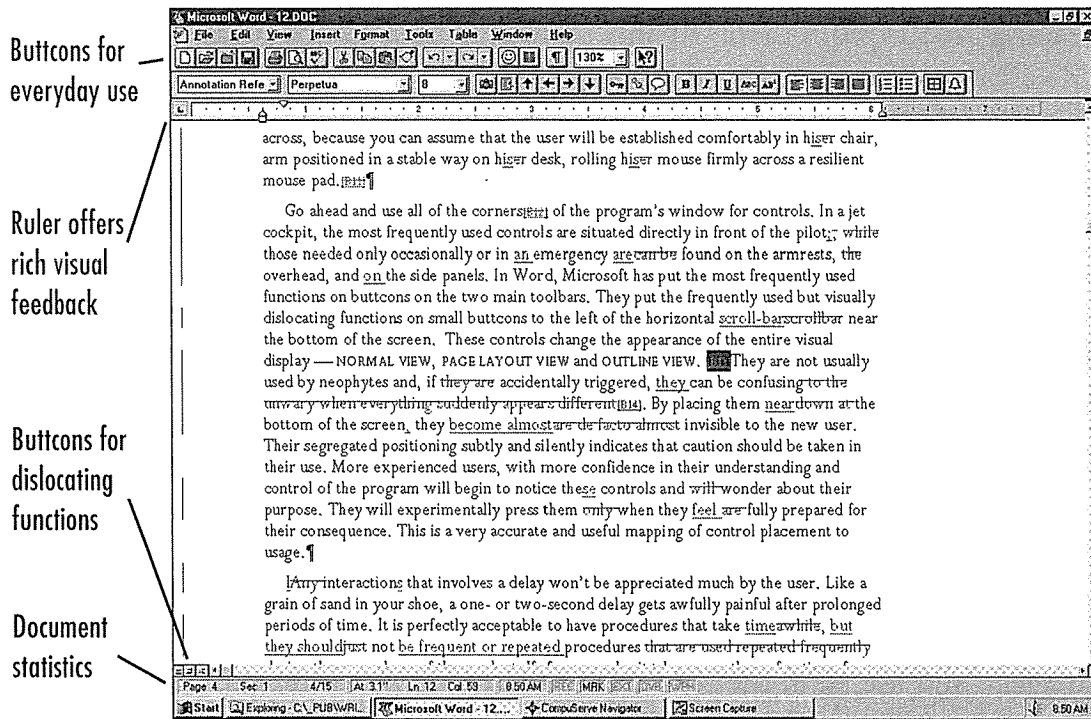
Buttcons for everyday use

Ruler offers rich visual feedback

Buttcons for dislocating functions

Document statistics

Figure 12-1

Microsoft Word is a classic example of a sovereign-posture application. It stays on-screen, interacting with the user for long, uninterrupted periods. Notice how Microsoft has built controls into both the top and the bottom of the application. Those at the top are more benign than those at the bottom, which are segregated because they can cause significant visual dislocation.

second to save the user's work to disk, that delay will quickly come to be viewed as unreasonable. On the other hand, inverting a matrix or changing the entire formatting style of a document can take a few seconds without causing irritation because the user can plainly see what a big evolution it is. Besides, he won't want to invoke the evolution very often.

The dictum that sovereign programs should maximize on the screen is also true of document windows within the program itself. Child windows containing documents should always be maximized inside the program unless the user explicitly instructs otherwise.

Many sovereign programs are also document-centric, making it easy to confuse the two, but they are not the same. Most of the documents we work with are

8½-by-11 inches and won't fit on a standard video screen. We strain to show as much of them as possible, which naturally demands a maximized aspect. If the document under construction were a 4-by-6 inch photograph, for example, a document-centric program wouldn't need to take the full screen. The sovereignty of a program does not come from its document-centricity nor from the size of the document—it comes from the nature of the program's use.

Transient posture

If a program manipulates a document but only does some very simple, single function, like scanning in a graphic, it isn't a sovereign application and shouldn't exhibit sovereign behavior. Such single-function applications have a posture of their own, which I call **transient**.

A transient-posture program comes and goes, presenting a single, high-relief function with a tightly restricted set of accompanying controls. The program is called when needed, appears, performs its job, then quickly leaves, letting the user continue his more normal activity, which is usually working with a sovereign application.

The salient characteristic of transient programs is their temporary nature. Because they don't stay on the screen for extended periods of time, the user doesn't get the chance to become very familiar with them. Consequently, the program's user interface needs to be unsubtle, presenting its controls clearly and boldly with no possibility of mistakes. The interface must spell out what it does: this is not the place for artistic-but-ambiguous images on buttcons—it *is* the place for big buttons with precise legends spelled out in big, 12-point type.

Although a transient program can certainly operate alone on your desktop, it usually acts in a supporting role to a sovereign application. For example, calling up the Explorer to locate and open a file while editing another with Word is a typical transient scenario. So is checking your email. Because the transient program borrows space at the expense of the sovereign, it must respect the sovereign by not taking more space on screen than is absolutely necessary. Where the sovereign can dig a hole and pour a concrete foundation for itself, the transient program is just on a weekend campout. It cannot deploy itself on screen either graphically or temporally. It is the Roto-Rooter truck of the software world.

While a transient program must conserve the total amount of video real estate it consumes, the gizmos on its surface can be proportionally larger than those

on a sovereign application. Where such heavy-handed visual design on a sovereign program would pall within a few weeks, the transient program isn't on screen long enough for it to bother the user. On the contrary, the coarser graphics help the user to orient himself more quickly when the program pops up. The program shouldn't restrict itself to a dull corporate gray, either, but should paint itself in brighter colors to help differentiate it from the hosting sovereign, which may be more appropriately attired in gray flannel.

Transient programs should use brighter colors and bold graphics to convey their purpose—there is little time for the user to visually orient himself with conservative coloration. The user needs big, bright, reflective road signs to keep him from making the wrong turn at 100 kilometers per hour. Animated buttons are certainly not out of place here. A little bit of animation goes a long way, as our eyes are drawn to movement. A couple of pixels changing slowly is all it takes to ensure that the user notices.

Transient programs should have instructions built into their surface. The user may only see the program once a month, and will likely forget the meanings of the choices presented. Instead of a button captioned "Setup," it might be better to make the button large enough to caption it "Set Up User Preferences." The meaning is clearer, and the button more reassuring. Likewise, nothing should be abbreviated on a transient program—everything should be spelled out to avoid confusion. The user should be able to see without difficulty that the printer is busy, for example, or that the audio clip is 5 seconds long.

Once the user summons a transient program, all of the information and facilities he needs should be right there on the surface of the program's single window. Keep the user's locus of attention on that window and never force him into supporting sub-windows or dialog boxes to take care of the main function of the program. If you find yourself adding a dialog box or second view to a transient application, that's a key sign that your design needs a review.

Transient programs are not the place for tiny scroll-bars and fussy point-click-and-drag interfaces. You want to keep the demands here on the user's fine motor skills down to a minimum. Simple push-buttons for simple functions are better. Anything directly manipulable must be big enough to move to easily: at least twenty pixels square. Keep controls off the borders of the window. Don't use the window bottoms, status bars or sides in transient programs. Instead, position the controls up close and personal in the main part of the window.

You should definitely provide a keyboard interface, but it must be a simple one. It shouldn't be any fancier than ENTER, ESCAPE and TAB. You might add the arrow keys, too, but that's about it.

Of course, there are exceptions to this monothematism, although they are rare. If a transient program performs more than just a single function, the interface should communicate this visually. For example, if the program imports and exports graphics, the interface should be evenly and visually split into two halves by bold coloration or other graphics. One half could contain the controls for importing and the other half the controls for exporting. The two halves must be labeled unambiguously. Whatever you do, don't add more windows or dialogs.

Keep in mind that any given transient program may be called upon to assist in the management of some aspect of a sovereign program. This means that the transient program, as it positions itself on top of the sovereign, may obscure the very information that it is chartered to work on. This implies that the transient program must be movable, which means it must have a caption bar. Making it reshapable may also be desirable, though not mandatory.

Having said that, it is vital to remember how important it is to keep the amount of management overhead as low as possible with transient programs. All the user wants to do is call the program up, request a function and then end the program. It is completely unreasonable to force the user to add non-productive window-management tasks to this interaction. Manipulating the Windows semi-standard multiple document interface (MDI) is a strong example of non-productive window management.

MDI can certainly be useful in some situations, but I cannot imagine a need for it in any transient program. For example, the Program Manager in Windows 3.x is a transient program, yet it insists on using MDI. The amount of time and frustration its users spend wondering where their icons are, zooming, moving, reorganizing and managing their group windows is high in relation to the benefit. All the program really does is launch applications, which isn't demanding enough to require all of that paperwork. This high demand for bureaucratic management overhead is one of the reasons why there is such a brisk business in Windows shell replacement programs. Even Microsoft has replaced it in Windows 95.

The most appropriate way to help the user with both transient and sovereign apps is, as usual, to give the program a memory. If the transient program

remembers where it was the last time it was used, the chances are excellent that the same shape and place will be appropriate next time, too. It will almost always be more apt than any default setting might chance to be. Whatever shape and position the user morphed the program into is the shape and position the program should reappear in when it is next summoned. Of course, this holds true for its logical settings, too.

On the other hand, if the use of the program is really simple and single-minded, go ahead and specify its shape—omit the thickframe, the directly resizable window border. Save yourself the work and remove the complexity from the program. I say this with some trepidation, though, as this can certainly be abused. The goal here is not to save the programmer work—that's just a collateral benefit—but to keep the user aware of as few complexities as possible. If the program's functions don't demand reshaping, and the overall size of the program is small, Occam's razor, the principle that "simpler is better," takes on more importance than usual. The Windows calculator, for example, isn't resizable. It is always the "correct" size and shape.

No doubt you have already realized that almost all dialog boxes are really transient programs. You can see that all of the above guidelines for transient programs apply equally well to the design of dialog boxes.

Daemonic Posture

I call programs that do not normally interact with the user **daemonic posture** programs. These programs serve quietly and invisibly in the background, performing possibly vital tasks without the need for human intervention. A printer driver is an excellent example.

As you might expect, any discussion of the user interface of daemonic programs will be necessarily short. Too frequently, though, programmers give daemonic programs full-screen control panels that are better suited to sovereign programs. Designing your fax manager in the image of Excel, for example, is a fatal mistake. At the other end of the spectrum, daemonic programs are too-frequently unreachable by the user, causing no end of frustration when adjustments need to be made.

Where a transient program controls the execution of a function, daemonic programs usually manage processes. Your heartbeat isn't a function that must be consciously controlled; rather, it is a process that proceeds autonomously in the background. Like the processes that regulate your heartbeat, daemonic

programs generally remain completely invisible, competently performing their process as long as your computer is turned on. Unlike your heart, however, daemonic programs must occasionally be installed and removed and, also occasionally, they must be manually adjusted to deal with changing circumstances. It is at these times that the daemon talks to the user. Without exception, the interaction between the user and a daemonic program is transient in nature, and all of the imperatives of transient program design hold true here also.

The principles of transient design that are concerned with keeping the user informed of the purpose of the program and of the scope and meaning of the user's available choices become even more critical with daemonic programs. If you recognize that in many cases the user will not even be consciously (or unconsciously) aware of the existence of the daemonic program, it becomes obvious that reports about status from that program can be quite dislocating if not presented in an appropriate context. Since many of these programs perform esoteric functions—like printer drivers or communications concentrators—the messages from them must take particular care not to confuse the user or lead to misunderstandings.

A question that is often taken for granted with programs of other postures becomes very significant with daemonic programs: if the program is normally invisible, how should the user interface be summoned on those rare occasions when it is needed? One of the most frequently used methods is to represent the daemon with an on-screen program icon the way the After Dark screen savers do. Putting the icon so boldly in the user's face when it is almost never needed is a real affront, like pasting an advertisement on the windshield of somebody's car. If your daemon needs configuring less than once a day, get it off the main screen.

A better approach is to create a "control panel" application that will be found by the Windows control panel program, CONTROL.EXE, and shown in its window. The user then has a consistent place to go for access to such process-centric applications. An equally effective solution is to create a transient program that runs as a launchable application to configure the daemon.

Parasitic Posture

I call programs that blend the characteristics of sovereign and transient programs parasitic posture programs. The parasitic program is continuously present like a sovereign, but it performs only a supporting role, is small and is

superimposed on another application the way a transient is. The Windows Clock and Microsoft Office Manager are two good examples of parasitic programs.

Parasitic programs typically are silent reporters of ongoing processes. In some cases, this reporting may be a function they perform in addition to actually managing that process, but this is not necessarily true. A parasite may, for example, monitor the amount of system resources either in use or available. The program constantly displays a small bar chart reflecting the current resource availability. There are many parasitic shareware applications that, for example, paint a clock on every program's caption bar, or display an icon with animated eyeballs that constantly watch the mouse cursor, or show how much memory is free.

A process-reporting parasitic program must be simple and often bold in reporting its information. It rides on top of a sovereign application, so it must be very respectful of the preeminence of that other program and should be quick to move out of the way when necessary.

Parasitic programs are not the locus of the user's attention; that distinction belongs to the host application. For example, recall the case I cited earlier of my client who has the automatic call distribution (ACD) program. An ACD is used to evenly distribute incoming calls to teams of human "agents" who are trained order-takers or customer-support representatives. Each agent has a computer running an application specific to his job. This application, because it is the primary reason for the system's existence, is always a sovereign-posture application; the ACD program is a parasite on top of it. For example, a sales agent will field calls from prospective buyers on an incoming toll-free number. The agent's order entry program is the sovereign, while the ACD program is the parasite, riding on top to feed incoming calls to the agent. The ACD program must be very conservative in its use of pixels because it always obscures some of the underlying sovereign application. It can afford to have small features because it will be on the screen for long periods of time. In other words, the gizmos on the parasite can be designed to a sovereign's sensibilities.

Other Postures

There are other program postures, but most programs you design will fall into one of these four categories. Visual Basic is a notable exception, following as it does none of the four posture paradigms. As the inventor of its somewhat

oddball configuration, I feel that I should explain its genesis. The idea behind VB was to create a visual programming language that had two distinct modes: programming and running the result. The generated program would be a standalone application, not part of the VB tool suite. This, in turn, dictated the design. In programming mode, the user's program would have one or more "forms" visible on the screen, surrounded by a suite of tools. The forms are independent of the tools, rather than being enclosed in the tool application window. This enabled the next step, wherein the user could shift to "running the result" mode, and only the tools would disappear, leaving the forms untouched and visible.

The actual need for a posture like Visual Basic's is exceedingly rare, yet it is astonishing how many programs copy it. This says to me that many programmers don't know how to select their program's posture and merely copy that of the language in which it is written. Properly determining your program's posture will tell you much about its behavioral persona which, in turn, will dictate many of the important guidelines for the design process. This is much like how a novelist or screenwriter constructs a story by creating characters, putting them in a situation, then letting them act "in character." As a user interface designer, you get a lot of bang for your buck merely by assuring that your program behaves in the posture most appropriate for its behavior.

Windows states

A Microsoft Windows programmer would call an application's primary window its top-level window.

The intrinsic behavior of a top-level window includes the ability to overlap other top-level windows, but this is not how they are normally used. Each top-level window has the native ability to be in one of three states, depending on how they are programmed. Oddly, only two of these three states have been given names by Microsoft: minimized and maximized.

They somehow manage to avoid directly referring to the third state, which you get to by using the button labelled with both the up- and down-arrows, and the only hint of a name is on the system menu where the verb "Restore" describes the other way to get to it. This function "restores" a minimized top-level window to its previous state, either maximized or that other state. In the interests of sanity, I call this third state pluralized, although it has been called "restored" more than once.

The pluralized state is that in-between condition where the window is neither an icon nor maximized to cover the entire screen. When a window is pluralized, it shares the screen with icons and other pluralized windows.

In Version 1.0 of Windows, the states of minimization and maximization were called "iconized" and "zoomed," terms that were more descriptive and certainly more engaging. IBM, then enjoying a cozy relationship with Microsoft, demanded the change to corporate-speak in the mistaken impression that America's executives would feel more comfortable. The weaker appellations have stuck.

The normal state for a sovereign application is maximized. There is no reason for such a program to be pluralized, other than to support switching between programs. Some transient applications, like the File Manager or the Explorer, are appropriately run pluralized, but these transient programs are used merely as springboards for sovereign applications. Many users, however, run their sovereign programs in the pluralized state, and I don't know why (other than merely because that is the program's default and the user is too timid to change it). By definition, a sovereign application will be in use for extended periods of time, and any pixels not used by it are wasted. There aren't enough pixels to waste. I suspect that those users who run sovereign applications pluralized do so because the exercise of switching between maximized applications is too great for them to bother fussing with: it's easier to just accept the loss of video real estate. For those of us who are persistent enough to master the technique, though, maximized sovereign applications are the normal mode of operating.

Why minimize?

Any application can be minimized, but why? I can think of two reasons, but neither of them makes much sense.

You can minimize to switch from one application to another, but this is an ungainly procedure. You minimize the active program, then maximize (or pluralize) the icon of the desired program. To switch back, you reverse the sequence. You must move the mouse all over the screen and the process is slow, complicated and ungainly.

In Windows 3.x, the ALT-TAB key sequence is a much more useful method of switching between applications, but it is obscure, not visual, demands a high level of user expertise, is relatively unknown outside of the power-user community and operates unlike any other idiom in Windows. Pressing ALT-TAB

moves you quickly and directly to the next running program. Holding down the ALT key and repeatedly pressing the TAB key cycles you through each running program. It does this by showing a small window in the center of the screen with the name and icon of the candidate program. The trick is that the actual selection of a program occurs when the user *releases* the ALT key! Nowhere else in Windows does an action occur on the release of a shifting key. This idiom is weird enough that most people don't know about it (I used Windows for seven years before I discovered it), and learning it can be difficult. Once the idiom is learned, though, it is remarkably powerful and clearly the best way to navigate between applications. Besides the speed of the technique for switching from program to program, the great advantage is that the various programs can each remain in their natural state, either maximized or pluralized, but usually maximized.

The ALT-TAB idiom is a classic example of how a programming staff can ingeniously solve a significant problem that baffled the experts. Many sharp software designers tried to create convenient program-switching idioms, but none are the equal of this one. The solution is brilliant but virtually undiscoverable—it's not documented and it doesn't appear on any menu, so someone must tell you about it. The solution is fabulously economical of overhead but requires a deep familiarity and dexterity with the computer, coupled with a clear sense of dominance over it—a good description of your average code-slinger. What we really needed was a more benign version of ALT-TAB that wasn't just for power-users and hackers. In Windows 95, we get this solution with the Startbar.

The Windows 95 Startbar finally acknowledges that most people want to work on one maximized sovereign application at a time, and that they want a more accessible idiom for accomplishing this. A significant percentage of the screen's real estate is devoted to this ever-present gray bar, but it is worth it for everyone but the most hard-core programmers (and they can always exercise their option of turning it off). The Startbar contains a button for every running program, regardless of its current state (except daemonic-posture programs, of course). The button for the active program is shown in its pushed-in state.

The Startbar is a simple and visual implementation of the ALT-TAB idiom: you press the button of the application you want, and it moves to the front of the screen and becomes active. If it was last in a maximized state, it will now be maximized. If it was last in a pluralized state, it will now be pluralized and in the same position it was in before. If you imagine your running programs as

cards in a deck, the buttons on the Startbar cut the deck directly to the desired program with a single click of the mouse button. Or you can imagine the running programs as channels on your TV—pressing buttons jumps from channel to channel.

The other reason to minimize a program is to reduce clutter on your screen. If you run several pluralized sovereign applications, it can simplify your screen to minimize some of them. However, this is treating the symptom rather than the cause of the problem. Running several pluralized applications is far too wasteful of pixels. If each application is maximized in turn, there will be no apparent clutter, and minimizing won't be necessary, as long as you have the Startbar or the ALT-TAB to navigate between them. Within MDI programs, it also can make sense to minimize document windows, particularly in a program like Program Manager. Although minimizing helps out here, the real problem isn't the state of the document windows. It's the Program Manager itself. This program provides extremely poor tools for organizing programs, even though that is its sole job. Thankfully, it, too, has been discarded in favor of the Startbar in Windows 95.

Windows 95 removes the only two rationales for minimizing a program. Managing your programs as a deck of cards is superior when you have adequate tools. The ALT-TAB has long allowed power-users to work this way, and the Startbar finally brings the capability to the rest of us.

Minimized applications have been used in curious and innovative ways, but ultimately their usefulness is immaterial, because most sovereign applications run maximized. The icons are always covered up, and whatever clever information they were displaying is invisible. Iconic programs have shown remaining memory, remaining resource storage and the amount of disk traffic. They have pointed to the cursor's location and reported on the results of background communications processes. Several of these functions are of interest only to that rapidly shrinking population with extremely small computers, and the rest can be duplicated by using the Startbar.

Why pluralize?

Is there any reason for a program to be pluralized? Well, maybe. Sometimes situations require two or more programs to be juxtaposed, frequently in development tasks. If all the user wants to do is run one sovereign program after another, with an intersprinkling of smaller transient programs temporarily

overlaid on them, pluralization is unnecessary. If the user wants to cut and paste information between sovereign programs, the clipboard will work just fine. However, if the user wishes to take advantage of the program-to-program drag-and-drop facility brought to fruition with OLE 2.0, the two programs doing the dragging-and-dropping must both be visible, sharing the screen. In other words, both programs must be pluralized.

Modern SVGA computer screens range from 640×480 pixels to 1600×1280 pixels. Arguably, 1024×768 is the most common resolution in the latter half of the '90s. In such limited physical environments, modern sovereign programs such as word processors or graphics programs are difficult and unpleasant to use when they own less than half of the screen. When giving demonstrations to the press or captains of industry, Microsoft proudly demonstrates the drag-and-drop of a spreadsheet into a word processor. The windows of the two applications are carefully posed in advance to illustrate this single function in isolation. What they don't show you is that the management overhead of pluralizing two windows and then adjusting them manually so that each one gets sufficient exposure is considerably greater than the management overhead of using the clipboard and Startbar and merely swapping between the two sovereign programs.

If you right-click on the Startbar, the context menu offers automatic control of tiling. This is certainly a boon if you want to tile, but why bother? Tiling for an SVGA screen is like a tool that allows you to put 40 people in an elevator; it's just easier and more pleasant to wait for the next one.

Program-to-program drag-and-drop is a powerful idiom, and one that we may see with increasing frequency in the future. However, we won't see it used too much between sovereign applications until our computer screens get a lot bigger, which doesn't promise to happen for several more years. However, program-to-program drag-and-drop can be a boon for moving information between a sovereign application and a transient application. For example, let's look at the process of adding a piece of clip art to a word processing document.

The word processor is the sovereign application, and it is running maximized. The clip art librarian is a transient application and would normally run as a fixed-size window approximately one-quarter of the full size of the screen (one half of the width and one half of the height). The clip art librarian could be easily positioned in the least obtrusive quadrant of the screen and, after the desired image is located, the image could be dragged directly into the appropriate place

in the word processor. Another click and the clip art librarian is stored away on the Startbar and the user can proceed. Window management overhead is a mere two clicks, one to open and one to close the librarian, with a possible click-and-drag operation to move the librarian out of the way so the critical area of the word processor's display can be seen more easily. Neither application needed to be explicitly pluralized.

Modern Microsoft Windows software can be effectively built without supplying the ability to either minimize or pluralize. Programs occupying the in-between state of pluralization are dying out, and Windows 95 is helping to kill them. The programs never have to pass through the pluralized or iconized state. Of course, that doesn't mean that you can actually dispense with these options. You must be able to minimize for backwards compatibility, and every program that can maximize must be able to be pluralized for those oddball cases when a user needs to tile the screen with the application (or just in case Microsoft decides to demonstrate your program at COMDEX). Although it is difficult to guess what the case might be, experienced users will probably be upset if the application can't acquit itself of this basic expectation, even though it is more an exercise in adaptability than in practical software design. In this, it is analogous to compulsory figures in figure skating or ground-reference maneuvers for private pilots—not tremendously useful in themselves, but they show a mastery of other necessary skills.

For practical purposes, we are left with only two program configurations: maximized sovereign programs and pluralized transient programs. The sovereign programs endure while the transient programs appear briefly on top of them. When you design your application, you must make this fundamental design decision: dominant or temporary. This will dictate the type of main window you will use.

MDI

Several years ago, Microsoft began proselytizing a new method for organizing the functions in a Windows application. They called this the multiple document interface, or MDI. It satisfied a need apparent in certain categories of applications, namely those that handled multiple instances of a single type of document simultaneously. Notable examples were Excel and Word.

Microsoft backed up their new standard with code built into the operating system, so the emergence of MDI as a standard was inevitable. For a time in the

late '80s and early '90s, MDI was regarded by some at Microsoft as a kind of cure-all patent medicine for user interface ills. It was prescribed liberally for all manner of ailments.

Now Microsoft seems to be turning its back on MDI and embracing something called single document interface, or SDI. It seems that MDI didn't fix all the problems after all.

If you want to copy a cell from one spreadsheet and paste it to another, the tedium of opening and closing both spreadsheets in turn is very clunky. It would be much better to have two spreadsheets open simultaneously. Well, there are two ways to accomplish this: You can have one spreadsheet program that can contain two or more spreadsheet instances inside of it. Or you can have multiple instances of the entire spreadsheet program, each one containing a single instance of a spreadsheet. The second option is technically superior, but it demands high-performance equipment.

In the early days of Windows, Microsoft chose the first option for the simple, practical reason of resource frugality. Remember, early Windows versions had to run in real mode on 286 processors. This was sort of like running an underwater sack race on Quaaludes. One program with multiple spreadsheets (documents) was more conservative of bytes and CPU cycles than multiple programs, and performance matters.

Unfortunately, the one-program-multiple-documents model violated a fundamental design rule established early on in Windows: Only one window can be "active" at a time. What was needed was a way to have one program active at a time along with one document window active at a time within it. MDI was the hack that implemented this solution.

Two conditions have emerged in the years since MDI was made a standard. First, the facility was tragically abused by well-meaning programmers. Second, our computers have gotten much more powerful—to the point where multiple instances of programs, each with a single document, are very feasible. Windows 95 with its 32-bit kernel and its preemptive multitasking make the formerly rejected model much more attractive, so Microsoft has made it clear that MDI is no longer politically correct, if not actually doomed.

The winds of change at Microsoft notwithstanding, MDI is actually a fine thing, as long as it is not abused. The main way to abuse it is to have more than one type of document window in a single program. Figure 7-1 shows what I

mean. The CompuServe Navigator program offers a dozen or more different types of document windows making it very difficult to understand what is going on. This is very frequently done, and is one of the reasons why some designers would like to see the whole facility thrown out. I see nothing wrong with MDI in a sovereign application like a word processor or spreadsheet, as long as there is only one type of document. Otherwise, confusion sets in as functions lose their sharp edges. Typically, as document windows of different types are selected, the menus change to keep up. This is not an absolutely bad thing, but it is absolutely not a good thing. The user depends on the permanency of menus to help keep them oriented on the screen. Changing them bleeds away this reliability.

Everything I said in the earlier discussion about minimizing, maximizing and pluralizing windows applies to document windows inside an MDI application. If you have to zoom and move and putz with little windows, it is bad design. It is much better to go cleanly from one window to the next. Going from one fully maximized spreadsheet to another fully maximized spreadsheet is powerful and effective. In a few years, when our computers will easily run multiple copies of all of our applications, there will be little effective difference between the MDI and SDI. In MDI, you go to the "Window" menu to change from spreadsheet to spreadsheet, but you go to the Startbar to change from Excel to Word. In SDI, you will go to the Startbar to change both. SDI is clearly better just for this fact, but it's still not a lot better.

Overhead and Idiocy
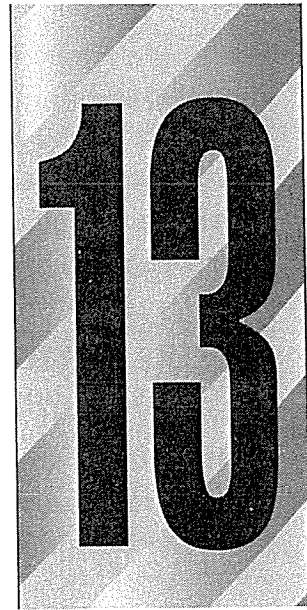
Software developers often implement stunningly elegant cases of user interface foolishness. They create interactions that are top-heavy with extra work for the user and programs that exhibit really idiotic behavior. Programmers tend to do this because they focus so intently on the enabling technology that they don't see things from a goal-directed point of view. Designers often do this because much of their design work is derivative. They do it the bad way because that is the way it has always been done. But we can free ourselves from these shackles of technology and the past. All we have to do is insist on holding every interaction up to the yardstick of the user's goals.

Overhead

When I want to drive to the office, I have to open the garage door, get in, start the motor, back out and close the garage door before I even begin the forward motion that will take me to my destination. All of these actions are in support of my automobile rather than in support of getting to my destination. If I had a mental-telepathy-matter-transference module, I'd just picture my destination in my mind and then be there—no garages, no motor. My point here is not to

171

complain about the intricacies of driving, but rather to point out the difference between two types of actions we take to accomplish our daily tasks. Any large task, such as driving to the office, involves many smaller tasks. Some of these, which I call revenue tasks, work to solve the problem directly; these are tasks like steering down the road toward my office.

Other tasks, which I call excise tasks, don't contribute directly to solving the problem but are necessary to accomplishing it just the same.

Such tasks include opening and closing the garage door and starting the engine, in addition to putting oil and gas in the car and performing periodic maintenance.

Excise is the extra work that satisfies the needs of our tools as we use them to achieve our objectives. The distinction is sometimes hard to see because we get so used to the excise being part of our tasks. In the above example, this is very true. Most of us drive so frequently that differentiating the act of opening the garage door from the act of driving towards our destination is difficult. Manipulating the garage door is something we do for the car, not for us, and it doesn't move us towards our destination the way the accelerator pedal and steering wheel do.

You may complain that opening and closing the garage door is a task of such monumental trivialness that fretting over it is silly. But imagine if you had to first put air in the car's tires, drive it to the pumps and fill the fuel tank with gasoline, drive it to the oil rack and put oil in the transmission and the crankcase, drive it to the other rack and put hydraulic fluid in the power steering reservoir, brake cylinders and differential, drive it to the vacuum and clean the floor mats, drive it the repair shop and replace the headlights and align the wheels. You'd quickly come to see the difference between excise and revenue tasks. These are all tasks that we perform for the benefit of our automobiles, and not for our benefit. We don't notice them because we don't have to do them every time we need to go somewhere. We only have to change the oil every few months and sometimes we go for a year or more without touching the air in our tires. You can draw a dividing line between serving the car and serving the driver—operating the garage door is on the car side of the line.

Software, too, has a pretty clear dividing line between revenue tasks and excise tasks. Like automobiles, some software excise tasks are trivial and performing them is no great hardship. On the other hand, some software excise tasks are as obnoxious as fixing a flat tire. Installation leaps to mind here, as do such excise

tasks as configuring networks, making backups, connecting to online services and installing sound cards.

The problem with excise tasks is that the effort we expend in doing them doesn't go directly towards accomplishing our goals. Instead, it goes towards satisfying the needs of the tool we use to accomplish our goal. It is overhead. It is the percentage that the house gets. It is friction in the system. Without exception, where we can eliminate the need for excise tasks, we make the user more effective and productive and improve the usability of the software. As a software designer, you should become sensitive to the presence of excise and take steps to eradicate it with the same enthusiasm a doctor would apply to curing an infection.

Design Tip: Eliminating excise makes the user more effective.

Fixing a flat tire and installing software are both obviously onerous excise tasks, and eliminating them from our necessary tasks offers clear benefits. But if we can identify enough software equivalents to opening the garage door, we can streamline our interfaces significantly in many tiny increments rather than in one big leap. Indeed, there are many such instances of petty excise, particularly in GUIs. Virtually all window management falls into this category. Dragging, reshaping, resizing, reordering, tiling and cascading windows qualify as excise actions on the order of the garage door.

GUI excise

One of the main criticisms leveled at graphical user interfaces by experienced computer users—notably those trained on command-line systems—is that getting to where you want to go is made slower and more difficult by the extra effort that goes into manipulating windows and icons. They complain that with a command line, they can just type in the desired command and the computer executes it immediately. With windowing systems, they must open various folders looking for the desired file or program before they can launch it; then, once it appears on the screen, they must stretch and drag the window until it is in the desired location and configuration.

These complaints are well-founded. Extra window manipulation tasks like these are, indeed, excise. They don't move the user towards his goal; they are overhead that the programs demand before they deign to assist the user. But everybody knows that GUIs are easier to use than command-line systems. Who is right?

The confusion arises because the real issues are hidden. The command-line interface forces an even more expensive excise budget on the user: He must first memorize the commands. Also, he cannot configure his screen to his own personal requirements; the command-line program occupies the whole screen without sharing. The excise of the command-line interface becomes smaller only after the user has invested significant time and effort in learning it.

On the other hand, for the casual or first-time user, the visual explicitness of the GUI helps him navigate and learn what tasks are appropriate and when. The step-by-step nature of the GUI is a great help to users who aren't yet familiar with the task or the system. It is also a benefit to those users who have more than one task to perform and who must use more than one program at a time.

Any user willing to learn a command-line interface automatically qualifies as a power user. And, any power user of a command-line interface will quickly become a power user of any other type of interface, GUI included. These users will easily learn each nuance of the programs they use. They will start up each program with a clear idea of exactly what it is they want to do and how they want to do it. To this user, the assistance offered to the casual or first-time user is just in the way. So one person's excise task is often another person's revenue task.

One user's excise task is another user's revenue task
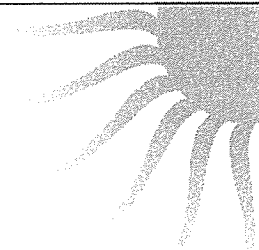
This axiom tells us that we must be careful when we eliminate excise. We must not remove it just to suit power users. Similarly, though, we must not force power users to pay the full price of being helpful to new or infrequent users.

Pure excise

Occasionally—actually, not so occasionally—we find actions that are excise of such purity that nobody needs them, from power users to first-timers. These include most hardware-management tasks like telling a program which IRQ or COM port to use. Any demands for such information should be struck from all user interfaces without a backward glance.

Sometimes, however, we find certain tasks like window management, that, although they are mainly for the program, are useful for occasional users or users with special preferences. In this case, the function itself can only be considered excise if it is forced on the user rather than made available at his discretion.

This brings us back to goal direction, of course. The only way to determine whether a function is excise is by comparing it to the user's goals. If the user wants to see two programs at a time on the screen in order to compare or transfer information, the ability to configure the main windows of the programs so that they share the screen space is not excise. If the user doesn't have this specific goal, any requirement that the user be able to configure the main window of either program is excise.

One of the areas where software designers can inadvertently introduce significant amounts of excise is in support for first-time or casual users. It is easy to justify adding to a program facilities that will make it easy for newer users to learn how to use the program. Unfortunately, these facilities quickly become excise as the user becomes familiar with the program. Facilities added to software for this purpose must be made so that they can be easily turned off. Training wheels are rarely needed for extended periods of time, and training wheels, while a boon to beginners, are a hindrance to advanced learning and use.

Visual metaphor excise

Designers also paint themselves into excise corners by depending on visual metaphors. Visual metaphors like desktops with telephones, copy machines, staplers and fax machines—or file cabinets with folders in drawers—are cases in point. These visual metaphors may make it easy to learn the purpose of the program and to understand the relationships between the pieces, but once these fundamentals are learned, the management of the metaphor becomes pure excise. In addition, the screen space consumed by the images becomes increasingly egregious, particularly in sovereign posture applications. The more we stare at the program from day to day, the more we resent the number of pixels it takes to tell us what we already know. The cute little telephone that so charmingly told us how to dial on that first day long ago is now a barrier to quick communications. It would be much better if we could directly select our goal, such as getting our mail or sending our outgoing letters, from a list rather than having to double-click on the telephone and then select the number to dial.

The visual metaphor that helped us learn the basics has become a significant impediment once we have learned the basics, and the metaphor really didn't teach anything beyond the barest of the basics.

Transient-posture applications can tolerate more training and explanation excise than sovereign applications. Transient-posture programs aren't used frequently, so their users will need more assistance in understanding what the program does and remembering how to control it. For sovereign-posture applications, however, the slightest excise will be like a grain of sand in your shoe: it will produce a painful blister after you've walked just a few miles.

Never make the user ask to ask
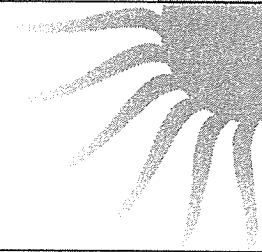
Back in the days of command lines and character-based menus, interfaces would often offer services to the user indirectly. If you wanted to change an item like your address, you had to first ask the program to change it. The program would then offer up a screen where your address could be changed. I call that first question a meta-question, because you are not asking the question but rather asking if you can ask the question.

Meta-questions are pure excise and there is no reason for them in user interface design. If you want to ask a question, the program should let you go ahead and ask. If you want to change some value, you should go ahead and change it. You shouldn't have to ask permission to ask.

Many install programs proffer meta-questions in abundance. They say, "I will install your program in this directory" and then show you the proposed directory. If you don't like the choice shown, you can't just change it in place in that dialog box. You instead press a button that says MODIFY and then get another cascading dialog box to change it.

The program should simply let the user edit the directory name in place. It could easily track those changes, verify their validity and assure that things get installed correctly without forcing the user into the meta-question.

Another way to look at this same problem is from the input-versus-output point of view. Programs and dialogs offer bits of information in the form of
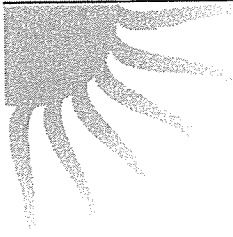
Allow input wherever you output

filenames, numeric values and selected options. If these options are modifiable by the user, he should be able to do the setting right where the program displays them. Many programs have one place where the values are displayed for output and another place where input to them is accepted from the user. This follows the implementation model, which treats input and output as different processes, but the user's mental model doesn't recognize a difference. He thinks, "There is the number; I'll just click on it and enter a new value." If the program can't accommodate this impulse, it is needlessly inserting excise into the interface.

Of course, if changing the information is dangerous or unrecoverable, inserting an interface layer can call attention to the fact that changing this information shouldn't be treated lightly.

Error and confirmation messages

There are probably no bigger excise elements than error message boxes and confirmation message dialogs. These nasty little buggers are so prevalent that eradicating them takes a lot of work. In Part VII, I devote a chapter to each of them, but for now, suffice it to say that they are high in poly-unsaturated excise and should be completely eliminated from your diet.

Other excise traps

You should be vigilant in finding and rooting out each small item of excise in your interface. The myriad little extra steps can add up to a lot of extra work in a complex program. This list should help you spot excise transgressions:

- Don't force the user to go to another window to perform a function that affects this window.

- Don't force the user to remember where he put things in the hierarchical file system.

- Don't force the user to resize windows. When a child window pops up on

the screen, the program should size it appropriately for its contents. Don't make it big and empty or so small that it requires constant scrolling.

● Don't force the user to move windows. If there is open space on the desktop, put the program there instead of directly over some other already-open program.

● Don't force the user to reenter his personal settings. If he has ever set a font, a color, an indentation or a sound, make sure that he doesn't have to do it again unless he changes his mind.

● Don't force the user to fill fields to satisfy some arbitrary measure of completeness. If the user wants to omit some details from the transaction entry screen, don't force him to enter them. Assume that he has a good reason for not entering them. The completeness of some database isn't worth badgering the user.

● Don't force the user to ask permission to ask a question. This is frequently a symptom of not allowing input in the same place as output.

● Don't ask the user to confirm his actions.

● Don't let the user's actions result in an error.

Idiocy

In Chapter 11, I introduced the concept of flow, where the user enters a very productive mental state by working in harmony with his software tools. Flow is a natural state, and people will enter it without much prodding. Actually, it takes some effort to break into flow once someone is there. Interruptions like

Don't stop the proceedings with idiocy

a ringing telephone will do it, as will an error message box. Some interruptions are unavoidable, but most others are easily dispensable. For a program to include the dispensable is unforgivable.

Software can behave with awesome idiocy. It never fails to astonish me how stupidly software can behave. It will ask questions and make assertions that no self-respecting individual would ever make. And worse, the software does it with a puffed-up self-righteousness unheard of in other media. It will state unequivocally, for example, that a file doesn't exist, merely because it is too stupid to look in the right place for it, then will implicitly blame *you* for losing it! Programs will state categorically that your computer is out of memory an hour after you have beefed it up to 32 MB! A program will go catatonic executing an impossible query that will execute for either another 4.3 million years or until you reboot, whichever comes first. I call behavior like this "stopping the proceedings with idiocy," giving rise to an important axiom: "Don't stop the proceedings with idiocy."

Stopping the proceedings

You might think that "idiocy" is too harsh a word, but it isn't. Let's look at some examples. Figure 13-1 is a good place to start.
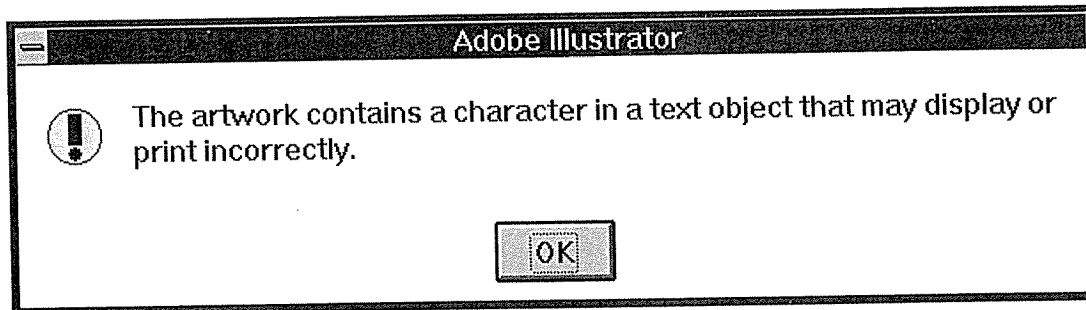

Figure 13-1

This is a totally useless error message box that stops the proceedings with idiocy. We can't verify or identify what it tells us and it gives us no options for responding other than to admit our own culpability with the OK button. This message only comes up when the program is loading; that is, when we have entrusted it to do something simple and straightforward for us. The program is so stupid that it can't even fetch a file (whose name we tell it) without help. If this program were a clerk or secretary, we'd fire it on the spot!

The typical error message box is unnecessary. It either tells the user something that he doesn't care about or demands that he fix some situation that the program could usually fix just as well. Figure 13-1 shows an error message box

displayed by Adobe Illustrator (Version 4) while trying to load an image that was previously drawn with it. The message stops an already annoying and time-consuming procedure, making it take even longer. The user cannot reliably fetch a cup of coffee after telling the program to load an image, because he might return only to see the function incomplete and the program mindlessly holding up the process merely to state the ridiculous. If the issue were data integrity, why didn't the program point out the problematic character when it was entered? If the issue were important, why doesn't the program show where the offending character is? If the issue were a true inability of the program to handle the bad character, why doesn't it just remove it or replace it with a valid character?

Not to pick on Illustrator, but as Adobe was kind enough to provide us with numerous fine examples of really dunderheaded software design, it seems only appropriate that we examine them in detail. Here's how to get the "artwork versus printer format" dialog shown in Figure 13-2.
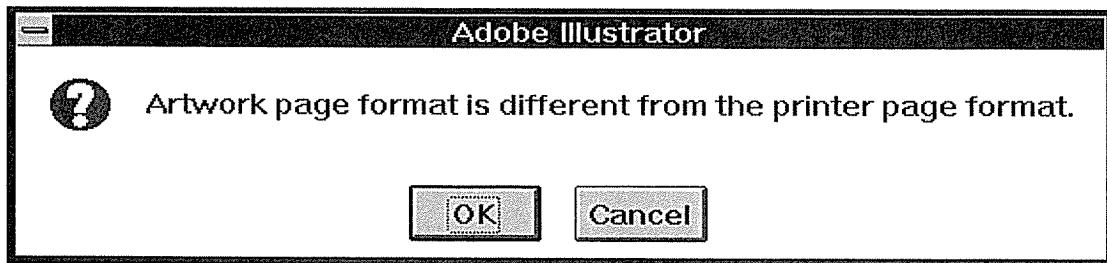

Figure 13-2

Here is another totally useless error message box that stops the proceedings with idiocy. If the program is smart enough to detect the difference, why can't it correct the problem? Why would I ever want to print with conflicting formats? The two options are insulting, telling me that I can either go ahead and shoot my dog, or admit that I shouldn't be carrying a gun. And am I canceling this warning box, or the print operation? Will it still go ahead and kill my dog? What were they thinking?

An image in Illustrator has an "orientation": landscape-oriented is wider than it is tall, and portrait-oriented is taller than it is wide. The printer can be set to either orientation, too. Most programs change the printer's orientation to match that of the image being printed, but not Illustrator. Adobe's product lets you set the orientation of your picture, but it demands that you explicitly set the orientation of the printer independently. This wouldn't be so bad if the

program would at least use the image's orientation as a default setting for the printer. Instead, the program ignores the printer's setting completely. The result is that you have to configure both the orientation of the image and the orientation of the printer, and if you get them wrong, the program stops the proceedings with the idiocy shown in Figure 13-2.

There is one other problem, though. Logically, the place that you define the format of the drawing is connected to the creation of the drawing, but, oddly, the only place in the program to set the drawing's orientation is in the "*Print Setup*" dialog box, shown in Figure 13-3. Of course, the format is important when it comes to printing, but couldn't the print procedure take its cue from the picture and make the appropriate setting before it begins to print? That is, if the drawing is in landscape orientation, print it in landscape orientation, and if the drawing is in portrait orientation, print it in portrait orientation.
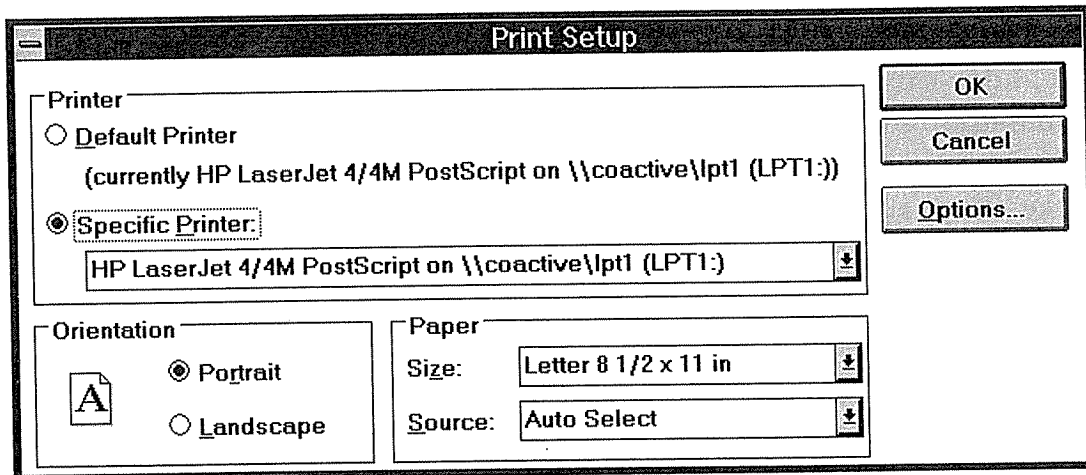

Figure 13-3

The only place to tell Adobe Illustrator what format of paper to use is in the Print Setup dialog box. The program subsequently remembers the format for drawing purposes but forgets it for printing purposes! Call me crazy, but the format of the electronic page has a lot to do with the way things are going to print out. Why would the program forget this connection? I'm used to bossing computers around, but what about the new user who is timid around computers? This program will crush his ego. It would be embarrassingly simple for the program merely to put the printer in whatever mode suits the document and finesse the entire problem.

If, for some truly bizarre reason, the user wanted to print a landscape-oriented drawing on a portrait-oriented piece of paper, he could then go to the Print Setup dialog box and request the desired specification. However, I can pretty

much guarantee Adobe that this will rarely happen. Very few of their users will want to take such unusual actions, so why should all of the normal print jobs be interrupted and delayed by an obscure possibility? Sounds like a good example of putting might on will, to me.

Even if Adobe insists on keeping the silly error message box, why don't they add a button to it labeled "Change to Landscape" or "Change to Portrait" as the case dictates? This would help the user rather than just irritating him.

Protecting me from myself

Another place where the proceedings get stopped with regular idiocy is in password-protection systems. Every morning, I boot up my computer, then go fetch a cup of coffee. When I return, instead of finding the computer waiting eagerly for my instructions, it is pouting and surly, demanding that I log on with my password.

Security is a big issue in many businesses, but it doesn't mean much around here. I don't use a password to protect my computer, yet there is no way (that *I* can find) to turn off the dialog box. I wish it knew how foolish it looked, asking me for my password every morning, only to be summarily dismissed with a stroke. Why can't I make this proceedings-stopping dialog box go away? Why isn't it smart enough to see that it is neither required nor wanted?

Getting stupid

It may seem tautological, but a good way to keep your program from stopping the proceedings with idiocy is for it to not act stupid. A program gets stupid when it becomes deaf, dumb and unresponsive while going off and computing for long periods of time.

Typically, programs with a proclivity for getting stupid are those that talk to remote devices, like servers, printers, networks and modems. Don't discount the ingenuity of programmers, however; they can make a spreadsheet or accounting program get stupid by going into some kind of internal loop—what programmers call "corebound." Every program that executes potentially time-consuming tasks must make sure that it occasionally checks to see if the user is still out there, banging away on the keyboard or madly clicking on the mouse, whimpering "No, no, no, I didn't mean to reorganize the *entire* database. That will take 4.3 million years!"

The Secret Weapon
of Interface Design

If your program could predict what the user will do next, could it provide a better interaction? If your program could know which selections the user will make in a particular dialog box, couldn't that dialog box be skipped? Wouldn't you consider advance knowledge of what actions your user will take to be an awesome secret weapon of interface design?

Well, I'm here to tell you that you *can* predict what your users will do. You *can* build a sixth sense into your program that will tell it with uncanny accuracy exactly what the user will do next!

Get a memory

All you have to do is give your program a memory! I'm not talking about RAM here, but a memory like that of a human being. If your program simply remembers what the user did the last time, it can use that remembered behavior as a guide to how it should behave the next time. Actually, as we'll see later in this chapter, your program should remember even more than just one previous choice. This simple principle is one of the most effective tools available for designing the interaction, yet it is arguably the most untapped resource available.

183

You might think that bothering with a memory isn't necessary; it's easier to just ask the user. Programmers are quick to pop up a simple dialog box to request some bit of information that isn't lying conveniently around. They see nothing wrong with it, but *people don't like to be asked questions*. You know that old adage "the customer is always right?" Well, in the information age, your user is your customer. The user is always right, and asking him questions is a subtle way of expressing doubt about his authority.

Questions aren't choices

Asking questions is quite different from offering choices. The difference between them is the same as that between browsing in a store and having a job interview. The individual asking the questions is understood to be in a position superior to the individual being asked. Bosses ask their subordinates questions and the underlings respond. Judges ask defendants questions and they must respond. Parents ask their children questions and they must answer truthfully. Asking users questions makes them feel inferior and is a sure way to put them down.

Dialog boxes ask questions. Buttcons on a toolbar offer choices. The dialog box stops the proceedings, demanding an answer, and it won't leave until it gets what it wants. The buttcons, on the other hand, are always there, quietly and politely offering up their wares like a well-appointed store, offering you the luxury of selecting what you would like with just a flick of your finger.

Contrary to what many software developers think, questions and choices don't necessarily make the user feel empowered. More commonly, it makes the user feel badgered and harassed. *Would you like soup or salad?* Salad. *Would you like cole slaw or green?* Green. *Would you like French, Thousand Island or Italian?* French. *Would you like lo-cal or regular?* Stop! Just bring me soup! *Would you like chowder or chicken noodle?*

Users don't like to be asked questions. It reflects poorly on the program doing the asking by showing it to be

- Ignorant

- Forgetful

- Weak

- Lacking initiative

- Unable to fend for itself

- Fretful
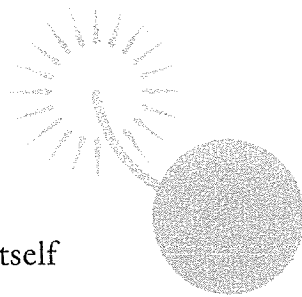
These are qualities that we typically dislike in people. Why should we not dislike them in software? The program is not asking us our opinion out of intellectual curiosity, the way a friend might, say, over dinner. The program is asking us because it is stupid. I use the word "stupid" quite deliberately. (If it was a person, I wouldn't use that word so as not to hurt his feelings. But a program has no feelings, so I'm not going to pull my punch.) The program isn't interested in our opinion; it needs a fact, and chances are it didn't really need to ask the user to get it. It was just too stupid to know where to look for it. That is stupid behavior.

Software that asks fewer questions appears smarter to the user. The questions someone asks you at a cocktail party may flatter you and seem interesting, but face it, no software is ever going to make social chit-chat with its user. Software can only ask the kind of questions that, if someone asked them of you at a party, would have you making excuses and quickly heading for the dip.

One thing that users hate more than questions is questions that are asked repeatedly and unnecessarily. Do you want to save that file? Do you want to save that file *now*? Do you *really* want to save that file? Are you *sure* you want to print? Are you sure you want to print on *that* printer? Are you *absolutely* sure you want to print? Help! Somebody stop this stupid software from asking me another dumb, redundant question.

And if the already-irritated user ever fails to know the answer to a question, it also makes him feel stupid. Do you want the professional install or the beginner install? In other words, do you want something you can't handle, or are you just a wimp?

Choices are certainly good things, but there is a big difference between being free to make choices and being offered ultimatums by the program. Instead of being interrogated by the software, users would much rather direct it the way

they direct their automobiles down the street. An automobile offers the user an infinity of choices without once issuing a dialog box. Imagine the scenario in Figure 14-1.

Figure 14-1

Imagine if you had to steer your car by pressing buttons on a dialog box! This will give you some idea of how normal people feel about the dialog boxes on your software. Humbling, isn't it?

Directly manipulating a steering wheel is not only a more appropriate idiom for communicating with your car, but it puts you in the superior position, directing your car where it should go. No user likes to be questioned like a suspect in a lineup, yet that is exactly what our software often demands of us.

Task coherence

The idea that you can predict what a user will do by simply remembering what he did last is based on a simple principle that I call task coherence. The idea is that what we do is generally the same from day to day, and this is not only true about how we brush our teeth and eat our breakfast, but also about how we use our word processors and email programs. Although Sally, for example, may use Excel in dramatically different ways than Kazu, Sally will tend to use it the same way each time she does. Although Kazu likes 9-point Times Roman and Sally prefers 12-point Helvetica, Sally will use 12-point Helvetica with dependable regularity. It isn't really necessary to ask Sally which font to use. A very reliable starting point would be 12-point Helvetica.

I devised the task coherence term by borrowing from the pioneers of computer graphics. They observed an interesting phenomenon that let them optimize their graphic computations to speed up their displays. A computer screen is composed of a "raster" of several hundred parallel horizontal lines of individual pixels. Graphics software must calculate the color value of each pixel.

Although the whole image may be changing rapidly and dramatically, the pixels in any two adjacent lines don't change very much from moment to moment. The likelihood that any given pixel will have the same value as its neighbor in the line above it is extremely high. Thus, by tracking only the changes in each line, 80 out of a 100 pixels don't have to be recalculated, because their values remain the same from line to line. This phenomenon is called "edge coherence," and it remains a fundamental optimization for graphics programming.

When we apply the task coherence principle to our software, we can realize great advantage from it. When a consumer uses your program, there is a very high-percentage chance that what he does will be the same as what he did the last time he used your program. With significant reliability, you can predict the behavior of your users by the simple expedient of remembering what they did the last time they used the program. This allows you to greatly reduce the number of questions your program asks the user.

We would all like to have an assistant who is intelligent and self-motivated—one who shows initiative and drive, who demonstrates good judgment and a keen memory. When we tell our assistant to fetch a document, we want that assistant to remember where he found it the last time we sent him off to get it. A program that makes effective use of its memory would be more like that self-motivated assistant. It would remember the settings the user specified from execution to execution. Simple things can make a big difference: The position of windows, particularly MDI children, should be remembered, so if I maximized the document last time, it should be maximized the next time. If I tiled it vertically with the on-screen window, it would be tiled vertically next time without any instruction from me.

The way to determine what information the program should remember is with a simple axiom:
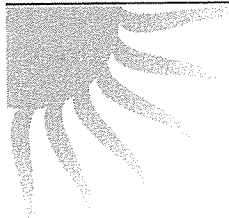
If it's worth asking the user, it's worth the program remembering

Any time your program finds itself with a choice, and especially when that choice is being offered to the user, the program should remember the

information from run to run. Instead of choosing a hard-wired default, the program can use the previous setting as the default, and it will have a much better chance of giving the user what he wanted. Instead of asking the user to make a determination, the program should go ahead and make the same determination the user made last time, and let the user undo it if it was wrong. Whatever options the user had set should be remembered, so that they could remain in effect until manually changed. If the user ignored facilities of the program or turned them off, they should not be offered to the user again. The user will seek them out when and if he is ready for them.

One of the most annoying effects of programs without memories is that they are so parsimonious with their assistance regarding files and disks. If there is one place where the user needs help, it's with files and disks. A program like Word remembers the last place the user looked for a file. Unfortunately, if the user always puts his files in a directory called LETTERS, then once edits a document template stored in the TEMPLATE directory, all his subsequent letters will be stored in the TEMPLATE directory rather than in the LETTERS directory. So the program must remember more than just the last place the files were accessed. It must remember the last place files *of each type* were accessed. On my computer, all template files—files with a suffix of .DOT—are stored in the template directory. Various other documents—ones with a suffix of .DOC—are stored in various other directories. When I'm editing a template file, there is no reason for the word processor to ever even suspect that it will go anywhere other than my template directory. Although this is a convention I never violate on my computer, the software takes no notice of the pattern and refuses to alter its behavior one iota. I always must explain the difference to the program. I should never have to step through the tree to a given directory more than once.

The user can benefit in several ways from a program with a good memory. Memory reduces excise, the useless effort that must be devoted to managing the tool instead of doing the work. A significant portion of the total excise of an interface is in having to explain things to the program that it should already know. For example, in my word processor, I often want to reverse-out text, making it white on black. To do this, I select some text and change the font color to white. Without altering the selection, I then set the background color to black. If the program paid enough attention, it would notice the fact that I requested two formatting steps without an intervening selection option. As far as a user is concerned, this is effectively a single operation. Wouldn't it be nice if the program, upon seeing this unique pattern, automatically created a new

format style of this type, or better yet, created a new "reverse-out" toolbar buttcon?

Most mainstream programs allow their users to set defaults, but this doesn't fit the bill like a memory would. I have Microsoft Word thoroughly configured for my preferences, but a colleague of mine uses Word only occasionally and doesn't have the inclination to learn how to customize it. Every time she runs the program, though, she must manually change the font to her preferred one. If the program only remembered her actions, it would make that maddening step unnecessary.

Most of our software is incredibly forgetful, remembering little or nothing from execution to execution. If our programs are smart enough to retain information, it is usually information that makes the job easier for the *programmer* and not for the user. The program willingly discards information about the way it was used, how it was changed, where it was used, what data it processed, who used it, and whether and how frequently the various facilities of the program were used. Meanwhile, the program fills INI files with driver names, port assignments and OLE details that ease the programmer's job.

Another flagrant violator of the rules of the get-a-memory-club are dialog boxes. These modal monsters almost never remember anything from instantiation to instantiation. They don't remember where they were placed, what they did, what settings were made and what parts were untouched.

A program with a better memory can reduce the number of errors the user makes. This is true simply because the user has to enter less information. More of it will be entered automatically from the program's memory. In an invoicing program, for example, if the software enters the date, department number and other standard fields from memory, the user has less opportunity to make typing errors in these fields.

If the program remembers what the user enters and uses that information for future reasonableness checks, the program can work to keep erroneous data from being entered. Imagine a data-entry program where zip codes and city names are remembered from run to run. When the user enters a familiar city name along with an unfamiliar zip code, the field can turn yellow, indicating uncertainty about the match. And when the user enters a familiar city name with a zip code already associated with another city, the field can turn pink, indicating a more serious ambiguity. He wouldn't necessarily have to take any action because of these colors, but the warning is there if he wants it.

When the user has to tell the program about excise trivia or explain to it information that he explained to it a week ago, his thoughts can be derailed from the real task at hand, distracting him from his real goal by the program's management. He misses his subway stop because he is too busy finding a strap to hold. In the time it takes to enter the correct date into the invoice, the user can forget the meaning of the invoice.

Task coherence predicts what the user will do in the future with reasonable, but not absolute, certainty. If our program relies on this principle, it's natural to wonder about the uncertainty of our predictions. If we can reliably predict what the user will do 80% of the time, it means that 20% of the time we will be wrong. It might seem that the proper step to take here is to offer the user a choice, but this puts us right back at square one. Rather than offering a choice, the program should go ahead and do what it thinks is most appropriate and allow the user to override or undo it. If the undo facility is sufficiently easy to use and understand, the user won't be bothered by it. After all, he will have to use undo only two times out of ten instead of having to deal with a redundant dialog box eight times out of ten. This is a much better deal for humans.

A new way of thinking

A remarkable thing happens to the software design process once developers accept the power of task coherence. Designers find that their thinking takes on a whole new quality. The normally unquestioned recourse of popping up a dialog box gets replaced with a more studied process, where the designer asks questions of much greater subtlety. Questions like: How *much* should the program remember? Which aspects should be remembered? Should the program remember more than just the last setting? What constitutes a change in pattern?

They start to imagine situations like this: The user accepts the same date format 50 times in a row, then manually enters a different format once. The next time the user enters a date, which format should the program use? The 50-times format or the more-recent one-time format? How many times must the new format be specified before it becomes the default? Just because there is ambiguity here, the program still shouldn't ask the user. It must use its initiative to make a reasonable decision. The user is free to override the program's decision if it is the wrong one.

I've identified a couple of characteristic patterns in the ways people make choices that can help us resolve these more complex questions about task coherence.

People tend to reduce an infinite set of choices down to a small, finite set of choices. Even when you don't do the exact same thing each time, you will tend to choose your actions from a small, repetitive set of options. I call this principle **decision-set streamlining**.

For example, just because you went shopping at Safeway yesterday doesn't necessarily mean that you will be shopping at Safeway today, too. However, the next time you need groceries, you will probably shop at Safeway again. Or, even though your favorite Chinese restaurant has 250 items on the menu, chances are that you will usually choose from your own personal subset of five or six favorites. Or, although most people drive home from work the exact same way every evening, some people drive home a different way every night. However, these people will choose from a set of four or five different routes that rarely change. Computers, of course, can remember four or five things without breaking a sweat.

Although simply remembering the last action is better than not remembering anything, it can lead to a peculiar pathology if the decision-set consists of precisely two alternating elements. If, for example, I alternately read files from one directory and store them in another, each time the program offers me the last directory, it will be guaranteed to be wrong. The solution is to remember more than just one previous choice.

Decision-set streamlining guides us to the idea that pieces of information the program must remember about the user's choices tend to come in groups. Instead of there being one right way, there will be several options that are all correct. The program should look for more subtle clues to differentiate which one of the small set is correct. For example, if I use a check-writing program to pay my bills, the program will very quickly learn that only six or eight accounts are used regularly. But how can it determine from a given check which of the eight accounts is the most likely for it? If the payees and amounts were remembered on an account-by-account basis, that decision would be easy. Whaddaya know, every time I pay the rent, it is the exact same amount! Same with my car payment. The amount paid to the electric company varies from check to check, but it always stays within 10 or 20 percent of the last check I wrote to them.

The decisions people make tend to fall into two primary categories: important and not important. Usually, any given activity will involve hundreds of decisions, but very few of them are important. All of the rest are insignificant. I call this principle **preference thresholding**.

Once you decide to buy that car, you don't really care who finances it as long as the terms are competitive. Once you decide to buy groceries, the particular checkout aisle you select is not important. Once you decide to ride the Matterhorn, you don't really care which toboggan they seat you in.

Preference thresholding can guide us in our user interface design by showing us that asking the user for successively detailed decisions about a procedure is unnecessary. Once the user asks to print, we don't have to ask him how many copies he wants or whether the image is landscape or portrait. We can make an assumption about these things the first time out and then remember them for all subsequent invocations. If the user wants to change them, he can always request the Printer Options dialog box.

Using preference thresholding, we can easily track which facilities of the program the user likes to adjust and which are set once and ignored. With this knowledge, the program can offer choices where it has a pretty good expectation that the user will want to take control, while simultaneously not bothering the user with decisions he won't be interested in.

Questions like these soon give rise to associated issues like how to inform the user of the assumptions that the program made. If the program saves a changed file without first discussing it with the user, how does it let the user know that it took this action? When programmers and designers begin to ask questions like these, it means that they are beginning to design software for *users* instead of for programmers. These questions are all ways of serving the customer—the user—instead of concentrating on the needs of the programmer. This kind of goal-oriented thinking is bound to not only create better software, but also better software designers.

One of the main reasons our software is often so difficult to use is because its designers have made rational, logical assumptions that, unfortunately, are very wrong. They assumed that the behavior of users is random and unpredictable, and that they must be interrogated to determine the proper course of action. While human behavior certainly isn't deterministic like a digital computer, it is rarely random, and asking questions is predictably unpleasant. The next time you find your program asking your user a question, make it ask itself one instead.

Part IV: The Interaction
Pointing and Clicking

The scientists who invented computers gave us the complex symbology of language as the tool for communicating with software. It has the advantage of precision, but it is far too labor-intensive and error-prone. Pundits outside of the industry—and some inside it who should know better— advocate instead an interface based on speaking to our computers. Anyone with children, however, knows that you teach people by demonstration, not lecture. Words are for discussing actions after they have been taken. It won't be any different with software. The idea of pointing with a mouse or stylus or finger is the right one. Because these actions are more direct, we show the computer what to do, instead of telling it what to do. This is a fundamental truth about interface design, and one that deserves a close look.

Elephants, Mice and Minnies
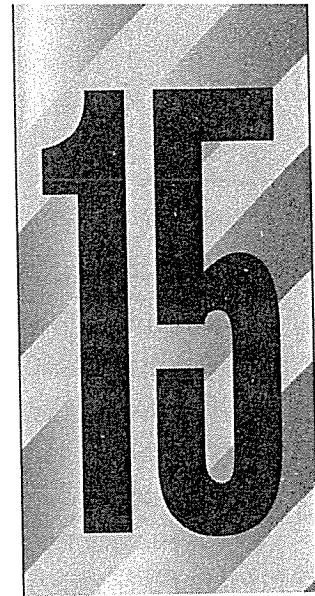
T he best way to point to something is with your finger. They're always handy; you probably have several of these convenient pointing devices nearby right now. The only real drawback they have is that their ends are too blunt for precisely pointing to SVGA screens. Because of this limitation, other pointing devices have taken their place, and each substitute has its own strengths and weaknesses. The mouse is the most omnipresent, but its days are numbered.

Why we use a mouse instead of a pen

The first computer pointing device, the light pen, was a very logical extension of the Mark I Finger. You held the light pen in your hand and pointed it at the screen like a pen. It was the perfect tool for direct manipulation, except for the tragic truth that it was completely unusable with computers.

When we use a stylus, or any other writing device, we exercise extremely fine motor control of our hand muscles to manipulate the tip of the stylus with our fingers. To do this reliably, we have to have something to rest the heel of our

195

hand on; otherwise, our movements are cast adrift. No matter how precise our finger motions are, they drift unless we provide our hand with a firm foundation.

Computers use big, clunky cathode-ray tubes as their display screens, and these CRTs face us vertically rather than lying flat on our desks like books and papers. As easy as it is to use a stylus on a sheet of paper on a firm horizontal surface, it is terribly difficult to make precise movements with that same stylus on a vertical surface with your arm and hand in the air, unsupported. Using a light pen on a CRT squanders the fine motor control of our fingers and forces us to rely on the gross motor control of the muscles in our arms. These muscles are well suited for moving much greater distances, but they cannot give us the precision we expect for accurate pointing.

It is also extremely difficult to draw on a vertical surface while resting the ball of your hand on it—try it on your wall. Your wrist just won't bend backwards far enough. Sign painters, who must paint on the vertical surfaces of walls, doors and windows, frequently use a tool called a mahlstick—a wooden dowel a half-meter long with a padded end. The artist rests the padded end on the wall and holds the other end in her free hand. Then she rests the heel of her drawing hand on the center of the stick. The mahlstick enables her to change the relative incidence of the painting surface from pure vertical to one that is better suited to keeping her drawing hand under control. Unfortunately, a mahlstick is impractical for computer users, so we invented other tools like the mouse.

Indirect manipulation

As we roll the mouse around on our desktop, we see a visual symbol, the cursor, move around on the video screen in the same way. Move the mouse left and the cursor moves left; move the mouse up and the cursor moves up. As you first use the mouse, you immediately get the sensation that the mouse and cursor are connected, a sensation that is extremely easy to learn and equally hard to forget. This is good, because perceiving how the mouse works by inspection is nearly impossible. There is a famous scene in the movie *Star Trek IV: The Voyage Home*, where Scotty comes to twentieth-century Earth and tries to use a computer. He picks up the mouse, holds it to his mouth and speaks into it. This scene is funny because of its underlying truth: the mouse has no visual affordance that it is a pointing device until someone shows us how its movements are related to the movements of the cursor. At that point, though,

understanding is instantaneous. All idioms must be learned. Good idioms need only be learned once, and the mouse is certainly a good idiom.

The motion of the mouse to the cursor is not usually one-to-one, however. Instead the motion is proportional. On most PCs, the cursor crosses an entire 30-centimeter screen in about 4 centimeters of mouse movement. With the heel of your hand resting firmly on the table top, your fingers can move the mouse with great accuracy. The fine motor control of the muscles in your hand enable you to precisely place the cursor, even with a 1:8 movement ratio. Those users who have a difficult time mastering the mouse usually don't place the heel of their palm firmly on their desk.

Although we use the term "direct manipulation" when we talk about pointing and moving things with the mouse, we are actually manipulating these things *indirectly*. A light pen points directly to the screen, and can more properly be called a direct-manipulation tool because we actually point to the object. With the mouse, however, we are only manipulating a mouse on the desk, not the object on the screen.

With a thin-bodied stylus, we can get very precise control of the point, but with the palm-sized mouse, the muscles in our fingertips don't come into play the way they can with a Scripto. This is why we cannot enter handwriting practically with a mouse. Although we utilize fine motor control with a mouse, it is nothing like the extremely detailed control we exercise with the tip of a pen. With our hand wrapped around the much larger mouse, we can easily move the cursor to a particular place, but we cannot effectively define shapes or make the continuous self-relative movements that are required either for cursive or block printing. Thus the mouse is great for pointing at things on the screen but miserable for entering graphical data. The stylus is fine for both tasks.

Mice are not here to stay

The mouse is a clever tool that allows us to point to things on a vertical screen without entangling ourselves with the drawbacks of pointing or drawing on a vertical surface. Don't for a minute imagine that the mouse is a superior tool for anything beyond this, though. In all other ways it is worse. The fact that you can enter cursive handwriting with a pen and that you cannot do so with a mouse should be clue enough that the pen is more accurately manipulable than the mouse. It is only when the writing surface goes vertical that the mouse emerges as the better tool.

When flat-panel displays become cheap and common, they will inevitably migrate down from their vertical perch to a horizontal one, like paper on a desktop. When that happens, the pen input device will have a resurgence of popularity that will ultimately place it at the top of the world of direct-manipulation devices. The mouse will go the way of acoustic modems and 8-inch floppies. This dominance will have nothing whatever to do with handwriting recognition. Instead, it will be based on the way human bodies are constructed and how we can best point to things.

Mousing around

When you mouse around on the screen, there is a distinct dividing line between near motions and far motions. That line is, simply, whether your destination is near enough that you can keep the heel of your hand stationary on your desktop or if you must pick it up. When the heel of your hand is down and you move the cursor from place to place, you use the fine motor skills of the muscles in your fingers. When you lift the heel of your hand from the desktop to make a larger move, you use the gross motor skills of the muscles in your arm. Gross motor skills are no faster or slower than fine motor skills, but transitioning between the two is difficult. It takes both time and concentration, because the user must integrate the two groups of muscles. Touch-typists dislike anything that forces them to move their hands from the home position on the keyboard, because it requires a transition between their muscle groups. For the same reason, moving the mouse cursor across the screen to manipulate a control forces a change from fine to gross to fine motor skills.

Pressing the button on the mouse also requires fine motor control—you use your finger to push it—and if your hand is not firmly planted on the desktop, you cannot press it without inadvertently moving the mouse and the cursor. This means that, while some compromise is possible between fine and gross motor control for the movement aspect of working a mouse, when it comes time to actually press the button—to pull the trigger—the user must first plant the heel of his hand, forcibly going into fine-motor-control mode. To manipulate a gizmo with a mouse, the user must use fine motor control to precisely position the cursor over the checkbox or push-button. However, if the cursor is far away from the desired gizmo, the user must first use gross motor control to move the cursor near the gizmo, then shift to fine motor control to finish the job.

It should be obvious at this point that any program that places its clickable areas more than a few pixels apart is inviting trouble. If a given control demands a click here and then a click waaaay over there, it is a tragic mis-design of the gizmo. Yet the ubiquitous scrollbar is just such a creature. If you are trying to scroll down in a document, you press the down arrow several times, using fine motor control, until you find what you are looking for, but you are likely to press it one too many times and overrun your destination. At this point, you must press the up arrow to get back to where you want to go. Of course, to move the cursor to the up arrow, you must pick up the heel of your hand and make a gross motor movement, then place the heel of your hand back down and make a fine motor movement to precisely locate the arrow and keep the mouse firmly positioned while you press the button.

Why are the arrows on scroll-bars separated by the entire length of the bar itself? Yes, it looks visually bolder and more symmetrical this way, but it is much more difficult to use. If the two arrows were instead placed adjacent to each other at either end of the scroll-bar as shown in Figure 15-1, changing the direction of the scroll could be accomplished by a single fine motor movement, instead of by the difficult dance of fine-gross-fine.
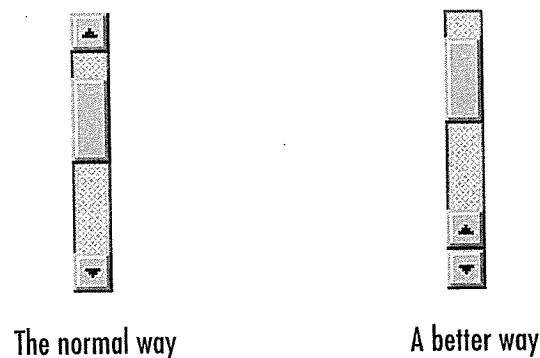

The normal way   A better way

Figure 15-1

The familiar scrollbar, shown on the left, is one of the more difficult-to-use gizmos in Windows. To go from scrolling up to scrolling down, you must transition from the fine motor control required by clicking the button to the gross motor control you need to move your hand to the opposite end of the bar, then change back to fine motor control to accurately position the mouse and press the button again. Bummer. If the scrollbar were modified only slightly, so that the two buttons were adjacent, the problem would go away. The other features, both good and bad, of the scrollbar are discussed in Part VI, "The Gizmos."

Not only do the less-manually dexterous find the mouse problematic, but many experienced computer users, particularly touch-typists, find the mouse difficult at times. For many data-intensive tasks, the keyboard is superior to the mouse. It is frustrating to have to pull your hands away from the keyboard to reposition a cursor with the mouse, only to have to return to the keyboard again. In the early days of personal computing, it was the keyboard or nothing, and today, it is often the mouse or nothing. Programs should fully support both the mouse and the keyboard for all motion and selection tasks.

Some people find it very difficult to manipulate a mouse. Their rodent fear, like that of the pachyderm's, leads me to stick my tongue in my cheek and call these people elephants.

A good percentage of computer users are elephants, so if we want to be successful, we must design our software in sympathy with them. This means that for each mouse-based idiom there should be at least one non-mouse alternative. Of course, this may not always be possible. Some very graphic-oriented actions in a drawing program, for example, would be ridiculous to try to support without a mouse, but these examples are in a clear minority. Most business or personal software lends itself pretty well to keyboard commands. Most users, even elephants, will actually use a combination of mouse and keyboard commands, sometimes starting commands with the mouse and ending them with the keyboard and vice versa.

What do you call a person who is the antithesis of an elephant? Someone who really loves mice? A minnie, of course!

The left mouse button

The inventors of the mouse tried to figure out how many buttons to put on it and couldn't agree. Some said one button was correct, while others swore by two buttons. Still others advocated a mouse with several buttons that could be clicked separately or together so that five buttons could yield up to 32 distinct combinations. I suspect that the actual decisions were made over beers at some long-forgotten sessions at a watering hole somewhere in Silicon Valley. Ultimately, though, Apple settled on one button for their Macintosh, while virtually everybody else agreed on two buttons.

Actually, one of the major drawbacks of the Macintosh is its single-button mouse. I understand that Apple's extensive user testing determined that the optimum number of buttons was one, thereby enshrining the single-button

mouse in the pantheon of Apple history. This is unfortunate, as the right mouse button usually only comes into play when a person has graduated out of beginner-hood. A single button sacrifices power for the majority of computer users in exchange for simplicity for beginners.

There is less difference between the one- and two-button camps than you might think, as the established purpose of the left mouse button is tacitly defined as "the same as the single button on the Macintosh mouse." In other words, the right mouse button is widely regarded as an extra button, and the left button is the only one the user really needs. This statement is certainly true today, although it is gradually becoming less so as the Windows user interface evolves.

In general, the left mouse button is used for all of the major direct-manipulation functions of triggering controls, making selections, drawing, et cetera. By deduction, this means that the functions the left button doesn't support must be the non-major functions. The non-major functions either reside on the right mouse button or are not available by direct manipulation, residing only on menus or the keyboard.

The most common meaning of the left mouse button is activation or selection. For a control such as a push-button or checkbox, the left mouse button means pushing the button or checking the box. If you are left-clicking in data, the left mouse button generally means selecting. We'll discuss this in greater detail in the next chapter.

Right mouse button

The right mouse button was long treated as nonexistent by Microsoft and many others. Only a few brave programmers connected actions to the right mouse button, and they were generally considered to be extra, optional or advanced functions. When Borland International embraced object-orientation on a company-wide basis, they used the right mouse button as a tool for accessing a dialog box that showed an object's properties. The industry seemed ambivalent towards this action although it was, as they say, critically acclaimed. Of course, most usability critics have Macs, which only have one button, and Microsoft disdains Borland, so the concept didn't achieve the popularity it deserved. This is changing, however, with Windows 95, as Microsoft finally follows Borland's lead. The right mouse button is stepping into its best role for enabling direct access to properties as a standard *de jure*. This is, indeed, a pyrrhic victory for Borland.

In Windows 3.x, Microsoft tentatively defined the right mouse button as the "shortcut" button. That is, operations that are also available via other idioms are the only ones allowed on the right mouse button (though they weren't shy about breaking their own rule). This stemmed from Microsoft's assumption that one-button mice would have a role to play on the PC. Since this latter assumption has proven to be baseless, Microsoft has restated their position in the Windows 95 style guide, attributing to the right button "context-specific actions" (a clever way to say "properties").

Middle mouse button

Although application vendors can confidently expect a right mouse button, they can't depend on the presence of a middle mouse button. Because of this, no vendor can use the button as anything other than a shortcut. In fact, in its style guide, Microsoft states that the middle button "should be assigned to operations or functions already in the interface," a definition they once reserved for the right mouse button. I agree.

I use a two-button mouse most of the time, but my other computer has a three-button Logitech mouse. I never find myself reaching for extra functionality on the middle button. I have some friends who do use the middle button. Actually, they swear by it. Mostly, they use it as a shortcut for double-clicking with the left mouse button—a feature they create by configuring the mouse driver software, and of which trickery the application remains blissfully ignorant.

Things you can do with a mouse

Physically, there aren't a lot of things that you can do with a mouse. You can move it around to point to different things and press the buttons. These are the primitives of the vocabulary as discussed in Chapter 4. Any further mouse actions beyond pointing and clicking will be made up of a combination of one or more of those actions, called compounds. The vocabulary of mouse actions is canonically formed, and this is a significant reason why mice make such good computer peripherals.

Mouse actions can also be altered by using the meta-keys: CTRL, SHIFT and ALT. We will discuss these keys later in this chapter. The complete set of mouse actions that can be accomplished without using meta-keys is summarized in the following list. For the sake of discussion, I have assigned a short name to each of the actions (shown in parenthesis). These names may not be standard (what is?), but they are brief and unique.

1. Point (Point)

2. Point, click, release (Click)

3. Point, click, drag, release (Click-and-drag)

4. Point, click, release, click, release (Double-click)

5. Point, click, click other button, release, release (Chord-click)

6. Point, click, release, click, release, click, release (Triple-click)

7. Point, click, release, click, drag, release (Double-drag)

Of course, each of these actions (except chord-clicking, of course) can be performed on either button of a two-button mouse. It's theoretically possible to quadruple-click, quintuple-click, and so on, but even triple-clicking takes a steady and practiced hand, and just trying to double-click can often demoralize an elephant.

Any self-respecting minnie will easily perform all seven actions, while only the first five items on the list are within the scope of normal users. Of these, only the first three can be considered reasonable actions for elephants. Windows 95 is designed to be 100% workable with only the first three actions. Of course, to avoid double-clicking, the user of Windows 95 may have to take circuitous routes to perform their desired tasks, but at least the access is possible.

Pointing

This simple operation is a cornerstone of the graphical user interface and is the basis for all mouse operations. The user moves the mouse until its corresponding on-screen cursor is pointing to, or placed over, the desired object.

Clicking

While the user holds the mouse in a steady position, he clicks the button down and releases it. In general, this action is defined as triggering a state change in a gizmo, or selecting an object. In a matrix of text or cells, the click means "bring the selection point over here." For a push-button gizmo, a state change means that while the mouse button is down and directly over the gizmo, the button will enter and remain in the pushed state. When the mouse button is released, the button is triggered, and its associated action occurs.

Design tip: Single-click selects data or changes the gizmo state.

If, however, the user, while still holding the mouse button down, moves the cursor off the gizmo, the push-button gizmo returns to its unpushed state. When the user then releases the mouse button, nothing happens. This provides a convenient escape route if the user changes his mind.

The drawback to this escape route is that it consumes one of the cooler idioms: dragging a push-button somewhere. The button or buttcon could be dragable, so an idiom could be created that would allow a verb-object grammar in addition to the normal object-verb form. For example, the user could click on the Justified Text buttcon in Word and drag it onto a paragraph of text. The margins of the paragraph would immediately change to justified. Instead of the user having to select the paragraph and then press the buttcon, the user would have the freedom to do the operation in reverse.

Clicking-and-dragging

This versatile operation has many common uses including selecting, reshaping, repositioning, drawing and dragging-and-dropping. We'll discuss all of these in the remaining chapters of this part.

Double-clicking

If double-clicking is composed of single-clicking twice, then it seems logical that the first thing a double-click should do is the same thing that a single-click does. This is indeed its meaning when the mouse is pointing into data. Single-clicking selects something; double-clicking selects something and then takes action on it.

Design tip: Double-click means a single-click plus action.

This fundamental interpretation comes from the Alto/Star by way of the Macintosh, and it remains a standard in all contemporary GUI applications. The fact that double-clicking is difficult for elephants—painful for some and impossible for a few—was largely ignored. But since Microsoft has embraced user testing, they have had to confront this awful truth. Despite mixed feelings, the double-click has assumed a significantly diminished role in Windows 95. I, too, have very mixed feelings about this role reduction for double-clicking. While a significant number of users are undoubtedly elephants, the majority of

users have no trouble double-clicking and working comfortably with the mouse. We should not penalize the majority for the limitations of the elephants. The answer is to go ahead and include double-click idioms, while assuring that their functions have corresponding single-click idioms.

While double-clicking on data is well-defined, double-clicking on most gizmos has no meaning (I class icons as data, not gizmos), and the extra click is discarded. Many gizmos don't discard the extra click but just ignore it. If the gizmo stays in place, it will be interpreted as a second click on it. Depending on the gizmo, this can be benign or problematic. If the gizmo is a toggle button, you may find that you've just returned it to the state it started in (rapidly turning it on, then off). If the gizmo is one that goes away after the first click, like the OK button in a dialog box, for example, the results can be quite unpredictable—whatever was directly below the push-button gets the second button-down message.

Chord-clicking

Chord-clicking means pressing two buttons simultaneously, although they don't really have to be either pressed or released at precisely the same time. To qualify as a chord-click, the second mouse button must be pressed at some point before the first mouse button is released.

There are two variants to chord-clicking. The first is the simplest, whereby the user merely points to something and presses both buttons at the same time. This idiom is very clumsy and has not found much currency in existing software, although some creatively desperate programmers have implemented it as a substitute for a shift key on selection.

The second variant is using chord-clicking to terminate a drag. The drag begins as a simple, one-button drag; then the user adds the second button. Although this technique sounds more obscure than the first variant, it actually has found wider acceptance in the industry, and it is one of my personal favorites, because it is perfectly suited for canceling drag operations. I'll discuss it in more detail in the next chapter.

Triple-clicking

Believe it or not, some otherwise-respectable programs have actions that involve triple-clicking. Triple-clicking can challenge even those minnies with a high level of manual dexterity. In Word, triple-clicking is used to select entire

paragraphs. The logic is simple: a single-click selects a character; a double-click selects a word; a triple-click selects a paragraph. But this idiom is so difficult to perform reliably, let alone to communicate to the user, that it is only useful for those who spend the majority of their time using the program (like authors of user interface design books).

For horizontal, sovereign applications with extremely broad user populations, like word processors and spreadsheets, triple-clicking can be worth implementing. For any program used less frequently than several hours each day, it is silly. In any case, by the time you resort to such an idiom, you should also have provided the user with several other methods of accomplishing the same task. To select a paragraph in Word without triple-clicking, for example, you can

- Double-click in the left-hand margin

- Click in the left-hand margin beside the first line and drag down to the last line

- Click at the beginning of the first word and drag to the end of the last word

- Click at the beginning of the first word and CTRL-SHIFT-RIGHT-ARROW until all of the words in the paragraph are selected

- Click at the beginning of the first word and CTRL-SHIFT-PAGEDOWN

- Double-click anywhere in the first word and drag to the end of the last word

Double-dragging

Double-dragging is another minnie-only idiom. Faultlessly executing a double-click-and-drag can be like patting your head and rubbing your stomach at the same time. Like triple-clicking, it is useful only in mainstream, horizontal sovereign applications. Use it as a variant of selection extension.

I use double-dragging in Word all of the time as a selection tool. You can double-click in text to select an entire word, so, expanding that function, you can extend the selection word-by-word by double-dragging. When I want to delete a phrase from the middle of a sentence, for example, I double-click in the middle of the first word, then drag until the phrase is selected. This is hard to do,

and Microsoft has added a feature to Word that *automatically* extends your selection to word boundaries by default. Evidently, some subjects in their user testing liked this feature. I don't, so I turn it off and tolerate double-dragging.

In a big, horizontal, sovereign application that has many permutations of selection, idioms like this one are appropriate. But unless you are creating such a monster, I suggest you stick with the basic mouse actions.

Up and down events

Each time the user presses a mouse button, the program must deal with two discrete events: the button-down event and the button-up event. With the bold lack of consistency exhibited elsewhere in the world of mouse management, the definitions of the actions to be taken on button-down and button-up can vary with the context and from program to program. These actions should be made rigidly consistent.

When you're selecting an object, the selection should always take place on the button-down. This is so because the button-down may be the first step in a dragging sequence. By definition, you cannot drag something without first selecting it, so the selection *must* take place on the button-down. If not, the user would have to perform the demanding double-drag.

> **Design tip:** Button-down means select over data.

On the other hand, if the cursor is positioned over a gizmo rather than selectable data, the action on the button-down event is to *tentatively* activate the gizmo's state transition. When the gizmo finally sees the button-up event, it then commits to the state transition.

> **Design tip:** Button-down means propose action; button-up means commit to action over gizmos.

This is the mechanism that allows the user to gracefully bow out of an inadvertent click. In a push-button, for example, the user can just move the mouse outside of the button and the selection is deactivated even though the mouse button is still down. For a checkbox, the meaning is similar: on button-down, the checkbox visually shows that it has been activated, but the check doesn't actually appear until the button-up transition.
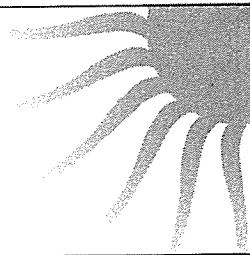
The cursor

The cursor is the visible representation on the screen of the mouse's position. By convention, it is normally a small arrow pointing slightly West of North, but under program control it can change to any shape as long as it stays relatively small: 32×32 pixels. Because the cursor frequently must resolve to a single pixel—pointing to things that may occupy only a single pixel—there must be some way for the cursor to indicate precisely which pixel is the one pointed to. This is accomplished by always designating one single pixel of any cursor as the actual locus of pointing, called the hotspot. For the standard arrow, the hotspot is, logically, the tip of the arrow. Regardless of the shape the cursor assumes, it always has a single hotspot pixel.

As you move the mouse across the screen, some things that the mouse points to are inert: clicking the mouse button while the cursor's hotspot is over them provokes no reaction. Other, more interesting things, react when you click on them. Any object or area on the screen that reacts to a mouse action I call pliant. A push-button gizmo is pliant because it can be "pushed" by the mouse cursor. Any object that can be picked up and dragged is pliant, thus any directory or file icon in the File Manager or Explorer is pliant. In fact, every cell in a spreadsheet and every character in text is pliant.

When objects on the screen are pliant, this fact must be communicated to the user. If this fact isn't made clear, the idiom ceases to be useful to any user other than experts (conceivably, this could be useful, but in general, the more information we can communicate to the user the better).

Visually hint at pliancy

Hinting

There are three basic ways to communicate the pliancy of an object to the user: by the static visual affordances of the object itself, its dynamically changing visual affordances, or by changing the visual affordances of the cursor as it passes over the object. If the pliancy of the object is communicated by the static visual affordance of the object itself, I call that static visual hinting.

Static visual hinting merely indicates the way the object is drawn on the screen. For example, the three-dimensional sculpting of a push-button is static visual hinting because of its manual affordance for pushing.

Some visual objects that are pliant are not obviously so, either because they are too small or because they are hidden. If the directly manipulable object is out of the central area of the program's face, the side posts, scrollbars or status bar at the bottom of the screen, for example, the user simply may not understand that the object is directly manipulable. This case calls for more aggressive visual hinting, which I call active visual hinting.

It works like this: When the cursor passes over the pliant object, the object changes its appearance with an animated motion. Remember, this action occurs merely when the cursor passes over the object, before any mouse buttons are pressed. LucasArts' X-Wing does this with great wit and panache: When the cursor passes over the doors to different elements of the program—the missions, training, or briefing room—the door itself slides smoothly upward with a sibilant pneumatic hiss to reveal the room beyond. Active visual hinting at this level is powerful enough to act as a training device, in addition to merely reminding the user of where the pliant spots are. I'm not suggesting that a business program be as bold as an arcade-game style program, but a more subtle implementation of active visual hinting could be just the ticket for bringing an important but latent idiom to the user's attention. There is remarkably little active visual hinting in the world of business and productivity software. Too bad. The edutainment field uses it often, and their software is better for it.

Cursor hinting

If the pliancy of the object is communicated by a change in the cursor as it passes over, I call that cursor hinting.

Because the cursor is dynamically changing, all cursor hinting is *active* cursor hinting.

Most popular software intermixes visual hinting and cursor hinting freely, and we think nothing of it. For example, push-buttons are rendered three-dimensionally, and the shading clearly indicates that the object is raised and affords to be pushed; when the cursor passes over the raised button, however, it doesn't change. On the other hand, when the cursor passes over a pluralized window's thickframe, the cursor changes to a double-ended arrow showing the axis in which the window edge can be stretched. This is the only definite visual

affordance that the thickframe can be stretched. In Windows 3.x, the thick-frame is a visually distinct area, so it has some visual hinting, but in the redesigned Windows 95, that hinting is attenuated significantly.

In a broad generalization, gizmos usually offer static visual hinting, while pliant data more frequently offers cursor hinting. We'll talk more about hinting in Chapter 18, "Drag-and-Drop."

> *Design tip:* Indicating pliancy is the most important role of cursor hinting.

Although cursor hinting usually involves changing the cursor to some shape that indicates what type of direct-manipulation action is acceptable, its most important role is in making it clear to the user that the object is pliant. It is difficult to make data visually hint at its pliancy without disturbing its normal representation, so cursor hinting is the most effective method. Some gizmos are small and difficult for users to spot as readily as a button or buttcon, and cursor hinting is vital for the success of such gizmos. The column dividers and screen splitters in Microsoft's Excel are good examples, as you can see in Figure 15-2.

Wait cursor hinting

Actually, there is a third variant of cursor hinting, called **wait cursor hinting**. Whenever the program is doing something that takes significant amounts of time in human terms—like accessing the disk or rebuilding directories—the program changes the cursor into a visual indication that the program has gone stupid. In Windows, this image is the familiar hourglass. Other operating systems have used wristwatches, spinning balls and steaming cups of coffee. Informing the user when the program becomes stupid is a good idea, but the cursor isn't the right tool for the job. After all, the cursor belongs to everybody, and not to any particular program. Too bad the idiom has wide currency as a standard and will undoubtedly live on for many years.

The user interface problem arises because the cursor belongs to the system and is just "borrowed" by a program when it invades that program's airspace. In a non-preemptive system like Windows 3.x, using the cursor to indicate the wait is a reasonable idiom because when one program gets stupid, they all get stupid.
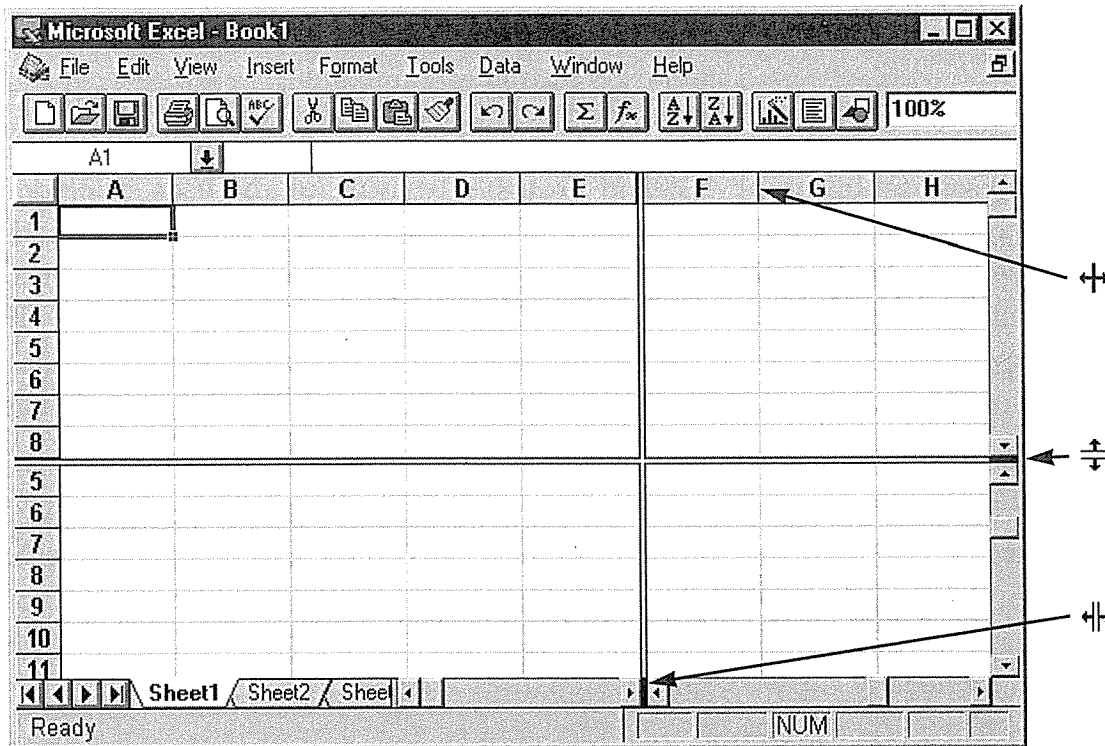
Figure 15-2

Microsoft Excel for Windows uses cursor hinting to highlight several gizmos that, by visual inspection, are not obviously pliant. The width of the individual columns can be set by dragging on the short vertical lines between each pair of columns, so the cursor changes to a two-headed horizontal arrow hinting at both the pliancy and indicating the permissible drag direction. The same is true for the two screen-splitter controls. The short, dark lines in the scroll-bars are fatter than those dividing each column, and the corresponding cursor is slightly different, using a double line, but the meaning is substantially the same.

In the *preemptive* multi-tasking world of Windows 95, when one program gets stupid, it won't necessarily make other running programs get stupid, and, if the user points to one of them, it will need to use the cursor. Therefore, the cursor cannot be used to indicate a busy state for any single program.

If the program must turn a blind eye and deaf ear to the user while it scratches some digital itch, it should make this known through an indicator on its own video real estate, leaving the cursor alone. It can graphically indicate the corresponding function and show its progress either on its main window, or in a dialog box that appears for the duration of the procedure. It's also important to offer the user a means to cancel the operation. (We'll discuss this more in

Chapter 21, "Dialog Boxes." Actually, the dialog box is a weaker implementation than drawing the same graphics right on the main window of the program.

Windows 95 forced Microsoft to rethink the wait cursor. Clearly, they wanted to maintain the familiar wait cursor hinting idiom even though the new dispatching algorithm meant it would probably be lying to the user. They decided that programs will only show the hourglass cursor within their own windows, which is logically correct. However, the result of this fix means the program that is busy now offers no visual feedback of its state of stupidity. If the user inadvertently moves the cursor off a busy program's main window and onto that of another—running—program, the cursor will revert to a normal arrow. The visual hinting is all wrong.

I don't believe that Microsoft has solved the problem with their compromise. Ultimately, each program must indicate its busy state by some visual change to its own visage. Using the cursor to indicate a busy state doesn't work if that busy state depends on where the cursor is pointing. Preemptive multi-tasking will kill the idea of the wait cursor. May it rest in peace.

Focus

Focus is an obscure technical state that is so complex it has confounded more than one erstwhile Windows programming expert. One of them, after a particularly grueling and fruitless week of focus programming, declared to me in disgust that focus was actually a contraction of two words, the second one being "us."

Windows is a multi-tasking system, which means that more than one program can be performing useful work at any given time. Despite allegations to the contrary, Windows has *always* been a multi-tasking system; Windows 95 is merely the first version of it that multi-tasks *preemptively*. Regardless of the dispatching algorithm, though, no matter how many programs are running concurrently, only one program can be in direct contact with the user at a time. That is why the concept of focus was derived. Focus indicates which program will receive the next input from the user. For the purposes of our discussion here, we can think of focus as being the same as "activation," as in, there is only one program active at a time. This is purely from the user's point of view. Programmers will generally have to do more homework. The active program is the one with the most prominent caption bar (it's usually dark blue or whatever color you have personalized your desktop to show).

In its simplest case, the program with the focus will receive the next keystroke. Because a normal keystroke has no location component, the focus cannot change because of it, but a mouse button press does have a location component and can cause the focus to change as a side effect of its normal command. I call a mouse click that changes the focus a **new-focus click**.

However, if you click the mouse somewhere in a window that already has the focus, an action that I call an **in-focus click**, there is no change to the focus.

An in-focus click is the normal case, and the program will deal with it as just another mouse click, selecting some data, moving the insertion point or invoking a command. The conundrum arises for the new-focus click: what should the program do with it? Should the program discard it (from a functional standpoint) after it has performed its job of transferring the focus, or should it do double-duty, first transferring the focus and then performing its normal task within the application?

For example, let's assume that both File Manager and Program Manager are pluralized and visible on the screen simultaneously. Only one of them can be active, so let's make it the Program Manager. By definition, if it is active, it has the focus and visibly indicates this with a highlighted caption bar. Pressing keys sends messages only to Program Manager. Mouse clicks inside the already-active Program Manager are in-focus and go only to the Program Manager. Now, if you move the mouse cursor over to the File Manager window and click the mouse, you are telling Windows that you want File Manager to become the active window and take over the focus. This new-focus click causes both caption bars to change color, indicating that the File Manager is active and the Program Manager is inactive. Now the question arises: should the File Manager interpret that new-focus click within its own context? Let's say that new-focus click was on a visible filename. Should that filename also become selected or should the click be discarded after transferring the focus? If File Manager were already active and I in-focus clicked on that same filename, the filename would be selected. As a matter of fact, in real life, the filename *does* get selected. Both File Manager and Program Manager interpret the new-focus click as a valid in-focus click.

Windows interprets new-focus clicks as in-focus clicks with some uniformity. For instance, if I change focus to Word by clicking and dragging on its caption bar, Word not only gets the focus but is repositioned, too. Ah, but here is where it gets sticky! If I change the focus to Word by clicking on a document

inside Word, Word gets the focus, but the click is discarded—it is *not* also interpreted as an in-focus click within the document. Adobe Illustrator for Windows and Microsoft's Excel also discard the new-focus click in this manner.

The Microsoft style guide weighs in on this point to say that "The reactivation of a window or pane does not affect any pre-existing selection there; the selection and focus are restored to the state that existed when the window or pane was last active." But just because it says so in the guide doesn't necessarily mean that Microsoft won't completely refute this in their next release. Nor does the guide blush over the fact that Program Manager, File Manager and the Explorer have blatantly violated the statement since they first shipped.

I suspect that the guide author looked more towards Microsoft's applications for archetypes than towards its operating systems. I know experts who strongly hold contradictory positions on this issue, so neither policy is necessarily "right." Generally, I think ignoring the new-focus click is a safer and more conservative course of action. On the other hand, I am loathe to demand extra clicks from the user. If you do choose to ignore the click, like Word and Excel do, it is difficult to explain the contradiction that a new-focus click in any non-client areas will be also used as an in-focus click, even though it somehow *feels* right.

Meta-keys

Direct-manipulation idioms can be extended by using one of the various meta-keys in conjunction with the mouse. Meta-keys include the CONTROL key, the ALT key and either of the two SHIFT keys.

There is a slightly sacrilegious joke floating around Silicon Valley: God must have loved standards because he gave us so many of them. In the Windows world, no single voice articulated user interface standards with the iron will that Apple did for the Macintosh, and the result was chaos in some important areas. This is certainly evident when we look at meta-key usage. Although Microsoft has finally articulated meta-key standards with Windows 95, their efforts now are about as futile as trying to eliminate kudzu from Alabama roadsides.

Even Microsoft freely violates their own standards for meta-keys. Each program tends to roll its own, but some meanings predominate, usually those that were first firmly defined by Apple. Unfortunately, the mapping isn't exactly the same. Apples have a CLOVER key and an APPLE key that roughly correspond to the CTRL and ALT keys, respectively. Keep in mind that the choice of which

meta-key to use, or which program to model your choices after, is less important than remaining consistent within your own interface.

Meta-key cursor hinting

Using cursor hinting to show the meanings of meta-keys is an all-around good idea and more programs should do it. This is something that must be done dynamically, too. As the meta-key goes down, the cursor should change immediately to reflect the new intention of the idiom.

Design tip: Use cursor hinting to show meta-key meanings.

ALT meta-key

The ALT meta-key is the problem-child of the family. Microsoft has studiously avoided imbuing it with meaning, so it has been rather a rudderless ship adrift in a sea of clever programmers, who use it as the whim strikes and ignore it otherwise. Doubtless, someone at Microsoft will latch onto it for some favorite idiom, and it will then grow into a de facto standard.

At one time, I favored using it to indicate a two-application drag-and-drop operation. However, first the ALT-TAB idiom and now the Startbar have gone a long way to disabuse me of the idea that two-application drag-and-drop will be the Next-Big-Thing. As a result, I'm just as adrift on the ALT key as Microsoft is. I'll probably come to regret that statement.

I will discuss the specific meanings and usage of the CONTROL and SHIFT meta-keys and how they affect selection and drag-and-drop in their respective chapters.

Selection

16
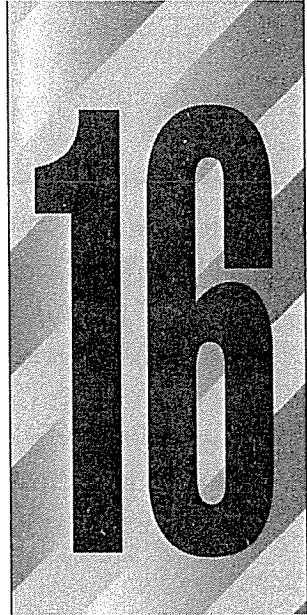
There are basically only two things you can do with a mouse: Choose something, and choose something to do to what you've chosen. These choosing actions are referred to as **selection**, and they have many nuances.

Object-verb

A fundamental issue in user interfaces is the sequence in which commands are issued. Most every command has an operation and one or more operands. The operation describes what action will occur, and the operands are the target of that operation. "Operation" and "operand" are programmer's terms; interface designers prefer to borrow linguistic terminology, referring to the operation as the **verb**, and the operand as the **object**.

You can specify the verb first, followed by the object, or you can specify the object first, followed by the verb. These are commonly called **verb-object** and **object-verb** orders, respectively. Either order is good, and modern user interfaces typically use both.

In the days when language compilers like COBOL and FORTRAN were the bee's knees in high technology, all

217

computer languages used verb-object ordering. A typical statement went like this: PERFORM ACTION ON X AND Y. The verb, PERFORM ACTION, came before the objects, X and Y. This ordering was intended to follow the natural formations of the English language. In the world of linguistic processing, though, this actually wasn't all that convenient, as the computer doesn't like this notation. Compiler-writers put considerable effort into swapping things around, making it easier to turn the human-readable source code into machine-readable executable code. But there was never any question that verb-object ordering was the right way to present things to the user—the programmer—because it was clear and natural and effective for written, text-oriented communications with the computer.

When graphical user interfaces emerged, it became clear that verb-object ordering created a problem. In an interactive interface, if the user chooses a verb, the system must then enter a state—a mode—that differs from the norm: waiting for an object. Normally, the user will then choose an object and all will be well. However, if the user wants to act on more than one object, how does the system know this? It can only know if the user tells it in advance how many operands he will enter, which violates the axiom of not requiring the user to ask permission to ask a question. Otherwise, the program must accept all operands until the user enters some special object-list-termination-command, also a very clumsy idiom. See the problem? What works just fine in a highly structured, linguistic environment falls apart completely in the looser universe of interactivity.

By swapping the command order to object-verb, we don't need all of that complex termination stuff. The user merely selects which objects will be operated upon and then indicates which verb to execute on them. The software very simply executes the indicated function on the selected data. Notice, though, that a new concept has crept into the equation that didn't exist—wasn't needed—in a verb-object world. That new concept is called selection.

Rather than the program remembering the verb while the user specifies one or more objects, we are asking the program to remember one or more objects while the user chooses the verb. This way, however, we need a mechanism for identifying, marking and remembering the chosen operands. Selection is the mechanism by which the user informs the program which objects to remember.

The object-verb model can be difficult to understand intellectually, but selection is an idiom that is very easy to grasp and, once shown, rarely forgotten.

Explained through the linguistic context of the English language, it is nonsensical that we must choose an object first. On the other hand, we use this model frequently in our non-linguistic actions. We purchase groceries by first selecting the objects—by placing them in our shopping cart—then specifying the operation to execute on them—by bringing the cart up to the checkout counter and expressing our desire to purchase. But we never say "Corn flakes, buy" in English conversation.

In a non-interactive interface, like a modal dialog box, the concept of selection isn't always needed. Dialog boxes naturally come with one of those object-list-termination-commands: the OK button. The user can choose a function first and an object second or vice versa because the whole operation won't actually occur until the confirming OK button is pressed. This is not to say that object-verb ordering isn't used in most dialog boxes. It merely shows that no particular command ordering has a divine right; the two orderings have strengths and weaknesses that complement each other in the complex world of user interfaces. Both are powerful tools for the software designer and should be used where they are best suited.

In its simplest variant, selection is trivial: The user points to a data object with the mouse cursor, clicks and the object is selected. However, this operation is deceptively simple and, in practice, many interesting variants are exposed.

Concrete and discrete data

Users select data, not verbs. When you invoke a verb, you may do it with the same type of click action you used to select the data, though, so don't get confused. The basic variants of selection, then, depend on the basic variants of selectable data, and there are two broad categories of data.

Some programs represent data as distinct visual objects that can be manipulated independently of other objects. The icons in the Program Manager and graphic objects in draw programs are examples. These objects are also selected independently of each other. They are discrete data, and I call selection within them discrete selection. Discrete data is not necessarily homogeneous, and discrete selection is not necessarily contiguous.

Conversely, some programs represent their data as a matrix of many little contiguous pieces of data. The text in a word processor or the cells in a spreadsheet

are concretions of hundreds or thousands of similar little objects that together form a coherent whole. These objects are often selected in solid groups, so I call them concrete data and selection within them concrete selection.

Both concrete and discrete selection support both single-click selection and click-and-drag selection. Single clicking selects the smallest possible discrete amount, and clicking-and-dragging selects some larger quantity, but there are significant differences.

The nature of discrete selection is discontiguous, while that of concrete selection is contiguous. I'll show you what I mean: There is a natural order to the text in a word processor's document—concrete data. Scrambling the order of the letters destroys the sense of it. The characters flow from the beginning to the end in a meaningful continuum; selecting a word or paragraph makes sense in the context of the data, while random, disconnected selections are generally meaningless. Although it is theoretically possible to allow a discontiguous selection—several disconnected paragraphs, for example—the user's task of visualizing the selections and avoiding inadvertent, unwanted operations on them is more trouble than it is worth. Generally, if the data can be scrolled off screen, it shouldn't be discontiguously selectable.

Discrete data, on the other hand, has no inherent order; like peas on your plate, the order in which you select and eat them is irrelevant. In a drawing program, where various graphic objects reside on the screen, the objects are independent. No relationship is integral to their meaning, and even the z-order, the order in which they overlay each other on the screen, is only significant if they directly cover each other. Scrambling the order of the objects might have no effect whatsoever on the collective image (again, except where objects overlay each other). Because there is no inherent order in these objects, contiguous selection has no meaning in this context, and each object is selected discretely.

Most drawing programs offer a grouping facility which allows more than one discrete object to be logically grouped together to form a single, new discrete object. That group object now behaves as though it were a single discrete object regardless of the number of component pieces it contains.

Of course, you can always select more than one discrete object, but it remains a series of independent selections rather than as a subset of ordered data.

Insertion and replacement

As we've established, selection indicates which data the next function will operate on. If that next function is a write command, the incoming data (keystrokes

or a PASTE command) writes onto the selected data. In discrete selection, one or more discrete objects are selected, and the incoming data is handed to the selected discrete objects which process them in their own way. This may cause a replacement action, where the incoming data replaces the selected object. Alternatively, the selected object may treat the incoming data as fodder for some standard function. In PowerPoint, for example, incoming keystrokes with a shape selected result in a text annotation of the selected shape.

In concrete selection, however, the incoming data always replaces the currently selected data. In a word processor, when you type, you replace what is selected with what you are typing. Concrete selection exhibits a unique quirk related to insertion, where the selection can shrink down to a single point that indicates a place *in between* two bits of data, rather than one or more bits of data. This in-between place is called the insertion point.

In a word processor, the blinking caret (usually a dark, vertical line indicating where the next character will go) is essentially the least amount of concrete selection available: a location only. It just indicates a position in the data, between two atomic elements, without actually selecting either one of them. By pointing and clicking anywhere else, you can easily move the caret, but if you drag to extend the selection, the blinking caret disappears and is replaced by the contiguous selection.

Another way to think of the insertion point is as a null selection. By definition, typing into a selection replaces that selection with the new characters, but if the selection is null, the new characters replace nothing; they are merely inserted. In other words, insertion is the trivial case of replacement.

Even though spreadsheets use concrete selection, they are different from word processors. The selection is concrete because the cells form a contiguous matrix of data, but there is no concept of selecting the space between two cells. In the spreadsheet, a single-click will select exactly one whole cell. There is currently no concept of an insertion point in a spreadsheet, although the design possibilities are intriguing.

A blend of these two idioms is implementable as well. In PowerPoint's slide-sorter view, insertion-point selection is allowed, but single slides can be selected, too. If you click on a slide, that slide is selected, but if you click in between two slides, a blinking insertion-point caret is placed there.

If a program allows an insertion point as a selection, objects themselves are selected by clicking and dragging across them. Even to select a single character

in a word processor, the mouse must be dragged across it. This means that the user will be doing quite a bit of clicking-and-dragging in the normal course of using the program, with the side effect that any drag-and-drop idiom will be more difficult to express. You can see this in Word, where dragging-and-dropping text involves first a click-and-drag operation to make the selection, then another mouse move back into the selection to click-and-drag again for the actual move. To do the same thing, Excel makes you find a special pliant zone that is only a pixel or two wide on the border of the desired cell. In discrete selection, all the user must do is click-and-drag on the object in a single motion.

To relieve the click-and-drag burden of selection in word processors, other direct-manipulation shortcuts are also implemented, like double-clicking to select a word.

Mutual exclusion

Generally, when a selection is made, any previous selection is unmade. This behavior is called **mutual exclusion**, as the selection of one excludes the selection of the other. Typically, the user clicks on an object, and it becomes selected. That object remains selected until the user selects something else. Mutual exclusion is the rule in both discrete and concrete selection.

Some discrete systems allow a selected object to be deselected by clicking on it a second, canceling, time. This can lead to a curious condition in which nothing at all is selected, and there is no insertion point. You must decide whether this condition is appropriate for your program.

Additive selection

I cannot imagine a concrete-selection program without mutual exclusion, because the user cannot see or know what effect his actions will have if his selections can readily be scrolled off the screen. Imagine being able to select several independent paragraphs of text in a long document. It might be useful, but it certainly isn't controllable. The problem is caused by the scrolling, not the concrete selection, but most programs with concrete-selectable data *are* scrollable.

However, if mutual exclusion is turned off in discrete selection, you have the simple case where many independent objects can be selected merely by clicking on more than one in turn. I call this **additive selection**. A listbox, for example,

can allow the user to make as many selections as desired. An entry is then deselected by clicking it a second time. Once the user has selected the desired objects, the terminating verb acts on them collectively.

Most discrete-selection systems implement mutual exclusion by default and allow additive selection only by using a meta-key. The SHIFT meta-key is used most frequently for this. In a draw program, for example, after you've clicked to select one graphical object, you typically can add another one to your selection by SHIFT-clicking.

Concrete selection systems should never allow additive selection because there should never be more than a single selection in a concrete system. However, concrete-selection systems do need to enable their single allowable selection to be extended, and again, meta-keys are used. Unfortunately, there is little consensus regarding whether it should be the CTRL or the SHIFT key that performs this role. In Word, the SHIFT key causes everything between the initial selection and the SHIFTED-click to be selected. It is easy to find programs with similar additive selection functions that have made different choices of meta-key variations. There is little practical difference between choices, so this is an area where following the market leader is best because it offers the user the small-but-real advantage of consistency.

Group selection

The click-and-drag operation is also the basis for group selection. In a matrix of text or cells, it means "extend the selection" from the mouse-down point to the mouse-up point. This can also be modified with meta-keys. In Word, for example, CTRL-click selects a complete sentence, so a CTRL-drag extends the selection sentence-by-sentence. Sovereign applications should rightly enrich their interaction with as many of these variants as possible. Experienced users will eventually come to memorize and use them, as long as the variants are manually simple.

In a collection of discrete objects, the click-and-drag operation generally begins a drag-and-drop move. If the mouse button is pressed in the open area between objects, rather than on any specific object, however, it has a special meaning. It creates a dragrect, shown in Figure 16-1.

A dragrect is a dynamic gray rectangle whose upper left corner is the mouse-down point and whose lower right corner is the mouse-up point. When the mouse button is released, any and all objects enclosed within the dragrect are selected as a group.
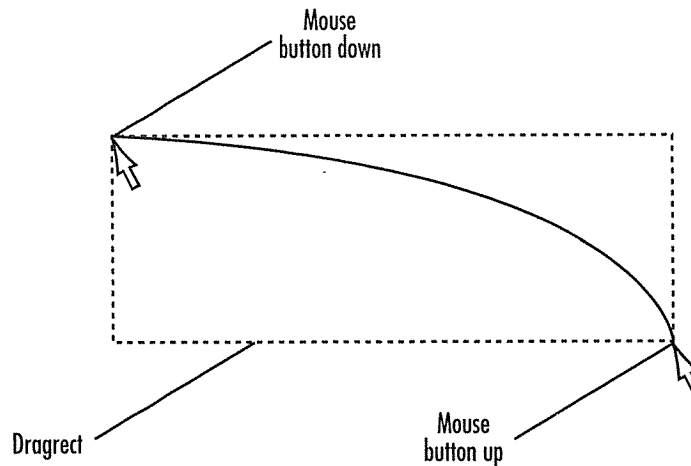
Mouse
button down

Dragrect

Mouse
button up

Figure 16-1

The simple click-and-drag operation, when the cursor is not on any particular object at mouse-down time, normally creates a dragrect that selects any object that is wholly enclosed in it when the mouse button is released. This is a familiar idiom to users of drawing programs and many word processors.

Visual indication of selection

It is critical that you visually indicate to the user when something is selected. The selected state must be easy to spot on a crowded screen, unambiguous, and must not obscure the object or what it is.

Design tip: Make selection visually bold and unambiguous.

The old Norton Utilities for DOS were infamous for putting up a dialog box with two push-button choices, the selected one in gray and the unselected one in blue—or was it the unselected one in gray and the selected one in blue? I couldn't tell, and neither could anybody else. Pressing the ENTER key was a gamble because the selection was ambiguous. Particularly if there are only two selectable objects on the screen, you must be careful about what you choose to indicate selection. You must assure that anyone can easily tell by visual inspection which one is selected and which isn't. It's not good enough just to be able to see that they are different. In Windows, it's harder to pull a stunt like that, but the lesson is still valid. Also, a significant portion of the population is color-blind, so color alone is insufficient to distinguish between selections.

The performance hack for indicating selection

Traditionally, selection is accomplished by inversion—by inverting the pixels of the selected object.

On a monochrome screen, this means turning all of the white pixels black and all of the black pixels white, but how many of you are still using black-and-white monitors? When the original Macintosh was released in 1984, it was a monochrome computer in spirit as well as in hardware. Because of this, Apple felt justified in using the inversion technique for indicating selections. Inversion was accomplished by the expedient of exclusive-ORing (or XORing) the pixels of the selected object with all 1 bits (or all 0 bits, depending on the processor). The XOR happens to be one of the fastest operations a CPU can execute, and with the limited computing power available in 1984, this was an easily justifiable choice. XORs are not only naturally fast but, by a curious quirk of digital circuitry, the action of an XOR can be undone merely by repeating the identical XOR. Fast! Microsoft continued the XOR technique in the first releases of Windows even though it was never a monochrome system in thought or in deed.

The hidden gotcha is that the result of the XOR operation is only defined when its operands are binary: on or off, one or zero, white pixels or black pixels. Color, however, is represented by more than a single bit. A 256-color screen uses eight bits. When the XOR is used on these more-complex numbers, the individual bits invert reliably, but a problem arises when the new value is sent to the physical video screen. Different video drivers interpret those bits in very different ways. The number may be split into smaller pieces to control individual red, green or blue bits, or they may result in a subscript for a color-table lookup. The result is that, although the XOR operation will be consistently represented on your computer, it may well be represented completely differently on another computer. XOR really is undefined for color video. Sure, it works, but the colors you get are defined only by accidents of hardware and not by any standard. What is the inverse of blue? In art class, it's yellow, but in Boolean algebra: who knows? In Windows, the bits are reasonably standard and the colors are generally predictable, but this technology is an accident waiting to happen.

Word processors and spreadsheets almost always show black text on a white background, so it is reasonable to use the XOR inversion shortcut to show selection. When colors are used, inversion still works, but the results may be

aesthetically lacking. For example, in Windows 1 and 2, if you used the Control Panel program to configure your screen colors, and you set your menus to yellow instead of gray, they inverted to blue. This was certainly noticeable but not necessarily desirable.

Microsoft acknowledged this problem in Windows 3.0 by defining two new system color settings: COLOR_HIGHLIGHT and COLOR_HIGHLIGHTTEXT. Of course, these manifest constants merely represent changeable colors rather than some fixed color. Each user can change these variable definitions, which then remain constant for all of their applications. Along with these new colors came a corresponding standard for use: When an object is selected, its color changes to whatever color is represented by COLOR_HIGHLIGHT. Any text or other contrasting pixels within the selected object change to whatever color is represented by COLOR_HIGHLIGHTTEXT. If the selection is concrete, as in a word processor, the background becomes COLOR_HIGHLIGHT and the foreground text becomes COLOR_HIGHLIGHTTEXT. This new standard normalizes the visual behavior of selection on a color platform. It is an excellent idea on Microsoft's part and should be followed widely.

> **Design tip:** Use COLOR_HIGHLIGHT and COLOR_HIGHLIGHTTEXT to show selection.

It is easy to see what colors these two constants represent simply by pulling down any menu in any program. The standard menu system in Windows uses COLOR_HIGHLIGHT and COLOR_HIGHLIGHTTEXT in the prescribed way. Of course, it is also interesting to look at a program like Excel or Word and notice what colors they use to indicate selection within cells or text. Yup, you guessed it: They invert instead of using the new standard selection colors.

Modern computing power makes the performance-hack of inversion moot. It just isn't that time-consuming anymore to use defined, consistent selection colors. I look forward to seeing the first word processor program to have a COLOR_HIGHLIGHT edit caret and, when dragging to select some text, have the selection background in COLOR_HIGHLIGHT and the text characters in COLOR_HIGHLIGHTTEXT. If you are responsible for a selection-intensive program like a word processor or spreadsheet, you might take the expedient route and use XOR inversion to indicate selection. If, however, you want to do it right and show the world your skill, use COLOR_HIGHLIGHT and watch the delighted faces.

Selecting multi-color objects

In drawing, painting, animation and presentation programs, where we deal with multi-color objects, the only decent solution I can see is to *add* selection indicators to the image, rather than by changing the selected image's color, whether by inversion or COLOR_HIGHLIGHT. Inversion can obscure details like accompanying text, while using the single system colors forces the program to reduce the selected image to two colors, foreground and background. This will likely obscure many details in multi-color objects.

Microsoft's PowerPoint is very color-intensive, and the slide view is rarely monochrome. I suspect that the authors tried to outwit the problems of inversion selection in multi-color objects by experimenting with the internal Boolean operations. When characters within text objects are selected for editing, the background turns black and the actual characters are inverted. The consistent black background can be reassuring, while the inverted pixels are otherwise fine. However, when the text is white (as it commonly is), its background turns black and the characters are inverted from white to, ta-da, black, and this makes for very difficult editing. The authors were caught by their own cleverness and created an idiom with significant areas of failure. I'm not impressed. I would rather have seen them force the background black and the foreground white. It would have been less clever but a lot better for the user.

Whatever color you choose, in a richly colored environment the selection can get visually lost. The solution is to instead highlight the selection with an additional graphic that shows its outline. This is often done with grapples (discussed in the next chapter): little boxes that surround the selected object. Grapples can still get lost in the clutter, particularly with modern, powerful image-manipulation programs. There is, however, one way to assure that the selection will always be visible regardless of the colors used: indicate the selection by movement.

One of the first programs on the Macintosh, MacPaint, had a wonderful idiom where a selected object was outlined with a simple dashed line, except that the dashes all moved in synchrony around the object. The dashes looked like ants in a column; thus, it earned the colorful sobriquet marching ants.

Unfortunately, this idiom has had little currency on the Windows platform. The animation is not hard to do, although it takes some care to get it right, and it works regardless of the color mix and intensity of the background. Adobe's PhotoShop uses this idiom to show selected regions of photographs, and it

works very well. I'd really like to see this idiom used more widely on Windows applications. I suspect the increase in multimedia applications will accelerate its arrival. Besides, subtle animation adds a very desirable sense of engagement and humanity to the interface.
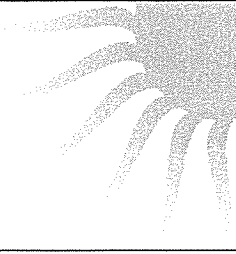
Direct Manipulation

17
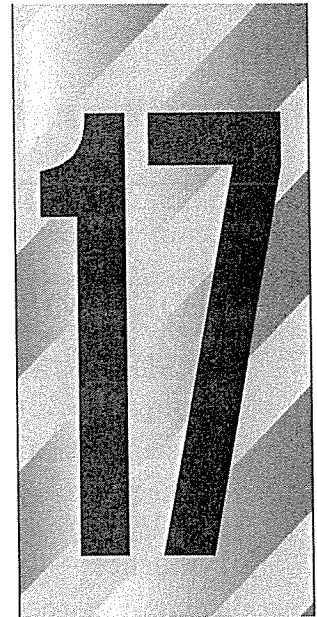
Ben Shneiderman coined the term direct manipulation in 1974. Here is my paraphrase of its three elements:

O   Visual representation of the manipulated objects

O   Physical actions instead of text entry

O   Immediately visible impact of the operation

A less-rigorous definition would say that direct manipulation is clicking-and-dragging things, and although this is true, it can easily miss the point that Shneiderman subtly makes. Notice that of his three points, two of them concern the visual feedback the program offers the user, and only the second point concerns the user's actions. It might be more accurate to call it "visual manipulation" because of the importance of what we see during the process. Unfortunately, I've seen many instances of direct-manipulation idioms implemented without adequate visual feedback, and these idioms fail to satisfy the definition of effective direct manipulation.

229

A rich visual interaction is the key to successful direct manipulation

Yet another observation about direct manipulation—one that is hidden by its obviousness—is that we can only directly manipulate information that is already displayed by the program; it must be visible for us to manipulate it, which again emphasizes the visual nature of direct manipulation. If you want to create effective direct-manipulation idioms in your software, you must take care to render data, objects, gizmos and cursors with good graphic detail and richness.

Direct manipulation is simple, straightforward, easy to use and easy to remember. Unfortunately, when users are first exposed to a given direct-manipulation idiom, they generally cannot intuit it or discover it independently. Direct manipulation should be taught, but the teaching of it is trivial—usually consisting of merely pointing it out—and, once taught, is never forgotten. It is a classic and archetypal example of idiomatic design. Adding metaphoric images may help, but you cannot depend on finding an appropriate one, and if you do, you cannot depend on it communicating clearly to all users. Resign yourself to the burden of teaching idioms. Console yourself with the ease of that teaching.

Apple's guide to human interaction says, with regard to direct manipulation, that "users want to feel that they are in charge of the computer's activities." Both these published guidelines and the Macintosh user interface make clear that Apple believes in direct manipulation as a fundamental tenet of good user interface design. However, cognitive psychology guru, Don Norman, says, "But direct manipulation, first-person systems have their drawbacks. Although they are often easy to use, fun, and entertaining, it is often difficult to do a really good job with them. They require the user to do the task directly, and the user may not be very good at it." Norman goes on to describe the inappropriateness of giving him a drawing program with great direct-manipulation idioms because he is such a poor artist. Which of these two contradictory statements should we believe?

The answer, of course, is both of them. As Apple says, direct manipulation is an extremely powerful tool, and as Norman says, the tool must be put into the hands of someone qualified to use it.

This contradiction should illustrate the differences between the various direct-manipulation types. Pushing a button is direct manipulation, and so is drawing with the pen tool in a paint program. Any normal user can push a button, but few are capable of drawing well with the pen tool. These examples illustrate the two variants of direct manipulation: management and content. Management includes gizmo-manipulation like button pushing and scrolling, and is generally accessible to all users. Content is drawing, and although it can be performed by anyone, its results will always be commensurate with the artistic talent of the manipulator.

All text and image manipulations such as those you find in programs like Corel Draw!, Adobe PhotoShop or Paint are drawing operations. Programs like ABC Flowcharter and Visio strain the definition, but even their more-structured interfaces are still content-centered and require some graphic talent from the user. Drawing will be discussed in detail in the next chapter.

In the management category, we find five varieties of direct manipulation:

O   Making selections

O   Dragging-and-dropping

O   Manipulating gizmos

O   Resizing, reshaping and repositioning

O   Arrowing
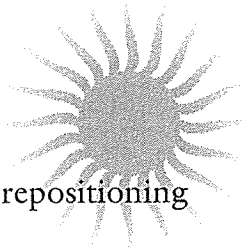
Selection was discussed in Chapter 16, and drag-and-drop will be discussed in Chapter 18, so I'd like to address the remaining three direct-manipulation idioms here, making some general observations along the way.

Manipulating gizmos

We can further divide up the types of direct manipulation by which mouse action they require: clicking or clicking-and-dragging.

Most gizmos—like buttcons, push-buttons, checkboxes and radio buttons—merely require the user to move the cursor over them and click the mouse button once. In terms of gizmo variants, these are a minority; but in terms of the number of actions a user will take in the average execution of a typical application, single clicking on buttcons and push-buttons is likely to be a majority.

Single-button click operations are the simplest of direct-manipulation idioms and the ones that work best with gizmos that specify operations immediately. Naturally, these functions are the ones that fall into the user's working set and will be invoked most frequently.

Beyond these simple gizmos, most direct-manipulation idioms demand a click-and-drag operation. This is a fundamental building block of visual interaction, and we will explore it in some detail.

Anatomy of a drag

A drag begins when the user presses the mouse button and then moves it without releasing the button. The set of cursor screen coordinates when the user first presses the mouse button is called the **mouse-down point** and that when the user releases the button is called the **mouse-up point**. The mouse-down point is a known quantity throughout any direct-manipulation operation. The mouse-up point only becomes known at the end of the process.

Once a drag begins, the entire interaction between the user and the computer enters a special state I call **capture**.

In programmer lingo, we say that all interaction between the system and the user is captured, meaning that no other program can interact with the user until the drag is completed. Any actions the user might take with the mouse or keyboard or any other input device go directly to the program—technically, the window—in which the mouse button first went down. I call this window that owns the mouse-down point the **master object**. If this master object is concrete data or a gizmo, the drag will likely indicate a selection extension or a gizmo state change. However, if the master object is a discrete object, it more likely indicates the beginning of a direct-manipulation operation like drag-and-drop, and capture will play an important part.

Technically, a state of capture exists the instant the user presses the mouse button, and it doesn't end until that mouse button is released, regardless of the distance the mouse moves between the two button actions. To the human, a simple click-and-release without motion seems instantaneous, but to the program, hundreds of thousands of instructions can be executed in the time it takes to press and release the button. If the user inadvertently moves the mouse before releasing the button, capture protects him from wildly triggering adjacent controls. The master object will simply reject such spurious commands.

Escaping from capture

One of the most important—yet most frequently ignored—parts of a drag is a mechanism for getting out of it. The user not only needs a way to abort the drag, if he does, he needs to have solid assurance that he did so successfully.

If the latter condition is met, the former idiom can be a lot more effective. That is, if the communication to the user that the drag action was canceled is clear, bold and unambiguous, he will be reassured and confident in using the cancel idiom, whatever it may be. Most applications, though, have no means of drag cancellation whatsoever. This is a grave lapse in user interface terms, as any good interface provides consistent and reliable ways out of a user's ill-starred action.

Provide an escape from dragging, and inform the user

At a minimum, the ESCAPE key on the keyboard should always be recognized as a general-purpose cancel mechanism for any mouse operation, either clicking or dragging. If the user presses the ESCAPE key while holding down the mouse button, the system should abandon the state of capture and return the system to the state it was in before the mouse button was pressed. When the user subsequently releases the mouse button, the program must remember to discard that mouse-up input before it has any side effect.

Because the meta-keys are often the only keys that have any meaning during drags, we could actually use any non-meta-keystroke to cancel a mouse stroke, rather than offering up only the ESCAPE. However, some programs allow the use of the arrow keys in conjunction with the mouse (we'll discuss this in the next chapter), so there are some exceptions to work around.

My personal favorite cancel idiom is the chord-click, where the user presses both mouse buttons simultaneously. Typically, the user likely begins a drag with the left mouse button, then discovers that he doesn't really want to finish what he has begun. He presses the right mouse button, then safely releases both. The idiom is insensitive to the timing or sequence of the release, and works equally well if the drag was begun with the right mouse button.

Design tip: Cancel drags on chord-click.

Microsoft used chord-clicking for drag cancel in their Word for DOS software, but unfortunately discarded the idiom when it went to Windows. Admittedly, the idiom is for minnies, but it is bad design to hobble an interface for minnies simply to pander to elephants. At least the current version of Word recognizes the ESCAPE key as a drag cancel.

Sad to say, the chord-click action is not defined in the Windows API. There is no system call to test for it, and no message is generated when the user chord-clicks. The messages are there for the asking, but it's hard for Visual Basic programmers to get them. However it is not difficult to code if you are writing in C or C++, and a DLL for VB would be easy to create.

Because Microsoft was so tentative in committing to the presence of a second mouse button, it is only fitting that they were reluctant to commit the chord-click to a cancel idiom. But now that Microsoft seems to have admitted, in Windows 95, that all of their users will have at least two mouse buttons, adopting the chord-click as a universal cancel idiom would only make good sense. Write your congressperson today.

Informing the user

If your program is well-designed and enables the user to cancel out of a drag operation with an ESCAPE key or a chord-click, the problem still remains of assuring the user that he is now safe. The cursor may have been changed to indicate that a drag was in progress, or an outline of the dragged object may have been moving with the cursor. The cancellation makes these visual hints go away, but the user may still wonder if he is truly safe. A user may have pressed the ESCAPE key, but is still holding the mouse button down, unsure whether it is entirely safe to let go of it. It is cruel and unusual punishment to leave him in this state. It is imperative that he be informed that the operation has been effectively canceled and that releasing the mouse button is OK. It can't hurt—and can only help—to make sure that he gets a reassuring message.

The message should clearly state that the drag is harmlessly over. I designed such an idiom for one of my clients that looked—and sounded—like a big, red, rubber stamp saying "Drag Canceled" had been thumped down in the middle of the screen. You can see this in Figure 17-1. At the instant the user cancels the drag, a bitmap about six by ten centimeters appears, centered on the screen,

and remains there for two seconds. The moment it appears, a prerecorded sound of a rubber stamp striking paper is played on the optional sound system. Users went crazy over this idiom, often starting drags just so they could joyously abort them.
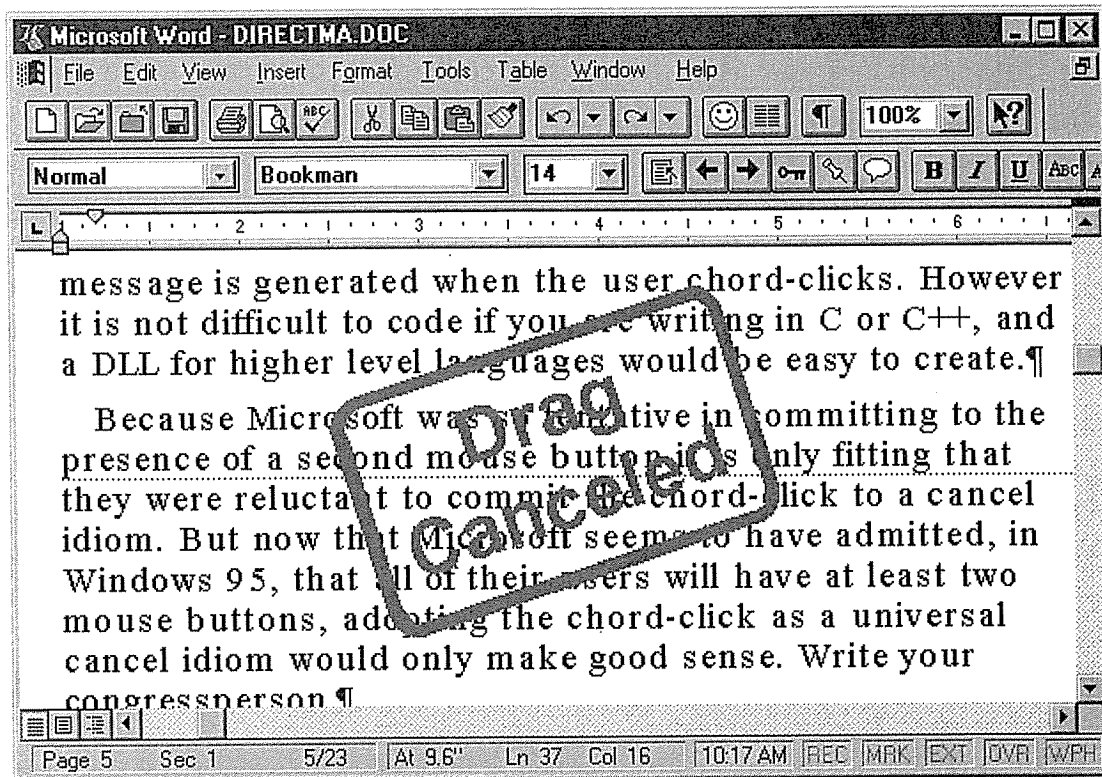

Figure 17-1

When the user cancels an unintended drag operation, they need to know positively and immediately that the operation has, indeed, been safely canceled. What could be better than the big, red, rubber stamp of implacable bureaucracy assuring them that the airplane has been grounded, the factory shut down, the train stopped, the groundwater cleaned, the nuke decontaminated, the forest saved, the criminal apprehended. Now, life can go on, happily ever after. And don't forget to add that satisfying "Thump" sound effect!

Let's go back to the drag itself: Once the drag begins, the meaning of the user's actions varies depending on the type of drag action. The drag action depends on the program, the context and the master object.

In the simplest case, concrete data, the drag means to extend the selection. The text or cells or whatever are selected contiguously from the mouse-down point to the mouse-up point.

If the mouse goes down inside a gizmo, the gizmo must visually show that it is poised to undergo a state change. This action is important and is often neglected by those who create their own gizmos. It is a form of active visual hinting that I call the **pliant response**.

A push-button needs to change from a visually outdented state to a visually indented state; a checkbox should highlight its box but not show a check just yet. The pliant response is an important feedback mechanism for any gizmo that either invokes an action or changes its state, letting the user know that some action is forthcoming if he releases the mouse button. The pliant response is also an important part of the cancel mechanism. When the user clicks down on a button, that button responds by indenting. If the user moves the mouse away from that button while still holding the button down, the button should return to its quiescent, outdented state. If the user then releases the mouse, the button will not be activated (as is consistent with the missing pliant response).

Dragging gizmos

Many gizmos, particularly menus, require the moderately difficult motion of a click-and-drag rather than a mere click. This direct-manipulation operation is more demanding of the user because of its juxtaposition of fine motions with gross motions to click, drag and then release the mouse button. Although menus are not used as frequently as toolbar gizmos, they are still used very often, particularly by new or infrequent users. Thus, we find one of the more intractable conundrums of GUI design: The menu is the primary gizmo for beginners, yet it is one of the more difficult gizmos to physically operate. I know of no solution to this problem other than to provide additional idioms to accomplish the same task. If a function is available from the menu and it is one that will be used more than just rarely, make sure to provide other idioms for invoking the function—idioms that don't require a click-and-drag operation.

One of the nice features of Windows 3.x is the ability to work its menus with a series of single clicks rather than clicking-and-dragging. You click on the menu and it drops down. You point to the desired item and click once to select it and close the menu. I find it remarkable that Apple hasn't included this idiom in their interface. In Windows 95, Microsoft has extended this idea even further by putting the program into a sort-of "menu mode" as soon as you click once on any menu. When in menu mode, all of the top-level menus in the program and all of the items on those menus are active, just as though you were

clicking-and-dragging. As you move the mouse around, each menu in turn drops down without having to use the mouse button at all. This can be disconcerting if you are unfamiliar with it, but after the initial shock has worn off, the action is generally more pleasant, mostly because it is easier on the wrist.

There are other types of click-and-drag gizmos. **Cascading menus** are another variant.

In a cascading menu, you pull down a menu in the normal way, then launch a secondary menu from an item on the first menu by dragging the mouse to the right. Cascading menus, like the one shown in Figure 17-2, can be stacked up so there are more than one. They form a hierarchy of menus.
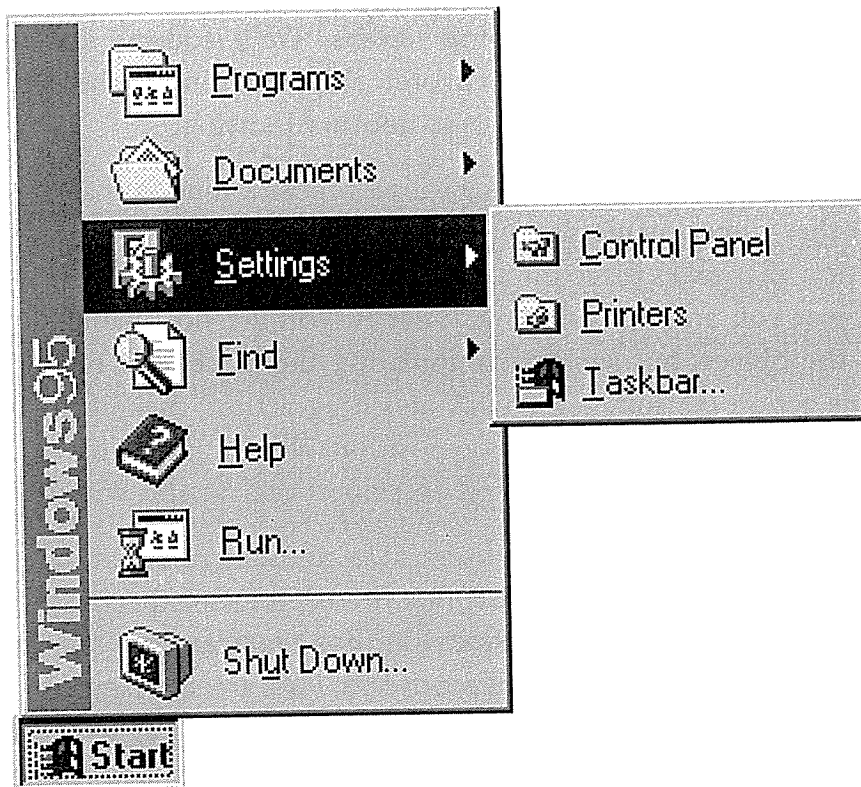

Figure 17-2

In Windows 95, Microsoft implemented the Startbar, with its rich array of cascading menus, for the avowed purpose of making life easier for neophytes and elephants. Physically navigating cascading menus is tough, and logically they are a hierarchy—one of the more difficult concepts for non-programmers to grasp. The Startbar is clearly a winner, and the new "menu-mode" certainly helps, but will cascades prove popular with elephants? Only time will tell.

Cascading menus demand a fair amount of skill by the mouse user, because any false move that causes the cursor to detour outside of the enclosing menu rectangle will cause one or another of the menus to disappear. Cascades can be a frustrating gizmo to manipulate, and although they have their place in interface design, I recommend against using them for frequently used functions. Of course, Microsoft and I are clearly not in agreement on that point, as Windows 95 makes extensive use of cascading menus throughout its interface. I suspect that the new "menu-mode" has convinced the designers in Redmond that the problems with cascades are eliminated. I doubt they are right, and a straw poll of my colleagues indicates agreement with my conclusion.

Repositioning

Gizmos that depend on click-and-drag motions include icons and the various repositioning, resizing and reshaping idioms. We'll address icons when we discuss drag-and-drop in the next chapter. Repositioning is the simple act of clicking on an object and dragging it to another location.

The most significant design issue regarding repositioning is that it usurps the place of other direct-manipulation idioms. Repositioning is a form of direct manipulation that takes place on a higher conceptual level than that occupied by the object you are repositioning. That is, you are not manipulating some aspect of the object, but simply manipulating the placement of the object in space. This action consumes the click-and-drag action, making it unavailable for other purposes. If the object is repositionable, the meaning of click-and-drag is taken and cannot be devoted to some other action within the object itself, like a button press.

The most general solution to this conflict is to dedicate a specific physical area of the object to the repositioning function. For example, you can reposition a window in Windows or on the Macintosh by clicking-and-dragging its caption bar. The rest of the window is not pliant for repositioning, so the click-and-drag idiom is available for more application-specific functions, as you would expect. The only hint of the window's draggability is the color of the caption bar, a subtle visual hint that is purely idiomatic: there is no way to intuit the presence of the idiom. But the idiom is very effective, and it merely proves the efficacy of idiomatic interface design. Generally, though, you need to provide some more-explicit visual hinting of an area's pliancy. The cost of this solution is the number of pixels devoted to the caption bar. Mitigating this is the fact that the caption bar does double-duty as a program identifier, active status indicator and repository for certain other system-standard controls such as the minimize, maximize and close functions, as well as the system menu.

To move an object, it must first be selected. This is why selection must take place on the mouse-down transition: the user can drag without having to first click-and-release on an object to select it, then click-and-drag on it to reposition it. It feels so much more natural to simply click it and then drag it to where you want it in one easy motion. When you pick up a book or a pencil, you select and move it in one combined action, rather than having to pick it up to select it, put it back, then pick it up again to move it. And yet, in Word, Microsoft has given us this clumsy click-wait-click operation to drag chunks of text. You must click-and-drag to select a section of text, then wait a second or so and click-and-drag again to move it. This idiom is very clumsy, but there is really no way around it in concrete selection. If Microsoft were willing to dispense with their meta-key idioms for extending the selection, those same meta-keys could be used to, say, select a sentence and drag it in a single movement. But this still wouldn't solve the problem of selecting and moving some arbitrary hunk of text.

Resizing and reshaping

When referring to the "desktop" of Windows and other similar GUIs, there isn't really any functional difference between resizing and reshaping. The user adjusts a rectangular window's size and aspect ratio at the same time and with the same control by clicking-and-dragging on a dedicated gizmo. On the Macintosh, there is a special resizing control on each window in the lower right corner, frequently nestled into the space between the application's vertical and horizontal scrollbars. Dragging this control allows the user to change both the height and width of the rectangle. Windows 3.x eschewed this idiom in favor of the thickframe surrounding each window. The thickframe is an excellent solution. It offers both generous visual hinting and cursor hinting, so it is easily discovered. Its shortcoming is the amount of real estate it consumes. It may only be four or five pixels wide (you can adjust it down to two pixels), but multiply that by the sum of the lengths of the four sides of the window, and you'll see that thickframes are expensive.

Windows 95 institutes a new reshaping-resizing gizmo that is remarkably like the Macintosh's lower-right-corner reshaper/resizer. The gizmo is a little triangle with 45°, 3D ribbing, which you can see in Figure 17-3. I've christened this new gizmo with a contraction of the words shaper and triangle: Shangle. The shangle still occupies a square of space on the window, but most Windows 95 programs have a status bar of some sort across their bottoms, and the reshaper-resizer borrows space from it rather than from the client area of the window.

Windows 95 still retains the thickframe and its cursor hinting, but it has dramatically changed so that virtually no visual hinting of the frame remains, although the cursor hinting remains. The user interface gurus in Redmond are clearly Mac-influenced, and the new shangle gizmo and the visual attenuation of the thickframe are prime evidence of this swing. I suspect that the thickframe will now begin to lose currency in favor of the shangle.
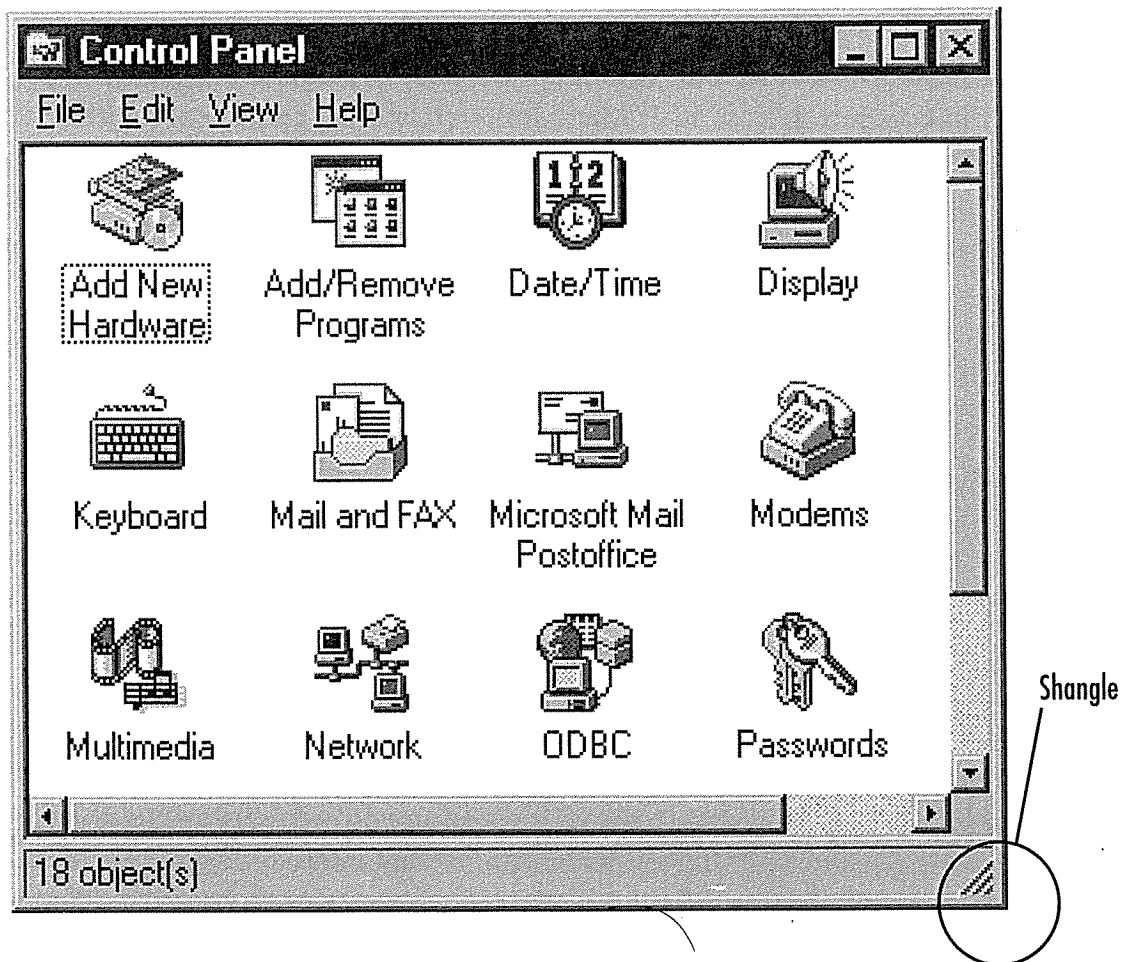

Shangle

Figure 17-3

I call the new-for-Windows-95 reshaper/resizer gizmo a shangle (a contraction of shaper and triangle). You can see an example of it in the lower right corner of the window. Notice that the shangle resides on the bottom status bar and not in the wasted square between the two scrollbars, as is normal on the Macintosh. The pliant area is actually a square, because rectangles are much more programmer-friendly in Windows than triangles are, but the effect is negligible. Actually, in this program, the thickframe is still active but has become invisible.

Thickframes and shangles are fine for resizing windows, but when the object to be resized is a graphical element in a painting or drawing program, it is not acceptable to permanently superimpose controls onto it. A resizing idiom for graphical objects must be visually bold to differentiate itself from parts of the drawing, especially the object it controls, and it must be respectful of the user's view of the object and the space it swims in. The resizer must not obscure the resizing action. There is a popular idiom that accomplishes these goals. It consists of eight little black squares positioned one at each corner of a rectangular object and one centered on each side. These little black squares, shown in Figure 17-4, are often called "handles," but that word is so overbooked in the programming world that I prefer to call them grapples to avoid confusion.

Grapples are a boon to designers because they can also indicate selection. This is a naturally symbiotic relationship, as an object must usually be selected to be resizable.
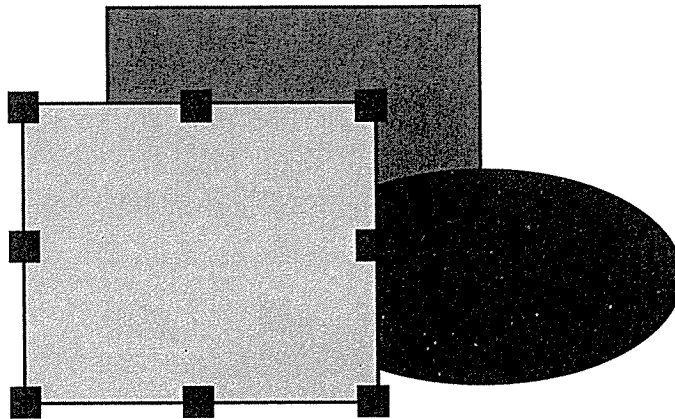

Figure 17-4

The selected object has eight grapples, one at each corner and one centered on each side. The grapples indicate selection, and are a convenient idiom for resizing and reshaping the object. Grapples are sometimes implemented with pixel inversion, but in a multi-color universe they can get lost in the clutter.

The grapple centered on each side moves only that side, while the other sides remain motionless. The grapples on the corners simultaneously move both of the sides they touch. Wow, is that logical!

Grapples tend to obscure the object they represent, so they don't make very good permanent controls. This is why we don't see them on top-level resizable windows. For that situation, the thickframe or shangle is a better idiom. If the selected object is larger than the screen, the grapples may not be visible. If they are hidden off screen, not only are they unavailable for direct manipulation, but they are useless as indicators of selection.

Notice that the assumption in this entire discussion of grapples is that the object under scrutiny is rectangular or can be easily bounded by a rectangle. Certainly in the Windows world, things that are rectangular are easy for programs to handle and non-rectangular things are best handled by enclosing them in a bounding rectangle. If the user is creating an organization chart, this may be fine, but what about reshaping more complex objects? There is a very powerful and useful variant of the grapple, which I call a **vertex grapple**.

Many programs draw objects on the screen with polylines. A **polyline** is a graphic programmer's term for a multi-segment line defined by an array of vertices. If the last vertex is identical to the first vertex, it is a closed form and the polyline is a polygon. When the object is selected, the program, rather than placing eight grapples as it does on a rectangle, places one grapple on top of every vertex of the polyline. The user can then drag any vertex of the polyline independently and actually change one small aspect of the object's internal shape rather than affecting it as a whole. This is shown in Figure 17-5.
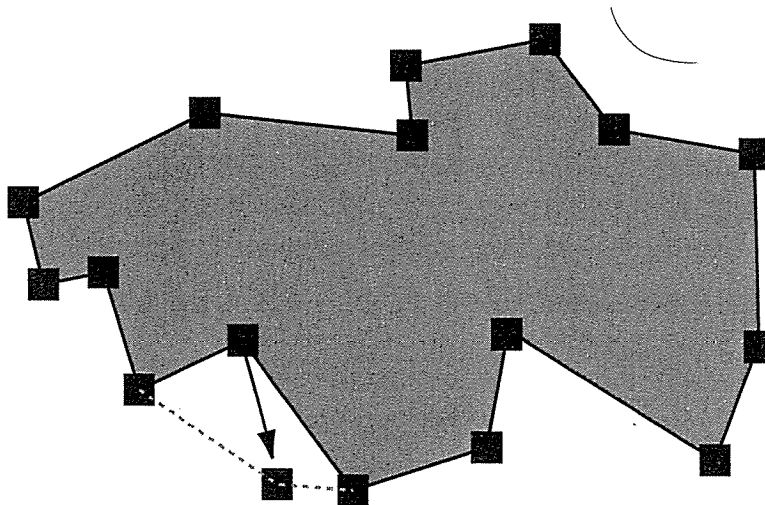

Figure 17-5

These are vertex grapples, so named because there is one grapple for each vertex of the polygon. The user can click and drag any grapple to reshape the polygon one segment at a time. This idiom is useful for drawing programs, but it may have application in desktop productivity programs, too.

Many objects in PowerPoint, including polygons, are rendered with polylines. If you click on a polygon, it is given a bounding rectangle with the standard eight grapples. If you double-click on the polygon, the bounding rectangle disappears and vertex grapples appear instead. It is important that both of these idioms are available, as the former is necessary to scale the image in proportion, while the latter is necessary to fine-tune the shape.

Resizing and reshaping meta-key variants

In the context of dragging, a meta-key is often used to constrain the drag to an orthogonal direction. This type of drag is called a **constrained drag**, and is shown in Figure 17-6.

A constrained drag is one that stays on a 90° or 45° axis regardless of how the user might veer off a straight line with the mouse. Usually, the SHIFT meta-key is used, but this convention varies from program to program. Constrained drags are extremely helpful in drawing programs, particularly when drawing business graphics, which are generally neat diagrams. The angle of the drag is determined by the predominant motion of the first few millimeters of the drag. If the user begins dragging on a predominantly horizontal axis, for example, the drag will henceforth be constrained to the horizontal axis. Some programs interpret constraints differently, letting the user shift axes in mid-drag by dragging the mouse across a threshold. Either way is fine.
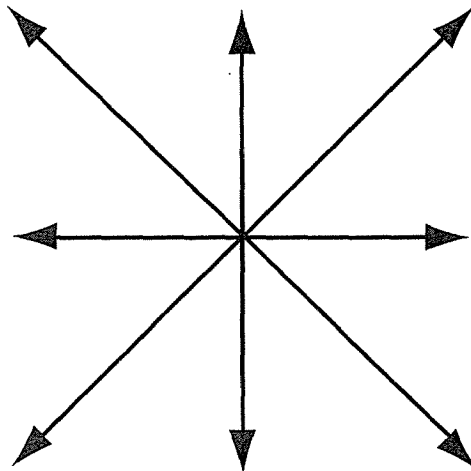
Figure 17-6

When a drag is constrained, usually by holding down the SHIFT key, the object is only dragged along one of the four axes shown here. The program selects which one by the direction of the initial movement of the mouse, an implementation of the drag threshold discussed later in the chapter.

The Paint program that comes with Windows 95 doesn't constrain drags when moving an object around, but it does constrain the drawing of a few shapes, like lines and circles. Most drawing programs (like PowerPoint) that treat their graphics as objects instead of as bits (like Paint) allow constrained drags.

The use of meta-keys gives rise to a curious question: where in the drag does the meta-key become meaningful? In other words, must the meta-key be held down when the drag begins—when the mouse button descends—or is it merely necessary for the meta-key to be pressed at some point during the drag? Or should the meta-key be pressed at the time the user releases the mouse button? In general, the answer is the latter case: If the computer detects that the meta-key is held down at the instant when the mouse button is released, the effect is considered valid. This is true in PowerPoint and Paint, for example.

Arrowing

A direct-manipulation idiom that can be very powerful in some applications is what I call **arrowing**, in which the user clicks-and-drags from one object to another, but instead of dragging the first object onto the second, an arrow is drawn from the first object to the second one.

If you use project management or organization chart programs, you are undoubtedly familiar with this idiom. For example, to connect one task box in a project manager's network diagram (often called a PERT chart) with another, you click-and-drag an arrow between them. The direction of the arrowing is significant: the task where the mouse button went down is the "from" task, and where the mouse button is released is the "to" task.

The visual arrows generally behave in a manner best described as **rubber-banding**.

Rubber-banding is where the arrow forms a line that extends from the exact mouse-down point to the current cursor position. The line is animated, so as the user moves the cursor, the position of the cursor-end of the line is constantly pivoting on the anchored end of the line (from the mouse-down point). Once the user releases the mouse button, the mouse-up point is known, and the program can decide whether it was within a valid target location. If so, the program draws a more permanent visual arrow between the two objects. Generally, it also links them logically.

As the user drags the end of the arrow around the screen, input is captured, and the rules of dragging in discrete data apply.

The arrowing function can't normally be triggered by the left button because it would collide with selection and repositioning. In some programs, it is triggered by the right button, but Windows 95 makes that problematic with its usurpation of the right click for the context menu. Hey! Is that ALT meta-key still unused?

Arrowing doesn't require cursor hinting as much as other idioms because the rubber-banding effect is so clearly visible. However, it would be a big help, in programs where objects are connected logically, to show which objects currently pointed-to are valid targets for the arrow. In other words, if the user drags an arrow until it points to some icon or widget on the screen, how can he tell if that icon or widget can legally be arrowed to? The answer, of course, is to have the potential target object engage in some active visual hinting.

What is indisputably vital, however, is a convenient means of canceling the action. Chord-clicking still works for this one.

Direct-manipulation visual feedback

As I said at the beginning of this chapter, the key to successful direct manipulation is rich visual feedback. Let's take a more detailed look at some visual feedback methods.

First off, we can divide the direct-manipulation process into three distinct phases:

1. *Free Phase*: Before the user takes any action

2. *Captive Phase*: Once the user has begun the drag

3. *Termination Phase*: After the user releases the mouse button

In the free phase, our job is to indicate direct-manipulation pliancy.

In the captive phase, we have two tasks. We must positively indicate that the direct-manipulation process has begun, and we must visually identify the potential participants in the action.

In the termination phase, we must plainly indicate to the user that the action has terminated and show exactly what the result is. We'll talk more about the captive and termination phases in the next chapter, "Drag-and-Drop."

Depending on which direct manipulation phase we are in, there are two variants of cursor hinting. During the free phase, I call any visual change the

cursor makes as it merely passes over something on the screen free cursor hinting. Once the captive phase has begun, I call changes to the cursor captive cursor hinting.
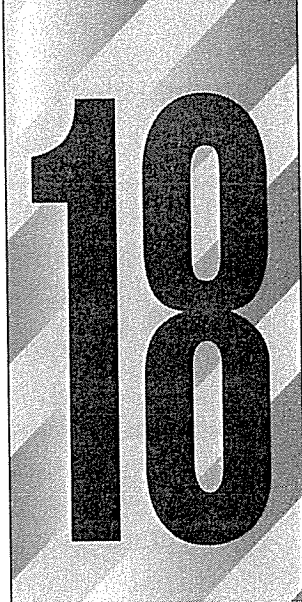
Microsoft Word uses the clever free cursor hint of reversing the angle of the arrow when the cursor is to the left of text to indicate that selection will be line-by-line or paragraph-by-paragraph instead of character-by-character as it normally is within the text itself. Many other programs use a hand-shaped cursor to indicate that the document itself, rather than the information in it, is draggable.

Microsoft is using captive cursor hinting more and more as they discover its usefulness. Dragging-and-dropping text in Word or cells in Excel are accompanied by cursor changes indicating precisely what the action is and whether the objects are being moved or copied. In Windows 95, when you drag a file in the Explorer, you actually drag the text of the name of the file from one place to another.

When something is dragged, the cursor must drag either the thing or some simulacrum of that thing. In a drawing program, for example, when you drag a complex visual element from one position to another, it may be too difficult for the program to actually drag the image (due to the computer's performance limitations), so it often just drags an outline of the object. If you are holding down the CTRL key during the drag to drag away a copy of the object instead of the object itself, the cursor may change from an arrow to an arrow with a little plus sign over it to indicate that the operation is a copy rather than a move. This is a clear example of captive cursor hinting.

Drag-and-Drop

O f all the direct-manipulation idioms characteristic of the GUI, nothing defines it more than the drag-and-drop operation, clicking and holding the button while moving some object across the screen. Surprisingly, drag-and-drop isn't used as widely as we imagine, and it certainly hasn't lived up to its full potential.

Whither drag-and-drop?

Any mouse action is very efficient because it combines two command components in a single user action: a geographical location and a specific function. Drag-and-drop is doubly efficient because, in a single, smooth action, it adds a second geographical location. Although drag-and-drop was accepted immediately as a cornerstone of the modern GUI, it is remarkable that drag-and-drop is found so rarely outside of programs that specialize in drawing and painting. Thankfully, this seems to be changing, as more programs add this idiom.

247

There are several variations of drag-and-drop, and they are only a subset of the many forms of direct manipulation. The characteristics of drag-and-drop are fuzzy and difficult to define exactly. We might define it as "clicking on some object and moving it elsewhere," although that is a pretty good description of repositioning, too. A more accurate description of drag-and-drop is "clicking on some object and moving it to imply a transformation."

The Macintosh was the first successful system to offer drag-and-drop. A lot of expectations were raised with the Mac's drag-and-drop that were never truly realized for two simple reasons:

1. Drag-and-drop wasn't a system-wide facility, but rather an artifact of the Finder, a single program.

2. As a single-tasking computer, the concept of drag-and-drop between applications didn't surface as an issue for many years.

To Apple's credit, they described drag-and-drop in their first user interface standards guide. On the other side of the fence, Microsoft not only didn't put drag-and-drop aids in their system, but it wasn't described in their programmer documentation. Nor was it implemented in their Finder equivalent, the notoriously brain-dead MSDOS.EXE, the first Windows shell. The only drag-and-drop anywhere in Windows was in the simple paint utility distributed with the system. Yet again, Microsoft shipped an operating system—a standard-defining tool—but abdicated their responsibility for adequately defining collateral standards. I'm not ungrateful, as Windows was still by far the best thing around on the PC platform. Still, had Microsoft defined even some rudimentary standards, the drag-and-drop world would have evolved stronger and more rapidly.

It wasn't until Windows 3.0 that any drag-and-drop outside of MSPAINT.EXE appeared. The new File Manager and Program Manager programs supported a rudimentary form of drag-and-drop. You could drag icons around in the Program Manager and files and directories around in the File Manager. Wonder of wonders, you could also drag an EXE file* from the File Manager into the Program Manager and create an icon, although few users knew this. This disappointing lack of design leadership has resulted in an industry-wide sluggishness to embrace drag-and-drop, much to our software's detriment.

After ten years, though, Windows is finally getting a drag-and-drop standard. It is not strictly a part of Windows, but rather a part of the OLE 2.0

---

*Or any other file, for that matter.

specification. To get a community of third-party developers to adopt a technology, there is something even better than having a defined standard: having a solid set of library routines that enable them to implement features in their applications without having to invent the technology themselves. No such libraries have ever been made available in the Windows environment. OLE 2.0 is so large and frustratingly complex that there is some peril that the drag-and-drop standard will become either lost or bastardized in various proprietary implementations. This unfortunate bind will only be resolved when some clever vendor encapsulates the functionality of drag-and-drop in a powerful, flexible and easy-to-program package, then makes it widely available to applications developers.

I find it amusing that the Microsoft style guide treats drag-and-drop so lightly. It makes it sound like a simple and commonly known process, as though it was describing how to put on your shoes in the morning. Sorry, it's just not that easy.

Dragging where?

Fundamentally, you can drag-and-drop something from one place to another inside your program, or you can drag-and-drop something from inside your program into some other program. I call these variants **interior drag-and-drop** and **exterior drag-and-drop**, respectively.

Interior drag-and-drop can be made pretty simple, both from a conceptual and from a coding point of view. Exterior drag-and-drop demands significantly more sophisticated support because both programs must subscribe to the same concepts, and they must be implemented in compatible ways. We'll talk more about the exterior variant after we get a look at the basics of drag-and-drop.

I classified repositioning as a direct-manipulation idiom and discussed it in the last chapter. Now we will discuss the remaining drag-and-drop variants. Primarily, there are two: master-and-target and tool manipulation.

Master-and-target

When the user clicks on a discrete object and drags it to another discrete object in order to perform a function, I call it **master-and-target**.

The object within which the dragging originates controls the entire process, so it is the master object, which will be a window. If you are dragging an icon, that icon is a window. If you are dragging a paragraph of text, the enclosing editor

is the window. When the user ultimately releases the mouse button, whatever was dragged is dropped on some **target object**.

The main purpose of the term "master-and-target" is to differentiate this operation from the kind of drag-and-drop operations we find in drawing and painting programs, where tools and graphical objects are dragged around on an open canvas. Master-and-target is a more function-oriented idiom, where manipulating logical objects represents some behind-the-scenes processes. The most familiar form of master-and-target drag-and-drop is rearranging icons in the Program Manager or in the Macintosh Finder.

Dragging data to functions

Instead of dragging a file or folder to another folder, you can drag it to a gizmo that represents a function. This idiom is arguably the most famous expression of direct manipulation because of the Macintosh's familiar trashcan. Windows 95 copies this familiar idiom with its "recycle bin." Someday, as we build software with better object-orientation, we'll be able to drag-and-drop objects onto gizmos representing functions other than just delete. Imagine targets representing a cloner, an archiver, a file compressor, a faxer or a contents-indexer.

Notice that all of the idioms in the above paragraph involve exterior drag-and-drop, because the target objects are separate programs. Within a single program, the code knows what objects are draggable—usually one type—and any function gizmo that it gets dropped on will easily handle it. In an exterior drop, the master object can come from any program, and the target gizmo may well not have any direct knowledge of the originating program or the dropped object. The target must be able to handle the unknown object in some reasonable way without necessarily understanding what it is or what is in it. The Program Manager, for example, can do this because it knows that it will only be handed files. What would it do if it were handed a paragraph of dragged text from a word processor, for example? If it can't handle the text, it isn't truly exterior capable. To Microsoft's credit, the Recycle Bin in Windows 95 can actually accept paragraphs of text dragged from Word or cells dragged from Excel. I have not yet been able to determine whether these are generic operations or just code specific to Microsoft applications.

To be truly exterior capable, an object must be able to accept a drop of anything from any other object, regardless of the originating program. At first, this sounds like a dauntingly complex implementation problem, but it doesn't have

to be. Mostly, it's a matter of defining interface standards. When data is dragged to an object, all the target object has to say is "yes, I can accept the drop" to the master object. The two objects then must negotiate over formats, because it is unreasonable to expect every object to accept data in every other program's proprietary formats. If the master object is Excel, say, it may initially offer the data in its internal format. Another Microsoft program may know how to decipher this format, but a Brand X product might not. So the Brand X target object politely demurs—not to the drop, but to the *format* of the drop's contents. Excel, the master, must then re-offer the data in successively more generic formats: SYLK, CSV, ASCII. The target object can turn up its nose at SYLK or CSV, but by convention, it must accept ASCII; it is the lowest common denominator format on all platforms. Every exterior capable object must minimally accept ASCII, simple bitmaps, pointers to files and, as we'll see, functions. Objects that hope to become successful in the open market will accept many more formats than that, but these four guarantee compatibility with everything. Even an audio file, for example, can ultimately be passed as a simple pointer to a disk file. I call exterior drag-and-drop protocols that support this type of haggling over formats **negotiated drag-and-drop**.

I call protocols like those in the Windows 3.x File Manager and Program Manager, which don't negotiate formats, **known-format drag-and-drop**.

Dragging functions to data

Proper, negotiated, exterior drag-and-drop capability includes dragging-and-dropping functions onto data as well as dragging-and-dropping data onto functions. Defining the scope of such actions can be problematic when working in concrete data, but it can still be generally quite useful. For example, a user could click on the italic buttcon on the toolbar and drag it down onto a cell in a spreadsheet. Clearly, the user's intent in this action is to turn the content in that cell to italic. Part of the format negotiation includes being able to recognize a function as a valid drop value. Conceptually, there is little difference between the function "delete" and the function "italic." In one, the target program deletes its internal copy of the data and hands it to the master. In the other, the target program hands a copy of the data to the italic function, which converts the text to italic and hands it back. This way the italic buttcon in Excel's window can be dragged onto text in Word's window, and Word will know what to do with it. Or, more meaningfully, the Brand X buttcon can be dragged onto the text in Microsoft Word. Once this interface is in place, little companies can begin to chip away at the big, monopolistic, mega-applications.

For now, there is no standard exterior drag-and-drop protocol, although OLE purports to offer one. Certainly, there is no negotiated drag-and-drop protocol.* A given protocol may allow format negotiation or function dragging. Your mileage may vary.

How master-and-target works

A well-designed master object will visually hint at its pliancy, either statically in the way it is drawn, or actively, by animating as the cursor passes over it.

The idea that an object is dragable is easily learned idiomatically. It is difficult to forget that an icon, selected text or other distinct object is directly manipulable, once the user has been shown this. He may forget the details of the action, so other feedback forms are very important *after* the user clicks on the object, but the fact of direct-manipulation pliancy itself is easy to remember. The first-timer or very infrequent user will probably require some additional help. This help will come either through additional training programs or by advice built right into the interface. In general, a program with a forgiving interaction encourages users to try direct manipulation on various objects in the program.

As soon as the user presses the mouse button over an object, that object becomes the master object for the duration of the drag-and-drop. On the other hand, there is no corresponding target object because the mouse-up point hasn't yet been determined: it could be on another object or in the open space between objects. However, as the user moves the mouse around with the button held down—remember, this is called the captive phase—the cursor may pass over a variety of objects inside or outside the master object's program. If these objects are drag-and-drop compliant, they are possible targets, and I call them drop candidates.

There can only be one master and one target in a drag, but there may be many drop candidates. Depending on the drag-and-drop protocol, the drop candidate may not know how to accept the particular dropped value, it just has to know how to accept the offered drop protocol. Other protocols may require that the drop candidate recognize immediately whether it can do anything useful with the offered master object. The latter method is slower but offers much better feedback to the user. Remember, this operation is under direct human control, and the master object may pass quickly over dozens of drop candidates before the user positions it over the desired one. If the protocol requires

---

*I offered the first (and possibly only) one several years ago called SPIDR (Standard PIck up and DRop). It was adopted by only a few major companies.

extensive conversing between the master object and each drop candidate, the interaction can be sluggish, at which point it isn't worth the game.

Visual indications

The only task of each drop candidate is to visually indicate that the hotspot of the captive cursor is over it, meaning that it will accept the drop—or at least comprehend it—if the user releases the mouse button. Such an indication is, by its nature, active visual hinting.

> ***Design tip:*** The drop candidate must visually indicate its dropability.

The weakest way to offer the visual indication of dropability is by changing the cursor. It is the job of the cursor to represent what is being dragged and leave all indications of drop candidacy to the drop candidate itself.

> ***Design tip:*** The drag cursor must visually indicate the master object.

It is important that these two visual functions not be confused. Unfortunately, Microsoft seems to have done so in both Windows 3.x and Windows 95. I suspect this decision was made more for the ease of coding than for any design considerations. It is much easier to change the cursor than it is to have drop candidates highlight to show their dropability. The role of the cursor is to represent the master, the dragged object. It should not be used to represent the drop candidate.

As if that weren't bad enough, Microsoft performs cursor hinting with the detestable circle with bend sinister, which I call a sinister-circle.

The sinister-circle is not a pleasant idiom because it tells users what they can't do. It is negative feedback. The sinister-circle is an idiom for "don't do it," and a user can easily construe its meaning to be "don't let go of the mouse now or you'll do some irreversible damage" instead of "go ahead and let go now and nothing will happen." Adding the sinister-circle to cursor hinting is a sad combination of two weak idioms and should be avoided, regardless of what the Microsoft style guide says.

Once the user finally releases the mouse button, the current drop candidate becomes the *target*. If the user releases the mouse button in the interstice between valid drop candidates, or over an invalid drop candidate, there is no target and the drag-and-drop operation ends with no action. Silence, or visual inactivity, is a good way to indicate this termination. It isn't a cancellation, exactly, so there is no need to show a cancel stamp.

Indicating drag pliancy

Active cursor hinting to indicate drag pliancy is a problematic solution. In an increasingly object-oriented world, more things can be dragged than not. A cursor flicking and changing rapidly can be more of a visual distraction than a help. One solution is to just assume that things can be dragged and let the user experiment. This method is reasonably successful in the Program Manager, the File Manager and the Explorer. Without cursor hinting, drag pliancy can be a hard-to-discover idiom, so you might consider building some other indication into the interface, maybe a textual hint or a ToolTip-style popup.

Once the master object is picked up and the drag operation begins, there must be some visual indication of this. The most visually rich method is to fully animate the drag operation, showing the entire master object moving in real-time. This method is hard to implement, can be annoyingly slow and very probably isn't the proper solution. The problem is that a master-and-target operation requires a pretty precise pointer. For example, the master object may be 6 centimeters square, but it must be dropped on a target that is 1 centimeter square. The master object must not obscure the target, and, because the master object is big enough to span multiple drop candidates, we need to use a cursor hotspot to precisely indicate which candidate it will be dropped on. What this means is that, in master-and-target, dragging a transparent outline of the object may be much better than actually dragging a fully animated, exact image of the master object. It also means that the dragged object can't obscure the normal arrow cursor either. The tip of the arrow is needed to indicate the exact hotspot.

Dragging an outline also is appropriate for most repositioning, as the outline can be moved relative to the master object, which is still visible in its original position.

Indicating drop candidacy

As the cursor traverses the screen, carrying with it an outline of the master object, it passes over one drop candidate after another. These drop candidates must visually indicate that they are aware of being considered as potential drop

targets. By visually changing, the drop candidate alerts the user that it can do something constructive with the dropped object.

A point so obvious as to be difficult to see is that the only objects that can be drop candidates are those that are currently visible. A running application doesn't have to worry about visually indicating its readiness to be a target if it isn't visible. Usually, the number of objects occupying screen real estate is very small—a couple of dozen at most. This means that the implementation burden should not be overwhelming.

Internally, the master object should be communicating with each drop candidate as it passes over it. A brief conversation should occur, where the master asks the target whether it can accept a drop. If it can, the target indicates it with visual hinting.

Microsoft not only doesn't insist on drop candidate visual hinting, it suggests that changing the cursor is sufficient. I believe that they do the industry a major disservice by taking this route. Certainly, it is easier to program this way, but in every user-centered way it is worse. It is difficult to understand what is being dragged, what the target is and whether the target can make sense of the drop. In Windows 95, at least on the desktop, icons now correctly indicate their drop candidacy by visually inverting. But I worry that this is just a shallow imitation of the Macintosh Finder specific only to the Window's desktop and not a new, system-wide standard for how master-and-target drag-and-drop should work.

Completing the drag-and-drop operation

When the master object is finally dropped on a drop candidate, the candidate becomes a bona fide target. At this point, the master and target must engage in a more detailed conversation than the brief one that occurred between the master and all of the other drop candidates. After all, the user has committed, and we now know the target. The target may know how to accept the drop, but that does not necessarily mean that it can swallow the particular master object dropped in this specific operation. This distinction is generally not important in interior drag-and-drop, but in exterior drag-and-drop, it is doubtful that there is enough time to resolve this issue during the captive phase. Of course, this is still a performance hack, and faster computers will someday allow sufficient communications detail during real-time drags.

The implication of this more-detailed conversation is that the transfer may fail. That is okay. It is better to show dropability and choke on the actual drop than it is to not indicate dropability. (If minimum common format standards are adhered to, after all, there should never be a physical failure.) If the drag-and-drop is negotiated, the format of the transfer remains to be resolved. If information is transferred, the master-and-target may wish to negotiate whether the transfer will be in some proprietary format known to both, or whether the data will have to be reduced in resolution to some weaker but more common format, like ASCII text.

Visual indication of completion

If the target and the master can agree, the appropriate operation then takes place. A vital step at this point is the visual indication that the operation has occurred. If the operation is a transfer, the master object must disappear from its source and reappear in the target. If the target represents a function rather than a container (such as a print icon), the icon must visually hint that it received the drop and is now printing. It can do this with an animation, or by changing its visual state.

A richly visual master-and-target drag-and-drop operation is one of the most powerful operations in the GUI designer's bag of tricks. I know that if this idiom is better supported by tool vendors, it will grow in popularity with application developers. Users will be the beneficiaries.

Tool-manipulation drag-and-drop

In drawing and painting programs, the user manipulates tools with drag-and-drop, where a tool or shape is dragged onto a canvas and used as a drawing tool. There are two basic variants of this that I call modal tool and charged cursor.

Modal tool

In **modal tool**, the user selects a tool from a list, usually called a toolbox or palette. The program is now completely in the mode of that tool: it will only do that one tool's job. The cursor usually changes to indicate the active tool.

When the user clicks-and-drags with the tool on the drawing area, the tool does its thing. If the active tool is a spray can, for example, the program enters "spray can mode" and it can only spray. The tool can be used over and over, spraying as much ink as desired until the user clicks on a different tool. If the user wants

to use some other tool on the graphic, like an eraser, he must return to the tool-box and select the eraser tool. The program then enters "eraser mode" and can only erase things until another tool is chosen. There is usually a just-plain-cursor tool on the palette to let the user return the cursor to a general-purpose pointer.

Modal tool works for both tools that perform actions on drawings—like an eraser—or for shapes that can be drawn—like ellipses. The cursor can become an eraser tool and erase anything previously entered, or it can become an ellipse tool and draw any number of new ellipses.

Modal tool is not bothersome in a program like Paint, where the number of drawing tools is very small. In a more advanced drawing program such as Adobe Illustrator, however, the modality is very disruptive because, as the user gets more facile with the cursor and the tools, the percentage of time and motion devoted to selecting and deselecting tools—the excise—increases dramatically. Modal tools are excellent idioms for introducing users to the range of features of such a program, but they don't usually scale well for experienced users of more sophisticated programs.

The difficulty of managing a modal tool application isn't caused by the modality as much as it is by the sheer quantity of tools. Or, more precisely, the efficiencies break down when the quantity of tools in the user's working set gets too large. A working set of more than about five modal tools tends to get hard to manage. If the number of necessary tools in Adobe Illustrator could be reduced from 24 to 8, for example, its user interface problems might diminish below the threshold of user pain.

To compensate for the profusion of modal tools, products like Adobe Illustrator use meta-keys to modify the various modes. The SHIFT key is commonly used for constrained drags, but Illustrator adds many non-standard meta-keys and uses them in non-standard ways. For example, holding down the ALT key while dragging an object drags away a *copy* of that object, but the ALT key is also used to promote the selector tool from single vertex selection to object selection. The distinction between these uses is subtle: If you click on something, then press the ALT key, you drag away a copy of it. Alternately, if you press the ALT key and *then* click on something, you select all of it, rather than a single vertex of it. But then, to further confuse matters, you must *release* the ALT key, or you will drag away a copy of the entire object. To do something as simple as selecting an entire object and dragging it to a new position, you

must press the ALT key, point to the object, press and hold the mouse button without moving the mouse, release the ALT key, then drag the object to the desired position! What were these people thinking?

Admittedly, the possible combinations are powerful, but they are very hard to learn, hard to remember and hard to use. If you are a graphic arts professional working with Illustrator for eight hours a day, you can turn these shortcomings into benefits in the same way that a race car driver can turn the cantankerous behavior of a car into an asset on the track. The casual user of Illustrator, however, is like the average driver behind the wheel of an Indy car: way out of his depth with a temperamental and unsuitable tool.

Adobe Illustrator is firmly rooted in the Macintosh world. One of the major errors that Adobe made in their Windows interface design was a refusal to take advantage of the benefits of the two-button mouse, something that comes cheap or free with Windows. Illustrator doesn't use the right mouse button at all. I suspect that someone in the company felt that interoperability with the Mac was more important—a bad notion, as I've discussed before. Adobe could have put all selection tools on the left button and all drawing tools on the right button, just for an example. Users could then go back and forth between drawing things and manipulating them just by deciding which mouse button to use, and, even better, each button would then have available to it three meta-keys: ALT, CTRL and SHIFT. Not taking advantage of the right mouse button was an error on their part.

Charged cursor

The second tool-manipulation drag-and-drop technique is what I call charged cursor.

With charged cursor, the user again selects a tool or shape from a palette, but this time the cursor, rather than becoming an object of the selected type, becomes loaded—or charged—with a single instance of the selected object. When the user clicks once on the drawing surface, an instance of the object is created—dropped, if you will—on the surface at the mouse-up point. Charged cursor doesn't work too well for tools, but it is nicely suited for graphic objects. PowerPoint, for example, uses it extensively. The user selects a rectangle from the graphics palette and the cursor then becomes a modal rectangle tool charged with exactly one rectangle.

Many common drawing programs work this way, but it is also very popular for graphic direct-manipulation idioms in programs that aren't normally thought of as drawing programs. A good example is Visual Basic. When the user clicks on one of the gizmos on the tool palette, the cursor becomes charged with that gizmo. The user then clicks again to create a single instance of it on a form. Borland's Delphi uses charged cursor too, but if you SHIFT-click on a gizmo in the palette, you get a modal tool instead for creating multiple instances of a gizmo. Nice touch.

In many charged cursor programs like PowerPoint, the user cannot always deposit the object with a simple click but must drag a bounding rectangle to determine the size of the deposited object. Some programs, like Visual Basic, allow either method. A single click of a charged cursor creates a single instance of the object in a standard size. The new object is created in a state of selection, so it is surrounded by grapples and ready for immediate precision reshaping and resizing. This dual-mode, allowing either a single click for a default-sized object or dragging a rectangle for a custom-sized object is certainly the most flexible and discoverable, and will satisfy most users.

I have seen charged cursor programs that forget to change the appearance of the cursor. For example, although Visual Basic changes the cursor to crosshairs when it's charged, Delphi doesn't change it at all. This is really silly: if the cursor has assumed a modal behavior—if clicking it somewhere will create something—it is imperative that it visually indicate this state. Charged cursor also absolutely demands good cancel idioms; otherwise, how do you harmlessly discharge the cursor?

Bomb sighting

As the user drags a master object around the screen, each drop candidate visually changes as it is pointed to, which indicates its ability to accept the drop. In some programs, the master object can instead be dropped in the spaces between other objects. I call this variant of drag-and-drop **bombardier**. Dragging text in Word, for example, is a bombardier operation, as are most rearranging operations.

The vital visual feedback of bombardier drag-and-drop is showing where the master object will fall if the user releases the mouse button. In master-and-target, the drop candidate becomes visually highlighted to indicate the potential drop, but in bombardier, the potential drop will be in some space where there

is no object at all. The visual hinting is something drawn on the background of the program or in its concrete data. I call this visual hint the bombsight.

Rearranging slides in PowerPoint's slide-sorter view is a good example of this type of drag-and-drop. The user can pick up a slide and drag it into a different presentation order. As you drag, the bombsight, a vertical black bar that looks like a big text edit caret, appears between slides. Word, too, shows a bombsight when you drag text. Not only is the loaded cursor moving, but you see a vertical gray bar showing the precise location, in between characters, where the dropped text will land.

One of my clients has a report generation program. You can rearrange the left-to-right order of the columns by clicking-and-dragging one of them. As you drag the outline of the column, a thick vertical-line bombsight shows up between the other columns, indicating where the column will be dropped.

Whenever something can be dragged-and-dropped on the space between other objects, the program must show a bombsight. Just like a drop candidate in master-and-target, it must visually indicate its candidacy.

Drag-and-drop problems and solutions

When we are first exposed to the drag-and-drop idiom, it seems pretty simple, but for frequent users and in some special conditions, it can exhibit problems and difficulties that are not so simple. As usual, the iterative refinement process of software design has exposed these shortcomings, and in the spirit of invention, clever designers have devised equally clever solutions.

Autoscroll

What interpretation should the program make when the selected object is dragged beyond the border of the enclosing application rectangle? The correct interpretation is, of course, that the object is being dragged to a new position, but is that new position inside or outside of the enclosing rectangle?

Let's take Microsoft Word for example. When a piece of selected text is dragged outside the visible text window, is the user saying, "I want to put this piece of text into another program" or is he saying, "I want to put this piece of text somewhere else in this same document, but that place is currently scrolled off the screen"? If the former, things are easy. If the latter, the application must scroll in the direction of the drag to reposition the selection at a distant,

not-currently-visible location in the same document. I call such scrolling **autoscroll**.

Autoscroll is a very important adjunct to drag-and-drop. Where you implement one, you will likely have to implement the other. Wherever the drop target can possibly be scrolled off screen, the program requires autoscroll.

> **Design tip:** Any scrollable drag-and-drop target must auto-scroll.

In early implementations, autoscrolling worked if you dragged outside the application's window. This had two fatal flaws, though. First, if the application was maximized, how could you get the cursor outside the app? And second, if you want to drag the object to another program, how can the app tell the difference between that and the desire to autoscroll?

Microsoft developed a very intelligent solution to this problem. Basically, they begin autoscrolling just *inside* the application's border instead of just *outside* the border. As the drag cursor approaches the borders of the scrollable window—but is still inside it—a scroll in the direction of the drag is initiated. If the drag cursor comes within three or four millimeters of the bottom of the text window, Word begins to scroll the window's contents upward. If the drag cursor comes equally close to the top edge of the text window, Word scrolls down. Unfortunately, Word's implementation doesn't take into account the power of the microprocessor, and the action occurs too fast to be useful on my (relatively slow) 486/66. Besides compensating for processor speed, a better way to implement this same idiom would be to use a variable autoscroll rate as shown in Figure 18-1, where the automatic scrolling increases in speed as the cursor gets closer to the window edge. For example, when the cursor is five millimeters from the upper edge, the text would scroll down at one line per second. At four millimeters, the text would scroll at two lines per second, and so on. This gives the user sufficient control over the autoscroll to make it useful. The autoscroll should never be unconstrained; computers are only getting faster.

Another important detail required by autoscrolling is a time delay. If autoscrolling begins as soon as the cursor enters the sensitive zone around the edges, it is too easy for a slow-moving user to inadvertently autoscroll. To cure this, autoscrolling should only begin after the drag cursor has been in the auto-scroll zone for some reasonable time cushion—about a half-second.
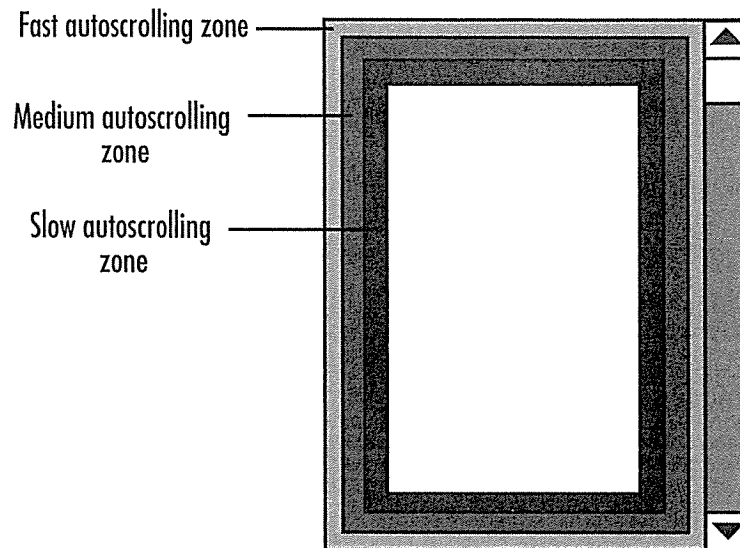
Fast autoscrolling zone

Medium autoscrolling zone

Slow autoscrolling zone

Figure 18-1

Microsoft unfortunately lets the scrolling go forth at whatever speed the computer is capable of, which is too fast to be useful, even on my 486/66. Not only should they put a maximum scroll limit on autoscroll, they should also make it graduated and user-controllable. It should autoscroll faster the closer the user gets to the edge of the window. To their credit, Microsoft's idea of autoscrolling as the cursor approaches the inside edges of the enclosing scrollbox, rather than the outside, is very clever indeed.

If the user drags the cursor completely outside the application's scrollable text window, no autoscrolling occurs. Instead, the repositioning operation will terminate in a program other than Word. For example, if the drag cursor goes outside Word and is positioned over PowerPoint when the user releases the mouse button, the selection will be pasted into the PowerPoint slide at the position indicated by the mouse. Furthermore, if the drag cursor moves within three or four millimeters of any of the borders of the PowerPoint edit window, PowerPoint begins autoscrolling in the appropriate direction. This is a very convenient feature, as the tight confines of contemporary video screens mean that we often find ourselves with a loaded drag cursor and no place to drop its contents—a very frustrating state and one that makes drag-and-drop less appealing in general.

Avoiding drag-and-drop twitchiness

When an object can be either selected or dragged, it is vital that the mouse be biased towards the selection operation. Because it is so difficult to click on something without inadvertently moving the cursor a pixel or two, the frequent act of selecting something must not accidentally cause the program to

misinterpret the action as the beginning of a drag-and-drop operation. The user rarely wants to drag an object one or two pixels across the screen. The time it takes to perform a drag is usually much greater than the time it takes to perform a selection, and the drag is often accompanied by a repaint, so objects on the screen will flash and flicker. This unexpected visual paroxysm can be very disturbing to users expecting a simple selection. Additionally, the object is now displaced by a couple of pixels. The user probably had the object just where he wanted it, so having it displaced by even one pixel will not please him. And to fix it, he'll have to drag the object back one pixel, a very demanding operation.

In the hardware world, controls like push-buttons that have mechanical contacts can exhibit what engineers call "bounce," in which the tiny metal contacts of the switch literally bounce when someone presses them. For electrical circuits like doorbells, the milliseconds the bounce takes aren't meaningful, but in modern electronics, those extra clicks can be significant. The circuitry backing up such switches has special logic to ignore extra transitions if they occur within a few milliseconds of the first one. This keeps your stereo from turning back off a thousandth of a second after you've turned it on. This situation is analogous to the oversensitive mouse problem, and the solution is to copy switch makers and **debounce** the mouse.

To avoid this situation, programs should establish what I call a **drag threshold**.

Essentially, all mouse-movement messages that arrive after the mouse button goes down and capture begins are ignored unless the movement exceeds some small threshold amount, say three pixels. This provides some protection against initiating an inadvertent drag operation. If the user can keep the mouse button within three pixels of the mouse-down point, the entire click action is interpreted as a selection command, and all tiny, spurious moves are ignored. The object has been debounced. As soon as the mouse moves beyond the three-pixel threshold, the program can confidently change the operation to a drag. This is shown in Figure 18-2. Anytime you have a situation where an object can be selected and dragged, the drag operation should be debounced.

Design tip: Debounce all drags.

The Program Manager in Windows 3.x has a one-pixel drag threshold, which is too small. It is far too easy to accidentally move an icon out of position when all you want to do is select it. Icons on the Windows 95 desktop appear to have a four-pixel debounce threshold.
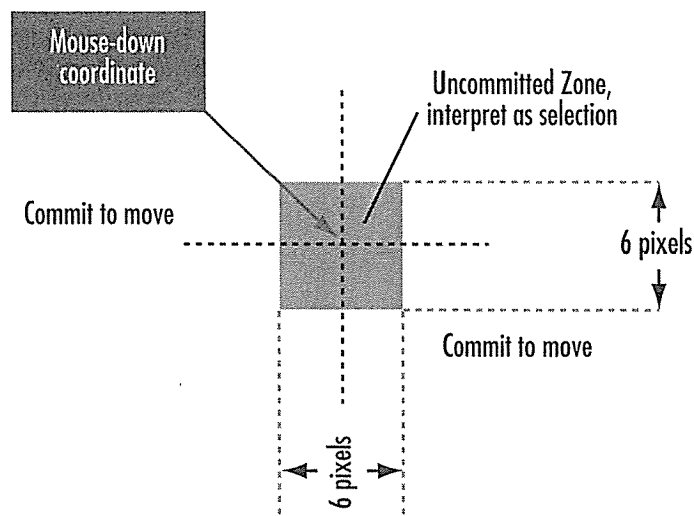
Figure 18-2

Any object that can be both selected and dragged must be "debounced." When the user
clicks on the object, the action must be interpreted as a selection rather than a drag, even
if the user accidentally moves the mouse a pixel or two between the click and the release.
The program must ignore any mouse movement as long as it stays within the uncommitted
zone, which extends three pixels in each direction. Once the cursor moves more than
three pixels away from the mouse-down coordinate, the action changes to a drag, and the
object is considered "in play." This is called a drag threshold, and it is used to debounce
the mouse.

My report-generator client (for whom we developed the column-reposition
bombsight) required more-complex drag threshold handling. The user could
reposition columns on the report by dragging them horizontally. The user
could put the FIRSTNAME column to the left of the LASTNAME column just by
dragging it into position. This was, by far, the most frequently used drag-and-
drop idiom. There was, however, another seldom-used technique. This one
allowed the values in one column to be interspersed *vertically* with the values
of another column, as shown in Figure 18-3. We wanted to follow the user's
mental model and enable him to drag the values of one column on top of the
values of another to perform this stacking operation, but this conflicted with
the simple horizontal reordering of columns. We solved the problem by differ-
entiating between horizontal drags and vertical drags. If the user dragged the
column left or right, it meant that he was repositioning the column as a unit.
If the user dragged the column up or down, it meant that he was interspersing
the values of one column with the values of another.

Because the horizontal drag was the predominant user action and vertical drags
were rare, we biased the drag threshold towards the horizontal axis. Instead of

Before

Name	Address	City
Ginger Beef	342 Easton Lane	Waltham
C. U. Lator	339 Disk Drive	Borham
Justin Case	68 Elm	Albion
Creighton Barrel	9348 N. Blenheim	Five Islands
Dewey Decimal	1003 Water St.	Freeport

Mouse down

Mouse up

After

Name	Address
Ginger Beef	342 Easton Lane Waltham
C. U. Lator	339 Disk Drive Borham
Justin Case	68 Elm Albion
Creighton Barrel	9348 N. Blenheim Five Islands
Dewey Decimal	1003 Water St. Freeport

Figure 18-3

This report-generator program offered an interesting feature that enabled the contents of one column to be interspersed with the contents of another by merely dragging-and-dropping it. This direct-manipulation action conflicted with the more-frequent drag-and-drop action of reordering the columns (like moving City to the left of Address). We used a special, two-axis drag threshold to accomplish this.

a square uncommitted zone, we created the spool-shaped zone shown in Figure 18-4. By setting the horizontal-motion threshold at four pixels, it didn't take a big movement to commit the user to the normal horizontal move, while still insulating the user from an inadvertent vertical move. To commit to the far-less-frequent vertical move, the user had to move the cursor eight pixels on the vertical axis without deviating more than four pixels left or right. The motion is quite natural and easily learned.

This two-dimensional thresholding can be used in other ways, too. Visio implements something similar to differentiate between drawing a straight and a curved line.

Mouse vernier

The weakness of the mouse as a precision pointing tool is readily apparent, particularly when dragging objects around in drawing programs. It is darned hard to drag something to the exact desired spot, especially when the screen resolution is 100 or more pixels-per-inch and the mouse is running at a six-to-one
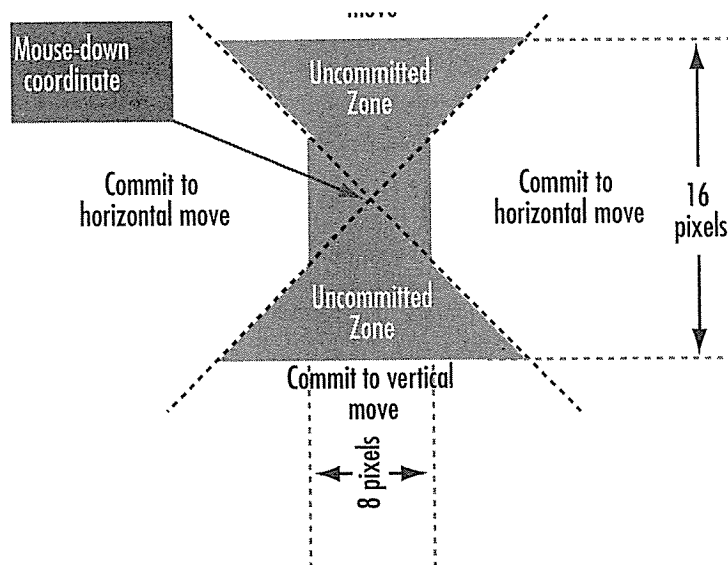
Figure 18-4

This spool-shaped drag threshold allowed me to create a bias toward horizontal dragging in a client's program. Horizontal dragging was, by far, the most frequently used type of drag in this application. This drag threshold made it difficult for the user to inadvertently begin a vertical drag. However, if the user really wanted to drag vertically, a bold move either up or down would cause the program to commit to the vertical mode with a minimum of excise. Before this method was instituted, a vertical move involved a nasty semi-permanent mode change by using a buttcon.

ratio to the screen. To move the cursor one pixel, you must move the mouse precisely one six-hundredth of an inch. Not easy to do.

This is solved by adding what I call a mouse vernier function, where the user can quickly shift into a mode that allows much finer-resolution for mouse-based manipulation of objects.

During a drag, if the user decides that he needs more precise maneuvering, he can change the ratio of the mouse's movement relative to the object's movement on the screen. Any program that might demand precise alignment must offer a vernier facility. This includes at a minimum all drawing and painting programs, presentation programs and image-manipulation programs.

Design tip: Any program that demands precise alignment must offer a vernier.

There are several acceptable variants of this idiom. A button can be pressed during the drag operation, like the ENTER key, and the mouse would shift into

vernier mode. In vernier mode, each ten pixels of mouse movement would be interpreted as a single pixel of object movement.

One method that seems popular is to make the arrow keys active during a drag operation. While holding down the mouse button, the user can manipulate the arrow keys to move the selection up, down, left or right one pixel at a time. The drag operation is still terminated by releasing the mouse button.

The problem with such a vernier is that the simple act of releasing the mouse button can often cause the user's hand shift a pixel or two, making the perfectly placed object slip out of alignment just at the moment of acceptance. The solution to this is, upon receipt of the first vernier keystroke, to "desensitize" the mouse. This is accomplished by making the mouse ignore all subsequent movements under some reasonable threshold, say five pixels. This means that the user can make the initial gross movements with the mouse, then make a final, precise placement with the arrow keys, and release the mouse button without disturbing the placement. If the user wanted to make additional gross movements after beginning the vernier, he would simply move the mouse beyond the threshold, and the system would shift back out of vernier mode.

If the arrow keys are not otherwise spoken for in the interface, as in a drawing program, they can be used to control vernier movement of the selected object without having to hold the mouse button down. This is nicely done in PowerPoint: the arrow keys move the selected object one step on the grid— about two millimeters using the default grid settings. If you hold the ALT key down while arrowing, the movement is one pixel per arrow click.

Part V: The Cast
The Actors in the Drama

Windows, menus, dialogs and push-buttons are the most visible trappings of the modern graphical user interface, but they are effects, rather than causes, of good design. They serve a purpose, and we have to understand how they fit into the designer's toolbox. More importantly, though, we must understand why each component exists and what purpose and effect they each have before we can profitably fit them into our creations.

The Meaning of Menus

The modern GUI with its pulldown menus and dialog boxes hasn't been around all that long—only since 1984—as a mainstream design idiom. Still, it is so ubiquitous that it is easy to take for granted. It is worthwhile for us to peer backwards and see the path we've taken in the development of the modern dialog and menu interface, not just to see how far we've come, but also to see how far we have yet to go.

The command-line interface

If you wanted to talk to an IBM mainframe computer in the 1970s, you had to manually keypunch a deck of computer cards, use an obscure language called JCL (job control language) to tell the computer how to read your program, and submit this deck of cards to the system through a noisy, mechanical card reader. Each line of JCL or program had to be punched onto a separate card. Even the first microcomputers, small, slow and stupid, running a primitive operating system called CP/M, had a much better conversational style than those hulking dinosaurs in their refrigerated glass houses. You could communicate directly with microcomputers running CP/M merely by

271

typing commands into a standard keyboard. What a miracle! The program issued a prompt on the computer screen that looked like this:

A>

You could then type in the names of programs, which were stored as files, as commands, and CP/M would run them. We called it the command-line interface, and it was widely considered a great leap forward in man-machine communications.

The only catch is that you had to know what to type. For frequent users, who at that time were mostly programmers, the command-line prompt was very powerful and effective because it offered the quickest and most efficient route to getting the desired task done. With his hands on the keyboard in the best tradition of touch-typists, the knowledgeable user could rip out "copy a:*.* b:" and the disk was copied. And today, if you possess the knowledge, the command line is still faster than using a mouse for many operations.

The command-line interface really separated the men from the nerds. The programmers of early desktop computers mostly just shrugged their shoulders and thought "if you wanna make an omelet, ya gotta break some eggs." As software got more powerful and complex, however, the memorization demands that the command-line interface made on users were just too great, and it had to give way to something better.

The hierarchical menu interface

Finally, sometime in the late '70s, some very clever programmer came up with the idea of offering the user a list of choices. He could read the list and select an item from it the way that you choose a dish at a restaurant by reading the menu. The appellation stuck, and the age of the hierarchical menu began.

The hierarchical menu enabled the user to forget many of the commands and option details required by the command-line interface. Instead of keeping the details in his head, he could read them off the screen. Another miracle! Circa 1979, your program was judged heavily on whether or not it was "menu-based." Those vendors stuck in the command-line world fell by the wayside in favor of the modern paradigm.

Although the paradigm was called "menu-based" at the time, I call it *hierarchical* menu-based to differentiate it from the menus in widespread use today. The old pre-GUI menus were deeply hierarchical: after making a selection from

one menu, you would be presented with another, then another and so on, drilling down into a tall tree of commands. Such menu-based interfaces would be judged terrible by today's standards. Their chief failing was the necessary depth of the hierarchy. This was coupled with a striking lack of flexibility and clarity in dealing with their users.

A typical menu would offer a half-dozen choices, each indicated by an ordinal from 1 to 6; the user would enter the number to select the corresponding option. Once the user made his selection, it was set in concrete—there was no going back. People, of course, made mistakes all of the time, so the more progressive developers of the day added confirmation menus. The program would accept the user's choice as before, then issue another menu to enquire: "Press the ESCAPE key to change your selection, otherwise press ENTER to proceed." This was an incredible pain in the butt, regardless of your choice. If you made a mistake and wanted to change your selection, you had to navigate through this clumsy meta-question, asking permission to ask a question. On the other hand, if you entered your response correctly, you still had to navigate through this clumsy meta-question.

Because only one menu at a time could be placed on the screen, and also because software at that time was still very heavily influenced by the batch-and-JCL style of mainframe computing, the hierarchical menu paradigm was very sequential in behavior. The user was presented with a high-level menu for choosing between major functions, for example:

1. Enter transactions

2. Close books for month

3. Print Income Statement

4. Print Balance Sheet

5. Exit

Once the user chose "1. Enter Transactions," he would then be prompted with another menu, subordinate to his choice from the first one, such as

1. Enter invoices

2. Enter payments

3. Enter invoice corrections

4. Enter payment corrections

5. Exit

The user would choose from this list and, most likely, be confronted with a couple more such menus before the actual work would begin. Then the "Exit" option would take him up only one level in the hierarchy. This meant that navigating through the menu tree was a real chore.

Still better than command lines, where you had to remember each operand, this technique lightened the user's memorization burden but forced him to laboriously navigate an archipelago of confusing choices and options, and it, too, had to give way to something better.

The Lotus 1-2-3 interface

The next great advance in user-interface technology came in 1979 from Lotus Corporation, with the original 1-2-3 spreadsheet program. 1-2-3 was still controlled by a deeply hierarchical menuing interface, but they added their own twist to it that helped make it the most successful piece of software ever sold up to that point. I call it a **visible hierarchical menu**.

Remember that computer screens in those days weren't capable of displaying high-resolution graphics the way they are today. In 1979, a computer screen offered exactly 2000 characters per screen, arranged in 25 horizontal rows of 80 characters each. 1-2-3 presented its menu horizontally along the top of the screen, where it consumed only two rows out of the 25 available. Take a look at Figure 19-1. This meant that the menu could coexist on the screen with the actual spreadsheet program. Unlike the hierarchical menu programs that came before it, the user didn't have to leave a productive screen to see a menu. He could enter a menu command right there.

Lotus used their new menu idiom with great abandon, creating a hierarchical menu structure of remarkable proportions, both in width and depth. There were dozens of nodes in the menu tree, and several hundred individual choices available. Each one could be found by looking at the top line of the screen and tabbing over and down to the desired selection. The program differentiated between data for the spreadsheet and a command for the menu by detecting the presence of a "/." If the user entered a slash, the keystrokes that

Figure 19-1

The original Lotus 1-2-3, which first shipped in 1979, exhibited a remarkable new menu structure that actually coexisted with the working screen of the program. You can see the menu at the second line down from the top of the screen. The highlighted word "Bar" is the currently selected menu item. All other menu-based programs at that time forced you to leave the working screen to make menu selections. Like all great ideas, this one was invisible in foresight and obvious in hindsight.

followed were interpreted as menu commands rather than data. To select an item on the menu, all you had to do was read it and type in its first letter. Sub-menus then replaced the main menu on the top line.

Frequent users quickly realized that the patterns were memorable, and they didn't necessarily have to read the menu. They could just type "SLASH-S" to save their work to disk. They could just type "SLASH-C-G-X" to add up a column of numbers. They could, in essence, bypass the use of the menu entirely. They became power-users, memorizing the letter commands and gloating over their knowledge of obscure functions.

It seems silly now, but it illustrates a very powerful point: that a good user interface enables its users to move in an ad hoc, piecemeal fashion from

beginner to expert. A given power-user of 1-2-3 might be on intimate terms with a couple of dozen functions, while simultaneously being completely ignorant of several dozen others. If he has memorized a particular slash-key sequence, he can go ahead and access it immediately. Otherwise, he can read the menu to find those less frequently used ones that he hasn't committed to memory.

But 1-2-3's hierarchical menu was hideously complex. There were simply too many commands, and every one of them had to fit into the single hierarchical menu idiom. The program's designers bent over backwards to make logical connections between functions in an attempt to justify the way they had apportioned the commands in the hierarchy. In the delirium of revolutionary success and market dominance, such details were easily ignored.

As you might imagine, 1-2-3's success in the mid '80s led to a time of widespread 1-2-3 cloning. The always-visible, hierarchical menu found its way into numerous programs, but the idiom was really the last gasp of the character-based user interface in the same way that the great, articulated steam locomotives of the late 1940s were the final and finest expression of a doomed technology. As surely as diesel locomotives completely eliminated all steam power within the span of a decade, the GUI eliminated the 1-2-3-style hierarchical menu within a few short years.

Monocline grouping

Hierarchies are one of the programmer's most durable tools. Much of the data inside programs, along with much of the code that manipulates it, is in hierarchical form, and many programmers offer hierarchies to the user in the interface. The early menus of personal software were hierarchical. But hierarchies are generally very difficult for users to understand and use. This basic fact is often difficult for programmers to grasp, as comfortable as they are with hierarchies.

Humans are familiar with hierarchies in their relationships, but they are not natural concepts for them when it comes to storing and retrieving information. Most storage systems outside of computers are very simple, composed either of a single sequence of stored objects, like a bookshelf, or a series of sequences, each one level deep, like a file cabinet. This method of organizing things into a single layer of groups is extremely common and can be found everywhere in your home and office. Because it never exceeds a single level of nesting, I call this storage technique **monocline grouping**.

Programmers are very comfortable with nested systems where an instance of an object is stored in another instance of the same object. Humans, on the other hand, generally have a very difficult time with the idea. Personally, I *love* hierarchies, but most users don't like them or work comfortably with them. Really complex manual storage systems get around this by comprising each level from a very different technology. In a file cabinet, you never see folders inside folders, Pendaflexes inside Pendaflexes or file drawers inside file drawers. Even the dissimilar nesting of folder-inside-Pendaflex-inside-cabinet rarely exceeds two levels of nesting.

Many people store their papers in a series of stacks based on some common characteristic: The Acme papers go here; the Project M papers go there; personal stuff goes in the drawer. Donald Norman, in *Things That Make Us Smart*, calls this a "pile cabinet." Normally, only inside computers would we put the Project M papers inside the Active Clients box, which, in turn, is stored inside the Clients box, stored inside the Business cabinet.

Computer science gives us the hierarchy as a tool to solve the very real problems of managing massive quantities of data. But when programmers render this implementation model as the manifest model, users get confused because it conflicts with their mental model of storage systems. Monocline grouping is the mental model the user typically brings to the software. Monocline grouping is so dominant outside the computer that the software designer violates it at his peril.

Admittedly, monocline grouping is an inadequate system for managing the large quantities of data we commonly find on computers, but that doesn't mean it is a bad *model*. The solution to this conundrum is to render the model as the user imagines it—as monocline grouping—but to provide the search and access tools that only a hierarchical organization can offer.

The Popup Menu

Many concepts and technologies had to come together to make the GUI possible: notably the mouse, memory-mapped video, powerful processors and popup windows. A popup window is a rectangle on the screen that appears, overlapping and obscuring the main part of the screen, until it has completed its work, whereupon it disappears, leaving the original screen behind, untouched. The popup window is the mechanism used to implement both pull-down menus and dialog boxes.

In a GUI, the menus are visible across the top row of the screen just like Lotus 1-2-3's visible hierarchical menu, but the resemblance ends there. The user points and clicks on a menu, and its directly subordinate list of options immediately appears in a small window just below it. This is called a **popup menu**.

The user makes a single choice from the popup menu by clicking once or by dragging and releasing. There's nothing remarkable about that, except that the menus generally go no deeper than this. The selection the user makes at the popup menu level either takes immediate effect or calls up a dialog box. The hierarchy of menus has been flattened down until it is only one level deep. In other words, it has finally become monocline grouping.

Arguably the most significant advance of the GUI menu was this retreat from the hierarchical form into monocline grouping. The dialog box, another use of the popup window, was the tool that simplified the menu. The dialog box enabled the software designer to encapsulate all of the sub-choices of any one menu option in a single, interactive container. With dialogs, the menu could flatten out tremendously, gathering all of the niggling details from further down the menu tree into a single dialog. The deeply hierarchical menu was a thing of the past.

Enough choices could be displayed on the main menu bar to organize all of the program's functions into about a half-dozen meaningful groups, each group represented by a one-word menu title. The menu for each group could be roomy enough to include all of the related functions. The need to go to additional levels of menus was made superfluous.

Of course, philistines and reprobates are always with us, and they have created methods for turning pulldown menus back into hierarchical menus. They are called pull-rights or cascading menus and, although they are occasionally useful, more often they merely tempt the weaker souls in the development community to gum up their menus for little gain. I'll discuss these in more detail in the next chapter.

The pedagogic vector

As the modern GUI evolved, two idioms developed that fundamentally changed the role of the menu in the user interface. These two idioms are direct manipulation and toolbars. The development of direct-manipulation idioms has been a slow and steady progression from the first days of graphical user

interfaces. Conversely, the toolbar was an innovation that swept the industry around 1989. Within a couple of years, virtually every Windows program sold had a toolbar covered with buttcons. Only a few years before, nobody had seen a toolbar.

I call each distinct technique for issuing instructions to the program a **command vector**. Menus are a command vector, as are direct manipulation and toolbar buttcons. Good user interfaces will conscientiously provide what I call **multiple command vectors**, where each function in the program has menu commands, toolbar commands, keyboard commands and direct-manipulation commands, each with the parallel ability to invoke a given command. This enables users of different skill sets and preferences to command the program according to their desires and abilities.

Both **direct-manipulation and toolbar**-buttcon command vectors have the property of being **immediate vectors**. There is no delay between pressing a buttcon and seeing the results of the function. Direct manipulation also has an immediate effect on the information without any intermediary. Neither menus nor dialog boxes have this immediate property. Each one requires an intermediate step, sometimes more than one.

In the same way that a stranger to town may take a roundabout route to her destination while a native will always proceed on the most economical path, experienced users of a program will commonly invoke a function with the most immediate command rather than one that requires intermediate steps. Naturally, the most frequently used commands in a program are those that migrate onto buttons on the toolbar. These functions are still supported by items on the menu—the menu command vector—where their use becomes increasingly the purview of beginners. Experienced users, however, gravitate toward the immediate vectors of buttcons and direct manipulation.

This bifurcation of usage along lines of experience is an important characteristic of software usage, and it affects how menus and dialog boxes are used. They are needed less and less for daily use, and have instead become a teaching tool for first-time and infrequent users.

The buttcons and other gizmos on the toolbar are usually redundant with respect to commands on the menu. Buttcons are immediate, while menu commands remain relatively slow and clunky. Menu commands have a great

advantage, however, in their English descriptions of the functions, and the detailed controls and data that appear on corresponding dialog boxes. This detailed data makes the menu/dialog command vector the most useful one for teaching purposes, which is why I call it the **pedagogic vector**.

One required element of effective pedagogy is the ability to examine and experiment without fear of commitment. The CANCEL button on each dialog box supports this well. Contrary to the user interface paradigms of just a few years ago, menus and dialog boxes have ceased to be the main method by which normal users perform everyday functions. Many programmers and designers haven't realized this fact yet, and they continue to confuse the purpose of the menu command vector. Its role is simply to teach new users and to remind those who have forgotten.

Design tip: Menus and dialogs are the pedagogic vector.

When a user looks at a program for the first time, it is often difficult for him to size up what that program can do. An excellent way to get an impression of the power and purpose of an application is to glance at the set of available functions by way of its menus and dialogs. We do this in the same way we look at a restaurant's menu posted at its entrance to get an idea of the type of food, the presentation, the setting and the price.

Understanding the scope of what a program can and can't do is one of the fundamental aspects of creating an atmosphere conducive to learning. Many otherwise easy-to-use programs put the user off because there is no simple, unthreatening way for him to find out just what the program is capable of doing.

The toolbar and other direct-manipulation idioms can be too inscrutable for the first-time user to understand or to even fit into a framework of possibilities, but the textual nature of the menus serves to explain the functions. Reading "Borders and Shading" is a heck of a lot more enlightening to the new user than trying to interpret a buttcon that looks like this:

For infrequent users who are somewhat familiar with the program, the menu/dialog vector's main task is as an index to tools: a place to look when he

knows there is a function but can't remember where it is or what it is called. This works the same way as its namesake brother, the restaurant menu, permits him to rediscover that delightful fish curry thing he ordered a year ago, without having to remember its precise name; the pulldown menu lets him rediscover functions whose name he's forgotten. He doesn't have to keep such trivia in his head but can depend on the menu to keep it for him, available when he needs it.

If the main purpose of menus were to execute commands, terseness would be a virtue. But because the main justification of their existence is to teach us about what is available, how to get it and what shortcuts are available, terseness is really the exact opposite of what we need. Our menus have to explain what a given function does, not just where to invoke it. Because of this, it behooves us to be more verbose in our menu item text. We shouldn't say "Open..." but rather "Open the Report...." We shouldn't say "Auto-arrange" but rather "Auto-arrange the icons." We should stay far away from jargon, as our menu's users won't yet be acquainted with it.

Many programs also use the status bar that goes across the bottom of their main window to display an even-longer line of explanatory text associated with the currently selected menu item. This idiom certainly enhances the teaching value of the command vector.

The pedagogic vector also means that menus must be complete, offering a full selection of the actions and facilities available in the program. Every dialog box in the program should be accessible from some menu option. A scan of the menus should make clear the scope of the program and the depth and breadth of its various facilities.

Another teaching purpose is served by providing hints pointing to other command vectors in the menu itself. Putting hints in that describe keyboard equivalents teaches users, as they work with the program more frequently, about quicker command methods that are available. By putting this information right in the menu, the user sees it subconsciously. It won't intrude upon his conscious thoughts until he is ready to learn it, and then he will find it readily available and already familiar.

Menus

20

In the last chapter, we discussed how menus fit into the grand scheme of user interface idioms. Now let's take a closer look at menus and talk about specifics.

Standard menus

HUH?

Menus are just about the hoariest idiom in the GUI universe—revered and surrounded by superstition and lore. We accept without question that traditional menu design is correct, because so many existing programs attest to its excellence. But this belief is like snapping your fingers to keep the tigers away. There aren't any tigers here, you say? See, it works!

Most every GUI these days has at least a "File" and an "Edit" menu in its two leftmost positions and a "Help" menu all the way over to the right. The Windows style guide states that these File, Edit and Help menus are standard. You might also think that this de facto cross-platform standard is

283

a strong indication of the proven correctness of the idiom. Wrong! It is a strong indication of the development community's willingness to blithely accept bad design, changing it only when the competition forces us to do better. My *least favorite menus are the File, Edit and Help menus*. The File menu is named after an accident of the way our operating systems work. The Edit menu is based on the very weak clipboard. And the Help menu is frequently the least helpful source of insight and information for the befuddled user.

The conventions of these three menus trap us into weak user interfaces for some pretty vital parts of our programs. Now, I'm not saying that we should discard conventions and scramble our menus on a whim. Rather, I think of changing these menus as life-saving surgery. I don't think that the fear of surgery should keep us away from the doctor too long.

I can hear the screaming already, programmers saying "How can you change something that has become a standard? People *expect* the File menu!" My answer is a simple one: People may get used to pain and suffering, but that is no reason to perpetuate it. Sure, prisoners of war often have a hard time adjusting to freedom after their release, but ask any former POW if he'd like to return to the camp and he'll laugh in your face. Users will adapt without significant problems if we change the File menu so that it delivers a better, more meaningful model. Changing only the menu items without accompanying this with significant changes to the model would indeed be the big mistake these programmers worry about.

Pathological manifest models aside, it is a Good Thing to group your program's functions into one of these more-or-less standard menus. As in all interface things, however, blindly following hard-and-fast rules will only make things worse. Microsoft did this in their latest release of their Office suite, falling victim to the Style Nazis. Every program has nearly identical menus. Their intent was good, but they went a little too far. In particular, PowerPoint suffered. It just doesn't gain much from having a menu structure similar to Excel and Word, and it loses quite a bit of its native ease-of-use by conforming to an alien structure. Face it, a presentation program just isn't the same as a word processor—forcing its menu to look like one can't be much help.

After the Windows style guide declares the File, Edit and Help menus as "standard," it then proceeds to mostly ignore several other de facto menu standards, like "Window," "View," "Insert," "Format," "Tools" and "Options." Just because they aren't common to all programs doesn't mean that they aren't

standard. The simple fact that the user has seen one of these menus before tells him something about the meaning of this instance's contents. This contributes to the trustworthiness of the application, which in turn encourages the user to explore and learn; and learning is the main purpose of the menu system.

The menus on most of our programs may be familiar, but are they good ways to organize functions? Words like "View," "Insert," "Format," "Tools" and "Options" sound like tools and functions, not goals. Why not organize the facilities in a more goal-directed way?

The correct menus

So what is the correct set of menus to have? What is the right way to classify the functions on them? I certainly wish I could answer these questions definitively, but I don't think anyone can. First, I don't believe that a definite answer exists. Second, you must consider the individual needs of the program under consideration. Third, we are fighting against the massive weight of established convention. Fourth, it would take years of development and iterative refinement to arrive at perfection. I do know, though, that our current standards are not even close yet.

But since I asked the question, it's only fair that I go out on a limb and present a framework for rethinking menus. It may not be right, but it should get your creative juices flowing, and who knows? Maybe we'll see some movement in menu design. We are close, but we are not yet where we should be.

Using every spatial and visual hint at our disposal, we should arrange the menus from left to right in some meaningful order. We could put Help in the far left position because it may well be used first. I don't think that is good, however, because we will generally not use it much after we get acquainted with the program. So putting Help in the far right position is better. We can depend on its location from program to program, where the menus to its left will certainly be different.

A reasonable sequence for the other menus would be to order them according to their scope. The most global items on the left, getting more and more specific as we move to the right. If we assume a document-centric program, we will find, in descending order of their scope, these topics: The *program*, the *document* and *pieces* of the document. For each of these components, we might have *properties*, *views*, *functions* and *access* to the outside world. Using this structure

as a framework, we would have a menu system that looks like the schematic in Figure 20-1.
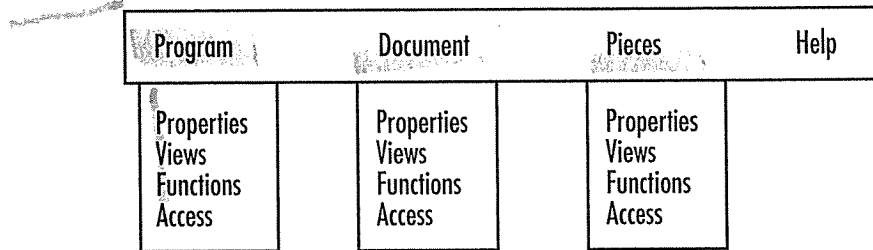
Program	Document	Pieces	Help
Properties Views Functions Access	Properties Views Functions Access	Properties Views Functions Access	

Figure 20-1

This is a highly stylized schematic of a suggested menu structure. On the far left is the topic of broadest scope: the program. Moving to the right, with diminishing scope as we go, is first the document, then the pieces that comprise the document. In descending order of importance are the four aspects of each component: Their properties, views of them, functions that operate on them and their access to the "outside world." In the Program menu, we can probably dispense with Views and Access. There is no reason that these schematic menus can't be productively broken into two or more menus. The Document menu, for example, could have Properties and Views on one and Functions and Access on another.

The program menus

The properties of a program include its default settings, what templates are available, and what modes it is in. It would include the configuration of standard interface idioms like toolbars. It would also include personalization items like colors and graphics.

Most likely, the menus for Views and Access could be omitted for the program, although if your program needed them, this would be the place to have them. Access might include items to save and restore program-wide settings.

The document menus

The next menu would cover the document under construction: what we are working on. The prime menu items would deal with the properties of the currently active document. This would include things like its size and type, its margin setup and page orientation.

Documents can certainly have different views, and this would be the place to set them. Things like draft versus presentation views; whether to show temporary guidelines, and what resolution to render images. This is also the place

where we would select which of the multiple open documents we wish to look at.

Next come functions that operate on an entire document at once. This would include operations like calculating spreadsheets and formatting text.

Access to the outside world at the document level is currently served by the top five items on most File menus. In our new framework, the bottom of the Document menu is the place where the user would go to open and close documents. The "outside world" includes the printer, fax and email, so access to those functions should reside here, as well.

This would also be the logical place to maintain the **most recently used** list of documents that we frequently see on the bottom of the File menu.

As you can see, the document menu is a big one, so most likely it would be broken into two or more popups. We could put properties and views on the first menu and functions and access on the second document menu.

Pieces of the document

The next menu to the right would cover the objects embedded in the document. If there are tables or images in the document, here is where the menu items that control them would reside. These menu items would only be active when the particular object they relate to is selected. And these objects don't necessarily have to be embedded by another program. In a drawing program, for example, they would be things like rectangles, ellipses and polylines. In a word processor, they would be the paragraphs of text and the headings, and the controls on this menu would be the "style sheets," and formatting controls.

Again, the first items cover properties of the object, such as its size and orientation. Control of its different views would follow. Often, objects have many possible transformations like formatting and rotation, and items for these functions would be next.

Last on this menu would be the ability to load and save objects from other documents or to and from disk.

The last menu, of course, would be the one that summons help for all or any of the others. Some programs would require another menu in between the Program and Document menus. Called "Group," it would house functions that operated on groups of documents. Assembling, formatting and printing chapters in a book would be an example of its power.

Meanwhile, back on Planet Earth

I don't have any illusions about the likelihood of seeing my new menu model getting implemented soon, so I'll return to reality and give you some more practical advice.

The File menu

In Chapter 8, I described a better "File" menu that was shown in Figure 8-4. Although I removed the SAVE function from the menu, I wouldn't dispense with it entirely. I'd just put it in some inconspicuous place for more advanced users to find. The program should save automatically for everyone else. The save function doesn't necessarily have to overwrite the original copy on disk the way it does now. It just needs to save the data in an easily recoverable way that is independent and invisible from within the application.

If we change from a file-centric view to this document-centric view, we should also change the name of the menu from "File" to "Document."

The Most Recently Used (MRU) list on Microsoft applications is an excellent shortcut idea and I recommend it. You can see it in Figure 20-2.

The Edit menu

The Edit menu contains facilities for cutting and pasting and importing and exporting. Don't use it as a catchall for functions that don't seem to fit anywhere else. Instead, gather them up into an Options or Preferences dialog that is accessible from the Tools menu.

The Windows menu

The Windows menu is for MDI only, providing a means for switching between MDI documents. It also offers tools for arranging multiple documents on screen simultaneously. Nothing else should go on this menu.

The Help menu

Today's Help menus are poorly designed reflections of poor help systems. We'll talk about help in Part VIII, but I would mention here that the Help menu sorely needs an item labeled "Shortcuts..." that would explain how to go beyond relying on the menus. It could offer pointers on more immediate idioms such as accelerators, toolbar buttons and direct-manipulation idioms.
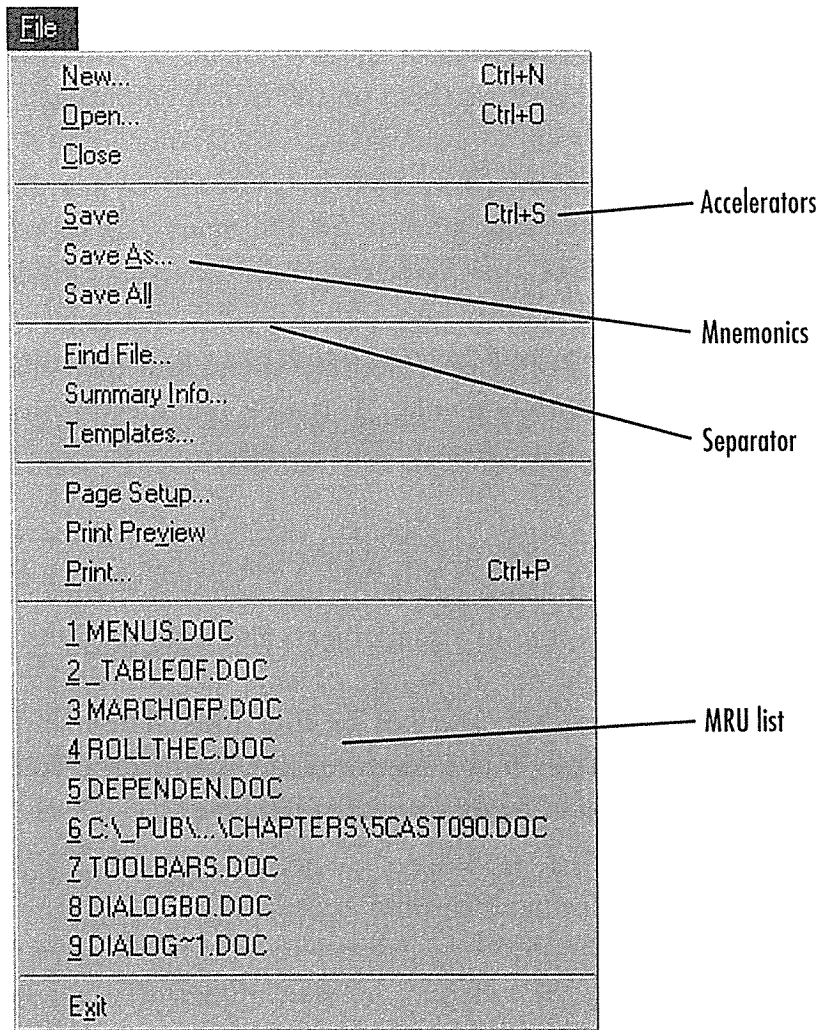
Figure 20-2

The File menu from Microsoft Word shows off the excellent Most Recently Used (MRU) list. In Chapter 8, I showed you how to reconstruct the first six items so that they better reflect the user's mental model, rather than following the technically faithful implementation model as shown here.

Optional menus

The View menu

The View menu should contain all options that influence the way the user looks at the program's data. Additionally, any optional visual items like rulers, templates or palettes should be controlled here.

The Insert menu

The Insert menu is really an extension of the Edit menu. If you only have one or two Insert items, consider putting them on the Edit menu instead and omitting the Insert menu entirely.

The Settings menu

If you have a Settings menu in your application, you are making a commitment to the user that anytime he wants to alter a setting in the program he will find the way to do it here. Don't offer up a Settings menu and then scatter other setting items or dialogs on other menus. This rule includes printer settings, which are often erroneously found on the File menu.

The Format menu

The Format menu is one of the weakest of the optional menus, as it deals almost exclusively with properties of visual objects and not functions. In a more object-oriented world, properties of visual objects are controlled by more-visual direct-manipulation idioms, and not by functions. The menu serves its pedagogic purpose, but you might consider omitting it entirely if you've implemented a more object-oriented format property scheme.

The page setup stuff that normally resides on the File menu should be placed here. Notice that page setup is very different from printer setup.

The Tools menu

The Tools menu, sometimes called options or functions, is where big, powerful transforms go. Functions like spell-checkers and goal-finders are considered tools. Also, the Tool menu is where what I call the hard-hat items go.

Hard-hat items are the functions that should only be used by real power users. These include various advanced settings. For example, a client/server database program has easy-to-use direct-manipulation idioms for building a query, while behind the scenes the program is composing the appropriate SQL statement to create the report. Giving power users a way to edit the SQL statement directly is most definitely a hard-hat function! Functions like these can be dangerous or dislocating, so they must be visually set off from the more benign tools available. In the past, I've segregated them from the other menu items and highlighted them with little hard-hat icons to indicate that they are for experts only.

Menu item variants

Disabling menu items

A defined Windows standard is to disable—or gray out—menu items when they are not relevant to the selected data item. Menus have robust facilities that make it easy to gray them out when their corresponding function is not valid, and you should take every advantage of this. The user will be well-served to know that the enabling and disabling of menu items reflects their appropriate use. This function helps the menu become an even more robust teaching tool.

Design tip: Disable menu items when they are moot.

It is important that each menu item clearly show when it is or isn't valid to fulfull its role as a teacher. Don't omit this detail.

Cascading menus

There is a variant of menus where a secondary menu can be made to pop up alongside a top-level popup menu. This technique, called **cascading menus**, was added to Windows in Version 3.1, so it has had only moderate penetration in interface design.

Where popup menus provide nice, monocline grouping, cascading menus move us into the nasty territory of nesting and hierarchies. Hierarchies are so natural to the mathematically inclined, but they are quite unnatural to the rest of us. The temptation to make menus hierarchical is nearly unavoidable for most programmers, who have a mathematical bent.

In the last chapter, we talked about how the modern GUI allowed us to leave hierarchical menus behind. It seems tragic to me that programmers would want to revive an idiom that lies happily in its grave. Cascading menus do serve a purpose: they allow lots of functions to be crammed onto a menu that would otherwise be way too long. There are occasionally enough items on a menu to justify putting some of the more obscure ones onto a second level, but I would consider it an idiom of last resort. I would make sure not to use cascading menus for anything that might be used frequently.
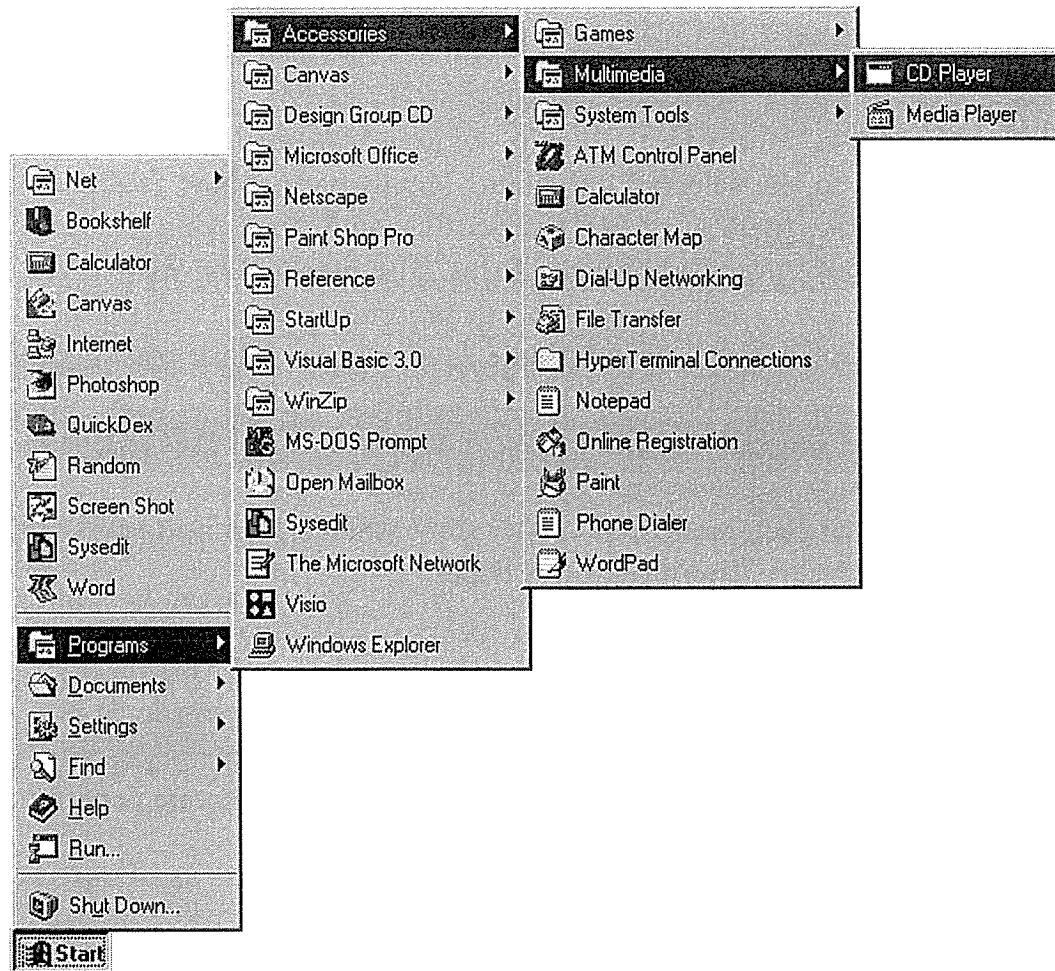
Figure 20-3

Here is a typical cascading menu on the Windows 95 Startbar. It's not very user friendly, but is quite software.dot.dweeb friendly. Hierarchies make logical sense to the programmers but rarely to users. Cascades also demand considerable skill with a mouse—putting them into real minnie territory—and this will frustrate infrequent users.

In Windows, it is difficult to say categorically that an idiom should not be used because the range of possible application software is so huge. Cascading menus, however, are a weak idiom, one that can be used as needed, but that should not be chosen before first considering other ways to solve the problem.

Windows 95 makes a tragically widespread use of cascading menus in the Startbar. The poor Start button is so overloaded with hierarchical menus that

even I find it jerky and unresponsive, and I'm a pretty good mouser. It really seems that Microsoft went to an incredible extreme to make double-clicks unnecessary. Figure 20-3 shows just how silly the cascading menus on the Windows 95 Startbar can get.

Flip-flop menu items

If the menu choice being offered is a binary one, that is, one that will be in either of two states, you might take advantage of a trick for saving menu space. For example, if you have two items, one called "Display Tools," and the other called "Hide Tools," you can create a single menu item that alternates between the two values, always showing the one currently *not* chosen. I call this technique a **flip-flop**.

This method saves space because otherwise it would require two menu items with mutually exclusive checkmarks. The flip-flop is a sucker's bet. As instructional clarity is the goal for menus, anything that obscures understanding is bad, and flip-flops can be very confusing for one simple reason: you can't tell if it is offering a choice or describing a state. If it says "Display Tools," does that mean tools are now being displayed or does it mean that by selecting the option you can begin displaying them? By combining roles, we make the meaning ambiguous. Although a menu is a list of functions and not a status display, a neophyte user can still easily get confused. If you can't label the states more unambiguously—"Display Tools Now"—then solve your space problem another way.

Graphics on menus

Visual symbols next to text items help the user to differentiate between them without having to read, so the items are understood faster. Because of this, adding small graphics to menu items can really speed users up. They also provide a helpful visual connection to other gizmos that do the same task. In particular, a menu item should show the same image as its corresponding toolbar buttcon.

Design tip: Parallel visual symbols on parallel command vectors

Microsoft Windows provides powerful tools for putting graphics in menus. Too few programs take advantage of this opportunity for providing an easy, visual learning trick. For example, the applications in Microsoft's Office suite all use a sheet of paper as an icon on a toolbar button to indicate the "New" function. Microsoft could put that same sheet of paper on the "File" menu next to the "New" menu item. The user would soon make the connection, probably without even thinking about it.

Microsoft PowerPoint has done an excellent job of incorporating teaching graphics into their menus in their latest release (4.0) as shown in Figure 20-4. Too bad Microsoft was only brave enough to use this excellent idiom on cascading menus and not on the normal popups, too.

Bang menu items

In the early days of Windows, a few smaller programs were shipped with a menu variant that has fallen out of favor, and for good reason. I'm referring to the top-level immediate menu item. Just as its name implies, it is a top-level menu item—on the horizontal menubar—that behaves like an immediate menu item on a popup; rather than displaying a popup menu for a subsequent selection, the immediate item causes the function to be executed right now! For example, an immediate menu item to compile some source code would be called "Compile!" In programmer's jargon, an exclamation mark is a "bang," and, by convention, top-level immediate menu items were always followed with a bang. Naturally, I call it a bang menu item, and bang it does!

Its behavior is so unexpected that it usually generates instant anger. The bang menu item has virtually no instructional value. It is dislocating and disconcerting. The same immediacy on a toolbar buttcon bothers nobody, though, and that is where immediate commands should stay. Surprisingly, this idiom resurfaces every once in a while.

Design tip: Don't use bang menu items.

Buttcons on a toolbar behave just like bang menu items: they are immediate and top-level. The difference is that buttcons on a toolbar advertise their immediacy because they are *buttons*. Menu items are things we trust to help us learn.
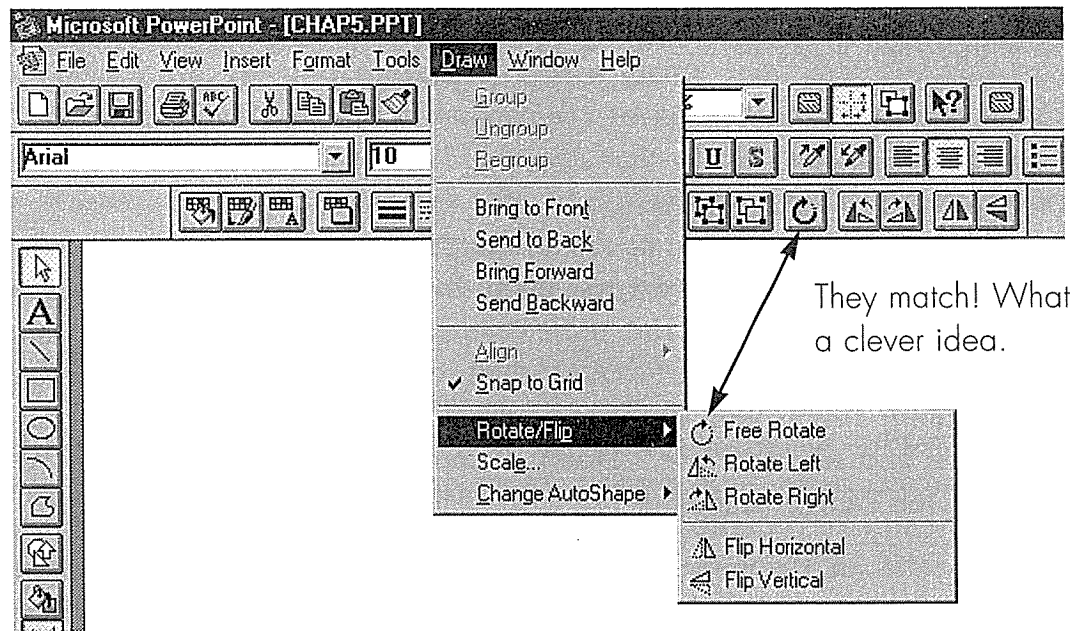
Figure 20-4

Microsoft PowerPoint offers us a regular smorgasbord of menu idioms. Most are toothsome, but some are a bit gamey. The Draw menu shows us disabled items, separators, cascades, menumonics, checks and vectors. The use of graphics on the little Rotate/Flip cascade menu is a nice implementation of visually linking menu items to other command vectors. The little graphic images are the same as those on the buttons that perform the identical tasks. What a wonderful way to build learning into the interface without it seeming pedantic or intruding on everyday usage. I'm sure Microsoft put those graphics in to better explain the functions and only inadvertently achieved the benefit of accelerating the user's growth to expertise by echoing the buttcon images. It's too bad they didn't put these cool little graphic dingbats on every menu item that has a corresponding buttcon.

So don't betray that trust by using bang menu items, please. Beware, however, because the capability to add them is still present in Windows.

Accelerators

Accelerators provide an additional, optional way to invoke a function from the keyboard. Accelerators are the keystrokes, which usually are a function key (like F9) or activated with a "CTRL," "ALT" or "SHIFT" prefix, that are shown on the right side of some popup menus. They are a defined Windows standard, but their implementation is up to the individual designer and they are often forgotten.

There are three tips for successfully creating good accelerators.

○ Follow standards

○ Provide for their daily use

○ Show how to access them
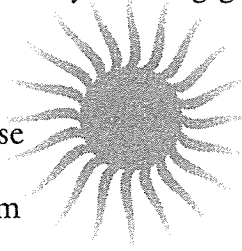
Where standard accelerators exist, use them. In particular this refers to the standard editing set as shown on the "Edit" menu. Users quickly learn how much easier it is to type CTRL+C and CTRL+V than it is to remove their mouse hand from the home row to pull down the Edit menu, select "Copy," then pull it down again and select "Paste." Don't disappoint them when they use your program. Don't forget standards like CTRL+P for print and CTRL+S for save.

Identifying the set of commands that will comprise those needed for daily use is the tricky part. You must select the functions likely to be used frequently and assure that those menu items are given accelerators. The good news is that this set won't be large. The bad news is that it can vary significantly from user to user.

The solution is to perform a triage operation on the available functions. Divide them into three groups: those that are definitely part of everyone's daily use; those that are definitely not part of anyone's daily use; and everything else. The first group must have accelerators, and the second group must not. The final group will be the toughest to configure, and it will inevitably be the largest. You can perform a subsequent triage on this group and assign the best accelerators, like F2, F3, F4 and so on to the winners of this group. More obscure accelerators, like ALT+7 should go to those least likely to be part of someone's everyday commands.

Don't forget to show the accelerator in the menu. An accelerator isn't going to do anyone any good if he has to go to the manual or online help to find it. Put it right there in the menu on the right side. Users won't notice it at first, but eventually they will and they will be happy to make the discovery. It will give them a sense of accomplishment and a feeling of being an insider. These are both good feelings well worth inducing in your customers.

Mnemonics

Mnemonics are another Windows standard for adding keystroke commands in parallel to the direct manipulation of menus and dialogs.

The Microsoft style guide covers both mnemonics and accelerators in detail, so I will just take this opportunity to stress that they should not be overlooked. Mnemonics are the underlined letter in a menu item. Entering this letter shifted with the ALT meta-key executes the menu item. The main purpose of mnemonics is to provide a keyboard equivalent of each menu command. For this reason, mnemonics should be complete, particularly for text-oriented programs. Don't think of them as a convenience so much as a pipeline to the keyboard. Keep in mind that your most experienced users will rely heavily on their keyboards, so to keep them loyal, assure that the mnemonics are consistent and thoroughly thought-out. Mnemonics are not optional.

For those designers among you who don't use mnemonics (I confess to be one, also), it is easy to put in bad mnemonics; to have non-unique characters within a menu, or to use really inappropriate and difficult-to-remember (thus becoming, by definition, non-mnemonic mnemonics) letters. Make sure that someone on the development or design team actually uses and refines the mnemonics.

The system menu

The **system menu** (inexplicably referred to as the "Control" menu in the style guide) is that standard little menu available in the upper left-hand corner of all independent windows. Curiously, it doesn't really do much. In Windows 3.x, there was a little box with a horizontal bar in it. In Windows 95, it is replaced by the program's icon.

Of all the menus, Microsoft has declared this one the most sacred: changing it is considered not, well, illegal, but akin to breaking the glass and ringing the fire alarm: You'd better have a darn good reason for it! Although I can't think of a single reason for changing it, I can imagine getting rid of it altogether.

Programs that use MDI give us two system menus because the document qualifies as a window and can be moved, minimized, maximized and so on just like its parent window.

Neil Rubenking claims that, in Windows 3.x, the horizontal bar in the main window's system menu box is actually a picture of the spacebar. You press ALT+SPACE to invoke it. He also says that the shorter horizontal bar in the MDI document window's system menu box is a picture of a hyphen, and—you guessed it—you can press ALT+HYPHEN to invoke it. If this is true, it is an example

of out-clevering yourself. The idea is a good one, but the execution fails badly. In Windows 95, the little bars are thankfully replaced by icons.

The irony of this sacred relic is just that: It is a relic. It serves no useful purpose. Originally, it was to be the home of system-level window management commands, but all of the initial ones have migrated to immediate gizmos on the other end of the caption bar, and no new ones have been added. I don't think that users actually *use* the system menu anymore, particularly since Windows 95 arrived with its handy immediate close box. The sole remaining purpose for the system menu is as a programming support for equivalent keyboard commands for moving, resizing, maximizing and minimizing the window. It would be no loss to the interface if the system menu were eliminated (as long as the keyboard commands were retained).

Further, the very existence of the system menu contributes to the general level of ambient confusion; It's just another lever on the mechanism with no evident purpose except to generate worry in the user's mind. When MDI programs give us two of these confusion generators, the uninitiated can be doubly misled.

Windows itself puts this menu on top-level and MDI windows, so the application designer doesn't really have much choice about it. But it would be nice if someone pointed out to the designers of Windows how truly useless this little appendix is.

Dialog Boxes

21

D ialog boxes are not part of the main program. If the program is the kitchen, the dialog box is its pantry. The pantry plays a secondary role, as does the dialog box. They are supporting actors rather than lead players, and although they may ratchet the action forward, they are not the engines of motion.

Suspension of normal interaction

Dialog boxes are superimposed over the main window of the owning program. The dialog box engages the user in a conversation by offering information and requesting some input. When the user has finished viewing or changing the information presented, he has the option of accepting or rejecting his changes. The dialog box then disappears and returns the user to the main program.

Many users and practitioners think of dialog boxes as the primary user interface idiom of the GUI. Many applications

299

have dialogs that provide the main method of interaction with the program (I'm not speaking of those smaller programs that are composed of just a single dialog box; in those cases, the dialog assumes the role of a main window). The user is constantly bouncing back and forth between the program's main window and its dialog boxes for no apparent reason.

When the application presents a dialog box, it is temporarily moving the action out of the mainstream; abandoning the main plot to develop a secondary issue. It is taking the focus of the dinner party away from the table and turning it onto the preparation of the food. It may be crucial, but it is not the main point.

Put primary interaction on the primary window

This understanding that dialog boxes are suspensions of normal processing is the key to their proper design. The main interaction of the program should be developed right on the main window of the program, while dialog boxes should be used only for secondary interaction.

If you asked your dinner party to temporarily abandon their soup and step into the kitchen, the smooth flow of conversation and warm friendship would be broken. In the same way, a dialog box breaks the smooth flow of rapport between a user and the program. Dialogs, for good or ill, interrupt the interaction and make the user react to the program instead of driving it.

Design tip: Dialogs break flow.

Dialogs boxes are appropriate for any functions or features that are out of the mainstream of interaction. Anything that is confusing, dangerous or rarely used can profitably be placed on a dialog box. I use the term **dislocating** to describe functions that make immediate and gross changes to the screen image. Such changes can be visually disturbing to the user and should be cordoned off from users unfamiliar with them. For this reason, dialogs are tools well-suited to managing dislocating actions.

Dialog boxes are good for presenting infrequently used functions and settings. The dialog box serves to isolate them from the more frequently used functions and settings. The dialog box is generally a roomier setting to present controls than toolbars or other primary control venues, so you can take a more leisurely approach to arranging and showing buttons and other gizmos. You have more space for explanatory labels than you do in a toolbar, for example.

Dialog boxes are also well-suited for concentrating information related to a single subject, such as the properties of an object in an application—an invoice or customer, for example. They can also gather together all information relevant to a function performed by a program—printing reports, for example. The dialog box, when used in this way, becomes an encapsulation tool, enabling you to box up and remove functions and settings that might have a dislocating or dangerous effect on the program from the normal flow of events. For example, a dialog box that allows wholesale reformatting of a document should be considered a dislocating action. The dialog helps prevent this from being invoked accidentally by assuring that a big, friendly CANCEL button is always present, and also by providing the space to show more protective and explanatory information along with the risky controls. The dialog can graphically show the user the potential effects of the function with a picture of what the changes will look like.

Most dialogs are invoked from a menu, so there is a natural kinship between menus and dialogs. As discussed in the last chapter, menus provide the pedagogic command vector—their primary purpose is to teach users about the program. By extension, dialog boxes also frequently play a part in the pedagogic vector.

Dialog boxes serve two masters: the frequent user who is familiar with the program and uses them to control its more advanced or dangerous facilities; and the infrequent user who is unfamiliar with the scope and use of the program and who is using dialogs to learn the basics. This dual nature means that dialog boxes must be compact and powerful, speedy and smooth, and yet be clear and self-explanatory in use. These two goals may seem to contradict each other, but they can actually be useful complements. A dialog's speedy and powerful nature can contribute directly to its power of self-explanation.

Dialog box basics

Most dialogs have buttons, comboboxes and other gizmos on their surface and, although there are some rudimentary conventions, generally the designer places them as she sees fit and not according to any conventional plan. The dialog's window may or may not have a caption bar or thickframe.

All dialog boxes have an owner. Normally, this owner is an application program—usually the one that created it—but it can also be the Windows system itself. Dialog boxes are always placed visually on top of their owning program, although the windows of other programs may obscure them.

Every dialog box has at least one **terminating command**, a control that, when activated, causes the dialog box to shut down and go away.

Generally, most dialogs will offer at least two push-buttons as terminating commands, OK and CANCEL, although the closebox in the upper right corner (upper left corner in Windows 3.x) is also a terminating command idiom.

It is technically possible for dialogs to not have terminating commands. Some dialogs are unilaterally erected and removed by the program—for reporting on the progress of a time-consuming function, for example—so their designers may have omitted terminating commands. This is bad design for a variety of reasons, as we will see.

Modal dialog boxes

There are two types of dialog boxes: modal and modeless. **Modal dialog boxes** are, by far, the most common variety.

Once the box comes up, the owning program cannot continue until the dialog box is closed. It stops the proceedings, which is how it got its name. Clicking on any other window belonging to the program will get the user only a rude "beep" for his trouble. All of the controls and objects on the surface of the owning application are deactivated for the duration of the modal dialog box. Of course, the user can activate *other* programs while a modal dialog box is up, but the dialog box will stay there indefinitely. And, when the owning program is reactivated, the modal dialog box will still be there waiting.

In general, modal dialogs are the easiest for users (and designers) to understand. The operation of a modal dialog is quite clear, saying to the user, "Stop what you are doing and deal with me now. When you are done, you can return

to what you were doing." It is a classic subroutine: a PERFORM, a GOSUB, a FUNCTION call. It is thus ideally suited for most functions like summarizing and printing. The rigidly defined behavior of the modal dialog means that, although it may be abused, it will rarely be misunderstood. There may be too many of them and they may be generally weak or stupid, but their purpose and scope will usually be clear to the user. Like death and taxes, you may not like modal dialog boxes, but you grasp their meaning.

If a modal dialog box is function-oriented, it usually operates on the entire program or on the entire active document. If the modal dialog box is process- or property-oriented, it usually operates on the current selection. In any case, you can't change the selection once you've summoned the dialog. This is the biggest difference between modal and modeless dialogs.

Actually, because modal dialog boxes only stop their owning application, they are more precisely named **application modal.**

It is also possible to create a dialog box, called **system modal**, that brings every program in the system to a halt. No application program should ever create one of these. Their only purpose is to report truly catastrophic occurrences that affect the entire system, such as the hard disk melting.

> **Design tip:** Never create a system modal dialog box.

Modeless dialog boxes

The other variety of dialog box is called **modeless**. They are less common than their modal siblings, and they are more misunderstood, too.

Once the modeless box comes up, the owning program continues without interruption. It does not stop the proceedings, and the application does not freeze. The various facilities and controls, menus and toolbars of the main program remain active and functional. Modeless dialogs have terminating commands, too, although the conventions for them are far weaker and more confusing than for modal dialogs.

A modeless dialog box is a much more difficult beast to use and understand, mostly because the scope of its operation is unclear. It appears when you summon it, but you can go back to operating the main program while it stays around. This means that you can change the selection while the modeless

dialog box is still visible. If the dialog acts on the current selection, you can select, change, select, change, select, change all you want.

In some cases, you can also drag objects between the main window and a modal dialog box. This characteristic makes them really effective as tool or object palettes in drawing-type programs.

The modeless dialog problem

I am unhappy with the way most modeless dialogs are currently implemented. Their behavior is inconsistent and confusing. They are visually very close to modal dialog boxes but are functionally very different. There are few established behavioral conventions for them, particularly with respect to terminating commands, and Microsoft is setting a disturbing precedent with terminating buttons that change legends contextually a poor construct.

Most of the confusion arises because we are more familiar with the modal form—and because of inconsistencies that arise in the way we use dialogs. When we see a dialog box, we assume that it is modal and has modal behavior. If it is modeless, users must tentatively poke and prod at it to determine how it behaves. There is just no clear archetype for it.

More confusion creeps into the situation because users are so familiar with the behavior of modal dialogs. A modal dialog can fine-tune itself for the selection in the program's main window at the instant it was summoned. It can do this with the sublime assurance that the selection won't change during its lifetime. Conversely, the selection is quite likely to change during the lifetime of a modeless dialog box. Then what should the dialog do? For example, if the modeless dialog box modifies text, what should it do if we now select some non-text object on the main window? Should gizmos on the dialog box gray out? Freeze up? Disappear? Should the dialog box just stay there with all of its gizmos "active" but having no effect if they are pushed? All of these options have been tried, and although each one has advantages, it is not clear which help and which hinder us. We'll take a closer look in the next few pages.

Modeless dialog boxes also lead us into situations that aren't, well, right. For example, in Word, request the modeless Find dialog box from the Edit menu. Now, from the Format menu request the modal Font dialog. Voilá! You now have a *modal* dialog box sitting on top of a *modeless* dialog box, each supporting its own, totally unrelated, functions. The modeless Find dialog is function-oriented, while the modal Font dialog is property-oriented. Functionally, there

is nothing wrong about this situation, but visually—comprehensibly—it is a nonsensical juxtaposition of unrelated dialogs. Is this helpful? Should such a circumstance be allowed to arise? I can't categorically say no, but I do think it is weak and confusing. The simple answer would be to eliminate modeless dialog boxes entirely, but that would be cutting off our nose to spite our face.

Two solutions

I'll take a stand right here and say that a solution must be found for the modeless dialog box problem. In fact, I'll offer two solutions. The first one is easy to swallow—an evolutionary step forward from our present peccadillo. The second one is more radical—a revolutionary leap. As you might suspect, the first solution is less thorough and effective than the second one. You might also guess—correctly—that I'm more fond of the revolutionary leap.

The evolutionary solution

In the evolutionary solution, we leave modeless dialog boxes pretty much the way they are, but we adopt two guiding principles and apply them consistently to all modeless dialog boxes. The first principle says that we must visually differentiate modeless dialog boxes from modal ones. The second principle says that we must adopt consistent and correct conventions for the terminating commands.

> ***Design tip:*** Visually differentiate modeless dialogs from modal dialogs.

If a programmer uses the standard modeless dialog box facility in the Windows API, the resultant dialog is visually indistinguishable from a modal one. We must break this habit. The designer must assure that all modeless dialog boxes are rendered with a clearly noticeable visual difference. A good method would be to use a distinctive hue for the dialog's background, or to add a pattern to it like desktop wallpaper. You can provide a colored border around the window or insert a colored stripe across its corner. You can change all of the buttons to visually distinct buttcons: make them a different shape or color or use a distinctive font.

You can visually differentiate a modeless dialog box by radically changing its shape: orient them all vertically instead of the usual horizontal shape. There are things you can do to the caption bar to set it apart visually, things like making it thicker or thinner, adding symbols or patterns to it, or animating it.

Whatever method you choose, you must stick with it consistently. It would be nice if vendors used a standard common to all, but that is wishful thinking. It will still be a significant improvement if each vendor adheres to his own, company-wide, standards for modeless dialog boxes.

> **Design tip:** Give modeless dialog boxes consistent terminating commands.

The other area where developers must follow consistent conventions is in the design of modeless dialog box terminating commands. Currently, this is one of the most inconsistent areas. It seems that each vendor, sometimes each programmer, uses a different technique on each individual dialog box. I simply do not see any reason for this cacophony of methods. Some dialogs say CLOSE, some say APPLY, some use DONE, while some DISMISS, ACCEPT, YES, and some even use OK. The variety is endless. Still others dispense with terminating buttons altogether and rely only upon the close box in the upper right corner (upper left corner in Windows 3.x). Terminating a modeless dialog box should be a simple, easy, consistent idiom, very similar—if not exactly the same—from program to program.

One of the most obnoxious constructions I've seen is terminating buttons that change their legend from CANCEL to APPLY, or from CANCEL to CLOSE depending on whether the user has taken action with the modeless dialog box. Microsoft's applications do this with frightening frequency. This changing is, at best, disconcerting and hard to interpret and, at worst, frightening and inscrutable. These legends should *never* change. If the user hasn't selected a valid option but presses OK anyway, the dialog box should assume the user means "dismiss the box without taking any action," for the simple reason that that is what the user actually did. Modal dialog boxes offer us the ability to cancel our actions directly, with the CANCEL button. Modeless dialogs don't usually allow this direct idiom—we must resort to UNDO—so changing the legends to warn the user just confuses things.

> **Design tip:** Never change terminating button captions.

The cornerstone of the cognitive strength of modal dialog boxes are their rigidly consistent OK and CANCEL buttons. The problem is that there is no equivalent for modeless dialog boxes. Modally, the OK button means "accept my input and close the dialog." Because the controls on a modeless dialog box are

always live, their equivalent concept is clouded in confusion. The user doesn't conditionally configure changes in anticipation of a terminal "execute" command like he does for a modal dialog box. Modally, the CANCEL button means "abandon my input and close the dialog." But because the changes made from a modeless dialog box are immediate—occurring as soon as an activating button is pressed—there is no concept of "cancel all of my actions." There may have been hundreds of separate actions on a number of selections. The proper idiom for this is the UNDO function, which resides on the toolbar or Edit menu and is active application-wide for all modeless dialog boxes. This all fits together logically, because the UNDO function is unavailable if a modal dialog box is up, but is still usable with modeless ones.

The only consistent terminating action for modeless dialog boxes is CLOSE or GO AWAY. Every modeless dialog box should have a CLOSE button placed in a consistent location like the lower right corner. It would have to be consistent from dialog to dialog: in the exact same place and with the exact same caption. Not to put too fine a point on this, but the word CLOSE is the one to use, and it should never deactivate or change its caption.

If the CLOSE button activates a function in addition to shutting the dialog, you have created a modal dialog box, and it should follow the conventions for that idiom instead.

Don't forget that modeless dialog boxes will frequently have several buttons that immediately invoke various functions. The dialog box should not close when one of these function buttons is pressed. It is modeless because it stays around for repetitive use and should only close when the single, consistently placed CLOSE button is pressed.

Another point is that modeless dialog boxes must be incredibly conservative of pixels. They will be staying around on the screen, occupying the front and center location, so they must be extra careful not to waste pixels on anything unnecessary.

A more radical, but better, solution

What I proposed earlier was a series of baby steps; an interim solution. What I will now propose is a more sweeping and radical solution, but one that delivers us from the full panoply of modeless-dialog maladies in one fell swoop. As I describe this solution, remember that idioms like toolbars and tabbed dialogs

were perceived as quite radical when they first appeared, but they are now widely accepted as normal. Here goes...

We currently have two modeless tool facilities in common use. The modeless dialog box is the older of them, but it is a clumsy and ineffectual one for all of the reasons outlined above.

The other modeless tool facility is a newcomer on the user interface scene, but it has achieved an unprecedented success. I'm referring, of course, to toolbars and buttcons. The toolbar idiom is only about five years old, but it has achieved a widespread success because of its demonstrable quality and convenience. *Well, it is nothing more than a modeless dialog box permanently attached to the top of the program's main window.*

The modelessness of toolbar buttcons is perfectly acceptable because they are not delivered to us in the familiar visual form of the dialog. Instead, they are visually presented as something clearly different. The row of omnipresent tools surrounding the workspace, without the usual trappings of dialog boxes, assures that they won't be confused with modal dialog boxes.

The buttcons and other tools on the toolbar are happy in their modeless role. We select something—text, say—and press the ITALIC buttcon; then we select something else, press ITALIC, select, press, select, press. We have no trouble understanding their scope even though it often confounds us when the same thing is positioned on a modeless dialog box.

If—depending on what is currently selected—a menu item can have no effect, it grays-out and deactivates. Toolbar buttcons can do the same, or they can merely take the "just ignore it" approach. However, when a buttcon's function becomes meaningless in the context, it should at least become unresponsive by not offering the pliant response. In other words, the buttcon should not visually depress.

Toolbars are just as modeless as modeless dialog boxes, but they don't introduce the conundrums that the dialogs do. They offer two characteristics that modeless dialog boxes don't: They are visually different from dialog boxes, and they have a consistent idiom for coming and going. They solve our other big problems, too. Toolbars are incredibly efficient in video space, particularly compared to dialog boxes.

I'm a firm believer in the principle that things that behave differently should look different. GUIs communicate visually, and we squander opportunity when we don't visually differentiate different things.
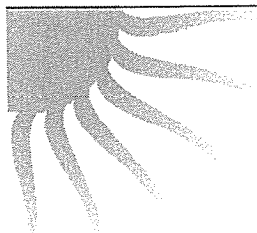
Things that behave differently should look different

If we make modeless dialog boxes look very different from modal dialog boxes, we will have solved half of our problem. Making our modeless dialog boxes into toolbars accomplishes this very effectively.

Now, you are probably thinking that the toolbar idea is good as far as it goes, but modeless dialog boxes are free-floating things that the user can position on the screen wherever he likes. Our friends in Redmond have created the perfect solution for this problem: the floating toolbar, sometimes called a floater. In all of Microsoft's current crop of applications, you can click-and-drag on any toolbar and pull it out away from the edge of the program, and it will instantly convert into a floater. A floater is a toolbar that isn't docked on one of the four edges of the program's main window.

A floater looks exactly like a docked toolbar, except that it has a thickframe for resizing and a mini-caption bar. A mini-caption bar is just what it sounds like: a caption bar that isn't as tall as a regular caption bar. It is about half the height of a normal one but is otherwise identical in operation and appearance.

The mini-caption bar, as shown in Figure 21-1, first appeared on Visual Basic's tool palette. The mini-caption bar hasn't achieved much currency in the industry for several reasons. First, it isn't standard and there is no easy way to get one. In code, you have to descend to the event-loop level and subclass the window, then perform some undocumented and non-standard actions to fool Windows into imagining that everything is normal. Because of the implementation hurdles and the lack of generally accepted usage conventions, no one is compelled to employ the idiom.
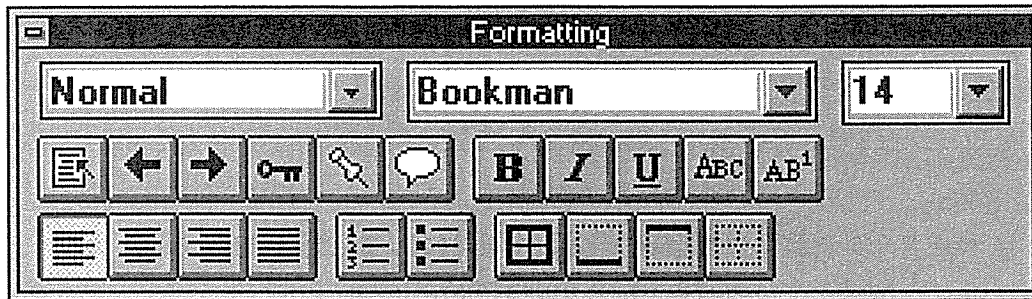
Figure 21-1

Here's a picture of a floating toolbar from Microsoft Word. But wait! This is a modeless dialog box! The mini-caption bar gives it a visual appearance distinct from modal dialog boxes, and the apparent conundrum of contextually inactive buttons is bothersome to nobody. If all modeless dialog boxes were rendered this way, much of their confusion would disappear. What's more, if you drag this floating toolbar to an edge of the application, it docks on that edge as a familiar, fixed toolbar. Imagine if you could do that with any modeless dialog box—the Find dialog, for example?

If we went ahead and gave all modeless dialog boxes mini-caption bars, we would immediately solve the visual differentiation problem. Look again at Figure 21-1. Notice that it is just a toolbar from Microsoft Word that has been undocked. It is normally docked in a horizontal row at the top of the main window, just below the menu bar. Floating toolbars can be docked merely by dragging them to one edge of the main window, whereupon they attach themselves to it as a fixed toolbar, and the mini-caption bar disappears.

Now let's turn the tables. Imagine Word's Find dialog, shown in Figure 21-2, rendered as a floating toolbar. It would have a mini-caption bar instead of its normal one. It would lack a terminating button, relying instead on the close box in the mini-caption bar. What would happen if we were to drag this new Find dialog to the upper edge of the main window? If its behavior were consistent, it would dock: the gizmos on the surface of the Find dialog would distribute themselves in a horizontal toolbar the way the Format toolbar does. If it works for all of those Format buttons and comboboxes, why can't it work for the Find dialog? Why can't we have a toolbar with a buttcon for FIND NEXT and with checkboxes for the various options?

Microsoft has made the floating/docking toolbar idiom a standard in the latest release of its Office suite. The programs all include a facility for customizing the toolbars to the user's taste. I not only think that this is a fine step in developing the user interface but that it should be used as the new idiom for replacing modeless dialog boxes.
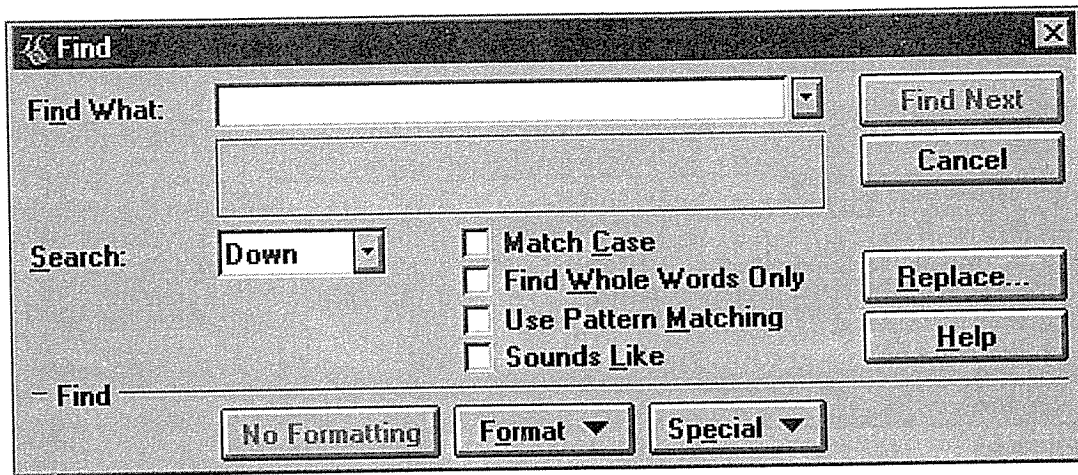
Figure 21-2

Here's a typical, state-of-the-art modeless dialog box. What a mess! It is big, obscures the text it needs to search within, and its buttons are clear as mud. That CANCEL one, for example: what does it cancel? Where can I tell this box to go away? Why can't functions like FIND be built into the main window interface, anyway? These aren't merely rhetorical questions; read the text for some real answers.

Property dialog boxes

The concepts of modal and modeless are derived from programmers' terms. They affect our design, but we must also examine dialogs from a goal-directed point of view. In that light, there are four fundamental varieties of dialog box which I call property, function, bulletin and process.

A property dialog box presents the user with the settings or characteristics of a selected object and enables the user to make changes to these characteristics. Sometimes the characteristics may relate to the entire application or document, rather than just one object.

The Font dialog box in Word, shown in Figure 21-3 is a good example. The user selects some characters, then requests the dialog box from the menu. The dialog enables the user to change font-related characteristics of the selected characters. You can think of property dialogs as a control panel with exposed configuration controls for the selected object. Property dialog boxes are usually modal. However, it is not uncommon for them to be modeless.