can be thought of as an operation on one object that brings into view another, related object. From the user's point of view, there should not be, and there is no need for there to be, any distinction between operating-system operations and applications operations.

That the interfaces of all applications arise from a small set of elementary operations confirms that the applications themselves, as rich and varied as they are from a task-oriented point of view, are not all that different from one another from an interface-oriented point of view. This fundamental similarity can be exploited to create powerful computer systems of unprecedented simplicity and productivity.

First, we need to define several methods for choosing and marking content on which we want to operate. Those methods are discussed in Section 5-2-1.

### 5-2-1 Highlighting, Indication, and Selection

**Highlighting** is adding, by any means, a recognizable distinction to a displayed object. The function of the highlight is to allow the user to determine, by passive observation, that the system has recognized a particular object as having a special status. The semantics of that status are dependent on the nature of the object and on the commands that the user can apply to it. For sighted users, the highlight is usually visual. Examples of visual highlighting mechanisms include brightness reversal, color change, change of contrast, underlining, blinking or other periodic change, and applying a stationary or animated border to an object. Nonvisual highlighting can be shown, for example, by different choice of voices or spoken inflection.

As the user moves the cursor over objects, the object to which the cursor is pointing should be highlighted. In text, the object typically would be an individual character. Highlighting the single object pointed to as a cursor is moved, without any other user action such as clicking, is **indication**. With indication, the user knows at all times to what object the system thinks she is pointing. In too many present systems, the user must guess at what will be selected or activated when she clicks the GID button; if her guess is incorrect, she will have to try again, wasting time and energy. Indication can be especially valuable when the objects that the user would like to select are small and close together, when they overlap, or when their limits are unclear. Indication is necessary if an interface is designed in accord with the principle of visibility.

The highlighting used for indication must not exhibit too much contrast or be overly dramatic, lest the flickering of objects become annoying as

the cursor moves over them; in some situations, it may be helpful not to indicate objects when the cursor is moving faster than a threshold velocity. Note that a smaller object, as measured by the visual angle the indicated object subtends, requires higher visual contrast of the indication, but this is an ergonomic issue.

Indication is underused in present systems. If you use indication aggressively in your interface design, you can eliminate a good deal of clicking inherent in present designs. In fact, indication can often replace clicking, and clicking can be used, as when you follow a link in a browser by means of a single click, in place of double clicking. For example, say that a user wants to remove inactive windows from a display. Each window has a Close button. Both the Windows and the Macintosh operating systems require that the user first click on a window to make it active and only then click on the Close button to banish the window. This extra click—the one that activates a window just so that the user can close it—is especially annoying. If, however, merely moving the cursor over a window made it active, a single click on the Close button would close it. Of course, if you design a system that exhibits activation in only certain places and under only certain conditions, you will create a modal inconsistency that will bedevil users. Activation should be systemic. As it becomes more familiar, consumer demand will force its adoption.

**Selecting** is a process by which a user identifies a set of one or more objects as having a special status that can be recognized by the system, thereby creating a **selection**. Usually, a user creates a selection with the intention of applying a command to it in the near future. Unlike indication's more transient highlight, the highlight that signals selection persists even after the user has moved the cursor away from the selection. The user creates a single-object selection by clicking the GID button while the object is indicated. A user can also create a selection by dragging a rectangle or other shape across a set of contiguous objects: All objects that intersect the area of the shape become selected. Another convenient method of selection is to create a polygon or free-form shape; all objects completely inside the shape are selected when the user closes the boundary of the shape. When a selection is made, the previous selection should become the **old selection**. (In most present systems, the old selection is simply deselected.) This process can be iterated so that a user can create, in addition to the first old selection, a second old selection, a third old selection, and so on up to an *n*th old selection. A mathematician would be tempted to call the current selection the zeroth old selection. The highlighting that signals selection should be distinct from and more readily apparent than that used for indication; high-

lighting for older selections should also make them clearly distinguishable from one another and probably of lower visual contrast than newer ones. An alphanumeric designation may have to accompany old selections so that they can be readily identified.

Selection can be of discrete objects or of geometrical regions of the display, or they can be **composite**, consisting of the union of selections. In much of today's software, the user creates composite—possibly discontiguous—selections from a set of smaller selections by first making an initial selection. Then, with one common method, she presses and holds Shift and, while in this quasimode, clicks on additional objects to toggle them into or out of the selection.

But this method has three drawbacks. First, the command for creating composite selections is invisible. Second, it is easy to make errors when setting up a large composite selection; for example, if the user accidentally releases the Shift key in the middle of the process and clicks on another object, the work in making the complex selection up to that point is lost. Third, the mechanism is a toggle: The same gesture deselects an object if that object was selected and selects an object if that object was not selected.

The first problem—lack of visibility—is easily corrected, by use of an on-screen hint, for example. The second drawback is the high risk the user runs of making a mistake during the composition process. A more comfortable method for making complex selections includes a command that redefines the current selection to be the union of the old selection and the current selection. Given such a command, the user can concentrate on making a selection without any concern for what she has done previously; only after she confirms that the current selection is correct does she add it to the composite selection. Making old selections available and, of course, marked by a special highlight so that they are visible also allows multiargument commands, such as the two arguments to a command that interchanges a pair of selections. Compare your present method of interchanging two pieces of text with the following technique: Make two selections, then apply the interchange command.

Most present systems do not apply their Undo and Redo commands to the process of making selections. This omission is unfortunate because errors in making selections are frequent. An essential feature of any humane interface is a universally applicable pair of Undo and Redo commands. Only the amount of available storage should limit the number or levels of undos permitted. Undo and Redo should be pervasive, applying to any operation where undoing and redoing is logically possible. They should also be—again, as far as is logically possible—inverse operations. Undo followed by

Redo and Redo followed by Undo should cause no change to the content. Obviously, the commands should not apply to themselves. *The undo and redo operators are fundamental and are of sufficient importance to deserve their own dedicated key in future systems.* Redo should be *Shift*↓ *Undo*↓↑↑, with the key cap clearly marked with the words *Undo* and *Redo* (Figure 5.1). This key would be a good replacement for the problematic Caps Lock key.

Regarding the third drawback, I described the trouble with toggles in Section 3-2, in which I suggested that toggle should not appear in a humane interface. A simple solution is to use one command or quasimode to add an object to a selection and a different command or quasimode to remove an object from a selection. Trying to add an object already selected will not change the selection, and trying to remove an object not in the selection also will not change the selection.

An interface usually has one point at which its designers have interaction take place: the **focus**. For example, if you are a touch typist and your typing appears on the display, the place where the typing appears is the focus and, often, is colocated with your locus of attention. If you are not a touch typist, your locus of attention will alternate between the keyboard and the display. In interfaces that have a cursor, there is, at any moment, typically only one cursor. Its position is controlled by a GID, cursor-control keys, or commands, such as Find.

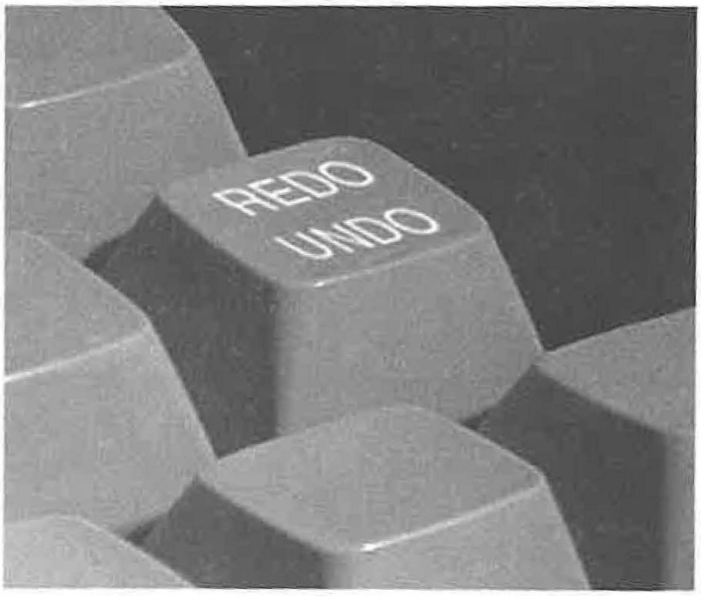



*Figure 5.1. An Undo/Redo key.*

Just as the locus of attention is always an object—physical, mental, or displayed—the same can be said for the system's focus. For example, in current word processors, it might seem that the cursor, when it has been clicked into a document—a click that should be unnecessary—is positioned between two letters, and thus, there is no object to be the focus. In fact, the focus is a pair of characters: the one to the left, which will be deleted if the next command is Delete, and the one to the right, where the next character inserted will appear.

When the human is leading the interaction, the focus will usually be the current selection. When the system is responding to a human or external action, the focus will usually be the result of the action.

### 5-2-2 Commands

*Me, I have a science fiction writer's conviction that the damn robot is supposed to speak human, not the other way around.*

*—Spider Robinson*

Some commands, such as Undo, are keyboard commands that are not necessarily related to selections. Other commands act only with respect to the current selection, such as the command that deletes the current selection. Certain of these commands are invoked by keystrokes; however, the number of keys on a keyboard or keypad is small relative to the number of possible commands. Each additional modifier key, such as Shift, Alt, Command, Control, or Option, doubles the number of possible key combinations. A full-chord keyboard, of which the computer can recognize any combination of keys, allows an astronomical number of key combinations; for example, software that uses any three-key double-quasimode combination on a 110-key keyboard can signal any one of more than one million commands with a single gesture. However, extensive use of modifier keys, especially in combination, quickly reaches finger-twisting, mind-numbing complexity. In addition, the combinations are rarely memorable or meaningful. (Do you know what *Control-Shift-Option-* tells your computer to do?) Learning arbitrary keyboard combinations is difficult; requiring such learning places an unacceptable burden on the user's memory. In addition, such commands violate the criterion of visibility unless the system displays what their effect will be whenever they can be invoked. Of course, if there are times when one of these gestures cannot be invoked or if the gesture has

different meanings at different times, the system is modal with respect to that gesture, giving rise to problems discussed in Chapter 3.

If you divide the system into applications such that a particular command can be reused but is given different meanings in different applications, you increase the number of commands that a user can invoke for a given number of key combinations, but reuse of commands by applications that assign those commands different meanings causes the user to make mode errors. Varying the meanings of a gesture can also place an unnecessarily heavy burden on a user's memory. This burden is partially relieved by menus, although she still has to remember in which menu the command she seeks is hidden. (She may have to first recall which application contained the desired command, especially if several applications have similar capabilities.) This process of looking through menus is sometimes trivial, but it can be frustrating, especially if the command the user seeks is a few submenus deep and if what was to the designer the obvious way to organize the menus is not obvious to the user.

What is needed for invoking commands is a method that is as fast and physically simple to use as typing a few keystrokes and that also makes the commands easier and faster to find than does a menu system. We do not want to duplicate the dual method used in most popular GUIs, which includes both a slow menu-based system and an inscrutable set of keyboard shortcuts. For example, there is nothing memorable about using

*Command* ↓ *v*↓ ↑↑

for insert, aside from the *v* key's being adjacent to the slightly mnemonic *c* key used in

*Command* ↓ *c*↓ ↑↑

for "cut" or "copy."

An alternative approach solves many of these problems. Assume, for the moment, that the keyboard has a key labeled Calculate. When this Calculate key is tapped, the current selection is treated as an arithmetic expression and evaluated. In the following discussion, I will use underlining to indicate selection. Suppose that your text was

```
I want to buy 3 + 4 shirts
```

A tap of the Calculate key would yield

```
I want to buy 7 shirts
```

Before using the Calculate key, the *3 + 4* was ordinary text. Except that it was selected, there was still nothing special about it; the five characters, including spaces, of which the selection was composed could have been deleted or moved, or another typical word-processing command could have been applied to it. But in this case, the operation *calculate* was applied. The user did not need to open a calculator window or to invoke a calculator application.

Now consider that there was no dedicated Calculate key on the keyboard. (Although evaluation of mathematical expressions is certainly a candidate for a dedicated key, such a key would certainly be more valuable than many of the keys we do have, such as F9.) What we need is a more general mechanism for commands.

Before discussing such a mechanism, consider the requirements for an improved method of invoking commands. They might include that

- It not be modal
- It accommodate any number of commands—in particular, that it not be limited by the size of the keyboard
- You be able to invoke a command without taking your hands from the keyboard
- You be able to invoke a command with the graphical input device
- It not require a plethora of special keys
- The system not end up with too many quasimodes

One quite general method can best be introduced by means of an example. (This somewhat trivial arithmetic example is chosen to demonstrate the method; more efficient means of applying the method will be discussed later.) Assume that the text was

```
I want to buy 3 + 4 shirts calculate
```

Select the sum `3 + 4`, and then select the word *calculate,* which makes the sum the old selection.

In an alternative method, one that gives the power of a command line interface to the user, a tap of the Command key can be used to invoke the selected command. If the command requires an argument, the old selection is used. In this method, the command itself is deleted, and the result of the evaluation is left selected.

```
I want to buy 7 shirts
```

The idea is that commands need not be restricted to supplied menus but can be part of your text, or, where appropriate, the command can be a graphical object instead of a word or a set of words. It is also important that commands can be provided by the user in the simplest possible way, by merely typing or drawing them anywhere. This use does not conflict with the method in which the command is selected from preexisting text.

Menus offer the advantage of making a list of commands visible. However, instead of selecting a command from a menu, the user could as easily select the command from a small document listing commands. It would not matter whether the document was provided for the user or whether she had typed it herself. Further, the document need not be limited to a bald list of commands. For example, each command could be accompanied by a description or even the user's own notes. A document that is used as a menu is an ordinary text document, not something that can be changed only by programmers or by using a special customization facility.

This approach has a number of advantages. For example, an online manual automatically contains functioning examples of the commands it is describing. In present systems, a menu command might or might not have a keyboard method. But in this approach, every command described by a sequence of characters in a menu has a keyboard equivalent. This is guaranteed not by the diligence of the designers but by the very nature of the system. If you find a command in a menu, that same spelling is the keyboard equivalent. It is the keyboard equivalent to which most users will habituate. Another advantage is that you can make up a menu comprising only those commands you use, by merely typing a list of the commands in your word processor. Of course, if you continually adjust the arrangement of a list as opposed to, for instance, adding to it, you lose the advantages of habituating to their location.

Just as web links in a text are often distinguished by being visually distinct—for example, they are often shown in color, typically blue, and are sometimes underlined—commands could be shown with some other special marking—red, with reverse italics, for instance. With such a distinction, the user would point to a command name so that a letter in the command is indicated, whereupon a tap of the Command key would invoke the command. This eliminates the step of having to select the command name in invoking a command.

If there were no special font or color for commands, it would be necessary to have another convention to show that a word or sequence of words was to be considered as a single command. It is wise to avoid some of the present-day conventions that are used to group separate words into a single entity delimited by spaces or other characters. For example, if there were a

command that we wanted to call *change to JPEG bitmap,* present conventions would have us notate it as change.to.JPEG.bitmap, <change to JPEG bitmap>, or change_to_JPEG_bitmap. These notations are "computery," ugly, and discouraging, especially to newcomers to computers. *The syntax we choose for commands should not keep us from putting spaces or returns in them.* Any restriction in the character set we can use to name a command is likely to bite at a future time; moreover, any such restrictions have to be remembered when naming a command. Another principle to keep in mind is that *use of conventions not in accord with our natural-language conventions helps make computers feel unfriendly. We must bend the machine to work the way we do rather than change our language conventions to suit what is easiest to program.*

Another interaction of typing and selection causes problems in current interfaces. In a humane interface, typing does not replace selected text or cause the current selection to become unselected. This is the opposite of the common convention that typing replaces the current selection, a practice that occasionally causes users considerable grief when the new material unexpectedly overlays text they did not want to delete. The idea that typing should replace a selection was introduced to save a single keystroke: In most editors, when you wish to replace a block of text, you simply select some text and then type. Without this convention, you would select some text, tap Backspace or Delete, and then enter some text. The only keystroke the current convention saves is the backspace. With the usual convention, the text vanishes at your first keystroke, and your typing is inserted. This happens whether the text to be replaced was on screen or not and (usually) whether it was a few characters or three quarters of the novel you were writing; you may be deleting text that is 40 pages away from your locus of attention. To be sure, if you notice the problem in time, you can possibly undo it. However, if you do not notice the deletion—and there is nothing that lets you know that text has been deleted—you may be out of luck. A humane interface never puts your work at risk; the one saved keystroke in this case is bought at too great a price; if you lose even one character inadvertently, that character might be part of a phone number or part of an e-mail address that is unguessable from the remaining text. *The interface should require you to explicitly delete text if you want to delete text and not delete it as a side effect of another action.*

The concept of locus of attention is useful in defining what, exactly, we mean by a side effect: A **side effect** is an effect of a command that alters contents or events that are not your locus of attention. In the case just discussed, your locus of attention is the text being inserted, and the side effect is a deletion. *The elimination of side effects should be one of the goals of any designer of humane interfaces.*

Another word processor feature often considered helpful is the ability to drag a selection from one place to another in text. However, this prevents you from being able to create a new selection overlapping the current selection or to create a selection that is a subselection of the current selection. If you try to create either of those selections, the system assumes that you are trying to move the selection. This means that you have to click somewhere outside the selection to deselect the selected text before you can proceed. The dragging gesture has thus been given two different meanings, namely, selection and moving a selection. This can interfere with habit formation. Errors arise because, though the characters in the selection are your locus of attention, the current state of the selection is not your locus of attention, even though it is visually indicated. I have observed users inadvertently dragging a selection when they intended to create a new selection.

Another problem arises from dragging in text and also occurs in graphics applications: You will sometimes start to drag a selection only to discover that the destination is not visible on the display, in which case you have to put the selection back or into another place and change to the cut-and-paste method. The principle of monotony suggests that having only one method is preferable. Some systems begin to scroll when you bring the dragged selection to the top or the bottom of the display, but scrolling is much too slow if the destination is more than a few pages away. Scrolling can also be too fast, making it impossible to stop at, or even see, the desired destination.

If marketing could be kept from screaming too loudly, I would not design an interface with drag-and-drop in text, at least as it is presently implemented on personal computers. Users accustomed to having drag-and-drop in text would, I think, find sufficient compensation for this "loss" in having a system that causes less frustration and fewer errors. It would be better still to provide separate quasimodes for selection and dragging, because you could then have both selection and drag-and-drop without cognitive interference. For example, if the GID had a button that you pressed to make selections and if it also had a facility, such as a side-mounted button, that allowed you to squeeze the GID—with tactile feedback, such as a click, to let you know that the squeeze has been registered—to indicate that you've grabbed a selection, there would be little or no confusion between the two functions. After a few seconds of instruction and one or two trials, you would know how to use it ever thereafter. More pedestrian methods for separating the selection and drag gestures would be to use a different mouse button for dragging or to use a quasimode, such as holding down a special duly marked key while using the main mouse button. (See Appendix A for a more detailed rationale.)

Another use for a grab feature on a GID is as a replacement for scrolling. You can grab anywhere in the document and start it moving up and down: exclusively up and down for narrow documents; in any direction for wider documents. When the grab cursor—cleverly indicated by the image of a hand in some present systems—reaches a border of the display, scrolling continues in the present direction until the grab function is released or the cursor is moved back into the window. The customary method of scrolling via scroll bars is confusing. For example, pressing on the down-pointing scroll arrow makes the content of the screen scroll up; designing the arrows the other way around is only slightly more confusing. In addition, scroll-bar arrows are small features and therefore time consuming to use; being able to grab anywhere on the document is much faster, as a Fitts' law analysis readily shows.

The example of the need for a grab function on a mouse incidentally demonstrates that working on software interface design often generates ideas for improved hardware, just as hardware considerations can inspire improvements in the software design. It is always better to design hardware and software together, although opportunities to do so are rare. Trying to shoehorn a pure software interface into hardware designed for another interface is seldom completely satisfactory. Nonetheless, for most projects, that is what we must do.

### 5-2-3 Display States of Objects

A humane-interface feature must be both accessible to the naive and efficient for the expert, and the transition from one to the other should not demand retraining; a good interface should provide the user with one mental model that works for both classes of user—recalling, of course, that we are beginners or experts for different parts of a system independently. In the previous section, it was proposed that a key that executes text as commands can be used with selected text, whatever its source, to cause a command to be executed, assuming that the selected text is the name of a command; otherwise, there is no effect on the content. It is convenient to also allow the Command key to be held to create a quasimode during which a command can be typed. The convenience is highly dependent on the ergonomics of the Command keys. This last feature is an improvement on command line systems, which are much loved for their speed and convenience of operation and much hated for the difficulty with which they are learned. The improvements are two: You can issue the commands anywhere and at any time, and the commands are identical to those appearing on menus, so that the transition from menu to direct command is trivial.

Because it would be a waste of time and display space to designate a special location for typing commands, a command should be typed wherever the cursor happens to be at the moment the need for the command becomes apparent. The typed command name should be deleted after being executed, so that command names are not left scattered among your content. On the other hand, when you execute a command from a list of commands, you do not want the command to disappear; in effect, the list is a menu. To create such a menu, nothing more is needed than to type a list of commands, select the list, and then use a command, perhaps called Make Menu, that can change them to a distinctive style typically used only for commands and, at the same time, lock the list of items so that it cannot be inadvertently changed.

Here are some other commands that change the state of text. One convenience is to be able to simply lock text or other content, with a **Lock** command. Locked content can be seen, selected, and copied but cannot be changed or moved. The inverse, **Unlock**, can be applied to selected content and will unlock the selection if the selection was locked. (Otherwise, it has no effect; it must not toggle.) Another command, **Lock with Password**, locks the old selection, using the current selection as the password. It, too, has an inverse (**Unlock with Password**). Locked content has a wide range of utility; for example, it can be used to create forms to be filled out. The fixed parts of the form are locked or password locked. Simple locking prevents accidental changes; password locking prevents unauthorized changes. If the online instruction manual to a computer system were included as part of the text that the computer initially came with—which is not a bad idea—the online manual would probably be password locked at the factory.

**Screen Lock** and **Screen Unlock** lock and unlock the position of selected objects with respect to the display. With this facility, you can create menus that stay in place on the display as other objects move underneath them: You could simulate today's fixed-position menus. (Whether this is a good use of this facility is a different question.) To use this command, you would place the object as desired with respect to the screen, select it, and use the Screen Lock command. There could be a password-protected version of this as well, for occasions when it is undesirable for the user to redesign the menus.

Another useful command is one that controls the transparency of a selection. In some situations, such as when displaying error messages, it is useful to render the selection transparent enough so that underlying material can be seen and operated through (Figure 5.2). Similarly, another command

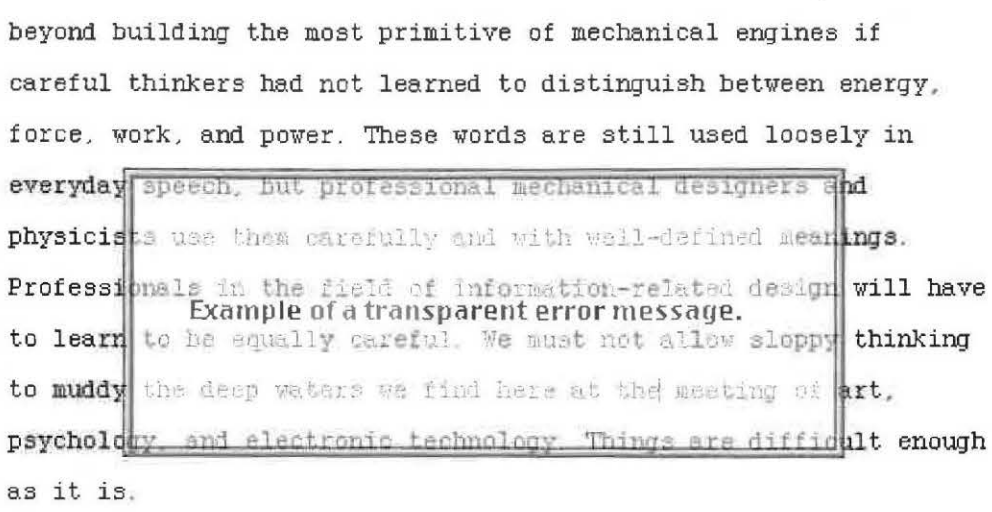



***Figure 5.2. A transparent error message over background text has an efficiency of 1.***

could specify whether an object hides or is hidden by or is viewed through another object. A transparent message box can fade gradually instead of abruptly, giving you time to notice it. A document that stores all messages for later review is essential.

Because you can click through a transparent dialog box, removing it requires no keystrokes; it is modeless and highly efficient, with an efficiency of 1. As with any method, this idea has its limitations and can be overused; a deluge of unnecessary messages is still distracting, even if they disappear when the user proceeds. In accord with the principle of visibility, a visible distinction should be provided to allow a user to determine whether text is locked, screen locked, locked with a password, and so forth.

A humane-interface principle is that the system itself should be built out of the same kind of pieces with which you are familiar from your everyday use of the system. This makes your product less forbidding.

## 5-3 File Names and Structures

*Mankind are more disposed to suffer, while evils are sufferable, than to right themselves by abolishing the forms to which they are accustomed.*

*—Thomas Jefferson, Declaration of Independence of the United States of America*

Because the restriction of file name length to 8 characters in earlier systems was truly dreadful, the 31-character maximum length offered by the Macintosh looked like heaven to many users. But this modification was just a different-size straitjacket. Aside from the real limits of the hardware, an interface should have few, if any, fixed-length limits. An interface should use

dynamic memory allocation, linked lists, hashing, or whatever techniques seem best but should never present a user with software implementation limitations, such as "you can have up to 255 categories" or "paragraphs can be up to 32,000 characters."[2]

What is a file name? From the user's perspective, it is a handle with which to grab a file. As we know from long experience, file names do not work as one would expect; they are an impediment when you want to save, futile when you want to find. Let me be more specific: File names are bothersome when you are about to save work, because you have to stop in the middle of your activity, which is trying to store your work away, and invent a file name.[3] Creating names is an onerous task: You are required to invent, on the spot and in a few moments, a name that is unique, memorable, and within the naming conventions of the system you are using. Furthermore, at that moment, the question of a file name is not your locus of attention; preserving your work is. File names are also a nuisance when you have to retrieve a file. The name you thought up was probably not particularly memorable, and you probably forgot it after a few weeks (or less). I, for one, can rarely remember a file name unless it is quite recent; even looking through lists of file names is frustrating. Just what is in that file that is labeled "notes ybn 32"? The name seemed so clever and memorable when I created it. Then, too, many files are nearly the same. How many different, creative, readily remembered names can you think up for letters to your accountant about last year's taxes? Filing them by date may be useful, but how many of us remember that the letter about the deduction for the company truck was written on August 14?

Having to name files increases the mental burden on the user. Giving a file a name does nothing more than add a few characters to the file, and you are required to remember the file by that one tiny portion and by nothing else. I count this as one of the major horrors visited upon us by conventional computer systems. Many information appliances have also adopted this inhumane methodology.

There should be no distinction between a file name and a file. A human mind can more effectively use a fast, whole-text search engine, so

---

2. A word processor I once used quite a while ago had this paragraph size limit. I exceeded it as soon as I inserted a photograph into a paragraph; when I spoke to the designers, they admitted they had never thought of that. The moral: Never put in a fixed limit because it makes the software easier to write; it will always be too small.

3. This is in the context of present systems. You should never have to explicitly perform a save in future, humane systems.

that any word or phrase from the file can serve as a key to it. (Eventually, we'd want more: A request for "a letter about dragonflies" would cause a search that looked for something that had the form of a letter and looked not only for the word *dragonfly* but also for related terms or expressions, such as *Odonata*—in case dragonflies had been referred to by their scientific name—and if no instances of such letters were found, the search would look for nonletter documents with that content, and so forth, extending out to networked computers and to the Internet.) You do not remember the content of "Letter 12/21/92 to Jim" when you see that title, but you do remember that you once wrote to Jim about the blue Edsel that ran across your eyeglasses. A search on *Edsel* is likely to find only one or two entries on your whole system—unless you are an Edsel fancier, in which case you would probably choose another pattern on which to search. An unlimited-length file name is a file. *The content of a text file is its own best name.*

Graphics and sound files often require names; Section 6-2 discusses an approach that avoids the memory burden that traditional file structures impose for nontext files. Aside from nontext files, given a fast whole-text search, file names—one whole species of unnecessary entities—can be eliminated. With the removal of file names go all the mechanisms for dealing with file names, such as directories of file names, rules for editing file names, and syntactic restrictions on file names. When file names are eliminated, a significant mental burden and much internal machinery—machinery that is currently part of what you have to learn and what programmers have to implement—vanishes.

The best interface to a whole-text search is interactive, where you see each found instance in context as it is found. In this interface, as soon as you see what you want, you are there. Some systems present copies of the found instances in the line that contains them (Drori 1998). However, this method is not as efficient as the first kind of search, because you must then perform a second operation to reach the instance itself; for example, you must click on the copy of the desired instance.

For users who insist on a system that looks like conventional file structures, there can be a command that creates an "information document" or an extra page at the end of each document when the document is selected and the command applied. The information document or page would contain such information as the date and time the document was created or modified, a revision history, the length of the first document, or whatever information seems of value. The software to implement such a command would have to acquire and store the necessary information invisibly to the user. Various vendors might provide different facilities, depending

on user needs. For users who wish to hold on to their old ways, a vendor could even go so far as to create utilities that create documents that look and act just like the annoying directories we now have.

Another source of organization, one that is easier to learn and to use than traditional file systems, comes from the inherent hierarchical structure of many natural languages: Words are separated by spaces. Sentences, or sequences of words, are separated by one of a small number of delimiters, followed by a space. (In English, these delimiters include periods, question marks, and exclamation points.) Paragraphs, or sequences of sentences, are separated by at least one explicit Return. Explicit page characters, or page breaks, separate chapters or whatever you wish to call the next level of organization.

In a consistent system, page breaks should be characters and, unlike most present systems, they should behave—in terms of insertion, deletion, and searching—just as any other characters do. As with Return, there may be implicit page breaks to accommodate the fixed length of physical pages, but these are not part of the content.[4]

There is good reason not to stop here in the hierarchy as many present systems do. Documents are sequences of pages separated by document characters, each typable, searchable, and deletable, just as is any other character. There can be higher delimiters, such as folder and volume characters, even section and library delimiters, but the number of levels depends on the size of the data. A set of two consecutive document characters makes a fine delimiter for sets of documents. If more levels of organization are needed, three or four consecutive document characters could be used as delimiters. It can be easier to repeatedly tap the Document key, even four times, than to type such rarely used keys as Folder, Volume, and Library. This convention also prevents an explosion of new keys on the keyboard. It is important for all delimiter characters to have dedicated keys, for otherwise, they would not behave as do all other typable characters. That is, we should not use a keystroke for, say, Return, and then use an Insert Page Break command; we must have a page character.

Because the various separator characters would behave exactly as did all other characters, there would be no need to teach users how to do searches for them. An individual who insists on having explicit document names can adopt the personal convention of placing the desired names

4. Dr. James Winter, at Information Appliance, further unified the structure by pointing out that English already has the same kind of hierarchical, character-delimited structure proposed for the higher levels of organization.

immediately following document characters. To find a document with the name "Dogs of Asia," you would search for a string that began with a document character followed by *Dogs of Asia*. Such a search would ignore all instances of *Dogs of Asia* except those used as document names. If you wanted a directory, there could be a command that would assemble a document comprising all instances of strings consisting of a document character, followed by any other characters, up to and including the first Return or higher delimiter.

Eliminating hierarchical file structures does not mean that you have to give up structuring your stored information. Nothing prevents you from creating tables of contents and indexes or putting all of your letters to Uncle Albert and Aunt Agatha on consecutive pages. Nothing prevents you from even putting a heading page (just another document) labeled "Letters to Uncle Albert and Aunt Agatha" in front of them. If that is done, you have created, in effect, a file name but without having added a special mechanism in the software. You can create, if you wish, a hierarchical file structure of your own so that, if you really love file names and hierarchies, you can have them. Structure, as you choose to create it, is part of your *content,* not part of the *interface.*[5] Instead of a file-finding facility, you simply use the general search mechanism to find the file names you have squirreled away. You can place a folder name as a document in front of a number of collected files; you can place a volume name in front of a bunch of collected folders. (Your pattern would be a volume character, followed by the name of the volume; such a pattern would eliminate matches to other occurrences of the name that happen not to be labels of volumes.) The absence of a built-in file organization does not prevent you from creating a file that fits your needs and one that, because *you* created it, you understand. But also, because the system has not been changed in any way, a guest can find things in your structure and, in fact, can ignore your structure and treat it as a flat, unstructured, file.

One advantage of filing information as you wish is that the structures were not dictated by the system designers, who may have ideas different from yours. Therefore, you do not have to develop a mental model of what the designers were trying to do. Many users do develop inaccurate models of how systems work; these mental models persist and cause continuing difficulties for those users (Norman 1988).

This discussion is not theoretical: On the SwyftWare and Canon Cat products, the elimination of file names, directories, and the various mechanisms usually provided for manipulating them proved one of their most

5. The same is true of file names and menus proposed in this book.

successful features. Users experienced at conventional computer systems sometimes found it difficult to make the transition to content-based organization, but once they had made the transition, the conventional methods soon began to seem cumbersome. Users who started on the Cat were not amused at having to learn the more complex and difficult methods of conventional file systems when they moved to a PC or a Macintosh.

To users accustomed to standard GUI practices, the methods outlined here may seem complex by comparison. But the apparent complexity is due to the new paradigm's lack of familiarity and to our having become habituated to the many steps we must take and to the problems we have with the current system. The learning curve has been long surmounted. But when you compare the progress of beginners with the two systems or compare the performance of experienced users, the advantages of the simpler system become apparent.

Consider that you have *n* documents that you want to copy to an external medium, such as a hard drive. With, for example, the Macintosh operating system (OS), you drag the icon of each document to the icon of the drive, and it is copied. In the new paradigm, it seems at first more complicated: You have to find the beginning and the end of each document, select the document, move the cursor to a place on the drive, and then move each document.

Recall that in the GUI, you start in the generating application. Your first step is to get to the desktop. You must also know which icons correspond to the desired documents, and you or someone else had to have gone through the steps of naming those documents. You also will have to know in which folder they are stored. So the apparent simplicity is arrived at only after considerable work has been done and the user has shouldered a number of mental burdens. A more efficient method involves an interface invention called LEAP. Assume, as we did for the GUI, that the cursor is in one of the documents you wish to move. With LEAP, the user can select the document with six keystrokes, and without having to look at the screen or recall the document's name. To type six keystrokes takes less time than to drag an icon.

LEAP works like this: There are two LEAP keys which lie under the thumbs. LEAP-Up searches forward and LEAP-Down searches backwards from the cursor position. Pressing and holding a LEAP key puts you into a quasimode during which whatever you type is used as a search pattern. To select the document you'd LEAP-Up to a document character (LEAP-Up↓ Doc↓↑↑). This puts the cursor at the beginning of the document. Then you'd LEAP-Down to a document character (the search will find the document character at the end of the document). A tap of both LEAP keys

together selects the text. (This is probably most easily done when the LEAP keys are operated by the thumbs, which are otherwise underutilized in typing. See Figure 2.1 for a typical keyboard designed for this use of LEAP. A dedicated Select key is another alternative.) In order that the function be visible, a legend adjacent to the LEAP keys is needed. The legend could, for example, read "Press both LEAP keys together to make a selection." Note that you do not have to watch the display while you select the document. Once a document is selected, the cursor is LEAPed to wherever you want to place the document. When the drive was plugged into the computer, its contents became part of the contents of the system, so no special mechanism is required to find it. A Copy command completes the operation. When selected this way, the document includes its separators. Thus, if the document is moved, it retains its character as a document because the document separators move with it.

The same technique used to copy a document—or a selection of any length from a character to a set of documents or the entire contents of the system!—from here to there is used to move a selection; the only difference is that a Move command rather than a Copy command is given. The process is functionally no more complex than that needed in the GUI. It is often faster, and the number of methods, concepts, and structures that an individual must understand is lower.

Consider how you would use the LEAP-based paradigm to put a few selections from different documents together onto a drive, and then consider how you would perform the same task using a GUI. With LEAP, the method is the same as that just described for moving documents to the drive: The selections are *found*—and once found, do not have to be opened, because the concept of opening a document is superfluous—*selected,* as described previously, except using the text rather than document characters at the beginning and end of the selection; and *copied* into place. In a GUI, the user must first *open* a new destination document, possibly by using the New command in the File menu of the application; *find* a document that contains a selection the user needs; *open* the document; *find* the selection within the document; *select* it; *use* the Copy command; *activate* the destination document; *paste* the selection in; *activate* the desktop; *find* the next document that has a desired selection; and *repeat* until all of the selections have been pasted into the destination document. Then you must *save* the result to the drive by using a dialog box.

Even if the complexity of doing any task was the same in either paradigm, the conceptual simplicity of the methods outlined here would be preferable. In most cases, the work required is also far less.

## 5-4 String Searches and Find Mechanisms

*One giant leap for mankind.*

*—Neil Armstrong (1969)*

Before going further into LEAP, it is useful to treat the subject of the interface to searches with somewhat more precision. A **string** is a sequence[6] of characters; ordinary English words and sentences are examples of strings. **String searches** look through a (usually lengthy) string, called the **text**, for an instance of a (usually brief) string that the user specifies, called the **pattern**. Each occurrence of a substring of the text that matches the pattern is called a **target**. For example, if you were trying to find where you had written of a cat called "little Tatsu" in a long letter, *little Tatsu* is a good choice of target, and you might choose the briefer string *Tatsu* as the pattern to use in the search. The match may be exact, may be case-independent, or may represent another relationship between the pattern and the target; they might rhyme, for example. A commonly used matching criterion, one that tests well, is that lowercase letters in the pattern match *either* uppercase or lowercase letters in the text

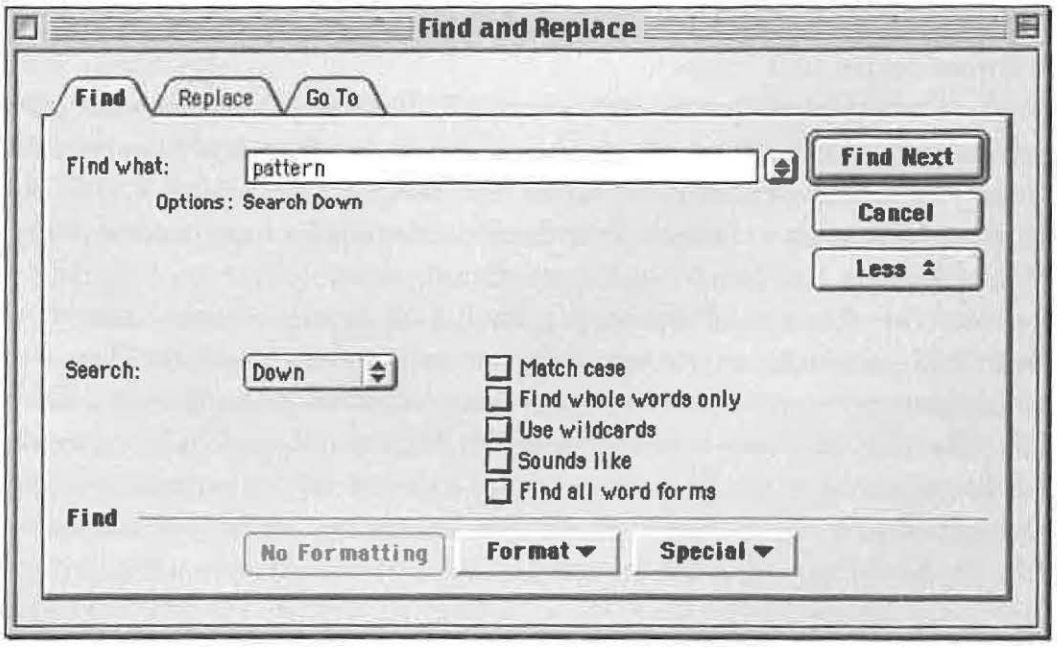



***Figure 5.3. A modal-search dialog box with modal-search types and direction settings.***

6. I use the term **sequence** as mathematicians do, to mean a set of objects having a first object, a second, and so forth.

but that uppercase characters in the pattern match only uppercase characters in the text. Searches usually start from the current cursor location and continue forward through the text. In most systems, a modal user preference setting can direct the search to proceed backward through the text (Figure 5.3).

Interfaces to searches are typically based on either of two search interface strategies. The most common strategy is the **delimited search**, found in most word processors. In a typical delimited search, the user enters a mode in which typing or other method of text input is regarded not as text but as a pattern. This is usually accomplished by using a dialog box containing a field into which the user can enter characters. After summoning the dialog box, the user types the pattern, followed by a delimiter, which is usually a character, such as Return, that is not permitted to occur in the pattern. In most dialog boxes, the user may also limit the pattern by using the GID to click a button with a label such as OK, Search, Find, or Find Next. When it is found in the text, a target is selected, and the cursor is placed immediately at the end of the selection.

This traditional method is rather punishing to the user, although most computer aficionados have become so accustomed to it that they no longer feel the pain. For example, an individual can type in a search string, make a typo—unnoticed until too late, of course, because he pressed Return by habit—and sit there, waiting for a search that he knows will fail. Most searches are not interruptible: a serious design error. Because the computer waits until the user has finished the pattern before beginning its search, delimited searches often keep him waiting unnecessarily.

The less common strategy is the **incremental search**, a popular example of which is found in EMACS, an editor used with the UNIX operating system (Stallman 1993). In most implementations of incremental searches, as with the delimited search, the user first summons a dialog box that contains a field in which the user can enter the pattern. When he types the first character of the pattern, however, the system uses this character alone as a complete pattern and immediately begins to search for the first instance of that character in the chosen search direction. If an instance of that first character is found before the next character of the pattern is typed, the instance is selected and the cursor placed immediately after the end of the selection. If the next character of the pattern is typed before an instance is found, that character is added to the pattern and the search continues, now looking for an instance of the now extended pattern. This method is repeated as each character is added to the pattern.

Using the Backspace or Delete key to delete a character from an incremental search pattern should return the search to the previously found instance that matched the pattern, as it was before the deleted character had

been added to the pattern. The user can then add to the pattern, and the search continues without his losing the search already accomplished on the partial pattern. Many implementations do not have this desirable characteristic.

Incremental searching has a number of other advantages over delimited searching. Incremental searching wastes less of a user's time: The search begins when the first character of the pattern is typed; the system does not wait until the pattern is complete. With a delimited search, the computer waits for the user to type the pattern and delimit it, after which it is the user who waits while the computer does the search. When using a delimited search, the user must guess, beforehand, at how much of a pattern the computer needs to distinguish the desired target from other, similar targets; with an incremental search, he can tell when he has typed enough to disambiguate the desired instance, because the target has appeared on the display. Thus, as soon as he sees that he has reached the point desired, he can stop entering the pattern. If he types too many letters of the pattern—if the hand is quicker than the search—the pattern will still match, and the cursor will stay approximately where he wanted it. If he mistypes a pattern in a delimited search, he must wait until the search for the incorrect pattern is complete—or, at best, he can operate a mechanism for stopping the search—before he can correct his error. In a large text, the search can take a considerable period of time. In a well-implemented incremental search, the user can backspace at any time and be returned to the last match found. Because backspacing after an error is habitual, the process of correcting an error is very fast, and the search stops immediately. He can also resume the search by typing the correct character.

Another advantage of incremental search is that, as the user enters the pattern, it provides constant feedback as to the results of the search. With a delimited search, he does not find out whether his pattern is suitable or even if it was typed correctly until it is fully entered and a search attempt has been made. From the point of view of interface engineering, the advantages of incremental searching are so numerous and the advantages of delimited searches so few that I can see almost no occasions when a delimited search would be preferred.[7] In spite of near universal agreement about the desirability of incremental searches on the part of both designers and users, almost all interface-building tools make it easy to implement delimited searches and difficult or impossible to implement incremental searches. JavaScript and Visual BASIC are two examples of such.

Building a pattern incrementally allows the user to adjust the pattern interactively during the search, which leads the user to improve his search

7. A search is either incremental or excremental.

strategies through the feedback received. Even building a Bóolean search pattern is made more effective when the early results of the pattern appear as the user adds to the pattern's specificity. The found instance should appear in the middle of a display area, not at the top or the bottom. Thus, material both before and after the instance is visible; that is, the found instance is displayed in context. The instance of the pattern found should always appear in the same place relative to the display or window so that the user soon learns where to look for the result of the search. In the Canon Cat, it always appeared at the vertical center of the display. It should not appear adjacent to any edge of the display area, so that material on all sides of the found instance is shown.

If an instance of the pattern does not occur in the text, the search fails. Many systems cease operating in this event and remain unusable until a particular key—typically Enter or Return—is pressed or a particular on-screen button is clicked. A modal message is placed on the display to let you know that you must make the required obeisance before you will be allowed to continue using the computer. In multidisplay systems or if the screen is visually busy, this message may be nowhere near your locus of attention. You may not notice it at all. It then seems to you that the computer will not respond to the keyboard; it seems to have crashed. In an incremental search, it is clear, without any special indication, that a search has failed; the cursor returns to its original location, and additional keystrokes do not have any effect. A short beep or a screen flash can also be helpful, especially if the search exceeded the duration of short-term memory, say, 10 seconds, so that the user has forgotten the appearance of the display prior to the search. The beep is also a useful cue for visually impaired users.

### 5-4-1 Search-Pattern Delimiters

Another major deficiency of delimited searches is that the delimiter used to end the pattern cannot be typed into the pattern. Often, other delimiters are excluded as well. I looked at four popular word processors, one did not allow a Return in a search pattern at all. In the second word processor, the user types ^r to insert a return into the search pattern. In the third, \\ is used; and in the fourth, a Return is put into a search pattern by the use of a special dialog box with a pull-down menu of delimiters (Figure 5.4). It is easier to simply press the Return key when creating a pattern. After all, that's how you enter it into text; why should it be different in entering a pattern? The general principle is that *the same sequence of characters should always be typed the same way; you should not have to use one method here and*

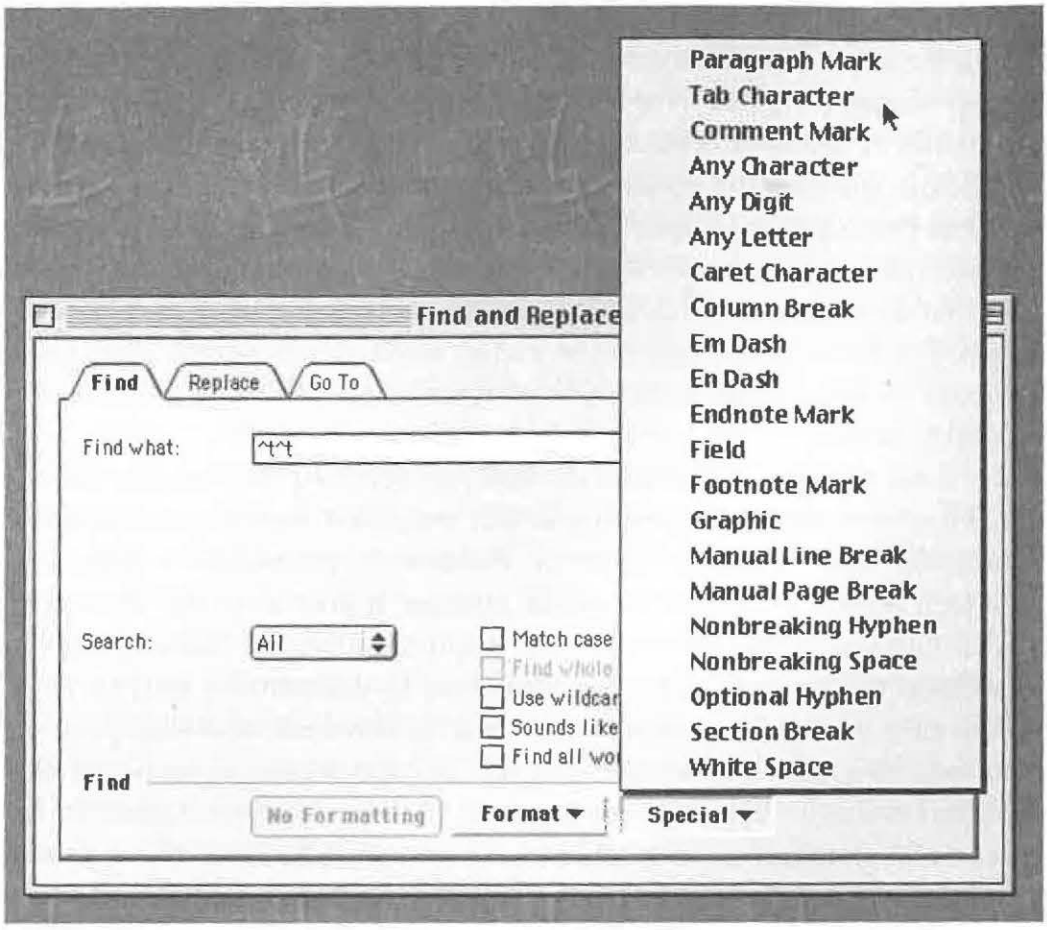



***Figure 5.4. The Word search box opened to show the list of characters that can be inserted. With a better search design, the user could put, say, a tab in a search string just by tapping the Tab key. Note the two codes for tab characters inserted into the pattern (the Find What box).***

*another method there*. Another way of saying this is that there should be nothing special about special characters.[8]

Although incremental searches are better than delimited searches, further interface improvement over the EMACS style of incremental search is possible. For example, the search should leave the cursor on the first character of the target rather than on the last. In general, you have little control over what will be the last character in your pattern, because you type just enough pattern to find your target. Thus, you do not know exactly where the cursor will be when you are finished. If the cursor lands on the first character of the pattern, you can predict how the target will appear. This also

8. A parallel problem is the use of reserved words in programming languages.

means that the search mechanism can also be used to quickly position the cursor in text, because your locus of attention is the character on which you want the cursor to be placed. Trying to invent a pattern where this is the last character in the pattern is far more difficult than typing the desired character and whatever follows it on the display.

In conventional GUIs, both delimited and incremental searches are initiated modally, by means of a dialog box. LEAP is modeless. The concept of using a quasimode for searches can be extended to a button on a microphone (or GID) that is held to create a quasimode during which words, sketches, or handwritten letters are used to create a search pattern. Other input techniques have analogous means for creating a search quasimode (Raskin and Winter 1991).

The speed of incremental searches is increased when a few implementation tricks are used; for example, when you type the first character of the search string, the computer instantly proceeds to find the first instance of that character in the text, which is highlighted and, if it is not there already, moved with its context into the display window. This is usually a fast process because there are a lot of potential matches, and one is likely to be close in text. While waiting for the user to type the next character, a search can be started for the next instance of the first letter, followed by each of the next possible characters, working in decreasing frequency-of-use order to maximize the probability of an early hit. The program can save pointers to each as they are found. When the second character is typed, the computer will often be ready to display the found target, seemingly instantly.

String searches can also be speeded by such methods as the Boyer-Moore algorithm for fast string search (Moore and Boyer 1977), whereby search time decreases as search-string length increases. In case the user backspaces through the pattern, keeping a pointer to the last-found location—one for each character in the search string—can make going back blindingly fast. Indexing all local mass-storage devices can make searching over a local system or network take only milliseconds. Interaction at high speeds, with wide area networks or the web, also depends on indexing methods. Incremental searching, in various versions, has been available in such commercial products as Borland's IDE, Global Village's fax program, the Canon Cat, and SwyftWare.

### 5-4-2 Units of Interaction

Incremental searches are just one example of a broader humane-interface design principle: *A program should interact with you on the basis of the*

*smallest meaningful unit of input.* Interaction with input from a keyboard should be on a character-by-character basis, not on a line-by-line basis. Voice input should interact on a word-by-word basis; for some applications, interaction could be on a morpheme-by-morpheme basis, and so on.

Interaction by means of entering an entire line of text, that is, text delimited by a Return, at a time is a holdover from the days of the Teletype and should be relegated to a museum alongside the hardware of that time.[9] Today, we can and should have interfaces able to react to each character as you enter it, whenever such a reaction could improve the quality of the interaction. As always, designers must be considerate: Character-at-a-time interaction should not be used to deliver spelling-error messages in the middle of typing a word, correctly judged a nuisance by touch typists.

In the small texts for which string searches were originally designed, searching usually proceeded from the current cursor position to the end of the text. In larger texts, it is generally a good idea to continue the search, if it has not found a target by the time it has reached the end, circularly from the beginning back to the cursor location, in case the user has forgotten that the desired target is farther back in the text. Unpublished testing at Information Appliance demonstrated that, if the search is fast, the circular search is especially favored by users. Fast means here that there is not enough time for a user action between the time the search is launched and the time it succeeds or fails: as usual, about 250 msec. In many systems, the user can also choose to have the search stop at the end of the file or search circularly. This causes a typical mode problem: If the search is set to noncircular and the user is not aware of the setting, a "string not found" message can give the misimpression that there is no instance of the pattern in the text. Or, what I have seen happen on many occasions, the user repeats the search a number of times because he distinctly recalls that there was an instance of the pattern in the text, and he is puzzled and frustrated by the search's repeated failure. It can be many seconds or even minutes before he either figures out the problem or gives up in frustration. If it is necessary to have different kinds of searches, this can be accomplished modelessly by using a different command or on-screen button to launch each kind of search.

In many circumstances, modality can be avoided by having a set of launch buttons, one for each variety of an operation, instead of having means for setting up the desired variety and providing only one launch but-

9. There is an ever increasing need for a museum of interaction (MOI), as we have museums for hardware and software. The Internet might be an appropriate home.

ton to initiate the operation under conditions you have established. This improvement not only eliminates a mode but also saves keystrokes. Also, the user's locus of attention is the task that he wants done, not the setting up for the task. Figure 5.5, a typical set-up-then-launch dialog box found in Microsoft Word, exhibits a second interface problem: Should the radio buttons be always in the state shown when the user opens the box? Should they be in the state that he last left them, or should there be a preference setting as to which is the default?

All three interface options are wrong. If the user is always updating the entire table, the first option forces two clicks—or a click and a Return—every time. If the box is left in the last-used position, the user cannot use the box habitually, because he has to stop and check on its state each time. If there is a user preference (see Section 3-2-2), then a mode has been set up.

The dialog box of Figure 5.6 solves all of these problems at once, and the larger buttons have a Fitts' law advantage over the radio buttons. The Cancel button could be eliminated if this box were transparent, as discussed in Section 5-2-3. The two buttons would not be transparent, showing that they are active.

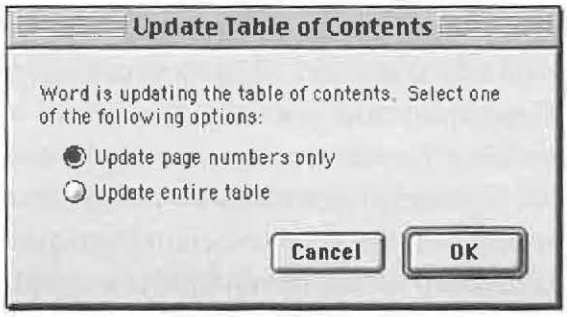



***Figure 5.5. Dialog box with check boxes for different kinds of searches, and one launch button.***

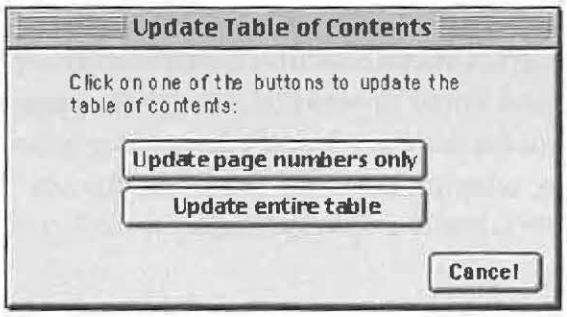



***Figure 5.6. More efficient dialog box with different launch buttons.***

Delimited-search dialog boxes usually provide a facility that allows you to reuse the current pattern in a search that begins just after the last instance was found. This can be thought of as a "look again" or "find next" facility; in some implementations, it is invoked by operating the same button used to launch the initial search. With an incremental search facility, a command dedicated to searching again for the same target is required, because no command was necessary to launch the original search. It is dangerous to use a tap of the Search quasimode key to repeat the search, as a user might intend to do a search, press the key, and then change his mind and release it. Under such circumstances, an unwanted search would be started, and he could lose his place as a result. Although a complete undo facility can ameliorate the danger, it is better not to create the problem. A specific method for repeated searches is discussed in Section 5-6.

*In larger texts, a search can proceed circularly not only through the local document but also afterward in automatically expanding domains, up to and including the entire Internet.*[10] After the local document has been completely searched, the search then proceeds through the following documents until the end of the folder is reached, when the search continues from the beginning of the first document in the folder until the current, already searched document is reached. After cycling through the folder, the next larger domain is treated similarly, and so forth. When, during an incremental search, a user sees that his search has gone too far afield, he can stop searching, confident that any nearer instance of the pattern does not exist. It is usually easy to tell in what domain you are searching, because you see the results in context; what you see is not just a list of file names, as with many present search systems.

Generally, people will use more efficient strategies than simply relying on this expanding sweep of the hierarchy. For example, if you are looking for a particular document in the current folder, you would be likely to do a repeated search for document characters and would thus quickly be able to review the beginning of, and any heading on, each document. When the desired document is found, an incremental search for the target is initiated. This guarantees that the chosen document is searched first; the advantage of this is that you can typically use a shorter pattern if the area to be searched is smaller. If you did not know in what document the target resided or if you did not want to look for the document, you could perform a specific search from the beginning, which would find the target anyway.

---

10. It is entirely possible, by such techniques as lookahead and indexing, to maintain the 250 msec response time.

## 5-5 Cursor Design and a Strategy for Making Selections

The traditional intent of both delimited- and incremental-string searches is to find and to select a target string in the text. Patterns that people use in searches tend to be short because long patterns are tedious to type and, in most systems, must be accurate to the character if they are to match the target. Therefore, string searches typically are not used to select even moderately large targets—say, those greater than 10 or 15 characters in length—much less truly large blocks of text. One application of string searches is to help you find the location of a desired selection, after which you use another technique, such as using a GID to drag from one end of the selection to the other, to establish the selection. However, if the ends of the selection are not visible simultaneously, you must use another strategy. The strategy consists of (1) marking one end of the selection, with the method of marking depending on the system being used; (2) using such facilities as scroll bars to make the other end of the selection visible; and (3) marking the other end of the selection. In most systems, marking the second end of the selection establishes the desired selection.

A more efficient approach is to design the search mechanism such that it positions the cursor on a particular character. Two such positionings can be used to define the first and last characters of a selection. The multiplicity of mechanisms conventionally needed to find the ends of a selection—cursor motion, scrolling, various page and pattern find facilities, and so on—and to mark them are replaced with one mechanism used twice, making learning, operation, and habituation easier and simplifying the implementation.

It is now useful to examine the graphical design of cursors. Currently, the most common form of text cursor is one that appears between characters, as shown in Figure 5.7. One problem with the standard text cursor is that users try to place it precisely between pairs of characters, aiming at a smaller horizontal target than is necessary and thus incurring a Fitts' law time penalty. During testing at Information Appliance, we were surprised to discover that this popular form of cursor caused an interesting cognitive

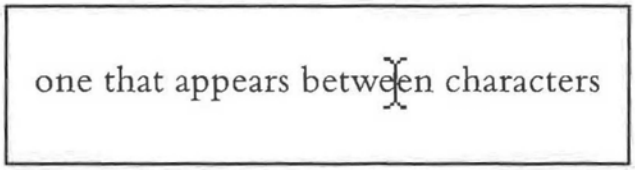



***Figure 5.7. The standard between-character cursor.***

problem as well: You must place the cursor differently, depending on what you plan to do next. In particular, if you wish to delete an existing character with the usual Backspace key, you place the cursor to the right of the character in English and other left-to-right languages; to insert a character in the location of an existing character—the existing character will be shifted to the right when you do so—you place the cursor to the left of the existing character. We were surprised because the standard cursor's operation is so familiar that we had not given any thought to its possibly being the cause of problems.[11] As anybody who has tried to use one knows, the standard cursor is not difficult to understand. Nonetheless, a small amount of confusion and a few errors can be observed when newcomers to computers first use it. Their confusion is exacerbated by the invisibility of the loci of action of the commands.

It is no solution to provide a mode in which deletion occurs in a direction opposite to the usual one. This method, called forward erase, does allow you to always place the cursor to the left of the character to be acted on, but you will sometimes unexpectedly delete in the wrong direction because the current deletion direction is not your locus of attention. If two-way deletion is desired, it is better to use separate buttons or to use a quasi-mode for opposite-direction deletion.

A useful improvement to text-related interfaces is to use a cursor that visually indicates both (1) the insertion location and (2) the character or characters that will be deleted if backspace is used. The second form of highlighting can be identical to that used for highlighting a selection. One way to do this is shown in Figure 5.8.

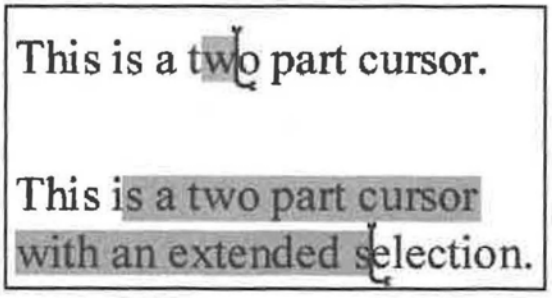



***Figure 5.8. A two-part cursor, both with a single character and with an extended selection. The insertion portion of the cursor is blinking to make it easier to find on a cluttered screen.***

11. This example is another demonstration of the importance of testing an interface on the intended audience.

Moving a cursor to a target with a GID is an iterative process; the user visually acquires the target first and then moves the cursor toward the target. If she does not remember where the cursor is, she may have to search visually to find it as well. She then reacquires the target, and so on, until both target and cursor are in the same, approximately 5 degree, foveal field of vision. This amounts to several time-consuming processes the human must undertake in cursor moves using a GID. Nonetheless, a GID is required for pointing to graphical objects or when working in a totally graphic environment.

The sloth of the GID in text pointing is exacerbated in the case of having to point to off-page locations. In these cases, exclusive use of a GID forces you to use scroll bars, page number selectors, or other mechanisms to view the content to which you wish to point. Each of these methods has to be learned, and many of them are slow.

LEAP offers the user the cognitive advantage of not having to choose, depending on the remoteness of the target, between different strategies (Alzofon et al. 1987). Having both LEAP and a GID does mean that there are two basic cursor-moving methods; however, instead of choosing your method depending on the distance between the cursor and your target, you choose your method depending on the content of the target. As is usually the case, the content is your locus of attention, which makes a LEAP/GID split cognitively simpler.

LEAP is useful in voice-controlled systems and for users with motor impairments or repetitive stress injuries, for whom the minimization of keystrokes is of especial value. A button, perhaps placed on the microphone, establishes the LEAP quasimode. The target of a LEAP is always a particular character in the text, and that character—whether the user is seeing it or thinking about it—is usually the locus of attention during the LEAP. Unlike using a GID, the user does not have to visually find the target before being able to move the cursor to it. This property is so strong that LEAP can be effectively used by the blind. This was tested at the Veteran's Administration Hospital in Palo Alto (unpublished).

What is most important, however, is that LEAP is used so widely in systems designed around it that its use soon becomes automatic, and there is nothing in the design of LEAP that prevents beneficial habit formation or that causes problems after habitual use is established. For LEAP to work properly, it must be fast, always finding the next instance of the current pattern within human reaction time, thus not giving the user any motivation or time to act during the search. The necessary speed can be achieved by the methods discussed in Section 5-4.

A GOMS keystroke-level analysis can be used to compare GID and LEAP timings. Where the user's hands start on the keyboard, using a GID to point to a letter in text requires operations *HPK*. By the rules for placing the *M* operator, we obtain *HMPK*, or 0.4 + 1.35 + 1.1 + 0.2 = 3.05 seconds. The time to perform a LEAP depends on the number of characters you have to type to reach your target. Testing showed that the average number of characters used in LEAPing was about 3.5 for operators tested during their first week of use of the Canon Cat. The operations required to move the cursor to a particular target are pressing the LEAP key and then typing 3.5 characters. An average LEAP thus consists of typing 4.5 characters. Adding the *M* operator gives, according to the rules, a time of 1.35 + (4.5 * 0.2) = 2.25 seconds.

A timed experiment, with experienced users as subjects, compared the mouse with LEAP. The cursor had to be moved between two randomly chosen characters on a display 25 lines by 80 characters and where the timing did not start until the mouse began to move or the LEAP key was pressed. The measured average times were about 3.5 seconds for the mouse and 1.5 seconds for LEAP. The longer-than-calculated times for the mouse are probably due to the small size of the targets, which were individual characters: a Fitts' law effect. The unexpectedly short times for LEAP are probably due to the small size of the text, which allowed an average pattern length of about 2 characters. The subjects were given unlimited time for planning their actions before using the mouse or LEAP. *Very often, you can complete a cursor move with LEAP faster than you can move your hands from keyboard to mouse.*

## 5-6 Cursor Position and LEAP

The target of a LEAP is a single character. Should the cursor land to the right or to the left of this character? Placing the cursor to the left of the character is correct only if you intend to insert at that character location. Placing the cursor to the right of the character is correct only if you intend to delete that character. The computer, it would seem, has to know your intentions before it can position the cursor properly.

For insertion, consider instead the old-fashioned cursor that forms a rectangle about or an underline beneath the letter. When you LEAP to a letter, the cursor should land *on* the letter itself. (The cursor does not interfere with the readability of the letter; see Figure 5.9.) Now you can insert or delete at that location without ambiguity. The block cursor allows you to more accurately indicate where insertion and deletion will occur than does the standard, between-character cursor.

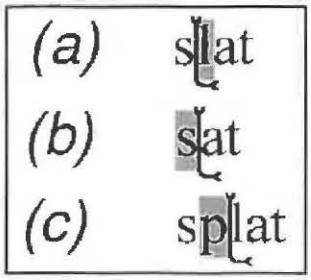



***Figure 5.9. (a) The cursor has been placed by a GID move or a LEAP to the letter* l *in* slat. *(b) If you backspace, you get* sat. *(c) But if you had typed a* p, *you'd get* splat. *Note that in (a), the insert and delete portions of the cursor are both on the* l, *indicating that you can either insert or delete there. After deleting (b) or inserting (c), the insert and delete portions of the cursor have separated, still indicating where insertion and deletion will take place. Using a cursor-moving (arrow) key also causes the cursor to separate.***

This solution engenders another problem: Suppose you had the word *tongs*, with the cursor on the *o*, and you type the letter *h*. You get *thongs*. But where should the cursor go? Again, you seem to need to know the intent of the user: If the cursor is on the *h*, does it indicate whether that letter will be the insertion point or the deletion point? Deletion should be the inverse of insertion, so that a tap of the Backspace key should delete the *h*, returning the word to *tongs*, but if you type another letter, say *r*, it should displace the *o* to the right of the *h* to make the word *throngs*. It would seem that an on-character cursor also has to have precognition to be in the right place.

The solution to the problem is this: When you move the cursor, it lands on a single character. As soon as you type or delete, the cursor splits into two parts that are associated with two consecutive characters, as shown in Figure 5.9: The first character has the delete cursor on it, the second has the insertion pointer wrapped partially under it. One part of the trick is to have a two-part cursor designed so that when the parts coalesce—as they must, after a LEAP or other cursor movement—it is graphically clear that they both reside on the same character independently. This may all seem complicated when written out in detail, but the LEAP cursor is easier for a newcomer to learn than is the conventional cursor of Figure 5.7. All that you have to tell a learner is what the selection and insertion markers look like, that the Delete key deletes the selection, and that what you type appears at the insertion marker.

The characters are consecutive rather than adjacent, for although two consecutive characters are usually adjacent, they are not when the first of the two is at the end of a line. For left-to-right languages, the delete cursor is

typically to the right of the insert character; for right-to-left languages, it is the other way around. For vertically oriented languages, the insert cursor would be below the delete cursor, and they would alternate directions on each line for boustrophedonic scripts.

Target acquisition is easier when using a GID with the two-part cursor, because the targets, or characters, are larger than the spaces between them, and the hot region is easier to visualize. (The hot region for the PARC cursor extends about halfway into each character, on both sides of the space between them, but nothing visually delineates the boundary, making the target seem smaller. In practice, many users try to place the cursor between characters, incurring the Fitts' law penalty that goes along with small targets.)

As we have discussed, naive users sometimes have trouble understanding how the standard, between-character, cursor works: a transient phenomenon involving an interface detail that is so familiar that it is rarely, if ever, questioned. With the two-part cursor that condenses on a single character after a cursor move, the initial confusion about which character a cursor points to disappeared, solving the original problem.

At times, it may be advantageous to limit the range of a LEAP. A limited search can be made by (1) selecting the region to be searched; (2) selecting a command or current selection, as discussed in Section 5-2-1, that limits the next LEAP to that region, which becomes the old selection; (3) and then using a LEAP. What must not be done is to create a mode that restricts LEAP. When the user is not aware of that mode, using LEAP will be very frustrating, and he will not be able to find things he has always otherwise been able to find. Canon introduced a concept called "Local Leap" on the Canon Cat, and it caused precisely the problems that a student of this book would predict.

Another function needed when LEAP is implemented is LEAP AGAIN, which simply performs a LEAP in the same direction as the most recent LEAP, to the next instance of the same pattern. In SwyftWare and the Canon Cat, searches usually took the form of LEAPing to a pattern and then using LEAP AGAIN repeatedly until the desired instance was found.

LEAP unifies searching and cursor motion in text, recalling that text includes spreadsheets, labels in graphics, and any other use of alphanumeric characters. After a short while using LEAP, the user becomes unaware of LEAPing, just as a touch typist concentrating on creating content is unaware of the mechanical motions of typing. When you want to put a target on your display, you LEAP to it; you do not stop to ask where in the system's universe of content the target is. You do not track down any hierarchies or open folders; you LEAP directly to what you want. But when people accus-

tomed to standard file structures first encounter a system with LEAP, they often try to think in terms of where the desired item is hierarchically, searching for it by trying to find the general location and then working their way in to the specific instance they seek. If the universe over which you are searching has been properly indexed, it makes no difference to the user whether the item searched for is in memory, on local or networked mass storage, or on a local or global network.

When considering searching methods, the observations of Landauer and his colleagues are pertinent. They have shown that the most common forms of advanced text search, where the Search key is specified in terms of Boolean combinations of strings or by means of regular expressions, are inferior to a full-text search mechanism. The latter is both faster and easier, and users find a higher proportion of the relevant items for which they are searching (Landauer 1995). LEAP, on the basis of both GOMS and efficiency measures, is superior to the search-launching methods used by Landauer and should therefore show an even greater advantage.

## 5-7 Applications Abolished

*The time it takes your favorite application to complete a given task doubles with each new revision.*

—*Lincoln Spector*

*The present structure of computer software, consisting of an operating system under which application programs execute, is inherently modal. This implies that for an interface to be nonmodal, an approach that does not include applications in their present form is required.*

Because gestures, such as those that invoke commands, in one application may not be available in another, you must be conscious of which application is currently active. But you cannot reliably do this when your locus of attention is the task you are trying to accomplish. You will sometimes use a gesture with either no result or the incorrect result. A separate difficulty caused by application programs is the unavailability of the facilities of one application when you are in another; you wish to do a task that you could have done in application *A*, but you are in application *B*, which has no comparable command. Computer scientist Dan Swinehart called this the dilemma of preemption (Tesler 1981, p. 90).

Three approaches at solving the preemption dilemma are well known. The most common method is to provide, in each application, all

those facilities the user is likely to need. This was discussed in Section 5-1, where it was pointed out that every personal computer has many different text editors, each part of an application or other facility. Most personal computer word processors have multiple and different text editors; for example, a typical word processor has a weaker editor for typing a pattern into the Find dialog box than for working in the main body of the text. This approach forces applications to become gargantuan, as each of them must solve a host of user needs that are peripheral to the primary thrust of the software. For example, my word processor has an embedded drawing program so that I can create simple illustrations without having to leave the text editor. At the same time, my drawing program has an embedded text editor so that I can include formatted blocks of text in my drawings without having to leave the drawing program. The drawing facilities in the editor and the text-editing facilities in the drawing program are inferior to the software dedicated primarily to those tasks. Ideally, all of the commands and abilities of both the drawing and the editing programs should be available at all times.

Similarly, every program has facilities for saving and retrieving named files. However, these facilities operate differently and have different features in different programs. This is confusing, difficult to operate, and requires immense amounts of largely redundant software, all of which you must pay for, learn to use, keep the documentation for, and provide adequate main memory and mass storage. The same is true of a host of other features, such as printing, that are duplicated or nearly duplicated among programs.

There has been some industry recognition of these difficulties; a number of companies designed software that permits a single **compound document** to have parts created by different applications. When you click at any point in a compound document, the application that created that portion of the document becomes active. Once the compound document has been created, this technique allows you to avoid having to open the individual applications explicitly. Of course, to create such a document, you must still manually invoke the applications, create the parts of the compound document, and then assemble them, usually by cutting and pasting or by dragging.

Although providing a modicum of convenience, the dilemma of preemption is not solved by this method, which is exemplified by Apple's OpenDoc, HP's NextWave, and Microsoft's OLE software and their descendants. When you are working in one part of a compound document, you do not have the facilities of the applications used to create the other parts of the document. Worse, you now have a document that seems to have no boundaries but whose behavior changes from place to place without warning. A

table and a spreadsheet may look identical, but one operates according to the rules of the word processing program, and the other operates according to the rules of the spreadsheet program. This is modality with a vengeance. The only warning you get as you click here and there is that the menus, which are usually positioned far from the locus of attention, change. As we have seen, this is an ineffective means of alerting a user as to system state; there is, of course, no completely effective means.

The original approach to eliminating the modes inherent in applications was to create the windowing paradigm. At Xerox PARC, Alan Kay proposed overlapping windows, in part to eliminate the modality of applications. He also wanted to eliminate the distinction between operating system and applications but succeeded primarily in making the functioning of the operating system visible in the form of the desktop. This was a true advance, but as PARC's Larry Tesler put it, "windows are, in a sense, modes in sheep's clothing" (Tesler 1981, p. 94). That is, windows do not eliminate the modality of applications but instead make multiple applications simultaneously visible and accessible. Kay's insight, and the other ideas developed around windowing, formed an important step forward that has benefited users for more than a decade, but the problem of modes and the preemption dilemma posed by the existence of applications had not been solved. Since that time, the sheep disguise has worn thin, and the wolf bites us all too often. Section 5-8 presents one way of dispatching the wolf completely.

---

### *Calculator or Computer?*

It's true. Many of us keep a calculator beside our computers. Why do you need this simple-minded device when you have a whole computer in front of you? You need it because you have to go through contortions worthy of a circus sideshow in order to do simple arithmetic with the computer. There you are, tapping away at your word processor, when you want to do a division: 375 packages of Phumuxx cost $248.93; what is the price for one package? On my computer, I have to open up a calculator window. To do this, I move my hand from the keyboard to the mouse, which I use to do a click-and-drag to open the calculator. Transferring my hands back to the keyboard, I type in the numbers I need or tediously cut and paste them from my document. Then I have to press a few more keys and finally copy the results from the calculator window into my document. Sometimes, the calculator window opens right on top of the very numbers I need, just to add insult to injury. In that case, I must

use the mouse to move the calculator window out of the way before proceeding. It is much faster to grab the pocket calculator.

A better solution is a Calculate button or omnipresent menu command that allows you to take an arithmetic expression, such as 248.93 / 375, select it, and do the calculation whether you are in the word processor, communications package, drawing or presentation application, or just at the desktop level. In other words, it is an example of a universal function that can be used anywhere.

Using an experienced computer and calculator operator as my test subject, with his word processing program open before him, I measured the total time it took for him to pick up a calculator, turn it on, do a simple addition, and return his hands to the keyboard to resume typing. It took about 7 seconds. I then measured the time it took for him to use the built-in calculator. He had to move the cursor to the menu bar at the top of the screen, find the calculator program, open the calculator, enter the sum, and then click back in the word processor so that he could resume typing. This took about 16 seconds.

The same subject was asked to do the calculation from the middle of a document on a Canon Cat, which has a built-in Calculate key and can do arithmetic in text. The time was 6 seconds, using the top-line numeral keys for data entry. There is no time advantage to using an external calculator in this case. On the Cat, the result was left in the document, in the likely case you needed the result inserted. But the result was also left selected, so that a tap of the Delete key made it vanish if you did not want it in your text.

Here's another facility that should be generally available: Anywhere a number can be entered, you should be able to enter an arithmetic expression that evaluates to the number. Commands such as

- Check the spelling of the current selection
- Treat the current selection as an arithmetic expression and evaluate it
- Transmit the current selection as an e-mail
- Transmit the current selection as a fax
- Let's see what's at this URL on the web
- Execute the current selection as a Java (or whatever) program

should be available at all times. It is eminently doable.

## 5-8 Commands and Transformers

*A good design is better than you think.*

*—Rex Heftman*

We are justifiably annoyed when a product, a piece of software, or a computer system imposes complexity beyond our needs and presents difficulties we do not understand. We want to do some simple word processing, but we are forced to take in hand hundreds or, as in the case of Microsoft Office, thousands of commands and methods that we do not need. If, on the other hand, we brought in each of those commands as we understood our need for them, then the feeling of imposition and the immense cognitive burden would be lifted, even if the eventual system turned out as complex as the original application.

By applying the concept that a system should not be more complex than your present needs and by allowing the system to increase its power incrementally, the dream of providing products that are initially truly simple can be achieved without their being made to merely look simple and without impairing their flexibility. To understand one means of doing this, recall from Section 5-1 that almost all that a computer does involves the content that you provide or obtain and a set of operations you wish to perform on this content. Also recall that the interface to every such operation consists of two parts: choosing the content and invoking the operation. For example, in a game, you might shoot a monster, or, in our more prosaic terms, perform the operation of changing the image of a monster into an image of an explosion. The content is chosen by moving the cursor to the monster, and the operation is invoked by pressing the GID button—a lovely interface, by the way: fast and satisfying.

Invoking a command on a selection results in three possibilities:

1. The operation can be performed on the selection.
2. It makes no sense to apply that operation to the selection.
3. The selection can be modified to make the operation possible.

In the first case, the operation is performed and the content modified. In the second case, the content is left unchanged. In the third case, another process must be invoked by the computer to modify the selection before the command can be performed.

Suppose that you select a part of a photograph of a city street scene and try to apply the command that checks spelling. The command expects a

sequence of characters (text), but what it finds is a photograph (a bitmap). Among the computer's capabilities is the ability to transform content from one data type to another.[12] In this case, the data type is bitmap and the command expects text, so the computer sees whether any transformer has bitmaps as inputs and text as output. Such transformers exist; they are called optical character recognition (OCR) programs. OCR programs are usually employed to convert input from scanners into editable text. (Some of the quotes in this book were obtained by scanning the article that contained the quote and then using an OCR program to convert the resultant bitmap into text.) Assume that the computer has an OCR program available. The OCR program is automatically invoked and inspects the bitmap. If it finds some characters it recognizes—say, a stop sign and a partly obscured sign with the characters *North First Stree,* then these texts are presented to the spelling checker, which reports finding one unknown word *(stree)* and suggests that perhaps tree, strew, or street was meant.

Instead of a computer's software being viewed as an operating system and a set of applications, the humane interface views the software as a set of commands, some of which are transformers that can be invoked automatically when the data type expected by the command does not match the data type of the object selected. More than one transformer may have to be invoked. For example, suppose that a computer with one transformer converts *A* to *B* and another that converts *B* to *C;* if the command expects data type *C* but the data type of the selection is *A,* both transformers will have to be applied before the command can be carried out. If no transformers are available to do the job, no change is made to the selection; the user is informed, if necessary, and the selection is left unmodified.

Instead of providing application programs, software vendors will provide command sets that offer a collection of related operations. For example, instead of a program for doing photographic manipulation, a vendor would provide a number of independent commands that, collectively, provide exactly the same range of operations. The user would install as many or as few as she finds useful rather than having to install a huge application, of which only a fraction would be used. By means of Internet sales of software, a forward-thinking vendor could sell software on a command-by-command basis, perhaps with significant discounts for buying certain sets or certain numbers of commands.

---

12. This class of abilities is sometimes called *filters,* but the term *filter* implies selectivity rather than a general transformational ability. Thus, I prefer the term *transformer,* which is seldom used in computer science, outside of circuit descriptions, whereas *filter* has many uses, such as a *digital filter program.*

When users complain about the absurd complexity of applications and ask for simpler programs without what are perceived to be unnecessary bells and whistles, industry observers respond that "lite" versions of software packages tend not to succeed in the marketplace. There is good reason for the failure of lite packages. Because the user never knows which of the facilities offered by the complete package he might some day need, he purchases the whole package because that is the only way he can obtain these facilities. If he buys a limited, lower-cost version, the only way he can upgrade is by buying the complete package, even if it is only one small feature from the larger program that he would need to add to the lite version. Therefore, it is safer to purchase the full version from the beginning and to put up with the complexity. No wonder we feel trapped. Commands that could be purchased as needed are a more humane option.

Various surveys report a range for the amount of typical software application features that are never used. The numbers have tended to rise over the last decade and have increased over that time from about 15 percent to the latest surveys nearing 50 percent. That is a lot of clutter. By providing command sets, each command of which can be installed independently, instead of applications, individual users can drive that statistic to nearly 0. Other advantages for vendors in this methodology are the ability to refine their products incrementally and to more easily and frequently provide (and sell!) new features, because they do not have to rerelease the whole package in order to do so. The web is an ideal medium for such repeated, piecemeal sales.

Vendors, not necessarily the vendors of application command sets, will supply transformers. These, too, can be sold individually. If a majority of users regularly employ most of the word processing commands from vendor *A*, and vendor *B* thinks of a useful command that *A* does not provide, then *B* can sell that command to *A*'s customers. However, *B* might use a different data structure. In that case, *B* would be wise to provide a pair of inverse transformers that go from *A*'s data structure to *B*'s data structure, and back again. If *A*'s product has a very large customer base, then *B* might be wise to create a version of the command that works directly with *A*'s data structure. Besides, another vendor, *C*, might be a specialist at providing transformers. It may become common practice for users to purchase transformers from such vendors and for command vendors to license the transformers they need. This commercial structure exists, in part, at present: Companies such as DataViz specialize in providing transformers.

The pieces needed for a commands-plus-transformers computer environment also exist. It will not be difficult to put them together into a

working system. Users will find such systems both easier to use and more flexible than today's application-bound designs. It is a solution that can decrease excessive feature bloat and complexity, incompatibility among applications, and the need to learn multiple solutions to the same problem. In the process, the operating system can disappear from the user's view entirely. If this program is carried out properly, not even the present, wool-coated version of the operating system—the desktop—will remain.

Of course, not all programs require all this mechanism. A game, for example, would simply be launched and would run independently: This is done in the usual way by clicking on, or selecting, the desired game's name, probably from a text consisting of a list of names of games, and then using the Execute command. Section 6-2 discusses an alternative method.

---

### *Considerate Programming: Applications as Visitors*

Consider that you have been invited to stay at the house of your good friends, the Grimbles. You even like their dog. The only problem is their beloved portrait of Aunt Ashtabula that hangs over the bed. It gives you and your wife the willies, and you can't possibly sleep in a room with Aunt Ashtabula looking on. You dare not tell the Grimbles how you feel, so how do you handle this situation when you visit?

**1.** Take down the picture and burn it.

**2.** Hide the picture in the wine cellar, where it will take the Grimbles a month to find it.

**3.** Put the picture in the closet, where they will find it sooner.

**4.** Put the picture in the closet and restore it to its original position just before you leave.

Any considerate guest knows that option 4 is the best of those presented. The ethical principle is: Make a change in someone else's environment if you must, so long as you leave things just as they were before the host returns.

Today's computers have many environmental parameters that can be set: speaker loudness, screen resolution and depth, menu behavior, and default system font. A Macintosh computer has hundreds of settings. In an IBM-compatible computer running Windows and Microsoft Office, the number is over 1,000. The parallel with the

visit to the Grimbles? If you go and use somebody else's machine and change any settings, it would be considerate of you to change them back when you leave.

Many programs demand a particular screen resolution, a certain number of bits per pixel, or another specific parameter setting to run properly. Their behavior, when the system does not have the correct parameter settings, ranges from genteel to uncouth to vandalism. Here is what various programs I tested do when they encounter incorrect display parameters:

**1.** Crash the computer, forcing a manual reboot.

**2.** Crash the computer, giving an indecipherable numeric message; you must press the Restart button to reboot the computer.

**3.** Give an error message saying that the screen settings must be adjusted, and when you click on the OK button, it crashes the computer.

**4.** Give an error message saying that the screen settings must be adjusted. When you click on the OK button, you can then open the appropriate control panel and adjust the settings.

**5.** Ask whether it should change the screen settings, and if you click on the OK button, the settings are changed. If you click Cancel, the program is not launched, and the screen settings are not touched.

**6.** Change the screen settings and run the program without comment.

**7.** Announce that it will change the settings in a dialog box. When you click on the OK button, it changes the settings and runs the program; when you quit the program, it changes the settings back to the way they were.

A reader from the future may think that crashing the computer is a bit extreme. None of these methods goes all the way to being a good guest, although item 7 is close.

The safest behavior is a combination of item 5's message and item 7's exit behavior. In the unlikely event that changing the screen settings will interfere with some other parallel or background process, you might decide not to run the program and might welcome the warning and a chance to opt out.

**8.** Change the screen settings and run the program without comment and reset the settings when exiting.

In other words, you'd launch the program, and it would simply run: doing just what you expected and wanted, and no other program would be affected. That is my definition of a well-behaved computer system. But what if two programs requiring different screen settings are displaying pictures at the same time? That is a horse of a different color or a fish of a different resolution, perhaps.

Item 7, which might seem acceptable and is in common use, fails us by giving us a "dialog" box that is no dialog. This is user abuse: The information imparted cannot be acted on, and it only wastes your time. (The transparent-message method of Section 5-2-3 solves this problem.)

Application programs per se are an imposition, as this book argues, but so long as we are still saddled with them, let us make sure that they are well behaved (also see Raskin 1993). To summarize, a program or, in the future, a command should automatically reset whatever it needs in order to run; it should execute properly; and when the user dismisses it or when it is done, it should restore all parameters to the state they had when it was launched. If resetting is irreversible or could have an undesirable side effect, a warning explaining what damage might be caused can be given, so long as that message gives the user a choice as to proceed or not.

Right now, you cannot run different resolutions on the same screen at the same time. But that's another kind of problem. In the meanwhile, I think that users would be happier if every user interface guideline added item 8.

SIX

# *Navigation and Other Aspects of Humane Interfaces*

*The average man suffers very severely from the pain of a new idea.*

*—Admiral William S. Sims*

One of the most laudatory terms used to describe an interface is to say that it is "intuitive." When examined closely, this concept turns out to vanish like the pea in a shell game and be replaced with the more ordinary but more accurate term "familiar."

Our present systems of navigation, never satisfactory in the first place, are completely inadequate in the face of the terabytes of information we have to scan. But people and animals have been navigating through complex environments for millennia and have some useful techniques for doing so. These abilities, which have evolved over the eons, can be co-opted to our purposes with what can be termed a "zooming interface paradigm."

## 6-1 Intuitive and Natural Interfaces

*In every respect the burden is hard on those who attack an almost universal opinion. They must be very fortunate as well as unusually capable if they obtain a hearing at all.*

*—John Stuart Mill, from "The Subjection of Women"*

Many interface requirements specify that the resulting product be intuitive, or natural. However, there is no human faculty of intuition, as the word is ordinarily meant; that is, knowledge acquired without prior exposure to the concept, without having to go through a learning process, and without having to use rational thought. When an expert uses what we commonly call his intuition to make a judgment, with a speed and accuracy that most people would find beyond them, we find that he has based his judgment on his experience and knowledge. Often, experts have learned to use methods and techniques that nonexperts do not know. Task experts often use cues of which others are not aware or that they do not understand. Expertise, unlike intuition, is real.

When users say that an interface is intuitive, they mean that it operates just like some other software or method with which they are familiar. Sometimes, the word is used to mean *habitual*, as in "The editing tools become increasingly intuitive over time." Or, it can mean *already learned,* as was said of a new aircraft navigation device: "Like anything, it can be learned, but it would take a lot of experience to do it intuitively" (Collins 1994).

Another word that I try to avoid in discussing interfaces is *natural.* Like *intuitive*, it is usually not defined. An interface feature is natural, in common parlance, if it operates in such a way that a human needs no instruction. This typically means that there is some common human activity that is similar to the way the feature works. However, it is difficult to pin down what is meant by *similar.* Similarities or analogies can occur in many ways. Certainly, that the cursor moves left when a mouse is pushed to the left and right when the mouse is pushed to the right is natural. Here, the term *natural* equates to *very easily learned.* Although it may be impossible to quantify naturalness, it is not too difficult to quantify learning time.

The use of the mouse itself is often claimed to be natural and intuitive. It is difficult to do the experiment now, when the use of this most famous of GIDs is so widespread, but when it was less well known, I asked people unfamiliar with the mouse to use a Macintosh. My protocol was to run a program called The Manhole, an entertaining and well-designed chil-

dren's exploration game that required no input beyond clicking at various locations on the display. With the keyboard removed from the computer, I would point to the mouse and say, "This is the mouse that you use to operate the game. Go ahead, give it a try." If asked any questions, I'd say something nonspecific, such as "Try it." The reaction of an intelligent Finnish educator who had never seen a Macintosh but was otherwise computer literate was typical: She picked up the mouse.

Nowadays, this might seem absurd, but the same point was made in one of the Star Trek series of science fiction movies. The space ship's engineer has been brought back into our time, where (when) he walks up to a Macintosh. He picks up the mouse, bringing it to his mouth as if it were a microphone, and speaks to it, with a heavy Scots accent: "Computer, . . ." The audience laughs at his mistake. I admired the creators of the film for recognizing that the use of the mouse was not something you could expect everyone to immediately guess. In the case of my Finnish subject, her next move was to turn the mouse over and to try rolling the ball. Nothing happened. She shook the mouse, and then she held the mouse in one hand and clicked the button with the other. No effect. Eventually, she succeeded in operating the game by holding the mouse in her right hand, rolling the ball on the bottom with her fingers, and clicking the button with her left hand.

These experiments make the point that an interface's ease of use and speed of learning are not connected with the imagined properties of intuitiveness and naturalness. The mouse is very easy to learn: All I had to do, with any of the test subjects, was to put the mouse on the desk, move it, and click on something. In five to ten seconds, they learned how to use the mouse. That's fast and easy, but it is neither intuitive nor natural. No artifact is.

The belief that interfaces can be intuitive and natural is often detrimental to improved interface design. As a consultant, I am frequently asked to design a "better" interface to a product. Usually, an interface can be designed such that, in terms of learning time, eventual speed of operation (productivity), decreased error rates, and ease of implementation, it is superior to both the client's existing products and competing products. Nonetheless, even when my proposals are seen as significant improvements, they are often rejected on the grounds that they are not intuitive. It is a classic Catch-22: The client wants something that is significantly superior to the competition. But if it is to be superior, it must be different. (Typically, the greater the improvement, the greater the difference.) Therefore, it cannot be intuitive, that is, familiar. What the client wants is an interface with at most marginal differences from current practice—which almost inevitably is Microsoft Windows—that, somehow, makes a major improvement. This can

be achieved only on the rare occasions when the original interface has some major flaw that is remedied by a minor fix. (Parts of this section are based on Raskin 1994.)

## 6-2 Better Navigation: ZoomWorld

If you wanted to design a navigation scheme intended to confuse, you might begin by making the interface mazelike. The maze would put you in a little room with a number of doors leading this way and that. The doors' labels are usually short, cryptic, or iconic, and they may change or disappear,[1] depending on where you've been. You cannot see what is on the other side of a door except by going through it, and when you have gone through, you may or may not be able to see the room you've just left. There may not be a way to get directly back at all. Some rooms may contain maps to part or all of the system of rooms, but you have to keep track of the relationship between the map representation and the rooms you are presented with; furthermore, maps are not well suited to situations best represented by three-dimensional networks. The rooms in this description correspond to computer interface windows and web sites, and the doors are the tabs, menus, or links that are provided to bring you to other windows or sites.

As legends and stories from ancient times inform us, humans always have been notoriously bad at mazes. If we could handle them easily, they wouldn't be used as puzzles and traps. When using a complex program, I often find, deep in a submenu, a command or a check box that solves a problem I am having. When I run into the same problem a few weeks later, I cannot remember how I got to the box with the solution. We are not good at remembering long sequences of turnings, which is why mazes make good puzzles and why our present navigational schemes, used both within computers and on the web, often flummox the user. Many complaints about present systems are complaints about trying to navigate. Partial solutions, such as "favorite locations" in browsers, have been created.[2] But what we are truly better at is remembering landmarks and positional cues, traits that evolution has bred into us and traits we can take advantage of in interface design.

The antithesis of a maze is a situation in which you can see your goal and the path to get there, one that preserves your sense of location

1. Adaptive menus have this annoying trait.

2. This works until you have so many that you cannot remember what they all are; then you need a "favorites of favorites" or another scheme to keep track of them.