> another key while they are typing rapidly. My wife got into a problem yesterday that took me quite a few minutes to figure out. Every time she pressed Return, she got a bullet. Looking at the Edit pull-down menu, I saw "undo Autoformat." After several minutes of searching and using Help, I found where Autoformat could be turned on or off and turned it off. Somehow, she had typed a keyboard shortcut that had turned it on. (personal communication 1998)

This user was injured by a customization, a mode, an invisible keyboard shortcut, and excessive design complexity all at once.

Customizations are software design changes that are not reflected in the documentation. For example, when I was using Word, I tried to turn off a feature that was unfamiliar to me. The Help facility instructed me simply to click a certain button on the standard toolbar; however, a previous user had modified this toolbar using a user-preference setting, so that the button was not there. It took me a long time to figure out how to make the program behave. More important, the incident pointed out another fundamental problem of having user preferences: How do you test for interface quality or even succeed at documenting a system whose configuration the designers and writers cannot know in advance? In my case, the previous user's mode change rendered the documentation wrong.

Allowing the user to change the interface design often results in choices that are not optimal, because the user will, usually, not be a knowledgeable interface designer. Typically, a user will choose the method closest to one with which he is already familiar or a customization needed only temporarily. Some designers have argued that applications for high-level users should have many preference settings so that a user can tailor the system as much as possible. However, high-level users have no special claim to being good interface designers and, being habituated to their software, have an especially strong need for a stable system so that their habits will not be rendered useless by changes, even changes they themselves make.

By providing preferences, we burden users with a task extraneous to their job function. A user of, say, a spreadsheet has to learn how to use not only the spreadsheet but also the customizing facilities. *Time spent in learning and operating the personalization features is time mostly wasted from the task at hand.* Managers complain about the time workers waste "playing with" the system preferences. Most users just want to get their jobs done and care not whether the numbers in the spreadsheet default to Palatino in purple, Garamond in green, or Bodoni Semibold Extended Italic in bellwether blue.

Personalizing an interface in a shared environment is an invitation to disaster, as it means that the interface can change without notice. The right action yesterday—say, clicking the red button—is clicking the blue button today, because someone thought it looked better in blue. Your training and habits are undermined. Especially over the telephone or via e-mail, it is also more difficult to help the user of a system that has preferences.

Customization sounds nice, democratic, open-ended, and full of freedom and joy for the user, but I am unaware of any studies that show that it increases productivity or improves *objective* measures of usability or learnability. Adding customization certainly makes a system more complex and more difficult to learn. I suspect that if you were to take a user survey, more people than not would be in favor of lots of personalizable features. But then, when GUIs were first coming in, a majority of users said that they'd never want to use one. It is also important to recognize that users will customize an interface in such a way that it appeals to their subjective judgment. As has been observed in a number of experiments, *an interface that optimizes productivity is not necessarily an interface that optimizes subjective ratings.* (For an example, see Tullis 1984, p. 137.)

The central point of this issue is that if we are competent user interface designers and can make our interfaces nearly optimal, personalizations can only make the interface worse. Therefore, we must be sparing and deliberate in offering user customizations. If a user can, by a few judicious choices, really improve the interface, we probably have done a poor job.

On the other hand, if a program's interface is as dismal—to voice an opinion—as that of Microsoft Word 97/98, the situation is reversed. Almost any change the user makes is an improvement, to exaggerate only slightly. However, Word's interface is not the kind of goal toward which we should be striving.

Modes that vanish after a single use cause fewer errors than do those that persist, if for no other reason than that they have less time to cause havoc. In the Vellum cursor example, fewer user errors would result if the cursor, after performing its trace function, reverted in form and function to its normal state. If you use a temporary mode immediately after setting it, the fact that you set the mode may not have evaporated from your short-term memory, and you will not make a mode error; you may even incorporate setting the mode as part of the gesture that carries out the command, which makes the situation completely nonmodal for you. However, if you set the interface mode for a command, and, prior to using the command, are delayed or distracted, you are likely to make a mode error.

To avoid a mode, the Canon Cat was designed without a power switch.[4] The reason was that products react to gestures differently, depending on whether they are on or off, and a power switch therefore introduces a mode. To save energy, the Cat went into a low-powered sleep state if it was not used for five minutes. To make sure that sleep was not a mode, any user action or incoming message turned on the Cat without perceptible delay; furthermore, the user action that turned it on was not lost but took effect just as if the machine had not been in the sleep state.

In many systems, computers come out of sleep mode at the touch of a key, but the keystroke that turns them on—and any keystrokes made subsequently but before the system is fully awake—are lost. Not losing any keystrokes turned out to be an elegant feature. For example, if you had a sudden inspiration or had to take a note during a telephone call, you could just start typing without worrying about the state of the Cat or even looking at the display. It is a characteristic of modelessness that, once you become habituated, you do not have to think or plan before you act; your attention stays on the content of your work. (If you happened to put your note in the midst of something else, you'd just select the note and move it after you were done memorializing your inspiration or finishing your telephone call.)

Occasionally, designers claim that modes are necessary because the number of desired software functions exceeds the number of gestures that a user can make with a keyboard and a graphical input device, making gesture reuse necessary. However, display-based commands, such as menus, and typed multiple-character commands, as in command line–driven systems, offer an unlimited number of commands and thus avoid that difficulty. (How you make sure that the various commands in a command line system are visible, rather than having to be memorized, will be discussed later.)

The bottom line on modes is this: *If you design a modal interface, users will make mode errors except when the value of the state that is controlled by the mode is the user's locus of attention and is visible to the user or is in the user's short-term memory. The burden is on the designer to demonstrate that a mode is being used under the appropriate condition or that the advantages of a particular modal design outweigh its unavoidable disadvantages.* It is always safe to avoid interface designs that have modes.

---

4. I was startled to find a power switch on the back of the Cat when the first units came back from Japan. I pointed out that the design specifications said that there was to be no power switch, and I had gone over the rationale for this repeatedly with the engineers. "We thought the specifications had an error," I was told.

Buttons that Change in the Night

Some aircraft have pushbuttons that contain a dot-matrix display used to change the legend that appears on the button. This change is done under the aircraft's computer's control so that the buttons can be relabeled on the fly. The advertising copy for one brand of these buttons made some human factors claims. For one thing, fewer buttons would be needed in the limited cockpit area. For another, avionics and other system changes could be effected without having to rewire the cockpit.

From the perspective of cognitive science, good labels enhance visibility. Assuming that the person choosing the labels bothers to test the wording, to make sure that the chosen labels are effective and unconfusing, it is, on the face of it, a good idea.

But more careful thinking reveals a number of potential pitfalls to the product: All legended buttons get obscured—by the very finger you are using—just as you are about to press them, obviating a last-second check that you are pressing the right button. But this is a minor problem, for normally you have looked at the legend before you press the button—if you have looked at all. (I am certainly not looking at the legends on the keycaps of my keyboard as I type this.)

Then there is the deeper problem of the disappearing button. You want to enable, say, the manual controls on the cabin air conditioner. There is, you recall, a button labeled MANUAL AIR, but it is nowhere to be found. The reason is that it now says COMM BACKUP. You have to figure out how to get that button to light up the way it did before. Perhaps a switch or a button somewhere else changes the legends on the buttons. Or perhaps the legends are context-sensitive and won't allow you to change the air conditioning right now. Whatever the cause, when a button has variable legends, the button you want will perhaps have vanished because the system is in a different mode.

But the gravest problem has to do with habituation. Imagine the experienced pilot reaching up and pressing a button. The radio's not working; something's wrong. A communications problem: Time to enable COMM BACKUP. She jabs out expertly at the COMM BACKUP button. Click! Too bad. The copilot was

adjusting the cabin temperature in response to a passenger complaint, and the buttons are now in climate-control mode. The pilot has just achieved complete manual control over the air conditioning system.

Variable-legend buttons could be useful, for instance, if several people use the same console, each using it in such a way that the buttons do not change meaning for any one user. But this is a situation seldom encountered. Soft keys, whereby a display has changable legends that affect on-screen buttons or buttons adjacent to the display, have the same liability as do variable-legend buttons. A similar problem exists with the use of function buttons, labeled F1 through F12 on many computers. If their functions are unchanging, the labeling is unmnemonic. If their functions change, you cannot use them automatically. In either case, they are a poor design.

Are Fewer Buttons Better?



Figure 3.4. Fluke scopemeter: powerful and rugged, zillions of functions, a paucity of keys, but difficult to learn and sometimes slow to get to the function you need. On the other hand, its color-coded leads eliminate a traditional source of confusion about which lead does what. (See color insert.)

A portable Fluke[5] oscilloscope I recently tested (Figure 3.4) is incredibly sturdy and capable. Its 35 buttons control a bounteous multitude of modes and functions. The device was difficult to learn and has proved impossible to operate quickly when trying to work with many of its features.

5. I am grateful to John Fluke Manufacturing Co. for providing the sample unit.

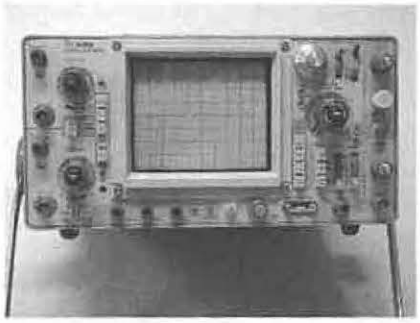


Figure 3.5. The Tektronix oscilloscope has a myriad settings, but although initially intimidating, it is quick and easy to operate. All knobs that can uncalibrate the unit are in red. (See color insert.)

A Tektronix oscilloscope (see Figure 3.5) with its myriad settings that seem so daunting at first glance, is easier both to teach and to use. The number of switches and knobs is about the same as on the digital scope, but many of the knobs have multiple settings that you can access immediately, and the settings are marked on the face of the product, so that they are visible rather than hidden in a menu that must be summoned to be seen. Fluke's product offers much greater functionality but at the cost of making some tasks that were previously easy more difficult. The rotating knobs are each far more powerful than each of the Fluke's buttons. Fluke also went further than necessary in creating submenus of excessive depth.

Operation of the oscilloscopes parallels the relative ease of use of two radios I own. My truck radio (Figure 3.6) has 18 preset stations, arranged in three groups of six. One button cycles through the three groups, and a linear arrangement of six buttons chooses the station within each group. An LCD panel shows the group number (1 through 3). Seven buttons suffice to select among 24 stations.

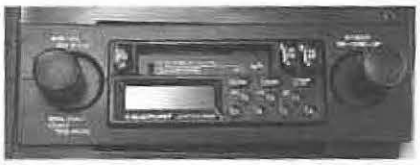


Figure 3.6. The radio in my truck is difficult enough to use as a radio, and I find it impossible to remember how to set the clock.

My other radio is the electronically superb Sony 2010 (see Figure 3.7), which has 32 presets on 32 identical buttons in four rows of eight buttons each. At first sight, this radio looks more complex, but as interviews with owners and long personal use have proved to me, this design makes it easier to find a preset station, especially in the

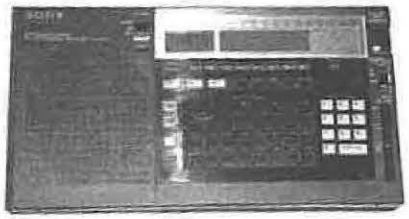


Figure 3.7. The Sony 2010 short-wave radio has many buttons but a fast, readily learned operation. (See color insert.)

dark; just count down and over to the button you want and tap it. The Sony 2010 stores all parameters of the station, such as whether the station is AM, short-wave, FM, or aircraft band, and any other settings you have chosen. Tap one of the 32 buttons, and everything needed to play that station is set automatically and at once. I put the most frequently accessed stations near the upper-left corner of the array.

The truck radio sacrifices usability to obtaining a low button count and small panel size. The worst part of the design is that you are forced to take your eyes from the road to know which set of presets you are using, or you can be distracted fishing for the station you want. (If you are switching from AM to FM or vice versa, you must find and tap two buttons; with the Sony, it is always one tap to select any preset.)

It is only slightly hyperbolic to suggest that it should be illegal to have a toggling, modal interface in a motor vehicle. Clearly, my truck radio has an unsatisfactory interface for selecting stations; even less satisfactory is its invisible method for setting the clock. I have to look it up every time: Press and hold the DISP button for 2 seconds; let it go. Turn the tuning knob counterclockwise to advance hours; turn it clockwise to advance minutes; press the DISP button to finish setting the clock. (The DISP button is the volume-control knob.) On the plus side, at least my radios have real volume-control knobs, unlike the pair of volume-control buttons on some more modern equipment. Volume-control buttons suffer from making it take too long to get to the desired loudness and from the fact that the current setting is neither visible to the eye nor accessible to the sense of touch. In a recent user survey that I conducted, every subject (n = 55) preferred a knob in this role.

Fewer buttons and simpler-looking controls are not always better. The lesson applies to screen design as well as to instruments and appliances.

3-2-3 Modes and Quasimodes

Using the Caps Lock key to type uppercase letters and holding the Shift key to the same effect are significantly different. The first case establishes a mode; the second case does not. A set of experiments at the University of Toronto confirmed that the act of holding down a key, pressing a foot pedal, and any other form of physically holding an interface in a certain state does not induce mode errors (Sellen, Kurtenbach, and Buxton 1992). Other studies revealed the neurophysiological roots of this phenomenon: Most of our nervous system operates such that a constant stimulus yields signals that, in time, decrease in their ability to capture our attention. This decrease continues until our cognitive system receives no signal at all. However, the signals that report back to us whether our muscles are actively producing a force do not fade.

Activating and holding a control while performing another user action has been referred to as a *spring-loaded mode* and a *spring-locked mode* (Johnson and Engelbeck 1989). But this terminology is inappropriate because no physical spring may be involved, the key or button is not locked, and holding a control does not cause mode errors. The phrase *user-maintained mode* (Sellen, Kurtenbach, and Buxton 1992) for this action is accurate but makes it difficult to frame a corresponding adjective. I have come to use the term *quasimode* and its adjectival form, *quasimodal*, to denote modes that are maintained kinesthetically. (See Figure 3.8.)

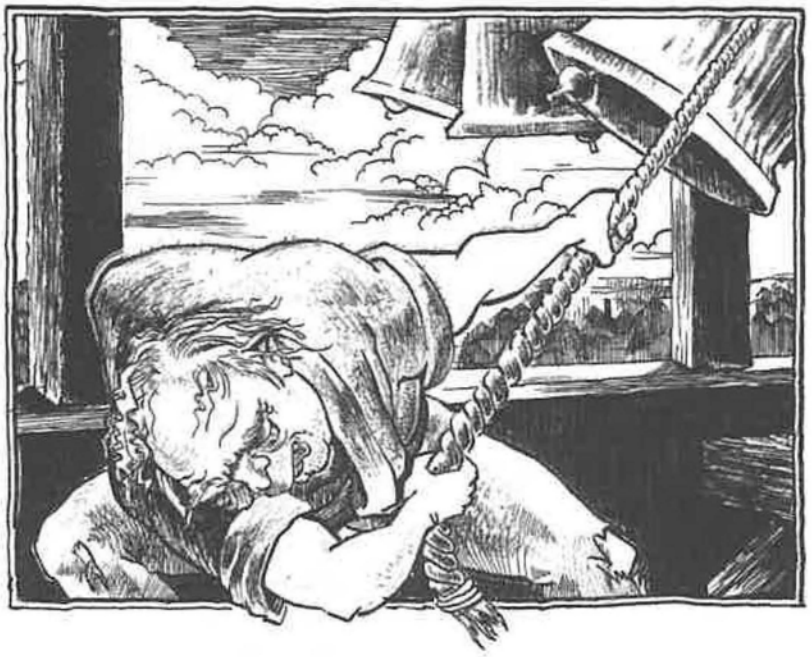


Figure 3.8. The hunchback of Notre Dame.

Quasimodes are very effective in eliminating modes (Raskin 1989); however, excessive use of quasimodes can lead to absurd interface conventions that require the user to remember dozens of commands, such as *Control*↓ *Alt*↓ *Shift*↓ *Esc*↓ *q*↑↑↑↑. The limit for the number of effective quasimodes is probably between four and seven. But one quasimode can solve a multitude of varied problems (see Section 5-4).

A typical problem that is readily solved by the use of quasimodes occurs when an interface presents you with a set of choices, such as the Macintosh's pull-down menus. In this application of quasimodes, you press and hold down the graphical input device button when, on the name of the menu, the other choices appear for as long as you are holding down the button. You move the cursor to the desired item and release on that item to make your menu choice.

Another use of quasimodes is to cycle through a set of options. So long as the cycle starts on the same element in the set and proceeds in the same order, and so long as there are not too many elements, a fixed number of taps chooses the option. For example, in the design of the Canon Cat, the user could change a selection to any of four paragraph styles: left-aligned, centered, right-aligned, and justified. Paragraph styles were chosen by repeatedly tapping the key that had on its front the legend

¶ STYLE

To access a key with a front legend on the Cat, you held down a special Shift key that was marked on its top Use Front (Figure 3.9). In general, the Use Front key allowed you to execute the function indicated by the legend on



Figure 3.9. Canon Cat's Use Front key and some of the command keys. The words USE FRONT are light blue, as are the legends on the fronts of the keys that are enabled by the Use Front key. (See color insert.)

the front of a key. Because paragraph style was a quasimode (you had to hold down Use Front while tapping the Paragraph Style key), the Cat knew how many times you had tapped it. As a consequence, you soon learned that one tap left-aligned the selection, two taps right-aligned the selection, and so forth. Had the software cycled from wherever you left off, as many interfaces do, there would have been no way of making your use of the paragraph style command habitual; you always would have had to look at the display to know what was happening. Note that a paragraph style button that did not require holding down Use Front would not have worked as well; it was the establishment of the quasimode, when you pressed the Use Front key, that allowed the system to know when to start counting.

A habituating feature is often one that can be operated successfully by a blind user. Following the principles in this book will often result in interface methods that can be used by the blind. We are all blind—in a very real sense—to the world outside of our locus of attention.

Some menus or palettes take the most-used or the most recently used item and position it at the top of the menu or in the palette: a simple example of an *adaptive menu* or an *adaptive palette*. A menu or a palette is made adaptive on the assumption that having the favored item available, without having to pull it down, will speed your use of the system. It is useful to compare two approaches: The first approach is to *remove* the chosen item from the list that has been pulled out and place that item in the main palette or menu; the second approach is to *duplicate* the chosen item into the main palette or menu (Figure 3.10).

It might seem that the first strategy is better. For one thing, there is one less choice in the list that has been pulled out. It also takes less screen space. But you also have to stop and check whether the tool you want is in the main palette or in the pulldown, especially if it has been more than a few seconds since you last used the tool. But it is the second alternative, whereby you can adopt the simple strategy of always using the *n*th pulldown for a desired task, that is usually faster. It is certainly easier from a cognitive standpoint. If you remember that a certain option is now in the main menu, you save having to pull down the complete list. But if you don't remember, you can use the pulldown as you always do.

Vellum (Ashlar 1995) is an example of a product interface with adaptive palettes done correctly (Figure 3.10). In Vellum, the palette is adaptive, *and* each tool can also be found in its accustomed place. However, an experienced user might ignore the adaptive feature, going automatically to the accustomed tool position. An additional aspect of providing multiple ways of invoking a feature will be discussed in Section 3-5.



Figure 3.10. Ashlar pull-out palette. Note the duplication of the first icon on the pulled-out group, so that the pullouts are always in the same order; their use can, therefore, become automatic.

It has been suggested that interfaces should adapt to accommodate the user's emotional state, and some products, such as Microsoft's Bob, attempt to tailor the interface to the personality of the user. It is not clear how an interface can adapt to the user in these ways without upsetting habituation. But even if we can, someday, reliably detect a user's emotional needs and use that information to improve the performance of an interface without interfering with our learned and automatic responses, our interfaces must still first satisfy the user's unchanging cognetic needs. The principles outlined in this book will still apply and will probably need to be applied prior to any adjustment for emotional needs.

Fundamentally, there are only two kinds of inputs you make to your computer or information appliance: creating content or controlling the system. The rule of thumb adopted here is this: *Quasimodes are reserved for control functions. Operations you perform when no quasimode is engaged create content.* It is this way rather than the other way around because it is more difficult to operate a system while a quasimode-establishing button is being held, and it is hoped that users will spend more time creating and working with content than in using commands.

3-3 Noun-Verb versus Verb-Noun Constructions

A large class of commands involve applying an *action* to an *object*. In operating a word processor, for example, you might take a paragraph and change its typeface; in this case, the object is the paragraph, and the action is the selection of a new font. The interface can allow you to sequence the operations in two ways. You choose either (1) the *verb* (change font) first and then select the *noun* (the paragraph) to which the verb should apply or (2) the noun first and then apply the verb. At first glance, it would seem that the situation is symmetrical and the order is of no importance, but in most interface designs, the situation is not symmetrical, and the order (either noun-verb or verb-noun)[6] makes a significant difference in usability.

Most interface guidelines correctly recommend noun-verb interaction (Apple 1987, Hewlett Packard 1987, IBM 1988, Microsoft 1995). A locus-of-attention analysis shows the benefits.

- *Error reduction.* Verb-noun style sets up a mode. Once you have chosen a command in this style, it will take effect on the next selection you make. If there is a delay or a distraction between issuing the command and making the selection, the ensuing action may be surprising when you next make a selection. With noun-verb construction, commands are executed when issued, when your locus of attention is the command.

- *Speed.* You do not have to shift your attention away from your content—which is what triggered the need to perform an operation—to the command and then back to find your place in the content again to make the selection. With noun-verb construction, you make the selection—your locus of attention—and then switch your attention to the command. There is one less change of locus of attention.

6. The terminology *object-action* versus *action-object* is also used.

- *Simplicity and reversibility.* In the verb-noun paradigm, you need to have an escape or a cancel feature associated with the command; if you issue a command and then decide against it, you are in a mode where the system expects you to make a selection, so a mechanism must be provided so that you can signal the system that you do not want to make a selection, you want to issue another command. In noun-verb construction, if you decide to change your selection, you simply make another selection. No Cancel button or cancel method is necessary.

Every interface guideline I have seen that advocates noun-verb construction also permits the verb-noun style of commands. The Microsoft manual states that verb-noun style is necessary for palettes, such as those for the various brush styles in paint programs (Microsoft 1995). This is not strictly true. A pure noun-verb model is possible: You draw in a default set of attributes, such as a thin black line, and then apply color, width, texture, or whatever via commands. However, we really want to see the full effect of each stroke as we make it—with all of its attributes in full flower.

The conventional method, whereby you first select the attributes from one or more palettes—much as you dip a real brush into this or that paint and then make your mark—does lead to the mode errors we have come to expect in such situations. It also makes sense to let the mode you have chosen persist until you deliberately change it. Therefore, it will happen that you start to draw and find unexpected attributes appearing. Fortunately, the appearance of what you are drawing is your locus of attention; if the software is humane, you will be able to immediately undo the offending mark, change the attributes to those desired, and continue working. The mode errors are annoying, and it would be gratifying to find a noun-verb or other nonmodal method that would be effective in this situation, but to the best of my knowledge, nobody has as yet found a solution to this particular problem.

In general, the noun-verb paradigm is preferred. Verb-noun methods should be limited to palette selections intended for immediate use.

Case Study of a Noun-Verb Fix to a Verb-Noun Problem

Workers in various departments of a multinational corporation used a computer system to requisition products. This example illustrates one way to change a seemingly natural verb-noun situation to a noun-verb one. In practice, this change eliminated the errors that

the initial design had caused and also speeded the requisitioning process.

The original process had three steps.

1. Choose the department from which the order was originating. This consisted of clicking on a check box alongside a department name. One of the check boxes was prechecked as a default. The default was the department in which the computer was situated.

2. Choose the desired items from a scrollable list of items, each item having an associated text box into which you could type the desired quantity for each item.

3. Click on one of two buttons at the bottom of the display: either Cancel Order or Confirm Order.

The first problem was that users failed to select their department when they were ordering from other than their usual computer station. A solution that had been suggested by the company's designers was to provide no default, forcing the user to select a department first, every time. Then, they pointed out, the only possible error would be to choose the wrong department. This solution, however, is annoying to users, who usually work at a specific station and thus might be irritated at having to supply information already available to the system.

The problem was a disguised verb-noun situation: You first entered what you wanted to do (deliver these items to the department) and then chose what you wanted to have delivered. When you entered the requisitioning screen, the item or items you needed were, or recently had been, at your locus of attention.

Part of the solution was to put the checklist of items at the top of the requisition display. Thus, the user would start by selecting the desired items. The location list, previously unlabeled, was given a descriptive label: "To which department do you want the requisitioned items delivered?"

If the system could determine—from the user's log-on information—the user's department, a new item was automatically added to the top of the list of departments. It was labeled "My Department" and, like the other items on the list, had a check box.

The rest of the list contained all of the departments in alphabetical order and a check box for each. When the user placed a check in one of the boxes, a message appeared: "Your items have been requisitioned." The message automatically disappeared at the user's next action.

With the redesigned interface, choosing a department represented an action—a verb—namely, to cause the previously selected items to be sent. Psychologically, it was now an act of closure rather than a preliminary step. Also, note that at this point, the requisition form had been filled in; the user's attention was no longer on the desired items but rather on having them delivered. The work flow now accorded with the path that the user's locus of attention would ordinarily follow.

No default location was needed, because the interface supplied psychological closure by indicating that the user had finished with choosing items and wanted them sent. The initial design also had required a confirmatory click, whereas the new design did not impose an additional click on the user; the key-click count had typically decreased by one. With the new design, users could still form a habit of choosing their usual location from the location list and, on the rare occasions this option was desired, might sometimes forget to specify that they wanted the order sent elsewhere. However, even with this potential fault, the new design was a major improvement over the previous one.

3-4 Visibility and Affordances

On a clear disk you can seek forever.

—Source unknown

Whether a product is a handheld two-way radio or a computer's desktop, it is not always clear what functions are available, what they do, or how they are accessed. You should be able to use your senses to easily discover both what abilities are available and how they are to be operated.

An interface feature is **visible** if it either is currently accessible to a human sense organ—usually the eyes, although this discussion applies also to other sensory modalities—or was so recently perceived that it has not yet

faded from short-term memory. If a feature is not visible, we say that it is **invisible**. For an interface to work well, "[j]ust the right things have to be visible: to indicate what parts operate and how, to indicate how the user is to interact with the device. Visibility indicates the mapping between intended actions and actual operations" (Norman 1988, p. 8).[7] If an interface forces you to memorize the fact that a feature exists, that feature is invisible. If you are forced to root around in the interface until, by luck and perseverance, you reach a sequence of actions that activates a feature, such a feature is not visible. If you have to use a help system to discover how to perform an operation, the methods for invoking that operation are invisible. Many computer games are, essentially, undocumented interfaces in which the controls, or their mapping to desired effects, are invisible. Add documentation and these games become trivial. *Most people do not want to play games when they are trying to get work done. It is up to the designer of an interface to make every feature of a product visible.*

· In designing to accommodate visibility, each function and the method of operating it would be apparent—to most people in the culture for which it is intended—by merely looking at it. A control that has this attribute has come to be called an **affordance** (Norman 1998, p. 123). "Affordances provide strong clues to the operations of things. . . . Knobs are for turning. Slots are for inserting things into. Balls are for throwing or bouncing" (Norman 1988, p. 9). If you, as a designer, put a knob such as is used in volume controls on a product, people will attempt to turn that knob. Put on something that looks like a pushbutton, and people will push it. Whether a feature is or is not an affordance depends on the experience and the background of the people who will be using the product and the context in which the feature appears.

In analyzing interfaces, we should always ask how the user knows that an action is possible, and we should always require that each visible feature provide a recognizable affordance. Icons are often considered the epitome of visible affordances, but they do not always serve this function, as we discuss in Section 6-3.

Visibility is more than just detectable existence. An object may be visible in the sense that it is there, but it might be too small to be noticed, or it may have too little contrast to be readily distinguished from the background. Optimizing the perceptual qualities of an interface is an important ergonomic consideration, but the concern here is with the cognitive properties of interfaces.

---

7. This book, *The Psychology of Everyday Things,* is not to be missed.

Affordance Avoidance: BART

It is the interface to its ticket machines that brings Bay Area Rapid Transit (BART) into this book. The machines at a local station are the result of the transit authority's ignorance of 25 years of experience with the old ticket-dispensing robots. As I stand in line at one of the two machines not presently out of service, I watch person after person baffled by the machine's layout. Although experienced with the machines myself, I still fumble at the unnatural sequence of operations that they require.

The obvious way to start the interaction, and what nearly everybody whom I've observed tries at first, is to put in money or an old ticket, which is a money-equivalent. That's how vending machines work: Put in money, make choice, get stuff. It is a cultural paradigm.

The BART ticket-vending machines must have been designed on another planet. First, you choose between BART and BART-MUNI buttons. (A pamphlet, available nowhere in the vicinity, explains the subtle difference.) These buttons are at the top of the wall of the machine face: a reasonable place to start but an unreasonable transaction with which to start. The slots into which you then put your ticket (to the left), coins (in the middle), or bills (lower right) are widely separated instead of being clustered. The LCD display is up high at top center, where short people cannot easily read it; in any case, it is far enough from the rest of the interface that many people do not notice it at all. Maybe it's good they do not see it, in that its response time is so slow that people who do watch it either repeat their actions in hope of forcing a response or stand there wondering whether the machine is broken, which it often is.

Then there's the "correction" button, labeled as though it is to be used in case you've made a mistake that has to be corrected. In fact, you do not use this button for correcting a mistake: You use it if you want the ticket to be issued for less than the total amount of money that you have put in. That is, it simply tells the machine you want some change back.

Another indication that the new machines are badly designed is the bright, colorful, large numbers "1, 2, 3, 4" and the big arrows presumably placed to guide you. If the front of the machine were well laid out, you wouldn't need crutches. The numbers do not help

tremendously anyway, because you must remember what step you are executing (say, step 2) while you put in your money ($5), knowing that you only want to get a $3 ticket so you must hit the correction button in the next step (step 3), keeping in mind that you'll want to count your change ($2) that will come in the form of quarters (eight of them). Hurry! You can hear the 7:06 train coming! What step did you say you were on? When you find a computer interface that requires bright colors and myriad explanatory legends to guide you, you can guess that the design has gone astray.

Here is one way in which the BART machines could work. Remember, however, that were you consulting to BART, you would request user testing of your proposed design to verify that people will use it as you have predicted.

On the left—because English and Spanish are written from left to right—is the slot for old tickets, a horizontal slot for bills, and a small vertical slot for coins, grouped together and well labeled. The display—located at average eye level—reads: "Put in money OR your old BART ticket OR both." As you put in each, the total value accrued appears *immediately* in the LCD display with this message:

> You now have put in $4.55.
>
> If you want a ticket for more than $4.55, put in more money or another old ticket
>
> If you want a ticket for less than $4.55 press:
>
> Change Ticket Amount
>
> If you want a ticket for $4.55, press one of these buttons:
>
> Issue BART Ticket
>
> Issue MUNI-BART Ticket

If the user selects "Change Ticket Amount," he is asked to type in the amount for which he wants the ticket issued and to press the large ENTER button that is under the numeric key pad. The user is returned to the display that shows the current total and asks him to increase it, decrease it, or OK it. When the ticket is issued, any change due the user is returned. For the experienced traveler with the right amount of money in hand, this interface reduces six actions to these two: put in money and then press an Issue-Ticket button.

3-5 Monotony

Man is an over-complicated organism. If he is doomed to extinction he will die out for want of simplicity.

—Ezra Pound

Designers of interfaces often present users with a choice of methods. For example, both a menu item and a keyboard shortcut may execute the same command. In most word processors, you can move a contiguous portion of the text by either (1) the three steps of selecting, cutting, and pasting or (2) the two steps of selecting and dragging. Also, the selection process itself can be invoked in more than one way. The user has a smorgasbord of methods.

One justification for having multiple methods to do a given task is that some users prefer one method and other users prefer a different method. For example, a novice user might find menus easier to learn, although an expert user might rather keep her hands on the keyboard and issue commands by typing them (see Section 3-6). Another justification is that one method—for example, selecting, cutting, and pasting—is useful between distant portions of a document, and the other method—selecting and dragging—is effective only when the source and destination are both visible on the display. Another reason for a plurality of methods is that each of them is sanctioned by custom, and the developers felt it wise to accommodate as many previously developed skills as possible.

This last reason, called *backward compatibility*,[8] is the weakest and can lead to absurd interfaces that are an accumulation of incompatible methods. When I was grounded and waiting for the weather to improve during a long flight by commercial jet, I went to the cockpit, where I studied an autopilot that had no fewer than five ways of entering coordinates and similar numbers of methods for performing most of its other functions. When I asked the pilot for the rationale, she said that it had been built to functionally resemble, as closely as possible, autopilots from other aircraft that pilots might have trained on and thus avoid the need for expensive retraining. The question was whether the tactic was successful and whether the old units had been duplicated exactly. As she explained, the pilots not only had to learn the small but annoying differences between their old system and the emulation of their old system on the new autopilot but also were required to learn the four other ways of using the autopilot. A pilot must be familiar with *every* aspect of *every* piece of equipment in the cockpit; besides, many of the newer features of the autopilot were

8. I usually suggest that if the word *compatibility* is eliminated from this phrase, its true meaning becomes apparent.

available in only some of the emulations; those of the earlier autopilots did not have those features because the autopilots being copied had not had them.

The tactic of portmanteau interface design, or just toss every method you can think of into the bag, increased training time, produced a harder-to-use autopilot, and, as the pilot noted, increased cockpit confusion and opportunities for errors. She did not say so, but it also probably increased the cost and complexity of the product, the manuals, and the cost of maintenance. The same is true for any interface that is a conglomeration of disparate design philosophies, accumulated over time; this includes the Macintosh and Windows.

I use the term *monotonous*, tongue partially in cheek, to describe an interface having only one way to accomplish a task (See Appendix B; Alzofon and Raskin 1985, p. 95). **Monotony** is the dual of modelessness in an interface. In a modeless interface, a given user gesture has one and only one result: Gesture *g* always results in action *a*. However, there is nothing to prevent a second gesture, *h*, from also resulting in action *a*. A monotonous interface is one in which any desired result has only one means by which it may be invoked: Action *a* is invoked by gesture *g* and in no other way. An interface that is completely modeless and monotonous has a one-to-one correspondence between cause (commands) and effect (actions). The more monotony an interface has for a given task space, the easier it is for the user to develop automaticity, which, after all, is fostered by not having to make decisions about what method to use.

Another cause of nonmonotonous design is managerial indecision. Given two or more choices of interface methods and no rational method of choosing one above the others, I have seen the problem "solved" by expediently putting in *all* of the proposed methods. This is rationalized as giving the user a choice, as if the user were an interface expert and will make the most productive choice.

A common myth, discussed in more detail in Section 3-6, is that beginners and expert users of a system require radically different interfaces. Designers speak in terms of "the tradeoffs between ease of learning and speed of execution in a system" (Card, Moran, and Newell 1983, p. 419). This may be true of particular interface designs, but I have seen no demonstration that it is necessarily a property of all interface designs; in particular, it is not a property of the kinds of designs discussed in this book. *Present desktop GUIs are a compound of at least two distinct interfaces, a relatively visible and learnable but time-consuming menu-based system and an incomplete keyboard-based collection of hard-to-learn and unmemorable shortcuts. Two wrongs do not make a right.*

When you have to choose among methods, your locus of attention is drawn from the task and temporarily becomes the decision itself. This is

a primary reason for choosing to design a monotonous system. If the conditions for making the decision are sufficiently clear and distinct, the path you take in each case can become habitual, and you have monotonized the situation. It still behooves the designer to seek an efficient monotonous solution to gain benefits, including ease of learning, simplicity of implementation, minimization of documentation, and lowered maintenance costs. These benefits are either free to the implementers or can be had for the relatively small one-time cost of careful design and testing. Monotony does not mean that the same content cannot be arrived at in many ways but that there should not be multiple gestures to invoke the same command.

Monotony happens spontaneously. Many users monotonize the interfaces they use by choosing one method and sticking to it, ignoring alternatives whatever the situation. Computer gurus who pride themselves on knowing every wrinkle of a system often decry such users as amateurs; nonetheless, the "amateurs" may be using the interface more efficiently than the gurus do. From the point of view of an implementer, such users are wasting features; from the point of view of the users, it is the implementers who are wasting resources.

I believe that an interface that is both modeless and, insofar as possible, monotonous—all other design features being of at least normal quality for a modern interface—would be extraordinarily pleasant to use. A user would be able to develop an unusually high degree of trust in his habits. The interface would, from these two properties alone, tend to fade from the user's consciousness, allowing him to give his full attention to the task at hand. The psychological effects of totally (or near totally) modeless and monotonous systems is an area of interface design ripe for experimental study.

If I am correct, the use of a product based on modelessness and monotony would soon become so habitual as to be nearly addictive, leading to a user population devoted to and loyal to the product. Its users would find moving to a competitor's product psychologically difficult. Unlike selling illicit drugs, marketing an addictive interface is legal, and the product is beneficial to its users; in another way, it is just like selling illicit drugs: extremely profitable.

3-6 Myth of the Beginner-Expert Dichotomy

We're humans first, beginners or experts second.

—Clifford Nass, CBC "Quirks and Quarks" radio program, 23 January 1994

Psychologist Clifford Nass's point is similar to one that this book makes: Our interface designs must begin by accommodating universal human frailties

and exploiting universal human strengths. We must make sure that every detail of an interface matches both our cognitive capabilities and the demands of the task (not that those two objectives exhaust our concerns). His comment also reflects the common assumption that users can be grouped into two classes: beginners and experts, perhaps with a few temporarily in transition. This dichotomy is invalid. As a user of a complex system, you are neither a beginner nor an expert, and you cannot be placed on a single continuum between these two poles. You independently know or do not know each feature or each related set of features that work similarly to one another. You may know how to use many commands and features of a software package; you may even work with the package professionally, and people may seek your advice on using it. Yet you may not know how to use or even know about the existence of certain other commands or even whole categories of commands in that same package. For example, a user of a photo-processing application who produces only online images may never need, or even learn of, that same application's facility for doing color separations, a feature needed primarily by commercial printers.

Interface designers have tried various approaches to accommodate the premise that users can be separated into beginners and experts. Because this premise is false, the approaches have failed. Adaptive systems that shift automatically from beginner mode to expert mode when they judge that your competence has reached a certain level are a good example. If you are using such a system in beginner mode and it suddenly shifts into expert mode, you will find yourself on unfamiliar ground, at least with regard to a portion of the system. A system that shifts piecemeal, feature by feature, is no better. It will feel unstable and unsettling, because the habits that you were developing as a novice yesterday become useless when the feature shifts into expert mode today.

One web-based program I studied promoted you to expert status after you had used it once successfully. The program put you back into beginner mode when you had not used it for six months. Any such arbitrary schedule may not accord with a user's personal rate of learning and memory decay. If a program that promoted you switches to beginner mode after too short a time, you will feel annoyed at being forced to use the tedious beginner method. If the program does not switch back to beginner mode in time, you will be faced with features that you have forgotten how to use. Given present technology, a system cannot know when you have forgotten how to use a given feature, so it cannot know when it should shift back to beginner mode. A program that quizzed you from time to time to assess your level of expertise would be obnoxious.

Most attempts to make interfaces adaptive are ill-advised; whenever a

system changes automatically, even if the change is as small as, say, a reordered set of items on a menu, your expectations are upset and your habituation is frustrated. (Microsoft features adaptive menus in its Windows 2000 operating system.[9]) On the other hand, there is no theory that tells us that the same fixed interface cannot work well over the full span of a person's experience with it, from novice to old timer. It seems best not to have to shift paradigms during your use of a product, and no elaborate analysis is needed to reveal the advantage in having to learn only one interface to a task.

It is easy to fall into the trap of designing different interfaces for different classes of users, because by doing so, you can make sweeping assumptions that simplify the design process. Few such assumptions are likely to be true of every user in any reasonably large class of users that you specify. The antidote is to view an interface not from the perspective of a class of users but rather through the eyes of an individual. Every person who uses software over a long period goes through a relatively brief period of learning for each feature or command and a far longer period of routine (and, we hope, automatic) use. We do need to design systems that are easy to learn and understand, but it is more important that we make sure that these systems can be efficiently used in the long run. The exceptions are applications that will be used only briefly, so that every user is a novice, and habituation is not an issue. One example of such an interface is that for a computer-driven kiosk at an exhibition.

The learning phase of working with a feature involves your conscious attention. Therefore, simplicity, clarity of function, and visibility are of great importance. The expert phase is predominantly characterized by unconscious use of the feature; such use is enhanced by such qualities as aptness to the task, modelessness, and monotony. These sets of requirements are not in conflict; therefore, *a well-designed and humane interface does not have to be split into beginner and expert subsystems.*

This is not to say that an interface must not be split on these lines. However, if you find yourself designing an interface and are tempted to provide "expert" shortcuts, consider whether you should instead redesign the existing method so that it satisfies the needs of all users with one mechanism.

---

9. Windows 2000 was a new product as this section was being written, and I was able to interview only a few users. A typical remark was, "Adaptive menus seemed like a cool idea, but the first time a menu changed on me, I found it upsetting. I don't like the idea any more."

FOUR

Quantification

The harmony of the world is made manifest in Form and Number, and the heart and soul and all the poetry of Natural Philosophy are embodied in the concept of mathematical beauty.

—*D'Arcy Wentworth Thompson,* On Growth and Form *(1917)*

A number of methods for analyzing interface details quantitatively are available. However, explicit directions on how to use them are rare. This chapter gives an easy-to-use treatment and fully worked-out examples of Card, Moran, and Newell's keystroke-level GOMS model, Raskin's measures of efficiency, Hick's Law, and Fitts' law.

4-1 Quantitative Analyses of Interfaces

He fingered and fingered the computer—it simply amazed Melrose that the machine supposed to take the pain out of all sorts of niggling jobs took more time to perform a simple one than it would have taken Bub to do by hand ten times over.

—*Martha Grimes,* The Stargazey *(a detective novel)*

Many qualitative methods and heuristics are useful for analyzing and understanding interface design. These methods form the majority of the

content of most books on the subject, including those cited in the references for Shneiderman, Norman, and Mayhew. For example, what an experienced interface designer can learn from passively observing a test of a new interface with a few subjects can be as valuable as what she can learn from any quantitative analysis. My concentration on quantitative methods is not meant to denigrate the importance of qualitative techniques but rather to help even the balance by emphasizing the numerical and empirically testable methods that are not yet widely used. Quantitative methods can often reduce argument to calculation; a further, and most important, benefit is that *understanding* why *the quantitative methods work guides us to understanding important aspects of how humans interact with machines.*

One of the best quantitative analyses of interface design is the classic model of goals, objects, methods, and selection rules (GOMS), which first gained attention in the 1980s (Card, Moran, and Newell 1983). GOMS modeling allows you to predict how long an experienced worker will take to perform a particular operation when using a given interface design. After discussing the GOMS model, I present quantitative methods for determining interface efficiency, cursor movement speed, and the time cost of decision making.

4-2 GOMS Keystroke-Level Model

The aim of exact science is to reduce the problems of nature to the determination of quantities by operations with numbers.

—*James Clerk Maxwell,* On Faraday's Lines of Force *(1856)*

I will introduce only the simplest—yet nonetheless valuable—aspect of the GOMS method: the keystroke-level model. We designers who know GOMS rarely use a detailed and formal analysis of an interface design, but that is due, in part, to our having absorbed the fundamentals of GOMS and of other quantitative methods such that our designs inherently incorporate GOMS teachings. We do bring formal analysis into play when choosing between two approaches to interface design in which small differences in speed can have significant economic or psychological effects. We can sometimes benefit from the impressive accuracy of the more complete GOMS models, such as critical-path method GOMS (CPM-GOMS) or a version called natural GOMS language (NGOMSL), which takes into account nonexpert behavior, such as learning times. We can, for example, predict how long it will take a user to execute a particular set of interface actions to within

an absolute error of less than 5 percent. In these advanced models, almost all predictions fall within 1 standard deviation of the measured times (Gray, John, and Atwood 1993, p. 278). In a field in which religious wars are waged over interface designs and in which gurus often have widely varying opinions, it is advantageous to have in your armamentarium quantitative, experimentally validated, and theoretically sound techniques. For a good overview and bibliography of the various GOMS models, including her own CPM-GOMS model, see John 1995.

4-2-1 Interface Timings

Numerical precision is the very soul of science.

—*D'Arcy Wentworth Thompson,* On Growth and Form *(1917)*

When they developed the GOMS model, its inventors observed that the time it takes the user-computer system to perform a task is the sum of the times it takes for the system to perform the serial elementary gestures that the task comprises. Although different users might have widely varying times, the researchers found that for many *comparative* analyses of tasks involving use of a keyboard and a graphical input device, you could use a set of typical times rather than measuring the times of individuals. By means of careful laboratory experiments, they developed a set of timings for different gestures. In giving the timings, I follow the original nomenclature, in which each of the times is designated by a one-letter mnemonic (Card, Moran, and Newell 1983):

$K = 0.2$ sec	Keying: The time it takes to tap a key on the keyboard
$P = 1.1$ sec	Pointing: The time it takes a user to point to a position on a display
$H = 0.4$ sec	Homing: The time it takes a user's hand to move from the keyboard to the GID or from the GID to the keyboard
$M = 1.35$ sec	Mentally preparing: The time it takes a user to prepare mentally for the next step
R	Responding: The time a user must wait for a computer to respond to input

In practice, these numbers vary widely; K can be 0.08 sec for a 135-wpm highly skilled typist, 0.2 sec for a more typical 55-wpm skilled typist,

0.28 sec for a 40-wpm average unskilled typist, or 1.2 sec for a beginning typist. Typing speed is not independent of what is being typed: It takes most people about 0.5 sec to type a random letter, given a set of randomly chosen letters to type. Typing messy codes—for example, e-mail addresses—takes most people about 0.75 sec per character. The value K includes time it takes the user to make corrections that he has caught immediately. Shift is counted as a separate keystroke.

The wide variability of each measure explains why we cannot use this simplified model to obtain absolute timings with any degree of certainty; by using the typical values, however, we usually obtain the correct *ranking* of the performance times of two interface designs. If you are evaluating complex interfaces that include overlapping time dependencies or if you must generate accurate absolute times, you should use the more complete models, which are not discussed in this book, such as CPM-GOMS.

Double Dysclicksia

The interface technique called double clicking, that is, tapping the GID button twice within a small time window and without any significant cursor movement between the taps, as an interface technique suffers from problems. You cannot always predict what objects on the display will or will not respond to a double click, and it is not always clear what will happen if there is a response. There is no indication on displayable items that double clicking is supposed to produce a response: The functionality is invisible. The way that double clicking is used in many current interfaces, the user must remember not only *which* items are double clickable but also how different classes of interface features respond to this action.

The first two burdens on the user could be at least partially alleviated by new screen conventions. The act of double clicking is, however, itself problematical. Double clicking requires operating a mouse button twice at the same location or at two locations in very close and, in most cases, within a short time, typically 500 msec. If the user clicks too slowly, the machine responds to two single clicks rather than to one double click. If the user jiggles the mouse excessively between clicks, the same error occurs. If the user taps the GID button twice in too short a time period, as when trying to select text within a word while working within certain word processors, the machine considers the two taps as a double click and selects the whole word.

A problem arises when the user is trying to select a graphical item that can be repositioned with the GID. Because the GID is likely to move when the user is pressing the GID buttons quickly, graphical applications, instead of reading a double click, may read a drag-and-drop and change the item's position. Similarly, to change the text in a text box, the user may find it necessary to reposition the accidentally moved box and to make the text edit originally intended.

Some of us are unaffected by dysclicksia: These lucky people never miss with the mouse; they single and double click with insouciance and panache, do not suffer from side effects of clicking, always remember what will and what will not respond to double clicking, and can shoot a flying bird with a .357-caliber revolver while driving along a twisty mountain road. But we can't assume that all users are so lucky. We must design for the dysclicksic user and remain aware of the problems inherent in using double clicks in an interface.[1]

The duration of the machine response time, *R*, can have an unexpected effect on user actions. If a user operates a control and nothing appears on the display for more than approximately 250 msec, she is likely to become uneasy, to try again, or to begin to wonder whether the system is failing.

We cannot build products that can complete any operation within human reaction time, but our interfaces can always, within that time, give feedback that the input has been received and recognized. Otherwise, user actions—often flailing at the keyboard, trying to get a response—during a delay can start the system off on unintended activities, causing further delay or damaging the user's content. For example, if you try to download a file while accessing America Online from a browser, such as Netscape's, there is often a long delay. No feedback lets you know that progress is being made; a small, static message far from the locus of attention says only that the computer is awaiting reply. After a few seconds, inexperienced users start clicking at buttons on the display, which stops the download—again without feedback.

It is important that interfaces provide feedback if delays are unavoidable; display a progress bar (Figure 4.1) that accurately reflects the time remaining. If you cannot predict how much time an operation will take, say so! Do not lie to or misinform users.

[1] The term *dysclicksia,* a disease for which the only permanent cure is good design, was coined by Pam Martin (personal communication 1997).

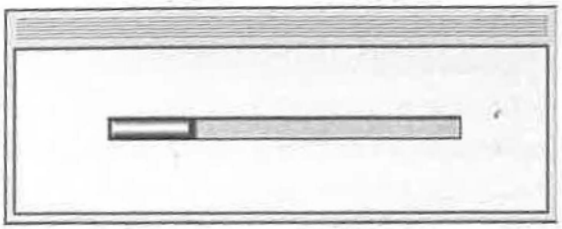


Figure 4.1. A progress bar. It is important that it represent time linearly. A textual statement of time remaining, if accurate, is also a humane feature when delays are unavoidable.

4-2-2 GOMS Calculations

We begin the calculation of the time it takes to perform a method, such as "move your hand from the graphical input device to the keyboard and type a letter," by listing the operations from the GOMS list of gestures (see Section 4-2-1) used in this method, in this case *H K*. Listing the gestures (*K*, *P*, and *H*) is the easy part of creating an instance of GOMS models. The more difficult part of developing an instance of a keystroke-level GOMS model is figuring out at what points the user will stop to perform an unconscious mental operation: the mental preparation (*M*) times. The basic rules—following the methods of Card, Moran, and Newell 1983, p. 265—for deciding where mental operations occur in a method are presented in Table 4.1. In Section 4-2-3, we look at how these rules are applied in practice.

In these rules, a **string** is a sequence of characters. A **delimiter** is a character that marks the beginning or the end of a meaningful string of text, such as a natural-language word or a telephone number. For example, spaces are the delimiters for most words; a period is the most common delimiter at the end of a sentence; parentheses delimit parenthetical remarks; and so on. The operators are *K*, *P*, and *H*. When a command needs information, such as when you use the command that sets the time for an alarm to go off and have to supply the time, the information you supply is an **argument** for that command.

4-2-3 GOMS Calculation Examples

An interface design usually begins with a task or a set of tasks that need to be accomplished. A statement of the task and the means available for implementing a solution are often formulated as a requirement or specification. In this example, the user is personified as Hal, a laboratory assistant.

TABLE 4.1. HEURISTICS FOR PLACING MENTAL OPERATORS

Rule 0 Initial insertion of candidate *M*s

Insert *M*s in front of all *K*s (keystrokes). Place *M*s in front of all *P*s (acts of pointing with the GID) that select commands, but do not place *M*s in front of any *P*s that point to arguments of those commands.

Rule 1 Deletion of anticipated *M*s

If an operator following an *M* is fully anticipated in an operator just previous to that *M*, then delete that *M*. For example, if you move the GID with the intent of tapping the GID button when you reach the target of your GID move, then you delete, by this rule, the *M* you inserted as a consequence of rule 0. In this case, *PMK* becomes *PK*.

Rule 2 Deletion of *M*s within cognitive units

If a string of *M K*s belongs to a cognitive unit, then delete all the *M*s but the first. A cognitive unit is a contiguous sequence of typed characters that form a command name or that is required as an argument to a command. For example, *Y*, *move*, *Helen of Troy*, or *4564.23* can be examples of cognitive units.

Rule 3 Deletion of *M*s before consecutive terminators

If a *K* is a redundant delimiter at the end of a cognitive unit, such as the delimiter of a command immediately following the delimiter of its argument, then delete the *M* in front of it.

Rule 4 Deletion of *M*s that are terminators of commands

If a *K* is a delimiter that follows a constant string—for example, a command name or any typed entity that is the same every time that you use it—then delete the *M* in front of it. (Adding the delimiter will have become habitual, and thus the delimiter will have become part of the string and not require a separate *M*.) But if the *K* is a delimiter for an argument string or any string that can vary, then keep the *M* in front of it.

Rule 5 Deletion of overlapped *M*s

Do not count any portion of an *M* that overlaps an *R*—a delay, with the user waiting for a response from the computer.

Requirement

Hal works at a computer, typing reports; he is occasionally interrupted by one or another of the researchers in the room, and is asked to convert a temperature reading from degrees Fahrenheit (F) or Celsius (C) to degrees C or F, respectively. For example, Hal might be asked, "Please convert 302.25 degrees from Fahrenheit to

Celsius." Hal must use the keyboard or GID to enter the temperature provided; voice or other input means are not available. Conversions from C to F and from F to C are approximately equally likely to be required. About 25 percent of the temperatures called out are negative, although the digits are unpredictable and equally distributed, and only 10 percent of the temperatures have integer values, such as 37 degrees. The numerical result must appear on the display; no other output means are available. Hal reads to the researcher the converted value from the screen. The input and the output must allow for at least ten digits on each side of the decimal point.

In designing an interface for a system that allows Hal to do his job, your goal is to minimize the time it takes Hal to do the conversion. Speed and accuracy must be maximized; screen real estate is not limited. The window, or area of the display in which the temperature conversion takes place, is already active and waiting for Hal's input via GID or keyboard. The way Hal interacts with the interface to return to his typing on the computer is not your concern; your job is finished as soon as the result is displayed.

In estimating the time it takes Hal to use the interface, assume an average of four typed characters in an entered temperature, including any decimal point and sign. Also assume—unrealistically, but for simplicity's sake—that Hal's typing is perfect; error detection and notification are not needed.

Now, I would like you to stop reading so that you can design an interface for this simple example. It will not take long to write down your proposed solution, along with sketches of the display that Hal will see; do not just think about this problem but rather write about it as well. (You will be tempted to read on without honoring my request. Please reconsider. The next few sections will make much more interesting reading if you have already tried to solve the problem yourself.) After designing your interface, read the two GOMS analyses that follow. Then you will be ready to analyze your own interface.

4-2-3-1 Hal's Interface: Solution 1, Dialog Box

The instructions in Figure 4.2 are reasonably clear; from them we can write down the method that Hal must use in terms of the gestures of the GOMS model. The GOMS representation is shown growing incrementally as each new gesture is added to the method.

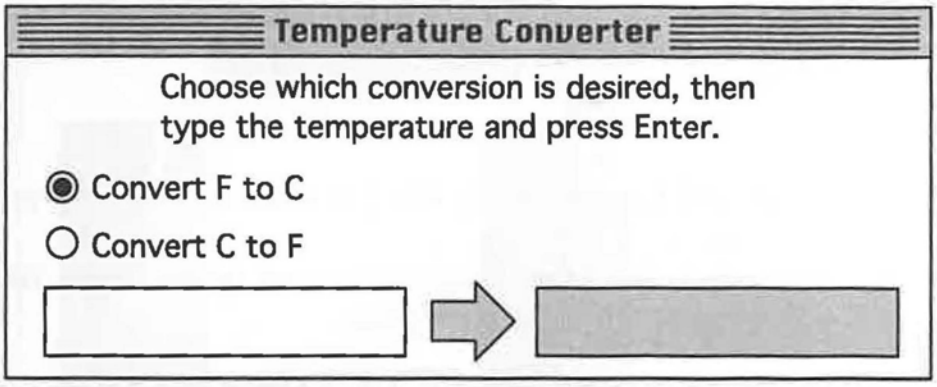


Figure 4.2. A dialog box solution with radio buttons.

- Move hand to the graphical input device:
  H
- Point to the desired radio button:
  HP
- Click on the radio button:
  HPK

Half of the time, the interface will already have the correct conversion chosen, and Hal will not need to click on the radio button. We consider first the case in which it is not the one already chosen.

- Move hands back to the keyboard:
  HPKH
- Type the four characters:
  HPKHKKKK
- Tap *Enter:*
  HPKHKKKKK

The keystroke for the tap of the Enter key completes the method portion of the analysis. Using rule 0, we add *M*s in front of all of the *K*s and *P*s except those *P*s that point to arguments, of which there are none in this example:

HMPMKHMKMKMKMKMK

Rule 1 tells us to change *P M K* to *P K* and to eliminate any other fully anticipated *M*s, of which there are none in this example. Rule 2 eliminates

*M*s in the middle of strings, such as in the string that represents the temperature. Applying these two rules leaves

$$HMPKHMKKKKMK$$

The *M* before the final *K* is required by rule 4. Rules 3 and 5 do not apply in this example.

The next step is to add the times represented by the letters. (Recall that $K = 0.2$, $P = 1.1$, $H = 0.4$, and $M = 1.35$):

$$H + M + P + K + H + M + K + K + K + K + M + K =$$

$$0.4 + 1.35 + 1.1 + 0.2 + 0.4 + 1.35 + 4 \star (0.2) + 1.35 + 0.2 = 7.15 \text{ seconds}$$

In the case in which the correct conversion is already selected, the method is

$$MKKKKMK$$

$$M + K + K + K + K + M + K = 3.7 \text{ sec}$$

By the requirements document, these two cases are equally likely. Thus, the average time it will take Hal to use this interface for one conversion task will be $(7.15 + 3.7) / 2 \approx 5.4$ seconds. But, because the two methods that Hal has to use are different, it will be difficult for him to operate this interface automatically. One of the open problems in the quantitative analysis of interfaces is how to estimate error rates from a given interface design.

Next, we explore a graphical interface that makes extensive use of a familiar metaphor.

4-2-3-2 Hal's Interface: Solution 2, GUI

The interface shown in Figure 4.3 uses realistic representations of thermometers to indicate temperature. Hal can lower or raise the pointer on each thermometer in Figure 4.3 by using the drag method with the GID. Hal indicates which conversion he wants by moving the arrow on either the Celsius or the Fahrenheit thermometer. He does not type any characters; he simply selects the temperature on the input thermometer. As he moves one of the pointers, the pointer on the other thermometer moves to the corresponding temperature. To set the required precision, Hal expands and contracts the scales; he can also change the range. When Hal changes the scale or the range on one thermometer, those on the other thermometer change automatically to cover approximately the same set of temperatures. Numeri-

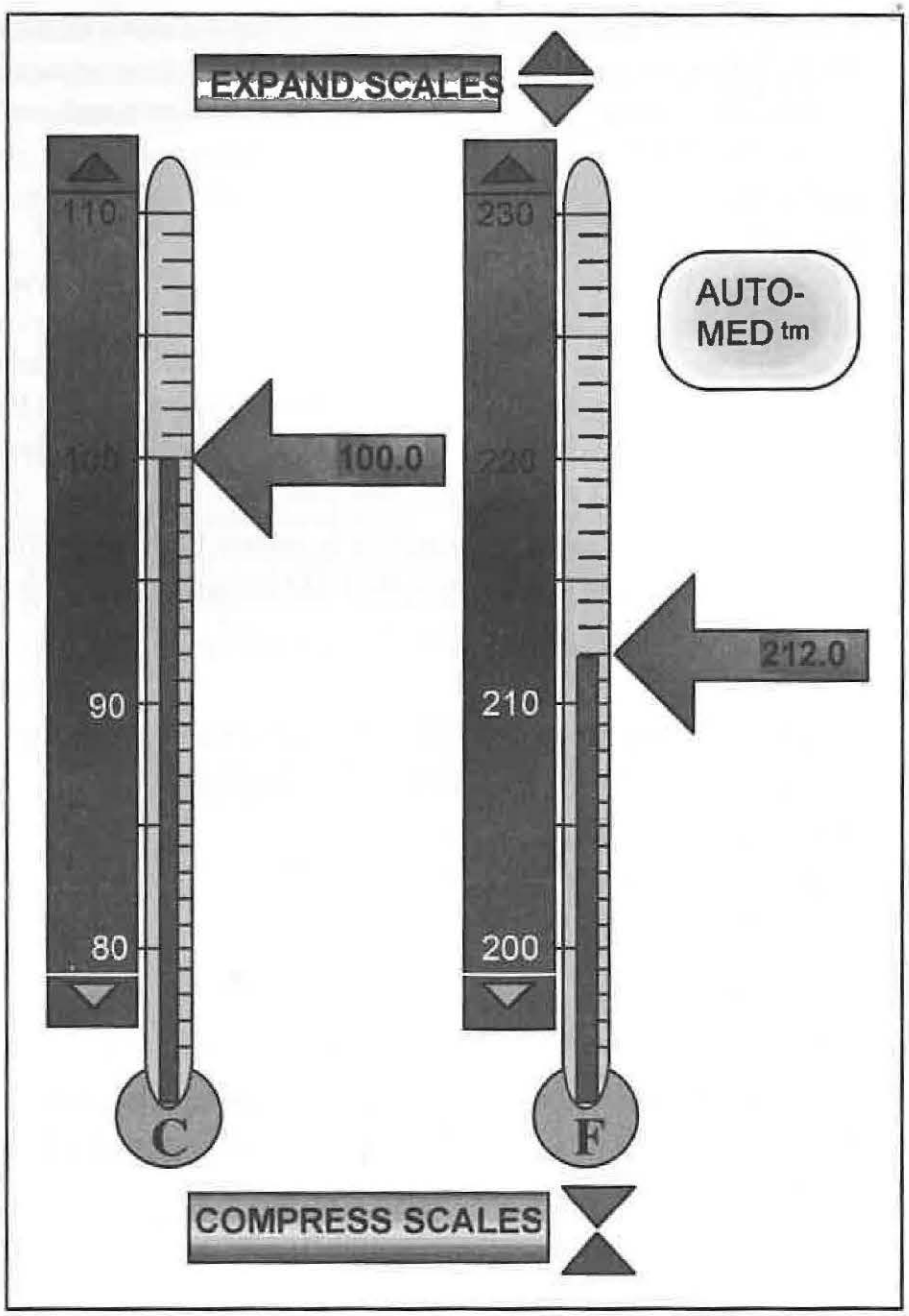


Figure 4.3. A GUI for Hal's interface. (See color insert.)

cal readouts are provided on the movable arrow. The temperature is indicated both numerically and with a bar, so Hal can use either the graphical or the character-based representations of the data to accommodate his learning style or personal preferences. The Auto-Med feature changes the ranges such that they are centered on 37 degrees Celsius and 98.6 degrees Fahrenheit, in case someone in the lab is working with human body temperatures; this feature is designed to save time.

Clicking on Expand Scales or Compress Scales increases or decreases by a factor of 10 the values at tick marks on the vertical thermometers. To get quickly to a far-distant temperature, Hal expands the scale and scrolls up or down until the desired range is in view, puts the arrow near the desired temperature, and then compresses the scale, adjusting the arrow if necessary, until the desired precision is attained.

A GOMS keystroke-level analysis of this graphical interface is complex because the method Hal uses depends on where the converter is presently set and what range and precision Hal needs. We look first at the fastest case, in which the range and the precision of the C or the F thermometer happen to be already set as Hal wants them to be. This analysis will give us the minimum time needed to use this interface.

- Write down the gestures Hal uses as he moves his hand to the GID and clicks and holds down the GID button on the desired arrow:

  HPK

- Continue listing gestures as Hal moves the arrow until it points to the correct value and then releases the GID button:

  $HPKPK$

- Place *M*s according to rule 0:

  $HMPMKMK$

- Eliminate two *M*s according to rule 1:

  $HMPKK$

There are no cognitive units, no consecutive terminators, and no other reasons to apply rules 2 through 5. We find the total time by adding the times for each gesture:

$$H + M + P + K + K$$

$$0.4 + 1.35 + 1.1 + 0.2 + 0.2 = 3.25 \text{ seconds}$$

This calculation applies to the lucky case in which the input thermometer was preset to the appropriate range and resolution. Now consider the case in which Hal wants to expand the scale factor so that he can see the desired temperature, change the range, compress the scale factor to get adequate resolution, and then move the arrow. I will write down the method Hal uses, without going through a step-by-step derivation. (I assume that Hal is a perfect user and does not have to juggle back and forth to find the right places on the thermometer.) Hal has to use the arrows to scroll several times. Each scrolling operation may require several gestures; the computer

then animates the scrolling operation, which takes time. To estimate scrolling times for the analysis, I built a similar interface and measured scrolling times, which were all 3 seconds or longer. Using *S* to represent the scrolling times, we can write the sequence of gestures that Hal uses as follows:

HPKSKPKSKPKSKPKK

Using the rules to place *Ms*, we get

H + 3(M + P + K + S + K) + M + P + K + K

0.4 + 3 * (1.35 + 0.2 + 3.0 + 0.2) + 1.35 + 0.4 + 0.2 + 0.2 = 16.8 seconds

Except for the rare case in which the thermometer scales are correctly set at the beginning of the problem, a perfect user will need more than 16 seconds to accomplish a temperature conversion using this method. A real—imperfect—user would jog the scales and the arrows back and forth and thus take even longer.

4-3 Measurement of Interface Efficiency

Every tool carries with it the spirit by which it has been created.

—*Werner Karl Heisenberg*

We have looked at two interfaces, one of which will take about 5 seconds to operate and the other of which will take more than 15 seconds to operate. It is clear which of the two better satisfies the requirement. The next question that we ask is how fast an interface that satisfies the requirement can be.

Given a design for an interface, you can use GOMS and its extensions to calculate how long a user will take to accomplish any well-defined task with that interface. But analysis models do not answer the question of just how fast you should expect an interface to be. To answer this question, we can use a measure from information theory. In the following discussion, *information* is used in the technical sense of a quantification of the amount of data conveyed by a communication, such as when two people have a telephone conversation or when a human sends a message, such as a click of the GID button when the cursor is at a certain location, to a machine. Before dealing with the technical details of measuring the amount of information a user must provide to accomplish a task, we establish the need for such a measurement.

To make a reasonable estimate of the time that the fastest possible interface for a task would take, we can proceed by first determining a lower

bound on the amount of information a user has to provide to complete that task; this minimal amount is independent of the design of the interface. If the methods of a proposed interface require an input of information that exceeds the calculated lower bound, the user is doing unnecessary work, and the proposed interface can be improved. On the other hand, if the proposed interface requires the user to supply exactly the amount of information that the task requires, you cannot make a more information-efficient interface for this task. In this latter case, there may yet be ways of improving—and there are certainly many ways of ruining—the interface, but at least this one efficiency goal will have been met.

Information-theoretic efficiency is defined similarly to the way efficiency is defined in thermodynamics; in thermodynamics we calculate efficiency by dividing the power coming out of a process by the power going into that process. If, during a certain time interval, an electrical generator is producing 820 watts while it is driven by an engine that has an output of 1,000 watts, it has an efficiency of 820 / 1,000, or 0.82. Efficiency is also often expressed as a percentage; in this case, the generator has an efficiency of 82 percent. A perfect generator—which by the second law of thermodynamics cannot exist—would have an efficiency of 100 percent.

The **information efficiency** E of an interface is defined as the minimum amount of information necessary to do a task, divided by the amount of information that has to be supplied by the user. As is true of physical efficiency, E is at least 0 and is at most 1. Where no work is required for a task and no work is done, the efficiency is defined as 1. (This formality is necessary to avoid the case of 0 divided by 0, as in responding to a transparent error message. See Section 5-5.)

E can be 0 when the user is required to provide information that is totally unnecessary (Figure 4.4). Surprisingly, a number of interface details achieve the dubious honor of having $E = 0$. A dialog box that allows the user only one possible action, such as clicking the box's OK button, is such an example. (JavaScript has a command, *Alert*, solely for creating such unneces-



Figure 4.4. A dialog box with an information theoretic efficiency of 0.

sary boxes: The designers were wise enough to remove *goto* from the JavaScript language to force structured code, but they failed to provide similar guidance on the interface side.)

E takes into account only the information required by the task and that supplied by the user. Two or more methods may have the same *E*, yet have different total times. It is even possible that a first method has a higher *E* yet is slower than a second method—for example, *MKMK* versus *MKKK*. In this example, only two characters have to be entered when the first method is used. In the second method, three characters are required, yet it takes less time to perform the task. It is difficult to construct many real-life situations that exhibit this inversion of speed and information efficiency.[2] For the most part, the more efficient interface is also the more productive, more humane interface.

Information is measured in bits; a single bit, which represents a choice between two alternatives—such as 0 and 1, on and off, or yes and no—is the unit of information.[3] For example, a choice made among four objects would require 2 bits of information: If the objects are A, B, C, and D, the first bit could choose either A and B, or C and D; once that choice was made—say C and D—the second bit would choose either C or D. Two binary choices, or 2 bits, suffice to separate one item from a set of four. To choose among eight alternatives, you need 3 bits; to choose among sixteen items, you need 4 bits; and so on. In general, given n equally likely alternatives, the amount of information communicated by all of them taken together is the power of 2 equal to n:

$$\log_2 n$$

And the amount of information in any one of them is

$$(1/n) \log_2 n \tag{1}$$

If the probabilities among the alternatives are not necessarily equal and the ith alternative has probability $p(i)$, the information associated with that alternative is

$$p(i) \log_2 (1 / p(i)) \tag{2}$$

The amount of information is the sum (over all alternatives) of expression (2), which reduces to expression (1) in the equiprobable case. It

2. It is possible to design more sophisticated measures of efficiency; for example, the *M* operator does not enter into our calculation. However, the simple measure defined here suffices for the purposes of this book.

3. *Bit* is mathematician John W. Tukey's contraction of the words BInary digiT (Shannon and Weaver 1963, p. 9).

follows that the information content of an interface that allows only the tap of a single button is 0 bits; not tapping the button is not permitted:

$$1 \log_2 (1) = 0 \tag{3}$$

It would seem, however, that the required tap of a single button can, for example, cause the ignition of dynamite used to demolish a building. Would this tap of the button then convey information? It would not, because not tapping the button was not an alternative; the interface "allows only the tap of a single button." If, however, the button was not tapped during, say, a five-minute time window in which the demolition was permitted, the building would not be demolished, and the tap or nontap would convey up to 1 bit of information because there were, in this case, two possible messages. From expression (2), we know that the calculation involves the probability, p, that the building will be exploded. The probability that it will not be exploded is therefore $1 - p$. From expression (2), we can calculate the information content of this interface:

$$p \log_2 (1 / p) + (1 - p) \log_2 (1 / (1 - p)) \tag{4}$$

When $p = ½$, expression 4 evaluates to

$$½ \times 1 + ½ \times 1 = ½ + ½ = 1$$

Expression (4) evaluates to less than 1 if $p \neq ½$. In particular, it evaluates to 0 when $p = 0$ or $p = 1$, as in expression (3).

This example illustrates an important point: We can measure the information embodied in a message only in the context of the set of possible messages that might have been received. To calculate the amount of information that has been conveyed by the reception of a message, we must know, in particular, the probability of that message having been sent. The amount of information in any message is independent of other messages past or future, is without reference to time or duration, and does not depend on any other events; similarly, the outcome of the flip of a fair coin is unaffected by previous tosses or by what time of day it is tossed.

As explained in Shannon and Weaver (1963), it is also important to keep in mind that

> information should not be confused with meaning . . . information is a measure of one's freedom of choice when one selects a message. . . . Note that it is misleading (although often convenient) to say that one or the other message [when just two are possible] conveys [1 bit of] information. The concept of information applies not to the individ-

> ual messages (as the concept of meaning would), but rather to the situation as a whole, the unit information indicating that in this situation one has an amount of freedom of choice in selecting a message, which it is convenient to regard as a standard or unit amount. (p. 9)

However, a user's actions in performing a task could be modeled with greater accuracy as a Markoff process, whereby the probability of a later action depends on earlier actions taken by the user, but the single-event probabilities discussed are sufficient for the purposes of this book; messages are assumed to be independent and equiprobable.

The amount of information conveyed by nonkeyboard devices can also be calculated. If your display is divided into two regions—one labeled Yes and the other labeled No—a single click in one or the other region would supply 1 bit of information. If there are n equally likely targets, with one click, you supply $\log_2 n$ bits of information. If the targets are of unequal size, the amount of information given by each does not change, but it does take longer to move the GID to smaller targets—by an amount that we shall show how to calculate presently. If the targets have unequal probability, the formula is the same as that already given for keyboard inputs with unequal probabilities. There is a difference in that a user can operate a keyboard key in 0.2 sec, whereas it will take 1.3 sec to operate an on-screen button, on average, ignoring homing time.

For our purposes, we can calculate the information content of voice input by treating speech as a sequence of input symbols, rather than as a continuous phenomenon with a certain bandwidth and duration.

This treatment of information theory and its relationship to interface design is a simplified account. Yet even in this rudimentary form, information theory—used in a manner analogous to our use of the simplified GOMS keystroke-level model—can give us first-order guidance in evaluating the quality of our interface designs.

4-3-1 Efficiency of Hal's Interfaces

> *Accurate reckoning: The entrance into the knowledge of all existing things and obscure secrets.*
>
> —*Rhind Papyrus, c. 1650 B.C.*

It is useful to go through a detailed example of a calculation of the average amount of information required for an interface technique. I will again use the temperature-conversion example. According to the requirement,

the input needed by the converter consists of an average of four typed characters; a decimal point occurs once in 90 percent of the inputs and not at all in the other 10 percent, and the negative sign occurs once in 25 percent of the inputs and not at all in the other 75 percent. For simplicity, and because there is no need for 1 percent precision in the answer, I will assume that all of the other digits occur with equal frequency, and I will ignore the 10 percent of the inputs that have no decimal point.

We need to determine the set of possible messages and the probability of each. Five forms are possible, where *d* denotes a digit:

1. *−.dd*
2. *−d.d*
3. .ddd
4. d.dd *and*
5. *dd.d.*

The first two each occur 12.5 percent of the time, and there are 100 of each of them; the final three each occur 25 percent of the time, and there are nearly 1,000 of each.[4] The probability for either of the first two types of messages is (0.125 / 100) = 0.00125; the probability for any one of the final three types of messages is (0.75 / 3000) = 0.00025. The sum of the probabilities of the messages is, as it must be, 1.

The amount of information of each message, in bits, is given by expression (2)[5]:

$$p(i) \log_2 (1 / p(i))$$

This expression evaluates to approximately 0.012 for the negative values and to 0.003 for the positive values. Calculating 200 × 0.012 + 3000 × 0.003 gives a total of 11.4 bits for each message.

Taking the probabilities into account can be important. If we took a simple-minded approach and assumed that all of the 12 symbols (minus, decimal point, and the 10 digits) were equally likely, the probability of each would be 1/12, and the information contained in a four-character message would be approximately

$$4 \log_2 (12) \approx 14 \text{ bits}$$

---

4. The "nearly" comes from the fact that the temperature of 0 degrees will not be entered as 0.00 or 00.0.

5. To get logs to the base 2 on a calculator or a computer that has only natural logs (ln), use: $\log_2 (x) = \ln (x) / \ln (2)$.

It is a theorem of information theory that the information is at a maximum when all symbols are equally likely. Therefore, making the assumption of equiprobable messages will give you a value that is equal to or greater than the amount of information in each message. Obviously, this assumption also makes estimating the information content of a message easier to compute. If the resultant value of the approximation is smaller than the amount of information your interface requires the user to supply, you do not yet need to bother with the more refined calculation.

We have just calculated that the task requires that Hal supply an average of about 11 bits of information each time he has to convert a temperature. We can—and will, presently—divide this quantity by the amount of information the interface requires him to supply. The result will be the efficiency of the interface.

Another simplification for quick analysis is to find the amount of information in a keystroke or a GID operation and then to count the various gestures. When a keystroke delivers information to a computer, the amount of information delivered depends on the total number of keys available—for example, the number of keys on the keyboard—and the relative frequency with which each key is used. Thus, keystrokes can be used as a rough measure of information. If a keyboard had 128 keys, each of which had the same frequency of use, each key would represent 7 bits of information. In practice, the frequency of use varies tremendously—for example, space and *e* are common, whereas *j* and \ are rare), and the information per keystroke is closer to 5 bits in most applications. The requirement stated that the average length of the input that specifies the temperature was four keystrokes.

For this analysis, it is easier to use a measure simpler than information-theoretic efficiency but that often achieves the same practical effect. **Character efficiency** is defined as the minimum number of characters required for a task, divided by the number of characters the interface makes the user enter.

Achieving an interface that required four keystrokes, on average, would give us a character efficiency of 100 percent. If we add a keystroke to decide which conversion is desired and then another to delimit the answer, our average length of input will grow to six keystrokes, and our keystroke efficiency will drop to 67 percent. If Hal has as his input device only a 16-key numeric keypad, the information provided by a single keystroke would be 4 bits, and the interface would be more efficient. (The requirements, however, do not permit us to use this option.)

Because any task in a GOMS analysis requires at least one mental operator, the most keystroke-efficient interface for the temperature-conversion problem will have, in theory, an average time of

$$M + K + K + K + K = 2.15 \text{ sec}$$

Thus, it will be considerably faster than either of the two interfaces already discussed. However, typing four characters on a standard keyboard supplies at least 20 bits of information, whereas only 11 bits are required—an information-theoretic efficiency of 55 percent—so we know that there is room for improvement. As we have seen, using a standard numeric keypad instead of a full keyboard drops the input information per four keystrokes to 16 bits, raising the efficiency to about 60 percent. A dedicated numeric keypad—one that has only the digits, the minus sign, and a decimal point—will permit a slightly higher score, of about 70 percent efficiency. We raise the score again by using special encodings of temperature information and novel input devices, but training difficulties and excessive costs begin to loom with these extreme approaches, so I will stop here and accept 70 percent information-theoretic efficiency. Theoretical limits may or may not be reached by a practical interface, but they do give us a star by which to steer.

4-3-2 Other Solutions for Hal's Interface

In Section 4-3-1, we stopped trying to improve information-theoretic efficiency when we reached 70 percent. We achieved that efficiency with an unspecified, theoretical interface that somehow managed to have 100 percent keystroke efficiency. Let us see how close we can come to this ideal with a standard keyboard and a GID.

Consider an all-keyboard interface. In this interface, a note appears on the display:

```
To convert temperatures, indicate the
desired scale by typing C for Celsius or F
for Fahrenheit. Type the numeric tempera-
ture; then press the Enter key. The con-
verted temperature value will be displayed.
```

A GOMS analysis finds that the user must make six keystrokes. Following the rules for placements of *M*s gives us

$$MKKKKKMK$$

The average time is 3.9 seconds.

We can decrease this time if we can use the C or the F itself as a delimiter. That is, consider an interface in which the following instructions appear:

> **To convert temperatures, type the numeric temperature, followed by C if it is in degrees Celsius or F if it is in degrees Fahrenheit. The converted temperature will be displayed.**

In this example, the Enter key is not used. Some primitive interface-building tools demand that the user tap Enter and will not permit us to use C or F as a delimiter; such tools are inadequate for building humane interfaces.

The GOMS analysis of the C/F-delimiter interface yields

MKKKKMK

The average time is 3.7 seconds. If we did not have an analysis that showed that the theoretical minimum time is 2.15 sec, this solution might strike us as satisfactory. It is considerably more efficient than the ones that we discussed previously, so we might stop here. Tempted by that theoretical minimum, however, we ask whether there is an even faster approach. Consider the interface depicted in Figure 4.5; we might describe it as *bifurcated:* One input will give us two outputs.

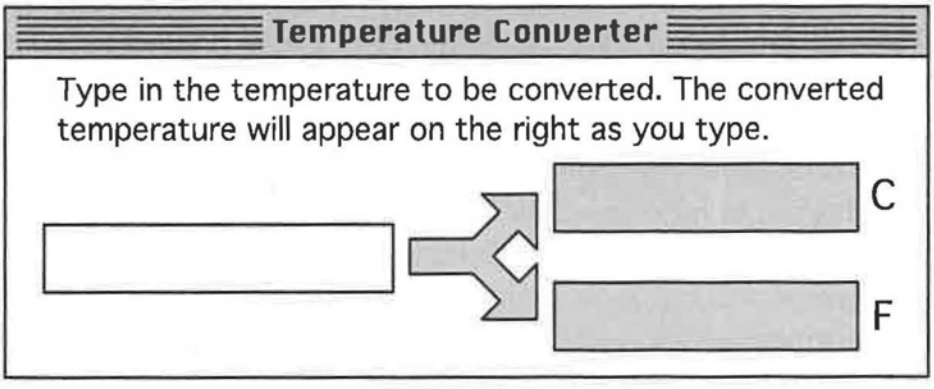


Figure 4.5. An interface that does not require a delimiter. A more efficient interface is made possible by taking advantage of character-at-a-time interaction, and by performing both conversions at once.

Under the bifurcated interface, no delimiter is required. Furthermore, the user does not have to specify which conversion is desired. The GOMS analysis for the average input of four characters is

MKKKK

The bifurcated interface achieves the minimum 2.15 seconds and has 100 percent character efficiency.

If, as in our example, the output sometimes changes when a character is typed, the flickering of the output does not distract you, because your locus of attention is the input. The continually changing output is often beneficial: The user will notice it only peripherally after the first few times that he uses the feature, at which point it will provide him feedback that the system is responding to his input. For single-character interaction to be effective, the system must respond quickly; in particular, the interaction must keep up with the user's typing speed. Only a slow network connection should exhibit this problem.

Although not part of the requirement, you might ask how this converter is "cleared" for the next operation. Does the clear operation add a keystroke? Not necessarily. For example, we could design the interface such that, whenever the operator returns to his background task or goes on to another task, the values in the converter are automatically grayed and the converter becomes inactive. The values shown are not cleared at this time, so that they can be referred to again if necessary. The next input to the converter does clear the old values.

Just because it has optimal speed of operation and is highly efficient, the bifurcated converter is not necessarily the best interface of those discussed or of those possible. Parameters other than speed also are of importance, such as error rate, user learning time, and long-term user retention of the way to use the interface. We should be especially concerned about the error rate of the bifurcated converter, due to Hal's possibly reading the wrong output box, especially because he may have just heard, for example, the word *Celsius* and thus be required to read out the Fahrenheit line. Nonetheless, the bifurcated converter would definitely be on the short list of interfaces to be tested for the temperature-converter application, and a few others that we have seen—solutions that might otherwise have seemed worth a try had we not learned how to do a GOMS analysis—would not make the cut.

Whether we use it in a simple keystroke-timing analysis or in a detailed information-theoretic extravaganza, a quantification of the theoretical minimum-time, minimum-character, or minimum-information interface can be a useful guide for our designs. *Without a quantitative guide, we are only guessing at how well we are doing and at how much room there is for improvement.*

4-4 Fitts' Law and Hick's Law

It behooves us to place the foundations of knowledge in mathematics.

—*Roger Bacon,* Opus Majus *(13th century)*

Various quantitative laws relating to interface design have sound cognetic underpinnings and have been validated repeatedly. These laws often give you additional data on which you can base interface-design decisions. Fitts' law quantifies the fact that the farther a target is from your current cursor position or the smaller the target is, the longer it will take you to move the cursor to the target. Hick's law quantifies the observation that the more choices of a given kind you have, the longer it takes you to come to a decision.

4-4-1 Fitts' Law

Consider that you are moving a cursor toward an on-screen button. The button is the *target* of the move. The length of a straight line from the position at which the cursor started to the closest point on the target is the *distance* used in the statement of Fitts' law. Given the size of the target and the distance to be moved, **Fitts' law** gives you the average time it takes a user to succeed in getting the cursor to the button.

In the one-dimensional case, in which the target's size, measured along the line of motion, is S and the target is at a distance D from the starting position (Figure 4.6), Fitts' law states that

$$\text{Time (in msec)} = a + b \log_2 (D / S + 1)$$

(The constants a and b are determined experimentally or are derived from human performance parameters.)[6] The time that you calculate begins when

---

6. Mathematics, supposedly a paragon of clarity, clings yet to that old-fashioned style whereby undefined variables appears in a formula before you know what they stand for. For example, you will see such statements as

$$A = \pi r^2,$$

where r is the radius of a circle and A is its area.

This can be confusing, forcing you to read ahead and then go back, especially if the equation is a long one with lots of as-yet-unexplained variables. Far better, from a reader's viewpoint, is to follow the obvious dictum to define terms *before* you use them:

A circle with radius r has an area A, given by:

$$A = \pi r^2$$

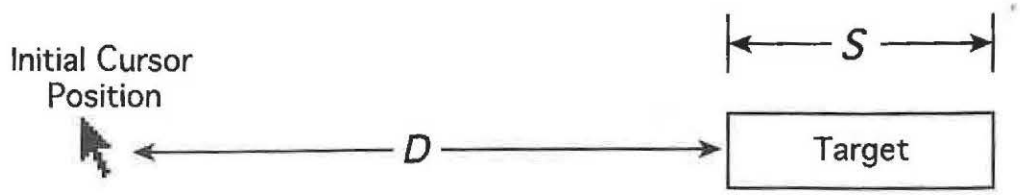


Figure 4.6. Distances used in Fitts' law to determine the time to move a cursor to a target.

the cursor is at the starting point and after the user has chosen the target. The logarithm to the base 2 gives a measure of the difficulty of the task in terms of the number of bits of information it takes to describe the (one-dimensional) path of the cursor.

The units of distance do not affect the calculated time, because D / S is the ratio of two distances and is therefore dimensionless. It follows that, even though we might move the pointing device a distance smaller or larger than the distance the cursor moves on the display, the law still works when the distances are measured on the display, assuming a linear relationship between GID and cursor motion. Fitts' law applies only to the kinds of motions we make when we are using most human-machine interfaces: motions that are small relative to human body size and that are uninterrupted, that is, movements that can be made in one continuous motion. For back-of-the-envelope approximations, I use $a = 50$ and $b = 150$ in the Fitts' law equation.

An extension of Fitts' law to more complex constraints, such as tracking a cursor between straight or curved walls, has been developed and tested empirically (Accot and Zhai 1997). For a two-dimensional target, you can usually obtain a reasonable approximation of the time needed to move the cursor to the target, using the smaller of the horizontal and vertical dimensions of the target for the value of S (Mackenzie 1995).

Fitts' law explains, for example, why it is much faster to move the cursor to an Apple Macintosh–style menu (Figure 4.7) that is on the edge of a display than to a Microsoft Windows–style menu (Figure 4.8) that floats away from an edge. The size S of the Windows menu on my display is 5 mm. The effective size of the Macintosh target is large because you do not have to stop within the confines of the menu bar but rather can continue to move the GID any comfortable distance beyond that needed to put the cursor in the menu: The cursor stops at the edge of the display.

A series of tests I performed determined that users typically stop within about 50 mm of the edge of the display on the Macintosh, so we can use 50 mm as S for the Macintosh. On a 14-inch flat panel display, the aver-

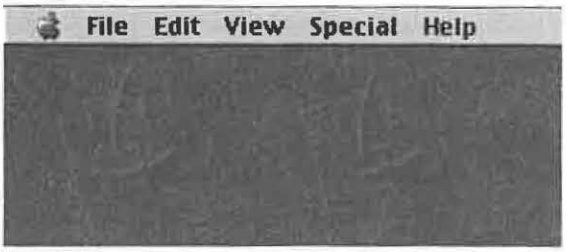


Figure 4.7. The Macintosh menu, at the top edge of the screen, effectively increases its size compared to a menu that floats away from the edge. (See color insert.)

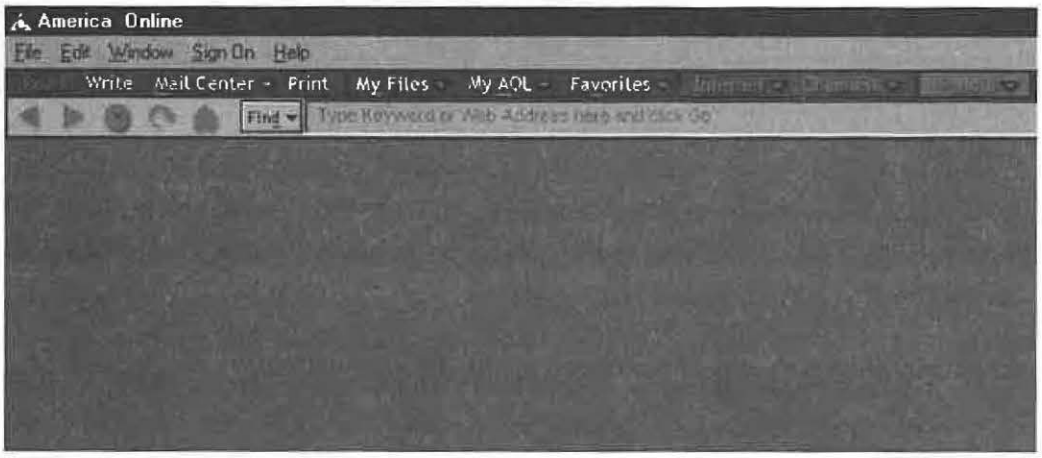


Figure 4.8. The Windows menu is below the top edge of the screen; you have to place the cursor more carefully to pull down a submenu. (See color insert.)

age distance the cursor must be moved to reach the menu bars is 80 mm; thus, the calculated time to move the cursor to a menu item on the Macintosh is

$$50 + 150 \log_2 (80 / 50 + 1) = 256 \text{ msec.}$$

This result is far less than the calculated time it takes to move the cursor to a corresponding menu item on a Windows-style menu:

$$50 + 150 \log_2 (80 / 5 + 1) = 663 \text{ msec.}$$

These calculations apply only to the time it takes the user to move the cursor. Clicking on the target to indicate that you believe that the cursor has reached its goal adds another 0.1 sec on average. (K = 0.2 in the GOMS model includes both the downstroke and the release of the button, whereas this timing is stopped by the downstroke.) In a typical experimental situation, you have to add the human reaction time of about 0.25 sec at the start

of the cursor movement. When we take these factors into account, we get times that agree with what I have observed: It takes about 0.6 sec, on average, for a user to open an Apple menu, whereas it takes a user more than 1 sec to open a Windows menu. This analysis makes it clear why menus were deliberately placed at the edge of the display when the Macintosh interface was developed.

4-4-2 Hick's Law

Before you move the cursor to a target or take any one action with a multiplicity of choices, you must first choose the target or action. **Hick's law** says that when you have to choose to take one among n alternative actions and when the probabilities of taking each alternative are equal, the time to choose one of them is proportional to the logarithm to the base 2 of the number of choices, plus 1. When put this way, Hick's law looks just like Fitts' law:

$$\text{Time (in msec)} = a + b \log_2 (n + 1)$$

If the probability of the ith choice is $p(i)$, then, instead of the logarithmic factor in the equation, you use

$$\sum_i p(i) \log_2 (1 / p(i) + 1)$$

The coefficients in Hick's law are strongly dependent on many conditions, including how the choices are presented and how habituated to the system the user has become. (If the choices are presented in a confusing manner, both a and b can increase; habituation decreases b.) These dependencies will not be discussed here; all we need to consider is that making decisions takes time, that making complex decisions takes more time than making simple ones, and that the relationship is logarithmic. In the absence of better information, we can use the same coefficients a and b as for Fitts' law to make off-the-cuff or relative estimates.

Whatever positive, nonzero coefficients we use for a and b, it follows from Hick's law that giving a user many choices simultaneously is usually faster than is organizing the same choices into hierarchical groups. Making choices from one menu of eight items is faster than is making choices from two menus of four items each. Assuming that the items are equally likely to be chosen—and ignoring the time it takes to open the second menu, which, if taken into account, would make the time taken for the two-menu interface even longer—we compare the time to select one item of eight, $a + b$

$\log_2 8$, with the time to select one item of four twice, $2\ (a + b \log_2 4)$. Because $\log_2 8 = 3$ and $\log_2 4 = 2$, and because both $a < 2a$ and $3b < 4b$, we see that

$$a + 3b < 2\ (a + 2b)$$

This accords with experiments on menu structures (see, for example, Norman and Chin 1988).

Our discussion of Fitts' and Hick's laws is incomplete. For example, it is no accident that they have the same form as the Shannon-Hartley theorem. Nonetheless, this brief treatment is sufficient to alert you to these useful guides to interface design. They can help you even if, as in the example, you do not know the empirical coefficients a and b. (For more detail, see Card, Moran, and Newell 1983, pp. 72–74.)

FIVE

Unification

It is immensely ingenious, immensely complicated, and extremely effective, but somehow at the same time crude, wasteful, and inelegant, and one feels that there must be a better way of doing things.

—C. Strachey (speaking not of Windows but of the IBM Stretch computer in 1962)

In trying to create a general-purpose interface that takes into account the requirements that we have delineated in the previous four chapters, we find that basic changes from present practice are necessary. Many directions are possible; one tack is to see what we can do within the limitations of the Internet and with the hundreds of millions of computers and information appliances that exist and that are being manufactured today.

The most common personal computer hardware configuration is nearly universal at present. By taking a point of view that emphasizes the commonality of physical actions across almost all applications, using the common hardware elements, rather than looking at the vast disparity of tasks, we can develop a general yet simple interface.

The list of actions a user can take to influence content—be that content textual, graphical, or multimedia—can be arranged into a simple taxonomy, which allows us to describe any application's interface in a uniform way. This organization can help guide us in simplifying interface designs. Implementing a universal undo/redo facility also helps to develop a unified interface, eliminating much of the need for individual programs to handle error recovery.

Different applications have different commands, and a user cannot in general use the commands from application A while working in application B and vice versa. By liberating commands from applications, we eliminate the inherent modality of applications. The total number of commands a user must master drops dramatically with this kind of unification, primarily because unification rids us of the immense duplication of commands. For example, on the Canon Cat, a total of 20 commands handled word processing: spreadsheet operations; database setup, retrieval, and sorting; calculations; and more. Contemporaneous systems of comparable power required more than 100 commands to accommodate the same set of tasks. In today's environment, the thousands of commands now provided would be reduced to the hundreds. Because not all commands apply to all data types, we would need to apply data-type transformers to objects to create new objects that, whenever possible, can be acted on by the chosen command.

Another division that can be erased is between facilities provided, at present, by commercial software and by the user. For example, menus are, at present, operating system objects that are set up by each application. However, they are just text. Is there any reason that a user cannot type up a list of regularly used commands, lock them so that they do not get changed accidentally, stick them up at the top of the display, and treat them like a system-provided menu? To make such menu creation easy, an interface could allow the user to lock and unlock text and to make the text either flow with other content or stay fixed with respect to the display. These are various states of text.

Eudora and Microsoft Word are programs that allow you to change the content menus, but you must use a facility that the program provides specifically for that purpose. Here we are discussing the ability to create menus with the same mechanisms you use for creating and editing any text. Menus become content in this view.

The next step in the simplification of the interface is to eliminate difficult-to-remember and annoying-to-create file names and system-provided file structures. If given suitable search mechanisms, a user will find both the file names and the provided file structures unnecessary.

5-1 Uniformity and Elementary Actions

Entities should not be multiplied unnecessarily.

—William of Occam

The hardware that makes up a computer interface has become formulaic: one or more text input devices (keyboard, handwriting tablet, speech recognizer), a GID, and a two-dimensional color display. The formula, not a bad one, has a few variations; for example, a touch screen can be the text input device, the GID, and the display. Microphones for sound, video inputs, and other electronic interfaces are devices that are not, except experimentally, part of the usual human-machine interface. Indeed, we use the interface to control the functioning of these devices.

If you were to watch a person who is operating today's standard hardware, without your seeing what is being displayed and without any idea of what the operator is trying to accomplish, you would generally not be able to guess what he is doing. There are exceptions: A user's fixated stare at the display, accompanied by manic gyrations of a joystick to the beat of a persistent and repetitive sound track, are strong hints that a game is being played. But in general, user actions in one application, such as word processing, look much like user actions in others, such as a data entry task or a spreadsheet manipulation.

This uniformity of user actions across applications is a clue that interfaces for various applications are not as different as they might seem when you are involved in using the computer yourself. Applications seem more different than they are because you are attending to the content of what you are doing—that is, to the widely varying semantics of each action. In particular, you are not paying attention to the physical manipulations you are performing.

Another way in which applications are similar is that nearly all of them require text entry. (Even games have you type your name when you do well.) It is thus worth ensuring that text handling—whether in the small, as when the user is entering the string on which to execute a Find command, or in the large, as when the user is writing a novel, is accomplished with a smooth and efficient set of operations.

We and our software are not perfect; not all text-entry keystrokes, pen motions, or acts of speech will cause the desired characters to be displayed. Therefore, an essential provision of an interface is that keystrokes can be erased with an immediate tap of the Backspace (or Delete) key, with analogous methods for other forms of input. Larger changes, such as adding a

paragraph, require that the user be able to select regions and to delete them. Another fundamental requirement, except for brief inputs, is that the user be able to move the cursor to any point in the text and to insert additional characters. In short, whenever text is to be entered, the user expects to have available many of the capabilities of a word processor.

Whenever you are entering text, you are placing it into a document or a field, such as an area provided for you to enter your name on a form. With present systems, the rules about what editing functions are available change from field to field and from one document type, such as a word processor document, to another, such as a spreadsheet. The editing rules can change within a single document in systems that allow the document to include portions generated by separate applications (in Section 5-7, we look at a solution to this problem).

Two different but similar pieces of software on a system form a prime breeding ground for user confusion and frustration. Yet that is precisely what we do have on nearly every personal computer system. My computer has 11 text editors, each with its own set of behaviors, and there may be a few editors that I missed. The situation is unnecessarily confusing.

One important step in creating a humane interface for computers and or for computerlike systems, such as a Palm Pilot, is to ensure that the same rules apply whenever text is being entered and when editable text is selected. For example, on the Macintosh or in Windows, you cannot spell-check a file name in situ, so if you are not sure of the spelling of *rendezvous* and you want to use that word as a file name,[1] you will have to guess at the spelling or open a word processor and retype or drag *rendezvous* into that word processor to check it. I suspect that, if I suggested to software developers that users should be able to check file name spellings, they might well add a new feature. Such an ad hoc addition—which would probably take the form of a new menu item in one of the desktop menus, probably the Edit menu, would only increase the already absurd complexity of the software. It is better to simplify by means of the idea presented here, that one command for spell-checking should suffice for any text, whatever role that text might be playing at the moment.

Interface design should be such that *any objects that look the same are the same.* Insisting on this principle results in a boon of simplicity for user and programmer alike and is a concept that extends far beyond text. Every object for which this can be done is an affordance. If a user cannot tell what he may and may not do with an on-screen object by looking at it, your interface fails

1. Of course, if your operating system doesn't allow long names, you have another problem.

to meet the criterion of visibility as discussed in Section 3-4. You put the user in the position of having to guess what operations are possible and to guess what will happen when a given operation is performed. Requiring the user to guess at what a piece of software will do is an interface technique more suited to games than to tools.

The ideal of having appearance invariably indicate function is not, in general, achievable. For example, one object can mimic or spoof another. A bitmap of text looks exactly like text, but in current systems, text-editing operations fail on bitmaps. This kind of problem can be partially surmounted if the system always attempts to transform the object into a type to which the operation applies, a notion we discuss in Section 5-8.

5-2 Elementary Actions Cataloged

As you design an interface, you should have the palette of possibilities arrayed in your mind, much as a painter has his colors organized. The spectrum of **elementary actions** that a user can perform is remarkably limited. From this set of elementary actions, all user interaction is built. With a keyboard, you can tap keys or press and hold them while you perform other actions. With a GID, you can move a cursor within the bounds of the display (or displays) on your system, and you can signal using the computer, using the speed, direction, and acceleration of the GID, although you usually use GID speed and acceleration only as an aid to pointing. With a GID button, you can signal display locations to which you are pointing with the cursor. These elementary actions have widely varying semantics, depending on the application being run.

Pressure-sensitive graphic tablets can detect the angle at which the pen is held, which results in one or two additional numerical values being associated with each location to which the user points. Except when the user is doing freehand drawing, these parameters are rarely used. Musical keyboards may provide, as inputs to the computer, both the velocity with which the key is pressed and the pressure with which the key is being held down after being pressed. There are also joysticks and three-dimensional input devices. Nonetheless, most interaction is accomplished with a keyboard and a standard, two-dimensional GID. This section will address primarily the standard input and output devices. In many cases, it will be clear how the principles extend to more exotic physical or, eventually, mental interfaces. Having an explicit taxonomy and vocabulary of elementary actions and the elementary operations built from them is, I find, a great aid in discussing and designing interfaces.

The user performs the elementary actions in various combinations to a set of **elementary operations**. The elementary operations are performed on content and are used in nearly every interface. We first note that content can be

- **Indicated**: pointed at
- **Selected**: distinguished from other content
- **Activated**: clicked on
- **Modified** or used (by means of commands) by being
  - **Generated:** modified from empty to nonempty
  - **Deleted**: modified from nonempty to empty
  - **Moved**: inserted in one place and simultaneously deleted from another
  - **Transformed**: changed to another data type
  - **Copied**: sent to or received from an external device or duplicated at a different internal location—for example, printed, e-mailed, stored on a hard disk, copied into another document.

These elementary operations can and should be fundamental to the computer or appliance itself; that is, they should be part of the internal hardware or software rather than being reimplemented in multiple software packages, and each of them should always be invoked in the same way, regardless of on what objects they are operated. For the most part, *the cognitive differences among applications lie in how selections are presented and how the user can operate on them*. In a spreadsheet, values are presented in tabular form, and an operation might consist of changing a column without a sum underneath into a column with its sum underneath. In a word processor, text and illustrations are presented in page-layout form, and a typical operation is to change text from a roman typeface to an italic typeface. In a web page program, a word processor page might be changed into HTML. In a photo-processing program, an image with low contrast might be modified into one of higher contrast.

Most operations performed on content can be described in terms of these elementary operations. For example, in many systems, an object can be queried about its properties. (In systems equipped with two-button GIDs, the user usually performs this function by a click of the right button when the cursor is on the object and the system is in an appropriate state.) To **query** is to bring up further information or options on the item, but it also