Usability Engineering

JAKOB NIELSEN

SunSoft
2550 Garcia Avenue
Mountain View, California

Morgan Kaufmann

AN IMPRINT OF ACADEMIC PRESS
A Harcourt Science and Technology Company

San Diego San Francisco New York Boston
London Sydney Tokyo

Chapter 5 *Usability Heuristics*

This chapter presents some basic characteristics of usable interfaces. The principles are fairly broad and apply to practically any type of user interface, including both character-based and graphical interfaces [Nielsen 1990e]. The principles are summarized in Table 2 (page 20). After detailed sections for each of the ten basic heuristics, Section 5.11 (page 155) concludes the chapter with information on how to use usability heuristics as a basis for a systematic inspection of a user interface to find its usability problems (the *heuristic evaluation* method).

5.1 *Simple and Natural Dialogue*

User interfaces should be simplified as much as possible, since every additional feature or item of information on a screen is one more thing to learn, one more thing to possibly misunderstand, and one more thing to search through when looking for the thing you want. Furthermore, interfaces should match the users' task in as natural a way as possible, such that the mapping between computer concepts and user concepts becomes as simple as possible and the users' navigation through the interface is minimized.

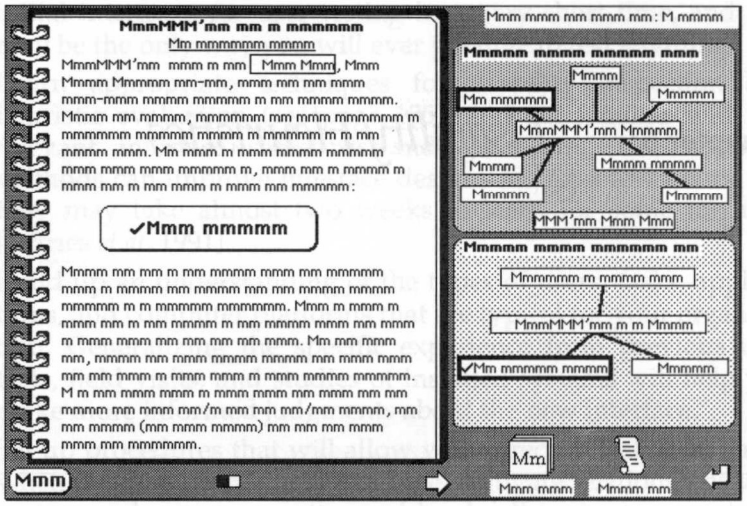

Figure 11 *Mumble screen layout for a hypertext system. The actual system is described in [Nielsen 1990a, 1990i]. The screen could be made to abstract even further from the information content in the full system by replacing the icons with generic shapes.*

The ideal is to present exactly the information the user needs—and no more—at exactly the time and place where it is needed. Information that will be used together should be displayed close together, and at a minimum on the same screen. Also, both information objects and operations should be accessed in a sequence that matches the way users will most effectively and productively do things. Sometimes such sequences are enforced by the user interface, but it is normally better to allow the user to control the dialogue as much as possible such that the sequence can be adjusted by the individual user to suit that user's task and preferences. Even so, the system may ease the user's understanding of the dialogue by indicating a suggested or preferred sequence, such as the sequence implied by the listing of fields in a dialog box from top to bottom.

Graphic Design and Color

Good graphic design is an important element in achieving a simple and natural dialogue for modern computer systems with graphical user interfaces [Marcus 1992]. In addition to getting help from a graphics designer, there are several simple considerations that may lead to simpler dialogues. By prototyping screen layouts using "mumble screens" like that shown in Figure 11 where all text has been replaced with the letter "m," one can abstract away from the detailed information content in the system and focus on layout issues.[1]

Screen layouts should use the gestalt rules for human perception [Rock and Palmer 1990] to increase the users' understanding of relationships between the dialogue elements. These rules say that things are seen as belonging together, as a group, or as a unit, if they are close together, are enclosed by lines or boxes, move or change together, or look alike with respect to shape, color, size, or typography. For example, in Figure 12, most people will perceive two major groups of objects due to the closeness of the objects within the groups compared with the distance between the groups. Then, most people would think that the left set of objects contained two sets of objects that were even more closely related, due to the enclosure of the six objects in the upper right corner and the highlighting of the four objects in the lower left corner. Also, the right set of objects will be perceived as containing three groups, and the middle one will be seen as standing out from the background since it is smaller.

These principles of graphic structure should be used to help the user understand the structure of the interface. For example, menus can use a dividing line or color coding [McDonald *et al.* 1988] to split options into related groups, each of which will be easier to understand because each option will be seen in a relevant context.

1. Mumble text has also been used as a task analysis technique for finding out whether users gain information simply from the way information is presented on a form without reading the detailed data [Nygren *et al.* 1992]. If they do, then one should design similar capabilities for users in a computerized information environment.

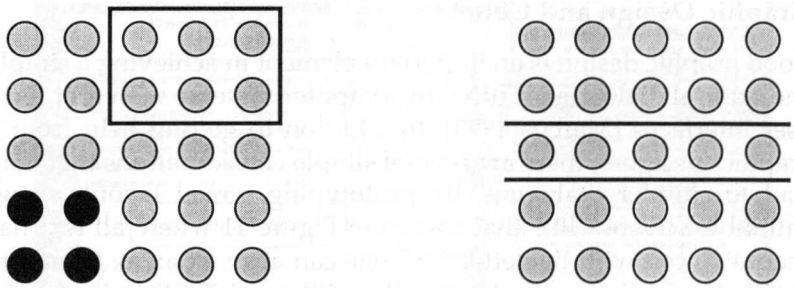

Figure 12 *Example of objects structured according to the gestalt principles of closeness, closure, and similarity.*

Similarly, dialog boxes can group related features and enclose them in boxes or separate them by lines or white space. Also, since users will perceive structure based on these principles, care should be taken not to separate out unrelated objects in ways that make them seem as belonging together. For example, consider a bank statement with the following layout:

```
Balance                                    $1,000.00
$2,000.00
```

What is the balance? One or two thousand dollars? The closeness rule will make many people perceive the label `Balance` as being matched with the number `$2,000`, even though it may have been intended as a label for the line containing the number `$1,000`.

Principles of graphic design can also help users prioritize their attention to a screen by making the most important dialogue elements stand out. As shown by the right part of Figure 12, a small delineated area will stand out from the background, and one can also make objects stand out by highlighting them in various ways, including the use of bolder colors or typefaces, and by making them larger. Also, information that is presented "first," given the usual reading direction (that is, at the top and to the left in many cultures) normally gets more attention. It is even possible to attract the users' attention by using blinking objects, but blinking is so distracting and annoying that it should only be used in extreme

cases. On alphanumeric terminals, UPPERCASE TEXT CAN ALSO BE USED TO GET THE USERS' ATTENTION, but upper case should be used sparingly as it is about 10% slower to read than mixed-case text.

With respect to the use of color in screen designs [Rice 1991; Travis 1991], the three most important guidelines are

- Don't over-do it. An interface should not look like an angry fruit salad of wildly contrasting, highly saturated colors. It is better to limit the design to a small number of consistently applied colors. Color-coding should be limited to no more than 5 to 7 different colors since it is difficult to remember and distinguish larger numbers, even though highly trained users can cope with about 11 colors [Durrett and Trezona 1982]. Also, light grays and muted pastel colors are often better as background colors than screaming colors are.
- Make sure that the interface can be used without the colors. Remember that many people (about 8% of males) are colorblind, so any color-coding of information should be supplemented by redundant cues that make it possible to interpret the screens even without being able to differentiate the colors. For example, icons that are about to be deleted could be turned red for fast identification by users with full color vision and they could also be marked with an X. The best test would be to have a selection of colorblind users try out the system, but it would be difficult to do so comprehensively as there are many different types of color blindness.[2] In addition to having at least some colorblind test users, one can also check how the interface looks on a monochrome screen. In many cases, some users will be limited to monochrome displays anyway.

2. Being "colorblind" normally does not mean than one cannot distinguish any colors at all, so the expression is somewhat inaccurate. About 6% of males and 0.4% of females are partially red–green colorblind (and so can distinguish yellows and blues as well as some shades of green and red), 2% of males and 0.03% of females are fully red–green colorblind, and only 0.005% of males and 0.003% of females are yellow–blue or completely colorblind [Silverstein 1987]. Therefore, a test with a single colorblind user (while much better than no such test) will not guarantee that all types of users with color-deficient vision will be able to use the interface without problems.

- Try to use color only to categorize, differentiate, and highlight, not to give information, especially quantitative information.

It is true that some colors and color combinations are more visible than others [Durrett 1987], and you certainly would not want to present your help screens in light blue text on a bright yellow background.[3] If the most obviously horrible color combinations are avoided, however, there is normally only a small additional benefit to be gained from searching out the absolutely optimal choice of colors.

Less Is More

The great rune stone in Jelling, Denmark (from the last half of the tenth century), bears the following inscription: *"King Harald ordered these memorials raised to Gorm, his father, and Thyre, his mother; that Harald who won for himself all Denmark and Norway and made the Danes Christian."* The stone does seem to focus excessively on King Harald, and the last half of the text distracts from the message regarding King Gorm and Queen Thyre.[4] Similarly, adding information and data fields to a user interface can distract the user from the primary information.

Based on a proper task analysis, it is often possible to identify the information that is truly important to users and which will enable them to perform almost all of their tasks. It will then normally be better to design a single screen with this information and relegate less important information to auxiliary screens than to cram all the information that might possibly be useful into a set of screens that will require the user to switch screens for even the most simple tasks.

Other information may not even be necessary at all. For example, many programs follow the example of King Harald and dedicate

3. Detailed guidelines for choosing screen colors can be found in part 8 of ISO 9241 (an international standard for user interface issues). The content of this standard is discussed further by Smith [1988].

4. Compare with the inscription on the small Jelling rune stone from the first half of the tenth century: *"King Gorm raised these memorials to his wife Thyre, the joy of Denmark."* The runes are fewer, but the message is focused.

large amounts of screen space to a display of the name of the program, the vendor's logo, the version number, and other similar information. Even though this information is potentially important and should be available for users making bug reports, it normally only takes up screen space that could have been used for other purposes (maybe even as white space to make for a better layout). And of course, *any* piece of information is something users will have to look at when they search the screen, and it will therefore slow down their performance by some fraction of a second. It is better to provide identifying information as part of a startup screen that can be extravagantly eye-catching and serve as feedback to the user that the appropriate program is being entered. Also, of course, identifying information about the program, its version, and its status should be accessible through the help system. As another example, headers with address and message routing information can be eliminated from displays of electronic mail and network news [Andersen *et al.* 1989] to be shown only in the rare case when a user actually needs this system-oriented information.

Extraneous information not only risks confusing the novice user, but also slows down the expert user. For example, a study of experienced telephone company directory assistance operators showed that finding a target that appeared in the top quarter of the screen took 5.3 seconds when the screen was half full of information and 6.2 seconds when the screen was full of information [Springer 1987]. Saving 0.9 seconds may not seem like a lot, but for this specific application, it was estimated to reduce call processing costs by more than 40 million dollars per year.

The "less is more" rule does not just apply to the information contents of screens but also to the choice of features and interaction mechanisms for a program. A common design pitfall is to believe that "by providing lots of options and several ways of doing things, we can satisfy everybody." Unfortunately, you do have to make the hard choices yourself. Every time you add a feature to a system, there is one more thing for users to learn (and possibly use erroneously) and the manual gets bigger, more intimidating, and harder to search. One study found that the users' planning time for formula entry was 2.9 seconds in one spreadsheet and 4.6 in

another [Olson and Nilsen 1987–88]. The first spreadsheet was faster because it only provided a single method for formula entry and therefore did not require users to think about which method to use. In contrast, the second spreadsheet provided multiple methods, with the result that the users were slowed down more by having to choose between methods than the amount of time they sometimes gained from being able to use a "faster" method for certain formulas.

This does not mean that one should never provide alternative interaction techniques. On the contrary, they are often a good idea as further discussed in Section 5.7, on Shortcuts, on page 139. Alternatives can be provided if users can easily recognize the conditions under which each one is optimal so that they can consistently choose the optimal interaction technique without additional planning. For training, users should at first be taught only the single, general method that is preferable in most common situations. Other methods can be taught later but should not be introduced at a stage when they will only confuse the novice user.

Sometimes, one can design an especially simple interface for novice users and shield them from any necessary complexity that may be needed by advanced users. Most systems doing this have only two levels of interface complexity: novice mode and expert mode, but in principle it might be possible to provide multiple nested levels of increased complexity. This nested design strategy is sometimes referred to as *training wheels* [Carroll 1990a].

Since novice users are often observed spending too much time recovering from errors, the training wheels approach can give them a system where they are blocked from even entering potential error situations. Of course, this limits the available functionality, but novice users probably do not need the advanced features anyway. In one experiment, novice users were able to learn basic use of a word processor and type a letter in 116 minutes when they were faced with the full system and in 92 minutes when they were given the training wheels system where actions leading to the most common errors were not available [Carroll and Carrithers 1984]. Not only did the training wheels users get started faster, but they

also liked the system better and scored slightly higher on a comprehension exam after the study. Even better, the initial use of the training wheels interface did not impair users when they later graduated to the full system. On the contrary, users who had learned the basics of the system with the training wheels interface learned advanced features *faster* than users who had been using the full system all the time [Catrambone and Carroll 1987].

5.2 *Speak the Users' Language*

As a part of user-centered design, the terminology in user interfaces should be based on the users' language and not on system-oriented terms. For example, a user interface for foreign currency transactions should not require users to specify British pounds with a code like `317`, even if it is the one used internally in the system. Instead, terms like `GBP` or simply `Pounds` should be used, depending on whether the system was intended for professional currency traders or for the general public.

As far as possible, dialogues should be in the users' native language and not in a foreign language (see Chapter 9 for a discussion of translation and other internationalization issues). Of course, concerns for the users' "language" should not be limited to the words in the interface but should include nonverbal elements like icons (see page 38 for a discussion of how to elicit ideas for intuitive icons).

As part of the general principle of speaking the users' language, one should take care not to use words in nonstandard meanings, unless, of course, a word meaning that would be nonstandard in the general population happens to be the standard use of the word in the user community. Special dictionaries exist to help distinguish common meanings of words from less common meanings. For example, [Dale and O'Rourke 1981] lists 44,000 English word meanings and provides statistics on how many Americans know each meaning. Even though such statistics are only valid in the country where they were collected, one can normally assume that

the avoidance of unusual word meanings will also be a way to improve international understandability of an interface.

To speak the users' language does not always imply limiting the interface vocabulary to a small set of commonly used words. On the contrary, when the user population has its own specialized terminology for its domain, the interface had better use those specialized terms rather than more commonly used, but less precise, everyday language [Brooks 1993]. Even for the general population, specific, distinguishing words are better than bland words.

Speaking the users' language also involves viewing interactions from the users' perspective. For example, a security transactions statement should read, "`You have bought 100 shares of XYZ Corp.`" and not, "`We have sold you 100 shares of XYZ Corp.`" As another example, consider a computer utility, such as a virus guard, that is continuously active, running in the background, but which might periodically need to be deactivated for whatever reason. One approach might be to introduce an "override mode" with a command that could be activated whenever the user needed to perform a task that conflicted with the background utility. Unfortunately, the override would be *on* when the user wanted the utility to be *off*, so using this model would conflict with the user-oriented perspective, even though it might in fact be a perfect reflection of the internal workings of the operating system. A better choice would be a reverse model using a dialog box with a checkbox for "`XYZ-utility active:` ❑." This design also simplifies the interface since it has no concept of a special override mode.

The system should not force naming conventions or restrictions on objects named by the user. For example, users should be allowed to use as long names as they want, even though the system may not always be able to show very long names without scrolling. If the system for some reason cannot handle names longer than a certain number of characters, it should not just truncate without warning the user's input after that number of characters. Instead, it should offer a constructive error message and allow the user to edit the

name to be as meaningful as possible within the limitations imposed by the system.

Given the advice to speak the users' language, an obvious idea is simply to ask users what words and concepts they would like to see in the interface. Unfortunately, the verbal disagreement phenomenon guarantees the failure of that approach: There are so many different words in common use for the same things that the probability is very low that a user will mention the most appropriate name when asked. Furnas *et al.* [1987] found that the probability that two users would mention the same name was no more than 7–18%, depending on the phenomenon being named. Even if one asked several users and then picked the name mentioned by most of them, one would still only match 15–36% of the users. In other words, the majority of users will be dissatisfied anyway, even if words are chosen by asking the users themselves.

A much better alternative is to let the users vote on the names, based on a shortlist of possible names. This list can be generated by several means, including suggestions from the developers, from usability specialists, and from asking a few users. In one experiment [Bloom 1987–88] names for the features in a mail merge facility were chosen in three different ways:

- Technical terms as suggested by the original developers of the system: variable field, token character, record, delimiter, etc.
- The terms suggested by the most users when they were given a short description of the concepts: part, marker, unit, period, etc.
- The winners when users were given a list of several alternative terms and asked to vote: component, placeholder, information package, separator, etc.

Not only do the winners of the user vote seem more descriptive, they also tested much better in a user test. Test users learning the system made 11.1 errors on average when using the original system with the technical terms, 10.3 errors when using the system with the terms suggested by the most users, and only 8.3 errors when using the system with the vote-winning terms. For a second test, only the technical terms and the vote winners were compared. Users made 14.7 errors when learning the system with the technical

terms and only 4.6 errors when learning the system with the vote-winning terms, confirming that the vote-winners made the system easier to learn. However, continued testing after the users had learned the system found exactly the same error rate (2.0 errors) for both sets of names, indicating that people can learn basically anything eventually. The test users were finally asked to perform a new set of tasks with the system without being allowed access to the documentation. During this transfer test, the users of the system with the technical terms made 23.6 errors, whereas the users of the system with the vote-winning terms made only 5.8 errors. This latter result shows that the vote-winning terms enabled the users to understand the system better in that they could generalize their knowledge to correctly use it in new ways.

Given that there are so many ways to refer to the same concepts, computers should allow for rich use of synonyms in interpreting command languages and in documentation indexes. It should also be possible for the users to define *aliases* (user-defined terms that are translated by the system). Not only may an alias be easier to remember for the user who defined it, but it can also serve as a shortcut for complex sequences such as commands with multiple parameters or electronic mail addresses. For some applications, such as the searching of online documentation or database queries, the system itself may build up a list of aliases over time through the use of adaptive indexing [Furnas 1985], where new names for objects are added every time a user tries a query term that is not known by the system, but where the user eventually succeeds in finding the relevant information anyway.

Mappings and Metaphors

A more general way of approaching the goal of a user-oriented dialogue is to aim at good mappings between the computer display of information and the user's conceptual model of the information. Such mappings are not always easy to discover, as exemplified by the case one would naively imagine to be the simplest of all: that of producing a world map [Monmonier 1991].

Unfortunately, the world is round, and the map is flat, leading to all kinds of geographical distortions and the need to select a mapping projection suitable for the user's task.

To discover such mappings, the first step is to perform a task analysis and build up an understanding of the users and their domain. In addition to talking with users and observing them, it is also possible to use more complex methods to build an understanding of the users' knowledge representation and the way they model their domain. Typically, users are asked to list or group concepts in the domain, and the orderings or groupings are assumed to correspond to the users' model of the domain. Some commonly used techniques include ordered recall (users are allowed to freely associate and mention as many concepts as they can think of, with concepts that are mentioned close together assumed to be associated in the user's mind), card sorting (each concept is written on a card, and the user sorts the cards into piles), and paired similarity ratings (users are given a questionnaire listing all possible pairs of concepts and asked to rate their similarity) [McDonald and Schvaneveldt 1988]. The outcome of these tests can either be used directly or be subjected to multidimensional scaling or cluster analysis, using a statistics program. For example, Loshe *et al.* [1991] used card sorting to elicit the users' mental models of a set of graphics and charts. Subjects who were graphic artists were found to structure the graphics significantly differently than other subjects, indicating the need to structure user interfaces to graphics and charting software differently for these two categories of users.

User interface metaphors [Carroll *et al.* 1988; Wozny 1989] are a possible way to achieve a mapping between the computer system and some reference system known to the users in the real world. For example, consider the task of installing or updating new software on a personal computer. The traditional user interface to this task is very system-oriented, listing files, features, disks, etc., from which the user has to choose. Alternatively, a mail-order catalog could be used as a metaphor to structure the interface and allow the users to utilize their existing knowledge about how one selects and previews options [Card and Henderson 1987].

Unfortunately, metaphors may mislead users, or users' understanding of the computer may be limited to those aspects that can be inferred from the metaphor [Halasz and Moran 1982]. For example, the metaphor "a word processor is like a typewriter" will help users discover features like backspace and scrolling, but may prevent them from looking for a global replace feature.

As another example, the operation "delete file" has often been metaphorized in graphical user interfaces, with icons like a trash can, a paper shredder, and even a black hole used to represent deletion. Even though the black hole cannot be said to be very user-oriented, all these icons represent ways to draw upon the users' non-computer-related experience and knowledge and they thus serve as good metaphors for the concept of file deletion. A problem arises when considering data security. Most current operating systems do not delete the *contents* of a file from the disk when the file is deleted. Often, the only result of a file deletion is to make the storage space occupied by the file available for use by other files at a later date. This means that as long as no other files have overwritten the ostensibly deleted storage blocks, it will be possible to read the contents of the deleted file. Users with sensitive data on their disks can therefore not rely on file deletion to safeguard their data in cases where others have access to the disk—for example because it is sold or sent in for repair. The paper-shredder icon may give users a false sense of security due to the connotations of physical paper shredders with respect to the destruction of confidential paper documents. In contrast, the trash-can icon at least implicitly suggests that others might look through the discarded documents.

The lesson from these examples is that one should take care when "speaking the users' language" not to inadvertently imply more than one intended. Specifically, discussions of interface metaphors in manuals should be supplemented with explanations of the differences between the real-world reference system and the computer system. Care should be taken to present the metaphor as a simplified model of a more detailed conceptual model of the system [Nielsen 1990c] and not as a direct representation of the system.

Metaphors also present potential problems with respect to internationalization, since not all metaphors are meaningful to all cultures. For example, a Danish interface designer might choose to use the pause signal as a metaphor for delayed system response, drawing upon the common knowledge that radio stations play a special endless tune of the same 13 notes repeated over and over when one show finishes before the scheduled starting time of the next. However, the concept of a pause signal would be quite foreign to users in many other countries, such as the United States, where radio stations fill every available moment with commercials and would never put on a special signal just to fill up time.

5.3 *Minimize User Memory Load*

Computers are very good at remembering things very precisely, so they should take over the burden of memory from the user as much as possible. In general, people have a much easier time at recognizing something that is shown to them than they have at having to recall the same information from memory without help. This phenomenon is well known to anybody who has learned a foreign language: Your passive vocabulary is always much larger than your active vocabulary. And of course, computers really speak a foreign language as far as the users are concerned.

The computer should therefore display dialogue elements to the users and allow them to choose from items generated by the computer or to edit them. Menus are a typical technology to achieve this goal. It is also much easier for the user to modify information displayed by the computer than to have to generate all of the desired result from scratch. For example, when users want to rename an information object, it is very likely that they will want the new name to be similar to the old one, so the text edit field in which the user is supposed to enter the new name should be prepopulated with the old name, allowing users to make modifications instead of typing everything.

Interfaces based on recognition rely to a great extent on the visibility of the objects of interest to the user. Unfortunately, displaying

too many objects and attributes will result in a relative loss of salience for the ones of interest to the user, so care should be taken to match object visibility as much as possible with the user's needs [Gilmore 1991]. As usual we find that "less is more."

Whenever users are asked to provide input, the system should describe the required format and, if possible, provide an example of legal and sensible input, such as a default value. For example, a system asking the user to enter a date could do it as follows:

```
• Enter date (DD-Mmm-YY, e.g., 2-Aug-93):
```

An even better dialogue design would provide the example in the input field itself as a default value (possibly using today's date or some other reasonable date), thus allowing the user to edit the date rather than having to enter all of it.

There is no need for the user to have to remember or guess at the range of legal input and the unit of measurement that will be used to interpret it. Instead, the system can supply that information as part of the dialogue, such as, for example:

```
• Left margin:   10 points [0-128]
```

A famous example indicating the need to display measurement units to help the user's memory was the positioning of a space-based mirror by the space shuttle *Discovery* [Neumann 1991]. The mirror was supposed to be aimed at a mountain top in order to reflect a laser beam, and the user had ordered the computer to point the mirror toward a point with an elevation of "10,023 above sea level." The user apparently entered the elevation as if it were measured in feet, whereas, in fact, the system used miles as its measurement unit, causing the mirror to be aimed *away* from the Earth, toward a point 10,000 miles out in space.[5]

To minimize the users' memory load, the system should be based on a small number of pervasive rules that apply throughout the

5. With respect to measurement units, other usability principles often lead to a need allow users to select between several alternative units, such as inches, feet, miles, centimeter, meter, and kilometer, depending on their needs.

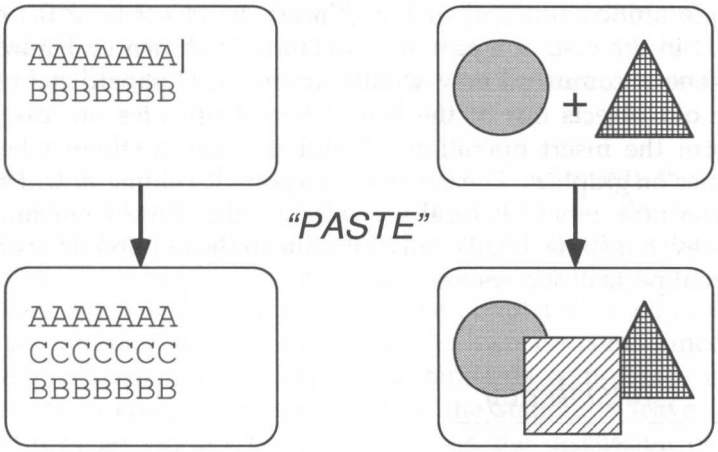

Figure 13 *A generic command: "Paste" can be used to insert a line of* C*'s (text) as well as a striped square (graphics) at the insertion point.*

user interface. If a very large number of rules is needed to determine the behavior of the system, then the user will have to learn and remember all those rules, making them a burden. On the other hand, if the system is not governed by any rules at all, then the user will have to learn every single dialogue element on its own, and it is impossible to predict the behavior of a dialogue element without already knowing (and remembering) how it works.

The use of generic commands [Rosenberg and Moran 1984] is one way to let a few rules govern a complex system. As shown in Figure 13, generic commands make similar things happen in different circumstances, making it sufficient for the user to learn a few commands in order to work with many different types of data. One of the main advantages of generic commands is that they support transfer of learning from one application to the next, since users do not need to relearn those commands they already know [Ziegler *et al.* 1986].

Generic commands need not perform exactly the same function in all circumstances, as long as the user can think of the command as

a single unified concept, such as "insert the object from the clipboard" in the case of a `paste` command. As shown by Figure 13, this generic command may actually insert the clipboard *and* move some old objects out of the way, when it operates on text, but perform the insert operation without moving anything when it operates on graphics. The designer of a generic command will need to determine what "naturally" feels like the same command to users, even if some details will differ due to the requirements of the different parts of the system.

5.4 Consistency

Consistency is one of the most basic usability principles. If users know that the same command or the same action will always have the same effect, they will feel more confident in using the system, and they will be encouraged to try out exploratory learning strategies because they will already have part of the knowledge needed to operate new parts of the system [Lewis *et al.* 1989].

The same information should be presented in the same location on all screens and dialog boxes and it should be formatted in the same way to facilitate recognition. For example, my heating bill contains a comparison between my current heating use and my use in the same month in the previous year, listed as a table with the current year in the left column and the previous year in the right. To facilitate my interpretation of these numbers, a footnote on the bill furthermore contains information about the average temperature in each of the two years. Unfortunately, the footnote mentions the previous year before (that is, to the left of) the current year, thus inverting the relation compared to that used in the table. Consistency considerations would have implied a design of this printout with the same spatial relation between the two periods for both kinds of information. An order where the previous year was mentioned before the current year might be preferred as being consistent with the way timelines work, but unfortunately one can also argue that the reverse order achieves a better match with the user's task of assessing current heat usage. As is often the case in

user interface design, one would have to decide which of these two considerations was most important; once this decision had been made, one should follow it consistently and not mix the two layout rules.

Many aspects of consistency become easier to achieve to the extent that one is following a user interface standard in the design, since the standard will then have specified many details of the dialogue, such as, for example, how to indicate a pop-up menu or which typeface to use in a list of font sizes. See Chapter 8 for a discussion of user interface standards and ways to increase compliance and thereby consistency. Unfortunately, standards compliance is not sufficient to ensure consistency, since the standards leave a fair amount of leeway for the designers. See the discussion of user interface coordination in Section 4.6 (page 90) for ways to promote consistency during interface design.

Consistency is not just a question of screen design, but includes considerations of the task and functionality structure of the system [Kellogg 1987, 1989]. For example, Eberts and MacMillan [1987] found that subjects were more confused when they switched between using a command-line mainframe and a command-line personal computer than when they switched between the command-line personal computer and a graphical personal computer. From a screen design perspective, the two command-line interfaces were very similar, but the underlying operating systems were in fact very different. And the two personal computer interfaces were built on top of systems with the same basic philosophy and features.

A study of a popular spreadsheet program found 10 consistency problems causing common errors for novice users [Doyle 1990]. Seven of these problems were due to inconsistencies between the spreadsheet and the users' task expectations, three were due to inconsistencies between the spreadsheet and other user interfaces, and only two problems were due to inconsistencies within the spreadsheet itself. The spreadsheet's menu navigation method was classified as being inconsistent in all three ways and was therefore counted in all three categories. Of course, other systems may have

different distributions of their consistency problems, but it is probably quite representative that the larger scopes of consistency are the most difficult to get right.

5.5 *Feedback*

The system should continuously inform the user about what it is doing and how it is interpreting the user's input. Feedback should not wait until an error situation has occurred: The system should also provide positive feedback, and it should provide partial feedback as information becomes available. For example, the way to write the German letter ü on many keyboards involves first typing the umlaut, ¨, and then typing the character that is to go under the two dots. Some systems provide no visible feedback as the first part of the character is typed, leading many novice users to conclude that the system does not know how to deal with umlauts. A better design would show the umlaut and then change the cursor in some way to indicate that the system was waiting for the second part of the character.

System feedback should not be expressed in abstract and general terms but should restate and rephrase the user's input to indicate what is being done with it. For example, it is a good idea to give a warning message in case the user is about to perform an irreversible action, such as overwriting a file (see Section 5.9). Assume that the user is about to copy a file to another disk and that the copy operation would overwrite a file with the same name. The worst feedback (except none, of course) would be to state that a file was about to be overwritten, without giving its name. Better feedback would include the name of the file, and even better feedback would include attributes of the two files, such as file type and modification date, to help the user understand whether the copy operation was just replacing an old copy with a newer copy of the same file or whether the file being overwritten was in fact a completely different file that happened to have the same name.

Different types of feedback may need different degrees of persistence in the interface [Nielsen 1987c]. Some feedback is only rele-

vant for the duration of a certain phenomenon, and can thus have low persistence, going away when it is no longer needed. For example a message stating that the printer is out of paper should be removed automatically once the problem has been fixed. Other feedback needs to have medium persistence and stay on the screen until the user explicitly acknowledges it. An example in this category would be a message stating that the user's output had been rerouted to another printer because of some problem with the printer specified by the user. Finally, a few types of feedback may be so important that they require high persistence, remaining a permanent part of the interface. An example might be the indication of remaining free space on a hard disk.

Response Time

Feedback becomes especially important in case the system has long response times for certain operations. The basic advice regarding response times has been about the same for many years [Miller 1968; Card *et al.* 1991]:

- 0.1 second is about the limit for having the user feel that the system is reacting instantaneously, meaning that no special feedback is necessary except to display the result.
- 1.0 second is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data.
- 10 seconds is about the limit for keeping the user's attention focused on the dialogue. For longer delays, users will want to perform other tasks while waiting for the computer to finish, so they should be given feedback indicating when the computer expects to be done. Feedback during the delay is especially important if the response time is likely to be highly variable, since users will then not know what to expect.

Normally, response times should be as fast as possible, but it is also possible for the computer to react so fast that the user cannot keep up with the feedback. For example, a scrolling list may move so fast that the user cannot stop it in time for the desired element to

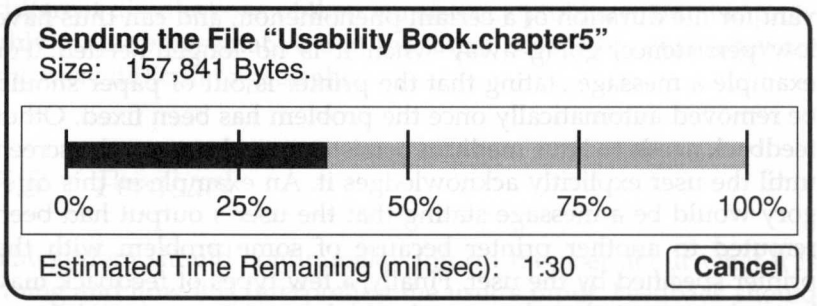

Figure 14 *Percent-done indicator for a hypothetical file-transfer program. The design not only provides feedback expressed in the user's terms (the name of the file) and with respect to the progress of the transfer, it also provides an easy way out (cf. Section 5.6) in case the user gets tired of waiting or discovers that the wrong file is being transferred.*

remain within the available window. The fact that computers can be *too* fast indicates the need for user-interface changes, like animations, to be timed according to a real-time clock rather than being timed as an indirect effect of the computer's execution speed: Even if a faster model computer is substituted, the user interface should still be usable.

In cases where the computer cannot provide fairly immediate response, continuous feedback should be provided to the user in form of a percent-done indicator like the one shown in Figure 14 [Myers 1985]. As a rule of thumb, percent-done progress indicators should be used for operations taking more than about 10 seconds. Progress indicators have three main advantages: They reassure the user that the system has not crashed but is working on his or her problem; they indicate approximately how long the user can be expected to wait, thus allowing the user to do other activities during long waits; and they finally provide something for the user to look at, thus making the wait less painful. This latter advantage should not be underestimated and is one reason for recommending a graphic progress bar instead of just stating the expected remaining time in numbers.

For operations where it is unknown in advance how much work has to be done, it may not be possible to use a percent-done indicator, but it is still possible to provide running progress feedback in terms of the absolute amount of work done. For example, a system searching an unknown number of remote databases could print the name of each database as it is processed. If this is not possible either, a last resort would be to use a less specific progress indicator in the form of a spinning ball, a busy bee flying over the screen, dots printed on a status line, or any such mechanism that at least indicates that the system is working, even if it does not indicate what it is doing.

For reasonably fast operations, taking between 2 and 10 seconds, a true percent-done indicator may be overkill and, in fact, putting one up would violate the principle of display inertia (avoiding flash changes on the screen so rapidly that the user cannot keep pace or feels stressed). One could still give less conspicuous progress feedback. A common solution is to combine a "busy" cursor with a rapidly changing number in small field in the bottom of the screen to indicate how much has been done.

System Failure

Informative feedback should also be given in case of system failure. Many systems are not designed to do so and simply stop responding to the user when they go down. Unfortunately, no feedback is almost the worst possible feedback since it leaves users to guess what it wrong. Systems can be designed for graceful degradation, enabling them to provide some feedback to users even when they are mostly down.

As an example, consider feedback to users of an automated teller machine (ATM). On February 13, 1993, all 1,200 ATMs belonging to a major bank in New York City refused to perform any user transactions for a period of four hours due to a bug in a software upgrade installed at the data center. According to newspaper reports, customers "crisscrossed the city on futile scavenger hunts for an operating cash machine"[6] since they did not know what was going on and hoped that other machines might be working. Since it would be unrealistic to expect a 1,200 node distributed computer

system to function perfectly all the time without any software, hardware, or network failures, the user interface at the individual ATMs should be designed to provide information to customers in case of any such downtime. Different messages should be given, depending on whether the error is due to the central system (in which case customers need not waste time finding another machine) or whether the error is local. In order to inform customers correctly, the ATM needs to be able to perform rudimentary diagnostics, and the entire system needs to be built with such diagnostics in mind. Assuming that the system is designed for it, it should be feasible to give users meaningful information about the likely cause and/or location of any system failures.

5.6 *Clearly Marked Exits*

Users do not like to feel trapped by the computer. In order to increase the user's feeling of being in control of the dialogue, the system should offer the user an easy way out of as many situations as possible. For example, all dialog boxes and system states should have a cancel button or other escape facility to bring the user back to the previous state.

In many cases, exits can be provided in the form of an undo facility that reverts to the previous system state [Abowd and Dix 1992; Yang 1992]. Users quickly learn to rely on the existence of undo, so it should be made pervasively available throughout the system as a generic command that undoes any state changes rather than being restricted to only undoing a special category of user actions. Given that undo and escape facilities are generally available, users will feel encouraged to rely on exploratory learning since they can always try out unknown options, trusting in their ability to get out of any trouble without ill effects. A basic principle for user interface design should be to acknowledge that users will make errors no

6. "At a bank, automatic frustration machines," *New York Times* February 14, 1993, p. 45.

matter what else is done to improve the interface, and one should therefore make it as easy as possible to recover from these errors.

As mentioned above, system response times should be as fast as possible. In cases where the computer cannot finish its processing within the 10-second limit for keeping the user's attention, it should always be possible for the user to interrupt the computer and cancel the operation. In general, interfaces should show a high degree of responsiveness [Duis and Johnson 1990], to the extent that paying attention to the user's new actions should get higher priority than finishing the user's old actions. For example, if a graphics program takes a fair amount of time to repaint the screen, it should allow the user to scroll or to change the zoom level even before the screen has been completely redrawn.

The various exit and undo mechanisms should be made visible in the interface and should not depend on the user's ability to remember some special code or obscure combination of keys. Visibility is of course a general user interface design principle, with the possible exception of some dialogue accelerators, but visibility is especially crucial for exit support since users will need these mechanisms in cases where they are in unfamiliar territory and may be afraid to lose data if they do the wrong thing.

5.7 Shortcuts

Even though it should be possible to operate a user interface with the knowledge of just a few general rules, it should also be possible for the experienced user to perform frequently used operations especially fast, using dialogue shortcuts. Typical accelerators include abbreviations, having function keys or command keys that package an entire command in a single keypress, double-clicking on an object to perform the most common operation on it, and having buttons available to access important functions directly from within those parts of the dialogue where they may be most frequently needed. Pen computers, vertual realities, and some mouse interfaces may may also use gestures as accelerators.

A good example of a shortcut to make a frequent operation faster is the use of a structure generator in a hypertext authoring system [Jordan *et al.* 1989]. Since hypertext authors may often want to generate large numbers of similar hypertext structures with a given pattern of typed nodes and links (for example, each of the courses in an online course catalog might have nodes for course content, prerequisites, instructor, textbooks, and location), they can work faster if the system allows them to define templates of these structures and to generate sets of nodes and links based on a template in a single operation. Macro and scripting facilities can be used to achieve similar effects in traditional command languages, and similar facilities are also being introduced to graphical user interfaces.

Type-ahead (typing the next input before the computer is ready to accept it) is not really a shortcut as such since it still requires the user to generate a complete sequence of input, but it can speed up the interaction by allowing the user to get ahead of the computer and not have to pay attention to all the steps in the dialogue. Similarly, in telephone-based interfaces and other speech-based interfaces, users should be allowed to interrupt the voice prompts as soon as they know what to say. Graphical user interfaces can support a feature similar to type-ahead, in what might be called *click-ahead*: Users can click on the spot where the "OK" button will appear to dismiss dialog boxes before they have even appeared, and they can click in partly obscured windows before they have been made active. It is dangerous to allow type-ahead and click-ahead in all circumstances, however. For example, a critical alert message should not go away without having been visible, and the type-ahead buffer should be cleared in case there is an error in the execution of a prior command which would tend to make the rest of the user's input invalid.

Users should be allowed to jump directly to the desired location in large information spaces, such as a file or menu hierarchy. Often, a hypertext-like approach [Nielsen 1990a] can be used with links between information elements that are likely to be used together. In file systems, such links are often called aliases, since they provide a way to name an information object (file) without having to specify

the full pathname. Alternatively, popular locations may be given easy-to-remember names that have to be typed in by the user. This approach is popular on many videotex services. Finally, users may be allowed to give their own names to those locations they find especially important. By doing so, users can build up a list of bookmarks that will enable them to return quickly to a small set of locations [Bernstein 1988; Monk 1989]. Of course, following the "minimize-user-memory-load" principle, the user should have easy access to a list of the bookmarks defined by that user [Olsen 1992].

Users should be able to reuse their interaction history [Greenberg 1993]. A study of a command-line system showed that 35% of all commands were identical to one of the five previous commands and that 74% of the commands had been issued at least once before [Greenberg and Whitten 1988]. Thus, a simple menu of the last few things the user had done would make it possible for the user to reissue a large number of commands without having to reenter them. Also, word processors, hypertext systems, and other systems where users navigate large amounts of information should have a backtrack feature or other history mechanisms to allow the user to return directly to prior locations.

Even though command reuse is simpler for command-language interfaces, some direct manipulation interfaces allow users to reissue the last formatting command or repeat the last search command by a simple command-key shortcut. It is also possible to use a kind of comic strip to show previous states of a graphical interface as miniatures [Kurlander and Feiner 1992] using a principle called a visual cache to allow fast direct access to those states [Nielsen 1990g; Wiecha and Henrion 1987].

As a simple example of the use of the user's interaction history to provide shortcuts, some applications keep track of which files users often open in those applications [Barratt 1991]. The applications can then offer users a special menu of the files they are most likely to open next, either because they have been used recently, because they are used a lot in general, or because they are normally used together with other files already opened in a particular

session. Statistics on such "working sets" of files that are often used together are slightly harder to get right than statistics on the most recently used files, but they can offer users a convenient shortcut to get at several files in a simpler way than having to find them one at a time in the file system.

System-provided default values constitute a shortcut since it is faster to recognize a default and accept it than having to specify a value or an option. In many cases, users do not even need to see the default value, which can remain hidden on an optional screen that is only accessed in the rare case where it needs to be changed. Defaults also help novice users learn the system since they reduce the number of actions users need to make before using the system, and since the default values give an indication of the kind of values that can legally be specified.

5.8 Good Error Messages

Error situations are critical for usability for two reasons: First, by definition they represent situations where the user is in trouble and potentially will be unable to use the system to achieve the desired goal. Second, they present opportunities for helping the user understand the system better [Frese *et al.* 1991] since the user is usually motivated to pay some attention to the contents of error messages, and since the computer will often have some knowledge of what the problem is.

Error messages should basically follow four simple rules [Shneiderman 1982]:

- They should be phrased in clear language and avoid obscure codes. It should be possible for the user to understand the error message by itself without having to refer to any manuals or code dictionaries. It might be necessary to include internal, system-oriented information or codes to help systems managers track down the problem, but such information should always be given at the end of an otherwise human-readable error message and should be combined with constructive advice, such as "Report this information to your systems manager to get help."

- They should be precise rather than vague or general. For example, instead of saying, "`Cannot open this document`," the computer should say "`Cannot open 'Chapter 5' because the application is not on the disk`" (also following the principle about giving feedback by restating the user's input).
- They should constructively help the user solve the problem. For example, the above error message that a document could not be opened could be made more constructive by replacing the words "`the application`" with the name of the application, indicating to the user what should be done in order to read the document. The message could also offer to try to open the document with some other application that was known to accept data of the given type.

  One useful way of generating constructive error messages is by guessing at what the user really meant to say. In the case of textual input, spelling-correction methods have been available for many years [Peterson 1980; Bentley 1985], and these methods can be especially fast and precise when the set of correct user inputs is restricted to a known set of terms such as the names of files and commands [Bickel 1987]. Durham *et al.* [1983] found that even a simple spelling corrector could handle 27% of all user errors in a text-oriented interface, thus confirming the value of this cheap method. The Interlisp programming system even had a feature called DWIM (**D**o **W**hat **I** *Mean*—not what I *say*) [Sandewall 1978; Teitelman 1972], where the computer automatically performed the action it assumed that the user wanted. DWIM is somewhat dangerous, though, unless the computer is absolutely sure.
- Finally, error messages should be polite and should not intimidate the user or put the blame explicitly on the user. Users feel bad enough as it is when they make errors. There is no need for the computer to make the situation even worse by accusing error messages like the classic "ILLEGAL USER ACTION, JOB ABORTED" (in upper case, at that—screaming at the user). Error messages should definitely avoid abusive terms like *fatal, illegal,* and so forth. Often, error messages can be worded such as to suggest that the problem is really the computer's fault—as indeed it is since the interface in principle ought to have been

designed to have made the error impossible. For example, the LOGO programming language will give the message *"I don't know how to foo"* if the user calls the undefined procedure *foo*, thus seeming to take a little of the blame [Nicol 1990].

In addition to having good error messages, systems should also provide good error recovery. For example, users should be allowed to undo the effect of erroneous commands, and they should be able to edit and reissue previous commands without having to enter them from scratch.

André Bisseret [1983] from the French INRIA research center tells a story about how he tried to define a user ID, giving rise to the following dialogue:

Computer: `Type user name`
Bisseret: `Bisseret`
Computer: `Error, type user name`

Monsieur Bisseret was *not* pleased to find that the computer considered his name illegal. Unfortunately, the computer only accepted user IDs up to seven characters in length, so it could at least have given a constructive error message by explicitly saying so. But actually, a better redesign would have allowed user names of arbitrary length since doing so would follow the principle of speaking the user's language. There is no need to force users to remember strange contractions of their own and other people's names. Such a redesign would avoid any need to have this error message in the first place since the potential error situation would be designed away. Doing so is also a major dialogue principle, as discussed on page 145.

Multiple-Level Messages

Instead of putting all potentially useful bits of information in all messages, it is possible to use shorter messages that will be faster to read as long as the user is given easy access to a more elaborate message. The most common way to implement multilevel messages is to have only two levels and to supplement the short initial message with a button that can be clicked for more informa-

tion. In principle, it is also possible to have many levels of detail, with the navigation between the levels based on some kind of hypertext.

In an integrated user-assistance facility based on hypertext, it would also be possible for the user to link from an error message to the location in the help system that gives further assistance on the problem. If the user's difficulty was not the error situation in general but a single incomprehensible word in the message, it would be possible to link from that word to the location in the online manual where it was defined. And if the user wanted further assistance than could be provided by the help system or the manual, it would be possible to link further, to the appropriate location in the tutorial component, to get a computer-aided instruction lesson.

As mentioned above, error messages should normally not reflect mysterious internal states of the computer that are completely incomprehensible to the regular users even though the information may help specialized support staff locate and fix the problem. The notion of multiple-level messages can provide access to such detailed information for those wizard-level users who want it while shielding less-knowledgeable users from being confused and intimidated. Ideally, it should be possible to dig steadily deeper into a set of messages from lower and lower levels of the system [Efe 1987] until the error has been identified.

5.9 *Prevent Errors*

Even better than having the good error messages recommended in the previous section would be to avoid the error situation in the first place. There are many situations that are known to be error-prone [Norman 1983; Reason 1990; Senders and Moray 1991], and systems can often be designed to avoid putting the user in such situations.

For example, every time the user is asked to spell out something, there is a risk of spelling errors, so selecting a filename from a

menu rather than typing it in is a simple way to redesign a system to eliminate an entire category of errors.

User errors can be identified as candidates for elimination through redesign either because of their frequency or because of their serious consequences. These two kinds of information can be gathered either through user testing (see Chapter 6) or by logging errors as they occur during field use of the system (see Section 7.4).

Errors with especially serious consequences can also be reduced in frequency by asking users to reconfirm that they "really, really mean this" before going ahead with the dangerous actions. One should not use confirmation dialogues so often, though, that the user's answer becomes automatic. If a long sequence of actions is performed so frequently that it is experienced as a unit, the users risk making a "capture error" [Norman 1983] if they ever need to deviate from the sequence: Because they are so used to going ahead in a certain way, they may continue and issue the fatal click on the OK button before they have even read the warning message.

Avoid Modes

Modes [Monk 1986] are a frequent source of user error and frustration and should be avoided if possible. The classic example of modes comes from early text editors, which had separate insert and edit modes. When the user was in insert mode, all keyboard input was interpreted as text to go into the file, and when the user was in edit mode, all keyboard input was interpreted as commands. An example from modern word processors is the use of special annotation text that is sometimes visible and sometimes hidden. Modes basically partition the possible user actions such that not all actions are available at all times, which can be frustrating. Also, modes makes it possible for the system to interpret what is seemingly the same user action in different ways depending on the current mode,[7] which will often lead to user errors. One famous user interface designer once had a T-shirt with the caption *"Don't Mode Me In,"* surrounded by a ring of barbed wire [Tesler 1981] to indicate the frustration of being in one mode and wanting to access a feature from some other mode.

Unfortunately, modes are almost impossible to avoid totally in an interface of some complexity. For example, most word processors have a word-wrap feature that causes text to move to the next line to prevent overflowing into the margin. In fact, this feature introduces modes into the interface, since the same action (typing) may or may not cause a new-line action to occur, depending on whether an "end-of-line" mode is true. Normally, this mode does not bother users who do not mind whether their writing get split over multiple lines. When users *do* mind, such as when constructing tables, this mode does cause problems, however [Monk 1986].

If modes cannot be avoided totally, one can at least prevent many mode errors by explicitly recognizing the modes in the interface design. By showing states clearly and distinctly to the user, a designer can follow the principle of providing feedback, and thus make it less likely that the user will mistake the current mode. In one experiment, adding different sound effects to each of the modes in a computer game decreased the users' mode errors by 70% [Monk 1986]. Even if sound effects are not appropriate, other means can be used to provide mode feedback such as different colors of windows. In my own case, I often connect from my personal computer to two different mainframes that have identical operating systems and look-and-feel. After several cases of getting confused about which system I was currently dealing with, I changed the definition of the windows used for the terminal sessions to use significantly different typefaces, thus providing me with constant low-key feedback.

In addition to having clear status indicators showing the current mode, the interface should also exhibit clear differences between user actions in different modes to minimize the risk of confusing individual interface elements. Similarly, system feedback should be sufficiently varied to provide additional differentiation between

7. Note that modern user interfaces often rely on a kind of mode in the use of window systems: Different user actions in different windows often have different results. This flavor of modes is not as harmful as the traditional modes, because the users do not have to issue special mode-changing commands to move between windows and because the difference between windows will be visually apparent in a well-designed interface [Nielsen 1986].

modes. Mode confusion can also be prevented by the use of so-called spring-loaded modes where the users are only in the mode as long as they actively hold down a button or perform some other action that will automatically take them out of the mode as soon as they let go [Sellen *et al.* 1990].

Even without the added usability problem of modes, one should in general avoid having too-similar commands. In one case, a user had trouble using a certain system and asked for technical support over the telephone. The support person told the user exactly what commands to type, but the user still had problems even after following the instructions to the letter. The support person was unable to diagnose the problem over the phone and finally went to the user's office to make sure that the instructions were actually being carried out. The problem turned out to be that the system required the commands to be typed in lower case and the user had typed them in upper case. This difficulty is known as a "description error," [Norman 1983] since the descriptions of the two situations are almost identical and therefore likely to be confused. In this system, input in the wrong case was simply rejected, but other interfaces may actually act differently on input depending on its case. For example, case-sensitive search is often an option in text editors. Because users can be expected to make description errors very frequently, it is normally preferable to make case-independent search the default and only provide case-sensitive search as an extra feature that has to be explicitly activated by the user.

5.10 Help and Documentation

Even though it is preferable if a system is so easy to use that no further help or documentation is needed to supplement the user interface itself, this goal cannot always be met. Except for systems like automated teller machines where true walk-up-and-use usability is necessary, most user interfaces have sufficiently many features to warrant a manual and possibly a help system. Also, regular users of a system may want documentation to enable them to acquire higher levels of expertise. Even so, the existence of help

and documentation does not reduce the usability requirements for the interface itself. "It is all explained in the manual" should never be the system designer's excuse when users complain that an interface is too difficult.

The fundamental truth about documentation is that most users simply *do not* read manuals [Rettig 1991]. Users prefer spending their time on activities that make them feel productive [Carroll and Rosson 1987], so they tend to jump the gun and start using the system without having read all the instructions. If you doubt this common observation and think that your users do read the documentation, try this simple experiment: visit a few users and place a $10 bill somewhere in their manuals. On your next visit, check how many of the bills are still there! (Of course, you can only use this technique once.)

A corollary to this phenomenon is that if users *do* want to read the manual, then they are probably in some kind of panic and will need immediate help. This observation indicates the need for good, task-oriented search and lookup tools for manuals and online documentation. Since many users rarely use the manual before they absolutely have to, they may not have the manual immediately available (it may have been lost or borrowed by another user), which is one reason for the trend toward supplementing printed manuals with online help and online documentation. Also, online information has the potential for getting users the precise information they need faster than a paper book through features like hypertext [Nielsen 1990a] and good search facilities.

As an example, users of the SuperBook® online information browser found information in an online manual in 4.3 minutes compared to 5.6 minutes for users of a printed version of the same manual [Egan *et al.* 1989]. This result was only achieved after usability testing and iterative design of the online interface. Users of the initial design performed the same test tasks in 7.6 minutes, indicating the value of applying usability engineering principles to online documentation as well as to the main system.

SuperBook is a registered trademark of Bellcore.

The main principle to remember about online help and documentation is that these facilities add an extra set of features to a system, thus complicating the interface just by virtue of existing. Even though it is tempting to design extremely advanced and feature-rich help and documentation systems, the need for an extra "help on how to get help" is an obvious symptom of overblown help design. Many users do not progress beyond the first one or two help screens, and they mostly scan the screens for potentially useful information rather than reading long paragraphs of text [Farrand and Wolfe 1992].

In one empirical study of use of online help [Senay and Stabler 1987], 52,576 help sessions on a mainframe system were logged. 23% of all help requests turned out to be erroneous, meaning that the user did not get any help whatsoever, confirming the observation that help is a system in its own right and can present usability problems to the users. Even in the cases where they did succeed in getting some help information, users only rated it as being useful in 35% of the cases. This study confirms the saying, *"help doesn't"* (or at least, it does not always help, so the system had better be usable even without online help).

Online help has the advantage over documentation that it can be context-sensitive. For example, in a graphical user interface, the user might point to elements in a dialog box to have them explained by "balloon help" [Farkas 1993; Sellen and Nicol 1990]. Also, error messages can be linked to online help with further explanation of the error and possible solutions. In any case, one should remember that the user's problem often is related to wanting to do something *else* than what is offered by the current system state, so it should also be possible for the user to ask task-oriented questions.

Of course, one important aspect of help and documentation, whether online or hardcopy, is the quality of the writing, especially including the structuring of the information and the readability of the text [Klare 1984]. In fact, a major in-depth study of online help concluded that "the quality of help texts is far more important than the mechanisms by which those texts are accessed" [Borenstein

1985]. Even so, teaching technical writing is beyond the scope of this book, so this section concentrates on the access mechanisms and the structuring of the information. See, for example, [Horton 1990] for further coverage of technical writing, and see [Mirel 1991] for a survey of research on the usability of printed documentation. See [Borenstein 1985; Duffy *et al.* 1992; Houghton 1984; Kearsley 1988; Lee 1992] for surveys of online help and [Walker 1987] for more information on online documentation. Perhaps most important of all, the information contained in the text should be correct and reflect the version of the system actually being used by the users.

A second corollary to the finding that users normally do not read the manual is that when they do want to read it, they often will not be able to find it. This problem can be overcome by using online documentation, since it normally stays on the computer once it has been installed. Of course, users have been known to remove documentation and help files when they clean up their disks, so even online documentation does not solve the problem completely.

In a field study of how people really use manuals [Comstock and Clemens 1987], the manuals were found to be stored in many strange locations. Some were nicely stored on bookshelves, though they were not organized by subject matter but rather according to their size and the color of the spines—indicating that it would be a good idea to use these visible features of manuals in a manner consistent with the content of the books. Other manuals were in desk/file drawers, in boxes, in briefcases, on the floor, and on top of computers and terminals. Users typically carried the manuals around, used them standing up, and used several manuals at a time.

A Model of Documentation Use

Users go through three stages in interacting with manuals and help systems [Wright 1983, 1991]:

1. *Searching:* Locate information relevant to a specific need.
2. *Understanding* the information.