CHAPTER 4

MORE IDEAS TO HELP YOU BUILD BETTER ENVIRONMENTS

Purpose of This Chapter

The previous chapter set forth, with little elaboration, the fundamental method for designing Environments. This chapter, and the appendices, provide more elaboration, as well as answers to questions that readers have posed.

Review of What an Environment Is

An Environment is a means for any member of the class of intended users to accomplish certain tasks with a software system. It is therefore a model of the use of a software system. The Environment *captures the use* of the system. Every possible path a user can take through the Environment constitutes every possible way the system can be used. Environment design is the *rationalization of the use* of software systems.

Because the Environment is a model of the use of the system, it is also *a measure of the complexity of use* of the system. Many readers are probably not aware that since the late sixties, computer science has had at its disposal a mathematical definition of complexity. In fact a branch of mathematics called *algorithmic information theory* has arisen out of this definition. The basic idea of the definition is that the complexity (degree of randomness) of a finite binary string (a finite sequence of 1s and 0s) is measured by *the length of the shortest program needed to generate that string*. A random string, e.g., a string we might typically get by flipping a fair coin a number of times in succession, and counting a head as 1 and a tail as 0, has a minimal generating program which is about as long as the string itself. Less random strings, i.e., strings which show a pattern, e.g., 01 repeated, say, a thousand times, have minimal generating programs which are much shorter than the length of the string. (I just gave, in a few words of English, such a program: Repeat 01 1000 times. Even in the most primitive computer language, such a program would be much shorter than the 2000-symbol string itself.)

The idea of a minimal description strongly suggests that there is no such thing as a software system which is *actually* simple to use by a given class of users even though its Environment is *necessarily* very complicated. In other words, programmers and project managers are deluding themselves if they believe that a really *good* technical writer or Environment designer *could* somehow be able to make the use of *any* software system easy and clear.

Documents versus Programs

Sometimes you run across a person who summarizes in a single sentence an idea you have been working on for years. Such was the case with a programmer I once met who, after hearing my description of the Environment concept, replied, "I've always said: a program is not a document, a document is a program." That's the whole thing in a nutshell. A program is not only a set of instructions for the machine to carry out, it is also (no matter how we may try to conceal or avoid the fact) an implied program of its *use* — by human beings. If you want the program to perform the computation you wrote it to perform, you must do certain things first, and, possibly, during the computation. The document which is the program is also a program (usually implied) which you, the user, must follow.

Software-Oriented versus Book-Oriented versus Environment-Oriented Paradigm of Software Use

The software-oriented paradigm of the use of a software system is the one still held by most programmers, engineers, and project managers. It is characterized by the belief that the software comes first, and that the software exists apart from its use; how to use it is a matter to be confronted after the software has been "completed."

The book-oriented paradigm of the use of a software system is the one that is implied by all documentation that uses the traditional book format. It is the traditional paradigm of learning: first learn the subject, then apply it. It is the one that is still universal in schools and universities, in which the student "learns a profession", i.e., takes a series of courses, each of which, ideally, (s)he masters, after which (s)he goes out into the real world and applies what (s)he has learned. It is the paradigm of the courses themselves, in which the student sits in a lecture hall and/or a classroom, and learns how *the subject* is constructed. Textbooks reflect this paradigm: title page, announcing what the course is about, then table of contents, describing how the *subject* is constructed, then preface and introduction (first day preliminaries), then chapter 1, followed, in the case of a technical course, by exercises (has the student understood everything so far?). Then chapter 2 (with exercises), followed by chapter 3 (with exercises), followed by chapter 4 (with exercises) ... followed by a bibliography (providing the student with opportunities for further reading if (s)he

wants to go on to *advanced* study). The book-oriented paradigm still dominates the new documentation technology as reflected in the typical document schemas (Document Type Declarations, or DTDs) for the text structuring language SGML.

The task-oriented paradigm asserts that we build software systems in order to enable people (really: people-plus-software, as described in chapter 2) to perform certain tasks, and that therefore, in the design of software, the *use comes first*. We should design, structure, the use *first*, then make the software match that design. Make no mistake: this is a Copernican revolution in the design of software, and is still considered a radical idea, among technical writers as well as among engineers and programmers! Yet it was set forth clearly in 1976 by one of the world's leading computer scientists:

"In the fifties I wrote for a number of machines of Dutch design what I called 'a functional description' — the type of document that I should encounter later under the name 'reference manual' — and I still remember vividly the pains I took to see that these functional descriptions were unambiguous, complete, and accurate: I regarded it as my duty to describe the machines as far as their properties were of importance for the programmer. Since then my appreciation of the relation between machine and manual, however, has undergone a change.

"Eventually, there are two 'machines.' On the one hand there is the physical machine that is installed in the computer room, can go wrong, requires power, air conditioning, and maintenance and is shown to visitors. On the other hand there is the abstract machine as defined in the manual, the 'thinkable' machine for which the programmer programs and with respect to which the question of program correctness is settled.

"Originally I viewed it as the function of the abstract machine to provide a truthful picture of the physical reality. Later, however, I learned to consider the abstract machine as the 'true' one, because that is the only one we can 'think'; it is the physical machine's purpose to supply a 'working model', a (hopefully!) sufficiently accurate physical simulation of the true, abstract machine." — Dijkstra, Edsger W., *A Discipline of Programming*, p 201. Englewood Cliffs, N.J.: Prentice-Hall, Inc. 1976.

The design of a software system should be the incremental implementation of a means (the software plus an Environment) for the user to perform tasks by using the system — a means that attempts to capture (on-line and/or off-line) *all* user activities required to accomplish the tasks. By imposing this discipline on themselves, software and interface designers always know how complex the system they are building is, relative to a given class of users. And, of course, documentation for such systems reflects the paradigm: there is no traditional table of contents (this is replaced by the

list of top-level tasks), no preface, introduction, chapter 1, chapter 2, chapter 3, etc., but simply the task tree, with an adt index.

The difference between the book-oriented and task-oriented paradigms is so important that I want to illustrate it with a concrete example. I feel this is necessary because again and again I see manuals and on-line help systems structured according to the book paradigm, even though they have various characteristics that at least suggest the author(s) had heard of the term "task-oriented", e.g., characteristics such as the use of gerunds in titles: "Using advanced features of the product," "Understanding the document type definition." Here is the concrete example:

Suppose someone asks you, "How do I get from Berkeley to Palo Alto?" There are at least two ways you can reply: one is by handing the person a map of the San Francisco Bay Area and saying, simply, "Here, follow the map." Another way is to give the person explicit directions: "Get on Highway 80 going south, veer right to the Bay Bridge, go across the Bay Bridge and then get on Highway 101 and follow it for about 30 miles until you see an exit sign saying 'University Ave., Palo Alto'. Turn off at that exit, bear right, and keep following University Ave. to downtown Palo Alto". In the first case, the person has to figure out his or her own route, and, if (s)he doesn't know Bay Area traffic very well, (s)he might choose a route that is much slower than another that an experienced commuter might recommend. The person, in effect, *has to study all of the territory in order to figure out how to make use of part of it.* His or her task is similar to that of a person who is attempting to learn to use a software system from a traditional manual. In the second case, the person benefits from the (presumably) experienced person who gives him or her explicit directions. Of course, if one of the roads happens to be closed, then the second person will either have to proceed by trial-and-error, or by asking someone, or by getting a map. Which is why in a good Environment, every set of tasks that can be performed on a given thing always includes the *optional* task, "Get background material on ..." In other words, "Study the roadmap". The giant step forward, in the case of Environments, is that it is not necessary *always* to study the roadmap first.

To summarize: most — no, *all* — software systems are badly designed because designers and documenters still think in terms of *things* — software, hardware — instead of in terms of the *structure of the use* of things. To software designers, we say, "Don't begin by structuring the documentation. Don't even begin by structuring the programs. Begin by structuring the *use* of the programs, and the structure of everything else will follow!"

Advantages of the New Method

Designers Can Guarantee Their Work

The first advantage of the new method is that Environment designers can guarantee their work.

Guarantee 1: The specified percentage of users can find the information they want within the specified maximum time.

How to test if Guarantee 1 has been met by an Environment: Select users at random from the class of intended users; select tasks at random from those made possible in the Environment; time how long it takes for users to find out how to perform the tasks.

Guarantee 2: All specified tasks are covered by the Environment.

How to test if Guarantee 2 has been met by an Environment: Go down the use-tree, beginning at the Start Menu/Page, compare tasks thus implemented against list of specified tasks. Check the correctness of each implementation (i.e., check procedure) by actual test.

Guarantee 3: Environment contains all reasonable synonyms for each term in minimum vocabulary of class of intended users.

How to test if Guarantee 3 has been met by an Environment: Check exhaustively, or by random selection, against list of reasonable synonyms for each term.

Guarantee 4: All terms not in user's minimum vocabulary are explained in terms of that vocabulary.

How to test if Guarantee 4 has been met by an Environment: Check exhaustively, or by random selection.

Greater Throughput

The second advantage has already been mentioned, namely, in chapter 2, under "Fundamental Concept 2: Structure, or Breaking Complex Things into Simpler Things." This advantage is that of increased throughput. Let us be clear on what *increased throughput* really means. We imagine two identical software systems being used in identical companies by identical classes of intended users. Software system A has an efficient Environment, software system B does not. Now since the systems are identical and the companies are identical, we can assume that, over the life of the software systems, the users will perform identical tasks with the the two systems. We

now compute the total person-time it took to perform *all* the tasks over the life of each system, including training time, manual reading time, time spent communicating on the phone, or via e-mail, with the software manufacturer's customer support facility. My claim is that this total time will be much less for system A than for system B, and therefore, that the manufacturer of system A can advertise the fact, and *make far more money than the manufacturer of system B*, even though both software systems, as far as their actual programs are concerned, are identical!

It is the total person-time that is important — the sum over all tasks performed over the life of the software system, or, as a mathematician might put it (metaphorically), the integral over the life of the system — and not just the execution time, by the cpu, of an instruction.

Users Can Learn New Systems Much Faster

I continue to be amazed that this book wasn't written by someone else ten or twenty years ago, because as long ago as that, it must have dawned on many people that the proliferation of software carried with it a new problem: namely, the problem of users having to learn how to use all that new software before it became obsolete. At the time of this writing, a good documenter of software systems is expected to know at least two or three complex desktop publishing systems — e.g., MS Word, FrameMaker, WordPerfect — plus one or two hypertext systems — e.g., Doc-to-Help, HyperHelp — plus something about the C programming language, plus two or three operating systems — e.g., Unix, Windows, OpenWindows or Motif — plus the software in his or her area of interest, e.g., data base management systems. And each of these software systems is changing every year if not every few months, while new ones are continually arising as contenders to the popularity of the existing ones.

Now the truth is that very few persons except consultants and leaders of training seminars acquire a deep knowledge and understanding of more than one or two of these systems. The truth is that, at least in the documentation field, most users of these systems learn enough to get by — enough to convince prospective employers that they know the systems, enough to get their job done when and if they are hired. It would be extremely interesting to have a histogram of FrameMaker tasks over all users of the system, i.e., a table of all tasks possible with FrameMaker, with, for each task, the number of times each task was actually performed. It would be extremely interesting — and valuable for the software manufacturer — to have such a histogram for *any* software system.

The truth is, there is less and less time to *learn* new software systems, if by *learning* we mean the old process of studying (possibly in a course), memorizing, and

practice-applying. Furthermore, outside of programming, I have never come across a software system which I felt *had* to be learned in this way. The hour-long, day-long exercises in frustrating trial-and-error in order to find out how to perform a perfectly reasonable task are always simply a matter of not knowing a sequence of steps, a sequence of menu selections, never a matter of not having a *skill*, as, e.g., the skill of writing recursive programs. Environments reduce the need for this old-fashioned intellectual labor to a minimum. They enable us to use new systems rapidly. In fact, they make all software systems look the same. Learn one, you know them all.

Scalability

One of the easiest, quickest ways to evaluate a new technical idea is to ask, How well does it scale? In other words, how well does it work when the amount of data in the problem domain becomes very large? If the idea has to be modified as the amount of data increases, then it may not be a good idea. On this basis, the book paradigm for computer documentation is not a good idea, because once the number of pages exceeds several hundred (which it does in the documentation for most modern computer systems), it becomes impractical for a user to follow the read-learn-apply model. The Environment idea, on the other hand, scales very easily. In fact, no change whatever is required in the structuring or rules of use for an Environment as the system grows. The user never has to read more than (s)he needs to perform the current task, and never has to memorize where information is located.

Ease of Revision

Closely related to the criterion of scalability is the criterion of ease of revision, which says that, when evaluating different approaches to documentation, it is always wise to ask, How easy is it to make revisions? The reply in the case of Environments is, very easy indeed, because it is always clear where existing information is and where new information must go. The rigorous task-orientation (which is simply a duplicate of the well-known computer science technique called *structured programming*) means that, even if a given task is used as part of many different other tasks, the instructions for that task (the sequence of sub-tasks that implement it) exist in one and only one place. The Environment designer or maintainer never has to ask, Now, in how many different places in the documentation does this task appear? In what places do I need to make this change? Or, to put it in more abstract terms, we always know the *coordinates* of a piece of information.

Disadvantages of the New Method

• Alphabetical organization of paper implementations makes some users uneasy.

The most frequent criticisms I have received of the alphabetical organization (as, for example, in appendices A and B) are "It's too hard to read," "It doesn't flow," this despite a careful explanation in the first-time user information referenced on the Start Page that, unlike manuals of the past, this one you don't have to read before you use; that you simply look up things when you need them, that this is just-in-time learning.

The attempt to organize procedures alphabetically has given me a first-class lesson in the difficulty which most people have in confronting what is now known as a *paradigm shift*, meaning a major change in the way they think about a given subject or task. A number of people have even commented that subtitles such as "file, copying a," or "data, inputting," don't look right because they contain commas (!). This is how deeply oriented to syntax many people in technical fields are.

• Complicated procedures require lots of page-turning in paper implementations.

Complicated procedures are procedures with several subtasks (just as, in structured programming, a complicated program is broken down into subprograms — subprocedures). I'm afraid there is nothing that can be done other than rewriting the software, unless, as in the past, you want to try to pretend that the procedure is really simpler than it is by, for example, not giving all the steps ("the user will understand").

Also, repeated use means gradual memorization, so that repeated use means less need to look up implementation of tasks, resulting in faster execution of the task. Which is precisely how speed should come about: not because experienced users have somehow figured out how to do the task, or have somehow discovered where the procedure for the task is in the Environment, and beginners have not, but because experienced users need to spend less time looking up information in the first place.

The equivalent of page-turning in on-line implementations — e.g., the look-up function in hypertext — usually goes much faster.

• Creating this new type of manual (or on-line system) is often tedious, requires greater attention to detail, in particular to cross-referencing.

At present, this seems to be unavoidable, although software systems are already appearing that make the development of on-line, as well as off-line (paper) Environments, much less tedious.

Exercises for the Skeptical

The following exercises are for readers who have grown so accustomed to their chains that they are unaware of them. I am referring, of course, to readers — in particular, technical writers and programmers — who believe that, on the whole, especially now that we have desktop publishing systems and hypertext and Graphical User Interfaces (GUIs) and multimedia, most software is easy to use. All of the following exercises come from real life: specifically, from my own on-the-job experience. None was thought up in advance as a test of a system's documentation. All, I think you will agree, concern tasks which should not require cleverness, in fact, any thinking at all, to find out how to do.

Exercise 1: Word spacing in FrameMaker

Background: In FrameMaker 3.0 for the PC, there are two types of inter-word spacing in paragraphs: automatic and manual. In the former, FrameMaker automatically computes a pleasing word spacing throughout the paragraph; in particular, it does not permit you to enter a succession of blank spaces between words. In the latter, it does let you enter blank spaces between words.

Exercise: Find any user of FrameMaker who doesn't know how to set the above word-spacing options. Give him or her all the FrameMaker documentation, then open a document, select a paragraph style in the document, and ask the user to change the option. Time how long it takes the user to find out how to do this with, of course, no help except the documentation.

Answer to procedural part of exercise: In the Format menu, select Document..., then check the Smart Spaces box for automatic spacing, uncheck it for manual.

Exercise 2: Finding what directories are on what hard disks in Unix

Background: Many Unix systems are connected to a network. On the network there are servers and clients (e.g., workstations). Each server and each workstation has one or more hard disks on which directories (groups of files) are stored. It is

sometimes necessary to find out which hard disk, hence which server or workstation, a given directory is stored on. In other words, you would like information which says, in some comprehensible form, "The hard disk whose ID is *x* is on the server/machine named *y* and contains the directory(s) *z*."

Exercise: Find any user who doesn't know how to find this information. Give him or her all the Unix documentation, log on, and ask him or her to find all hard disks and directories (top-level representations are adequate) on all servers and workstations which are accessible from the workstation being used. Time how long it takes the user to find out how to do this.

Answer to procedural part of exercise: To find which hard disks contain which directories, use the df command. As for the names of servers and machines that contain the disks, there seems to be no way to do this. No Computer Support department I have asked has ever been able to give me any answer beyond "you just have to know."

Exercise 3: Erasing a backed up disk in MS-DOS

Background: You have several disks containing backups of a certain directory on your PC. These backups were made using the backup command in MS-DOS 5.0. You have just made a fresh backup and want to reclaim the other disks for other use. You have tried simply reformatting each disk, but for some reason the formatter refuses to format the disks. You can't delete the individual files on the disk because they are not listed when you do a dir command on the disk. How do you erase the disks?

Exercise: Find any user who doesn't know the answer to this question, give him or her all the MS-DOS 5.0 documentation, plus any other documentation you wish, and time how long it takes him or her to find out how to erase the disks.

Answer to procedural part of exercise: I don't know the answer.

Exercise 4: Inserting page numbers in a FrameMaker document

Background: You have been asked to work on chapter 4 of a book being written in FrameMaker (FrameMaker for the PC, as in Exercise 1). The chapter has no page numbers. You are to make page numbers of the form 4-*n* , *n* beginning at 1, centered at the bottom of each page, in the same typeface as used for the body of the document. This numbering is to appear in the document itself, not merely when it is printed out as part of the entire book.

Exercise: Find a FrameMaker user who doesn't know how to perform this task. Time how long it takes the user to find out how to make the page numbers appear as requested.

Answer to procedural part of exercise: In the Page menu, select Master Pages. Scroll to the footer box at the bottom of the page. Select the entire bar, including the End of flow symbol. In the Paragraph Catalog, click the format of body paragraphs in the document (e.g., "Body"). Now, in the Format menu, select Paragraph... . In the dialogue box that appears, change Alignment to Center if necessary. Click Apply and exit the dialogue box. The End of flow symbol should now be in the center of the footer box. Place the insertion point immediately to the left of this symbol and type "4-." In the Special menu, select Variable... In the Variables: list, select Current Page #. A "#" should now appear immediately following "4-". In the Page menu, select Body Pages. The page numbers should now appear as required.

Against Release Notes and Readme Files

If you have understood the Environment concept so far, then you probably can understand why I think Release Notes and Readme Files shouldn't really be necessary: these last-minute modifications to the Environment (for that's what they really are: changes in the Environment occasioned by the last-minute discovery of bugs or of accidental omissions in software functionality) — should be entered where they belong, namely, at the proper location in the Environment. Installation instructions, of course, must typically exist outside on-line Environments, since using an on-line Environment assumes that the system has been installed. But Release Notes and Readme Files reflect the book paradigm that continues to govern the documentation thinking of engineers and programmers. The assumption here is that users of a system first want to learn about it, that they will want to first read the Release Notes and the Readme File and at least part of the manual and *then* (having absorbed all that information) they will be ready to use the system. Not at all necessary with a good Environment!

Why I No Longer Give Opinions On Manuals Until I Use Them

Technical writers like to show other writers the manuals or on-line help systems they have written: a perfectly natural desire. In fact, each year, in many parts

of the country, there are exhibits of manuals and other technical documentation, with prizes awarded for those which a panel of judges deems the best. But I say that no writer and no judge can properly evaluate a piece of documentation without using it. Typeface, layout, and other characteristics of visual design, and yes, even prose style, are *as nothing* as far as the real, long-term value of the documentation is concerned, compared to how fast the typical user can find out what (s)he wants to find out. Yet speed of access cannot be measured by a mere few minutes' inspection unless: (1) the product is available for use; (2) the person who is doing the inspecting has a clear idea of what the tasks are that the product makes possible; (3) the person actually tests to see if the procedures implementing the tasks are complete (do they give the user *all* the information (s)he needs, or is (s)he assumed to have read other unnamed parts of the documentation?).

So now, when someone shows me a manual or on-line help, I try to be as honest as I can in my response: "It looks very nice, but I hope you will understand that I can't say anything more about it until I use it."

The Final Arbiter in All Arguments About Usability...

...is the user, or, more precisely, is a test on randomly selected users. Everyone agrees with this, I know, and I wouldn't even bother to mention so obvious a fact, except that even at this late date the rule continues to be endless discussions carried on among technical writers about what the user will want and what the user will understand. Let's be honest: it's much easier, and a lot more fun, to have endless meetings on this subject — especially if, as is the case with the vast majority of technical writers, the participants haven't a clue what it's like to have to use, on a daily basis, the type of software system for which the documentation is being prepared — than it is to come up with a prototype and dive into the boring, uncomfortable, often embarrassing labor of selecting a few users at random and trying out the alternatives on them (and for the writers to start trying to perform actual tasks with the system). Most meetings on what the user will want amount to little more than high-paid socializing.

There is no reason why any potential buyer (or user) of a computing system should take the manufacturer's word for it that a system is easy to use unless the manufacturer can demonstrate its claim through actual tests on new users who use the system with no other aid but what is provided by computer screen, keyboard, and

manuals. In fact, let me go further: I think it is as irresponsible for a manufacturer to sell an untested Environment as it is to sell untested software or hardware.

There is Nothing Magic About On-Line Documentation

The computer industry runs on innovation, which is often confused with new technology, which in turn is often confused with new buzzwords. I sometimes think there is entirely too much technology in the hands of people with too little understanding of what the central issues are. Despite all the windows systems, graphic displays, multimedia, the fact is that it still takes a training course to learn how to perform the most basic functions on most of these systems: move files around on a network; put a bullet at an arbitrary place in a text, get data from a database.

The way to combat all this confusion is via healthy applications of *What* versus *How* thinking, i.e., by asking yourself, What *task* does this innovation enable me to perform? How often do I need to be able to perform the task? Are there other ways of performing the task? How do they compare in cost, including the cost, in time, of learning to perform them?

Most programmers and project leaders seem to think that putting documentation on-line in itself improves the usability of a product. And if an on-line Help facility improves usability, why, then, an on-line Hints facility will improve usability even more!

The way to keep these impulses under control is simply to say to those who are subject to them: Show me, in the Environment, how the user learns when to use on-line Help, when to use on-line Hints, and when to use the off-line documentation. Show me that the user will never have to *figure out* which to use. Show me how each of these increases the speed at which the user can find out how to perform tasks.

"If They Can't Use the System, Just Keep Writing More Manuals"

It is no exaggeration to say that this is the rule followed by the vast majority of companies. Technical publications departments are certainly not inclined to protest

against it, since it provides job security. Furthermore, technical writers seldom if ever are users of the products they write about, so they have no idea of the frustration involved in having to search through five or ten manuals, all with poor indexes (at best), to find an elementary piece of information. And so we get a *Getting Started Manual*, and a *User's Manual*, and a *Reference Manual*, and a *System Manual*, and an *Applications Manual*, and a manual on each of the major components of the system — in many cases, over a thousand pages of information, full of the same things repeated in slightly different wordings, all produced on the dim assumption that if a few words don't do the job, clearly, what is needed is more words. But what is needed is not more words, but a strict, easily measurable criterion of success like look-up time, and the discipline to meet that criterion.

A Shameful Deception

I once saw a well-known computer scientist demonstrate a new office-automation system to a group of secretaries. A secretary was seated at the terminal, her task being to type a business letter. The computer scientist sat at her side.

Computer Scientist: Now what do you want to do?

Secretary: Well, I want to type the date and make it flush right.

CS: Simple. Press Control — that's the key marked CTL, then keep holding it down and press R — no, don't worry about the Shift key. There. Now just type the date.

Secretary types the date, which remains flush right as she proceeds.

CS: There, wasn't that simple?

Secretary nods.

CS: Now all you have to do to clear flush-right mode is to press CTL, R again. Go ahead, just like before.

Secretary does it hesitantly.

CS: See how simple that was? And notice how the screen always shows you exactly how your letter looks.

Enough said.

How to Make Nothing Ever Weigh Too Much

People sometimes ask me why I insist that Environments be tested with no instructional material available to the user except that contained in the Environment itself. Why don't I let the user ask questions of knowledgeable persons? Why not offer a course? (Whenever you hear someone say, "We'll offer a course!" you can be sure it's the same old hamburger stand as far as Environment design is concerned.) The reply to these questions that always comes to mind is the following:

How to Make Nothing Ever Weigh Too Much: (1) Put the thing on the scale. (2) If it weighs too much, put your hand underneath the scale and push up until it weighs the right amount.

Features

Whenever I see months of meetings taking place about the design of a new computing system — "Well, the user will want to to add a feature that will ..." — I always want to tell the participants, "One way or the other, your future user will have to retrace all these discussions, because there's no other way he'll ever find out what you've put into the system!"

Every feature comes with a string attached (pun intended), namely, the sequence of instructions which must be added to the Environment in order to guarantee that every member of the CIU will know:

- that the feature exists whenever he or she needs to know it.
- how to use the feature.
- how to view the results.
- how to change the feature (or have it changed).

Nowadays it is often easier to add a feature to a software system than it is to attach the above string.

What Is a Command?

We still think of a command as a sequence of letters and/or digits we type on a computer keyboard, and, nowadays, we believe that windowing systems have vastly reduced the need to type commands. But the proper way to think about commands is to think of each command as a sequence of atomic operations performed on the computer: the atomic operation may be the pressing of a key, or it may be the clicking of a mouse button. Thus, for example, the sequence of menu selections involved in printing a document is the entry of a *command*. The sequence of traditional commands and menu selections that are often required to resume yesterday's work in a workspace is a *command*, one for which one might like to define a macro, i.e., an abbreviation for the name of the command. Windows facilities did not reduce the number of commands that computer users have to use, they simply made their entry much faster.

Macro (Script) Facilities

A macro, or script, facility — i.e., a feature that enables us to have sequences of commands executed automatically — is not a feature that may or may not be added to the Environment as the system grows; it is an inherent part of the design and use of an Environment because it arises from our natural desire to abbreviate things we do repeatedly.

The best way to think about macros is to think of an Environment as being made of a set of primitive functions plus a means for combining functions to make new functions. This means for combining functions is the macro facility.

Expert versus Non-Expert Users

The concept of an efficient Environment enables us to specify simply and precisely the difference between beginning users and expert users (all of whom must be in the CIU, of course). With an efficient Environment, the difference is *not* that

experts have somehow learned how to get the system to do what they want it to do, and inexperienced users haven't — *every* user knows how to carry out every task provided by the Environment: that's the whole purpose of an Environment! The difference is that expert users can get what they want done *faster* than inexperienced users *because* they do not need to do as many look-ups, having memorized more of the procedures for carrying out tasks. *That* is the difference.

A sure sign that a technical writer hasn't grasped the Environment concept is when (s)he talks about whether this or that piece of documentation will help the user to "learn" the system. *Learning is a red herring!* Don't allow it to lead you astray! One of the primary advantages of an Environment is that it reduces to a minimum the amount of learning that a user has to do in order to use the system. Learning is a *side-effect* of using an Environment. The question, Have you learned to use this system? should, if the system has a good Environment, be greeted with a reply of the sort, Give me a few typical tasks, watch how often I need to use the Environment to find out how to do them, and then judge for yourself.

Against Cleverness

Even at this late date, the excellence of a programmer is still a function of his or her cleverness — meaning, here, not ingenuity at devising algorithms, but knowledge of obscure, usually undocumented facts about the computer system he or she is using, this knowledge often marking the difference between success and failure in getting the day's work done efficiently. Let it be said that there are men and women who seem to have an extraordinary ability at rapidly learning and retaining such knowledge, and they are invariably described as outstanding, even brilliant, programmers. Given today's computing Environments, they are. They deserve every dollar of their high salaries. The question is (as always), Is there a better way? *Should* it be necessary to have this knowledge to accomplish one's daily work efficiently? If so, is it really necessary that this knowledge be obscure and inaccessible to users who have not paid the expensive tax of learning it by "playing with the system"? Keep in mind that we are not talking about *skills* here — e.g., the *skill* of recursive programming, which is an entirely different kind of knowledge — but rather knowledge which boils down to sequences of commands: If you want to do *z*, just enter *w*, *x*, *y*.

There was a parallel to the present requirement of this kind of cleverness in the early days of compiler development. It was then believed that only (human) Assembly language programmers could ever handle the difficult task of programming. A program might be able to substitute a correct sequence of Assembly language commands for a given high-level program statement, but no program could know how to *optimize* such a correct sequence, i.e., transform it into another sequence that would perform the statement's function more rapidly. Only a human could do that. Well, the results are before us, and nowadays, optimizing compilers are the rule rather than the exception, and only in rare circumstances is it necessary for a human to improve on the compiler's commands.

Having said all that, I must also remind the reader that no-cleverness Environments do not come free. Designers of after-the-fact Environments must somehow track down the clevernesses and enter them into the proper sub-Environments, a task which in all likelihood will last for the remainder of the system. Furthermore, the Environment itself must make such constant updating relatively easy. Furthermore the designer must find a way to announce, in all appropriate places in the Environment, when such changes in procedure have occurred, since expert users will not normally look up procedures they have since memorized.

School or Tool?

I once complained to a programmer about the difficulty of finding the most elementary information in the Unix documentation — or even of finding whether this information existed (see "Exercises for Skeptics" above). He replied that I should keep in mind that the version of Unix we were using was developed in a university environment, namely, the University of California at Berkeley, and that one of the things that graduate students and professors are under continual pressure to do is demonstrate their exceptionally high intelligence. What better way to do this than to design software that only the special few will be able to use efficiently *from the first day* on? (Otherwise, what will separate us from the rabble?)

But every product designer and Environment designer must ask him- or herself on the first day of his or her career and on every day thereafter, *What business am I in?* Am I designing a school or a tool? Is my job to separate those who know from those who don't? Am I in the business of testing my customers' intelligence and memory and cleverness? Am I in the business of testing my customers' knowledge of the latest

software culture? Am I in the business of keeping the Training Department busy? Or am I in the business of enabling my customers to get certain kinds of work done faster than they can with the competition's products?

What's All That Stuff On the Screen When I Turn My Computer On?

If you use computers on a daily basis, then almost certainly you get a daily demonstration of how far computer manufacturers are from understanding, much less delivering on, good Environment design. I am referring to the screensful of hardware and software facts you get when you turn on your computer. On my PC — an excellent IBM clone, incidentally — which I bought in early 1993, I get several screens full of all sorts of information I (a) don't need, (b) don't understand, and (c) could not understand completely without spending time searching through manuals and, probably, having to call a knowledgeable friend as well as the manufacturer.

The only things on the screen at any time in an efficient Environment are things that represent choices the user might make to carry out his current task.

This does not mean that the information on the screen when we turn on a computer is not useful, or should be made inaccessible to the user! It means only that it should be made accessible when, and only when, the user needs it to carry out a task, e.g., making more memory space available for running programs in, fixing a problem with hardware or software, etc.

Despite all our windows facilities, graphics user interfaces, and multimedia, we are still in kindergarten when it comes to understanding basic rules of Environment design like this one.

Straight Thinking about Explaining Buttons, Menu Selections, Commands

A further indication of our present primitive state of Environment design is the universal practice in computer documentation of explaining what buttons and other menu symbols, and commands, *do*. This is a programmer's view of software: he or

she has worked hard to write the programs and translate their control into actions performed on the keyboard and/or on the screen (more precisely, in the window). What could possibly be more natural than for the programmer to want to tell the user all about his or her work, all about what his or her software *does*, all about what the controls *do*? But as a user I don't care what the software does or what all the controls do! I only care about getting my job, my tasks, done as rapidly as possible. Why must I be compelled, *first*, to learn all the buttons and menu choices and commands, and *then*, every time I want to carry out a task, be forced to map that knowledge into the proper sequence of activities? Why must I have to pay this continual *tax*, in time and intellectual effort, to the programmer's pride in his or her work? *Why must I have to know everything in order to do something?*

Exercise: Take any Windows desktop with one or more icons, or any pull-down menu, and write down all the tasks that are possible from that desktop.

This exercise suggests a *rigorous* way of thinking about the functionality available in a given window. Consider a table, call it Table 1, which consists of, on the left, a list of all the buttons and menu choices available in a window, and, on the right, for each such action, an informal expression of the effect of the action on the left. (In some cases, the effect will simply be the invoking of a dialogue box offering further choices.) Now consider another table, call it Table 2, which consists of, on the left, all *sequences* of one or two button clicks and/or menu selections, with repetitions allowed, and on the right, for each such action, an informal expression of the effect of the action. Define Table 3, Table 4, ..., similarly. Now this sequence of tables in fact defines the total functionality of the window. Reading a table from left to right provides us with a map from button clicks and menu selections to tasks. Reading a table from right to left provides us with a map from tasks in the window to sequences of button clicks and menu selections implementing each tasks. Such a map, judiciously pruned and presented in a pleasing manner, is precisely what an Environment provides.

I should point out here that GUIs — Graphical User Interfaces, e.g., windows — are slowly evolving toward task-orientation, witness the appearance of browsers, i.e., GUIs which are designed to make it easy to perform the most common tasks associated with a Thing, i.e., create, modify, delete, list (current instances of the Thing).

Let me conclude with a concession to current ways of thinking: at present it seems that users are still not ready to think from task to sequence of actions in a window. Therefore, at present, I think it is also necessary to continue to provide maps from buttons and menu selections to tasks, as in the past.

The Inside, True, Story on Tutorials

"I don't know about you, but I'd rather have all my teeth removed without anesthetic than follow the tutorial in most manuals. They're the equivalent of being strapped into a chair and forced to listen to scales for three months, then 'Three Blind Mice', then 'Twinkle, Twinkle, Little Star', then Lawrence Welk — all under the guise of teaching you music appreciation." — Naiman, Arthur, et al. *The Macintosh Bible*, 4th ed., p 45. Berkeley, Calif: Peachpit Press, 1992.

Environments make clear for the first time why tutorials are so boring and, actually, unnecessary. *Traditional tutorials have been a colossal case of missing the point, of not grasping what is really going on*. A traditional tutorial is an attempt at showing the user how to use the system, which is, of course, precisely what an efficient Environment does. The real flaw in traditional tutorials, the fundamental dishonesty in them, is that they never tell the user how (s)he *should find out how to do what the tutorial tells him or her to do*. Traditional tutorials are a cheat in this respect, a game that is fixed in advance, in the same way that the keyboard demonstration described above under "A Shameful Deception" was a fixed game. A traditional tutorial says, If you know how to find out how to do what you want to do, here is how you do it. Very boring, indeed.

Training courses are even worse. They amount to sitting in a classroom and having the trainer, in effect, read the phone book to you.

Given an efficient Environment, tutorials and courses are all but unnecessary. They can, of course, be included as a comfort to users who are made uneasy by new things (even if the new things work much better than the old), but a tutorial for an efficient Environment will seem rather silly — a demonstration of the obvious, which is exactly what it is.

Environments and Architecture

As I pointed out in chapter 1 under "The Birth of a New Profession", an Environment is analogous to a building. We move around in an Environment in the same way we move around in a building.

Essential reading for every Environment designer is Christopher Alexander's *Notes on the Synthesis of Form* (Harvard University Press, Cambridge, Mass., 1970).

This book, though about architecture, was one of the anticipations of structured programming.

Given the existence of this book, it is amazing to me that there is no recognition in computer science — at least none that I know about — of the important parallels between the architecture of buildings and the design of computing systems. I do not mean parallels between the architecture of buildings and computer architecture (design of hardware to perform logic functions). I mean parallels between building architecture and what we should call the *use architecture* of computing systems. Use architecture refers to what the user has to do — what physical and mental actions he must perform — in order to accomplish certain functions on a given system. Environments correspond to rooms or floors or spaces in physical architecture.

We still occasionally hear arguments for modeless systems, and perhaps this is understandable, considering the extraordinary ineptitude with which mode-based systems have been designed in the past. Nevertheless, if by a *mode* we mean a cluster of related activities — i.e., an Environment or sub-Environment — I do not see how we can seriously believe that a modeless system is desirable, any more than a completely general room or completely general building is desirable ("all things to all people"). Such a system would mean that every function the user could perform on the system was at the same "distance" from every other one. In programming, the analogy is completely unstructured programming — the programmer is allowed to transfer control from any instruction or statement to any other — and the undesirability of that practice has long been recognized.

"For software to be easy to use, it should be hierarchically organized. This means that most basic operations are simple and central to how the program works and the more advanced operations are off to the side, so you don't even know about them until you need them." — Naiman, Arthur, et al. *The Macintosh Bible*, 4th ed., p 41. Berkeley, Calif: Peachpit Press, 1992.

Environments, Ships and Aircraft

Another metaphor for software systems is ships and aircraft. *Weight* in ships and aircraft is analogous to *complexity of use* in computing systems. Hulls of ships, and fuselage and wings of aircraft, are analogous to Environments of computing systems. *Weight* in ships and aircraft is analogous to *complexity of use* in computing systems. Captains and pilots are analogous to users. The key point is that everything

the captain or pilot needs to run his or her ship *must be on the ship*. (You could, I suppose, argue that the captain or pilot could receive all instructions from a distant control facility via radio and/or television, but that is certainly not the normal situation.) Furthermore, each additional "feature" (capability) added to the ship carries with it a price in weight: not only the weight of the feature itself, but the weight of the controlling mechanisms. If it were decided that the ship should be usable by a less skilled person, there would need to be a corresponding increase in the hardware (weight) needed to translate that person's actions into the appropriate actions of the controlling mechanisms. Hence the environment (hull, or fuselage and wings) must be changed to accommodate this additional weight.

"Fill-In-the-Blanks" Environments and Semi-Automatic Environment Generators

Experience with the (after-the-fact) implementation of many Environments suggests that Environments are not as different as current practice suggests. Indeed, I now carry around a mental "fill-in-the-blanks" Environment for each of the computing systems I most frequently use (word-processors and integrated-circuit computer-aided design (CAD) systems), and for each such system, simply fill in the actual commands I need. In a certain important respect, all word-processors are the same; one way or the other, they provide the same fundamental set of functions: typing of text, erasing of text, moving text around, storing text, retrieving text, printing text, etc. Similarly, all CAD systems are "the same", all data-base systems are "the same".

I think we may look forward in the near future to the development of Environment *generators* — i.e., programs taking the description of an Environment as input, and yielding as output an easily modifiable framework, in a given language, into which the program pieces that implement the various functions can easily be inserted.

Environments are Data Bases Whose Content Is the *Use* of Software Systems

Environments, and mechanisms that help generate Environments, such as those described in the previous section, are really *data bases* — data bases whose content is the *use* of software systems. It is important to keep this in mind as you engage in potentially long-winded discussions about Environments. "Tell me all the tasks I can perform on the Thing *x*" is an example of a data base *query.* So is "Tell me the meaning of *y*." So is "Tell me all the tasks that are directly performed on, or with, or to, the Thing *z*." So is "Tell me all the tasks of which task *w* is a constituent." So is "Tell me all the tasks which contain the verb *v* or a reasonable synonym thereof." Usage data bases (as opposed to financial or manufacturing or human resources data bases) is where the Environment concept is heading. Fortunately, the technology already exists, and, in fact, is more than adequate for the purpose.

Against Cognitive Psychology, Learning Theory and Other Opiates

Readers of this book with a strong liberal arts background invariably point out to me that I have overlooked the benefits of cognitive psychology and learning theory in making software easy to use. I haven't overlooked them; I have deliberately omitted them because I think they are a complete waste of time at the present state of the art. The names are certainly impressive. You can get a Ph.D with some bit of fluff about their application to pleasantness-of-use. With that Ph.D you might be able to get a job in a major corporation, with your own office, and a staff of similarly soft-minded professionals. And after you are done, it will still take week-long training courses for users to learn material that is fundamentally look-up-able and shouldn't require a training course at all. In all likelihood, you will remain as ignorant of technical concepts, including those that make products work, much less those that govern the actual use of the product, as you were when you first set foot in graduate school. Worst of all, you will be proud of the fact.

I have never come across an idea in these disciplines that I didn't think was obvious from common sense. I have never seen, heard of, or read about any kind of usability testing in the computer industry which had the slightest statistical validity. So what, exactly, have these disciplines brought us except the illusion that we are working in a scientific profession?

Why Progress Has Been So Slow

Until the eighties, there was no real progress in documenting computer systems. Manuals were written in the traditional format: the user was expected to read the manual, learn how the system worked, then apply this knowledge to getting the system to do what the user wanted ("Read, learn, use") — the same approach that had been used for manuals for *any* kind of technical product, as well as for textbooks for any kind of technical subject.

In the eighties, the important concepts of task-orientation and usability testing began to take hold in the documentation world, although some of us had been attempting to get a hearing for these ideas a decade before. Neither of these concepts is difficult. Their benefit is, I think, obvious to anyone who is capable of asking him- or herself a few basic questions about the design of computer systems, e.g., What is the purpose of a computer system?, How can that purpose best be achieved?

But computers have been commercial products since the fifties. Why has progress in documenting their use been so slow? I think the following are some of the answers.

(1) Programmers and engineers are technology oriented. For them, the central reality is the computer program or the computer itself. The *use* of the product is a secondary matter which, they believe, follows more or less obviously from the construction of the product itself. Furthermore, being technology oriented, programmers and engineers naturally see the solutions to problems posed by programs and machines to lie in more technology. Thus, I have heard project managers proudly boast that their new product will solve all the usability problems posed by previous products of the type, because the product will have four-color menus, the naivete of which I hope is clear to any reader of this book.

Those who think in terms of technology think in terms of *syntax* (the *How*). They tend not to see the forest for the trees. For them, an increase in speed, or the introduction of windows, or the increase in number of colors available on the screen, all make new products different from old products. Functionality for them tends to mean number of features. They continually confuse *usability* with *pleasantness of use*.

Some of this narrowness is, let us admit it, understandable in a marketplace economy. What sells is what is new and different. Hype and superhype are part of the computer culture. Since computer users are typically even less able to think clearly about the use of computer products than those who design them, it is not surprising that the mere mention of a new technological feature is enough to get them to buy a

product. They do not keep data on how many hours it takes them to figure out how to perform tasks. Experience with technical courses in school, and with technical products throughout their lives, has taught them to expect to be frustrated when they attempt to use a technical product.

(2) Documenters are natural-language oriented. Whereas programmers and engineers see only technology, documenters see only communication, meaning, prose explanations. They believe their purpose is to smooth and soften the hard edges of technology with their liberal arts prose. The reasons for this belief are obvious: most documenters are English majors, or at least, majors in one of the other humanities. A nationwide study by the Society of Technical Communication (STC) in 1992 found that less than 20% of the members surveyed had bachelor's degrees in technical subjects.

Most documenters are women trying to earn a living in a world that is alien to them, namely, the world of machines and, even worse, of mathematics, including the branch of mathematics known as computer programming. (*Why* this world is alien to these women is another matter; my belief is that the fault lies with primary and secondary school math and science teachers.) These women typically have highly developed language skills and thus, quite naturally, believe that the employment of these skills is what is needed to make computer systems easier to use. Unfortunately, they are wrong, and I will try to explain why.

The fundamental difference between technical subjects and non-technical subjects is that the former are essentially *geometrical*. What I mean by this can be explained by the example of the auto mechanic who is good at fixing your car, but who is not good at explaining what he does, or why. In mathematics and the sciences, likewise, there are original thinkers who make fundamental contributions to their field, yet who have great difficulty setting forth these ideas clearly, even to their colleagues.

To understand how such things can be, imagine that you had the task of teaching auto mechanics to a group of students who didn't speak your language. To a very large extent, you could do this with pictures (including animations) and symbols like arrows, and perhaps exclamation points (No! No!), and smiling and sad faces to indicate Do this! and Don't do that!. Similarly, there are many subjects in mathematics and the sciences which could be taught with a minimum of natural language, given numerous pictures (including animations). This is what I mean by saying that technical subjects are fundamentally geometrical.

This is not true of the humanities, in particular, of literature and philosophy. We may be able to draw pictures and diagrams to represent characters and plots of novels, but I doubt if anyone who understands literature would say that pictures and

diagrams can represent as well as the novel itself does, what the novel expresses. This is even more true for poetry.

Nevertheless, the vain hope that technical subjects, in particular, the technical subjects which are the uses of software systems, are fundamentally language-like, continues to persist among technical writers and editors. I recall one woman who, in a discussion of the curriculum for a Master's program in technical writing, firmly maintained that the fundamental discipline for technical writing was *rhetoric*.

The truth is that — as anyone knows who has observed how software systems have changed over the years — prose is on the way out as a means of communicating the use of these systems. Menus, icons, dialogue boxes, mice — all of these are *geometric* devices.

(3) Another reason why progress has been so slow is the inflation of simple ideas in the technical communication literature. The reasons are clear and understandable: technical communications has always been at the bottom of the ladder in industry, looked down upon by managers, engineers, and programmers as a necessary inconvenience — in the shameful vernacular of an earlier age, primarily "women's work". Anyone who doubts this need only review the literature, including the monthly magazines of the technical communications' societies, in which the question of what can be done to raise the low prestige, and lack of appreciation, of the profession is an ongoing theme.

The natural reaction is to inflate whatever research is done to make it seem as profound, hence as respectable, as the research in other disciplines. A programmer I knew once phoned the head of the human factors department at the computer company where he worked and asked her for some advice and references to literature and/or training courses on doing usability testing. She replied that if he wanted to do usability testing, he should get a Ph.D at Stanford, since, just as you shouldn't expect to do brain surgery without the proper training, so you shouldn't expect to do usability testing without proper training. It is to the credit of the company that this person, and her entire department, were eliminated during a corporate downsizing soon after.

The inflated two- and three-day seminars with titles like Designing for Usability, Modular Documentation, which are taught on the industrial training circuit by Ph.D.s in communication or some other academic puffery, are another example. The concepts in these courses could be expressed on a couple of word-processor pages. Those who teach these courses do a disservice to their profession, and to their students, by not making that clear from the start, and then spending the remaining time in discussing the application of these concepts to the students' real-life projects.

The wordiness, the endless repetition of the same laments, the endlessly repeated elaboration of the same handful of simple ideas, in technical communication literature, is yet another example.

(4) Until the mid-eighties, there was no general appreciation of the importance of notions like provability, testability, guarantee of success, which comes from studying, if not mathematics, then at least computer science.

Already in the sixties, computer scientists were worried about proving the correctness of their programs, yet, in more than thirty years in the computer business, I never once heard a technical writer or editor ask that crucial question, How do we know that our work is accomplishing its goal? The question is asked in the technical writers' society publications, but in my experience is never an urgent concern of writers and, for that matter, managers of writers.

Where are the Scholars?

Early in this chapter I said : "The Environment *captures the use* of the system. Every possible path a user can take through the Environment constitutes every possible way the system can be used." (See "Review of What an Environment Is.") There are other ways of capturing the use of a system, e.g., through finite-state machine diagrams like those that are used to describe machines that parse strings generated by formal grammars. In this case, each node in the diagram represents a "state" of the system as seen by the user, and the arrows connecting nodes represent actions the user can take, e.g., the entering of a command, clicking or doubleclicking a windows icon or button, etc.

In other words, there are ways to *rigorously* describe the use of a computer system relative to a given class of users, which means there are ways to *rigorously* answer such questions as:

(1) What is an upper bound on the complexity of the use of this system?

(2) What is the minimum number of user actions required to accomplish the task *x*?

In the case of (1), the reader may wonder why I do not simply say the complexity of the system? The answer is that the term *complexity* as defined in algorithmic information theory does not mean the number of bits currently needed to store information in a computer, in which case the complexity of use of a system would simply be the number of bits required to store its complete Environment.

Instead, complexity means the *minimum* number of bits required to store a *representation* of a binary string. (We can consider the complete Environment as a single, very long, binary string.) Now, one of the theorems of algorithmic information theory is that there does not exist an algorithm to determine, for any given binary string, what its minimum representation is. Sad, but true. So if the complete Environment of a system can be stored in some number *n* of bits, then the best we can say is that the complexity of use, relative to the intended class of users, is no more than the number of bits needed to store the Environment.

In the case of (2), such questions can in principal be answered by machine, given the complete finite-state machine diagram representing the complete Environment.

Even without these two representations of use, we can rigorously answer another pair of important questions, namely:

(3) What is the relative frequency of tasks/actions performed by some specified set of users of a given system?

(4) What sequences of tasks/actions consume the most user time?

These questions can easily be answered by recording all actions performed by each user, along with the time of day. In my experience, technical writers and computer-human interface designers occasionally talk, during the planning stages of a project, about the desirability of keeping such records, but with the exception of an on-line help project I managed in the seventies, I have never seen this talk materialize. Similarly, the literature often mentions the desirability of maintaining such records, as though the mentioning were enough to demonstrate to the world how advanced the profession really is.

I say that it is shameful that the keeping of such records is not a universal practice. It is shameful that this practice is not considered at least as important as writing English well, or having the latest GUI technology. Even apart from their use in improving the Environment, and of revealing a great deal about how users actually use a system, as opposed to how they (and the system designers) think they use it, such records can also be of great profit to all users who wish to improve their own daily performance by finding shortcuts for those sequences of tasks and actions which they in fact spend most of their time doing.

If you think that most computer users already know what they spend most of their time doing, you should consider a study which Stanford professor Donald Knuth once carried out on the optimizing of computer programs. Knuth asked programmers to speculate on what parts of their programs consumed the most execution time, then ran the programs with typical inputs, and plotted the time spent by the various parts of the programs. It turned out that most of the speculations were wrong: the

programmers had no idea where in fact the programs spent most of their time. Which means, among other things, that the labor of optimizing programs was, and probably still is in many computer firms, largely a matter of wasteful trial and error. The same can no doubt be said about attempts to improve pleasantness-of-use, in which, typically, technical writers and human factors experts with far too much technology and far too little data (not to mention understanding of the key issues), continue to throw more manuals and ever fancier graphics at a problem they don't understand.

The four questions listed above are interesting, important questions. The writing down, the capturing, in an efficient Environment, of the entire use structure of a computer system, or of any apparatus, is a *scholarly activity.* It demands the same care and attention to detail and concern for completeness and correctness, as does any scholarly endeavor. Where are the scholars, the academics, who understand this, and who are willing and eager to carry out such scholarship? Where are the scholars who are curious about the whole question of finding rigorous ways of answering the question (whether applied to computers or airport terminals or VCR's): *How is this system used?*

The New Culture

Every new profession, or branch of a new profession, develops its own culture. The mark of an Environment designer professional will, I think, be an acute sensitivity first to usability issues in everyday life — signs at airports and other mass transit systems, instructions on boxes of soap and, of course in the manuals of all equipment — as well as to usability issues associated with new computer technology. A good designer will reserve a healthy skepticism toward all such technology, asking the all-important questions, What evidence is there that this significantly improves the throughput of the systems with which it is, or will be, associated? and What is the cost-effectiveness of this new technology relative to throughput? A good designer will have an on-going curiousity about how people learn to use new technology, about how they actually go about figuring out how to use it, about what they think of the difficulties they confront.