CHAPTER 3

HOW TO BUILD A ZERO-SEARCH-TIME ENVIRONMENT

Definition of *Environment*

An Environment for a software system is all the information that any intended user of the system needs in order to perform the tasks made possible by the system.

Environments typically consist of some combination of the following: keyboards, softkeys, windows and/or other menu facilities, error and other messages issued by programs, manuals, quick-reference cards, on-line help systems, hypertext facilities, training courses, on-line support, and a considerable body of information transferred from user to user by word-of-mouth ("systemlore," "folklore").

An Environment, therefore, is a *data base* — although not one that is always implemented in software — whose content is all information required to *use* a software system. Or, viewed more abstractly, an Environment is a mapping (i.e., a function) from tasks to those procedures and other information required to accomplish those tasks on the system.

Before we proceed, let me make a distinction which may seem pedantic: in the first paragraph above, instead of "tasks made possible by the system" I should have said "tasks made more rapid by user *and* system." The reason that this distinction is important is that good Environment design requires that the designers always hold before them the "new" view of a computing system that was described in chapter 2, as well as the all-important difference between the *What* and the *How.* No digital computer ever does anything that a person couldn't, in principle, do; it just does it much more rapidly. Long before computers existed, people wrote and edited manuscripts, kept financial records, drew graphs, performed calculations, communicated with each other. The binary symbol manipulations that take place in the cpu could, in principal, be done by a person — or, I should say, taking into account how many of these manipulations are done per second nowadays — by many generations of persons. Even though, for brevity, I will use the phrase, "tasks made possible by the system," you should understand that I mean this phrase in the sense explained in this paragraph.

Every computer system comes with an Environment of some sort. The question is, How good is the Environment? On the basis of the definition of usability given in chapter 1, we can see that the answer is, That depends on how rapidly users are able to find out *how to perform* the tasks made possible by the system, in other words, by how *usable* (my definition) it makes the system. More specifically, the goodness of an Environment can be measured by the average length of time it takes any user to find out how to perform any task he or she wants to perform. In the sample Environment in the appendices, a goal of 25 seconds *maximum* look-up time was set — any user in the class of intended users should be able to find out either how to

perform any task offered by the system, or to find out that the task cannot be performed on the system, within 25 seconds, except, possibly, for the first time the user uses the Environment. In the case of the Appendix A Environment, a goal of under three minutes is set for the first look-up time because in this case the user has to learn how the Environment works.

Here we see the importance of the distinction between usability and pleasantness-of-use. The procedure for actually performing the task — the sequence of subtasks required to perform the task — may be pleasant or unpleasant, may be easy to carry out or tedious, but that is not a quality of the Environment: it is a quality of the underlying software. (I am assuming, here, that the procedures are *correct*.)

To illustrate the point, suppose a project manager proclaims that the Environment for his product is perfectly adequate: why, not only are there several manuals — *Introduction*, *User's Guide*, *Theory of Operation*, and *User's Reference* — but the company also offers training courses, and eight-hour-a-weekday access via phone and e-mail to customer support (for a modest annual fee). Let us see what, in fact, (s)he is saying:

- First, (s)he is saying that any user who makes use of all these facilities will be able to accomplish any task made possible by the system. (How does (s)he know?)
- Second, (s)he is saying that, for some tasks, it may take hours, and, in fact, *days*, to find out how to perform them. Not 25 seconds, or three minutes and 25 seconds, but *days* — namely, in the time required to attend training courses. And after that, certainly hours per week in searching through manuals and talking to other users and calling customer support. I hope that no project manager reading this will be in doubt about the influence which a difference of *days* versus three minutes, 25 seconds in information-retrieval time would have on a prospective buyer — even if the underlying software were identical to that of a competitor's product!

Definition of *Zero-Search-Time* Environment

Since *every* software system has an Environment — even if that Environment consists of no manuals, no on-screen menus, nothing but the name and address of the manufacturer (a rather inefficient Environment, of course, since every first-time user is forced to either discover how to perform each task by trial and error, or else by calling or writing the manufacturer) — the purpose of this book is to present a method for developing *efficient* Environments, i.e., Environments which have been designed to

reduce the look-up time to some specified maximum, e.g., 25 seconds. In that case, we say that no user should be forced to spend more than *25 seconds* trying to find out how to do what he or she wants to do in a software system — regardless of the size of the system, regardless of whether the documentation is on-line or on paper. No user should ever have to *figure out* where the instructions on performing a task are. This should be as easy — should require as little thought — as looking up a word in the dictionary. Any technical writer or human factors expert who doesn't deliver on this central criterion of success is depriving his or her company of major profits.

The essence of such Environments is that the user never has to *figure out* where the procedure implementing a task is. In other words, the user never has to spend any time *searching* for the procedure in the sense of trying to find something whose location one does not know. Hence the adjectival phrase, *zero-search-time*. Of course there will always be a certain, nonzero look-up time, simply because we cannot turn pages, or manipulate a mouse, or expect hypertext or menu software to execute in zero time.

Incidentally, if you doubt the importance of minimizing look-up time, a study of present, and prospective, customers by one of the Big Five computer companies in the late 1980's found that the factor which would *most strongly* influence a future purchase of the company's computer products — was that the time to locate information was less than one minute!

Another way of thinking about the design of efficient Environments is that it is an attempt to *engineer the use — structure the use* —of a proposed software product.

The Real Contribution of Windowing Systems

This is a good place to take a critical look at what exactly windowing systems have done for usability and pleasantness-of-use. They have certainly increased the *speed* of issuing commands, a fact which could be easily verified by having one user enter a list of commands as rapidly as possible by typing them, and another user issue the equivalent of the same list of commands by clicking menu choices with a mouse. Furthermore, for a certain limited set of tasks, they have improved usability (my definition) by visually clustering related tasks, e.g., those pertaining to printing a document. Windows, furthermore, have established an interaction language which is now widely known and used, one involving menus, dialogue boxes, use of the mouse, and conventions for moving dialogue boxes and other display elements around on the

screen, reducing their size, etc. But if you are inclined to say that icons and windows have significantly improved speed-of-access to information about how to perform tasks, I encourage you to conduct a few experiments. Select any person you consider to be computer literate, and, in particular, knowledgeable about windowing environments, but who is not familiar with a given program that uses a windowing interface, and ask that person to perform a common, simple task. For example, in the FrameMaker desktop publishing system, ask the user to insert a bullet (•) at a specified point in the text, not as part of a bulleted list. Or in OpenWindows running on a Sun workstation, ask the user to change the placement of an icon which appears as part of the initial Desktop to a new, specified location. Or ask a person who is Unix literate, but not a Unix expert, to change the printer on which e-mail messages are printed, or to define a new alias (abbreviation) for a list of people who are to receive messages. (Other examples are given in chapter 4 in the section "Exercises for the Skeptical.") In all of these cases, you will probably not need a stopwatch to do the timing.

Effect of Look-Up Speed on Environment Design

Look-up speed is the great vacuum cleaner of Environment design. (I don't like to think how long it took me to come to this important realization!) The criterion of look-up speed provides an easy way to resolve those endless arguments which technical writers get into, especially when they are working on on-line documentation systems, about "what the user will want" and what the user will do when confronted with this or that screenful of icons, titles, messages, and whatnot in the documentation system. Given two competing proposals for how a documentation window should be designed or how the documentation itself should be structured, you simply ask, Which of these is more likely to enable a user, including a beginning user, to find out how to do what (s)he wants to do in less than 25 seconds? and give a convincing scenario for your reply.

The greater you want look-up speed to be, the less *thinking* about how to find information you can ask your users to do. In fact, as you will see, in efficient Environments, your goal is "no-thinking" look-up-ability. This can easily be achieved with today's technology; in fact, it can be achieved with Environments that are totally implemented on paper!

This property of Environments can be expressed informally by saying that once a user has decided *what* he or she wants to do on the system, he or she should not have to figure out *how* to do it, or *if* it can be done on the system! In the vernacular, it should be a "no-brainer" to find out how to perform the task, or to find out that the task is not offered by the system.

Look-up speed is the central, the most important, criterion of usefulness of an Environment. If you ever find yourself in a course on documentation design in which this criterion is not mentioned, or is considered "also desirable", run, don't walk, to the nearest exit, because you are about to waste your time and money. Everything follows from look-up speed: structure, format, correctness, and the method by which the documentation is created.

Replies to Criticisms of the Idea of Developing a Method for Environment Design

Before I present the method, I would like to respond to a few common criticisms of the idea of developing a method.

Criticism 1: A method is an attempt to reduce thinking, but thinking is precisely what we need more of, not less.

My first reply is the following quote:

"It is a profoundly erroneous truism, repeated by all copy books and by eminent people when they are making speeches, that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case. Civilization advances by extending the number of important operations which we can perform without thinking about them." — Whitehead, Alfred North, quoted in Newman, James R., *The World of Mathematics*, Vol. 1, New York: Simon and Schuster, 1956, p 442.

My second reply is that we must be clear about the type of thinking that the Environment method proposes to reduce: it is the thinking presently required to *find out how to perform tasks on computer systems*. It is certainly not the thinking required to organize those tasks to accomplish some larger task, e.g., to write a program, or construct a complicated search of a data base.

Criticism 2: The design of a software system is too complicated a process to be reduced to a method. (The reason why I say "software system" here, instead of just "documentation system," will become clear below.)

My reply is that, nevertheless, attempts have been going on for years to find methods for designing software, e.g., through the development of Computer-Aided Software Engineering (CASE) systems, as well as for maintaining it, e.g., Source Code Control Systems (SCCSs), and Bug Tracking Systems (BTSs). (The difference, incidentally, between CASE systems and the method I am setting forth is that CASE systems are aimed at engineering *software*, whereas my method is aimed at engineering *use*.) It is precisely the most complicated technical tasks that we should, and generally do, try to simplify by reduction to a method.

Third, it is much easier to talk about, and apply, variations of a simple design method than it is to talk about, and apply, a collection of ill-defined ad hoc techniques.

Criticism 3: The method is too rigorous.

My reply is, first, to ask if "too rigorous" means too rigorous to result in an improvement over present practice, or if it means too difficult for current writers and human factors experts to follow. In either case, we can only determine the answer by actual trial.

Criticism 4: It would nice to be able to apply a method, but it will be too expensive in time and/or money.

My reply is that the method doesn't have to be applied in its entirety in all cases. *It is not an either/or proposition! In particular, Environments can be created after the fact* — after the product has been designed and released to production. We will discuss these Environments below under "After-the-Fact Environments". The only disadvantage is that in this case, if the software is indeed difficult or tedious to use, it is much more difficult to modify.

Criticism 5: (A leftover from the "Artificial Intelligence Spring" (AI Spring) of the mid eighties): we should postpone concerning ourselves with usability and just wait for the natural language interpreters which will make the usage problem trivial.

My first reply is that many tasks which we perform with computing systems, e.g., text and graphics editing, routine business activities such as order entry, inventory control, are most efficiently done *not* by natural language communication, but by more concise forms such as templates and menus. (Imagine doing routine editing by voice control alone: "Now take the phrase 'is required' in the — let's see: first, second, third, fourth, fifth — fifth paragraph of the current screen, and move it to immediately preceding the — let's see: first, second, third, fourth — fourth period in the second paragraph."

Second, many systems will be written before these natural language systems become available, if they ever do, and the complexity of use issue must be faced in these.

Criticism 6: We can simply improve our present manuals, courses, and customer support services. It is not necessary to devise an entire new method for making software usable.

My reply is that this criticism is made without data as to the throughput resulting from the present way of doing things, versus the Environments produced by the method described below.

Furthermore — and here the burden of proof rests on me — I claim that the method produces better *software* as a by-product, i.e., software which is cheaper and easier to design, debug, and maintain.

Criticism 7: There is nothing new in the method being proposed.

My reply is that it is true that the idea of iterative design is certainly not new, but it has yet to be tried in the way described, namely, by *beginning* with the desired use structure for the software, and then iterating toward its implementation. No one designs software that way now, though occasionally people talk about doing it in the future.

A Method for Designing Zero-Search-Time Environments

I will present the method in its ideal form, in the firm belief that it is much easier, in practice, to produce something that is a modification of something you clearly understand, than it is to produce something that is a gluing together of pieces you believe will add up to something you will understand. The method consists of two major steps:

(1) Initialize the Environment.

(2) Use the Environment.

It's that simple, at least at the top level. Following are details on each step.

(1) Initialize the Environment

There are three steps to initializing an Environment. Details on each step are given immediately below.

(1.1) Establish the Class of Intended Users (CIU).

(1.2) Establish the Success Criteria.

(1.3) Establish the Primary (or "Top-Level") Tasks.

(1.1) Establish the Class of Intended Users (CIU)

To establish the class of intended users (CIU) of the Environment, we define the minimum knowledge (understanding of words, phrases, concepts) *and* the minimum set of abilities which *every* intended user of the Environment will be expected to have. All knowledge not assumed to be already possessed by a user *must* be provided by the Environment. More precisely, this means that every word, phrase, or concept which the user is not assumed to know:

- *must* be look-up-able directly (i.e., via an equivalent of the index function) *and* indirectly (i.e., through any reasonable synonym which is itself look-up-able via the equivalent of the index function);
- *must* be explained, directly or indirectly, in terms of words, phrases, and concepts which the user *is* assumed to know. An Environment designer cannot shirk this responsibility by arguments of the sort, "The user will be able to figure out...," "The user will be bright enough to understand that...," without risking failure to meet the look-up time criterion specified in "(1.2) Establish the Success Criteria."

Let us consider a few examples: Normally, you will assume that every user will know how to use a standard computer keyboard. But can you assume that every user knows what a function key is? If not, then you must explain how these keys work. If the software system runs under a windows facility, can you assume that every user is familiar with the use of that facility? If not, then you must explain common words and phrases associated with the facility, e.g., *double-click*, *drag*, etc.

The *smallest* CIU is, of course, the programmers who wrote the software system. Presumably, they know what every word and phrase concerning the use of the system actually means. On the other hand, the company probably does not intend to sell the system only to these programmers. The *largest* CIU, strictly speaking, is the entire human race. For such a CIU, you could not even assume literacy, much less

literacy in a language understood by a significant number of software system purchasers.

Thus, we have an informal rule: the larger the CIU, the larger the Environment must be, due to the space required to explain unfamiliar words and phrases.

The CIU, along with the success criteria explained in the next section, enable us to give a more precise meaning of the word *clear*, as in, "It will be clear to the user...,": *clear* in this context simply means enabling the user to accomplish the task in question in accordance within the time defined by the criterion of success.

A word about hypertext is appropriate at this point: because the time to learn to use the Environment is part of the measure of its efficiency, guiding rules for the user must be as simple as possible. A short rule is, "Every word and phrase whose meaning you can look up is underlined. Clicking such a word or phrase will cause a short definition to be displayed." Now the fact is that these underlinings may clutter the text. So you have to make a choice: make the rule more complicated, e.g., "Only the first occurrence of a given word or phrase in a section will be will be hypertext-linked, unless the section is more than three paragraphs long, in which case ... ". This now introduces more complexity in the use of the Environment. Or else you can stick with the simple rule and hope that these underlinings will become less and less annoying to the frequent user.

Such issues are not trivial. If your rule governing the looking up of definitions is so cumbersome that no one bothers to read it, a major convenience — and speed enhancement — of your Environment will be lost.

(1.2) Establish the Success Criteria.

The mark of a science is the degree to which measurement of results — of success — is possible. In the case of Environments, as stated at the start of this chapter, the most important criterion of success is the speed at which any user in the CIU can find out how to carry out a given task. However, experience suggests that not every user will be able *always* to look up a procedure within the specified time limit. (Some people take longer to do basic mechanical operations on some days.) Therefore, Environment designers and the project manager need to come up with an acceptable success rate, for example, 80%, so that the criterion for success will be that, in 80% of user attempts, the user will be able to find the procedure for carrying out a

given task in less than 25 seconds after the first use of the machine. To measure the actual success rate, users randomly selected from the class of intended users are asked to perform randomly selected tasks, and their speed of success is measured.

This means no person-to-person communication of any kind unless such communication is an explicit part of the Environment. Our aim is to capture *all* the information which any member of the CIU needs to perform any task offered by the Environment.

Of course, the performance of a given task may require several steps, each of which in itself is implemented by a procedure. The 25-second figure applies to the procedure for each step, not to all procedures together.

At the start of the design process, therefore, the designers must establish what they will consider the Minimum User Success Rate (MUSR) in order that the Environment be considered guaranteeable, in other words, the minimum percentage of attempts by users to find out how to perform a task that must lead to their finding the information within the specified time period.

This testing of the Environment on members of the CIU proceeds throughout the evolution of the Environment and its contained system. (The early stages of this evolution are often called the design stage.) The evolution occurs incrementally, i.e., it is broken up into a sequence of segments each the length of, say, a few weeks, as shown in the pseudoprogram under "(2) Use the Environment", below. At the end of each segment, that version of the Environment and its system become available for user testing, while feedback from the use of the previous segment becomes the basis for the next set of modifications.

In the early stages of the evolution, of course, it will frequently be the case that the system program to execute a given task does not yet exist. It is, however, essential that the testing be conducted anyway, namely to ensure that users are able to know that the task exists whenever they need to know it. (Test it *before* you build it!)

Many Environments and systems evolve within a context in which the tasks to be implemented already exist in cruder forms, i.e., they exist in other software systems, or are done manually. Thus it is perfectly acceptable, and even encouraged, for the Environment initially to implement tasks by such commands to the user as: "Use system *y*," "See John Doe." This again reminds us that the fundamental reality is tasks, not software.

(1.3) Decide on Primary ("Top-Level") Tasks.

The system and the Environment are being designed to allow users to perform certain tasks: to do text editing, to write and run programs in a certain language, to store and retrieve data from a data base. These tasks we call *primary tasks*. In paper Environments, the primary tasks are listed on the Start Page (see Appendix A); in on-line Environments, the primary tasks are listed on the initial screen. For example, the (top-level) primary tasks for a text editor would probably be:

Create a new text.

Edit an existing text.

View existing texts.

For an operating system, the (top-level) primary tasks would probably include:

Write a program (e.g., an application).

Run a program (e.g., an application).

Create or edit or print a text.

Communicate with other users.

In addition to the primary tasks there will be metatasks which operate on, or are concerned with, the system and the Environment themselves. We call these *secondary tasks*. One secondary task which must be present in *every* (sub)Environment is:

Exit the present (sub)Environment (i.e., return to the previous (sub)Environment).

Secondary tasks which must be present at least at the top-level Environment would include:

List vocabulary and abilities this Environment presumes on the part of the user (i.e., display definition of CIU for the Environment).

Change or repair (debug) the system or the Environment.

With the designers acting as the initial users of the system, the structuring by use begins to develop, a la structured programming: each top-level task (primary or secondary) is broken down into constituent second-level tasks, and each of these into third-level tasks, etc., until, in each case, a level is reached at which the task can be performed *entirely* by the system once the user has initiated it.

For example, the second- or third-level primary tasks available under "Edit an existing text" in a word-processing system might include the following. Curly brackets mean that any one of the enclosed items can be chosen:

{Find, delete, insert, move, copy, display} {character, word, string, line, paragraph}

Print {character, word, string, line, paragraph(s), file(s)}

Format {character, word, string, paragraph, line(s), file} as follows...

Observe that the question of how the choice is made, as well as the question of how the character, word, string, etc. is to be indicated, are subordinate questions. The important thing is that the *What* (the task) always precedes the *How.*

As the evolution of the system and Environment proceeds, the designers continually observe, for possible later implementation:

(1) what they (and other) users find themselves *wanting* to do in the Environment;

(2) *when* they find themselves wanting to do it, i.e., in what (sub)Environments.

(2) Use the Environment

Using the Environment consists of a sequence of iterations of steps which are described in the following pseudo-program. A process like this is called *iterative design*, as contrasted with the more traditional linear design process (sometimes called

a *waterfall process*), which consists of a succession of stages, e.g., Initial Design, Design Review, Breadboarding (i.e., creation of first prototype), Breadboard Review, Coding, Documentation, Testing, Product Release.

while product is viable **do**
begin current iteration
Incorporate changes from last iteration.

Break down next task(s) into subtasks which implement it.
(The presentation of the subtasks is an extremely important part of the Environment concept, and is further described below under "The Presentation of Subtasks.")
(At some point, these subtasks are performed by computer programs. The use structure at that point is described below under "Universal Flowchart for Tasks Performed by Software.")

Do user test(s).
(Product designers, programmers, technical writers can and should *act* as users, as long as they don't assume skills and vocabulary not possessed by everyone in the CIU.
Every Environment can *always* be tested down to the level of each task statement. The question is, Can the user get to the right place in the Environment within the minimum established time?)
end current iteration

After product release, of course, there will probably be little or no further breaking down into tasks, all activity consisting of incorporating feedback from the previous iteration.

Let me remove any doubts, or perhaps I should say wishful thinking, on the part of Environment designers and technical writers working on Environments: "Use the Environment" means that you, too, must use the Environment! You must learn enough of the subject matter to put yourself in the CIU, and then you must attempt to solve typical problems using the Environment. How important is this? The FrameMaker documentation is a good example. I don't know much about FrameMaker, despite months of using and attempting to use it, but one thing I do know

is that the documentation, including Help, was not developed by people who *had to* use it in order to get their work done. No one in that position could possibly have wanted to write, much less would have written, documentation which requires as much searching and reading and figuring out and and phone calling and learning as the FrameMaker documentation does.

A few words on iterative design before we proceed: In computer science, this concept goes back to at least the early seventies, where it was used in the Artificial Intelligence/LISP programming culture. ("Build the first one to throw away.") It is often *mentioned* in discussions of the design of new software products, almost like an incantation to assure the product's success. Technical writers, who are much more prone to believe that to recognize, much less utter, a popular technical term is somehow to understand it and to *have applied it*, usually dismiss the idea because they vaguely remember having heard of it somewhere. But in many years in the computer field, working at several different companies, I have seen iterative design put into actual practice only once. The idea of creating successive versions of the product that *can be* used, and *are* used from day one — certainly from *month* one — much less the discipline of actually doing so, is still thoroughly radical. Programmers still want to *finish* the product and *then* use it. They cannot understand how one can separate use from functionality.

However, signs of change are in the air. Designers of GUIs (Graphical User Interfaces — programs which use a windows format to communicate with the user) are now beginning to follow a design philosophy called *Structured Rapid Prototyping* (or *Iterative Protyping*), which is essentially iterative design, and which came about because new software tools make it possible to build and modify GUIs much more rapidly than one can write the plans for building and modifying them! The technical writing community has still not caught on to the enormous advantages that such an approach affords, and that the approach can be applied to on-line documentation as well: you still hear writers (and managers) complain that they can't *really* start writing until the product is finished, or very nearly finished, or at least until what they have to write about has stopped changing. But this is nonsense, as I hope you will understand by the time you finish this book. Documentation can and *should be* started on the first day of product development, and the top levels of the documentation can and *should be* viewed and used from the first week or two, and it doesn't matter how often the documentation changes during the course of development. Changes are easy to make, just as they are easy to make in the new GUI environments. That is one of the main advantages of rigorous top-down task-oriented Environment design. Documentation should always track GUI development — the two should be present, side by side, on the same screen, from the start. Instead of writers spending months planning and

discussing "what the user will want" (the vast majority of writers haven't a clue about what the user will want because they have never been daily, long-term users of the systems they write about) — instead of writers wasting all this valuable company time, they should get the top levels of the Environment on the screen and start modifying it based on their own and others' use, and, of course, based on additional information as it becomes available from the product designers.

The Presentation of Subtasks

A sub-Environment is a set of subtasks implementing a given task, and the essential requirement of every sub-Environment is that it give the user a rigorous presentation of the subtasks that implement that task. By a *rigorous presentation* I mean that the list:

- makes clear to the user *all and only* the subtasks that can be performed to implement the given task;

- makes clear the sequence in which the subtasks must be performed in all cases where sequence is important, and makes clear when sequence is not important;

- makes clear where the sub-sub-Environment can be found which implements each subtask.

- makes clear how to return to the next subtask (if any) in the previous sub-Environment (the super-Environment) and to the initial Environment (i.e., the Start Page or Start Menu). (The returning to the next subtask in the super-Environment is analogous to the behavior of a program upon completing a "call" to a procedure or subroutine. In on-line documentation systems, there should be a button which is always present that performs this return function.)

This rigorous adaptation of the procedure-call protocol of programs is one of the most important features of an efficient Environment. However, it often arouses an objection among beginning Environment designers, namely, that users "will get lost" in the hierarchy — they will no longer remember what task they are carrying out, and where the current sub-tasks are in the task tree relative to the others.

My replies to this objection are: (1) I know of no reports from long-term users of such rigorous task structures that corroborate this opinion; (2) it is possible to provide various kinds of orienting sub-titles in each set of subtasks; (see below under

"Alphabetical Titles = Numerical Titles!") (3) the window or screen in which the software is used provides a visual context; (4) we seldom if ever know at what task level we are in when we use computer systems: we just "remember the steps", or follow the instructions in a manual.

Examples of sub-Environments are given in the appendices.

Let us consider the sequencing of steps in a procedure. There are some subtasks that need to be done in sequence, and some that don't, and it is absolutely irresponsible for an Environment designer not to make the difference clear. Of course, in most cases there will be more than one sequence of steps that could perform a task; you are merely recommending the best one that you, and previous user experience, has dictated. (And you need to state that explicitly whenever appropriate.)

I sometimes get the impression that nowadays technical writers, with the new tool of task-orientation in their possession, feel it is somehow indecent to impose an order on steps. They seem to think that the only kind of acceptable subtitle is one whose first word ends in *ing*, e.g., "Inputting Data," "Setting the Initial Parameters," "Testing for Successful Access," etc. But at the very least, there is a sequence in the way the product is used, namely, install the product if necessary, then turn it on, then use it (here there may be a great deal of freedom in the order in which tasks are done, as will be clear from considering a desktop publishing system, since you can edit an existing document, create a new document, print an existing document, in any order you wish).

Sometimes it is legitimate to put a sequence of steps in a paragraph, e.g., when the sequence is short, say, less than four steps or so, and when the steps themselves are short, e.g., the clicking of buttons in dialogue boxes. But even here, always put the task first, then the procedure. Say, "To do *x*, do the following..." not "Clicking the *x* button, then the *y* button, will result in..." unless you are warning the user of what not to do. In these cases of short sequences of short steps, it is legitimate to omit numbers.

But otherwise, let me repeat: you should number steps when, and only when, sequence is important. When sequence is not important, use bullets or other symbols that do not imply sequence. In the Environments I build, I use language such as the following whenever a step involves choices: "Do one or more of the following, in any order you wish, as often as you wish. When you are done, go to step ..."

It is perfectly legitimate, by the way, for a step to say, in effect, "Read the following sections... and then use your best judgement as to how to produce the following result..." [or "...a result having the following properties..."]

Finally, in order to stay within our maximum look-up time limit, we must make it clear where the user can find how to do any subtask he doesn't know how to do. In paper Environments, this is done by a statement such as "See 'data base, inputting data

to a'". In on-line Environments this is done by an explicit menu choice or by a hypertext link.

There is one more very important issue concerning tasks which we need to confront, and this we do in the next section.

Tasks and "Things"

As a result of the growing trend toward task-oriented manuals, technical writers nowadays often give a brief summary of the tasks to be described in a chapter, along with page references. This is a step in the right direction. Unfortunately, these writers still have one foot in their literary heritage, and believe that the brief summary is, in effect, simply a kind of table of contents, with titles and sub-titles written as tasks, setting forth what will be *covered* in the chapter. In other words, it's still the old "Read it, learn it, use it." In the worst cases, these writers have learned nothing more than the trick of converting old-style table-of-contents headings — "Installation," "Input," "Output" — to the new gerund style — "Installing the System," "Inputting Data," "Outputting Results" — flattering themselves that somehow this constitutes an advance in their craft. Against naivete of that magnitude even the gods strive in vain.

But the chapter concept itself is obsolete, as I assume the astute reader has already realized. It is surprising that this table-of-contents type of listing continues among people who themselves must often have been frustrated in rapidly finding the instructions that they were looking for, because, as often as not, the referenced pages do not contain *all and only* the steps needed to accomplish what they want to accomplish.

This is why the maximum time constraint on finding procedures for carrying out tasks is so important. Among other things, it forces writers to reduce to a minimum — to zero! — the amount of reading and searching that the user needs to do.

Let us think as clearly as we can about this question of tasks. First we realize that tasks normally involve *things*: we perform tasks on *things*: we modify (a task) the format of a paragraph (a thing); we change (task) a tab setting (thing); we print (task) a file or document (things); we input (task) data (thing) to a data base (second thing); we modify (task) the format of data (thing) in the data base (second thing); we run (task) a program (thing).

Of course, the chapter in a traditional manual is usually about one or more things: in a desktop publishing system, a chapter may be about creating and modifying illustrations, or rather, about the drawing and illustrating module of the system; in a

data base management system, a chapter may be about schemas or queries or security; in an operating system, a chapter may be about utilities, scripts, command interpreters.

Next we realize that, in the course of using existing software, i.e., software for which a zero-search-time Environment does not exist, we often don't know under which heading to look up a given task. If we want to find out the names and locations of printers to which we have access from our workstation, which manual's index should we turn to, and once in that index, what heading should we look under? "Printers"? If we don't find it there should we then try "devices, output"? If we don't find it there, should we then look under "output devices"? Or, in Unix, should we then look under "environment variables" (or "variables, environment") because we vaguely remember that some of these are concerned with printers? Or should we look under "spool" because the spooling utility may have associated with it a list of all accessible printers?

So it seems clear that if we are to achieve our goal of 25-second maximum look-up time, we will, once and for all, have to come to terms with the relationship between tasks and Things. (From here on, I will write the word with an initial capital to distinguish this particular type of thing. I realize the word is not a good one, but two of the alternatives seem to me to be worse: "object" has several other specific meanings in computer science; "entity" seems too general.)

Fortunately, computer science has already provided a way to deal with this relationship, namely, in the concept of *abstract data structures*. Here is the background.

A data structure, as its name implies, is a way of organizing data. For example, a list, (a, b, c, d, ..., z), is a data structure, the items of data being represented by a, b, c, etc., and the address of any item of data being specified by a number, k, representing how far down the list, from left to right, you need to count in order to reach that item. A table is a data structure, with each item of data having a row and column address.

Such structures are indispensable to programmers, but, as programmers gradually found out, there were many ways to implement each data structure, depending, for example, on the programming language they were using, and the constraints of program operating speed they were faced with. It began to dawn on programmers that it didn't really matter how the data was *stored* in the computer; what differentiated one data type from another was the *kind of operation* you had to perform to access the data in the structure. Thus, you have a list data structure when you can access any piece of data in it by a command (operation) which in effect says, "Get me the k'th item in the structure." You have an array data structure or table when you can access any piece of data in it by an operation which says, in effect, "Get me the data in the i'th row and the j'th column of the structure."

So the key idea here is that a Thing *is defined by the operations that can be performed on it*. Period.

We adapt this idea to Environment design by the requirement that every type of Thing, e.g., drawing, paragraph, document, schema, query, program, command interpreter, etc., have associated with it, in the Environment, *all and only* the tasks (operations) that can be performed on the Thing, with an explicit reference to the location in the Environment where the procedure for performing the task can be found. Thus, in the case of tabs in a desktop publishing system, these operations include:

look up the definition of the word *tab*.

define (set) a tab.

delete a tab.

indent a line of text to a pre-defined tab.

move an existing tab.

view all currently defined tabs.

If this were, in fact, the complete list of operations on tabs, then the user would know immediately that any other operation he or she wanted to perform on tabs, was not possible.

In an operating system like Unix, of course, the list of Things is large, and includes directories, files, permissions on directories and files, processes, jobs, environment variables, users, shells, and much more.

In the case of complex Things, e.g., data models, schemas, data bases, it is a good rule to make the first task in the list, "Get definition and background on ...." with a reference to the appropriate prose. This prose, of course, can be in traditional book format. It certainly does not in itself need to be task-oriented. Making it available at the head of the task list is an example of what has been called *just-in-time learning*: the user does not have to read the material until (s)he is ready to use it.

In every Environment, each Thing must have a page or screen that lists all and only the operations that can be performed on the Thing. Some examples are given in the appendices.

Basic Tasks versus Complex Tasks

The astute (i.e. skeptical) reader may well raise an objection at this point, namely, that it is in fact *impossible* to list all the tasks associated with any given Thing, because the list is infinite! Consider the computer keyboard as an example of a Thing. It is true that the tasks I can perform on a keyboard include typing each letter and number marked on the keys (a finite number of tasks), but I can also type any arbitrary finite sequence of letters and numbers, and the list of all such sequences is infinite (*countably infinite* as mathematicians say, meaning that the items in the list can be matched, one for one, with the positive integers). The same is true for the tasks associated with any other Thing. So how can I require a list of all and only the tasks that can be performed on each Thing? The answer is that this list, as in the case of the list of keys on a keyboard, must consist of all and only the *Basic Tasks* that can be performed on the Thing. In principle, *any* task that can be performed on the Thing, no matter how complex, *must* be able to be performed by a finite sequence of these *Basic* Tasks. (A simple example of a set of these Basic Tasks is given under "index" in the partial FrameMaker Environment in Appendix A.) Of course, it is perfectly legitimate — and highly desirable — to include other tasks in the list, specifically tasks that are performed frequently. The Basic Tasks are simply the minimum acceptable set.

In chapter 5, I make the outrageous suggestion that Environment designers should have a background in at least some of the more important subjects in undergraduate mathematics and computer science. The reason I give is that these subjects supply important *templates* for thinking about problems that arise in the development of Environments. The problem of basic versus complex tasks is an example. It is a problem which was first confronted by the ancient Greeks as a result of their attempts to develop a minimum set of axioms for geometry — a set from which all the truths of geometry could be derived. The problem received expanded attention toward the end of the nineteenth century as a result of researches into the foundation of mathematics. It became a central issue in computer science in the 1950s and 1960s as computer scientists began to confront the problem of compiler design, a problem which involves the study of formal grammars. (A formal grammar is a generalization of the notion of a minimum set of rules that yield an infinite set of strings of symbols.)

An Environment designer who knows something of these subjects has tools to deal with, to think about the question of listing the tasks associated with a given Thing. Others, I can only suppose, will attempt to deal with the problem in the usual ways: by attempting to write still better explanations ("clearer prose"), or another manual, or

perhaps by using hypertext and a GUI and multimedia. (I hope there is no doubt in the reader's mind as to what I am criticising here: it is not these various means in themselves, but rather the use of them inappropriately.)

Properties of Things

So far, we have discussed Things and tasks on Things. To complete the picture, although we will not exploit it in this book, we must understand that Things — directories, files, documents, paragraphs, drawings, disks, applications, data bases, schemas, users — have *properties*. Normally, tasks operate on these properties. For example, the properties of users include the user name (or names), the user's present role or set of permissions, the user's home terminal or workstation, his or her home directory, etc. Putting it as simply as possible: each Thing has properties and among these properties are the basic operations that can be performed on the Thing.

The Universal Flow Chart for Tasks Performed by Software

The Universal Flowchart for Tasks Performed by Software, shown in Fig. 3-1, describes the *use* of the Environment at the level where a task is entirely performed by the computer software. Such a task is called an *atomic* task; an example is clicking the Print button after you have entered all the required information in a Print... dialogue box; or clicking the Compile button in a programming Environment; or typing a command in Unix. An atomic task causes something to happen as opposed to merely bringing up more choices. The flowchart is mostly self-explanatory. The "you" in the figure is, of course, the user, not the machine. The following are a few comments on the nodes in the flow-chart.

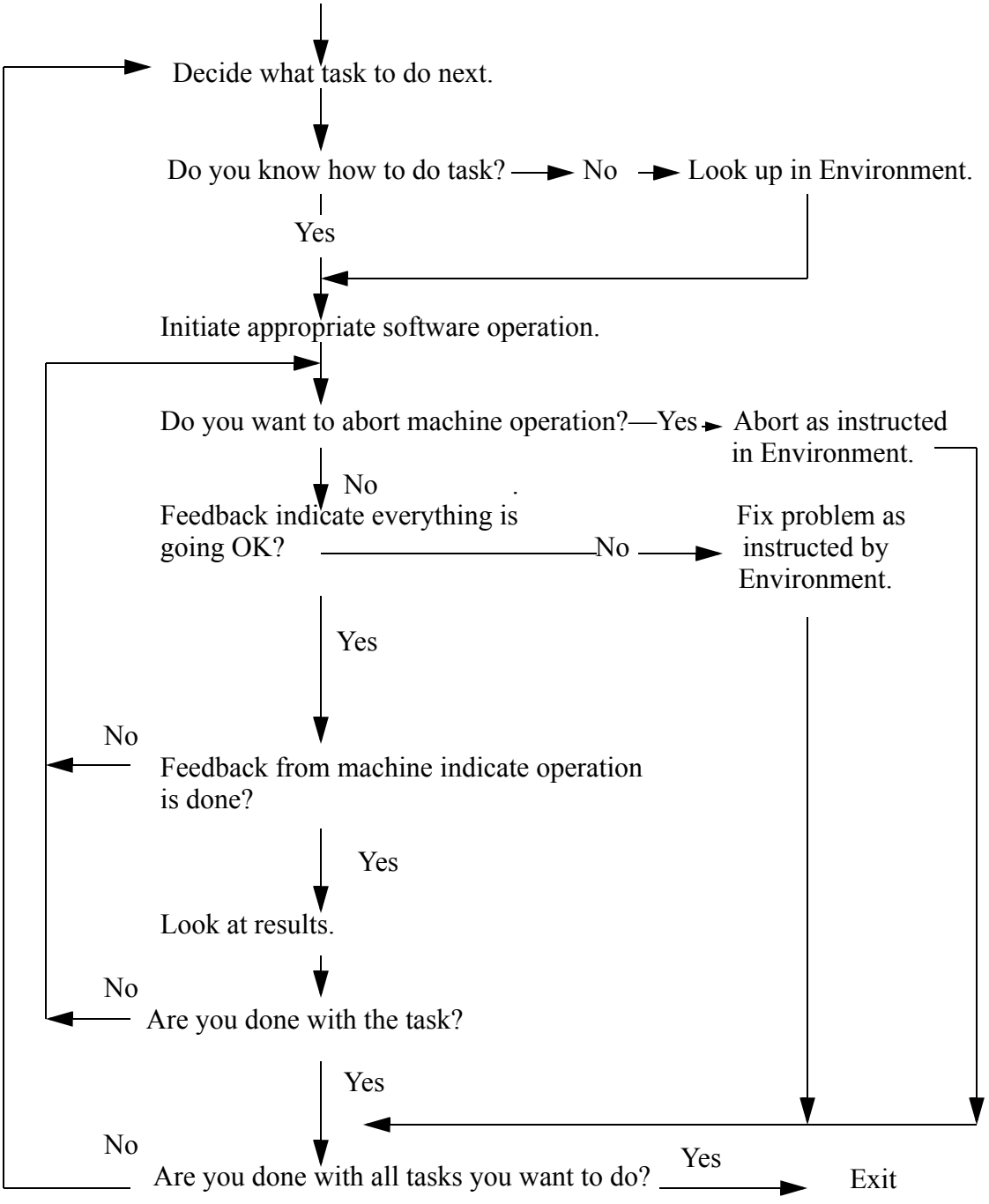
Decide what task to do next. The assumption here is that the user is confronted by more than one task, each of which is entirely implemented by software.

Feedback indicate everything is OK? Nowadays, many software systems employ various visual devices to indicate that processing is proceeding in what the

program(s) believes is the normal way. The Apple Macintosh, for example, displays a little wristwatch, or a parallel-moving bar, to indicate this. PCs display an hourglass.

Initiate appropriate software operation. This is normally done by clicking, or double-clicking, a mouse key, or by typing a command.

Figure 3-1 The Universal Flowchart for Tasks Performed by Software



The Index: Most Important Part of Any Documentation System

"All I want from a manual is a good index, so I can look up what I need to and get out of there. Unfortunately, the only purpose most indexes serve is to allow the company whose manual it is to say, 'Look! Look! An index!' And it's true — they do *look* like indexes. Why should I spoil the illusion by pointing out that you can never *find* anything in them?

"As far as I'm concerned, if the index is worthless, so is the manual. And if the manual is useless, so — usually — is the program (the exception being those programs that are so intuitive, so easy to use, that you don't need the manual at all). — Naiman, Arthur, et al. *The Macintosh Bible*, 4th ed., Berkeley, Calif: Peachpit Press, 1992, pp 45-46.

Such programs you now know are simply programs with efficient on-line Environments.

The *Mac Bible* authors are, again, right on the money, and in recent years I have found a reliable criterion of the real intelligence, the *sharpness*, of any author — regardless whether their subject is computers or mathematics or any science or any humanities subject — to be their attitude toward indexes. If the author considers the index as a last-minute nuisance, something added on to a book, I know that I'm dealing with a person who doesn't think about what he or she is doing.

There is a systematic way to index any technical subject, and to check the index of any technical document, and that is via the approach called *abstract data-type (adt) indexing*. This is simply an index constructed in the way described above under "Tasks and Things". The rule for looking up something in an adt index is simple: (1) Think of the Thing you want to perform a task on, e.g., if you wanted to copy a file from Unix to a PC, you would look under "file"; if you wanted to establish a connection to a server, you would look under "connection" or "server"; (2) Find the task under that Thing and go to the referenced location in the document. Thus, for example, under "file" the task might be listed as, "copy a, from Unix to PC". Other tasks listed under file would be "copy a, from PC to Unix"; "copy a, from Unix to Unix", etc. The concept of adt indexing not only provides a simple rule for creating indexes, it also provides a simple means of checking the completeness — the *usefulness* — of any given index. Unfortunately, it also reveals the wooliness of most professional indexers' approach to their work, an approach which is derived entirely from the liberal arts, and hence is void of any comprehension of the real nature of technical subjects.

Like most of the ideas in this book, the concept of adt indexing does not go down easily with most technical writers. First of all, they wonder at the need for such a single, uncompromising rule which clearly restricts their creativity (when they have time to do indexes at all), and, second, which clearly requires more of their time than normal (read: haphazard) indexing. The answer is that adt indexing is essential for achieving 25-second look-up-ability, because it eliminates the need for the user to *search* the index for the reference (s)he wants. (I am taking the liberty of not calling the looking up of something alphabetically "searching.")

One question that has been asked about adt indexing is: why not index on verbs instead of on Things? The answer here is, first, that verbs are *not excluded* from adt indexing! Certainly the verb "abort" and its synonyms and near-synonyms, "shut down," "turn off," "kill," "terminate," must be in any index to software documentation, adt or not. But the reason why verbs should not be the central indexing focus is that there are far more synonyms for even the most common verbs, than there are for Things. There are not many synonyms for "file," "server," "data base," "paragraph," "sentence," "figure," "table," but there are many synonyms for "delete" (e.g., "remove," "erase"), for "create" (e.g., "make," "write," "produce," "construct"). The more synonyms, the more searching the user might have to do to find the reference (s)he wants.

The Waste! The Waste!

Think of the thousands of users of any popular product: Windows, FrameMaker, Unix,... Each day, some of these users need to perform tasks which they don't know how to perform — ordinary, perfectly reasonable tasks, such as those listed in the previous section and elsewhere in this book. Somehow or other — by trial and error, by searching through volumes of manuals, by using this or that on-line search engine and wading through twenty or thirty hits, by bothering the person in the next cubicle, or by some combination of these — somehow or other they find out how to perform the task. Think of all the users *repeating these same searches for instructions on performing the same task*, day after day, month after month, over the life of the product! Surely it is a natural question to ask, Why not have one user go through the effort once, record what (s)he finds, and then make that information easily available to all other users? What could be a more natural question? That user is, of course, the Environment designer (and the programmers that are his or her information

sources), and the way (s)he knows (s)he got all the "ordinary, perfectly reasonable tasks" is described in the previous chapter, under "Tasks and Things." The results of these searches go in the adt index.

All of which can be summed up by a slogan which is not original with me, but was created by the director and the marketing manager of a new product team: *Put the intelligence in the Environment!* That is precisely it: we want to put as much of the intelligence required to use the system, *in the system itself* — in the Environment — and not require that each user supply it, in addition to the intelligence each user *must* supply to use the system for his or her purposes. In other words, we want to provide the user with *pre-searched* documentation, and that is what an adt index makes possible.

The One-Two Punch: Task Orientation and Adt Indexing

Our goal is that, say, 80% of the time, users will be able to find out *how to do* what they want to do in less than 25 seconds. We have now seen the two ways by which that goal is achieved: (1) strict task orientation, so that, in principle, any user can accomplish any task made possible with the Environment by moving down the task tree, and (2) an adt index, so that any user can look up how to perform any task made possible with the Environment simply by thinking of the Thing (or Things) (s)he wants to peform the task on, then looking up that Thing in the index. That's the solution in a nutshell.

After-the-Fact Environments

Let me repeat: Environments *can* be created after the fact, i.e., after the software has been completed and the product is on the market! The only disadvantage is that if, in fact, the procedures for performing certain tasks are time-consuming and tedious and clumsy, there will be little chance to change the software at this late stage. The software will dictate the use instead of the use dictating the software, as it should. Similar thinking has been applied in the software community regarding programs. Here are two examples:

(1) If it is possible to write program-provers — i.e., programs which, given a high-level specification of a program, and the program itself, will tell if the program is correct — then why not write a program that will generate correct programs from high-level specifications to begin with?

(2) If it is possible to write a Help system which understands natural language inquiries about commands, then why not write a program that will execute those commands directly from a natural language request?

And so you can see that it is perfectly natural to ask: If it is possible, with great labor and time expenditure, to explain (some of) the use of a software system after the software has been written, then why not explain it *before* the software is written — meaning, why not describe the use structure *before* we write the software? That is the goal, but we can still design efficient Environments in the years before this practice becomes common.

Techniques for Developing Environments

The following are rules and guidelines that seem to me to be the most important in carrying out the day-to-day work of designing and building Environments. No doubt many readers are already using some of these. I don't claim that each is the best of its kind, only that, at present, after many years in the business, it is the best that I know of. The order of presentation is not significant.

Get the Environment to the Users (and Reviewers) as Soon as Possible

The essence of incremental design is that the Environment is *always* usable, even though it may not be "complete", relative to all the tasks it intends to describe. From the first day — or, at least, the first month, say — the Environment should be in place. If part or all of the Environment is on paper, then the clearly labeled binder should be next to all workstations on which the software is being developed and/or tested, with a pencil attached, and a sign encouraging all users to jot down on the pages any complaints, thoughts they have, especially information they were unable to find in the Environment. Smart project leaders will make the raises of programmers

and engineers subject to, among other things, their contribution to the development of the Environment.

Proceed by the Rule of Maximum Disambiguation

The Rule of Maximum Disambiguation — or, more precisely, of Maximum *Rate* of Disambiguation — says that, in general, you should work next on that part of the Environment which, for a given amount of effort, *will reduce user uncertainty the most*. Thus, if nothing at all exists in a sub-Environment about a certain task, then your first question should be, given the class of users, what are they most likely to be unsure about? How much *could* you expect them to figure out for themselves, if you and all engineers and programmers dropped dead tomorrow? This last question is not a contradiction to our goal of creating an efficient Environment, namely, an Environment in which users never have to figure out where procedures for carrying out tasks, are located. This question is merely a guide toward achieving that goal — a way of establishing priorities of things to be done.

Another way of implementing the Rule is by asking yourself: Suppose I only had *one more day* to work on the Environment: what would I put in it, what do I have reasonable grounds for believing the users could figure out for themselves? Which demonstrates again the importance of knowing the skills and knowledge of your class of users.

Still another way of implementing the Rule is by imagining that you are participating in a contest in which $1 million will be awarded to the Environment designer who enables a set of users to accomplish the most tasks with the fewest errors in the shortest time after the start of the Environment design. The users, whose skills and knowledge are known to the designer, receive no information about the product except what is in the Environment.

Virtually *any* representation of task information will reduce ambiguation: to go to extremes: hand-written text, hand-drawn figures would serve the purpose initially, since they give the user more information than he or she had previously. Some later pass at disambiguation might then be converting this hand-written, hand-drawn information into a typeface. Some much later pass might involve insertion of italics, boldface, or fancier typefonts.

More time is wasted on fussing over typefaces and formats than anything else in the development of documentation. (Typefaces and formats certainly wouldn't be

your main concern if you were out to win the $1 million!) Technology is seductive! It invites us to waste time on premature refinements: after computer typefaces became available on desk-top systems, middle managers earning upwards of $75,000 a year (that's more than $35 an hour) were spending hours in preparing memos and reports which would have been perfectly legible if written in a single typeface — even, in most cases, if written in the author's handwriting!

Similar arguments apply to the use of spelling and grammar checkers, and to page layout refinements in column widths and margins. The types of spelling and grammatic error that you make are rarely so bad that you would literally be unable to figure out what you meant when you make the next pass through the document. Use spelling and grammar checkers just before you are ready to print a copy to submit to others.

Alphabetical Titles = Numerical Titles!

Most writers don't realize that a common form of index entry can be used *as a form of title* which serves exactly the same purpose as the traditional numerical form does. Here are two examples:

File, ASCII, creating a
.
.
.
File, binary, creating a

which would correspond to the traditional

5.3.1 Creating an ASCII File
.
.
.
5.3.2 Creating a Binary File

There is no reason why we shouldn't take advantage of this highly useful characteristic of alphabetical ordering! It parallels the form of entries in an adt index,

it makes instantaneously clear to the user where (s)he is in the documentation structure, and, as shown in appendices A and B, it makes possible the unifying of index and text, at least in paper implementations of Environments. Getting used to the new format is a small price to pay for the enormous increase in speed of access which the format makes possible.

The Proper Place of Editing

The old ways die hard. Sometimes, even at this late date, you will run across a manager who believes that before documentation is released, it must be "edited." This view is, of course, a relic of the book era, when books (e.g., manuals) went through a series of discrete stages prior to publication: first they were written, then, if the author was lucky, they were edited, then typeset, then proofread, then printed, then distributed. In modern documentation, including Environment design and development, editing is an ongoing process: it is part of normal quality assurance. Furthermore, it is far less important than it is in the case of old-fashioned, pre-Environment documentation, because much of what used to be handled by prose is now handled by form, e.g., the form of recursively structured tasks. Ultimately, when Environments have advanced to their inevitable final form, namely data bases (see next chapter under "Environments are Data Bases Whose Content is the *Use* of Software Systems"), the need for editing will be all but eliminated.

One and Only One Name for Each Thing!

Settle on one and only one name, or term, you will use for each task and Thing throughout the Environment. Obviously you should try for the term that will be recognized by most of the potential users. Then, of course, you must include all reasonable synonyms, and refer each to the term you have chosen. This is also more efficient, since it saves you duplicating information under several different terms. This practice should apply to commonly occurring phrases too.

Include Definitions of Terms Already in User's Minimum Vocabulary

Strictly speaking, you never explain anything in an Environment that is already in the minimum vocabulary of your intended users. But good design practice always includes a safety factor, a degree of redundancy or overlap, which, in this case, means that you *do* include definitions of those terms and phrases which are in the user's minimum vocabulary but which you suspect, or know from usability testing, have more than one meaning.

Handle Error Messages the Right Way

Users probably waste more time in trying to figure out error messages than in any other information searching activity. With our task-oriented, zero-search-time orientation, however, the solution is simple. Ask, first, what *task* is associated with an error message? (In Figure 3-1, the occurence of an error message is indicated by a "No" reply to the question, "Feedback from machine indicate everything is going OK?") The associated task is that of fixing the error so that the original task can proceed ("Fix problem using Environment"). Therefore, each error message must have associated with it a reference (typically a number) to a location in the Environment where information on corrective tasks will be found. But the user should not have to figure out that location (or else we won't be able to deliver on our 25-second maxmum look-up time). Therefore, in the index there need to be entries such as, "errors," "error messages," "problems," "problem-solving," "trouble," "trouble-shooting," "warnings," "warning messages," and anything else that user experience plus your own intuition tell you might be a heading under which users will look for explanations of error messages. Each entry should refer to the one section (or two, if you treat warnings separately) where the explanations and corrective tasks are given. If messages and/or warnings begin with numbers alone, then there must also be a reference to the explanatory section in the section of the index where strings beginning with numbers are listed.

In the explanation section(s), messages should be listed alphabetically or by number, with possible causes, and possible remedies, and at least a statement at the start about calling Customer Support (phone numbers and hours). Nothing less will do.

Finally, even though error messages are almost invariably written by engineers and programmers, for whom English is an annoyance that gets in the way of the important things in the world, namely, machines and programs, do whatever you can to make error messages clear and grammatically correct. Statements still begin with capital letters, and end in periods or exclamation points. The terms in error messages are *not* exempt from the rule given above under "(1.1) Establish the Class of Intended Users," namely, that every term in the Environment must be either in the user's minimum vocabulary, or explained somewhere in the Environment in terms of that vocabulary.

Maintain Consistent Meaning of "I" and "You" in All Documentation and Messages

I suppose this must come under the heading of a minor point, but since our goal should be to create *superb* Environments, in which details have been attended to, and not merely to create acceptable Environments, it needs to be mentioned. If the software (i.e., the computer) speaks to the user as "I" in one message, then it must speak to the user that way in all messages. If "you" means the user in one message, then "you" must mean the user in all messages. On the other hand, if it is deemed more desirable to couch messages in objective language — "No such file in current directory" — then that language must be maintained in all messages.

Embed Questions and Notes to Yourself in the Text

During the course of creating an Environment, numerous questions will occur to you. Some will have to be answered by the engineers or programmers who are creating the software, others you will answer yourself. You will also want to remind yourself of things to be done. Since it is very easy to forget these, and since it is a nuisance to write them down on paper, the easiest, and most efficient, solution is simply to embed them into the text as you go. Nowadays the more advanced desk-top publishing systems make it possible, through hidden text and conditional text mechanisms, not only to hide these questions and notes whenever you want in order to

produce a clean, current version of the text, but also to make them appear obnoxious and ugly, through proper selection of typeface, and hence demanding attention! In more primitive systems, you can simply identify each question and note by an annoying symbol such as ***.

Warning! Be absolutely certain that the reviewers of your text understand that the questions and notes are not a permanent part of the text! I know of a man who was fired from a job because his manager — who had never supervised a technical writer or any other kind of Environment designer — thought that (a) if the writer had not been able to answer all questions by himself by the time he gave the document out for review, that could only indicate that the writer was incompetent, and (b) that users would be put off by a document that was full of questions and notes! (The manager did not understand that the notes and questions could be removed at the click of a mouse button by setting their display property to "hide" or the equivalent.)

Don't Waste Time Fussing With Style Guides

The Rule of Maximum Disambiguation says, among other things: Don't waste time fussing with style guides! If one exists, use it, modifying it only when potential confusion of the reader is at stake or when company dictates demand it. If none exists, use an existing one for the industry you are working in, or one published by a professional association for the industry.

Arrange Information "For the Eye"

Write for the eye. Write so that the user can understand what to do as quickly as possible!

Prose is good for one or two readings, bad thereafter. Typically, we remember the gist of a prose instruction, but forget the hard information therein, e.g., the values of command parameters. Set commands apart by placing them on separate lines.

Lists are *much* easier to read in vertical, stacked form than when strung out end-to-end in prose

Encode Syntax Rules

Sooner or later you will have to deal with syntax issues which occur again and again throughout the Environment. Specifically, you will have to decide what indents, typefaces, and *language* you will use to express:

- Commands, menu- and window-selections made by the user, including variables
- On-screen responses made by the software
- Hypertext words and phrases
- Names, e.g., file and directory (folder) names

Nowadays, word-processors and desk-top publishing systems have ample facilities for encoding these rules, so that you can easily change them as desired during the course of Environment development and have the changes immediately take effect in all locations where the rules have been applied.

Use the Active Voice, Personify Programs, Refer to User as "You"

The old rule still holds: in general, prefer the active voice. Don't hesitate to personify programs, e.g., "The search program then tries to find a match for ...," "The output program doesn't care if the string represents a file name or a number."

Use "you", not "the user", e.g., "If you want the data printed in column format, then..."