CHAPTER 2

FUNDAMENTAL CONCEPTS FOR DESIGNING ENVIRONMENTS

Importance of Fundamental Concepts for Designing Environments

The means for making software usable, as I have defined the term in the previous chapter, is through Environments. Environments are explained in the next two chapters, but it is essential that you have a clear idea of the fundamental concepts we use in this design task. If you are sorely pressed for time, you can skip this chapter and go on to the next, then come back to this one later on. But if you find yourself starting to develop Environments on an ongoing basis, you should make a point of reading this chapter sooner or later.

I sometimes feel that the concepts in this chapter should be called "the fundamental concepts of Western civilization", although that would be an exaggeration, since there are certainly other fundamental concepts as well. But the ones in this chapter are so important in all technical subjects (not only computer science) — indeed, in all technical *thinking* — that if you do nothing more than understand them and start to apply them, you will have gotten your money's worth out of this book.

The first of these fundamental concepts is the What versus the How, or Semantics versus Syntax.

Fundamental Concept 1: the What versus the How, or Semantics versus Syntax

We will begin with an example.

Example 1

The following story has been told many times. I do not know its source — it seems to have first appeared in the early seventies — or if this version is the same as the original one, but that is not important as far as the point of the story is concerned.

In a high-school physics exam, the question was asked, "Suppose you had only a barometer and were asked to measure the height of a very tall building. How would you do it?"

The teacher received various answers from his students, including the one he wanted. However, one answer he didn't expect was the following: “I would tie the barometer to the end of a very long string, go to the top of the building, lower the barometer until it touched the ground, then measure the length of the string.”

The teacher marked the answer wrong. The student protested. The teacher agreed to give the student another chance. This time, the student answered: “I would go to the top of the building, drop the barometer off, and time how long it took to reach the ground. Then I would use the formula

$$s = \frac{gt^2}{2}$$

where:
g is the acceleration of gravity,
t is the number of seconds the barometer took to reach the ground, and
s is the height of the building.”

Again the teacher, not getting the answer he wanted, marked the answer wrong. Again the student protested, and now his parents joined in. Eventually the teacher again relented and agreed to give the student one last chance. This time the student wrote the following on his answer sheet: “I would go to the superintendent of the building and say, ‘Here, Mr. Superintendent, I will give you this nice new barometer if you will tell me how tall this building is.’”

How long the controversy raged, or what its final outcome was, I do not know, but the lesson of the story is clear, namely, that there is seldom only one way to do something. Or, in other words, for a given What (finding the height of a very tall building), there are usually many Hows.

The What: measure the height of a very tall building.

The Hows: the three ways described. You can probably think of at least one or two additional ones.

Here are a few more examples.

Example 2

The What: Go from your house to work.

The Hows: Go by car, bus, train, bicycle, on foot, or use some combination of these. Of course, some ways are better than others, depending on what is most important to you at the time (e.g., speed of getting there, convenience, cost, not polluting the environment).

Example 3

The What: Obtain a master's degree in technical communications.

The Hows: All the different colleges and universities you could attend. Some, of course, are better than others (e.g., will enable you to get a higher paying job, or are cheaper).

Example 4

The What: Solve the following quadratic equation for x:

$x^2 + x - 2 = 0$.

The Hows: Among these are:

1. Use the quadratic formula we learned in high school math courses. It works for any quadratic equation:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where a is the coefficient of x^2 (hence $a = 1$ in our example), b is the coefficient of x (hence $b = 1$ in our example), and c is the constant standing alone (hence $c = -2$ in our example).

We find that $x = 1$, $x = -2$, are the solutions.

2. Factor the left-hand side of the quadratic equation by trial and error and find that

$(x - 1)(x + 2) = 0$,

hence

$x = 1$, $x = -2$ are the solutions.

Semantics versus Syntax

Other terms for the *What* versus the *How* are *semantics* versus *syntax*. In normal everyday circumstances, what you want to say (what you mean, i.e., the semantics) can usually be said in many ways. A grammar book for a natural language gives you the rules for stringing words together, i.e., the syntactic rules, but it only incidentally discusses the meanings of the strings of words.

In computer programming, the distinction between *What* and *How* is especially clear. The *What* is the function, e.g., addition or subtraction or multiplication or division or sorting a set of numbers or accessing data from a data base or displaying information on a computer screen in some format; the *How*s are the various programs that can compute (implement) the function. Some programs, of course, are better than others for the goals at hand; some produce an answer faster than others, some programs are easier to write and test than others, some require less memory space, etc.

In Environments, the *What* is the *task* that the user can perform; the *How* is the subtasks into which the task is broken down. Eventually, of course, we reach a subtask that can be entirely performed by the software.

Knowing how to recognize the difference between the *What* and the *How* is one of the most important skills you can have, and one that will serve you in fields outside of Environment design or technical communications. In industry, one of the most common, and most expensive, failings of managers and the engineers who work for them, is the belief that there is only one way to develop a new product, namely, by designing it from scratch. But in fact the best design solution is often to do *as little* original design as possible, and to use *as many* existing parts and assemblies as possible— something that most new engineers do not like to realize. But that realization can only come to one who is used to asking himself, *What* is the goal here? and then, *How* can we go about accomplishing it? and then Of all the ways of accomplishing it, which are the best for our purposes?

Fundamental Concept 2: Structure, or Breaking Complex Things into Simpler Things

The advantage of breaking complex things into simpler things is so taken for granted in Western culture that we usually consider it obvious. We break speech into certain constituent sounds, then represent those sounds with strings of letters. We (and nature) break matter into molecules, then molecules into atoms, then atoms into subatomic particles. We break a large business corporation into divisions, then each division into various departments, e.g., Research and Development, Marketing, Manufacturing. In the U.S., we break the government into the Executive, Legislative, and Judicial branches. All perfectly obvious. But the recognition of the importance of the concept — the recognition of its nearly universal applicability — was, as far as I know, limited to the West until Western culture began to spread throughout the world in the nineteenth century. Perhaps, as Marshall McLuhan, the sixties philosopher of communications media argues, the reason this was so clear to the West was that, from the time of the ancient Greeks, literate Western man has had the benefits of an alphabetic writing system staring him in the face. I don't know. But in any case, this is an example of an idea whose importance is in no way diminished by the fact that its value is obvious.

Breaking a complex thing down into smaller things is a way of expressing the idea of *structure*. One field in which complexity is always in danger of overwhelming those who work in the field is computer science, in particular, computer programming. Thus, soon after the art of programming came into being in the 1940s, programmers recognized the need to break programs up into more easily manageable pieces which became known as *subroutines*. A subroutine performs a specific task, e.g., properly handling the carries in addition, or printing the result of a calculation. Strangely enough, the importance of structure, or, rather, of certain kinds of structure over other kinds, did not become clear to the programming community until the early 1970s. The story, in brief, is that in the March 1968 issue of *Communications of the ACM* (the Association of Computing Machinery), there appeared a letter to the editor titled, "Go To Statement Considered Harmful." A go-to statement is a statement in a computer program that commands the computer to execute not the next sequential statement in the program, but some other statement elsewhere in the program. In effect, it enables the programmer to introduce all sorts of special cases into the process by which the program performs its computation; it enables him or her to make the program "jump all over the place" during the course of a computation.

According to the author of the letter, computer scientist Edsger Dijkstra, this leads to programs that are difficult to understand, hence difficult to check for correctness and difficult to debug. Hence the go-to statement should be avoided as much as possible.

This letter is generally considered the beginning of the programming methodology now known as *structured programming*. In 1972, Dijkstra, along with O.-J. Dahl and C. A. R. Hoare, published a book, *Structured Programming*, which set forth the structured programming methodology in more detail. (This methodology was based on the use of block-structured, or Algol-like (nowadays, Pascal-like, or C-like) languages.) In 1976, Dijkstra developed the idea still further, introducing a method of writing correct programs (i.e. programs that compute the function we want them to compute) by, in effect, writing them in a way that permits the programmer to easily prove the correctness of each successive approximation to the final, complete program.

As with many new ideas, the technique of structured programming, as well as its value, was obvious after it had been pointed out. In fact, structured programming is nothing but the application to programming of the old technique of outlining that writers of term papers learn to use (sooner or later). Structured programming is simply a method of breaking down a large programming task into a manageable set of smaller programming tasks, and then breaking each of these down into a manageable set of still smaller programming tasks, etc., and doing so in a way that enables the programmer to prove the correctness of the program as it develops. In other words, structured programming is a prime example of breaking a given *What* down into a *How* consisting of smaller *What*s.

The idea of a structured program lies at the very root of our solution to improving usability, namely, the development of Environments. An Environment is like a structured program with the user as the computer that executes it. This is an important point, and I will explain why.

The traditional view of a computer, its software, and the user of both, is shown in Fig. 2-1a. Here, the computer is a machine (a "black box") *over there* that is capable of solving certain problems. The user of the machine is a separate entity that brings problems to the machine for it to solve — specifically, gives inputs to the software in the form of commands and data which may be in a file, or accessed by the software from some specified remote source, or else manually input by the user. The user starts the software and hardware running (the hardware being a central processing unit (cpu) and memory), and together they then perform various computations, or, more correctly, information processings, and produce the result as output. The computing entity, in this view, is the computer and the software, as shown by the dotted line in the figure.

The new view, and one which leads to the design of software that is much easier to use, is that shown in Fig. 2-1b. Here, the computing entity is hardware, software, *and user.* The *user* is the "central processing unit" that runs the hardware and software. The user's "program" is the user Environment — the keyboard plus the terminal screen plus the manuals. (I will write this type of program in quotation marks because it is not a program in the accepted sense of the word, i.e., it is not necessarily capable of being executed by a machine.) The input is given to this entire entity — by another person who gives it to the user, or by the user bringing it him- or herself. Similarly, the output is an output from this entire entity.

All of which is summed up by the motto, "Not human alone, nor machine alone, but human and machine as a unit" — a motto for the design of the computers and software of the future.

Viewed in this way, it is clear that the "program" (Environment) that makes computer *and* user solve a given problem is only partially the software that runs the computer. The Environment must also tell the user how to operate the computer so that the computer's software can then complete the work of solving the problem.

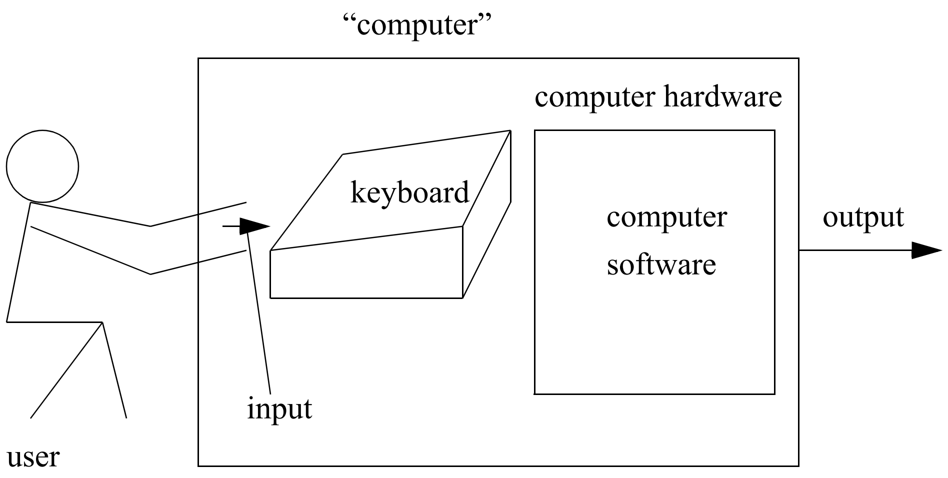
The "program" (i.e., Environment) is what computer scientists call *non-deterministic*, in the sense that there are usually a variety of ways by which the user can achieve the desired result: (s)he may have a variety of different programs to use, some of which may be interactive, meaning that the user may guide the computation as it proceeds (this type of operation is also known as *on-line*); some may not be interactive, meaning that once the computation is started, it runs to completion without user intervention; this type of operation is also known as *off-line* or *background*.

Several things become clear from this new point of view. First of all, the obsession of most computer hardware designers and programmers with speed of computation — i.e., the speed at which the hardware cpu can execute program instructions — misses the point. Equally important, and probably more so, is the average (or typical) speed at which a given type of problem can be solved or type of "job" can be processed, e.g., a job such as typing and printing a report, or obtaining a certain type of information from a data base. The speed to increase is the speed at which the average user *plus* the computer can solve a given problem. This speed must be averaged over *all* users, including first-time users. In the computer industry, the rate at which jobs can be processed through a system is sometimes called the *throughput*. The speed measurement must include all training, if training is in fact necessary in order for first-time users to use the system, and/or all time spent reading the manuals, and/or all time getting help from the manufacturer's customer support service. It is by no means always the case that increasing the computer's speed will increase the throughput, especially when it takes hours, even days, to figure out how to make the computer solve the problem in the first place.

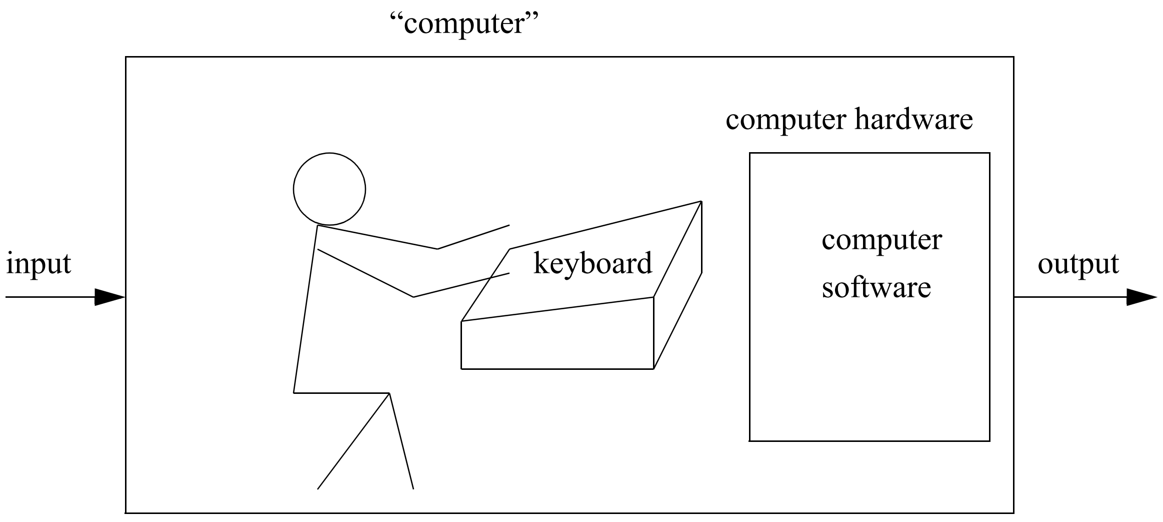
An analogy to Environments outside the field of computers is that of an astronaut in a space suit exploring the moon. The astronaut alone does not carry out such exploration, nor does the space suit alone; both together do.

You should know that the idea of regarding the user of a computer Environment as a kind of central processing unit (cpu) that executes the non-deterministic "program" which is the Environment, in order to solve the problem represented by the input — that this idea is considered a giant step backward by some computer scientists, especially by those with a vested interest in some of the woolier branches of the discipline, e.g., cognitive science. "Man is not a machine!" these researchers proclaim. But, as you will see, the idea of an Environment is in no way aimed at making the user do dull, repetitive machine-like work. It is aimed rather at rendering "look-up-able" as much as possible of the information required for the use of a computer system — or, as I have shown in another book, *How to Improve Your Math Grades*, for solving problems in a mathematical subject.

Figure 2-1. Old and New Views of User and Computer



a) Old View of User and Computer



b) New View of User and Computer

Fundamental Concept 3: Algorithms and Heuristics

An algorithm is a mechanical procedure for producing an answer to a problem in a mathematically based subject. The phrase *mechanical procedure* means a procedure that can be carried out by a machine. (This definition can be made more precise using the concept of an idealized, simplified computer called a "Turing machine," which is named after the British mathematician who first used it in certain proofs in formal logic in the 1930s.) Actually, an algorithm is not merely a mechanical procedure, but one which is guaranteed to produce an answer no matter what the input. As it turns out, there are many mechanical procedures which will produce answers for *some* inputs, but not for all; in these latter cases the procedure may simply repeat certain steps over and over, forever. Some examples of algorithms are: the familiar rules for adding, subtracting, multiplying, and dividing which students are taught in grammar school; variations on these rules which are implemented in pocket calculators and computers; the procedures, implemented as programs, that operate automatic tellers in banks; just about all procedures, implemented as programs, for processing information in business, e.g., inventory control, payroll check writing, employee record keeping, etc.

A heuristic, on the other hand, is a procedure, and not necessarily one that can be performed by a machine, which may or may not yield an answer, but which experience suggests will do so in most cases. Webster's New Collegiate Dictionary defines the term *heuristic* as "of or relating to exploratory problem-solving techniques that utilize self-educating techniques (as the evaluation of feedback) to improve performance." In this book, I will use the term *procedure* instead of *heuristic*.

In paper implementations of Environments, you normally write procedures which, in principle at least, are algorithms, e.g., a procedure to make all page numbers in a preface print as roman numerals, or a procedure for transferring a file from one disk drive to another. In fact your goal is to reduce as many tasks as possible to such procedures! We will say more about this in the next chapter.

Fundamental Concept 4: Alphabetical Order

You may not be inclined to consider alphabetical order a particularly important idea, but it is. For one thing, it is an order which every literate person knows, and it works systematically for strings of letters and numbers of any length.

Do not take alphabetical order for granted! In many languages, its equivalent is difficult, if not impossible, to achieve. In ideogrammatic languages such as Chinese, for example, the nearest thing to alphabetical order is a listing of written characters in terms of the number of strokes needed to make them. Thus all one-stroke characters come first, then all two-stroke characters, then all three-stroke characters, etc. But there is no systematic way to further break up each set of characters having a given number of strokes.

As you develop Environments, you will also develop a new appreciation of this natural, universal (in the Western world) ordering. For some interesting speculations about the extraordinary influence that alphabetic systems of writing may have had on Western man — in particular, on the *thinking* of Western man — you might enjoy some of Marshall McLuhan's books, e.g., *The Mechanical Bride*, *The Gutenberg Galaxy*, *Understanding Media*.

And now, with these fundamental ideas as a basis, we can get down to the business of constructing Environments.