The Open Channel is exactly what the name implies; a forum for the free exchange of technical ideas. Try to hold your contributions to one page maximum in the final magazine format (about 1000 words).

We'll accept anything (short of libel or obscenity) so long as it's submitted by a member of the Computer Society. If it's really bizarre we may require you to get another member to cosponsor your item.

Send everything to Jim Haynes, Applied Sciences, UC Santa Cruz, CA 95064.

Continued from July . . .

Thoughts on Computers as Behavior Fields

Peter Schorer
Hewlett-Packard, Inc.

The process of design is itself a behavior field surrounding (in the case of software design) such objects as existing computer systems, pencils and paper, listings, etc. The function being executed is the design function; the output is some new computer system.

In real life what happens is that the designer(s) dream up a new system, perhaps a very elegant one which in fact tries to optimize its behavior field, and then they "somehow" bring the system into existence. That is, they forget that they are working in a behavior field throughout the design process, and (just as in the writing of correct programs a la Dijkstra and Wirth) the nature of this field may in fact determine how rapidly and successfully their system is implemented.

A useful technique is to consider turning structured programming "on its side"—i.e., lay it horizontally along the time axis. Then we consider the design process as a recursive operation taking place in increments of, say, weeks. That is, after each increment, we aim at having a "finished" product, except that, at least initially, one in which various functions to be performed by the final product can only be performed rather awkwardly. In other words, we consider the product as always, from the very first few weeks onward, to be in the hands of "users." Indeed, there is no difference between "users" and "designers," no more back room where the initial dirty work is done. A version of the product is always finished and available for use. It could be shipped at any time. (I am deliberately simplifying and exaggerating somewhat here to make the point clear.) Further work on the system simply attempts to make it more convenient to use for some specific application—i.e., to optimize the behavior field for that application.

There should be no difference between "users" and "designers," no more back room where the initial dirty work is done.

This whole approach arises from, and is justified by, the fact that the computer is a tool which can be adjusted to its environment—unlike, say, a hammer. Furthermore, as has been observed, the computer usually changes its environment, or, in the jargon of this paper, it changes the existing behavior fields into which it is introduced—e.g., the way incoming orders are handled. Thus, it is wrong to talk about building a computer system that will fit a given context, since it will very likely change that context anyway.

This fact is reflected (though not yet recognized) in industry, where a programmer often has the feeling that his product never seems to be truly "done." The specifications change during the actual design process; they change as a result of initial use at test sites; they change after the system is first sold to customers; they change during the life of the product. Yet the whole design process is set up as though one could speak of stages at which certain things were "done," meaning "finished," no longer to be changed except in extreme emergency.

Programming in Lisp is a good model of the horizontal structured design I am talking about (as well as of vertical structured design!). There are five primitive functions (assume that these are implemented in the initial week—i.e., in the first increment). With these five primitives, any Lisp program can be written. Once these primitives exist, the system is complete, though by no means necessarily "easy to use" for all applications. Furthermore the entire syntax of the language can be written on a single page; the rules are the same no matter how simple or complex the program. One does not need to run into new special cases as with so many other programming languages. Thus, after the five primitives are implemented, there is truly no difference between "user" and "designer." (The Adam System produced by Logical Machines Corp. seems to implement this idea.)

Is there a way of determining what an existing behavior field is—e.g., the behavior field surrounding an existing computer system? Yes. In essence it consists of determining the set of instructions (in short, the "program") which any of the normal users of the system must know in order to carry out any of the functions which the system supposedly implements.

Determining an existing behavior field in this manner can be an interesting and revealing process. Recall, in accordance with the method outlined above, that the rules require

(a) that the knowledge which any prospective user is to have when he first approaches the system must be specified in advance;
(b) that all means by which the new user obtains new knowledge about the system must be specified in advance; and
(c) that all subsequent tests of the system must check against these two conditions.

It is wrong to talk about building a computer system that will fit a given context, since it will very likely change that context anyway.

Consider a typical real-life situation:
(1) User sits down at console to perform some task. Gets error message consisting of a number or alphanumeric string he doesn't understand.
(2) Tries to find explanation in manuals. Can't. Eventually finds person who wrote the program, or one who has used it extensively.
(3) Knowledgeable person shows user that the error number is an extremely simple code which is explained in a docu-
ment the user hadn’t thought of, but conceivably could have thought of.

(4) User corrects error and proceeds.

Almost invariably, there are two results of such a situation:

(1) The user feels embarrassed for not having thought of the now obvious document and for not having kept up with the full story of where information on the system is.

(2) The knowledgeable person feels good about how easy his error messages are to understand—didn’t the user understand almost immediately how the code worked and where the explanation was kept and why it was kept in just that document?

An incredible amount of information must be “in the air” in order for most computer systems to be usable.

It is one of the main points of this paper that such situations are not desirable or necessary, and furthermore that they arise out of an almost complete blindness to the fact that we are always engaged in processes (here called “behavior fields”), and that it is the process as a whole that we must pay attention to, not “things” (e.g., this error message code, that document).

So, to find out what an existing behavior field really is, we try to write down every instruction a normal user needs to know apart from the background knowledge (abilities) we expect him to arrive at the system with. In other words, we try to write a program for a user of the system which alone will enable the user to successfully use the system. If at some point it is necessary for the user to refer to a given document, then the location of that document must be specified, as well as how he should search for the information inside it.

In my own experience, such exercises almost immediately reveal the incredible amount of information which must be “in the air” in order for most computer systems to be usable (“in the air” in the form of folklore, daily conversations, and notes on walls, human memory, etc.). It almost makes you want to say that the real sea of information is not in the machine, but all around it.

Is there a way we can talk meaningfully of the complexity of a computer system, where here we mean, “the complexity as far as user is concerned”?' I claim yes. And here I will borrow a very interesting idea about the nature of complexity which has been developed over the past few years by Gregory Chaitin and others. The subject is called “algorithmic information theory” and is established on a firm mathematical basis; I must emphasize that my very loose application of the idea here is entirely of my own doing.

Algorithmic information theory asks the question, “What do we mean by the ‘complexity’ of a finite string of 1’s and 0’s?” Consider two examples: first, a string of 1000 1’s and 0’s arrived at by flipping an unbiased coin 1000 times, where heads is 1, tails is 0. Second example: a string consisting of 10 repeated 500 times. The answer which the theory gives is as follows: the complexity of a string is measured by the length of the shortest program needed to generate the string. Thus the string produced in the first example is “maximally complex” because in most cases, the length of the minimal program needed to produce the string will be as long as the string itself (the program will essentially be a table with 1000 entries); in the second case, the minimal program is very short: “Print 10. 500 times.” So we say that the second string is of very low complexity, or, in other words, has a pattern. (The reason for the “minimal” program condition is, of course, that we can always write a very long, wasteful program to compute any string.)

So, the definition I want to make of the complexity of a computer system as far as the user is concerned, is simply this: it is the minimum length (as measured, say, in words) of the “program” (as defined above) which any person in the set of intended users needs in order to accomplish all the functions which the system offers.

The complexity of the system varies as the set of intended users varies.

Note how, under this definition, the complexity of the system varies as the set of intended users varies. Consider one extreme: the set of intended users is to be the original designers only. Then the complexity of the system will be relatively low, since the required initial abilities of any user probably includes (at least in the case of small design teams) a good deal of knowledge about the system. In short, there will be relatively little explicit instruction about the system required.

Consider the other extreme: the same system is to be usable by anyone who can read and write English and operate a typewriter. Now we see an extraordinarily large “program” will be required for any such user. Somehow he or she will have to be taught about powering up the system, about what can be done (in language) on the system, then how to do it, the names of commands, the grammar of the command language, the meaning of special keys on the keyboard, about error messages and where to look them up, perhaps about line printers and their operation (reloading paper, etc.)—the “program” would doubtless be so large that we would decide either to narrow the set of users, or take other steps to reduce their initial abilities (e.g., by training courses) or to redesign the system.

The important point here is that it isn’t how “simple” any given part of the system is that is important—after all, the value of any digit in a finite binary string is extremely simple also, it’s either 1 or 0—but rather how much additional information is needed by anyone in the intended set of users, in order to use all the functions which the system offers.


Shortly after the “Tale of Two Computers” was published (in the May issue) we received an odd-shaped, thin, oblong package in the mail. With memories of international intrigue and letter bombs, we handed it carefully over to the disposal squad; but on being opened it was found to contain nothing more sinister than a bumper sticker, which reads.

“HONK IF YOU LOVE JCL”

(Compliments of a source whose business partner wishes to remain anonymous.)

WATNEWS?

The Canadian universities have always been very professional about software development. One of the oldest and best known products, at least in educational circles, is a Fortran compiler done at the University of Waterloo, Ontario. It was originally named “WATFOR”; and if you must know, it’s for compiling and executing student-type Fortran jobs, being faster in compilation, slower in execution, and much more informative and thorough in error-checking than the manufacturer-supplied compiler. A few years ago a substantially enhanced version was released, and named (naturally enough) “WATFIV” (probably with no little help from Victor Borge). Waterloo has a number of other software activities, all of which are treated from time to time in a new publication called “WATNEWS.” It’s available at no charge on request to K. Ian McPhee, University of Waterloo, Computer Systems Group, Waterloo, Ontario, Canada N2L 3G1.

J. H.