acpa-v proceedings

For the first three days of October, 1975, the Carson Inn / Nordic Hills Country Club in Itasca, Illinois was the scene of one of the best conferences of 1975.....ACPA—V!

Keynoting the ACPA—V Conference was Illinois Attorney General William J. Scott, who spoke on Computers and White Collar Crime. ACPA—V also brought some of the best minds in the business of Information Processing, including Dr. Harlan Mills, Terry Baker, and Glen Myers of the IBM Corporation; Ed Bride of COMPUTERWORLD; and Theodor Nelson of the University of Illinois. ACPA—V presented an up-to-date picture of the Federal and State Privacy issues with such knowledgeable people as George B. Trubow of the President's Domestic Council on the Right of Privacy and Illinois State Senator David J. Regner, Chairman of the Illinois Legislative Information Systems Committee. Also presented were in-depth discussions on Licensing, Certification, EDP Career Development, and Virtual Systems.

If you attended ACPA—V, then you know and can appreciate what an excellent conference it was. If you did not attend, then these proceedings will give you a review of the conference and perhaps the finest collection of technical, social, and professional ideas which have been presented in some time.

I want to express my thanks to the ACPA—V Conference Committee Ron Stewart, Grant Berning, and Jim Brown, for helping to make ACPA—V the most successful conference that ACPA has sponsored thus far. A special thank-you goes to Marianne Fujara and her staff from the Word Processing Center of The First National Bank of Chicago for the original typing and editing of the manuscripts.

I sincerely hope these proceedings will become a permanent part of your library and that the information presented will be useful to you in some way. My best wishes to you and yours for a happy and safe 1976.

Yours truly,

Martin A. Morris, Jr., CDP
ACPA—V Conference Chairman
Chairman—ACPA Board of Directors

whether he wants to stay with his present major software suppliers, Weizer said.

If the user plans to stay with the same mainframe manufacturer, "it becomes very important to follow his lead," Weizer advised.

"All of the major mainframe manufacturers are showing their primary direction for fourth-generation software by their pattern of software announcements and enhancements," he noted.

"It is this 'primary direction' software which will be able to get the most out of the next generation of hardware," he said.

"Remain current on your operating system or start plans now for conversion to an operating system that will be supported in the fourth generation," Weizer urged attendees.

"You should never get more than one or two releases behind the manufacturers' current operating system release level. Many of the changes being put into current operating systems are in preparation for the conversion to the fourth generation," he said.

"If you remain current, you may have several minor conversion efforts which will be more of an annoyance rather than a problem. However, if you remain static for three or four years, you may find the conversion to the new generation will be very painful."

Users who don't want to stay with their present software vendor should try not to get themselves too locked into the vendor's software, Weizer added.

They should also "stay away from unique, extensive modifications o the operating system" and should not get tied into unique Cobol or Fortran features that only their current manufacturer's equipment supports.

For these users, the purchase of outside software packages which run on several manufacturers' equipment might prove to be economical in the long run, even if slightly more expensive in the short run, he noted.

Weizer offered some thoughts on what form he felt fourth-generation computer systems would take.

SYSGEN will probably vanish, and "current operating system functions may be spread across several hardware boxes," he said.

Users will be able to spread their processing load among several systems "as terminals and remote systems become more intelligent and as users and management demand more convenience and reliability," he forecast.

Cobol will remain the primary language for the next generation and, in fact, "the entire internal architectures of processors (hardware and firmware) are being optimized for high level language use."

"Vastly improved" data base and data communications systems "will offer greater ease of use, more device independence, the ability to distribute and access one or more data bases on several geographically separated systems and greater data security and privacy."

There will also be greater use of independently produced software packages, with more turnkey, full-application packages for smaller systems, he said.

A push toward greater convenience in DP use will emphasize "distribution of processing power and emphasis on on-line processing."

Fourth-generation systems will also aim at reducing the time and resources taken by application development and maintenance, he said.

Whether you're a programmer or a manager, this is a message that cannot be ignored, no matter hard it is to get people to listen. You may not perceive the change brought about by your persistence, but there's no substitute for continually pushing for something you know is important...which is quite different from pushing for a new machine because it's good. The amount of purchasing you do, of course, depends on the openness of the ear at the top, whether we're talking about the top of the department or the top of the organization itself; sooner or later, the falling drops of water begin to wear away the stone. Keep that in mind as you try to explain something to a stone-headed boss.

Incidentally, if this all sounds very important, be aware that you must also be brave. After all, people do make mistakes even such as the simple one I referred to earlier, when it was discovered that the acquisition of a new machine wasn't really justifiable. And to cover this contingency, I have a quotation from H. L. Wayland that I carry around with me...you might consider jotting it down for use when needed: "I saw a man last week who had not made a mistake for four thousand years. He was a mummy in the Egyptian Department of the British Museum." And when you run up against the problem of changing goals, or on the other hand the reluctance to change when something truly traumatic happens such as a layoff during a recession or a new program for new taxes or a new log-on procedure or new record-keeping procedures becaus of privacy legislation, you might remember that Publius Syrus stated, "It is a bad plan that admits of no modification." I'm beginning to wonder if my basic theme of little progress in EDP management is because there's essentially nothing new in technology. Machines copy human clerical functions, people get criticized for speaking or acting like robots. Where are we going?

I realize that I've touched a lot of bases here today, without going into a great bit of detail in any one particular area. But I do hope that you've seen a central theme around the need for better communications, leading to an improved role for data processors in today's computer-using organizations. I've tried to serve somewhat as a conduit of information from people who are closer to the scene than I am. I have drawn from many different sources from many different ages, but none applies better...none is more current than this: "The modern age has a false sense of superiority because of the great mass of data at its disposal. But the valid criterion of distinction is rather the extent to which man knows how to perform, and must master the material as his command." Because you get all positive-thinking that there is relief in sight because of the foresightedness of this information source, I think I should inform you that these words were spoken by John Wolfgang Von Goethe, in the year 1810.

IBM has compressed his message into many fewer words: Not just data, reality. But we need **people** to turn data into something meaningful. We can collect and massage all the data we want to today, but it takes communications and planning to make all these efforts more realizistic, more meaningful, and to keep computers in the proper place.

The computer room is a machine shop. Let's get back to managing people.

Data Realms and Magic Windows

Theodor H. Nelson

This talk will contain approximately one thought, which I think is the optimum number for an after-lunch address. None would be too few, though it is a number often chosen, and two thoughts would get in each other's way and become confused.

I want to talk about computer programming for business, a subject in which some of you are involved, even interested. But I will talk about it as an outsider, perhaps with some of the inspired vagueness which is permitted on these occasions.

First, however, let me explain the point of view I am starting from. I have for some time been engaged in a project which is intended, to put it modestly, to be the library and publishing system of the future.

Now, I am going to have to talk about that for a little while—what that system is about, and what it is attempting to do. I thought I would talk first about the assumptions and then about the actual design, but these easy distinctions do not work out in practice. The early assumptions are embedded in a philosophy, but that philosophy has gotten so deeply intermingled with the developing final design that there is no good starting place. Here is most of it.

We will have to stop reading from paper. The trees are running out, the distribution and disposal of documents cost too much, and we can no longer find what we need in an increasingly fragmented world of information.

Though most manufacturers steadfastly refuse to understand it, the next big computer market is at the consumer level. (Thousands of Altair computers, priced under a thousand dollars each, have been sold this year already.) But consumers will rarely want to program. The next grand market in the computer field will be variety of consumer turnkey products—first games, then simple interactive bookkeeping and retrieval programs, such as for phone numbers, checking accounts and magazine subscriptions.

Then will come the libraries. I think it is obvious that library service is the frontier of consumer applications, and that few computer people see what this entails in all its magnitude.

A few years from now many people—later "everybody"—will have a computer terminal. But not just a slow old printer: a fast, high-performance vectoring CRT, which is cheaper to make and simpler to use. Writings can then be stored in full-text digital form, and readers will have highly responsive consoles—preferably of the vector-display type, permitting the text to move smoothly at any speed under throttle control.

But merely supplying text to the reader's screen is hardly enough. Not only must your library-CRT system supply vast quantities of text at any usable rate of speed; it should also bring capabilities which are not now available in the best libraries. More about that later.

User actions must have immediate results, preferably under three seconds. This means computer capability in the terminal, as well as some local mass memory. Now, computer **mainframes**—the processor chips—will cost very little; likewise core (or RAM chips) will be cheap as dirt; but there will always be **fast expensive** memory and **slow cheap** memory. Thus for cost reasons the common notion of "core being cheap enough so we can have everything in all the time" is forever absurd. The system must come to grips with this great truth, in terms of managing large multilevel stores on small machines.

Similarly, "whole libraries" cannot be stored at the user's terminal, but must be rapidly accessible through communication lines on demand. It follows that the system will use computers in a net. These need not be big computers, as they are to be used only for storage and distribution—communications processors. Thus we come to the notion of a network of minicomputers. But the simplest and most elegant approach is to have all units on the network—user nodes and communications processors—use essentially the same program.

This is the fundamental idea of the system.

Now, when you say "library," computer people say, "Ah, yes! The Encyclopedia Britannica and **Moby Dick**." But we need to do more than rapidly deliver linear text. It is my working belief that linear format is not the best organization for anything, but is forced on us by the constraint of printing on paper. The real use of information by people, for a spectrum of uses involving everything from CAI to IR, will be best served by new non-linear forms of writing, generically called hypertext. I have discussed this elsewhere.

Theodor H. Nelson, **Computer Lib** (published by the author; available from Hugo's Book Service, Box 2622, Chicago IL 60690). See esp. pp. 85-70.

This is not clear to many people, because they are so used to linear writing—but our **use** of writing is rarely beginning-to-end linear, and neither are the thoughts that have been pressed into linear sequence.

Indeed, ordinary writing (ignoring illustrations for the moment) is only quasi-linear. While the author gives us a principally linear string, there are also little jumpouts—usually footnotes, sometimes "boxes" and headlines—that break out of linear format and permit the reader to pursue matters he guesses to be attached there. The succession of points and cross-references in a document are seldom linear. Moreover, we do not ordinarily read in linear sequence; and after we have read something, what remains in our minds is rarely linear either.

But to recognize this is just the beginning. When we acquire a high power non-linear text capability, considerable research and inspiration will be needed to find the best forms of non-linear organization for writing.

Very well; we must supply fast-access, non-linear library services. But even that is not enough; we want also to provide users with various helpful features for their own reading and note-taking. For instance, placemarkers (to work like bookmarks), allowing instant jump to things a user has previously flagged; and software allowing users to make marginal notes.

This leads to an interesting compound capability. Not only can an author create materials in complex non-linear document form, but a reader can incorporate such a non-linear document in a complex of his own comments, annotations and additions. Now this is a curiously involved outlook. Such compound linkages are not like anything seen usually in the computer world. But that doesn't mean it has to be disorderly. While the notion of infinite cross-coupling unnerves some people, it may be kept orderly. Whatever the complexity of the author's original version, and whatever the complexity of the user's linkages and even modifications, an orderly systems approach can keep it all systematic and sensible.

The fundamental way of keeping all this straight is with a concept implemented basically in the system. Namely, every document has an **author**, who owns it and is the only one who may change it. Each reader's **version**, containing links and modifications of his own creation, belongs to him—except for the parts showing through from some other original. In other words, the concept of authorship is to be scrupulously preserved, as well as the author's canonical version or versions. These provide the anchor for a number of interesting services.

I expect that the roles of writer and reader will remain as they always have been, and so will their motivations: for the authors, money and respect (sometimes fame); for readers, orientation and background information, the answering of questions, the enhancement of general knowledge, self-improvement, the pursuit of interests, entertainment and fiction.

To further this general idea, an unusual system of proprietary techniques has been developed for the storage, linking, retrieval and editing of large non-linear documentary structures. The intention is to use this retrieval subsystem as a basis for a product line, beginning with a simple text editor (slated for 1976), and working up to the full system. We have discovered certain structures and techniques we consider fundamental and highly generalizable. Although our data structures and algorithms were designed for the library network application I have described, it may be possible for this system to have applications in more "practical" environments. For instance, it is possible that a whole package may be marketed as a virtual-memory data-base add-on for regular general-purpose setups.

I have been frequently criticized for saying what this system will do without either showing it or telling how; in this situation I am somewhat like Frank Marchuk, the man who says he has a "laser computer." But I know that a few of you are interested in keeping track of the system prior to its actual appearance, and these remarks are for your benefit.

The system under implementation has several properties of note.

A single system is complete: from the minicomputer the user may access and revise files of any size, provided they are on local storage.

A plurality of units automatically becomes a network when hooked together, with each processor looking to the other processors merely like a terminal that wants access to a file.

"Linked text" may be thought of as the canonical storage type of this system; but properly understood, linked text can be anything. The system holds strings of any length (up to total hardware limit — with no access penalty for length); and tables and arrays, either thick or spares. (An element of an array can be a superlong string.) Between elements there may also be **links**; such links may be modal, and defined and flagged arbitrarily.

A file, or organized collection of data, is stored as what we may call a **hyperblock**—a virtual structure of potentially great size whose parts may nevertheless be rapidly accessed. (For optimization in nonliterary applications, there is a certain leeway in the declaration of preferred access methods and classess of path. But this will be discussed at a future time.)

The problem of "data integrity," especially under the multiple ingressions of inquiry and update, has been cited as one reason for needing big computers. This is otherwise settled in the present system. Our approach is that every file has a current hardware captain, which can be any computer in the network. Only one computer is at any time the captain of a file, and it will process no more ingressions until all vital updates have been made.

Certain postponements of update are possible. Catching up with these is **background** to the foreground features of display, retrieval and edit. The rapidity of certain astonishing events is actually sleight-of-hand. Certain "instantaneous" complex events are actually not completed at the moment they appear to be, but they are in an update pipeline, and their far-flung ramifications are actually consummated in a system of ripples propagated through the file as a background task. The system is thus slightly marred by, ahem, catchup stains.

In describing this system, I have left for last the parts I find most interesting, those which lead to the one idea I wish to present. Let me describe several unusual facilities this system is intended to have for non-linear writing.

One is the facility for **collateration**, or the multipoint linkage between two text units. This resembles (and can be used like) marginal notes or commentary between two texts, or for various other purposes.

Another is the facility for **quote-windowing**. This is a very interesting and unusual capability, around which I expect some remarkable uses to evolve.

The idea is that one piece of text can have a quotation from any other piece of text in the system. But it is not **merely a quotation**. The quoted text is actually still sitting in its own file; the user reading the quote is actually **at** that location in the quoted file, as well as still being at that location in the quoting file. In other words, a directional splice-point exists from the quoting file into the quoted.

The main purpose of this is, of course, so that the reader may, should it please him, transfer into the quoted file and stay there.

Scholars should find this useful for checking sources. A researcher looking for something may use it to go from a vaguely relevant source to a more exactly relevant one. The quote-windowing facility also provides a way that an anthologist, writer or correspondent can provide his reader with instantaneous entry to a number of things he thinks the reader ought to see. And, finally, it suggests an interesting solution to the copyright problem: since the quoted material is never put into another file, it can be a simple matter to count and pass on royalties everytime that file is entered by a user.

While this windowing function is most easily described for text, it is intended to apply to graphics as well. Thus one "picture" from the library may contain a "window" into another picture from the library, ad infinitum. These excerpted pictures may even be animated.

Very well. I have described an unusual system now undergoing implementation, both to show the point of view I'm starting from, and to initialize certain ways of thinking. The application of this Windowing philosophy to business programming may turn out to be rather interesting.

Now recently, on the basis of this background, I had cause to think about The Business Problem. The stimulus was the June 30, 1975 issue of **Business Week**, which offered an "executive briefing," whatever that is, on "the office of the future," whatever that is. The Office of the Future as they described it—especially as a marketing concept advanced by a major manufacturer for mechanized typing pools—was so appalling, bleak and absurb that no thinking man could contemplate it without a shudder.

Business Week, "Putting the Office In Place," June 30, 1975, pp. 56-70.

The general view was this: the Office of the Future will be sharply divided between peons and executives. The peons will be typing clerks who have been taught to work such "word processing" gizmos IBM's mag tape and mag card typing machines. The peons will be herded into central typing pools, where they will finger in what the executives dictate. Only a few lucky secretaries, the real smarties, will escape this fate. Lucky them, they will get to file the paper that the Executives dictate and the Peons finger.

Now, this is so wrong and awful it needs no direct reply. Let me speak instead about how things **should** be. Let's talk about how computers **should** work, and call it ten years from now to keep your minds off implementation problems.

Look, isn't it obvious? There should be a general system of some kind, and everybody should have a display terminal. Your job determines what you may or may not see or change.

For each job there are clear and simple interactive programs—let us call them "workfaces"—for all the things to be done. One person may have access to dozens of workfaces, and the choice, perhaps, of which workface he prefers for a given problem. The workfaces are clear and simple. Some are more general than others, like the workface that helps you type in a letter or a memo.

There is no paper anywhere, except for letters which arrive from elsewhere or those which finally leave the office. All putters-in of information have clear and simple interactive programs to help them. Moreover, all info-dippers, too, have simple and clear interactive programs to help them. But this distinction, long with us, will erode: probably every job definition in the future will include input, viewing and revision.

Good interactive programs for everybody will enable and encourage us to use our minds better. It is my belief that people will turn out smarter, happier and more productive in a system where they understand what they are doing, and are encouraged to use their minds.

So much for the preamble. I will now endeavor to express the one thought with which this talk is concerned.

Let us suppose that I am generally correct about what the office of the future should be like. Let us further suppose that a data system such as the one I described earlier—holding vast quantities of stuff on a network of minis with big secondary storage—is feasible. How then can we put this righteous Office of the Future up on it?

Earlier I made a couple of major points about such a future.

First. Central to this view was the notion of future life being organized around big files of text and graphics, stored in distributed form on a network of minicomputers. According to this idea, information may be thought of as making up big "documents" of one sort or another, especially text but also graphics.

Interactive business systems, too, can be defined in terms of "documents." If everyone in the business is to have a screen terminal, written clarifications will be needed for most types of data. Let us consider, then, a hypothetical business system that consists of a long written report on the state of the business—with windows that can show the changing summary figures, status information on outgoing and incoming orders, sales reports and prospects; and so on, for every facet of the business. (We will overlook here the problems of security and restricted access, but these are naturally a part of the problem to cope with eventually.) Here, too, then, we have a virtual space which is also a virtual document, that is, a series of presentations with windows into the data.

Now that document, some large part of which can be text—a sort of standing, up-to-the-minute corporate report—has "distances" in it. Relative "distance" in the data structure is the key concept. For instance, the two elements at the ends of a long text string are far apart, and will not ordinarily be held in core at the same time. A customer's name and i.d. number, however, are as close as two things can get; his address and phone number, are presumably at one removed (requiring the following of a pointer), and the salesman's comments on his purchasing prospects somewhat further away (requiring both pointer and lock-words).

Now, it seems to me that things are still too complicated in this field. Data base programming should be no big deal and should be virtually taken care of by the operating system; defining the data types and their access chainings should be about equivalent in complexity to a Sysgen. Now, as I envision it, the data base system would have certain given characterisitics any programmer can learn in a hurry.

Because core is small, the programmer declares what forms of information will need to be considered together. He declares access paths between types of information, such as what kind of thing points to what and what has to be found independently (such as alphabetically or by number). Things are either in core together or not, directly accessible or farther away, and so on.

Second. You will recall that I also said the key to such a data network was the use of staged update. While the system always has instantaneous integrity, it is not always **caught up**. That is, various temporary mechanism take note of whatever changes the user invokes on the data, whereupon the system replies, "It is done—" but the actual meaning is, it is **as good as** done. Finalization of the changes in canonical stored form occurs gradually thereafter. But meanwhile, all inquiries see only the current virtual structure, as ingeniously declared in various exceptions and memoranda in the system.

I submit that the way to make this distributed-mini office of the future work is a simple development of the same idea. The data base is to be a staged system. Rather than consummate all the consequences of each input keystroke—an obviously absurd extreme alternative to the batch approach—inputs are accumulated in partial forms. These partial-products are assimilated to the data base and consummated either in the course of events or as demanded.

But demand is in the form of user requests to look at something. As in the previous system, **wherever he looks**, it appears to be finished.

As the user gets **near** something, the appropriate crunches begin. The procedures of final data consummation are done in catchup mode, clued by the user's present location. This is rather like burglars peering at a night watchman on his rounds.

In other words, the programmer, in the course of preparing an interactive system or "workface," ascertains what partially-processed data must be further refined for a given user activity; these catchup actions are then cued as the user approaches places where these results will be needed.

Now, this is rather interesting from a computer-science point of view. **Places** invoke procedures. A user's arrival at specific places in the user space means actions on the data.

Obviously it would be senseless for each individual routine to test where the user is; every routine would have to be called in all the time to see if it were needed. Obviously, instead the general monitor tests where the user is, and invokes the routines called for by the user's postion. But that means there has to be a table of what routines occur at what places.

This means in turn that a table of program cues must be compiled from the total collection of all catchup procedures, so that those affected are appropriately notified by the user's position. This calls for a sort of backwards compiler—let's call it a Relipmoc—that assembles such a list ✓ and associates or inserts it in the appropriate places.

Now obviously there is a problem. It will readily be seen that certain contentions will occur. Certain "places" in the system, as it evolves, may come to trigger too many separate actions.

There are two major lines of solution. Programming staff may have some political procedure of arbitration that decides among the candidate procedures, such as the arbitrary decision of the lead programmer.

Or steps may be taken to control the user's movement, slowing him down when there are too many things to be done. If the catchup time for a given function is longer than it will take the user to get there, various delaying techniques may be employed which keep the user occupied until he's ready. Indeed, usefully occupied. If need be, we can delay the user with other things—entertainments or reminders of useful sorts—till what he wants is ready. ("Vamp till ready,"they called it in vaudeville.) Or we may modify the "documentary" setting, with new writings—preambles, updates, and reminders that slow him down.

(This is an example of one of the most interesting options of the man-machine system: the option of changing the user's experience when the program can't be done as originally intended.)

Third. I have spoken of saving various keystrokes in preprocessed and summary form, as "partial products." In the scheme envisioned here, the keystroke capture programs will update the partial prducts, and the display will invoke their final conversion and assimilation.

Thus the major decisions to be made at the systems level, curiously enough, are those concerned with the form and content of such partial products. They will of course include sums, transactions (such as the latest news about whether a specific invoice gibed with an incoming delivery), and updates to individual records. I will admit that the weakest part of the scheme lies here: in the practicality of the half-deferred update for general cases. I simply pass the idea on to you, who understand more about business programs.

So much for my one thought and its possible ramifications for business programming. Through these proposed mechanisms we get our keenly responsive, people-oriented office of the future. I assume here, of course, the availability of a high-performance retrieval system roughly like that already discussed, whose principal foreground task is interactive display. I also assume that people's time is more valuable than computer time, and the computer must do its "work" at times most convenient for its users.

Obviously the programmer's task has changed. Rather than living the batch life, or even the usual database life, in a way he now scuttles in shadows. His function is to implement the desired user experiences, not to dictate.

But there is still work for programmers. This consists of declaring the database (roughly a one-shot deal); and, to service the users, the creation of the workfaces. This in turn requires programming to and from the partial products by whatever transformation techniques seem most appropriate.

In all this the user's experience is central. When the techniques must be traded off against user experience, it is the experience that counts, not programming convenience.

It is this curious tradeoff that has made me coin the term "fantics" for the art and technology of showing things to people. I believe this is the correct generic term under which interactive programming falls. I have been criticized for making the term too broad. Yet this is all an indivisible whole, and if we narrow the discussion to a smaller part, such as computer graphics, we overlook both the overall effects of what we do and the options we might have tried. **"Fantics" is a single tradeoff domain.** We can present the same things on paper, on screen or by ear (or using the techniques of Madison Avenue, as a joke, in a song or a playlet); we can have the user point with a light pen, type on a keyboard, tap his feet or whistle.

These options are none of them right or wrong. It is how we tie them together that is good or bad, clear or confusing, dreary or fun.

Programmers are very goal-directed, sometimes too narrowly. There are different kinds of goal-directedness. A truck driver, or tank corps commander, is given an objective and perhaps a few limitations on what he may do to get there. I submit that where users' convenience is concerned, that is not the right kind of goal-directedness.

Consider the goal-directedness of a stage director: his objective is not merely to move the actors in and out and have them speak the correct lines; that is just the beginning. His objective is to create a gratifying overall experience for the people out front, merging the talents and facilities that he has available. This kind of goal has not single event, with Boolean success or failure as its culmination, like driving a truck to Detroit or computing pi to a million places. The goal is a series of events, each of which can be carried off well or poorly, each of which can gratify or annoy, and which finally result in somebody else feeling bad or good.

That feeling may be the only purpose, as in game programs and computer-controlled artistic events. More often the user's feeling is a byproduct, as in a hospital sign-in system; but that makes his feeling good no less of a goal. Few organizations today are unconcerned with the morale of their members and their clients. If you insult a user, confuse him, make him feel inferior or bore him, good will has been lost that will undercut all the other goals of the system.

The programmer may look at his job narrowly, and pretend he is only driving a truck to Detroit. He may even convince himself (and his supervisor) that making a mare's-nest of code run "correctly" is all there is to be done. But it isn't so.

The interactive programmer should understand that the mere computer is not the real focus of his work. He is working in **idea-space**. The feeling and the idea of the system to the user emerge from all the parts acting together. The user's idea of what he is doing, and what he is doing it to; the constructs and the spaces he works with and in; the clarity of it all; these are generated in non-simple ways by the program. The computer is a paintbrush for a mural of experiences you are trying to give the user.

Through the Magic Windows we give him, we want the user to see his Data Realms in bright sunshine.

How to Write Correct Programs and Know It

Harlan D. Mills

(This paper was previously presented at the 1975 International Conference on Reliable Software and published in the Conference Proceedings. Permission has been granted by the conference for this paper to appear in these Proceedings.)

Keywords and Phrases. Structured programming, program correctness, programming practices.

Abstract. There is no foolproof way to ever know that you have found the last error in a program. So the best way to acquire confidence that a program has no errors is never to find the first one, no matter how much it is tested and used. It is an old muth that programming must be an error-prone, cut-and-try process of frustration and anxiety. The new reality is that you can learn to consistently write programs which are error free in their debugging and subsequent use. This new reality is founded in the ideas of structured programming and program correctness, which not only provide a systematic approach to programming but also motivate a high degree of concentration and precision in the coding subprocess.

Introduction

An Old Myth and a New Reality. It is an old myth that programming must be an error prone, cut-and-try process of frustration and anxiety. The new reality is that you can learn to consistently write programs which are correct **ap initio**, and prove to be error free in their debugging and subsequent use.

By practicing principles of structured programming and its mathematics you should be able to write correct programs and convince yourself and others that they are correct. Your programs should ordinarily compile and execute properly the first time you try them, and from then on. If you are a professional programmer errors in either syntax or logic should be extremely rare, because you can avoid them by positive actions on your part. Programs do not acquire bugs as people do germs - just by hanging around other buggy programs. They acquire bugs only from their authors.

There is a simple reason that you should expect your own programs to be completely free of errors from the very start, for your own peace of mind. It is that you will never be able to prove that such a program has no errors in it in a foolproof way. This is not because programs are so complex that it isn't worth the effort; it is because there simply is no human way - logical or mathematical - to prove it, no matter how much effort you might put into it.

The ultimate faith you can have in a program is in the thought process that created it. With every error you find in testing and use, that faith is undermined. Even if you have found the last error left in your program, you cannot prove it is the last, and you cannot know it is the last. So your real opportunity to know you have written a correct program is to never find the first error in it, no matter how much it is inspected, tested, and used.

Now the new reality is that professional programmers, with professional care, can learn to consistently write programs which are error-free from their inception - programs of twenty, fifty, two-hundred, five-hundred lines, and up Just knowing that it si possible is half the battle. Learning how to write such programs is the other half. And gaining experience in writing such programs, small ones at first, then larger ones, provides a new psycholigical basis for sustained concentration in programming, that is difficult to imagine without direct personal experience. Professional programmers today are producing code at the rate of one error per year in their finished work; that performance is not possible by cut-and-try programming. The professional programmer of tomorrow will remember, more or less vividly, every error in his career.

What is a Correct Program? Cut-and-try programming faces three kinds of difficulties. 1. Specification cahnges. 2. Programming errors. 3. Processor discrepancies. A correct program defines a procedure for a stated processor to satisfy a stated spcification. If you don't know what a program is supposed to do, or don't know how the processor is supposed to work, you can't write a correct program. So we presume a known specification and a known processor throughout. Even so, a practicing programmer must be preapred to deal with incomplete and changing specifications, and with processors which behave differently than their manuals say. For those difficulties we have no systematic remedy, except for radical reducations of programming errors which can help isolate difficulties in these other areas. Nevertheless, the usual experience in programming often fails to separate these three sources of difficulty, so that programming errors - lumped in with everything else - seem much more inevitable than they really are.

Writing correct programs does not mean you can write programs once and for all. It means you can write programs to do exactly what you intend them to do. But as intentions change, program changes are required as well. The same opportunities and peinciples apply to these program changes. You should be able to modify programs correctly, if they are well designed and explained, as well as write them correclly to begin with.

This distinction between correctness and capability is critical in understanding this new reality. Determining what a program should do is usually a much deeper problem than writing a program to do a predetermined process. It is the latter task that you can do correctly. For example, you might wish to program a world champion chess player - that is a matter of capability, and a problem you may or may not be able to solve. Or you could wish to program a chess player whose move has been determined for every situation that can arise. You can write such a program correctly, but whether or not it becomes a world champion is another matter.

The Difficulty with Correctness Proofs. W e begin with a fundamental difficulty, which may seem fatal to our ob-