Reliability rites of the 50's and software users of the 80's

Dorothy A. Walsh

The gap between DP hardware and software is taking on the dimensions of a canyon as hardware becomes cheaper, faster, and more reliable, and software becomes more expensive, cumbersome, and error-ridden. One of the reasons for the difference lies in the retention, in software design and development, of concepts and methodologies rooted in a dead past. So-called reliability tools are among the worst offenders in this regard.

Despite many pseudo-scientific approaches to parts of the problem of unreliability, there has never been a disciplined attack integrating prevention and cure of that chronic illness of software. Most recently, preventive measures such as top-down design, structured programming, and code inspections have been emphasized. They are generally offered, however, in a kind of “we know you won’t use preventive practices” tone, along with “cures” that are worse than the disease. Such cures take the form of debugging techniques aimed at giving the programmer a look at what is happening inside the computer in a last-ditch effort to understand what his program is doing. They make odd bedfellows indeed with development techniques supposedly geared to providing a clean, functional view of well-thought-out logic. These debugging aids have respectable origins in the dark ages of programming but do not have sufficient value, even as curiosities, to be immortalized.

As it was in the beginning, . . .

Some of you may remember the dark ages of programming. One wrote in machine code and usually punched his or her (in those days it was usually her) own cards. There was no assembler (some people still regard assemblers as advanced language processors) to generate the correct operation codes and to create address fields. Therefore, it became important to have a means of seeing what actually got into the machine. So ocal dumps were born. With the years, dumps have grown sophisticated. They no longer initiate from within a program, having eaten a piece of memory in order to get started. They give alphanumeric, registers, and the usual hex or octal “image.” Nonetheless, dumps retain their original orientation, that is, a witch-hunting technique. When software was doing strange and incomprehensible things inside the machine where no one could get at it, and heaven alone knew what was actually in there, the only recourse was to look into the machine representations. The basic idea was that there must be some discrepancy between what was supposed to be there and what was really there. This discrepancy was attributed to some hostile agency—the hardware, for instance—not, of course, a logical error. Logical errors can be discovered by analysis and checking. Possible slips of the binary punch, on the other hand, cannot. In the days when one’s own assembler, this was an understandable approach to verifying and to checking for human error. (The author vividly remembers punching 36-bit instructions, two to a row, plus check-sum, word count, and address information, with a binary hand punch to “correct” 704 programs.) However, in the modern age of “systematic software” and software engineering, the holdover of such practices is difficult to understand.

The trace is a second time-honored tool of necromancy for determining what is going on as a piece of software churns out unwanted results, or worse, none at all. Like the dump, the trace originated in a historical context of software development that has long since disappeared. Early machines had relatively few words of storage available for programs and data. File supports of the times—cards and tapes—had their disadvantages in handling, speed, etc. Therefore, one attempted to write the tightest code possible to allow maximum space for data retention. (Once was usually enough to make a programmer abandon the attempt to reduce everything to no more than 500 words.) Writing tight code involved putting unconditional jumps here and there, setting switches that were start-stop (branch-no operation), and the like. As a result, a dump was often of little use because it was a static representation of an essentially dynamic entity. So the trace came into being. As a concept, as a dynamic instrument of program path observation, that is, it has its uses. Traces can be used to count iterations through paths in carefully planned testing situations, for example. On the other hand, as commonly used, the trace remains inextricably bound by its origins; it is an attempt to follow the process by which the spaghetti got knotted up the way it is. And it is just as useful for unknotting as watching spaghetti cook in a glass pot that is too small for the quantity it contains.

Is now, . . .

Thus, even today, as software engineering is emerging as a systematic
concept, the practice of looking at software after the fact to find out what it is doing is sustained and encouraged. The enormity of having to look at a hexadecimal encryption turned out by a compiler and the I/O software to determine the action of a supposedly well-designed process for carrying out a controlled set of functions on well-known data seems to escape the computer professional. Worse, there are some who are proud of the ability to read dumps, to figure out the values of actual addresses based on the contents of the registers, etc. They pity the poor applications programmer who knows only Cobol, say, and who has to rely on his wit, his understanding of the problem, and the logic of his solution to it.

Computer manufacturers take a middle-of-the-road stance. On the one side, they have their systems programmers, a strange breed who think in bits, bytes, and hexadecimal representations and whom it is better not to cross or the next release will be even worse than the last (though that one certainly seemed to touch rock bottom). Thus, manufacturers duly develop dumps, traces, online patches, and the other tools of dirty and distracted programming which pass for debugging aids (or reliability tools!).

---

**We still look at software after the fact to find out what it's doing.**

---

On the other side of the road, there are the customers. If all potential customers are to reap all the potential benefits of information processing, there has got to be some way around the need for professional programmers. And there is. (No, not unprofessional programmers—there have always been quite enough of these.) High-level programming languages permit the user to express his application in problem-oriented, not machine-oriented terms. To make sure the user's experience is a satisfactory one, manufacturers have begun to pour out the distilled wisdom of their staffs. Thus, top-down design and development concepts are formulated and disseminated. The techniques of structured programming are explained and phrases for expressing them are incorporated into existing languages or new languages. Where structured phrases can be built up from combinations of existing statements, the forms for doing so are illustrated. Preprocessors are prepared for the more recalcitrant programming languages such as Fortran.

With this strong dose of preventive medicine, a general therapy for the chronic complaint of software malfunction is also provided. Management techniques to ensure that reviews are carried out, documentation is included in the development task, and testing is exhaustively performed are promulgated. Test data generators are provided to permit "volume testing." Finally, rather than overlook any possibility, manufacturers furnish the standard trappings of sorcery so that the applications programmer who is an adept can look into the entrails of the machine and try to read the signs he finds there when all else fails.

**Is likely to be, . . .**

It doesn't appear that any great breakthrough in software reliability is at hand. The January issue of Datamation devoted a number of articles to software development techniques, quality control, and reliability practices. (It is interesting how many such articles have appeared in Datamation during the twenty-odd years of its lifetime.) These articles rehashed the same advice about clean design and disciplined, exhaustive testing. Yes, dear reader, there are, in the Year of the Lord 1979, people who innocently believe that if you just test it long enough you can make bad software good. Such people prodigiously offer advice about how and what to test in repairing the faulty product. A recent survey of management techniques gave a very poor showing to the use of top-down design, structured programming, etc. Thus, even if the preventive measures might have worked, they clearly aren't being given the chance.

At the same time, new systems are being announced as ready to be put into the hands of the end user—a layman (i.e., incapable of understanding the finer points of the DP arts). The manager choosing equipment is advised not to worry about high costs of programming personnel. The mass of end users out there (mostly pictured as women so you know they aren't overly bright) need to know only how to do whatever they are doing anyway. The software will take care of all the rest—getting information into and out of the system, organizing it, etc. The software at the terminal which is the user's means of communication with the system is more intelligent than she is. However, just to cover all bets, the manufacturer provides interactive languages with debug and on-the-spot patch facilities so that the somewhat brighter "man of average intelligence" (and we all know how low the average is these days) can dirty up his data and programs.

Proponents of this approach do not seem to perceive its ambivalence. The proclamation of the coming of age of software development as a disciplined activity leading to the preparation of a quality-controlled product contrasts sharply with the creation of the standard means of doing a dirty, undocumented job. Fanfare at the advent of the use of computers by people who don't know, and don't want to know, the inner workings of hardware and software is hard to reconcile with the development of means of looking into those same hardwares to see what the software is doing. It would appear that even the most experienced sector of the professional computer world is still confused and that the mentality which governed the creation of dumps and traces will continue to influence reliability instruments for the future. At the same time, the mind-set which regards everyone else as likely to err if not guided and disciplined will perpetuate the preaching of systematic methodologies and the practice of hit-or-miss techniques.

**But need not be.**

What makes this dim future of software development practices almost tragic is the fact that it is all so unnecessary. Software professionals are capable of producing error-free code, at least in quantities small enough to be controllable logically—i.e., retained by the mind and analyzed. (Certainly E. Dijkstra has pointed this out often enough.) By now, the standard building blocks of DP systems are known: wage and tax calculations, updates, control breaks, etc. Current data base systems allow a data-handling flexibility that could be strengthened to ensure smooth interfaces between standard modules, seen as elements in a data base. Data base management systems offer query and selection facilities that could be used to create directories of possible building blocks by which the end user—in this case really just a "layman" who knows his own business well—could select what he needs to create the system that fills his requirements. For payroll, for example, he might select from social security calculations, personal pension plans,
Louisiana is the spot for marine electronics

"Servicing the rapidly expanding marine electronics market from a new manufacturing plant in the State of Louisiana is an eminently sound business proposition."

The Fantus Company, a subsidiary of Dun & Bradstreet Companies, Inc., the nation's leading consultant to major corporations. From "Industry Location Opportunities for Marine Electronics."

The cost associated with operating a new marine electronics plant in Louisiana was compared by Fantus with operating an identical facility at representative locations in New York, California, and Florida. Based upon a facility generating $20 million in annual sales, geographically variable operating cost savings achievable in Louisiana range to a high of over $2 million annually.

The report is not theoretical. It is based on sound location strategies that manufacturers are adopting to cope with changing geographic and competitive conditions in the marine electronics industry.

In Louisiana, we offer such right-to-profit legislation as low-cost financing, a program of exemptions from local property taxes on new and expanded manufacturing plants and equipment for 10 years, and a strong right-to-work law.

The State of Louisiana would like to send you the Fantus report free.

Take a look at Louisiana, the spot for marine electronics.

For a free copy of the Fantus study, complete and return coupon to:
Andrew F. Flores, C.I.D.
Executive Director, Louisiana Office of Commerce & Industry
P.O. Box 44185
Capitol Station, Dept. B3
Baton Rouge, La. 70804
504/342-5361

Look, Ma, no programmers!

At least, none in the usual sense of the term. An approach such as that described above could lead to a kind of one-man-band in the computer depart-
ment—the data base administrator and an assistant to explain to applications personnel how to choose from the menu of offerings. If reduction of the programming task doesn’t come about in this way, it may come about in another. The evolution of programming languages suggested above as a way to build reliability into a product and to eliminate the need for bug-hunting tools is offered as food for thought about how things might be. It is clear that software reliability techniques cannot continue to develop in the direction taken back when it all began. These techniques determine the ultimate attitude toward software development. There are other evolutionary processes going on that might spell the end of such techniques. Consider microcomputers, for example.

Every day, it seems, some “breakthrough” occurs in microcomputer technology: bigger, cheaper, faster memory; new interfaces for mass storage or communications; new languages to make access easier—the list is endless. The OEM market for components of micros has replaced the West as frontierland for entrepreneurial types. All they have to do is put themselves to creating a sufficiently varied set of dedicated micro “turn-key” systems, and a new trend in computer use could be born. The computer center of the future could be a group of micros, each doing its own thing—personnel, payroll, inventory, billing, etc.—connected to a multi-megabyte general-purpose computer (have you noticed how cheap the latter have become in response to recent announcements of a well-known DP giant?) to act as a back-end data processor and coordinator. The sight could not be any less esthetic than that of a group of bearded hacks peering into the displays of hexadecimal values, trying to correct some procedure immersed in a mire of operating system software that has suddenly gone haywire. And, in the long run, this set of micros would probably cost less. They eat little, don’t have families to support, and don’t belong to unions.

Conclusions

Information processing technology presents a mixed bag of space-age hardware and stone-age software development techniques and practices. Among the latter are a set of error-prone divination systems for looking into machine representation rather than procedural logic. These systems are called reliability tools. (And there are those who say that computer professionals have no sense of humor!) Attempts to perpetuate such tools long beyond the historical context in which they originated has resulted in a lack of defined techniques geared to today’s users and environments. It is unlikely that potential software users of the 80’s will relate to the frame of mind that produced the present attitude toward software development techniques for reliability control. The means of building reliability into software systems should be rethought in the light of present and future technology, user psychology, and applications areas.


Dorothy A. Walsh is an independent consultant working from Rome, where she also teaches computer science at John Cabot International College. She has worked as a DP consultant for over 24 years, starting as a programmer and rising to the vice-presidency of a consulting firm, Advanced Computer Techniques Incorporated. Her experience includes software project management, compiler design, software quality control, computer center management, and consulting in network design. Walsh received the MS in mathematics from New York University.

November 1979
**UNIX**—from one software marketeer’s view

Jim Isaak

UNIX has many strong points that make it not only very useful but also an excellent model for more recent systems. However, I have had the privilege of going through the birth pangs of one of these recent systems—MP/OS at Data General—and can add some additional insight into the UNIX phenomenon.

First, one of UNIX’s most attractive aspects is its popularity and wide deployment throughout the Bell system. When one of the largest companies in the world likes something you take notice. Moreover, it is a good system. As a result, any general-purpose system design done in the last five years has used UNIX as a source of ideas, or at least used the same background theories. Few, however, are UNIX per se.

Many of UNIX’s benefits involve independence from a specific vendor—but do we know the real price tag for such independence?

But is there too much emphasis on UNIX as a possible standard when it should just be a model? Viewing UNIX as a standard is dangerous for the end-user—a company evaluating what system to purchase can get caught in the trap of specifying their requirements based on theoretical standards rather than on their actual application needs. This can severely limit the range of possible choices and even leave them with a selection that meets the spec but doesn’t do the job. (In government bidding this can be a real trap.)

In addition, UNIX as standard would exclude consideration of more recent systems that have used UNIX as a launching point. Let me cite two examples of such systems showing some of their advantages...

• Data General’s Advanced Operating System, or AOS, is built on concepts like those of UNIX and Multics. AOS requires large-scale system security techniques and the ability to offload character peripherals to a separate processor and still operate within the realm of minicomputers. It also needs both real-time “resident” processes and batch streams to cover a wide range of application requirements. However, AOS is a system built with UNIX in mind, it has a wider market and takes a more recent view of that market.

• Data General’s MP/OS is built for the other end of the spectrum. A system with development facilities and sophisticated disk file management, it is nonetheless modular enough to be configured without a disk or even burned in PROM for operations on small, dedicated microsystems. There are two points here—MP/OS’s modularity allows a wide range of configurations and compatible growth into the Advanced Operating System; MP/OS applies concepts of software engineering and reliability that were not even considered five years ago.

Between AOS and MP/OS there is enough additional functionality, configuration flexibility, reliability, performance, and vendor support that I would be hesitant to give these up for a “standard system” based on the current implementation of UNIX. The industry still doesn’t have enough maturity in system design to truly strike out for standards, except in some rather limited areas. (I feel personal computing would benefit from some standards, for example—see my article, “Standards for the Personal Computing Network,” *Computer*, Oct. 1978, pp. 60-63.)

Many of the benefits of UNIX mentioned in an earlier Open Channel article (“UNIX—a software marketing phenomenon,” June 1979, pp. 79-80) involve independence from a specific vendor and a corresponding relief of the vendor’s software support responsibilities. But few non-educational institutions can afford their own group of systems programming experts, which creates a need for a third-party software support/service group. Here is the final rub—a third party would need to charge a lot—both for each new system and for ongoing support services. A lot because software isn’t cheap, and because the third party has to match the development and service facilities of a computer vendor. End result—the vendor’s unique product will tend to cost less. A third party’s top-notch operating system with support probably costs at least $500/month plus the additional charges for software packages and languages (per system). Universities have a large source of low-cost labor to offset this expense but the rest of us — vendors and users — have to make money and pay for labor on a full-time employment basis.

UNIX has had a significant effect on software marketing and systems over the last few years; I expect it will continue to be a high visibility system. However, I’m not sure the real price tag for a vendor-independent system has been established, or can actually beat a vendor’s hardware/software/service prices for a substantial portion of the commercial/scientific user community. Finally, UNIX still does not exploit certain useful concepts, due to its initial design objectives and just plain age; this limits its value in a number of significant markets.

Jim Isaak is senior marketing specialist for the Microproducts Marketing Group at Data General. His most recent responsibilities have involved technical support for computer systems sales and services. Isaak received the BS degree in computer studies and the MSEE degree in computer engineering from Stanford University.

In response to our request, Mr. Isaak has offered to prepare a more detailed comparison of AOS and UNIX for later publication.

— J. H.

More about UNIX

Several readers have written to ask where to get more information about UNIX. The latest source we have seen is:

Patent Licensing Manager
Western Electric
PO Box 25000
Greensboro, NC 27420

or call L. Isley at (919) 697-6530.

The UNIX users’ organization is currently undergoing reorganization, we are told, as it has grown to be too much work for volunteer personnel. Those interested should write the individual named below and ask to be contacted about the group’s next meeting. Rumor is that it will be in Boulder during ski season.

Prof. Mel Ferenz
Box 8
The Rockefeller University
1230 York Avenue
New York, NY 10021

— J. H.