THE ARROGANT PROGRAMMER: DIJKSTRA AND WEGNER CONSIDERED HARMFUL

by
Robert J. Du Works
Stephen W. Smoliar

TECHNION
Haifa-Israel

(Ed.: I think this article should definitely spur further comments on the Kandel-Wegner "debate" on the proper path for Computer Science Education to follow. (Comm. of ACM, June, 1972.) This debate has led to a great deal of discussion of many aspects of Computer Science Education. I think this is a sign that perhaps we are maturing as a discipline. I hope this article adds more "fuel to the fire," and that many of you will react to the comments in this paper. The more debate we have, the better are our chances to sift out what we feel are the most important aspects of Computer Science Education. Though agreement on these "basic principles" might not be forthcoming for awhile, I think articles and discussions on this topic are crucial to our development. So please take the time to write me a letter elaborating on your particular philosophy.)

The first snows are beginning to fall in the winter of our discontent. Peter Wegner's ACM election statement (ACM) stresses the current crisis of identity in the computing profession. A recent symptom of this crisis has been the rising proclivity of computer professionals for great debates on such issues as the virtues of the goto (Leavenworth) and, perhaps, the number of angels which may dance on the head of a chip. Abraham Kandel's qui bono statement in the June, 1972 CACM (Kandel) neatly summarizes the crisis -- to whose good is the current educational approach to computer science?

Kandel advocates an educational emphasis on computer engineering, as opposed to computer science. The mathematical foundations of the latter, he bemoans, have become ends in themselves; and the practical values of the computer are in danger of being totally ignored. In a rebuttal statement Wegner asserts that these practical values essentially involve "'good' programming habits" which, at the present time, we really do not know how to teach. He admits that we may be teaching "the wrong kind of theory", but that we should be seeking out the "right kind of theory" rather than decrying the formal approach.

One thing is certain: science has arisen from man's quest to formulate terse generalizations of his myriad observations. What is important is that observation precedes generalization--preferably by as great a distance as possible. As Suzanne (Langer) has written:

> General theories should be constructed by generalization from the principles of a special field, known and understood in full detail. Where no such systematic order exists to serve as a pattern, a general theory is more likely to consist of vague generalities than of valid generalizations.

Now, has the 25-year history of the ACM really given us adequate time to formulate valid generalizations?

Curiously enough, if existing theories are not adequate to explain the behavior of computers, they may possibly be employed to explain the behavior of computer professionals. Kandel and Wegner demonstrate a standard "metasituation": each admits that "some" of the other's attitude is necessary; but each, in the standard manners of academic ritual, remains sceptical that "some" may degenerate to "too much". Now if we really want to fire up the available theoretical machinery, we may regard academic computer science as three parallel processes with limited cooperation:

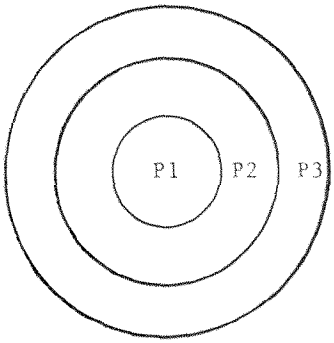
process1: formal mathematics, automata, formal languages, etc.

process2: applied mathematics, numerical analysis, applications programming

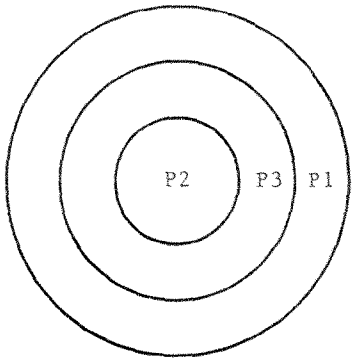
process3: software development, operating systems, compiler-writing techniques, process control, etc.

Unlike "cooperating sequential processes", however, these processes demonstrate two malignancies which we may call "greed" and "interference". "Greed" arises when in theoretical terms, one process performs so many P-operations (Dijkstra) on the "critical areas" of computer science that the other processes are "frozen out". "Interference", on the other hand arises when, in defiance of all semaphores, several processes jump into critical areas at the same time, turning the common data base into a nebulous inter-disciplinary muddle. Needless to say, the road to interference is usually paved with good intentions... but so is another famous road.

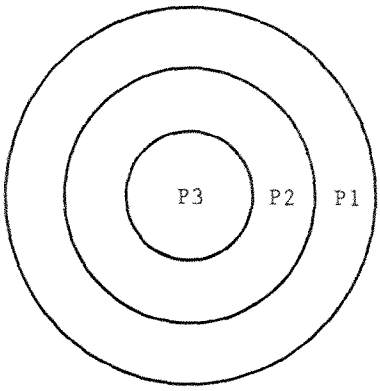
Theory is helping us to better formulate our problems. Now what about solutions? Can we, perhaps, harness our knowledge of, say, priority structures? Needless to say, if we try to be fair, we shall probably arrive at three hierarchies of "rings of power". In terms of the model of (Graham), we may represent an assignment of priorities suitable to process1 as follows:



In other words, the formal mathematicians tend to dismiss the software process as worthless "hackery". process2 would prefer the following structure:



Software is at least close to the heart of the applications programmer while formal mathematics tends to be too arcane. Finally, process3 takes the following view:



The affinity with process2 is still recognized; but the priorities are, of course, reversed.

Thus, it is unlikely that we may arrive at a "fair" priority structure. Furthermore, whether they like it or not, all three processes must interrelate; certainly, we cannot expect any of them to just go away. Thus, the trick is to attempt to cut down on "greed" and such manifestations as goto debates, which are essentially priority quibbles (in this particular case between process1 and process3). If "greed" is suitably curtailed, then "interference" may possibly be converted into interdisciplinary cooperation.

Actually, such priority considerations might well be avoided altogether if we remember that computer science does not exist in a vacuum. Formal mathematics may, after all, be relegated to "formal" mathematics departments--probably with little hard feelings. Likewise, process2 may readily rise to its desired level of priority in an applied mathematics or engineering department. This leaves software which, when all is

said and done, would probably be much happier with an apprenticeship program than with an academic department (were it not for the fact that "Professor" is a far more desirable title than "Master Programmer"). Indeed, going back to Wegner's argument, until we have more observations upon which to generalize, is there any better way than apprenticeship to teach "'good' programming habits"? God forbid, if we don't watch out, process2 may also find itself happier in the environment of apprenticeship, leaving process1 alone in academe except for those rare (at least nowadays) occasions when it has something tenable to offer.

Lest we have distorted our view of the situation, we should emphasize that good software techniques require a firm grasp of formal mathematics which should not be underrated, not only for its ideas but for its "way of thinking". Indeed, any aspect of computer science above the programming level must be able to formulate itself clearly and concisely--a task with which formal mathematics is quite familiar. Thus as Kandel has observed, mathematics can provide us with viable models. Our caveat is to make sure that the modeled object does not become dominated by the model itself.

In conclusion, we may draw on the "case study" approach (which (Kandel) seems to advocate) to observe some of the unfortunate effects of interference. ALGOL 60, for example, is a clear-cut case of interference between process1 and process3. The conflict between a desire for mathematical purity and a need for "'good' programming habits" could only yield a rather anomalous specimen of the second-generation-programming-language genus. Indeed, the very conflict from which ALGOL 60 emerged was but the first presage of the goto controversy--a debate in which only an ALGOL programmer would dare to get involved.

On the other hand, PL/I, notwithstanding IBM's implementation or the Viennese attempt at a formal description (Rosen), provides a far superior language, complete with what (Langer) would call the "generative ideas" of third-generation computers. The fact is that purity (i.e., mathematical "elegance") is not the solution to the world's problems; and "formal overkill" may even cause some good ideas to be "aborted" or "still born". Here, our caveat must extend to attempts at formal descriptions of operating systems. What we lack most of all is the demand for "system elegance", which is independent of, without necessarily excluding, mathematical "elegance". This distinction was clearly manifested at a Technion colloquium in which (Wegner) made the comment that operating systems do not halt "nicely". We would like to believe that operating systems will behave "nicely" in Wegner's "right kind of theory", although it is quite obvious that they have a hard time conforming to his current notions of information structures.

As a final example, Dijkstra's "formal" foundations of parallel-process theory constitute a clear-cut cart-before-the-horse case of observation-generalization inversion. The funny thing is that Dijkstra appears to be totally oblivious to the fact that the ENQ and DEQ system macros in (IBM) OS/360 together with the POST and WAIT macros, constitute the essential elements of the P- and V-operations. The ENQ macro is actually more closely related to the TSL (test and set lock) operation presented by (Lampson), who's own approach is thoroughly geared to the real world of third-generation computers, while Dijkstra's so-called "style" leaves him hopelessly entrenched in second-generation ALGOL.

This whole issue of style is, in fact, really too potentially dangerous to be overlooked. Dijkstra makes a predominant issue out of the "provability", or "formal verification", of programs. This is, indeed, a key issue in the goto controversy; and (Manna) and Vuillemin present but one piece of evidence that it is harder to prove properties of programs with goto statements. However, just how applicable are these formal verification techniques to third-generation programming systems? Once again, we find "the wrong kind of theory" in ascendance; and, in the words of John von Neumann (Revens), "the general situation has not yet matured sufficiently" to nurture the development of "the right kind of theory".

To recapitulate, software would probably fare best if freed from its academic shackles. Until we have a substantial corpus of material upon which to theorize, we would do better to take a "seat of the pants" approach. Barring this, we would support Kandel's argument for computer engineering. While the current theories provide us with many useful tools, for now, at least, we remain "information grease monkeys" at heart who, by our very nature, throw all sorts of "monkey wrenches" into the elegant world of mathematics, our "reluctant parent"...but such is life.

REFERENCES

(ACM) 1972 ACM Election: Candidates, Comm. ACM 15, 4 (April, 1972), 286-292.

(Dijkstra) Dijkstra, E.W., Cooperating Sequential Processes, Programming Languages, F. Genays, ed., Academic Press, New York, New York, 1968, 43-112.

(Graham) Graham, R.M., Protection in an Information Processing Utility, Comm. ACM 11, 5, (May, 1968), 365-369.

(IBM) IBM Systems Reference Library, IBM System/360 Operating System Supervisor Services and Macro Instructions, GC28-6646-6, 1972.

(Kandel) Kandel, A., Computer Science--A Vicious Circle, Comm. ACM 15, 6, (June, 1972), 470-471.

(Lampson) Lampson, B.W., A Scheduling Philosophy for Multiprocessing Systems, Comm. ACM 11, 5 (May, 1968), 347-360.

(Langer) Langer, S.K., Philosophy in a New Key, The New American Library, New York, New York, 1951.

(Leavenworth) Leavenworth, B.M., Programming With(out) the GOTO, Proceedings of the ACM Annual Conference, August, 1972, 782-786.