ACM FORUM

READERS REACT TO COLUMNISTS' COMMENTS

Arguing Against Certification

I am writing to comment on Peter G. Neumann's column, "Certifying Professionals" (Feb. 1991, p. 130). I am an academic and thus "immune from the problem," as J.H. Whitehouse is quoted to say, yet I feel strongly about the issue.

I believe that certification is a bad idea and hope that it never happens to the software industry. The hodgepodge of arguments for certification contains such contradictory statements as "Nurses, physicians, pilots, civil engineers (even hair stylists) are licensed" and on the other hand that certification is desirable "for staff who hold key positions of responsibility on projects that have significance for society." Which one is it, certification for every coder, or for the top honchos only?

The arguments for certification are unconvincing.

(1) *Other professions do it.*
Some do (lawyers, hair dressers), some don't (business managers, research chemists, politicians)

(2) *Professional errors could cost society dearly.*
Indeed they could. However, in a project of any size such errors can rarely be attributed to a single incompetent individual. Software design and programming are error-prone activities. Who should have their certification stripped off in case of a fatal flaw? The coder who wrote that function (if it was a single one)? The manager of the debugging team? The project leader?

The arguments against certification are much stronger.

(1) *Certification is monopolistic, protects mediocrity and shields incompetence.*
The best example is the legal profession.

(2) *The field of software engineering is too immature. Certification would freeze (or at least gel) today's unsatisfactory state of the art.*
There are few accepted standards for analysis, design and programming. Sure, gotos are bad, but what else? Is object-orientation good? Should it be required? How will the next advance be absorbed? A colleague of mine in the civil engineering department groans about the inability of his profession to switch to faster, more reliable computerized techniques for earthquake safety design because they do not conform to existing construction codes.

I am glad that *Communications* is covering this important issue.

Cay Horstmann
School of Science
San Jose State University
San Jose, CA 95192

You Can Always Tell a Hacker . . .

In the article "The United States vs. Craig Neidorf," (Mar. 1991, p. 24) Dorothy E. Denning suggests lessening the punishment given for accessing computer systems without authorization. Whereas this may seem like a good idea at first, how will this help to solve the problem of hacking? It simply will not. Hackers will continue to log onto systems where they should not be simply for the thrill of doing so, and for the fun of toying with others.

In fact, most small-time hackers probably do not even realize the legal implications of their meddling. For them, changing the law will not affect them because they will not even realize it has been changed. What we need is to get the message out to young hackers telling them what may happen if they continue their illegal activities. We must also make potential hackers aware of the law, since reducing the

number of potential hackers will reduce the risk to us all.

Jon Gettler
9260 S. 94th St.
Franklin, WI 53132

"Bugs" and a Balanced Tedium

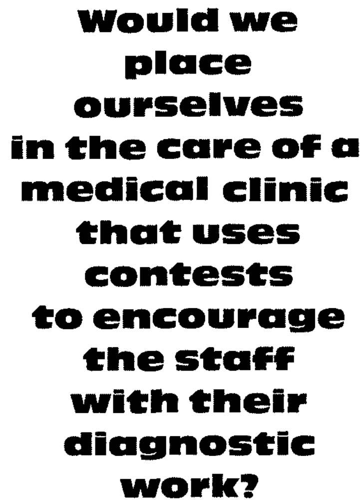
I found what was said between the lines of the Practical Programmer column, "Testing Made Palatable" (May 1991, pp. 25–29) disconcerting. In relating the experiences of his development group, Marc Rettig provides the attentive reader with insight into one of the most significant causes of the "software crisis." There is a definite lack of a professional ethic behind our view and discussion of our work within the software "profession." I find two notable examples in Rettig's column: the problem of "bug" and the response to tedium.

I look forward to the day when I can read a copy of *Communications* cover to cover and never catch sight of the word "bug" as a second-rate synonym for error or defect. For the last year, I have followed Edsger W. Dijkstra's exhortation to trade in "bug" for the more honest "error." Doing so has had the profound effect that he predicted: "While before, a program with only one bug used to be 'almost correct,' afterwards a program with an error is just 'wrong' . . . " [3]

I cannot disagree with Rettig that we must find ways to increase the effectiveness of testing as a means to assuring that we have created defect-free software. I am, however, concerned with the view (and perhaps the reality) that a significant hurdle to this goal is the tedium of software testing.

Should we strive for a "pleasant balance between rigor and tedium?" We may need to find a practical balance between rigor, schedule constraints, and reliability requirements. Does it matter whether testing is "no fun" or can be "great fun?"

How do our perceptions of our work implied by Rettig's column



compare with other professions? Would we place ourselves in the care of a medical clinic that uses contests to encourage the staff with their diagnostic work?

My purpose is not to cast aspersions upon Rettig's professionalism and that of his colleagues. Rather, I assert that the very vocabulary we use significantly colors our professionalism. Until we change our perceptions of our work and the way we describe it, there is little hope that we can become professional (as the rest of the world understands that term) as well as practical programmers. The fact that Rettig quotes David Parnas's concern about the lack of professionalism among software engineers and misses the possible application of that concern to his own words is an indication of how deep-seated the problem is.

Corey Huber
Fraser Consulting, Inc.
1 Liberty St.
Cazenovia, NY 13035

Response

Huber makes interesting points, and I appreciate his diplomacy. We agree on the problem of professional discipline in the software community. We address the problem from different points of view.

He is concerned about the notion that tedium is the enemy of rigor. In my experience, this is a fact of life. (A "problem" is something you can hope to change. A "fact of life" is something you must learn to live with.) In my experience, most programmers work best during the rewarding problem-solving and coding phases of a project. When it is time for tedious testing and mindless paperwork, their enthusiasm dwindles, and with it their effectiveness. Sermons about rigor make them feel guilty, but rarely affect work habits. This is not scandalous irresponsibility, it is just human nature.

So it really does matter whether testing is "no fun" or can be "great fun." It affects the quality of our software. People *need* to have fun, and they need the recognition and support of their peers. A person can focus only so long on a detailed, tedious task.

Then it is time for a ping-pong game, or rigorous attention to detail will degrade to mindless staring at the screen. Gerald Weinberg discusses this and many other factors in his *Psychology of Computer Programming* [7] (see especially Chapter 10, "Motivation, Training, and Experience"). And recognition of personality needs is one of the reasons that recent approaches to team development, like Structured Open Teams, are so effective.

Instead of working against human personality traits to enforce a reluctant discipline, let us recognize their role in software quality and use them to our advantage. Let us develop tools and management techniques that encourage people to develop quality software. Of course, this does not take us all the way—some things are just plain hard work, and professionals must be prepared to apply themselves.

So, to answer one of Huber's questions, the knowledge that a medical clinic uses contests to encourage their diagnostic staff would be a mark in their favor, in my opinion. I would expect them to be enthusiastic about their work. If they have a ping-pong table in the

lab, I will trust them with my very bowels.

When it comes to "defect," "error," or "bug," I guess I am happy either way. But I would like to make a point about programming in practice as a comment on Huber's concern.

Although most programmers would say their ideal is to produce "defect-free" software, the real goal of most projects is to produce something that works to the customer's satisfaction, on time and under budget. For mission-critical and most commercial software, "to the customer's satisfaction" means "defect-free." But for some large percentage of projects it means "with reasonable efficiency and without major error" ("major" being an admittedly ambiguous word).

Most projects keep a "bug list" on which defects are given a severity rating. A misspelled word in an error message is given low severity, incorrect results would be mortally severe. For the majority of projects, the large cost and delay of removing every last item from the bug list is not justified by the customer's demands.

Professional programmers will be able to produce defect-free software when it is called for. They will also be able to determine when it is not called for, properly prioritize defects, and deliver software that meets the customer's need, on schedule and within the budget.

Universities and organizations like the ACM are trying hard to foster the attitudes and habits of engineering discipline in the software community. We are all better off for their efforts. Out in the world of computing practice, Huber and I are recognizing some hard facts of life, and learning to live with them.

Marc Rettig
Summer Institute of Linguistics
7500 W. Camp Wisdom Rd.
Dallas, TX 75236

Defining Formalism

In his March editorial [1] and another column [2], Peter Denning relates problems in U.S. management styles and in software development, and concludes that these problems are attributable to excessive formality on both areas. He believes we must go "beyond formality." Unfortunately, the gist of his writings suggests not so much that we should go beyond formality, but that we should discard it.

We believe that the problem is not that methods are formal but that they have been misused.

We agree that the development of useful software requires discovering what users really need. Good communication between users and designers is essential to such discoveries and need not be formal. But formalism permits us to communicate user requirements precisely in a way subject to rigorous analysis, and we are concerned that some may interpret Denning's article as a justification for discarding one of the few effective methods we have for managing the software development process.

Many of the problems Denning identifies are real, such as the failure of software developers to focus on how software can best support effective human performance of a task. However, his attribution of these problems to excessive use of formal methods in both management and software development is dubious. The fact is that software development practice today makes little use of formalism. Indeed, although Denning defines "management," he never defines just what he means by "formal" or "formal methods." Characterizing Taylor's management techniques first as "scientific" and then as "formal," Denning next shifts the discussion to the use of formal specifications and formal methods in software development, implying that the failures (as he sees them) of formalism in one domain carry over to the other. Without a definition of "formal" it is hard to evalute this argument. Here and throughout his articles he makes "formal" into what some would call an accordian word—i.e., a word whose meaning can be expanded and contracted to suit the author's tune.

We believe that the problem is not that methods are formal (i.e., there is an effective procedure for determining whether they have been correctly applied) but that they have been misused. If a corporation creates for itself a structure that prevents the internal communications necessary for it to compete, it will fail, but one cannot conclude from this that a structureless corporation would do better. Nor can we conclude that because software specifiers have difficulty anticipating how a system may be used and what changes may be required after users gain experience with it that they should forget about specifying it and simply build (somehow) what the user wants.

Denning questions whether "what people do" is formalizable at all. It may indeed be impractical to specify some human behaviors (particularly those we do not thoroughly understand) formally, but anything we can program a computer to do must, of necessity, be formalizable. If there is no more abstract formalization, the program itself provides one. We view with concern the anthropomorphic view of computers that is all too common in popular writing.

Denning does express the legitimate concern that formalism makes

software builders unresponsive. One concern is that once we have gone through the effort of formalizing system requirements, we are unwilling to change them. However, we believe that introducing formalism *per se* into software engineering is not what leads to unresponsiveness; rather, it is once again the misuse of formalism. Methods do exist for developing software that can be easily adapted to changed requirements [6]. Such methods couple formalism with information hiding and abstract specifications (i.e., specifications that postpone premature design decisions). Applying them to software development produces specifications that can be changed without undue effort. In fact, the type of precise, abstract specifications required by these methods are best written in a formal language that eliminates the ambiguity and implementation-dependent artifacts that plague more informal specifications.

In closing, Denning encourages us to learn more about user-centered design [1], and he endorses the notion of basing software designs on linguistic analyses of work [2]. Describing an example, the Coordinator [5], he notes that its design is simple because "it is based on an interpretation in which all networks of conversations for action are composed from a base set of four recurrent conversational moves." Dobson and McDermid have taken a similar approach to discovering appropriate characterizations of system security properties [4]. Both of these examples strike us not as "beyond formalism" but as precisely the appropriate application of formalism to capture essential structure.

Carl Landwehr, John McLean, and Constance Heitmeyer
Naval Research Laboratory
Washington, D.C., 20375-5000

Response

The thrust of my argument is this: Software development is a management process and suffers from the same rigidity that organizations are all too often. At best, formal methods fail to address rigidity because they focus exclusively on the computer system; at worst, they contribute to the problem because of the large investment of resources to create or change the formal specification. Therefore, we need to supplement our formal processes with new ones that explicitly take the broader context into account, adapt to change, and promote communication among people in the organization.

Landwehr, McLean and Heitmeyer worry that my words "beyond formalism" will be construed as "instead of formalism." I do not advocate abolishing formal methods. In the domain of machines built as systems of interacting components, a formal, rule-based approach to describing the function of each component and the exact ways that they interact is essential to the design. Most technologies today could not have been built without this way of thinking about systems.

The problem arises when we assume that rule-based thinking will be effective in all domains, such as the domain of human interactions in organizations, and when we forget that designers must respond to changes in organizations as well as changes in technology. For me, moving "beyond formalism" does not mean discarding formalism in the domain of machines, but adding a new domain of awareness of the human organizations in which the machines will work.

Peter J. Denning
Computer Science
George Mason University
Fairfax, VA 22030

APL 92

I was very pleased to see mention of the forthcoming APL92 conference in the June 1991 President's Letter, nevertheless I feel that the piece does not fully reflect the relevance of this event to ACM's members, nor fully accredit the efforts which have led to this agreement.

APL92 will be the latest in the series of conferences on APL which have been running annually since 1979 and less regularly for several years earlier. It is the annual international conference of ACM's SIGAPL, which is making a very significant contribution of personal and financial energy. The APL conferences have alternated between the Western and Eastern hemispheres very successfully. Serious discussion about Leningrad as the venue for APL92 began at the APL90 conference last year, which was the first occasion on which Soviet delegates had been able to attend one of these conferences.

The origins of the APL92 initiative stretch back a very long way, at least to a visit following the APL84 conference held in Helsinki, Finland, by Ken Iverson (originator of the APL notation) and Robert Bernecky (current vicechair of ACM SIGAPL). This visit was instrumental in introducing APL to the Soviet computing scene, forging links which continue to this day between ACM SIGAPL, the Finnish APL Association and the Soviet groups in both Leningrad and Moscow. Much of the credit for arranging this pioneering visit must be given to Timo Seppala of the Finnish APL Association; among the venues at which talks were given were the Moscow Academy of Sciences and the Tallin, Estonia, Institute of Cybernetics.

SOVAPL is noteworthy also in being the first affiliation of a national group as an ACM local SIG. It is also worth mentioning that ACM's NY/SIGAPL local group sponsors two Soviet members of ACM and that APL91 in conjunction with ACM SIGAPL have arranged support of four Soviet attendees at APL91 (In Palo Alto, Calif.) through individual, corporate and local group contributions.

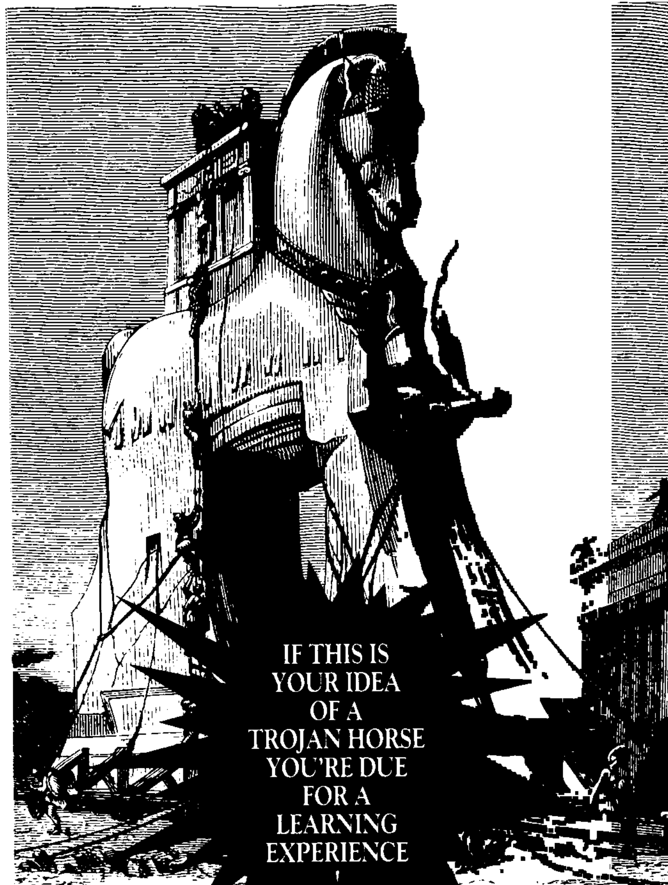
As you see, APL92 is an event which forms part of a continuum of

effort both by ACM's active SIGAPL membership and by associated groups worldwide. I feel that it is important that this perspective be given to the rest of ACM's membership.

Dick Bowman
2 Dean Gardens
London E17 3QP
England

References

1. Denning, P.J. Technology or management? *Commun. ACM 34,* 3 (March, 1991), 11–12.
2. Denning, P.J. Beyond formalism. *American Scientist 79,* 1 (Jan–Feb, 1991), 8–10.
3. Dijkstra, E.W. On the cruelty of really teaching computing science, *Commun. ACM, 32,* 12 (Dec. 1989), 1398–1404.
4. Dobson, J.E. and McDermid J. Security models and enterprise models. *Database Security, II: Status and Prospects,* C.E. Landwehr, ed., North-Holland, 1989, pp. 1–40.
5. Flores, F., Graves, M., Hartfield, B., and Winograd, T. Computer Systems and the design of organizational interaction. *ACM Trans. on Office Information Systems 6,* 2 (April 1988) 153–172.
6. Parnas, D.L. Software engineering principles. *INFOR Canadian Journal of Operations Research and Information Processing 22,* 4 (Nov. 1984) 303–316.
7. Weinberg, G. *The Psychology of Computer Programming.* Van Nostrand Reinhold, NY 1971.



Trojan horses, hackers, file corruption, worms, natural disasters – just when you think you've done everything to protect your computer resources, the attack begins.

But don't despair. You can defend your computer network from unauthorized access and disruption with the new techniques you'll learn from The Computer Security Seminar Series. The Computer Security Seminar Series is sponsored by ACM/SIGSAC, ADAPSO, American Express, *Computerworld* and Ernst & Young in preparation for Computer Security Day on December 2.

The program costs only $195 ($175 for ACM and ADAPSO members), including presentation materials, books and a luncheon. That's a small price to pay to avoid some very expensive problems. But registration is limited. So, call today for registration details.

THE COMPUTER SECURITY SEMINAR SERIES

800-524-4023
(In Maryland, 301-662-8087)

The Seminar will be presented in these 12 cities:

10/8 Phoenix	10/29 Chicago	11/7 Boston
10/21 Atlanta	10/30 Minneapolis	11/8 New York
10/22 Los Angeles	11/4 Houston	11/15 San Francisco
10/25 Detroit	11/6 Philadelphia	11/18 Washington DC