A Conversation with Barry Boehm

Recollections from 50 Years of Software Engineering

Hakan Erdogmus and Nenad Medvidović

Barry W. Boehm is the TRW Professor of Software Engineering and the director of the Center for Systems and Software Engineering at the University of Southern California. He's a pioneer of software engineering, with seminal contributions in software economics, process, requirements, and architecture. He has received the ACM Distinguished Research Award and the IEEE Harlan D. Mills Award; he's a Fellow of ACM, IEEE, AIAA, and INCOSE and a member of the National Academy of Engineering.

***IEEE Software*: This magazine is celebrating the 50th anniversary of the 1968 NATO Software Engineering Conference, an important milestone. How did you first hear about that conference?**

Barry Boehm: In 1968, I was working at the RAND Corporation, the US Air Force's think tank, as head of a computer-systems-analysis group, when a RAND colleague, Jim Babcock, returned from participating in the NATO Software Engineering Conference. He gave a talk on the conference and its assessment of the increasing role of software in NATO command-and-control (CC) operations, and the need for better ways of improving its reliability and productivity. I had been participating in some of RAND's studies on the use of computers in Air Force CC, and was able to take advantage of a trip to Europe to meet with some of the principals in the NATO conference, particularly Friedrich Bauer, the conference chair, Brian Randell, the coauthor of the conference report, and Edsger Dijkstra, whose 1968 contributions also included his famous "Go To Statement Considered Harmful" letter to the *Communications of the ACM* and his THE multiprogramming system.

What were some interesting insights from that trip?

The session with Dijkstra was a unique experience. I would ask a question—for example, on improving software reliability—and there would be about two minutes of silence, followed by a fully organized minilecture on the futility of trying to estimate software reliability by extrapolating defect frequencies, and on the need to precisely specify the software objectives and to logically prove that the resulting software satisfied the specified objectives. There were similar silences and minilectures on the shortfalls of popular programming languages, such as Fortran with its GOTOs and global variables, and with responses to other questions.

What was the fallout from the NATO conference and the follow-up to the meeting in Europe?

In 1969, I was asked to provide a briefing on the information-processing aspects of Air Force space missions as part of an Air Force Scientific Advisory Board meeting on military preparedness in space. The briefing content was published as a RAND report called *Some Information Processing Implications of Air Force Space Missions: 1970–1980.*[1] It included data on trends in information-processing aspects of Air Force and NASA space missions, and also drew on key issues from the January 1969 published report on the 1968 NATO conference.[2]

In 1970, the Air Force prepared to perform a major year-long mission analysis on the future of Air Force CC information processing, with recommendations for improved management processes and information technology research to meet the determined needs. The Air Force asked RAND to lend me to the Air Force for a year to run the mission analysis, which it did. The study team included an Air Force colonel with extensive CC experience as deputy leader, and about a dozen full-time and part-time Air Force CC and information-processing experts from the Air Force and some of its study organizations. Each of us had copies of the 1968 and 1969 NATO conference reports,[2,3] and the project had a budget for industry studies. Its title was CCIP-85, and it produced an overview report plus five detailed reports on CC information-processing hardware, software, staffing, management, and CC operational trends, opportunities, and challenges.

The most valuable inputs to the study were week-long visits to the Air Force's major CC centers, the Strategic Air Command (SAC), Tactical Air Command (TAC), and Air Defense Command (ADC), to observe how they operated and what information-processing opportunities and challenges they saw for the future. Originally, the expectation was that the greatest needs and opportunities were for more powerful computing, large-screen displays, and data storage and retrieval, but consistently across the CC centers, the biggest needs were for more cost-effective, rapid-response, reliable, scalable, and interoperable ways of developing and evolving large CC software systems. Also, a trend analysis found that CC information-processing software costs were growing much faster than hardware costs, going from near zero in 1960 to 30 percent of total costs in 1971. A projection of the hardware-to-software cost percentages indicated that the ratio would go from 30:70 in 1971 to 70:30 by 1985.

What main recommendations came out of that study?

Some major study recommendations were to significantly increase Air Force investment in improved approaches to software development and evolution. These also drew on recommendations from the two NATO conferences. The top recommendation was for greater research into system design and exercise technology. This extended the NATO recommendations on structured programming into structured analysis and rapid CC prototyping. Next highest was for software and system certification. This extended the NATO emphasis on software reliability to include security and safety. The third of the top three was for research into software timeliness and flexibility, into what is now called set-based design for avoidance of brittle point-solution architectures. Further recommendations were for increased research investments in computer hardware survivability, data security, multisource data fusion, and image processing.

What evidence was provided to support those recommendations?

The evidence included summaries of the SAC, TAC, and ADC CC system challenges and opportunities, and the results of the quantitative studies of software and hardware costs, schedules, reliability, maintainability, and availability. The study had a steering group, including three CC-experienced Air Force major generals, who helped focus the recommendations and helped vouch for their value.

What were these initiatives' results in terms of concrete actions, better awareness, the community coming together, or what-have-you?

They helped the Air Force Research Laboratories to significantly increase their investments in CC hardware and software technologies. Later, I published an article in the widely read journal *Datamation* called "Software and Its Impact: A Quantitative Assessment" that drew further attention to the software problems and opportunities, and also referenced the NATO reports on software reliability and management.[4] This also involved me in becoming the co-program chair with Tony Hoare, one of the NATO conference principals, of the large 1975 ACM/IEEE Conference on Software Reliability, often called ICSE 0.

ICSE 0, as a precursor to the International Conference on Software Engineering, right?

Yes. The NATO reports' strong concern with software reliability spawned a large 1973 IEEE Conference on Software Reliability in New York City. Subsequently, IEEE and ACM combined to sponsor a follow-up April 1975 ACM/IEEE Conference on Software Reliability in Los Angeles. It was a large conference: 62 papers and over 600 attendees. One outcome was an agreement to broaden the scope of future conferences to cover the whole of software engineering. This was picked up by IEEE, which held the 1st National Conference on Software Engineering in September 1975 in Washington, DC. Due to the short lead time and US focus, NCSE 1 was much smaller (11 papers, 10 from the US; around 100 attendees). With longer lead times, sponsorship by both ACM and IEEE, and an international outreach, ICSE 2 in San Francisco in October 1976 was much larger, and the ICSE series was on its way.

Previously you mentioned quantitative studies to support the recommended changes. That sounds nontrivial. Was this common? Or was it the beginning of a new, more rational approach?

During 1973, I was captivated by the prospect of making software engineering into a quantitative discipline. As this would have been hard to do at RAND, whose largest software team was four developers, I explored the prospect of doing this with the companies that provided data for the CCIP-85 software study contracts. This led to my being offered the position of Director of Software Research and Technology at TRW, which I accepted in September 1973.

TRW was a stimulating place to work. It had enlightened managers, highly talented system and software engineers, and a culture of continuous learning and improvement. In 1973, it was in the top five in the *Datamation* list ranked by annual software income, and subsequently reached number two, second only to IBM. At the time, I thought that a fully quantified software engineering methodology would be complex and take a good five years to work out. This was by far my largest underestimate of a software-related project, as I'm still at it today.

So how much progress have you been able to make since then?

TRW's needs stimulated several significant contributions. Its concern with other software quality factors besides reliability resulted in a 1973 National Bureau of Standards-sponsored Characteristics of Software Quality study, summarized in an ICSE 2 paper and a subsequent 1978 book. TRW's concern with software cost estimation and availability of software cost data led to the development of the Constructive Cost Model (COCOMO) and the 1981 *Software Engineering Economics* book. Its concern that the waterfall process model was a poor fit to increasingly human-interactive software systems led to the development of the 1986 spiral process model. Its concern with software risk management led to the publication of the 1989 *Software Risk Management* book.

What came next, in terms of continuing repercussions of the NATO conference? What important events helped establish software engineering as a discipline and mobilized the software community?

ICSE 4, in Munich in September 1979, chaired by Friedrich Bauer, celebrated the 1968 NATO Conference, also in Germany (Garmisch) and chaired by Bauer, by having four seminal invited presentations. They were about the past, present, desirable, and future state of software engineering. "Past" as in "Software Engineering in 1968," delivered by Brian Randell; "present" as in "Software Engineering—as It Is," which I delivered; "desirable" as in "My Hopes of Computing Science," by Edsger Dijkstra; and "future" as in "Look Ahead at Software Engineering," by Wlad Turski.

Tell us more. Let's start with "Software Engineering in 1968."

Brian Randell's 1968 survey reflected his thorough approach to computing history. In terms of the marketplace, software was becoming a commodity, and some chief information officers were finding they were spending about as much on software as they were on hardware, and they were getting concerned that the hardware vendors would start charging separately for their systems software (unbundling). Some related 1968 developments were the issuance of the first patent for software and the results of the SDC Sackman–Grant study of the impact of interactive programming on software productivity. Interactive programming was helpful, but its influence was small compared to the 25:1 difference in individual programmers' productivity.

1968 also witnessed Edsger Dijkstra's "Go To Statement Considered Harmful" letter and Larry Constantine's definition of modularity, coupling, and cohesion. These were followed by Dijkstra's THE multiprogramming system and in 1969 by his *Notes on Structured Programming*, which was to spawn offshoots such as structured analysis, structured design, structured testing, etc. Another modular approach was Frank Zurcher and Brian Randell's iterative multilevel modeling. A further 1968 observation about how many software systems were really organized was Conway's law: The structure of a software system reflects the structure of the organization that developed it.

What about you? What did you talk about? What defined software engineering in 1979, a decade after the NATO conference?

My challenge was to summarize how well the software engineering field was coming along as an engineering discipline. Fortunately, I had been teaching an MS-level software engineering course at the University of Southern California with about 50 students, and had come across a paper that summarized 10 key principles learned on developing several large projects: William Hosier's "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming." We can summarize these principles as testable requirements, precise interface specifications, early planning and specification, lean staffing in early phases, core and time budgeting, careful choice of language, objective progress monitoring, defensive programming, integration planning and budgeting, and early test planning.

How did these principles manifest themselves in the students' projects?

For example, with respect to testable requirements, Hosier had stated, "It is easy to write specifications in such terms that conformance is impossible to demonstrate."[5] My students' experience confirmed this characterization—for example, with this anecdote: "A requirements spec was generated. It has a number of untestable requirements, with phrases like 'appropriate response' all too common."

With respect to precise interface specifications, Hosier had said, "This is apt to be a monumental and tedious chore, but every sheet of accurate interface definition is quite literally worth more than its weight in gold."[5] The student's project experience, again, supported this: "The interface schematics were changed over the years and not updated, so when we interfaced with the lines, fuses were burned, lights went out, …." We had similar confirmations for the other principles.

Hosier's paper was published in 1961, but its lessons learned were frequently not being practiced 18 years later. Some reasons for this were included in the ICSE 4 paper. First, the field has been growing rapidly. Different approaches are appropriate for open-source software, agile methods, and multiorganization systems of systems such as supply chain management and crisis management, although many of the older principles still apply. The field is also growing in the number of people assimilated per year, leading to the next point, which is, we haven't been teaching many of the lessons learned to students. A 1979 survey by Richard Thayer found that 18 of the 20 major software engineering issues were only lightly covered in the instructors' courses. The main reasons given for the light coverage were lack of expertise, lack of texts and other teaching materials, and inappropriateness for computer science departments.

The next reason is that we have our role models mixed up. In one of TRW's non-aerospace companies, the heroes were the indispensable programmers that carried the designs around in their heads, but were there to pull three all-nighters to get the system delivered on time. Jerry Weinberg, a highly humanitarian person, said in his 1971 book *The Psychology of Computer Programming*, "If a programmer is indispensable, get rid of him as soon as possible."[6]

The fourth reason is that we often take a restricted view of software engineering, which focuses on how soon the project can get through with requirements, architecting, and planning, so that it can get on to the more familiar job of programming.

Finally, we have been resisting the required discipline.

To come full circle, could you comment on the third and fourth topics by Dijkstra and Turski? First, what was the ideal of software engineering according to Dijkstra?

Edsger Dijkstra had changed the title of his contribution from "Software

Engineering as It Should Be" to "My Hopes for Computing Science." This reflects his emphasis on separation of concerns. His belief was that computer science was best focused on making computer programming into a precise mathematical science, and that the software aspects of other engineering disciplines such as engineering management, engineering ergonomics, and engineering economics were best left to others. Toward the end, he identified two approaches for addressing the need to cover both correctness concerns and efficiency concerns—abstract data types and program transformation—and concluded that they are useful but not complete solutions. A particular shortfall in their coverage is that there can be several forms of efficiency besides computing speed.

Another part of the Dijkstra's contribution provides his recommendations on how to proceed in addressing such challenges:

1. separation of concerns and effective use of abstraction;
2. the design and use of notations, tailored to one's manipulative needs; and
3. avoiding case analysis—in particular, combinatorially exploding ones.

A difficulty with step 3 is that there may be many users with different priorities, and as [I mentioned] above, satisfying them all may be impossible.

And how did Wlad Turski see things progressing at the time? How does that compare to what has transpired since then?

If Wlad Turski were to come back today, I think he would be much surprised at how different software engineering is from his predictions in "Look Ahead at Software Engineering." He was right on in predicting that nearly everyone would be relying on computers and software. His vision featured that nearly everyone would be learning how to program, starting in primary schools, and graduating to increasingly powerful programming methods, languages, and tools. A good part of his paper discusses the challenges of such powerful languages: they should be extensible, modular, adaptable, and scalable, but also having versions embodying a person's natural language—a formidable problem.

As we know now, the evolution of human–computer interfaces diverged incredibly from Wlad's projection of human–computer interaction as programming. It was also in 1968 that Doug Engelbart at SRI gave the "mother of all demos" at the ACM/IEEE Fall Joint Computer Conference. It demonstrated almost all of the fundamental elements of modern personal computing: windows, hypertext, graphics, efficient navigation and command input, videoconferencing, the computer mouse, word processing, dynamic file linking, revision control, and a collaborative real-time editor.

A further paradigm shift had emerged in October 1969 with the first message sent over the Arpanet from the University of California, Los Angeles to SRI, and a working 4-node version of the Arpanet by December 1969. Further productization of these capabilities came with Xerox PARC [Palo Alto Research Center] and their engineering of the technology into the Alto workstation, Steve Jobs in making a reasonably priced version of the Alto with the Macintosh, and Bill Gates converting his Microsoft infrastructure into Windows. Further Apple exploitation of microelectronics technology led to the emergence of smartphones, and calling up your desired services by poking at something you hold in your hand, and more recently by speaking to it: a far cry from Wlad Turski's

ABOUT THE AUTHORS

HAKAN ERDOGMUS is a teaching professor of electrical and computer engineering at Carnegie Mellon University's Silicon Valley campus. Contact him at hakan.erdogmus@sv.cmu.edu.

NENAD MEDVIDOVIĆ is a professor in the University of Southern California's Computer Science Department. Contact him at neno@usc.edu.

everyone-a-programmer vision in 1969. This is tremendously powerful technology that can be used to empower people or also to empower governments to control people.

Any final thoughts on what the future holds?

Sure. For a couple of looks at the possible future evolution of smartphone technology, I'd recommend for empowering people that you look at the article, "Estonia, The Digital Republic," by Nathan Heller in the December 18, 2017 issue of the *New Yorker*. For empowering governments, I'd recommend that you look at the article, "Inside China's Vast New Experiment in Social Ranking," by Mara Hvistendahl in the December 14, 2017 issue of *Wired*.

Thank you, Professor Boehm. It has been very enlightening.

Thanks for the opportunity. If you're in the software field, at least you'll have some appreciation of the challenges of living in a world where autonomous systems are making decisions for you, the platforms you count on are changing every few seconds, hackers are getting smarter at taking advantage of you, and continuous learning will be essential. Keep on learning!

References

1. B. Boehm, *Some Information Processing Implications of Air Force Space Missions: 1970–1980*, tech. report RM-6213-PR, RAND Corp., Jan. 1970.
2. P. Naur and B. Randell, eds., *Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968*, NATO Scientific Affairs Div., Jan. 1969.
3. J. Buxton and B. Randell, eds., *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969*, NATO Science Committee, Apr. 1970.
4. B. Boehm, "Software and Its Impact: A Quantitative Assessment," *Datamation*, May 1973, pp. 46–59.
5. W. Hosier, "Pitfalls and Safeguards in Real-Time Digital Systems with Emphasis on Programming," *IRE Trans. Eng. Management*, vol. EM-8, no. 2, 1961, pp. 99–115.
6. G.M. Weinberg, *The Psychology of Computer Programming*, Von Nostrand Reinhold, 1971.