DOI:10.1145/1787234.1787249

Thomas J. Misa, Editor

Interview

An Interview with Edsger W. Dijkstra

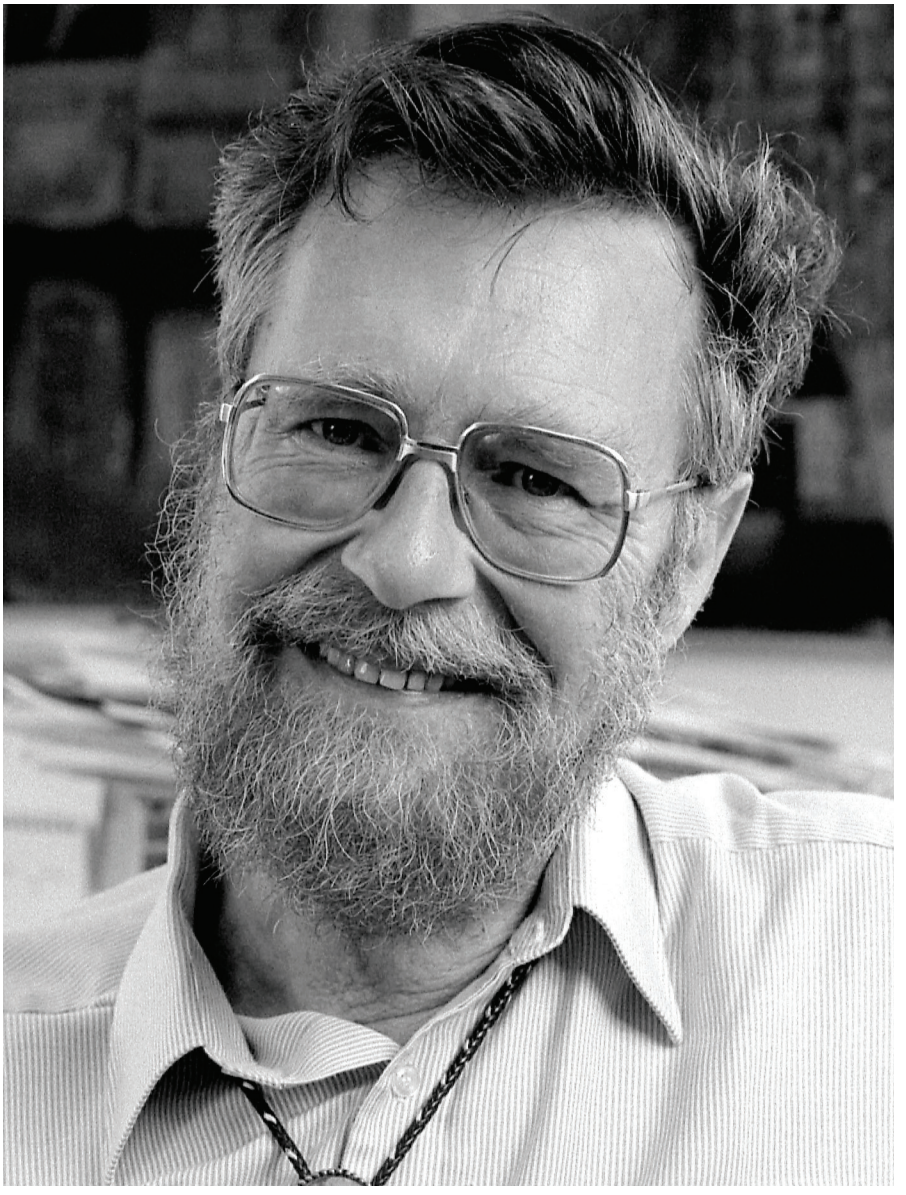
The computer science luminary, in one of his last interviews before his death in 2002, reflects on a programmer's life.

THE CHARLES BABBAGE INSTITUTE holds one of the world's largest collections of research-grade oral history interviews relating to the history of computers, software, and networking. Most of the 350 interviews have been conducted in the context of specific research projects, which facilitate the interviewer's extensive preparation and often suggest specific lines of questions. Transcripts from these oral histories are a key source in understanding the history of computing, since traditional historical sources are frequently incomplete. This interview with programming pioneer Edsger Dijkstra (1930–2002) was conducted by CBI researcher Phil Frana at Dijkstra's home in Austin, TX, in August 2001 for a NSF-KDI project on "Building a Future for Software History."

Winner of ACM's A.M. Turing Award in 1972, Dijkstra is well known for his contributions to computer science as well as his colorful assessments of the field. His contributions to this magazine continue to enrich new generations of computing scientists and practitioners.

We present this interview posthumously on the eighth anniversary of Dijkstra's death at age 72 in August 2002; this interview has been condensed from the complete transcript, available at http://www.cbi.umn.edu/oh.

—Thomas J. Misa



PHOTOGRAPH COURTESY OF THE UNIVERSITY OF TEXAS AT AUSTIN

How did your career start?

It all started in 1951, when my father enabled me to go to a programming course in Cambridge, England. It was a frightening experience: the first time that I left the Netherlands, the first time I ever had to understand people speaking English. I was all by myself, trying to follow a course on a totally new topic. But I liked it very much. The Netherlands was such a small country that Aad van Wijngaarden, who was the director of the Computation Department of the Mathematical Centre in Amsterdam, knew of this, and he offered me a job. And on a part-time basis, I became the programmer of the Mathematical Centre in March of 1952. They didn't have computers yet; they were trying to build them. The first eight years of my programming there I developed the basic software for a series of machines being built at the Mathematical Centre. In those years I was a very conservative programmer. The way in which programs were written down, the form of the instruction code on paper, the library organization; it was very much modeled after what I had seen in 1951 in Cambridge.

When you got married in 1957, you could not enter the term "programmer" into your marriage record?

That's true. I think that "programmer" became recognized in the early 1960s. I was supposed to study theoretical physics, and that was the reason for going to Cambridge. However, in 1955 after three years of programming, while I was still a student, I concluded that the intellectual challenge of programming was greater than the intellectual challenge of theoretical physics, and as a result I chose programming. Programming was so unforgiving. If something went wrong, I mean a zero is a zero and a one is a one. I had never used someone else's software. If something went wrong, I had done it. And it was that unforgivingness that challenged me.

I also began to realize that in some strange way, programs could become very complicated or tricky. So it was in 1955 when I decided not to become a physicist, to become a programmer instead. At the time programming didn't look like doing science; it was just a mixture of being ingenious and being accurate. I envied my hardware

I had never used someone else's software. If something went wrong, I had done it. And it was that unforgivingness that challenged me.

friends, because if you asked them what their professional competence consisted of, they could point out that they knew everything about triodes, pentodes, and other electronic gear. And there was nothing I could point to!

I spoke with van Wijngaarden in 1955, and he agreed that there was no such thing as a clear scientific component in computer programming, but that I might very well be one of the people called to make it a science. And at the time, I was the kind of guy to whom you could say such things. As I said, I was trained to become a scientist.

What projects did you work on in Amsterdam?

When I came in 1952, they were working on the ARRA,[a] but they could not get it reliable, and an updated version was built, using selenium diodes. And then the Mathematical Centre built a machine for Fokker Aircraft Industry. So the FERTA,[b] an updated version of the ARRA, was built and installed at Schiphol. The installation I did together with the young Gerrit Blaauw who later became one of the designers of the IBM 360, with Gene Amdahl and Fred Brooks.

One funny story about the Fairchild F27: On my first visit to Australia, I flew on a big 747 from Amsterdam to Los Angeles, then on another 747 I flew to Sydney or Melbourne. The final part of the journey was on an F27 to Canberra. And we arrived and I met my host, whom I had never met before. And he was very apologetic that this world traveler had to do the last leg of the journey on such a shaky two-engine turboprop. And it gave me the dear opportunity for a one-upmanship that I never got again. I could honestly say, "Dr. Stanton, I felt quite safe: I calculated the resonance frequencies of the wings myself." [laughter]

In 1956, as soon as I had decided to become a programmer, I finished my studies as quickly as possible, since I no longer felt welcome at the university: the physicists considered me as a deserter, and the mathematicians were dismissive and somewhat contemptuous about computing. In the mathematical culture of those days you had to deal with infinity to make your topic scientifically respectable.

There's a curious story behind your "shortest path" algorithm.

In 1956 I did two important things, I got my degree and we had the festive opening of the ARMAC.[c] We had to have a demonstration. Now the ARRA, a few years earlier, had been so unreliable that the only safe demonstration we dared to give was the generation of random numbers, but for the more reliable ARMAC I could try something more ambitious. For a demonstration for non-computing people you have to have a problem statement that non-mathematicians can understand; they even have to understand the answer. So I designed a program that would find the shortest route between two cities in the Netherlands, using a somewhat reduced roadmap of the Netherlands, on which I had selected 64 cities (so that in the coding six bits would suffice to identify a city).

What's the shortest way to travel from Rotterdam to Groningen? It is the algorithm for the shortest path, which I designed in about 20 minutes. One morning I was shopping in Amsterdam with my young fiancée, and tired, we sat down on the café terrace to drink a cup of coffee and I was just thinking about whether I could do this, and I then

a Automatische Relais Rekenmachine Amsterdam = Automatic Relay Calculator Amsterdam.
b Fokker Electronische Rekenmachine Te Amsterdam = Fokker Electronic Calculator In Amsterdam
c Automatische Rekenmachine MAthematische Centrum = Automatic Calculator Mathematical Centre

designed the algorithm for the shortest path. As I said, it was a 20-minute invention. In fact, it was published in 1959, three years later. The publication is still quite nice. One of the reasons that it is so nice was that I designed it without pencil and paper. Without pencil and paper you are almost forced to avoid all avoidable complexities. Eventually that algorithm became, to my great amazement, one of the cornerstones of my fame. I found it in the early 1960s in a German book on management science—"Das Dijkstra'sche Verfahren" ["Dijkstra's procedure"]. Suddenly, there was a method named after me. And it jumped again recently because it is extensively used in all travel planners. If, these days, you want to go from here to there and you have a car with a GPS and a screen, it can give you the shortest way.

When was the "shortest path" algorithm originally published?

It was originally published in 1959 in *Numerische Mathematik* edited by F.L. Bauer. Now, at the time, an algorithm for the shortest path was hardly considered mathematics: there was a finite number of ways of going from A to B and obviously there is a shortest one, so what's all the fuss about? It remained unpublished until Bauer asked whether we could contribute something. In the meantime I had also designed the shortest sub-spanning tree for my hardware friends. You know, on the big panel you have to connect a whole lot of points with the same copper wire because they have to have the same voltage. How do you minimize the amount of copper wire that connects these points? So I wrote "A note on two problems in connection with graphs."[2] Years later when I went to my ophthalmologist—he did not even know that I was a computing scientist—he said, "Have you designed the algorithm for GPS?" It turned out he had seen the *Scientific American* of November 2000.[10]

How could you tell if early programs were correct?

For those first five years I had always been programming for non-existing machines. We would design the instruction code, I would check whether I could live with it, and my hardware friends would check that they could build it. I would write down the formal specification of the machine, and all three of us would sign it with our blood, so to speak. And then our ways parted. All the programming I did was on paper. So I was quite used to developing programs without testing them.

There was not a way to test them, so you've got to convince yourself of their correctness by reasoning about them. A simple writing error did not matter as long as the machine wasn't there yet, and as soon as errors would show up on the machine, they would be simple to correct. But in 1957, the idea of a real-time interrupt created a vision of a program with non-reproducible errors, because a real-time interrupt occurs at an unpredictable moment. My hardware friends said, "Yes, yes, we see your problem, but surely you must be up to it..." I learned to cope with it. I wrote a real-time interrupt handler that was flawless and that became the topic of my Ph.D. thesis.[3] Later I would learn that this would almost be considered an un-American activity.

How was the computing culture in America different?

Well, the American reaction was very different. When IBM had to develop the software for the 360, they built one or two machines especially equipped with a monitor. That is an extra piece of machinery that would exactly record when interrupts took place. And if something went wrong, it could replay it again. So they made it reproducible, yes, but at the expense of much more hardware than we could afford. Needless to say, they never got the OS/360 right.

In the mathematical culture of those days you had to deal with infinity to make your topic scientifically respectable.

The OS/360 monitor idea would have never occurred to a European?

No, we were too poor to consider it and we also decided that we should try to structure our designs in such a way that we could keep things under our intellectual control. This was a major difference between European and American attitudes about programming.

How did the notion of program proofs arise?

In 1959, I had challenged my colleagues at the Mathematical Centre with the following programming task. Consider two cyclic programs, and in each cycle a section occurs called the critical section. The two programs can communicate by single reads and single writes, and about the relative speeds of the programs nothing is known. Try to synchronize these programs in such a way that at any moment in time at most one of them is engaged in its critical section.[d] I looked at it and realized it was not trivial at all, there were all sorts of side conditions. For instance, if one of the programs would stay for a very long time in its noncritical section, the other one should go on unhampered. We did not allow 'After-you-after-you' blocking, where the programs would compete for access to the critical section and the dilemma would never be solved. Now, my friends at the Mathematical Centre handed in their solutions, but they were all wrong. For each, I would sketch a scenario that would reveal the bug. People made their programs more sophisticated and more complicated. The construction and counterexamples became even more time-consuming, and I had to change the rules of the game. I said, "Sir, sorry, from now onward I only accept a solution with an argument why it is correct."

Within three hours or so Th. J. Dekker came with a perfect solution and a proof of its correctness. He had analyzed what kind of proof would be needed. What are the things I have to show? How can I prove them? Having

d This is an implementation of the mutual exclusion problem, which later became a cornerstone of the THE multiprogramming system [THE = Technische Hogeschool Eindhoven (Technical University Eindhoven)].

settled that, he wrote down a program that met the proof's requirement. You lose a lot when you restrict the role of mathematics to program verification as opposed to program construction or derivation.

Another experience in 1959 was attending the "zeroth" IFIP Congress in Paris. My international contacts had started in December 1958, with the meetings for the design of ALGOL 60. My boss, Aad van Wijngaarden, had had a serious car accident, and Jaap Zonneveld and I, as his immediate underlings, had to replace him. Zonneveld was a numerical analyst, while I did the programming work. The ALGOL 60 meetings were about the first time that I had to carry out discussions spontaneously in English. It was tough.

You've remarked that learning many different languages is useful to programming.

Oh yes, it's useful. There is an enormous difference between one who is monolingual and someone who at least knows a second language well, because it makes you much more conscious about language structure in general. You will discover that certain constructions in one language you just can't translate. I was once asked what were the most vital assets of a competent programmer. I said "mathematical inclination" because at the time it was not clear how mathematics could contribute to a programming challenge. And I said "exceptional mastery" of his native tongue because you have to think in terms of words and sentences using a language you are familiar with.

How was ALGOL 60 a turning point?

Computing science started with ALGOL 60. Now the reason that ALGOL 60 was such a miracle was that it was not a university project but a project created by an international committee. It also introduced about a half-dozen profound novelties. First of all, it introduced the absence of such arbitrary constraints as, say, ruling out the subscripted subscript, the example I mentioned. A second novelty was that at least for the context-free syntax, a formal definition was given. That made a tremendous difference! It turned parsing into a rigorous discipline, no longer a lot of handwaving. But perhaps more important, it made compiler writing and language definition topics worthy of academic attention. It played a major role in making computing science academically respectable. The third novelty was the introduction of the type "Boolean" as a first-class citizen. It turns the Boolean expression from a statement of fact that may be wrong or right into an expression that has a value, say, "true" or "false." How great that step was I learned from my mother's reaction. She was a gifted mathematician, but she could not make that step. For her, "three plus five is ten" was not a complicated way of saying "false"; it was just wrong.

Potentially this is going to have a very profound influence on how mathematics is done, because mathematical proofs, can now be rendered as simple calculations that reduce a Boolean expression by value-preserving transformations to the value "true." The fourth novelty was the introduction of recursion into imperative programming. Recursion was a major step. It was introduced in a sneaky way. The draft ALGOL 60 report was circulated in one of the last weeks of December 1959. We studied it and realized that recursive calls were all but admitted, though it wasn't stated. And I phoned Peter Naur—that call to Copenhagen was my first international telephone call; I'll never forget the excitement!—and dictated to him one suggestion. It was something like "Any other occurrence of the procedure identifier denotes reactivation of the procedure." That sentence was inserted sneakily. And of all the people who had to agree with the report, none saw that sentence. That's how recursion was explicitly included.

All the programming I did was on paper. So I was quite used to developing programs without testing them.

Was this called recursion at that time?

Oh yes. The concept was quite well known. It was included in LISP, which was beginning to emerge at that time. We made it overlookable. And F.L. Bauer would never have admitted it in the final version of the ALGOL 60 Report, had he known it. He immediately founded the ALCOR Group. It was a group that together would implement a subset of ALGOL 60, with recursion emphatically ruled out.

What were other novelties in ALGOL 60?

A fifth novelty that should be mentioned was the block structure. It was a tool for structuring the program, with the same use of the word "structure" as I used nine years later in the term "structured programming." The concept of lexical scope was beautifully blended with nested lifetimes during execution, and I have never been able to figure out who was responsible for that synthesis, but I was deeply impressed when I saw it.

Finally, the definition of the semantics was much less operational than it was for existing programming languages. FORTRAN was essentially defined by its implementation, whereas with ALGOL 60 the idea emerged that the programming language should be defined independent of computers, compilers, stores, etc.; the definition should define what the implementation should look like. Now these are five or six issues that for many years the United States has missed, and I think that is a tragedy. It was the obsession with speed, the power of IBM, the general feeling at the time that programming was something that should be doable by uneducated morons picked from the street, it should not require any sophistication. Yes... false dreams paralyzed a lot of American computing science.

When did you understand that programming was a deep subject?

I had published a paper called "Recursive Programming," again in *Numerische Mathematik*.[8] In 1961, I was beginning to realize that programming really was an intellectual challenge.

Peter Naur and I were main speakers at a workshop or a summer school in Brighton, England; there were quite a number of well-known British scientists in that audience. In the audience was Tony Hoare, but neither of us remembers that. I don't remember him because he was one of the many people in the audience, and he doesn't remember it because in his memory Peter Naur and I, both bearded and both with a Continental accent, have merged into one person. [laughter] We reconstructed years later that we were both there.

In 1962, my thinking about program synchronization resulted in the P- & V-operations. The other thing I remember was a conference in Rome on symbol manipulation, in April or so. Peter Naur was there, with his wife. There were panel discussions and Peter and I were sitting next to each other and we had all sorts of nasty comments, but we made it the rule that we would go to the microphone in turn. This had gone on for an hour or so, and van Wijngaarden, my boss, was sitting next to an American and at a given moment the American grabs his shoulder and says "My God! There are *two* of them." [laughter] This may be included in an oral history? It's not mathematics, it isn't computer science either, but it is a true story....

In September 1962, I went to the first IFIP Congress, in Munich, and gave an invited speech on advancing programming. I got a number of curtain calls: clearly I was saying something unusual. Then I became a professor of Mathematics in Eindhoven, and for two years I lectured on numerical analysis. By 1963–1964, I had designed with Carel S. Scholten the hardware channels and the interrupts of the Electrologica X8, the last machine my friends built, and then I started on the design of THE multiprogramming system.

Of course, 1964 was the year in which IBM announced the 360. I was extremely cross with Gerry Blaauw, because there were serious flaws built into the I/O organization of that machine.[7] He should have known about the care that has to go into the design of such things, but that was clearly not a part of the IBM culture. In my Turing Lecture I described the week that I studied the specifications of the 360, it was [laughter] the darkest week in my professional life. In a NATO Conference on Software Engineering in 1969 in Rome,[11] I characterized the Russian decision to build a bit-compatible copy of the IBM 360 as the greatest American victory in the Cold War.

Thanks to my isolation, I would do things differently than people subjected to the standard pressures of conformity. I was a free man.

Okay now, 1964–1965. I had generalized Dekker's solution for *N* processes and the last sentence of that one-page article is, "And this, the author believes, completes the proof." According to Doug Ross, it was the first publication of an algorithm that included its correctness proof. I wrote "Cooperating Sequential Processes," and I invented the Problem of the Dining Quintuple, which Tony Hoare later named the Problem of the Dining Philosophers.[5]

When did you first visit the U.S.?

My first trip to the U.S. was in 1963. That was to an ACM Conference in Princeton. And I visited a number of Burroughs offices; that was the first time I met Donald Knuth. I must already have had some fame in 1963, because there was an ACM workshop with about 60 to 80 participants and I was invited to join. And they paid me $500. I didn't need to give a speech, I didn't need to sit in a panel discussion, they just would like me to be there. Quite an amazing experience.

What about your first two trips to America surprised you about the profession?

Well, the first lecture at that ACM workshop was given by a guy from IBM. It was very algebraic and complicated. On the blackboard he wrote wall-to-wall formulae and I didn't understand a single word of it. But there were many people that joined the discussion and posed questions. And I couldn't understand those questions either. During a reception, I voiced my worry that I was there on false premises. "The first speaker, I did not understand a word of it." "Oh," he said, "none of us did. That was all nonsense and gibberish, but IBM is sponsoring this, so we had to give the first slot to an IBM speaker." Well, that was totally new for me. Let's say that the fence between science and industry, the fence around a university campus, is here [in the U.S.] not as high as I was used to.

How did GO TO become 'harmful'?

In 1967 was the ACM Conference on Operating Systems Principles in Gatlinburg. That, I think, was the first time that I had a large American audience. It was at that meeting where one afternoon I explained to Brian Randell and a few others why the GO TO statement introduced complexity. And they asked me to publish it. So I sent an article called "A Case Against the GO TO Statement" to *Communications of the ACM*. The editor of the section wanted to publish it as quickly as possible, so he turned it from an article into a Letter to the Editor. And in doing so, he changed the title into, "GO TO Statement Considered Harmful."[4] That title became a template. Hundreds of writers have "X considered harmful," with X anything. The editor who made this change was Niklaus Wirth.

Why is "elegance" in programming important?

1968 was exciting because of the first NATO Conference on Software Engineering, in Garmisch. In *BIT* I published a paper, on "A Constructive Approach to the Problem of Program Correctness."[1] 1968 was also the year of the IBM advertisement in *Datamation*, of a beaming Susie Meyer who had just solved all her programming problems by switching to PL/I. Those were the days we were led to believe that the problems of programming were the problems of the deficiencies of the programming language you were working with. How did I characterize it? "APL is a mistake, carried

through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums." I thought that programmers should not be puzzle-minded, which was one of the criteria on which IBM selected programmers. We would be much better served by clean, systematic minds, with a sense of elegance. And APL, with its one-liners, went in the other direction. I have been exposed to more APL than I'd like because Alan Perlis had an APL period. I think he outgrew it before his death, but for many years APL was "it."

Why did your "structured programming" have such impact?

In 1969, I wrote "Notes on Structured Programming,"[6] which I think owed its American impact to the fact that it had been written at the other side of the Atlantic Ocean; which has two very different sides. I can talk about this with some authority, having lived here [in the U.S.] for the better part of 17 years. I think that thanks to the greatly improved possibility of communication, we overrate its importance. Even stronger, we underrate the importance of isolation. See, look at what that 1963 invitation to the ACM workshop illustrates, at a time when I had published very little. I had implemented ALGOL 60 and I had written a real-time interrupt handler, I had just become a professional programmer. Yet I turned out to be quite well known. How come? Thanks to my isolation, I would do things differently than people subjected to the standard pressures of conformity. I was a free man.

What were other differences between Europe and the U.S.?

One of the things that saved Europe was that until 1960 or so, it was not considered an interesting market. So we were ignored. We were spared the pressure. I had no idea of the power of large companies. Only recently I learned that in constant dollars the development of the IBM 360 has been more expensive than the Manhattan Project.

I was beginning to see American publications in the first issue of *Communications of the ACM*. I was shocked by the clumsy, immature way in which they talked about computing. There was a very heavy use of anthropomorphic terminology, the "electronic brain" or "machines that think." That is absolutely killing. The use of anthropomorphic terminology forces you linguistically to adopt an operational view. And it makes it practically impossible to argue about programs independently of their being executed.

Is this why artificial intelligence research seemingly doesn't take hold in Europe?

There was a very clear financial constraint: at the time we had to use the machines we could build with the available stuff. There is also a great cultural barrier. The European mind tends to maintain a greater distinction between man and machine. It's less inclined to describe machines in anthropomorphic terminology; it's also less inclined to describe the human mind in mechanical terminology. Freud never became the rage in Europe as he became in the United States.

You've said, "The tools we use have a profound and devious influence on our thinking habits, and therefore on our thinking abilities."

The devious influence was inspired by the experience with a bright student. In the oral examination we solved a problem. Together we constructed the program, decided what had to be done, but very close to the end, the kid got stuck. I was amazed because he had understood the problem perfectly. It turned out he had to write a subscripted value in a subscript position, the idea of a subscripted subscript, something that was not allowed in FORTRAN. And having been educated in FORTRAN, he couldn't think of it, although it was a construction that he had seen me using at my lectures.

So the use of FORTRAN made him unable to solve that?

Indeed. When young students have difficulty in understanding recursion, it is always due to the fact that they had learned programming in a programming language that did not permit it. If you are now trained in such an operational way of thinking, at a given moment your pattern of understanding becomes visualizing what happens during the execution of the algorithm.

The only way in which you can see the algorithm is as a FORTRAN program.

And what's the answer then for our future students to avoid the same trap?

Teach them, as soon as possible, a decent programming language that exercises their power of abstraction. During 1968 in Garmisch I learned that in the ears of the Americans, a "mathematical engineer" [such as we educated in Eindhoven] was a contradiction in terms: the American mathematician is an impractical academic, whereas the American engineer is practical but hardly academically trained. You notice that all important words carry different, slightly different meanings. I was disappointed in America by the way in which it rejected ALGOL 60. I had not expected it. I consider it a tragedy because it is a symptom of how the United States is becoming more and more a-mathematical, as Morris Kline illustrates eloquently.[9] Precisely in the century which witnesses the emergence of computing equipment, it pays so much to have a well-trained mathematical mind.

In 1963 Peter Patton, in *Communications of the ACM*, wrote that European programmers are fiercely independent loners whereas Americans are team players. Or is it the other way?

At the Mathematical Centre, we used to cooperate on large projects and apply a division of labor; it was something of a shock when I went to the Department of Mathematics at Eindhoven where everybody worked all by himself. After we had completed the THE System, for instance, Nico Habermann wrote a thesis about the Banker's Algorithm, and about scheduling, sharing, and deadlock prevention. The department did not like that because it was not clear how much he had done by himself. They made so much protest that Cor Ligtmans, who should have written his Ph.D. thesis on another aspect of THE System, refused to do so.

Is the outcome of the curricula different in Europe and America?

I must be very careful with answering this because during my absence, the role of the university, the financing of the university, and the fraction of the population it is supposed to address have changed radically. That already started in the 1970s. So whatever I say about the [European] university is probably idealized by memory. Yes. But a major difference was that the fence around the university campus was higher. To give you an example, when we started to design a computing science curriculum in the 1960s, one of the firm rules was that no industrial product would be the subject of an academic course. It's lovely. This immediately rules out all Java courses, and at the time it ruled out all FORTRAN courses. We taught ALGOL 60, it was a much greater eye-opener than FORTRAN.

In many places, departments of computer science were founded before the shape of the intellectual discipline stood out clearly.

Is there a relationship between the curriculum and the nature of funding of universities?

Yes. It has the greatest influence on the funding of research projects. Quite regularly I see firm XYZ proposing to give student fellowships or something and then, somewhere in the small print, that preference will be given to students who are supervised by professors who already have professional contact with the company.

Why do computer science departments often come out of electrical engineering in the U.S.—but not in Europe?

A major reason is timing. For financial reasons, Europe, damaged by World War II, was later. So the American computing industry emerged earlier. The computing industry asked for graduates, which increased the pressure on the universities to supply them, even if the university did not quite know how. In many places, departments of computer science were founded before the shape of the intellectual discipline stood out clearly.

You also find it reflected in the names of scientific societies, such as the Association for Computing *Machinery*. It's the British *Computer* Society and it was the Dutch who had Het Nederlands Reken*machine* Genootschap; without knowing Dutch, you can hear the word "machine" in that name. And you got the departments of Comput*er* Science. Rather than the department of computing science or the department of computation. Europe was later, it coined the term Informatics. Tony Hoare was a Professor of Computation.

"Information" came a bit later on?

It was the French that pushed informatique. Today the English prefer Information Technology, IT, and Information Systems, IS. I think the timing has forced the American departments to start too early. And they still suffer from it. Here, at the University of Texas, you can still observe it is the Department of Computer Scienc*es*. If you start to think about it, you can only laugh, but that time there were at least as many computer sciences as there were professors. C

References

1. Dijkstra, E.W. A constructive approach to the problem of program correctness. *BIT 8*, 3 (1968), 174–186.
2. Dijkstra, E.W. A note on two problems in connection with graphs. *Numerische Mathematik 1* (1959), 269–271.
3. Dijkstra, E.W. Communication with an automatic computer. Ph.D. dissertation, University of Amsterdam, 1959.
4. Dijkstra, E.W. Go To statement considered harmful, *Commun. ACM 11*, 3 (Mar. 1968), 147–148.
5. Dijkstra, E.W. Hierarchical ordering of sequential processes. *Acta Informatica 1* (1971), 115–138.
6. Dijkstra, E.W. Notes on structured programming. In O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, Eds., *Structured Programming*. Academic Press, London, 1972, 1–82.
7. Dijkstra, E.W. Over de IBM 360, EWD 255, n.d., circulated privately; http://www.cs.utexas.edu/ users/EWD/ewd02xx/EWD255.PDF
8. Dijkstra, E.W. Recursive programming. *Numerische Mathematik 2* (1960), 312–318.
9. Kline, M. *Mathematics in Western Culture*. Penguin Books Ltd., Harmondsmorth, Middlesex, England, 1972.
10. Menduno, M. Atlas shrugged: When it comes to online road maps, why you can't (always) get there from here. *Scientific American 283*, 11 (Nov. 2000), 20–22.
11. Randell, B. and Buxton, J.N., Eds., *Software Engineering Techniques: A Report on a Conference Sponsored by the NATO Science Committee* (Rome, Italy, Oct. 1969), NATO, 1970.