A Brief History of Software Engineering

Niklaus Wirth
ETH Zürich

This personal perspective on the art of programming begins with a look at the state of programming from about 1960, and it follows programming's development through the present day. The article examines key contributions to the field of software engineering and identifies major obstacles, which persist even today.

The term *programming* was commonly used through the mid-1960s, and referred essentially to the task of coding a computer. The term *software engineering*—referring to the highly disciplined, systematic approach to software development and maintenance—came into existence after a NATO-sponsored conference in 1968. At that conference, the difficulties and pitfalls of designing complex systems were explored in depth, and a search for solutions began that concentrated on better methodologies and tools. The most prominent of these tools were languages reflecting procedural, modular, and object-oriented styles of programming. Since 1968, the development of software engineering has been intimately tied to these tools' emergence and improvement, as well as to efforts for systematizing or automating program documentation and testing. Ultimately, analytic verification and correctness proofs were supposed to replace testing, but that has not happened.

As this article will explain, the rapid growth of computing power has made it possible in recent years to apply computing to tasks that are ever more complicated. This trend has dramatically increased the demands on software engineers. Programs and systems have become increasingly complex and almost impossible for a single individual to fully understand. The abundance of computing resources coupled with a significant drop in their cost inevitably reduced the attention given to good design. At the expense of quality, the pursuit of profit became paramount, but we should be concerned about the resulting deterioration in programming quality. Our limitations in designing complex systems are no longer determined by slow hardware, but by our own intellectual capability. From experience, we know that most programs could be significantly improved, made more reliable, more economical, and easier to use.

The 1960s and the origin of software engineering

It is unfortunate that people dealing with computers often have little interest in the history of their subject. As a result, many concepts and ideas are propagated and advertised as being new, when in fact they existed decades ago, perhaps under a different name. I believe it worthwhile to occasionally consider the past and to investigate how computing terms and concepts originated.

I regard the late 1950s as a period essential to the era of computing. At that time, large computers became available to research institutions and universities. Computers were then used primarily in engineering and the natural sciences, but they soon became indispensable in business, too. The time when they were accessible only to a few insiders in laboratories, when they frequently broke down whenever one wanted to use them, belonged to the past. Computers' emergence from the closed laboratory of electrical engineers into the public domain meant that computers' use, in particular their programming, became an activity of many. As a result, a new profession was born, and all kinds of companies began to hire programmers. The actual computers, however, remained hidden, enclosed within special rooms built to house them in those same companies. Programmers would write code and bring their programs to a counter where a dispatcher would pick up the programs and queue them for processing. The programs'

results would be fetched from the counter hours or days later. There was no interactivity between man and computer: programming and computing were separate tasks.

Programming was understood to be a sophisticated task requiring a painstaking attention to detail, along with a love for obscure codes and what might even be called "tricks." To facilitate this coding, formal notations were created that we now call programming languages. The primary idea was to replace sequences of special instruction code by mathematical formulas. The first widely known language, Fortran, was developed by IBM in 1957, soon followed by the first Algol version in 1958 and its official successor in 1960. As computers were then used for computation rather than for data storage and communication, these early languages catered mainly to numerical mathematics. In 1962, the language Cobol was issued by the US Department of Defense specifically for business applications.

But as computing capacity grew, so did the demands on programs and on programmers: tasks became ever more intricate. It was slowly recognized that programming was a difficult task, and that mastering complex problems was nontrivial, even though—or perhaps because—computers were so powerful. Salvation was sought in "better" programming languages, in more "tools," and in automation.

To be considered "better," a language should be useful in a wider area of application; it should be more like a "natural" language and also offer more facilities. For example, PL/1, developed by IBM in the early 1960s, was designed to unify the scientific and commercial worlds. It was advertised under the slogan "Everybody can program thanks to PL/1." Programming languages and their compilers became a principal cornerstone of computing science, but they fitted into neither mathematics nor electronics, the two traditional sectors where computers were used. A new discipline soon emerged, which was called *computer science* in the US and *informatics* in Europe.

In 1963, the first time-sharing system appeared. It was designed by John McCarthy at MIT and implemented on a slightly extended DEC PDP-1 computer. This time-sharing system provided the interactivity that batch-processing systems lacked. Computer manufacturers seized on the idea and soon announced time-sharing systems for their large mainframes (for example, the IBM 360/67 and the General Electric GE-645).

> **The transition from batch processing to time sharing was more difficult than anyone anticipated. As a result, systems could not be built or delivered on time, bringing some companies to the brink of collapse.**

It turned out that the transition from batch-processing to time-sharing systems was vastly more difficult than anyone had anticipated. Among other difficulties, systems were announced but then could not be delivered on time; the operational problems were too complex; research had to be conducted "on the job." The topics of multiprocessing and concurrent programming—central ingredients of time-sharing systems—had not been encountered before and were insufficiently mastered. Consequently, systems were promised but could not be completed and delivered on time. The difficulties brought big companies to the brink of collapse.

In the midst of this situation, in 1968 NATO sponsored a conference dedicated to the topic of software engineering.[1] Although critical comments had occasionally been voiced earlier about the programming profession,[2,3] it was not until the 1968 conference that programmers' difficulties were openly discussed in a public forum—and with unusual frankness—and that the terms *software engineering* and *software crisis* were coined. It was recognized that current techniques of software development were inadequate and that new, more methodical ways had to be adopted.

Programming as a discipline

The software crisis in 1968 existed despite efforts made several years previously to specifically address the issue of programming's increasing complexity. In the academic world, it was mainly Edsger W. Dijkstra and C.A.R.

Hoare who had recognized the problems endemic to programming and who offered new ideas. In 1965, Dijkstra had written his famous "Notes on Structured Programming"[4] and declared programming to be a discipline rather than a craft. Also in 1965, Hoare had published an important paper about data structuring.[5] These ideas had a profound influence on new programming languages—in particular, Pascal.[6] New languages were the vehicles in which these new ideas were to be expressed. Structured programming became supported by a structured programming language.

In 1966, Dijkstra had written a seminal paper about harmoniously cooperating processes,[7] postulating a discipline based on semaphores. Semaphores, primitives for the synchronization of concurrent processes, can be regarded as a data type with the values 0 and 1 and the atomic operators P and V. ($P(s)$ delays the calling process until s has—by action of another process—obtained the value 1, and then sets the semaphore to 0. $V(s)$ sets s to 1.) Hoare had followed in 1966 with his Communicating Sequential Processes (CSP),[8] based on the concept of channels and conditions, and also two operators, which he denoted by "?" (inquiring) and "!" (asserting a condition). Hoare had fully recognized that in the future programmers would have to cope with the difficulties posed by concurrent processes. The necessity of dealing with concurrency would clearly make a structured and disciplined methodology even more compelling.

All these efforts aside, the field of programming was still in a state of upheaval and even disarray by the time of the 1968 NATO conference. The important developments by Dijkstra, Hoare, and others did not—could not—change the software situation nor dispel all difficulties overnight. Industry could change neither its policies nor its tools rapidly enough to be of use to programmers, who were restricted to working with available tools and languages, none of which incorporated these new ideas in the 1968 time frame. Nevertheless, intensive training courses on structured programming began to be organized, notably by Harlan D. Mills at IBM. Even the US Department of Defense realized that software problems were urgent and becoming more so. For its part, the DoD initiated a project that ultimately led to the programming language Ada, a highly structured language suitable for a wide variety of applications. Software development within the DoD would then be based exclusively on Ada, and still is to a significant extent.[9]

Unix and C

While the concepts of structured programming slowly gained acceptance, notably in academia, another movement started to invade the programmers' world. It was spawned by the spread of the Unix operating system, which Ken Thompson had developed at Bell Labs, and in its simplicity contrasted markedly with the complexity of MIT's Multics. Unix was specifically designed for, and was small enough to fit, the rapidly emerging minicomputers. It was a highly welcome relief from the large operating systems established on mainframe computers.

In its wake came the language C,[8] which had been explicitly designed by Dennis Richie, also at Bell Labs, to support the development of Unix. It was therefore at least attractive, if not mandatory, to use C for developing applications that ran under Unix, which thus acted like a Trojan horse for C.

But C did not carry the spirit of structured programming. It was rather like an assembler code in the disguise of a remotely Algol-like syntax. Neither did it allow for strict checking of data types. From the point of view of software engineering, the rapid spread of C therefore represented a great leap backward (a nod to the Chinese cultural revolution, the Great Leap Forward, taking place at that time). It revealed that the community at large had hardly grasped the true meaning of the term "high-level language," which became a poorly understood buzzword. What, if anything, was to be "high level" now? Because this issue lies at the very core of software engineering, we need to elaborate.

Abstraction

Computer systems involve machines of great complexity. This complexity can be mastered intellectually by one tool only: abstraction. A language represents an abstract computer whose objects and constructs lie closer to, and reflect more directly, the problem to be represented than the concrete machine. For example, in a high-level language we deal with numbers, indexed arrays, data types, and conditional and repetitive statements, rather than with bits and bytes, addressed words, jumps, and condition codes. However, these abstractions are beneficial only if they are consistently and completely defined in terms of their own properties. If this is not the case, if the abstractions can be understood only in terms of the facilities of an underlying computer, then the benefits are marginal, almost given away. If debugging a program—undoubtedly the most pervasive activity in

software engineering—requires a "hexadecimal dump," then the use of a high-level language is hardly worth the trouble.

The widespread use of C effectively, if unintentionally, sabotaged the programming community's attempt to raise the level of software engineering. This was true because C offers abstractions which it does not in fact support: arrays that remain without index checking, data types without a consistency check, pointers that are merely addresses where addition and subtraction are applicable. One might have classified C as being somewhere on a scale between misleading and (possibly) dangerous. Regardless, people (particularly those in academia) found C to be intriguing and "better than assembly code" because it featured some syntax.

The trouble was that C's rules could easily be broken, exactly what many programmers valued. C made it possible for programmers to manage access to all of a computer's idiosyncrasies, even to those items that a high-level language would properly hide. C provided freedom, whereas high-level languages were considered straitjackets, enforcing unwanted discipline. The nature of C was effectively an invitation for programmers to use tricks and loopholes that had been necessary to achieve efficiency in the early days of computers, but which now were pitfalls that made large systems error-prone, and costly to debug and "maintain."

Languages appearing around 1985, such as Ada and C++, tried to remedy this defect and to address a much wider variety of foreseeable applications. As a consequence, these newer languages became large and their descriptions voluminous. Compilers and support tools became bulky and complex. Ultimately, instead of solving problems, these new languages became problems themselves. As Dijkstra frequently said: They belonged to the problem set rather than the solution set.

Consequently, progress in software engineering seemed to stagnate. The difficulties grew faster than new tools could resolve or restrain them. Pseudotools like software metrics, like counting the number of go-to statements in a program or measuring the average length of identifiers, had revealed themselves as being of no help, but at least software engineers were no longer judged by the number of lines of code produced per hour.

Advent of the microcomputer

The propagation of software engineering and Pascal notably did not occur in industry,

The emergence of cheap Pascal implementations involved nothing less than a turning point. Suddenly, there was a mass market. Computing went mainstream.

but on other fronts: in schools and the home, in no small way because of the emergence of microcomputers. Microcomputers first appeared on the market in 1975 (Commodore, Tandy, and Apple; much later, IBM entered the market). They were based on single-chip processors (such as the Intel 8080, Motorola 6800, and Rockwell 6502) with 8-bit data buses, 32 Kbytes or less of memory, and clock frequencies less than 1 MHz. These advances made computers affordable for individuals, in contrast to large organizations: companies and universities. But microcomputers at that time were toys, not useful computing engines.

The breakthrough for microcomputers, in the context of software development, came when it was shown that high-level languages could be used in conjunction with microcomputers. This happened in 1975 when the group that Ken Bowles led at the University of California, San Diego, built a text editor, a file system, and a debugger around the portable Pascal compiler (P-code) developed at ETH in Zurich, and they distributed it for $50. Soon, Borland Software came out with its version (Turbo Pascal) of a high-level compiler. This was at a time when other compilers were very expensive, so the emergence of these comparatively cheap Pascal implementations involved nothing less than a turning point in commercializing software for development purposes. Suddenly, there was a mass market. Computing went mainstream.

Meanwhile, requirements on software systems continued to grow and, in turn, so did the complexity of programs. The craft of programming turned to "hacking" in many cases. Methods were sought to systematize if not construction, then at least program testing and documentation. Although this was helpful, the real problems of hectic programming

under time pressure remained. In conversation, more than once, Dijkstra would often pinpoint the difficulty by saying that testing may show the presence of errors, but it can never prove their absence. He also sneered that ''software engineering is programming for those who can't.''

Programming as a mathematical discipline

In 1967, Robert W. Floyd had suggested the idea of assertions of states, of truths always valid at given points in a program.[10] It led to Hoare's seminal paper titled ''An Axiomatic Basis of Computer Programming,'' postulating the so-called Hoare logic.[11] A few years later, in 1975, Dijkstra deduced from it the calculus of predicate transformers.[12] Programming was obtaining a mathematical basis. Programs were no longer just code for controlling computers, but static texts that could be subjected to mathematical reasoning.

Although these developments were recognized at some universities, they passed virtually unnoticed in industry. Indeed, Hoare's logic and Dijkstra's predicate transformers were explained on interesting but small algorithms such as multiplication of integers, binary search, and greatest common divisor. But industry was plagued by large, even gargantuan, systems. It was not obvious that mathematical theories would ever solve real problems when the analysis of simple algorithms was demanding enough.

An eventual solution to the dilemma of mathematical rigor for small programs against the intractability of large programs as they existed in industry emerged in the form of a disciplined manner of programming, rather than from rigorous scientific theory. A major contribution to structured programming was made by David Parnas in 1972 with the concept of information hiding,[13] and at the same time by Barbara Liskov with the concept of abstract data types.[14] Both concepts embody the notion of breaking up large systems into parts called modules, and clearly defining their interfaces. If module A uses (or imports) module B, then A is called a client of B. The designer of A then need not know the details, the functioning of B, but only the properties as stated by its interface.

This principle, modularization, probably constituted the most important contribution to software engineering, that is, to the construction of systems by large groups of people. The concept of modularization is greatly enhanced by the technique of separate compilation with automatic checking of interface compatibility.

Just as structured programming had been the guiding spirit behind Pascal, modularization was the principal idea behind the language Modula-2, Pascal's successor, published in 1979.[15] In fact, the motivation for Modula-2 actually came from the language Mesa, an internal development of the Xerox Palo Alto Research Center (PARC), and itself a descendant of Pascal. The concept of modularization and separate compilation was also adopted by the language Ada (1984), which was also based largely on Pascal. In Ada, modules were called packages.

Era of the personal workstation

However, another development influenced the computing field more profoundly than all programming languages. It was the workstation, whose first incarnation, the Alto, was built in 1975 by the Xerox PARC lab.[16] In contrast to the aforementioned microcomputers, the workstation was powerful enough to allow serious software development, complex computations, and the use of a compiler for an advanced programming language. Most important, the Alto pioneered the bit-mapped, high-resolution display and the pointing device called a mouse, which together brought about a revolutionary change in computer usage. Along with the Alto, the concept of a local area network was introduced, and that of central servers for (laser) printing, large-scale file storage, and email service.

It is no exaggeration at all to claim that the modern computing era started in 1975 with the Alto. The Alto caused nothing less than a revolution, albeit slowly, and as a result many people today have no idea how computing was done before 1975, without personal, highly interactive workstations. The influence of these developments on software engineering cannot be overestimated.

As the demand for ever more complex software persistently grew, and as the difficulties became more menacing, illustrated by some spectacular failures (of which the most conspicuous was the crash of a rocket that abruptly ended a space mission), the search for panaceas began. Many cures were offered, sold, and soon forgotten. One of them, however, proved fruitful and has survived: object-oriented programming (OO).

Up until 1980, the commonly accepted model of computing was transforming data from their given state to the result, gradually transforming input into output. In its simplest

abstract form, this is the finite-state machine. This view of computing stemmed from the original task of computers: performing numerical computations. However, another model gained ground in the 1960s, originating from the simulation of discrete-event systems (such as supermarkets, assembly lines, and logistics). A discrete-event system consists of actors (processes) that come and go, that pass phases in their lifetime, and that carry a set of private data representing their current state. It proved natural to think of such actors with state as a unit, as an *object*.

Some programming languages were designed on the basis of this model, their ancestor being Ole-Johan Dahl and Kristen Nygaard's Simula in 1965. But such languages remained confined to the field of simulation of discrete-event systems. Only after the emergence of powerful personal computers did the OO model gain wider acceptance. Now, computing systems would feature windows, icons, menus, buttons, toolbars, and so on, all easily identifiable as visible objects with individual state and behavior. Languages appeared supporting this model, among them Smalltalk (developed by Adele Goldberg and Alan Kay in 1980), Object-Pascal (Larry Tesler, 1985), C++ (Bjarne Stroustrup, 1985), Oberon (Niklaus Wirth, 1988), Java (Sun Microsystems, 1995), and C# (Microsoft, 2000).

Object orientation became both a trend and a buzzword. Indeed, choosing the right model for an application is important. Nevertheless, OO is not appropriate for *all* applications.

Abundance of computer power

The period since 1985 has, until a few years ago, chiefly been characterized by enormous advances in hardware technology. Today, even tiny computers, such as mobile telephones, have 100 times more power and capacity than the biggest of 20 years ago. It is fair to say that semiconductor and disk technologies have recently determined all advances. Who, for example, would have dreamed in 1990 of memory sticks holding several gigabytes of data, of tiny disks with dozens-of-gigabytes capacity, of processors with clock rates of several gigahertz?

Such speedy development has vastly widened the area of computer applications. This has happened particularly in connection with communications technology. It is now hard to believe that before 1975 computer and communications technologies were considered separate fields. Electronics has united them, and has made the Internet pervasive, featuring a bandwidth that is apparently unlimited. I am overwhelmed when I compare this development with the first, stand-alone minicomputer that I worked with in 1965, a DEC PDP-1. The PDP-1 had a clock rate of less than 1 MHz, memory of 8,000 18-bit words, and drum storage of some 200 Kbytes. It was time-shared by up to 16 users. It is a miracle that some people persisted in believing that one day computers would become powerful enough to be useful for more than just accounting and academic exercises.

In the 1990s, the *open source* phenomenon took hold and started to spread. The distrust with which programmers regarded huge systems designed in industrial secrecy became manifest. Although programmers had previously made a limited amount of software available for free, once the Internet became ubiquitous, an even wider community of programmers decided to build software and more systematically distribute their products for free through the Internet. Although it is difficult to recognize this as a sound business principle—making the idea of patents obsolete—the bandwagon turned out to have a rather successful following. The notions of quality and responsibility in case of failure seemed irrelevant. Open source appeared as the welcome alternative to industrial hegemony and abrasive profit, and also against helpless dependence on commercial software.

It is often difficult in software engineering to distinguish between business strategies and scientific ideas. Concerning the latter, open source appears to be a last attempt to cover up failure. The writing of complicated code and the nasty decryption by others is apparently considered easier or more economical than the careful design and description of clean interfaces of modules. The easy adaptability of modules when available in source form is also a poor argument. In whose interest would a wild growth of varieties of variants ever be? Not in that of anyone concerned with high-quality engineering and professionalism.

Wasteful software

Whereas the incredible increase in the power of hardware was very beneficial for a wide spectrum of applications (we think of administrations, banks, railways, airlines, guidance systems, engineering, science), the same cannot be claimed for software engineering. Surely, software engineering has profited too from the many sophisticated development tools. But the quality of its products hardly reflects signs of great progress. No wonder: after

all, the increase of power was itself the reason for the terrifying growth of complexity. Whatever progress was made in software methodology was quickly compensated for by still higher complexity of the software tasks. This is reflected by Martin Reiser's "law": "Software is getting slower faster than hardware is getting faster."[17] Indeed, new problems have been tackled that are so difficult that engineers often have to be admired more for their optimism and courage than for their success.

What has happened in software engineering was predictable, because inherent in a field of engineering, where the demands rise, work is done under time pressure, and the cost of "materials" dramatically drops. The consequence is a true waste of cheap resources—of both computing power and storage capacity, resulting in inefficient code and bulky data. This waste has become ever present and represents a grave lack of concern for software quality. Programs' inefficiency is easily covered up by the use of faster processors, and poor data design by the use of larger storage devices. But their side effect is a decrease of quality—of reliability, robustness, and ease of use. Good, careful design is time-consuming, and costly. It is, however, still cheaper than unreliable, difficult software, when the cost of "maintenance" is factored in. The trend is disquieting, and so is the complacency of customers.

Personal reflections and conclusions

What can we do to release this logjam? There is little point in reading history unless we are willing to learn from it. Therefore, I dare to reflect on the past and will try to draw some conclusions. A primary effort must be education concerning a sense of quality. Programmers must become engaged crusaders against homemade complexity. The cancerous growth of complexity is not a thing to be admired; it must be fought wherever possible.[18] Programmers must be given time and afforded respect for work of high quality. This is crucial, ultimately more effective than more and better tools and rules. Let us embark on a global effort to prevent software from becoming known as *softwaste!*

Recently I have become acquainted with a few projects where large, commercial operating systems were discarded in favor of the Oberon system, whose principal design objective had been perspicuity and concentration on the essentials.[19] The project leaders, being obliged to deliver reliable, economical software, had recognized that they were unable to

do so as long as they (however carefully) built their own work on top of complex base software—a platform—that was neither fully described nor dependable. We know that any chain is only as strong as its weakest link. This holds also for module hierarchies. Systems can be designed with utmost care and professionalism, yet they remain error-prone if built on a complex and unreliable platform.

The crazy drive for more complexity—euphemistically called sophistication—long ago had also afflicted the most essential tool of the software engineer: the programming language. Modern languages like Java and C# may well be superior to old ones like Fortran, PL/1, and C, but they are far from perfect, and they could be much better. Their manuals of several hundred pages are an unmistakable symptom of their inadequacy. Engineers in industry, however, are rarely free from constraints. Their work theoretically must be compatible with the rest of the world, and to deviate from established standards might prove fatal to the engineers' companies.

But this cannot be said about academia. It is therefore a sad fact that academia has remained inactive and complacent about the state of programming languages. Not only has research in languages and design methodology lost its attractiveness and glamour, but worse, the tools common in industry have now quietly been adopted in the academic world, without debate and criticism. Current languages may be inevitable in industry use, but for teaching, for an orderly, structured, systematic, well-founded introduction to programming and designing algorithms, they are wrong and obsolete.

This situation is notably, and sadly, in accord with other trends of the 21st century: we teach, learn, and perform only what is immediately profitable, what is requested by students. In plain words: we focus on what sells. Universities have traditionally been exempt from this commercial focus. Universities were places where people were expected to ponder about what matters in the long run. Universities were spiritual and intellectual leaders, showing the path into the future. In our field of computing, I am afraid, they have simply become docile followers. Universities appear to have succumbed to the trendy yearning for continual innovation, and to have lost sight of the need for careful craftsmanship.

If we can learn anything from the past, it is that computer science is in essence a methodological subject. It is supposed to develop

(teachable) knowledge and techniques that are generally beneficial in a wide variety of applications. This does not mean that computer science should drift into all these diverse applications and lose its identity. Software engineering would be the primary beneficiary of a professional education in disciplined programming. Among its tools, languages figure in the forefront. A language with appropriate constructs and structure, resting on clean abstractions, is instrumental in building artifacts, and mandatory in education. Homemade, artificial complexity has no place in languages. And finally: it must be a pleasure to work with languages, because they enable us to create artifacts that we can show and be proud of.

References and notes

1. P. Naur, and B. Randell, eds., *Software Engineering, Report on a Conference Sponsored by the NATO Science Committee*, Scientific Affairs Division, NATO, 1968.
2. E.W. Dijkstra, ''Some Meditations on Advanced Programming,'' *Proc. IFIP Congress*, 1962, North-Holland, pp. 535-538.
3. R.S. Barton, ''A Critical Review of the State of the Programming Art,'' *Proc. Spring Joint Computer Conf.*, AFIPS Press, 1963, pp. 169-177.
4. E.W. Dijkstra, ''Notes on Structured Programming,'' *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, eds., Academic Press, 1972, pp. 1-82.
5. C.A.R. Hoare, ''Notes on Data Structuring,'' *Structured Programming*, O.-J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, eds., Academic Press, 1972, pp. 83-174.
6. N. Wirth, ''The Programming Language Pascal,'' *Acta Informatica*, vol. 1, 1971, pp. 35-63.
7. E.W. Dijkstra, ''Cooperating Sequential Processes,'' Sept. 1965. Reprinted in *Programming Languages,* F. Genuys, ed., Academic Press, 1968, pp. 43-112.
8. C.A.R. Hoare, ''Communicating Sequential Processes,'' *Comm. ACM*, vol. 21, no. 8, 1978, pp. 666-677.
9. J.G.P. Barnes, ''An Overview of Ada,'' *Software—Practice and Experience*, vol. 10, 1980, pp. 851-887.
10. R.W. Floyd, ''Assigning Meanings to Programs,'' *Proc. Symp. Applied Mathematics*, Am. Mathematical Soc., vol. 19, 1967, pp. 19-32.
11. C.A.R. Hoare, ''An Axiomatic Basis for Computer Programming,'' *Comm. ACM*, vol. 12, no. 10, 1969, pp. 576-580.
12. E.W. Dijkstra, ''Guarded Commands, Nondeterminacy and Formal Derivation of Programs,'' *Comm. ACM*, vol. 18, no. 8, 1975, pp. 453-457.
13. D.L. Parnas, ''Abstract Types Defined as Classes of Variables,'' *ACM SIGPLAN Notices*, vol. II, no. 2, 1976, pp. 149-154.
14. B. Liskov and S. Zilles, ''Programming with Abstract Data Types,'' *Proc. ACM SIGPLAN Symp.*, ACM Press, 1974, pp. 50-59.
15. N. Wirth, *Programming in Modula-2*, Springer, 1974.
16. C.P. Thacker et al., *Alto: A Personal Computer*, Xerox PARC, tech. report CSL-79-11, Aug. 1979.
17. N. Wirth, ''A Plea for Lean Software,'' *Computer*, Feb. 1995, p. 64.
18. Ibid., pp. 64-68.
19. M. Franz, ''Oberon: The Overlooked Jewel,'' *The School of Niklaus Wirth: The Art of Simplicity*, L. Böszörményi, J. Gutknecht, and G. Pomberger, eds., Morgan Kaufmann, 2000, pp. 41-54.

Niklaus Wirth is professor emeritus of the Swiss Federal Institute of Technology (ETH) in Zürich. He designed the programming languages Pascal (1970), Modula (1979), and Oberon (1988), and the workstations Lilith (1980) and Ceres (1986), as well as their operating software. Wirth received a PhD from the University of California at Berkeley in 1963. His honors include the IEEE Emanuel R. Piore Award (1983), the ACM A.M. Turing Award (1984), and the IEEE Computer Pioneer Award (1987). Wirth is a Foreign Associate of the National Academy of Engineering.

Readers may contact Niklaus Wirth about this article at wirth@inf.ethz.ch.

For further information on this or any other computing topic, please visit our Digital Library at http://computer.org/csdl.