A DEBATE ON Teaching Computing Science

At the ACM Computer Science Conference last February, Edsger Dijkstra gave an invited talk called "On the Cruelty of Really Teaching Computing Science." He challenged some of the basic assumptions on which our curricula are based and provoked a lot of discussion. The editors of *Communications* received several recommendations to publish his talk in these pages. His comments brought into the foreground some of the background of controversy that surrounds the issue of what belongs in the core of a computer science curriculum.

To give full airing to the controversy, we invited Dijkstra to engage in a debate with selected colleagues, each of whom would contribute a short critique of his position, with Dijkstra himself making a closing statement. He graciously accepted this offer.

We invited people from a variety of specialties, backgrounds, and interpretations to provide their comments. David Parnas is a noted software engineer who was outspoken in his criticism of the proposed Strategic Defense Initiative. William Scherlis is known for his articulate advocacy of formal methods in computer science. M. H. van Emden is known for his contributions in programming languages and philosophical insights into science. Jacques Cohen is known for his work with programming languages and logic programming and is a member of the Editorial Panel of this magazine. Richard Hamming received the Turing Award in 1968 and is well known for his work in communications and coding theory. Richard M. Karp received the Turing Award in 1985 and is known for his contributions in the design of algorithms. Terry Winograd is well known for his early work in artificial intelligence and recent work in the principles of design.

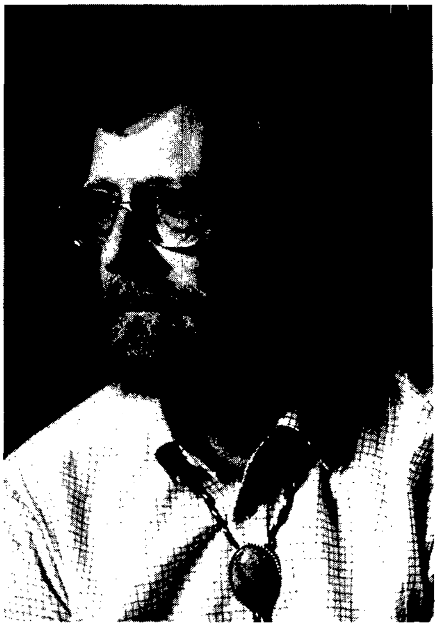
I am grateful to these people for participating in this debate and to Professor Dijkstra for creating the opening.

Peter J. Denning
Editor-in-Chief

Edsger W. Dijkstra was born in 1930 in Rotterdam, The Netherlands, where he lived until 1948. He studied mathematics and theoretical physics at the University of Leiden, from which he graduated in 1956. During that period, in September 1951, he was introduced to programming in Cambridge, England, and in March, 1952, he was appointed to the Mathematical Centre as The Netherlands' first professional programmer. In 1959 he earned his Ph.D. in Computing Science from the University of Amsterdam. In 1962 he was appointed Full Professor of Mathematics at the Eindhoven University of Technology. In 1973, while retaining links with the University, he became Research Fellow of the Burroughs Corporation, a position he enjoyed until 1984, when he received the Schlumberger Centennial Chair in Computer Sciences at the University of Texas at Austin.

Dijkstra, recipient of the 1972 ACM Turing Award, is known for early graph-theoretical algorithms, the first implementation of ALGOL 60, and the first operating system composed of explicitly synchronized sequential processes. He is also credited with the invention of guarded commands and of predicate transformers as a means for defining semantics, and programming methodology in the broadest sense of the term.

His current interests focus on the formal derivation of programs and the streamlining of the mathematical argument. His publications represent only a minor fraction of his writings—he writes, in fact, so much that he cannot afford the use of time-saving devices such as word processors. He owns, however, several fountain pens, three of which are Mont Blancs, for which he mixes his own ink.



ON THE CRUELTY OF REALLY TEACHING COMPUTING SCIENCE

The underlying assumption of this talk is that computers represent a radical novelty in our history. The first part of this talk will make much more precise what we mean in this context by the adjective "radical" and will then supply ample evidence in support of our assumption. The second half pursues some of the scientific and educational consequences of the radical novelty the computers embody.

The usual way in which we plan today for tomorrow is in yesterday's vocabulary. We do so because we try to get away with the use of concepts that we are familiar with and that have acquired their meanings in our past experience. Of course, the words and the concepts do not quite fit because our future differs from our past, but then we stretch them a little bit. It is the most common way of trying to cope with novelty. Linguists are quite familiar with the phenomenon that the meanings of words evolve over time, but also know that this is a slow and gradual process.

By means of metaphors and analogies, we try to link the new to the old, the novel to the familiar. Under sufficiently slow and gradual change, it works reasonably well; in the case of a sharp discontinuity, however, the method breaks down. Though we may glorify it with the name "common sense," our past experience is no longer relevant; the analogies become too shallow; and, the metaphors become more misleading than illuminating. This is the situation that is characteristic of the "radical" novelty.

Coping with radical novelty requires an orthogonal method. One must consider one's own past, the experiences collected, and the habits formed in it as an unfortunate accident of history, and one has to approach the radical novelty with a blank mind, consciously refusing to try to link history with what is already familiar, because the familiar is hopelessly inadequate. One has, with initially a kind of split personality, to come to grips with a radical novelty as a dissociated topic in its own right. Coming to grips with a radical novelty amounts to creating and learning a new foreign language that *cannot* be translated into one's own mother tongue. (Anyone who has learned quantum mechanics knows what I am talking about.) Needless to say, adjusting to radical novelties is not a very popular activity for it requires hard work. For the same reason, the radical novelties, themselves, are unwelcome.

By now, you may well ask why I have paid so much attention to and have spent so much eloquence on such a simple and obvious notion as the radical novelty. My reason is very simple: radical novelties are so disturbing that they tend to be suppressed or ignored to the extent that even the possibility of their existence, in general, is more often denied than admitted.

On the historical evidence I shall be short. Carl Friedrich Gauss, the Prince of Mathematicians but also somewhat of a coward, was certainly aware of the fate of Galileo—and could probably have predicted the calumniation of Einstein—when he decided to suppress Galileo's discovery of non-Euclidean geometry, thus leaving it to Bolyai and Lobatchewsky to receive the flak.

It is probably more illuminating to go a little bit further back: to the Middle Ages. One of its characteristics was that "reasoning by analogy" was rampant; another characteristic was almost total intellectual stagnation, and we now see why the two go together. A reason for mentioning this is to point out that by developing a keen ear for unwarranted analogies, one can detect a lot of medieval thinking today.

The other thing I cannot stress enough is that the fraction of the population for which gradual change seems to be all but the only paradigm of history is very large, probably much larger than one would expect. Certainly, when I started to observe it, their number turned out to be much larger than I had expected.

For instance, the vast majority of the mathematical community has never challenged its tacit assumption that doing mathematics will remain very much the same type of mental activity it has always been. New topics will come, flourish, and go as they have done in the past, but with the human brain being what it is, our ways of teaching, learning, and understanding mathematics, of problem solving, and of mathematical discovery will remain pretty much the same. Herbert Robbins clearly states why he rules out a quantum leap in mathematical ability:

> "Nobody is going to run 100 meters in five seconds, no matter how much is invested in training and machines. The same can be said about using the brain. The human mind is no different now from what it was five thousand years ago. And when it comes to mathematics, you must realize that this is the human mind at an extreme limit of its capacity."

My comment in the margin was "so reduce the use of the brain and calculate!" Using Robbins's own analogy, one could remark that for going from A to B fast, there could now exist alternatives to running that are orders of magnitude more effective. Robbins flatly refuses to honor any alternative to time-honored brain usage with the name of "doing mathematics," thus, exorcizing the danger of radical novelty by the simple device of adjusting his definitions to his needs. Simply, by definition, mathematics will continue to be what it used to be. Enough said about the mathematicians.

Let me give you just one more example of the widespread disbelief in the existence of radical novelties and, hence, in the need to learn how to cope with them. It is the prevailing educational practice for which gradual, almost imperceptible, change seems to be the exclusive paradigm. How many educational texts are not recommended for their appeal to the student's intuition! They constantly try to present everything that could be an exciting novelty as something as familiar as possible. They consciously try to link the new material to what is supposed to be the student's familiar world.

It already starts with the teaching of arithmetic. Instead of teaching $2 + 3 = 5$, the hideous arithmetic operator "plus" is carefully disguised by calling it "and," and the little kids are given lots of familiar example, first, with clearly visible objects such as apples and pears, which are *in*, in contrast to equally countable objects such as percentages and electrons, which are *out*. The same, silly tradition is reflected at the university level in different introductory calculus courses for the future physicist, architect, or business major, each course adorned with examples from the respective fields. The educational dogma seems to be that everything is fine as long as the student does not notice that he is learning something really new; more often than not, the student's impression is indeed correct.

I consider the failure of an educational practice to prepare the next generation for the phenomenon of radical novelties a serious shortcoming. (When King Ferdinand visited the conservative University of Cervera, the Rector proudly reassured the monarch with the words: "Far be from us, Sire, the dangerous novelty of thinking." Spain's problems in the century that followed justify my characterization of the shortcoming as "serious.") This was to show the extent to which education has adopted the paradigm of gradual change.

The concept of radical novelties is of contemporary significance because, while we are ill-prepared to cope with them, science and technology have now shown themselves expert at inflicting them upon us. Earlier scientific examples are the theory of relativity and quantum mechanics; later technological examples are the atom bomb and the pill. For decades, the former two gave rise to a torrent of religious, philosophical, or, otherwise, quasi-scientific tracts. We can daily observe the profound inadequacy with which the latter two are approached, be it by our statesmen and religious leaders or by the public at large. This illustrates the damage done to our peace of mind by radical novelties.

I raised all this because of my contention that automatic computers represent a radical novelty, and that only by identifying them as such can we identify all the nonsense, the misconception, and the mythology that surround them. Closer inspection will reveal that it is even worse, *viz.*, that automatic computers embody not only one radical novelty but two of them.

The first radical novelty is a direct consequence of the raw power of today's computing equipment. We all know how we cope with something big and complex: divide and rule, i.e., we view the whole as a compositum of parts and deal with the parts separately. And if a part is too big, we repeat the procedure. The town is made up from neighborhoods that are structured by streets, that contain buildings, that are made from walls and floors, that are built from bricks, etc., eventually down to the elementary particles. And we have all our specialists along the line, from the town planner via the architect to the solid state physicist, and further. Because, in a sense, the whole is "bigger" than its parts, the depth of a hierarchical decomposition is some sort of logarithm of the ratio of the "sizes" of the whole and the ultimate smallest parts. From a bit to a few hundred megabytes, from a microsecond to half an hour of computing, it confronts us with the completely baffling ratio of 10^9!

The programmer is in the unique position that his is the only discipline and profession in which such a gigantic ratio, which totally baffles our imagination, has to be bridged by a single technology. He has to be able to think in terms of conceptual hierarchies that are much deeper than a single mind ever needed to face before. Compared to that number of semantic levels, the average mathematical theory is almost flat. By evoking the need for deep conceptual hierarchies, the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history.

Again, I have to stress this radical novelty because the true believer in gradual change and incremental improvements is unable to see it. For him, an automatic computer is something like the familiar cash register, only somewhat bigger, faster, and more flexible. But the analogy is ridiculously shallow. It is orders of magnitude worse than comparing, as a means of transportation, the supersonic jet plane with a crawling baby, for that speed ratio is only a thousand.

The second radical novelty is that the automatic computer is our first large-scale digital device. We had a few with a noticeable discrete component. I just mentioned the cash register and can add the typewriter, with its individual keys. With a single stroke you can type either a Q or a W but, though their keys are next to each other, not a mixture of those two letters. But such mechanisms are the exception, and the vast majority of our mechanisms are viewed as analogue devices whose behavior is, over a large range, a continuous function of all parameters involved. If we press the point of the pencil a little bit harder, we get a slightly thicker line; if the violinist slightly misplaces his finger, he plays slightly out of tune. To this I should add that, to the extent that we view ourselves as mechanisms, we view ourselves, primarily, as analogue devices. If we push a little harder we expect to do a little better. Very often, the behavior is not only a continuous but even a monotonic function. To test whether a hammer suits us over a certain range of nails, we try it out on the smallest and largest nails of the range, and if the outcomes of those two experiments are positive, we are perfectly willing to believe that the hammer will suit us for all nails in between.

It is possible, and even tempting, to view a program as an abstract mechanism, as a device of some sort. To do so, however, is highly dangerous. The analogy is too shallow because a program is, as a mechanism, totally different from all the familiar analogue devices we grew up with. Like all digitally encoded information, it has, unavoidably, the uncomfortable property that the smallest possible perturbations—i.e., changes of a single bit—can have the most drastic consequences. (For the sake of completeness, I add that the picture is not essentially changed by the introduction of redundancy or error correction.) In the discrete world of computing, there is no meaningful metric in which "small" changes and "small" effects go hand in hand, and there never will be.

This second radical novelty shares the usual fate of all radical novelties: it is denied because its truth would be too discomforting. It is hard to estimate the damage done by this denial, but with million programmers in the world, a cost of millions of dollars per day seems a very modest guess.

Having described—admittedly in the broadest possible terms—the nature of computing's novelties, I shall now provide the evidence that these novelties are, indeed, radical. I shall do so by explaining a number of otherwise strange phenomena as frantic—but, as we now know, doomed—efforts at hiding or denying the frighteningly unfamiliar.

A number of these phenomena have been bundled under the name "Software Engineering." As economics is known as "The Miserable Science," software engineering should be known as "The Doomed Discipline": doomed because it cannot even approach its goal since its goal is self-contradictory. Software engineering, of course, presents itself as another worthy cause, but that is eyewash. If you carefully read its literature and analyze what its devotees actually do, you will discover that software engineering has accepted as its charter, "How to program if you cannot."

The popularity of its name is enough to make it suspect. In what we denote as "primitive societies," the superstition that knowing someone's true name gives you magic power over him is not unusual. We are hardly less primitive. Why do we persist here in answering the telephone with the most unhelpful "hello" instead of our name? Nor are we above the equally primitive superstition that we can gain some control over some unknown, malicious demon by calling it by a safe, familiar, and innocent name, such as "engineering." But it is totally symbolic, as one of the U.S. computer manufacturers proved a few years ago when it hired, one night, hundreds of new "software engineers" via the simple device of elevating all its programmers to that exalted rank. So much for that term.

The practice is pervaded by the reassuring illusion that programs are just devices like any others, the only difference admitted being that their manufacture might require a new type of craftsmen, *viz.*, programmers. From there, it is only a small step to measuring "programmer productivity" in terms of "number of lines of code produced per month." This is a very costly measuring unit because it encourages the writing of insipid code, but, today, I am less interested in how foolish a unit it is from even a pure business point of view. My point, today, is that if we wish to count lines of code, we should not regard them as "lines produced" but as "lines spent." The current conventional wisdom is so foolish as to book that count on the wrong side of the ledger.

Besides the notion of productivity, quality control continues to be distorted by the reassuring illusion that what works with other devices works with programs as well. It is now two decades since it was pointed out that program testing may convincingly demonstrate the presence of bugs but can never demonstrate their absence. After quoting this well-publicized remark devoutly, the software engineer returns to the order of the day and continues to refine his testing strategies, just like the alchemist of yore who continued to refine his chrysocosmic purifications.

Unfathomed misunderstanding is further revealed by the term "software maintenance," of which many people continue to believe that programs—and even programming languages themselves—are subject to wear and tear. Your car needs maintenance too, does it not? Famous is the story of the oil company that believed that its Pascal programs did not last as long as its Fortran programs "because Pascal was not maintained."

In the same vein, I must draw attention to the astonishing readiness with which the suggestion has been accepted that the pains of software production are largely due to a lack of appropriate "programming tools." (The telling "programmer's workbench" was soon to follow.) Again, the shallowness of the underlying analogy is worthy of the Middle Ages. Confrontations with insipid "tools" of the "algorithm-animation" variety has not mellowed my judgement; on the contrary, it has confirmed my initial suspicion that we are primarily dealing with yet another dimension of the snake-oil business.

Finally, to correct the possible impression that the inability to face radical novelty is confined to the industrial world, let me offer you an explanation of the continuing popularity of artificial intelligence. You would expect people to feel threatened by the "giant brains or machines that think." In fact, the frightening computer becomes less frightening if it is used only to simulate a familiar noncomputer. I am sure that this explanation will remain controversial for quite some time for artificial intelligence, as a mimicking of the human mind, prefers to view itself as being at the front line, whereas my explanation relegates it to the rearguard. (The effort of using machines to mimic the human mind has always struck me as rather silly. I would rather use them to mimic something better.)

So much for the evidence that the computer's novelties are, indeed, radical.

And now comes the second—and hardest—part of my talk: the scientific and educational consequences of the above. The educational consequences are, of course, the hairier ones, so let us postpone their discussion and stay for a while with computing science itself. What is computing? And what is a science of computing about?

Well, when all is said and done, the only thing computers can do for us is to manipulate symbols and produce results of such manipulations. From our previous observations, we should recall that this is a discrete world and, moreover, that both the number of symbols involved and the amount of manipulation performed is many orders of magnitude larger than we can envisage. They totally baffle our imagination, and we must, therefore, not try to imagine them. But before a computer is ready to perform a class of meaningful manipulations—or calculations, if you prefer—we must write a program.

By evoking the need for deep conceptual hierarchies, the automatic computer confronts us with a radically new intellectual challenge that has no precedent in our history.

What is a program? Several answers are possible. We can view the program as what turns the general-purpose computer into a special-purpose symbol manipulator, and it does so without the need to change a single wire. (This was an enormous improvement over machines with problem-dependent wiring panels.) I prefer to describe it the other way round. The program is an abstract symbol manipulator which can be turned into a concrete one by supplying a computer to it. After all, it is no longer the purpose of programs to instruct our machines; these days, it is the purpose of machines to execute our programs.

So, we have to design abstract symbol manipulators. We all know what they look like. They look like programs or—to use somewhat more general terminology—usually rather elaborate formulae from some formal system. It really helps to view a program as a formula. First, it puts the programmer's task in the proper perspective: he has to derive that formula. Second, it explains why the world of mathematics all but ignored the programming challenge: programs were so much longer formulae than it was used to that it did not even recognize them as such. Now back to the programmer's job. He has to derive that formula; he has to derive that program. We know of only one reliable way of doing that, *viz.*, by means of symbol manipulation. And now the circle is closed. We construct our mechanical symbol manipulators by means of human symbol manipulation.

Hence, computing science is—and will always be—concerned with the interplay between mechanized and human symbol manipulation usually referred to as "computing" and "programming," respectively. An immediate benefit of this insight is that it reveals "automatic programming" as a contradiction in terms. A further benefit is that it gives us a clear indication where to locate computing science on the world map of intellectual disciplines: in the direction of formal mathematics and applied logic, but, ultimately, far beyond where those are now for computing science is interested in *effective* use of formal methods and on a much, much larger scale than we have witnessed so far. Because, these days, no computing endeavor is respectable without an acronym, I propose that we adopt for computing science VLSAL (Very Large Scale Application of Logic), and, to be on the safe side, we had better follow the shining examples of our leaders and make a trademark of it.

In the long run, I expect computing science to transcend its parent disciplines, mathematics and logic, by effectively realizing a significant part of Leibniz's Dream of providing symbolic calculation as an alternative to human reasoning. (Please note the difference between "mimicking" and "providing an alternative to." Alternatives are allowed to be better.)

Needless to say, this vision of what computing science is about is not universally applauded. On the contrary, it has met widespread—and sometimes even violent—opposition from all sorts of directions. I mention as examples

0. the mathematical guild, which would rather continue to believe that the Dream of Leibniz is an unrealistic illusion
1. the business community, which, having been sold the idea that computers would make life easier, is mentally unprepared to accept that they only solve the easier problems at the price of creating much harder ones
2. the subculture of the compulsive programmer, whose ethics prescribe that one silly idea and a month of frantic coding should suffice to make him a life-long millionaire
3. computer engineering, which would rather continue to act as if it is all only a matter of higher bit rates and more flops per second
4. the military, which is now totally absorbed in the business of using computers to mutate billion-dollar budgets into the illusion of automatic safety
5. all soft sciences for which computing now acts as some sort of interdisciplinary haven
6. the educational business that feels that if it has to teach formal mathematics to CS students, it may as well close its schools.

And, with this last example, I have reached, imperceptibly but alas unavoidably, the most hairy part of this talk: educational consequences.

The problem with educational policy is that it is hardly influenced by scientific considerations derived from the topics taught and is almost entirely determined by extra-scientific circumstances such as the combined expectations of the students, their parents, and their future employers, and the prevailing view of the role of the university is the stress on training its graduates for today's entry-level jobs or on providing its alumni with the intellectual baggage and attitudes that will last them another fifty years? Do we grudgingly grant the abstract sciences only a far-away corner on campus, or do we recognize them as the indispensable motor of the high-technology industry? Even if we do the latter, do we recognize a high-technology industry as such if its technology primarily belongs to formal mathematics? Do the universities provide for society the intellectual leadership it needs or only the training it asks for?

Traditional academic rhetoric is perfectly willing to give to these questions the reassuring answers, but I do not believe them. By way of illustration of my doubts, in a recent article on "Who Rules Canada?," David H. Flaherty bluntly states:

> "Moreover, the business elite dismisses traditional academics and intellectuals as largely irrelevant and powerless."

So, if I look into my foggy crystal ball at the future of computing science education, I overwhelmingly see the depressing picture of "business as usual." The universities will continue to lack the courage to teach hard science; they will continue to misguide the students, and each next stage of infantilization of the curriculum will be hailed as educational progress.

I now have had my foggy crystal ball for quite a long time. Its predictions are invariably gloomy and usually correct. However, I am quite used to that, and they will not keep me from giving you a few suggestions, even if it is merely an exercise in futility whose only effect is to make you feel guilty.

We could, for instance, begin with cleaning up our language by no longer calling a bug "a bug" but by calling it an error. It is much more honest because it squarely puts the blame where it belongs, *viz.*, with the programmer who made the error. The animistic metaphor of the bug that maliciously sneaked in while the programmer was not looking is intellectually dishonest as it is a disguise that the error is the programmer's own creation. The nice thing of this simple change of vocabulary is that it has such a profound effect. While, before, a program with only one bug used to be "almost correct," afterwards a program with an error is just "wrong" (because in error).

My next linguistical suggestion is more rigorous. It is to fight the "if-this-guy-wants-to-talk-to-that-guy" syndrome. *Never* refer to parts of programs or pieces of equipment in an anthropomorphic terminology, nor allow your students to do so. This linguistical improvement is much harder to implement than you might think, and your department might consider the introduction of fines for violations, say a quarter for undergraduates, two quarters for graduate students, and five dollars for faculty members; by the end of the first semester of the new regime, you will have collected enough money for two scholarships.

The reason for this last suggestion is that the anthropomorphic metaphor—for whose introduction we can blame John von Neumann—is an enormous handicap for every computing community that has adopted it. I have now encountered programs wanting things, knowing things, expecting things, believing things, etc., and each time that gave rise to avoidable confusions. The analogy that underlies this personification is so shallow that it is not only misleading but also paralyzing.

It is misleading in the sense that it suggests that we can adequately cope with the unfamiliar discrete in terms of the familiar continuous, i.e., ourselves, quod non. It is paralyzing in the sense that because persons exist and act *in time*, its adoption effectively prevents a departure from operational semantics and, thus, forces people to think about programs in terms of computational behaviors, based on an underlying computational model. This is bad because operational reasoning is a tremendous waste of mental effort.

When all is said and done, the only thing computers can do for us is to manipulate symbols and produce results of such manipulations.

Let me explain to you the nature of that tremendous waste, and allow me to try to convince you that the term "tremendous waste of mental effort" is *not* an exaggeration. For a short while I shall get highly technical, but do not get frightened. It is the type of mathematics that one can do with one's hands in one's pockets. The point to get across is that if we have to demonstrate something about *all* the elements of a large set, it is hopelessly inefficient to deal with all the elements of the set individually. The efficient argument does not refer to individual elements at all and is carried out in terms of the set's definition.

Consider the plane figure Q, defined as the 8 by 8 square from which, at two opposite corners, two 1 by 1 squares have been removed. The area of Q is 62, which equals the combined area of 31 dominos of 1 by 2. The theorem is that the figure Q cannot be covered by 31 such dominos.

Another way of stating the theorem is that if you start with squared paper and begin covering this by placing each next domino on two new adjacent squares, no placement of 31 dominos will yield the figure Q.

So, a possible way of proving the theorem is by generating all possible placements of dominos and verifying for each placement that it does not yield the figure Q: a tremendously laborious job.

The simple argument, however, is as follows. Color the squares of the squared paper as on a chess board. Each domino, covering two adjacent squares, covers 1 white and 1 black square, and, hence, each placement covers as many white squares as it covers black squares. In the figure Q, however, the number of white squares and the number of black squares differ by 2—opposite corners lying on the same diagonal—and, hence, no placement of dominos yields figure Q.

Not only is the above simple argument many orders of magnitude shorter than the exhaustive investigation of the possible placements of 31 dominos, it is also essentially more powerful for it covers the generalization of Q by replacing the original 8 by 8 square with *any* rectangle with sides of even length. The number of such rectangles being infinite, the former method of exhaustive exploration is essentially inadequate for proving our generalized theorem.

And this concludes my example. It has been presented because it illustrates, in a nutshell, the power of down-to-earth mathematics; needless to say, refusal to exploit this power of down-to-earth mathematics amounts to intellectual and technological suicide. The moral of the story is: deal with all elements of a set by ignoring them and working with the set's definition.

Back to programming, the statement that a given program meets a certain specification amounts to a statement about *all* computations that could take place under control of that given program. And since this set of computations is defined by the given program, our recent moral says: deal with all computations possible under control of a given program by ignoring them and working with the program. We must learn to work with program texts while (temporarily) ignoring that they admit the interpretation of executable code.

Another way of saying the same thing is the following one. A programming language, with its formal syntax and with the proof rules that define its semantics, is a formal system for which program execution provides only a model. It is well-known that formal systems should be dealt with in their own right and not in terms of a specific model. And, again, the corollary is that we should reason about programs without even mentioning their possible "behaviors."

And this concludes my technical excursion into the reason why operational reasoning about programming is "a tremendous waste of mental effort" and why, therefore, in computing science the anthropomorphic metaphor should be banned. Not everybody understands this sufficiently well.

I was recently exposed to a demonstration of what was pretended to be educational software for an introductory programming course. With its "visualizations" on the screen, it was such an obvious case of curriculum infantilization that its author should be cited for "contempt of the student body," but this was only a minor offense compared with what the visualizations were used for. They were used to display all sorts of features of computations evolving under control of the student's program! The system highlighted precisely what the student has to learn to ignore; it reinforced precisely what the student has to unlearn. Since breaking out of bad habits, rather than acquiring new ones, is the toughest part of learning, we must expect from that system permanent mental damage for most students exposed to it.

Needless to say, that system completely hid the fact that, all by itself, a program is no more than half a conjecture. The other half of the conjecture is the functional specification the program is supposed to satisfy. The programmer's task is to present such complete conjectures as proven theorems.

Before we part, I would like to invite you to consider the following way of doing justice to computing's radical novelty in an introductory programming course.

On the one hand, we teach what looks like the predicate calculus, but we do it very differently from the philosophers. In order to train the novice programmer in the manipulation of uninterpreted formulae, we teach it more as boolean algebra, familiarizing the student with all algebraic properties of the logical connectives. To further sever the links to intuition, we rename the values {true, false} of the boolean domain as {black, white}.

On the other hand, we teach a simple, clean, imperative programming language, with a skip and a multiple assignment as basic statements, with a block structure for local variables, the semicolon as operator for statement composition, a nice alternative construct, a nice repetition, and, if so desired, a procedure call. To this, we add a minimum of data types, say booleans, integers, characters, and strings. The essential thing is that for whatever we introduce, the corresponding semantics are defined by the proof rules that go with it.

If I look into my foggy crystal ball at the future of computing science education, I overwhelmingly see the depressing picture of "business as usual."

Right from the beginning and all through the course, we stress that the programmer's task is not just to write down a program, but that his main task is to give a formal proof that the program he proposes meets the equally formal functional specification. While designing proofs and programs hand in hand, the student gets ample opportunity to perfect his manipulative agility with the predicate calculus. Finally, in order to drive home the message that this introductory programming course is primarily a course in formal mathematics, we see to it that the programming language in question has *not* been implemented on campus so that students are protected from the temptation to test their programs. And this concludes the sketch of my proposal for an introductory programming course for freshmen.

This is a serious proposal and utterly sensible. Its only disadvantage is that it is too radical for many, who, being unable to accept it, are forced to invent a quick reason for dismissing it, no matter how invalid. I will give you a few quick reasons.

You do not need to take my proposal seriously because it is so ridiculous that I am obviously completely out of touch with the real world. But that kite will not fly for I know the real world only too well. The problems of the real world are primarily those you are left with when you refuse to apply their effective solutions. So, let us try again.

You do not need to take my proposal seriously because it is utterly unrealistic to try to teach such material to college freshmen. Would not that be an easy way out? You just postulate that this would be far too difficult. But that kite will not fly either for the postulate has been proven wrong. Since the early 80s, such an introductory programming course has successfully been given to hundreds of college freshmen each year. (Because, in my experience, saying this once does not suffice, the previous sentence should be repeated at least another two times.) So, let us try again.

Reluctantly admitting that it could, perhaps, be taught to sufficiently docile students, yet you reject my proposal because such a course would deviate so much from what 18-year-old students are used to and expect that inflicting it upon them would be an act of educational irresponsibility. It would only frustrate the students. Needless to say, that kite will not fly either. It is true that the student that has never manipulated uninterpreted formulae quickly realizes that he is confronted with something totally unlike anything he has ever seen before. But fortunately, the rules of manipulation are, in this case, so few and simple that very soon, thereafter, he makes the exciting discovery that he is beginning to master the use of a tool that, in all its simplicity, gives him a power that far surpasses his wildest dreams.

Teaching to unsuspecting youngsters the effective use of formal methods is one of the joys of life because it is so extremely rewarding. Within a few months, they find their way in a new world with a justified degree of confidence that is radically novel for them; within a few months, their concept of intellectual culture has acquired a radically novel dimension. To my taste and style that is what education is about. Universities should not be afraid of teaching radical novelties; on the contrary, it is their calling to welcome the opportunity to do so. Their willingness to do so is our main safeguard against dictatorships, be they of the proletariat, of the scientific establishment, or of the corporate elite.

Edsger W. Dijkstra
Dept. of Computer Sciences
The University of Texas
Austin, TX 78712-1188

Colleagues Respond to Dijkstra's Comments

There is much in Dijkstra's statement with which I can agree. Since agreement results in dull commentary, I shall focus on his vigorous dismissal of the term "software engineering" and his denunciation of software testing.

Most introductory engineering texts define an engineer as one who uses science and mathematics to produce useful products. That definition clearly allows programmers to be considered engineers, but professional societies and/or government licensing agencies add educational requirements and may require an examination before one is permitted to use the title "Professional Engineer." In fact, there is much better control of the word "engineer" than of the word "mathematician." Just as some physicists claim to be mathematicians, other people use the word "engineer" quite loosely. The fact that some people write poor papers and textbooks under the title "software engineering" does not justify abandonment of the hope that, some day, programs will be produced by properly educated professional engineers.

Those with training in mathematics or one of the physical sciences often have the impression that engineering involves the use of sloppy, heuristic methods and undefined notations. Nothing could be further from the truth! Good engineering programs emphasize the use of formal methods in exactly the way suggested by Dijkstra.

My own engineering education included more courses in the Mathematics Department (taken side-by-side with mathematics students) than courses offered by the Electrical Engineering Department. Further, each of the engineering courses emphasised the use of mathematical methods for design and design verification. In exams and homework, we were often deliberately given the opportunity to apply formulae in situations where they were not applicable; in this way, we were taught to appreciate the importance of understanding the derivation of all equations and being certain that the assumptions made in the derivation were valid for the case at hand. We were taught to understand mathematics, its development, and its use in solving engineering problems. Great emphasis was placed on the use of mathematics to define precisely what we were trying to achieve with a particular design and its further use to confirm that our design met those requirements.

Our engineering instructors repeatedly showed us the folly of using anthropomorphic analogies. No engineer talks about "how much the electrons want to get to the other side of a capacitor." Instead, he uses mathematical models to determine the intensity of field at all points between the capacitor's plates, to look for points of maximum field intensity, etc.

Those who graduated from such programs were solidly grounded in the use of formal methods. They would wonder why comments such as Dijkstra's need to be made. They need to be made because the majority of those in our profession do not have the benefit of an engineering education.

I am just as critical of much of the work done at software engineering institutes and presented at software engineering conferences as Dijkstra. Most of us who use the term "software engineering" do so, not because we think that programming is currently an engineering profession, but because we think it should become one. The ability to use formal methods to derive and validate designs is one of the essential characteristics of an engineer. However, an engineering education also teaches students to understand the limits of mathematical methods. We were taught methods of dealing with problems where exact mathematical models were intractable. More important, we were taught the importance of testing to (a) provide a check on our mathematical analysis and (b) verify that our mathematical model was an adequate model of the actual devices with which we worked. Anyone can err when using mathematics, and it is not uncommon to find that devices have characteristics that were not reflected in our mathematical models.

The situation is no different in software engineering. Errors in formal proofs are not uncommon. Further, the computers and programs that we must use may not have the characteristics implied by the mathematical axioms that we use. Testing can reveal both of those problems. While careful review can reveal mathematical errors, only testing can reveal the failure of a component, software or hardware, to conform to a mathematical model.

It is certainly true that, in most practical circumstances, testing cannot show the absence of errors; however, properly designed statistical testing can provide information about the probability of an error remaining in the code or the probability of a failure occurring in use. In the imperfect world in which we live such data can be very important.

There is no engineering profession in which testing and mathematical validation are viewed as alternatives. It is universally accepted that they are complementary and that both are required.

Dijkstra states that the "computer confronts us with a radical new intellectual challenge that has no precedent in our history." He bases this on two factors: (1) the sensitivity of program behavior to minor changes and (2) the drastic increase in the "size" of the problems. The first of these is not confined to digital technologies and it may be possible, in time, to overcome the second. The sensitivity to minor changes can also be present in older technologies where resonance can lead to very steep gradients when frequency is varied. Such resonances can cause unexpected structural collapse or component "burnout." Dijkstra has led the way in showing us how to master drastic increase in complexity by application of a divide-and-conquer approach. That approach has been used with great success in other engineering disciplines.

Those who are participating in a major upheaval in society rarely have the perspective to judge its historical importance. Only the passage of time will reveal whether we need a radical new way of thinking or just a return to the more careful and disciplined approaches used by good engineers in the past.

David L. Parnas
Queen's University
Kingston, Ontario
K7L 3N6 Canada

Dijkstra recommends two principal actions concerning computer science education which are (1) that imperative and anthropomorphic thinking be eschewed by all programmers and (2) that formal methods, specifically those based on imperative programming languages with semantics grounded in first-order predicate logic, be the basis for introductory programming courses. To motivate these recommendations, he makes several observations about the nature of computing and software which I summarize roughly as follows:

1. computing is a "radical novelty" due to the immense scaling range it encompasses and due to its discontinuous nature; radical novelties, like great tragedies, tend to be denied.
2. software engineering as a worthy cause is eyewash, and the very notion of a "programming tool" is corrupt.
3. computer science concerns the interplay of symbol manipulation activity by human and by machine.

The extensive and forceful arguments Dijkstra makes in support of these findings are very interesting and bear examination. I was surprised that after developing these arguments, the overall thrust of Dijkstra's conclusion was so benign:

> If we are to have a high level of confidence in the software systems we develop, then formal methods will have a central role in their development, and our teaching of programming should support this view from the start.

While I agree with the overall thrust of Dijkstra's conclusion, I do not fully concur with the particulars of Dijkstra's recommendation. There are, of course, nits to pick. For example, if scaling is one of the two "radical novelties" in computing, why propose teaching the use of a programming language that does not provide the principal linguistic means for scaling up—namely, composite data types, date abstraction mechanisms, and other means to explicitly represent system interfaces and abstraction boundaries. Edinburgh ML demonstrates how very simple these language structures can be. Also, if operational thinking is to be eschewed, why insist on an imperative programming language?

More significantly, I am troubled by the suggestion of the first recommendation that certain modes of thinking be avoided in problem solving. A successful problem solver will have a broad array of means at hand to tackle problems (many of which are enumerated by Polya in his books) together with the maturity to make choices of which means are appropriate to circumstances. For the programmer, one of the means available is the use of informal operational intuition. Of course, this does not excuse failures to think intensionally when appropriate (as in the case of the checkerboard example, which, indeed, is often used in AI

If we are to have a high level of confidence in the software systems we develop, then formal methods will have a central role in their development, and our teaching of programming should support this view from the start.

courses, too, for the same purpose). It also does not excuse failure to come to grips with underlying computational models, as in parallel systems. Even visualization has a role—mathematicians make use of visualization, for example, in the illustration of solutions for fluid flow equations.

But, let us set aside Dijkstra's recommendations for the moment and consider his motivating observations, an analysis of which suggests to me that bolder recommendations are called for.

Concerning the first observation: I agree with Dijkstra that, while large scaling ranges and discontinuous behavior do exist in other human-engineered systems, few have the uniformity of structure over a broad range of scale that exists in computing technology. I do take exception, however, to Dijkstra's implication that software systems are unremittingly discontinuous and should be managed as such. While this statement is obviously true in principle, designers of large systems (such as a workstation application environment) deliberately seek a kind of continuity property at the higher levels of scale. The idea is to lower risks for both customers and developers by making it possible to obtain incremental improvments in capability for incremental investments. This amounts to a notion of "coarse continuity," which has been well approximated in many clever software engineering designs (such as UNIX and GNU Emacs). Of course, the appearance of continuity at one level of abstraction may, in fact, be obtained only through gross redesign of a subcomponent. There is, nonetheless, a kind of continuity at the higher level of abstraction. "Software maintenance" is, thus, the (attempted) incremental adaptation of large systems in response to small changes in requirements for which this "coarse continuity" property holds.

Concerning Dijkstra's second observation: most large software problems involve complexity and detail that exceeds the capacity of one person to develop solutions.

In these cases, major decisions must be made under circumstances where all the details cannot be presented to all parties involved since there are too many details. The numbers of people involved and the lack of perfect knowledge create the software engineering challenges of estimation of costs and risks, the implementation of procedures to enable management of the process, the development of automation to assist, the need to design systems that can be readily adapted, and so on. Basic automation support can include object management, version and configuration management, process and administrative support, and consistency support for both formal and informal documentation. These needs are real, even if many products are indeed "snake oil."

Concerning the third observation: one of the greatest difficulties in software development is formalization—capturing in symbolic representation a worldly computational problem so that the statements obtained by following rules of symbol manipulation are useful statements once translated back into the language of the world. The formalization problem is the essence of requirements engineering, an area left largely untouched in Dijkstra's position statement. Concerning symbol manipulation, at the level both of specific problem domains and of programming itself, it is important to have fluidity in shifting various aspects of the symbol manipulation task between humans and machine and to have linguistic means to organize large problems to facilitate this.

More can be said about Dijkstra's observations and the arguments supporting them, but the above remarks are enough, I believe, to suggest a set of conclusions concerning formal methods that more effectively addresses Dijkstra's observations, particularly concerning the "radical novelties" of scaling and non-continuity.

My first conclusion is that we should distinguish the necessary qualities of the result of the programming process from the means by which the results are obtained. The software practitioner should be able to bring all intellectual and mechanical tools to bear on devising a software system as long as certain constraints concerning the structure and presentation of the result are satisfied. The "theorem" Dijkstra refers to is, thus, really part of a larger result, which is more likely a documentation record that links together implementation, specification, design and interface decisions, informal rationale, and formal proof, all in their several versions and configurations. The intent is that this documentation-record supports adaptation, reuse (of appropriate assets), and analysis by capturing information ordinarily lost (or, worse, never present) during conventional software development, including the means, formal and informal, by which confidence is raised concerning consistency of specification and implementation.

The second conclusion is that an appropriate means must be found to incorporate formal-methods techniques into software engineering practice and tools in order to provide a "scalable" approach to applying formal methods to larger systems. Specifications and documentation can involve formal and informal components. Formal methods can, thus, be used to establish certain key properties while informal (and less reliable) means are used to increase confidence with respect to other system properties. The effect is of proving small theorems about large systems rather than large theorems about (inevitably) small systems. This "scalable" approach has the advantage of drastically moderating adoption risks for software managers.

The final conclusion concerns education. In many curricula, as Dijkstra has noted, we are already teaching formal methods at the earliest levels. My experience is that the principal difficulties in teaching formal methods do not have to do with symbol manipulation skills, but rather, with formalization issues—representing symbolically and providing structure for actual computational problems from the world—and with metatheoretic issues—understanding, from example, what sorts of assurance can be provided by proofs in a given formal deductive system. It is even more challenging to create an appreciation for the extent of the scaling range of computing (as Dijkstra has noted) along with techniques for addressing scaling problems, particularly abstraction mechanisms for programs and specifications.

W.L. Scherlis
Carnegie Mellon University
Dept. of Computer Science
Pittsburgh, PA 15213–3890
scherlis@vax.darpa.mil

I agree that the proposed program derivation course is more valuable than the other courses in a typical computer science curriculum. In the following, I explain that such a course should not be taught purely as exercises in symbol manipulation and that the ability to derive a program from a formal derivation only solves a minor part of the problems we are having with software.

People who know neither programming nor mathematics (i.e., almost everyone) take for granted that programming is like mathematics. Yet, it turns out that English majors are as likely to be as successful at programming as mathematics graduates are. In practice, the worlds of mathematics and programming are just about disjoint. Dijkstra argues that this should not be the case, that mathematics is a formal game of symbol manipulation and that programming should become one. I must confess that the role of symbol manipulation in mathematics has me thoroughly confused.

On the one hand, it is clear that symbolism is extremely powerful in a positive way. To see that, try to do something simple, say long division, in English. The most powerful, single experience I had in my university education was getting a grip on ghostly things like gravitational or electromagnetic fields with formulas using div, grad, curl, and, possibly, other operators long forgotten.

On the other hand, what makes all this so appealing is that there is an intuition behind it. Axiomatization and formalization, by themselves, do not make a theory into mathematics; they only mark its maturity. To be mathematics, the theory has to be significant, to appeal to the intuition. The experts do not agree: one mathematician may denounce another's paper as empty formalism, devoid of mathematical content. This sounds fuzzy, and it is.

Hilbert is famous for viewing mathematics as a game with symbols, devoid of meaning. What he must have meant (and probably also said) is that the meaning is irrelevant to the validity of a formal argument. But to make the formal game worth playing, it better have meaning. Even games have meaning, at least the ones, like chess, that get played. It is easy to make up a system that formally is a two-person game. It is hard to make it such that people will want to play it. Successful games have a meaning to their players.

Why am I saying all this? I think the course described by Dijkstra, where students derive programs from logic specifications into an unimplemented language, is a great idea. However, it will only work if the symbols manipulated have meaning for the players. And I do not see where that meaning can come from other than having messed around with programs that run or fail to. But, perhaps that should be another, earlier course so that the program derivation course can be kept pure.

I agree with Dijkstra that great improvements can be made by using a formal specification and by proving that the program conforms to it. Should such proofs be formal, i.e., valid deductions in a sound formal system? I suggest not to wait till this is technically possible for interesting programs. The analogy with mathematics is valid: theorems worth believing are not believed because of formal proofs which probably do not even exist. Because this is the current state of affairs, it does not follow that things will always be this way. It may be that the enterprise started with Automath ultimately continues to success. If so, formal derivations of interesting programs will also be possible. In this way, programming may become like mathematics, even though it is different from both programming and from mathematics in their present form.

Suppose we reach the stage of always formally specifying our programs and proving them correct. Then we will only have eliminated the minor part of the problems we are having with software. The big problem is that it is hard to say precisely what we want. It is hard to do so in C; it is not as hard to do so in predicate calculus, but it is still hard.

The reason why it is so hard to say what we want is that we do not really know what we want (at least not in a complex system, which is where the problems are). One way of discovering what we want is to draw up a formal specification, derive a program conforming to it, and run it. Often, we will discover that the specification is wrong, or, rather, we discover that we want something different than what we thought we did. I would like to teach that also to the students. I would also like to teach students that it is necessary to run the programs, not to debug the programs, but to debug the specifications. But, perhaps, that should be another, later course so that the program derivation course can be kept pure.

M. H. van Emden
Dept. of Computer Science
University of Victoria
Victoria, B.C.
V8W 2Y2 Canada

The early papers of Dijkstra are gems polished by the hand of an expert lapidary. I first became acquainted with his work in the early 1960s. He had then published a paper on a stack machine model for the interpretation of Algol 60: a landmark in the design and implementation of block-structured languages. Following that work, there were two major contributions by Dijkstra which appeared in *Communications*. A short article ("Go-To's Considered Harmful") gave us ample food for thought about writing readable programs. The third, a concise article on synchronization of parallel processes, introduced the concept of semaphores and paved the way for the current work on programming parallel computers.

These early articles and papers have made an implicit revolution in computer science. The statement appearing in this issue explicitly advocates a revolution and, in my view, presents a somber, often sarcastic view of our field; it offers disappointingly few constructive suggestions.

First, let me attempt to summarize Dijkstra's statement to bring out its true purpose (without the embellishments and inconsistencies of the author). The scientific community is unprepared to understand the "radical novelty" brought up by computers because of (1) the sheer speed of these machines and their vast range of computing and storing power and (2) the fact that they are digital, which implies that a minuscule change in a program may elicit the most unexpected of responses. Remedy: (1) teach the new generation of scientists *one* way of dealing with this problem and (2) choose the simplest (imperative) language and use logic (predicate calculus) to ascertain that a program really works for all its intended data. Dijkstra presented this message in detail in the book *A Discipline of Programming* which provides convincing arguments for this approach.

My initial comments about the statement have to do with inconsistencies. For example, economics is referred to as "the miserable science"; yet, the author proceeds to guess the enormous daily economic costs of introducing redundancy or error correction into computer hardware and software. In another instance, he claims that the problems that computer scientists have to face make the application of "the average mathematical theory [look] almost flat." Yet, he later argues in favor of "the power of down-to-earth mathematics."

Unfortunately, the tone of the statement is uniformly negative. The quotations include a view from *one* mathematician about the limitations of the brain, a remark by the rector of a university reassuring the King of Spain of its faculty's conservativeness, the statement of a technocrat from an oil company comparing the virtues of Fortran and Pascal, and so forth. These quotations may render the statement colorful, but they do not provide evidence for its message. It is hard to imagine that these specific views are espoused by a majority. The same holds true when all practitioners of artificial intelligence and software engineering are lumped together as evil groups. There is no doubt in my mind that there is high-quality work being done in these areas! Hackers are also unfairly treated in Dijkstra's statement. In my view, we owe them a great deal for their creativity and for the interesting new paths they have suggested.

Dijkstra feels that analogue models (i.e., those with gradual changes) are inadequate to cope with the existence of digital machinery. In a way, he wishes that the scientific community and society would react digitally (i.e., by step functions) to face the problems posed by digital machines. As I see it, gradual changes are inherent to human nature and human societies, and even the revolutions we have had in history actually took place at a relatively gradual pace, the definition of "gradual" being necessarily vague (e.g., consider the current views about the French revolution.) Dijkstra's statement also conveys the message that the dictum "to err is human" is hardly acceptable.

The example on covering a given chessboard with dominos and the arguments used by Dijkstra in favor of simple proofs instead of exhaustive searches are to be taken with a grain of salt. There are many instances in which such searches are unavoidable and constitute the only hope of finding solutions to interesting problems. For example, the proof of the four-color conjecture using a large search space is definitely a valid approach, even if some mathematicians may yearn for a simpler proof (just in case it exists).

I agree with Dijkstra that computer programs should be considered as symbolic formulas. What is not yet established and is constantly evolving is the kind of formulas (i.e., the level of languages) which will allow us to perform convenient manipulations, minimize the possibilities of errors, and yield efficient code. As computers become more powerful, it is likely that very-high-level languages will play an important role in expressing programs as increasingly more concise formulas. This situation parallels the one which occurred 30 years ago when the first programming languages replaced assembly languages. After all, as believers in computer science, we should be among the first to exploit the advantages of computers.

This response gives me an opportunity to express my own views as an educator in computer science. Our goal should be to provide not *one* but several approaches in teaching our students how to reason about programs. Dijkstra's approach is one of them. Among the others are the functional and logic programming paradigms. They all share a common denominator: urge the student to have a healthy respect for the craft of programming. By offering a menu of approaches, the students will be better prepared to face whatever "radical novelties" may appear in the future. I also feel that learning the foundations for writing sound programs ought to be fun and there should be a genuine sense of accomplishment when a student actually runs a program, finds unexpected errors, and corrects them. This, of course, requires inspired, broad-minded teachers; forming them is a problem for any field of endeavor, not only computer science.

To summarize: on one hand, it is distressing to see the lack of realistic constructiveness in a statement by a pioneer in our field; on the other hand, the airing of the issues stemming from Dijkstra's statement should make computer science a better, more mature discipline.

Jacques Cohen
Zayre/Feldberg Professor
Department of Computer Science
Ford Hall
Brandeis University
Waltham, MA 02254

Perhaps it is best to begin by recalling that long ago Dijkstra led a crusade for the *total* abolition of the GOTO instruction. Currently, it is widely believed that the GOTO instruction is used much too often but that it also has its place in programming. The present statement by Dijkstra is another example of Moses laying down the law to us sinners in programming. As before it is both very right and very wrong.

Unfortunately, in his statement, there is much "sound and fury" and non sequiturs, the extensive use of color words, a gratuitous swipe at the military, and, often, little content beyond his assertions; you can detect the extent of this if you try to rewrite his statement, as I did, in simple language and clear reasoning.

Just as in the GOTO article, there is much truth in this statement. I agree wholeheartedly that we should replace the word "bug" with the word "error" and suppress all anthropomorphic words as being misleading to the beginners.

The major trouble is, I think, that Dijkstra believes that programming should resemble mathematics and believes that mathematics is what one is taught *à la* Euclid. One first lays down postulates, makes a few appropriate definitions, states theorems, and then proves them—after all, that is how mathematics is typically taught. He refuses to recognize that often the postulates follow from the theorems, as do many of the definitions, and often the theorems follow from the proofs we are able to generate—they are then called "proof driven theorems." Furthermore, Dijkstra, in his sober moments, well knows that human proofs in mathematics are unreliable, that many famous proofs have been repeatedly "patched up" by subsequent generations; hence, even if we tried to use his idea that programs should be "proved" by humans before they are run, the proofs are fallible, and, in any case, are merely paper problems run on a paper machine. It is this that I believe is *behind* much of the statement.

One of Dijkstra's major points is that the rapid growth of computers represents a unique change of magnitude so large that no one can comprehend it; but large changes are more common than he thinks, for example, the bandwidth available for signalling. The unique features he attributes to computer science occur in other fields of human activity:

1. particle accelerators have similarly increased in size and power consumption
2. the complexity of the telephone system of interconnected central offices was around long before the first of the electronic computers and is still probably more complex than any computer
3. the claim of unique vulnerability to a single error is certainly shared by our common languages.

Dijkstra excoriates "software engineering" by deliberately comparing it to his conception of mathematics rather than to, say, "effective writing" which I feel is a far better analogy. I also doubt that it is always wise to equate a program to a mathematical formula as he does.

Dijkstra uses the well-known example of trying to cover with dominos a checker board, without the diagonal corners, to illustrate the value of the mathematical, analytical approach and concludes immediately that *all* programs will similarly benefit—the reasoning is fallacious and, from him, shocking!

Dijkstra flatly asserts that he knows "reality" and his opponents do not, but I put about as much faith in this as in the statement, "I am Napoleon."

Dijkstra flatly asserts that he knows "reality" and his opponents do not, but I put about as much faith in this as in the statement, "I am Napoleon." Indeed, in my opinion he comes to grief simply because his "reality" is so far from most other people's, which he admits.

Dijkstra objects to measuring programming by lines of code, conveniently forgetting that authors are often paid by the word. In both cases, it is ridiculous at times to do so, but he offers no other *practical* measure of programming (or writing) effort.

Dijkstra willfully misunderstands "software maintenance," pretending that it means repairing parts that have failed rather than what everyone else understands, mainly, altering the current software to meet changing needs and environments; hence, contrary to his sneer, software that is not maintained is apt to be of much less value to the user than software that is.

Dijkstra seems to identify programming languages with "imperative languages" and deigns not to notice "object-oriented" and "functional" programming, to mention two other approaches—after all, he is Moses, and he knows what programming is.

Returning to the main theme of this reply, his desire to map software onto his conception of mathematics is foolish. His idea that a program be "proved" to be correct before running it applies, as noted before, to a paper program on a paper machine and not to reality. When you have to make a compiler for a new mathematically defined language, this approach is both very reasonable and valuable, but in many, if not most, engineering cases where the design criteria arise from what you are able to do this approach simply does not work very well (as can be seen from the government procurement policy in action). Even more than in mathematics, in engineering, there is a "give and take" between the design proposal and what can be done in current practice; hence, his desire to start with an exact description for the program that is to be written works mainly in his "reality."

Anyone who reads the above objections to mean that what Dijkstra writes can be safely ignored is a fool. I have documented some of the errors that his extremism has produced, but there is much truth in his statement. Apparently, reformers must often be extreme in what they say and do if they are to achieve a reformation. Read with charity, Dijkstra's statement is a valuable contribution; Moses has indeed led us a bit further out of our wilderness.

R. W. Hamming
Naval Postgraduate School
Monterey, CA 93943

In the space available it would be impossible to comment on all the provocative suggestions in Dijkstra's statement. Instead, I shall confine myself to challenging two of his basic premises: that computing science can be equated with the very large scale application of logic and that the programmer's main task should be to give a formal proof that the program he proposes meets the equally formal functional specification. I will also take issue with his proposal that the introductory programming course should primarily be a course in formal mathematics and that students should be protected from the temptation to test their programs. I believe that the wide acceptance of these ideas would be damaging to the field of computer science.

Dijkstra's dangerous recommendations stem from a misunderstanding of the role of formal logic in mathematical reasoning. It is, indeed, a fundamental insight that, in principle, a mathematical proof can be reduced to the manipulation of uninterpreted strings of symbols according to formal syntactic rules. But, as important as this insight may be for the foundations of mathematics, it has little bearing on the way mathematical truths are discovered or communicated in practice.

The discovery of mathematical truth is invariably an unsystematic, trial-and-error process that leans on models, pictures, analogies, examples, counterexamples, and intuitions about time, space, and number that have little relation to the rearrangement of symbols according to formal rules. Once a mathematical truth is discovered, the "proofs" used to communicate it from one human to another are never formalized completely, if only because such formalization is a hideously tedious process and because the resulting formal proofs would be opaque to human readers. The proofs that are communicated between humans are best viewed as persuasive arguments which, although not entirely complete or airtight, are often remarkably effective in transmitting mathematical insights from the mind of their discoverer to the minds of their readers.

One might argue that inasmuch as computer programs are nothing more than formal expressions and their execution is nothing more than the application of formal transformation rules, computer scientists must inevitably work in the arena of formal methods. Indeed, it seems to me that formal proof can play an indispensable role in certain special situations such as the validation of short, logically intricate programs involving the coordinated action of several processors. But I am convinced that, for the great majority of programming problems, the modalities of thought and expression that are most useful are very similar to those that prevail in other areas of mathematical work. The mass of tedious detail that is involved in a formal correctness proof of even a moderately complex algorithm is beyond the limits of human toleration; and, alas, at the present state of the art of automatic theorem proving, the prospects of obtaining correctness proofs automatically or semi-automatically are rather bleak.

Of course, it is possible to reject formal correctness proofs and still embrace the use of formal program specifications. However, a major impediment to the use of formal specifications is the inevitable intertwining of software development with specification. This point has been well made by two of my colleagues who have extensive experience in the design of large software systems:

"I do not view the process of programming, especially programming in the large, as that of discovering an algorithm to a prespecified and unalterable logic. It is more often a process of discovery and adaptation, as various relevant components of a problem are examined in turn, and a goal, initially specified incompletely, is reached." [Richard Fateman]

"In my view, large computer programs tend to be inherently unspecifiable, in the sense that there is not even a clear notion of how the programs "should" behave. In such systems, many of the most subtle bugs occur because our view of how the system "should" behave turned out to be wrong (we've spent a lot of time over the last couple of years fixing bugs like this in Sprite). In such an environment, it seems to me that formal logic is unlikely to root out the problems ... Dijkstra would probably claim that we have no business building a system until we understand exactly how it should behave, but I think this is impossible in any new engineering domain." [John Ousterhout]

I am led to the conclusion that formal proof and formal specification are not among the most promising avenues toward getting useful and dependable results from computers. Instead, I would advocate increased research on debugging methods, on the use of modern programming environments including work stations, revision control aids, editors, and high-level programming languages, and on the development of special-purpose software packages, such as database systems and spreadsheet programs, that enable even non-programmers to use computers productively.

It would be ironic if Dijkstra's prescriptions for introductory computer science education were to gain acceptance and the next generation of students were to be denied the thrill of seeing their programs come to life on the CRT screen. The result would be a fiasco unmatched since the introduction of the New Math.

Computers are becoming indispensable in nearly all fields of endeavor. For example, despite Dijkstra's claims to the contrary, they play an increasingly important role in mathematical research, where they are used to simplify algebraic expressions, to explore examples, and to generate graphical representations of complex geometric objects. In view of this trend, it would be ironic if Dijkstra's prescriptions for introductory computer science education were to gain acceptance and the next generation of students were to be denied the thrill of seeing their programs come to life on the CRT screen. The result would be a fiasco unmatched since the introduction of the New Math.

Nevertheless, many of the concepts behind Dijkstra's approach, if leavened with a bit of common sense, are well worth conveying. It is important for our students to gain facility with the elegant notation of set theory and the predicate calculus since these notations are useful in informal proof as well as formal proof. Our students should certainly be taught to give reasonably precise informal specifications of what their programs are intended to do and what their data structures are intended to represent before plunging into writing code. And the discipline of maintaining loop invariants and satisfying preconditions and postconditions for subroutines (even if these conditions are not stated in a formal language) is highly valuable to any programmer. But, it is even more important that the next generation of computer science students be exposed, as early as possible, to the practical art of constructing large programs, with its intertwined processes of specification, programming, testing, and documentation.

Richard M. Karp
Dept. of Computer Science
571 Evans Hall
Univ. of California
Berkeley, CA 94720

Behind Dijkstra's bravado and invective there lurks a coherent argument about the nature of computer science education. Coherent and interesting, but wrong, because it is based on faulty premises. To start out with, Dijkstra is wrong about what computers do, wrong about what programmers do, and wrong about what engineers do.

The problem begins with his definition of computers: "When all is said and done, the only thing computers can do for us is to manipulate symbols and produce results of such manipulations." In my vicinity of the "real world," I see computers doing lots of other things. They issue payroll checks, control the motions of metalworking machines, format and print architectural drawings and newsletters, and keep my car's brakes from locking. In doing all this, they may manipulate symbols (and also manipulate electrical and magnetic fields), but that is instrumental—the means to an end.

Now Dijkstra may object that computers are not doing all those things—I am talking about devices that employ computers in their functioning. Fine, let us not quibble over terms. Let us call this thing on my desktop a "computing device" which employs a computer in helping me with editing, formatting, and printing this response. As a profession, we are concerned with the overall education of people who will be responsible for specifying and implementing such devices (the software and hardware) in such a way that they will work effectively—not just perform symbol manipulations validly, but actually print a paycheck with the legal deductions, put the specified headings at the top of newsletter pages, and produce the desired machine part.

The second error is in his vision of what programmers do: "The programmer's task is . . . to give a formal proof that the program he proposes meets the equally formal functional specification." This is a noble goal, but it presupposes that someone else has done all the hard work by managing to create a formal functional specification that is appropriate to the task at hand. Dijkstra must face the unpleasant truth that it is the job of someone (again to avoid quibbling, let us call this person a "computing professional" rather than a "programmer") to produce a collection of instructions that allow the computing device to function appropriately in practice. In some cases, this might be best done by producing a full formal specification and then converting that into code, but that methodology is debatable, at best, and far from universally applicable.

In all his carping about the sad state of computer education, Dijkstra seems not to have noticed that a huge number of complex programs DO work...

Dijkstra's idealized view of programming shows its inadequacy immediately when I try to imagine a formal functional specification of the drawing program I use. He says, "deal with all the elements of a set by ignoring them and working with the set's definition." Consider the set of different ways in which the motions of the mouse and its buttons are used to create and modify images on the screen. He says, "we should reason about programs without even mentioning their possible 'behaviors'." How can we do this and consider what is to happen if I drag the cursor outside of the window while in the process of specifying a rectangle? In other words, once we recognize that we are engaged in the design of operational computing devices, we must train people to think well in operational terms.

The third error is in his peculiar view of "software engineering" and his pronouncement that its goal is self-contradictory. Engineering disciplines are concerned with the construction of devices that can be relied upon to perform a function. The success of such disciplines derives from the fact that past experience can help us avoid future breakdowns. By drawing on the accumulated experience of the profession, an engineer approaches a design task with a collection of techniques, tools, and previous designs which make it possible to create reasonably reliable devices at reasonable cost with a reasonable amount of effort. The key word in this is "reasonable," not "optimal" or "perfect." There are indeed failures. Sound engineering is not a guarantee of perfect results. And there are times when a radical departure is required to achieve a new design, but this is the rare exception, not the work of the everyday practitioner.

In all of his carping about the sad state of computer education, Dijkstra seems not to have noticed that a huge number of complex programs DO work, not always, and not necessarily in the most elegant way, but in an incredible variety of circumstances and without the benefit of formal proof. Furthermore, we are able to make more and more complex programs work, as we learn from the experience of past successes and failures. The goal of systematizing the knowledge gained from this experience is hardly self-contradictory, unless one has unrealistic or idealized expectations.

There are, indeed, important differences between computing devices and other kinds of devices, for the reasons that Dijkstra points out. It would be foolish to expect the methods of other engineering disciplines to apply without change. On the other hand—despite Dijkstra's grandiose statements about "a radically new intellectual challenge that has no precedent in our history" and his questionable analogies with quantum physics and non-euclidean geometry—the alleged "radical novelty" of computers is not so earth-shattering as to justify throwing away past experience in order to gain the pristine virtue of the "blank mind."

This is not to say that everything done under the banner of "software engineering" is good or that all of Dijkstra's criticisms are wrong. Occasionally, one of his errant fusillades hits a deserving target, and I find myself agreeing with the content of one of his remarks, if not with the ill-tempered style. But it would be just as wrong to condemn software engineering for the occasional stupidities of people who claim to be doing it as it would be to condemn all of mathematics for the occasional stupidities of mathematicians.

From the fact that Dijkstra's premises are false, it does not logically follow that his conclusions are false, so let us examine them separately. He mixes two arguments, one of which is appealing and the other of which is misguided.

The primary subtext of his diatribe is a complaint about the lack of rigor in computer science education. I fully support his pleas that as educators we demand rigorous thinking, teach the beauty of mathematics, and encourage the virtue of facing uncomfortable truths. But, he confuses this with the claim that the essential part of computing education lies in the ability to manipulate formal abstractions, detached from considerations of operational devices, their behaviors, or their embedding in a world of people and activities. If he deludes his students into thinking this, they are in for a rude awakening when they try to function as computing professionals.

If he is talking about the education of "computer scientists" in the narrow sense of theoreticians of formal computation, there is a bit of sense to his claim. Much of what he says may be applicable, for example, to the training of theoretical physicists who at times need to engage in purely formal manipulations without preconceptions in order to allow for radical novelty. But, it does not apply to the people who apply physical principles to designing bridges, airplanes, or disk-drives. Although they should certainly be proficient in mathematics, they would fail miserably at their tasks if they did not have a substantial knowledge of a very different kind. I take Dijkstra's argument not to be just about the training of the small elite cadre of theoreticians, but, rather, about his hundreds of freshmen—the broad population of people who work with computing devices.

Their education should include a solid grounding in formal methods, but this is just one piece of the preparatory background. To be educated, they need a grounding in the experience of the profession—the examples, methods, and practices. They need more than just a description of these. Effective learning comes from the experience of developing the skills they will be employing in their work, with observation and coaching from those who have expertise. This experience should go beyond the building and testing of small programming exercises to include working with larger-scale systems and the design considerations that come from their embedding in situations of use.

In this last question—the relation between mechanism and use—the current vision of "software engineering" is too narrow and needs to be expanded into a vision of design. A building is created by a combination of people including both engineers and architects. The architect's skill includes knowledge of materials and building techniques, but has a primary focus on function. The key questions go beyond construction to "What will make the building be appropriate to its uses? What will function as a good design when people move in?" We need to develop effective (and rigorous) training that builds the skills to answer these questions in designing computing systems.

It may well be that in the future there will be specialized professions within computing, and that there will be different kinds of training for its architects, engineers, and theoreticians. It would be foolish to ignore the value of the abstract mathematical skills Dijkstra advocates, but it would be even more foolish to indulge the fantasy that they offer some magic that allows students to escape the hard work of learning about real computing.

Terry Winograd
Dept. of Computer Science
Univ. of California at Stanford
Stanford, CA 93439

Dijkstra's Reply To Comments

Dear Colleagues,

Since your comments are not disjoint, allow me to address you collectively in principle.

Of course, there is more to digital system design than the formal derivation of programs from their equally formal specifications; accordingly, I expect a full-blown computer science curriculum to comprise more than "an introductory programming course for freshman." Please remember that my modest proposal only pertained to the latter.

The choice of functional specifications—and of the notation to write them down in—may be far from obvious, but their role is clear: it is to act as a logical firewall between two different concerns. The one is the "pleasantness problem," i.e., the question of whether an engine meeting the specification is the engine we would like to have; the other one is the "correctness problem," i.e., the question of how to design an engine meeting the specification. I firmly believe that whenever we succeed in erecting such a firewall, the effort will pay off handsomely. The reason for this belief of mine is that the two concerns deserve separation because the two problems are most effectively tackled by totally different techniques. (They are currently psychology and experimentation for the pleasantness problem and symbol manipulation for the correctness problem.)

Several of you have voiced a wider concern, *viz.*, serious doubts about the viability of a much more formalized mathematics. I can understand your concern because, for many years, I shared your doubts. Over the last decade, however, those doubts have largely evaporated, to the extent that I expect, say 50 years from now, the mathematical informality of today definitely to be a thing of the past. (I realize that this is a somewhat gratuitous statement because, if my expectation will not be fulfilled, I will not be there to be confronted with my mistake. But, I cannot help having expectations beyond my lifetime.) The expectations are based on the experiences and observations collected since I started to explore the extent to which the experience gathered in programming methodology could be transferred to mathematical methodology in general. I shall indicate them briefly.

1. The presumed dogma that calulational proofs are an order of magnitude too long to be practical has not been confirmed by my experience. On the contrary, well-chosen formalisms provide a shorthand with which no verbal rendering can compete.

2. Through most of this century, mathematical logic has primarily been used for soul-searching. As a tool for daily, practical proof design it has hardly been given a fair chance. To realize its potential, it seems essential to view the purpose of logic not as mimicking human reasoning but as providing a calculational alternative to it.

3. In proof design, strong heuristic guidance can be extracted from a syntactic analysis of the theorem and from (baby) proof theory. Both possibilities require formalization of the proof structure for their exploitation.

4. While informal mathematics, with its ties to elusive entities such as "intuition" and "the human mind," is a hard topic to teach explicitly, symbol manipulation is tangible. The effective techniques of symbol manipulation are well within the teachable domain, hence my estimate of 50 years.

The possible future of mathematics that I envisage is very different from what the dyed-in-the-wool member of the mathematical guild of today is used and probably attached to. My dream could very well be his nightmare. So be it. I cannot consider such discrepancy my fault. If you have a technical argument—why my expectation of the future of mathematics cannot come true—please let me know, for it would save me a lot of work and a lot of animosity.

Finally, allow me to express my appreciation for the care with which you phrased your comments. I end with my greetings and best wishes to you all.

E. D.