1 9 7 2
Turing
Award
Lecture

The Humble Programmer

EDSGER W. DIJKSTRA

[Extract from the Turing Award Citation read by M. D. McIlroy, chairman of the ACM Turing Award Committee, at the presentation of this lecture on August 14, 1972, at the ACM Annual Conference in Boston.]

The working vocabulary of programmers everywhere is studded with words originated or forcefully promulgated by E. W. Dijkstra—display, deadly embrace, semaphore, go-to-less programming, structured programming. But his influence on programming is more pervasive than any glossary can possibly indicate. The precious gift that this Turing Award acknowledges is Dijkstra's style*: his approach to programming as a high, intellectual challenge; his eloquent insistence and practical demonstration that programs should be composed correctly, not just debugged into correctness; and his illuminating perception of problems at the foundations of program design. He has published about a dozen papers, both technical and reflective, among which are especially to be noted his philosophical addresses at IFIP,*[1] *his already classic papers on cooperating sequential processes,*[2] *and his memorable indictment of the go-to statement.*[3] *An influential series of letters by Dijkstra have recently surfaced as a polished monograph on the art of composing programs.*[4]

[1]Some meditations on advanced programming, Proceedings of the IFIP Congress 1962, 535–538; Programming considered as a human activity, Proceedings of the IFIP Congress 1965, 213–217.

[2]Solution of a problem in concurrent programming, control, CACM 8 (Sept. 1965), 569; The structure of the "THE" multiprogramming system, CACM 11 (May 1968), 341–346.

[3]Go to statement considered harmful, CACM 11 (Mar. 1968), 147–148.

[4]A short introduction to the art of computer programming. Technische Hogeschool, Eindhoven, 1971.

Author's present address: Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712.

We have come to value good programs in much the same way as we value good literature. And at the center of this movement, creating and reflecting patterns no less beautiful than useful, stands E. W. Dijkstra.

As a result of a long sequence of coincidences I entered the programming profession officially on the first spring morning of 1952, and as far as I have been able to trace, I was the first Dutchman to do so in my country. In retrospect the most amazing thing is the slowness with which, at least in my part of the world, the programming profession emerged, a slowness which is now hard to believe. But I am grateful for two vivid recollections from that period that established that slowness beyond any doubt.

After having programmed for some three years, I had a discussion with van Wijngaarden, who was then my boss at the Mathematical Centre in Amsterdam — a discussion for which I shall remain grateful to him as long as I live. The point was that I was supposed to study theoretical physics at the University of Leiden simultaneously, and as I found the two activities harder and harder to combine, I had to make up my mind, either to stop programming and become a real, respectable theoretical physicist, or to carry my study of physics to a formal completion only, with a minimum of effort, and to become..., yes what? A programmer? But was that a respectable profession? After all, what was programming? Where was the sound body of knowledge that could support it as an intellectually respectable discipline? I remember quite vividly how I envied my hardware colleagues, who, when asked about their professional competence, could at least point out that they knew everything about vacuum tubes, amplifiers and the rest, whereas I felt that, when faced with that question, I would stand empty-handed. Full of misgivings I knocked on van Wijngaarden's office door, asking him whether I could speak to him for a moment; when I left his office a number of hours later, I was another person. For after having listened to my problems patiently, he agreed that up till that moment there was not much of a programming discipline, but then he went on to explain quietly that automatic computers were here to stay, that we were just at the beginning and could not I be one of the persons called to make programming a respectable discipline in the years to come? This was a turning point in my life and I completed my study of physics formally as quickly as I could. One moral of the above story is, of course, that we must be very careful when we give advice to younger people: sometimes they follow it!

Two years later, in 1957, I married, and Dutch marriage rites require you to state your profession and I stated that I was a programmer. But the municipal authorities of the town of Amsterdam did not accept it on the grounds that there was no such profession. And, believe it or not, but under the heading "profession" my marriage record shows the ridiculous entry "theoretical physicist"!

So much for the slowness with which I saw the programming profession emerge in my own country. Since then I have seen more of the world, and it is my general impression that in other countries, apart from a possible shift of dates, the growth pattern has been very much the same.

Let me try to capture the situation in those old days in a little bit more detail, in the hope of getting a better understanding of the situation today. While we pursue our analysis, we shall see how many common misunderstandngs about the true nature of the programming task can be traced back to that now distant past.

The first automatic electronic computers were all unique, single-copy machines and they were all to be found in an environment with the exciting flavor of an experimental laboratory. Once the vision of the automatic computer was there, its realization was a tremendous challenge to the electronic technology then available, and one thing is certain: we cannot deny the courage of the groups that decided to try to build such a fantastic piece of equipment. For fantastic pieces of equipment they were: in retrospect one can only wonder that those first machines worked at all, at least sometimes. The overwhelming problem was to get and keep the machine in working order. The preoccupation with the physical aspects of automatic computing is still reflected in the names of the older scientific societies in the field, such as the Association for Computing Machinery or the British Computer Society, names in which explicit reference is made to the physical equipment.

What about the poor programmer? Well, to tell the honest truth, he was hardly noticed. For one thing, the first machines were so bulky that you could hardly move them and besides that, they required such extensive maintenance that it was quite natural that the place where people tried to use the machine was the same laboratory where the machine had been developed. Secondly, the programmer's somewhat invisible work was without any glamour: you could show the machine to visitors and that was several orders of magnitude more spectacular than some sheets of coding. But most important of all, the programmer himself had a very modest view of his own work: his work derived all its significance from the existence of that wonderful machine. Because that was a unique machine, he knew only too well that his programs had only local significance, and also because it was patently obvious that this machine would have a limited lifetime, he knew that very little of his work would have a lasting value. Finally, there is yet another circumstance that had a profound influence on the programmer's attitude toward his work: on the one hand, besides being unreliable, his machine was usually too slow and its memory was usually too small, i.e., he was faced with a pinching shoe, while on the other hand its usually somewhat queer order code would cater for the most unexpected constructions. And in those days many a clever programmer derived an immense intellectual satisfaction from the cunning tricks by means of which he contrived to squeeze the impossible into the constraints of his equipment.

Two opinions about programming date from those days. I mention them now; I shall return to them later. The one opinion was that a really competent programmer should be puzzle-minded and very fond of clever tricks; the other opinion was that programming was nothing more than optimizing the efficiency of the computational process, in one direction or the other.

The latter opinion was the result of the frequent circumstance that, indeed, the available equipment was a painfully pinching shoe, and in those days one often encountered the naive expectation that, once more powerful machines were available, programming would no longer be a problem, for then the struggle to push the machine to its limits would no longer be necessary and that was all that programming was about, wasn't it? But in the next decades something completely different happened: more powerful machines became available, not just an order of magnitude more powerful, even several orders of magnitude more powerful. But instead of finding ourselves in a state of eternal bliss with all programming problems solved, we found ourselves up to our necks in the software crisis! How come?

There is a minor cause: in one or two respects modern machinery is basically more difficult to handle than the old machinery. Firstly, we have got the I/O interrupts, occurring at unpredictable and irreproducible moments; compared with the old sequential machine that pretended to be a fully deterministic automaton, this has been a dramatic change, and many a systems programmer's grey hair bears witness to the fact that we should not talk lightly about the logical problems created by that feature. Secondly, we have got machines equipped with multilevel stores, presenting us problems of management strategy that, in spite of the extensive literature on the subject, still remain rather elusive. So much for the added complication due to structural changes of the actual machines.

But I called this a minor cause; the major cause is . . . that the machines have become several orders of magnitude more powerful! To put it quite bluntly: as long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now that we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them — it has created the problem of using its products. To put it in another way: as the power of available machines grew by a factor of more than a thousand, society's ambition to apply these machines grew in proportion, and it was the poor programmer who found his job in this exploded field of tension between ends and means. The increased power of the hardware, together with the perhaps even more dramatic increase in its reliability, made solutions feasible that the programmer had not dared to dream about a few years before. And now, a few years later, he *had* to dream about them and, even worse, he had to transform such dreams into reality! Is it a wonder that we found ourselves in a software crisis? No, certainly not, and as you may guess, it was even predicted well in advance; but the trouble with minor prophets, of course, is that it is only five years later that you really know that they had been right.

Then, in the mid-sixties something terrible happened: the computers of the so-called third generation made their appearance. The official literature tells us that their price/performance ratio has been one of the major design objectives. But if you take as "performance" the duty cycle of the machine's various components, little will prevent you from ending up with a design in which the major part of your performance goal is reached by internal housekeeping activities of doubtful necessity. And if your definition of price is the price to be paid for the hardware, little will prevent you from ending up with a design that is terribly hard to program for: for instance the order code might be such as to enforce, either upon the programmer or upon the system, early binding decisions presenting conflicts that really cannot be resolved. And to a large extent these unpleasant possibilities seem to have become reality.

When these machines were announced and their functional specifications became known, many among us must have become quite miserable: at least I was. It was only reasonable to expect that such machines would flood the computing community, and it was therefore all the more important that their design should be as sound as possible. But the design embodied such serious flaws that I felt that with a single stroke the progress of computing science had been retarded by at least ten years; it was then that I had the blackest week in the whole of my professional life. Perhaps the most saddening thing now is that, even after all those years of frustrating experience, still so many people honestly believe that some law of nature tells us that machines have to be that way. They silence their doubts by observing how many of these machines have been sold, and derive from that observation the false sense of security that, after all, the design cannot have been that bad. But upon the closer inspection, that line of defense has the same convincing strength as the argument that cigarette smoking must be healthy because so many people do it.

It is in this connection that I regret that it is not customary for scientific journals in the computing area to publish reviews of newly announced computers in much the same way as we review scientific publications: to review machines would be at least as important. And here I have a confession to make: in the early sixties I wrote such a review with the intention of submitting it to Communications, but in spite of the fact that the few colleagues to whom the text was sent for their advice urged me to do so, I did not dare to do it, fearing that the difficulties either for myself or for the Editorial Board would prove to be too great. This suppression was an act of cowardice on my side for which I blame myself more and more. The difficulties I foresaw were a consequence of the absence of generally accepted criteria, and although I was convinced of the validity of the criteria I had chosen to apply, I feared that my review would be refused or discarded as "a matter of personal taste." I still think that such reviews would be extremely useful and I am longing to see them appear, for their accepted appearance would be a sure sign of maturity of the computing community.

The reason that I have paid the above attention to the hardware scene is because I have the feeling that one of the most important aspects of any computing tool is its influence on the thinking habits of those who try to use it, and because I have reasons to believe that the influence is many times stronger than is commonly assumed. Let us now switch our attention to the software scene.

Here the diversity has been so large that I must confine myself to a few stepping stones. I am painfully aware of the arbitrariness of my choice, and I beg you not to draw any conclusions with regard to my appreciation of the many efforts that will have to remain unmentioned.

In the beginning there was the EDSAC in Cambridge, England, and I think it quite impressive that right from the start the notion of a subroutine library played a central role in the design of that machine and of the way in which it should be used. It is now nearly 25 years later and the computing scene has changed dramatically, but the notion of basic software is still with us, and the notion of the closed subroutine is still one of the key concepts in programming. We should recognize the closed subroutine as one of the greatest software inventions; it has survived three generations of computers and it will survive a few more, because it caters for the implementation of one of our basic patterns of abstraction. Regrettably enough, its importance has been underestimated in the design of the third generation computers, in which the great number of explicitly named registers of the arithmetic unit implies a large overhead on the subroutine mechanism. But even that did not kill the concept of the subroutine, and we can only pray that the mutation won't prove to be hereditary.

The second major development on the software scene that I would like to mention is the birth of FORTRAN. At that time this was a project of great temerity, and the people responsible for it deserve our great admiration. It would be absolutely unfair to blame them for shortcomings that only became apparent after a decade or so of extensive usage: groups with a successful look-ahead of ten years are quite rare! In retrospect we must rate FORTRAN as a successful coding technique, but with very few effective aids to conception, aids which are now so urgently needed that time has come to consider it out of date. The sooner we can forget that FORTRAN ever existed, the better, for as a vehicle of thought it is no longer adequate: it wastes our brainpower, and it is too risky and therefore too expensive to use. FORTRAN's tragic fate has been its wide acceptance, mentally chaining thousands and thousands of programmers to our past mistakes. I pray daily that more of my fellow-programmers may find the means of freeing themselves from the curse of compatibility.

The third project I would not like to leave unmentioned is LISP, a fascinating enterprise of a completely different nature. With a few very basic principles at its foundation, it has shown a remarkable stability. Besides that, LISP has been the carrier for a considerable number of, in a sense, our most sophisticated computer applications. LISP has jokingly been described as "the most intelligent way to misuse a computer." I think that description a great compliment because it transmits the full flavor of liberation: it has assisted a number of our most gifted fellow humans in thinking previously impossible thoughts.

The fourth project to be mentioned is ALGOL 60. While up to the present day FORTRAN programmers still tend to understand their programming language in terms of the specific implementation they are working with — hence the prevalence of octal or hexadecimal dumps — while the definition of LISP is still a curious mixture of what the language means and how the mechanism works, the famous Report on the Algorithmic Language ALGOL 60 is the fruit of a genuine effort to carry abstraction a vital step further and to define a programming language in an implementation-independent way. One could argue that in this respect its authors have been so successful that they have created serious doubts as to whether it could be implemented at all! The report gloriously demonstrated the power of the formal method BNF, now fairly known as Backus-Naur-Form, and the power of carefully phrased English, at least when used by someone as brilliant as Peter Naur. I think that it is fair to say that only very few documents as short as this have had an equally profound influence on the computing community. The ease with which in later years the names ALGOL and ALGOL-like have been used, as an unprotected trademark, to lend glory to a number of sometimes hardly related younger projects is a somewhat shocking compliment to ALGOL's standing. The strength of BNF as a defining device is responsible for what I regard as one of the weaknesses of the language: an overelaborate and not too systematic syntax could now be crammed into the confines of very few pages. With a device as powerful as BNF, the Report on the Algorithmic Language ALGOL 60 should have been much shorter. Besides that, I am getting very doubtful about ALGOL 60's parameter mechanism: it allows the programmer so much combinatorial freedom that its confident use requires a strong discipline from the programmer. Besides being expensive to implement, it seems dangerous to use.

Finally, although the subject is not a pleasant one, I must mention PL/I, a programming language for which the defining documentation is of a frightening size and complexity. Using PL/I must be like flying a plane with 7,000 buttons, switches, and handles to manipulate in the cockpit. I absolutely fail to see how we can keep our growing programs firmly within our intellectual grip when by its sheer baroqueness the programming language—our basic tool, mind you!—already escapes our intellectual control. And if I have to describe the influence PL/I can have on its users, the closest metaphor that comes to my mind is that of a drug. I remember from a symposium on higher level programming languages a lecture given in defense of PL/I by a man who described himself as one of its devoted users. But within a one-hour lecture in praise of PL/I, he managed to ask for the addition of about 50 new "features," little supposing that the main source of his problems could very well be that it contained already far too many "features." The speaker displayed all the depressing symptoms of addiction, reduced as he was to the state of mental stagnation in which he could only ask for more, more, more. . . . When FORTRAN has been called an infantile disorder, full PL/I, with its growth characteristics of a dangerous tumor, could turn out to be a fatal disease.

So much for the past. But there is no point in making mistakes unless thereafter we are able to learn from them. As a matter of fact, I think that we have learned so much that within a few years programming can be an activity vastly different from what it has been up till now, so different that we had better prepare ourselves for the shock. Let me sketch for you one of the possible futures. At first sight, this vision of programming in perhaps already the near future may strike you as utterly fantastic. Let me therefore also add the considerations that might lead one to the conclusion that this vision could be a very real possibility.

The vision is that, well before the seventies have run to completion, we shall be able to design and implement the kind of systems that are now straining our programming ability at the expense of only a few percent in man-years of what they cost us now, and that besides that, these systems will be virtually free of bugs. These two improvements go hand in hand. In the latter respect software seems to be different from many other products, where as a rule a higher quality implies a higher price. Those who want really reliable software will discover that they must find means of avoiding the majority of bugs to start with, and as a result the programming process will become cheaper. If you want more effective programmers, you will discover that they should not waste their time debugging—they should not introduce the bugs to start with. In other words, both goals point to the same change.

Such a drastic change in such a short period of time would be a revolution, and to all persons that base their expectations for the future on smooth extrapolation of the recent past—appealing to some unwritten laws of social and cultural inertia—the chance that this drastic change will take place must seem negligible. But we all know that sometimes revolutions do take place! And what are the chances for this one?

There seem to be three major conditions that must be fulfilled. The world at large must recognize the need for the change; secondly, the economic need for it must be sufficiently strong; and, thirdly, the change must be technically feasible. Let me discuss these three conditions in the above order.

With respect to the recognition of the need for greater reliability of software, I expect no disagreement anymore. Only a few years ago this was different: to talk about a software crisis was blasphemy. The turning point was the Conference on Software Engineering in Garmisch, October 1968, a conference that created a sensation as there occurred the first open admission of the software crisis. And by now it is generally recognized that the design of any large sophisticated system is going to be a very difficult job, and whenever one meets people responsible for such undertakings, one finds them very much concerned about the reliability issue, and rightly so. In short, our first condition seems to be satisfied.

Now for the economic need. Nowadays one often encounters the opinion that in the sixties programming has been an overpaid profession, and that in the coming years programmer salaries may be expected to go down. Usually this opinion is expressed in connection with the recession, but it could be a symptom of something different and quite healthy, *viz.* that perhaps the programmers of the past decade have not done so good a job as they should have done. Society is getting dissatisfied with the performance of programmers and of their products. But there is another factor of much greater weight. In the present situation it is quite usual that for a specific system, the price to be paid for the development of the software is of the same order of magnitude as the price of the hardware needed, and society more or less accepts that. But hardware manufacturers tell us that in the next decade hardware prices can be expected to drop with a factor of ten. If software development were to continue to be the same clumsy and expensive process as it is now, things would get completely out of balance. You cannot expect society to accept this, and therefore we *must* learn to program an order of magnitude more effectively. To put it in another way: as long as machines were the largest item on the budget, the programming profession could get away with its clumsy techniques; but the umbrella will fold very rapidly. In short, also our second condition seems to be satisfied.

And now the third condition: is it technically feasible? I think it might be, and I shall give you six arguments in support of that opinion.

A study of program structure has revealed that programs—even alternative programs for the same task and with the same mathematical content—can differ tremendously in their intellectual manageability.

A number of rules have been discovered, violation of which will either seriously impair or totally destroy the intellectual manageability of the program. These rules are of two kinds. Those of the first kind are easily imposed mechanically, *viz.* by a suitably chosen programming language. Examples are the exclusion of go-to statements and of procedures with more than one output parameter. For those of the second kind, I at least—but that may be due to lack of competence on my side—see no way of imposing them mechanically, as it seems to need some sort of automatic theorem prover for which I have no existence proof. Therefore, for the time being and perhaps forever, the rules of the second kind present themselves as elements of discipline required from the programmer. Some of the rules I have in mind are so clear that they can be taught and that there never needs to be an argument as to whether a given program violates them or not. Examples are the requirements that no loop should be written down without providing a proof for termination or without stating the relation whose invariance will not be destroyed by the execution of the repeatable statement.

I now suggest that we confine ourselves to the design and implementation of intellectually manageable programs. If someone fears that this restriction is so severe that we cannot live with it, I can reassure him: the class of intellectually manageable programs is still sufficiently rich to contain many very realistic programs for any problem capable of algorithmic solution. We must not forget that it is *not* our business to make programs; it is our business to design classes of computations that will display a desired behavior. The suggestion of confining ourselves to intellectually manageable programs is the basis for the first two of my announced six arguments.

Argument one is that, as the programmer only needs to consider intellectually manageable programs, the alternatives he is choosing from are much, much easier to cope with.

Argument two is that, as soon as we have decided to restrict ourselves to the subject of the intellectually manageable programs, we have achieved, once and for all, a drastic reduction of the solution space to be considered. And this argument is distinct from argument one.

Argument three is based on the constructive approach to the problem of program correctness. Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness. But one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand. Argument three is essentially based on the following observation. If one first asks oneself what the structure of a convincing proof would be and, having found this, then constructs a program satisfying this proof's requirements, then these correctness concerns turn out to be a very effective heuristic guidance. By definition this approach is only applicable when we restrict ourselves to intellectually manageable programs, but it provides us with effective means for finding a satisfactory one among these.

Argument four has to do with the way in which the amount of intellectual effort needed to design a program depends on the program length. It has been suggested that there is some law of nature telling us that the amount of intellectual effort needed grows with the square of program length. But, thank goodness, no one has been able to prove this law. And this is because it need not be true. We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad of cases is called "abstraction"; as a result the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer. In this connection it might be worthwhile to point out that the purpose of abstracting is *not* to be vague, but to create a new semantic level in which one can be absolutely precise. Of course I have tried to find a fundamental cause that would prevent our abstraction mechanisms from being sufficiently effective. But no matter how hard I tried, I did not find such a cause. As a result I tend to the assumption — up till now not disproved by experience — that by suitable application of our powers of abstraction, the intellectual effort required to conceive or to understand a program need not grow more than proportional to program length. A by-product of these investigations may be of much greater practical significance, and is, in fact, the basis of my fourth argument. The by-product was the identification of a number of patterns of abstraction that play a vital role in the whole process of composing programs. Enough is known about these patterns of abstraction that you could devote a lecture to each of them. What the familiarity and conscious knowledge of these patterns of abstraction imply dawned upon me when I realized that, had they been common knowledge 15 years ago, the step from BNF to syntax-directed compilers, for instance, could have taken a few minutes instead of a few years. Therefore I present our recent knowledge of vital abstraction patterns as the fourth argument.

Now for the fifth argument. It has to do with the influence of the tool we are trying to use upon our own thinking habits. I observe a cultural tradition, which in all probability has its roots in the Renaissance, to ignore this influence, to regard the human mind as the supreme and autonomous master of its artifacts. But if I start to analyze the thinking habits of myself and of my fellow human beings, I come, whether I like it or not, to a completely different conclusion, *viz.* that the tools we are trying to use and the language or notation we are using to express or record our thoughts are the major factors determining that we can think or express at all! The analysis of the influence that programming languages have on the thinking habits of their users, and the recognition that, by now, brainpower is by far our scarcest resource, these together give us a new collection of yardsticks for comparing the relative merits of various programming languages. The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague. In the case of a well-known conversational programming language I have been told from various sides that as soon as a programming community is equipped with a terminal for it, a specific phenomenon occurs that even has a well-established name: it is called "the one-liners." It takes one of two different forms: one programmer places a one-line program on the desk of another and either he proudly tells what it does and adds the question, "Can you code this in less symbols?" —as if this were of any conceptual relevance!—or he just says, "Guess what it does!" From this observation we must conclude that this language as a tool is an open invitation for clever tricks; and while exactly this may be the explanation for some of its appeal, *viz.* to those who like to show how clever they are, I am sorry, but I must regard this as one of the most damning things that can be said about a programming language. Another lesson we should have learned from the recent past is that the development of "richer" or "more powerful" programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally. I see a great future for very systematic and very modest programming languages. When I say "modest," I mean that, for instance, not only ALGOL 60's "for clause," but even FORTRAN's "DO loop" may find themselves thrown out as being too baroque. I have run a little programming experiment with really experienced volunteers, but something quite unintended and quite unexpected turned up. None of my volunteers found the obvious and most elegant solution. Upon closer analysis this turned out to have a common source: their notion of repetition was so tightly connected to the idea of an associated controlled variable to be stepped up, that they were mentally blocked from seeing the obvious. Their solutions were less efficient, needlessly hard to understand, and it took them a very long time to find them. It was a revealing, but also shocking experience for me. Finally, in one respect one hopes that tomorrow's programming languages will differ greatly from what we are used to now: to a much greater extent than hitherto they should invite us to reflect in the structure of what we write down all abstractions needed to cope conceptually with the complexity of what we are designing. So much for the greater adequacy of our future tools, which was the basis of the fifth argument.

As an aside I would like to insert a warning to those who identify the difficulty of the programming task with the struggle against the inadequacies of our current tools, because they might conclude that, once our tools will be much more adequate, programming will no longer be a problem. Programming will remain very difficult, because once we have freed ourselves from the circumstantial cumbersomeness, we will find ourselves free to tackle the problems that are now well beyond our programming capacity.

You can quarrel with my sixth argument, for it is not so easy to collect experimental evidence for its support, a fact that will not prevent me from believing in its validity. Up till now I have not mentioned the word "hierarchy," but I think that it is fair to say that this is a key concept for all systems embodying a nicely factored solution. I could even go one step further and make an article of faith out of it, *viz.* that the only problems we can really solve in a satisfactory manner are those that finally admit a nicely factored solution. At first sight this view of human limitations may strike you as a rather depressing view of our predicament, but I don't feel it that way. On the contrary, the best way to learn to live with our limitations is to know them. By the time we are sufficiently modest to try factored solutions only, because the other efforts escape our intellectual grip, we shall do our utmost to avoid all those interfaces impairing our ability to factor the system in a helpful way. And I cannot but expect that this will repeatedly lead to the discovery that an initially untractable problem can be factored after all. Anyone who has seen how the majority of the troubles of the compiling phase called "code generation" can be tracked down to funny properties of the order code will know a simple example of the kind of things I have in mind. The wide applicability of nicely factored solutions is my sixth and last argument for the technical feasibility of the revolution that might take place in the current decade.

In principle I leave it to you to decide for yourself how much weight you are going to give to my considerations, knowing only too well that I can force no one else to share my beliefs. As in each serious revolution, it will provoke violent opposition and one can ask oneself where to expect the conservative forces trying to counteract such a development. I don't expect them primarily in big business, not even in the computer business: I expect them rather in the educational institutions that provide today's training and in those conservative groups of computer users that think their old programs so important that they don't think it worthwhile to rewrite and improve them. In this connection it is sad to observe that on many a university campus the choice of the central computing facility has too often been determined by the demands of a few established but expensive applications with a disregard of the question, how many thousands of "small users" who are willing to write their own programs are going to suffer from this choice. Too often, for instance, high-energy physics seems to have blackmailed the scientific community with the price of its remaining experimental equipment. The easiest answer, of course, is a flat denial of the technical feasibility, but I am afraid that you need pretty strong arguments for that. No reassurance, alas, can be obtained from the remark that the intellectual ceiling of today's average programmer will prevent the revolution from taking place: with others programming so much more effectively, he is liable to be edged out of the picture anyway.

There may also be political impediments. Even if we know how to educate tomorrow's professional programmer, it is not certain that the society we are living in will allow us to do so. The first effect of teaching a methodology—rather than disseminating knowledge—is that of enhancing the capacities of the already capable, thus magnifying the difference in intelligence. In a society in which the educational system is used as an instrument for the establishment of a homogenized culture, in which the cream is prevented from rising to the top, the education of competent programmers could be politically unpalatable.

Let me conclude. Automatic computers have now been with us for a quarter of a century. They have had a great impact on our society in their capacity of tools, but in that capacity their influence will be but a ripple on the surface of our culture compared with the much more profound influence they will have in their capacity of intellectual challenge which will be without precedent in the cultural history of mankind. Hierarchical systems seem to have the property that something considered as an undivided entity on one level is considered as a composite object on the next lower level of greater detail; as a result the natural grain of space or time that is applicable at each level decreases by an order of magnitude when we shift our attention from one level to the next lower one. We understand walls in terms of bricks, bricks in terms of crystals, crystals in terms of molecules, etc. As a result the number of levels that can can be distinguished meaningfully in a hierarchical system is kind of proportional to the logarithm of the ratio between the largest and the smallest grain, and therefore, unless this ratio is very large, we cannot expect many levels. In computer programming our basic building block has an associated time grain of less than a microsecond, but our program may take hours of computation time. I do not know of any other technology covering a ratio of 10^{10} or more: the computer, by virtue of its fantastic speed, seems to be the first to provide us with an environment where highly hierarchical artifacts are both possible and necessary. This challenge, *viz.* the confrontation with the programming task, is so unique that this novel experience can teach us a lot about ourselves. It should deepen our understanding of the processes of design and creation; it should give us better control over the task of organizing our thoughts. If it did not do so, to my taste we should not deserve the computer at all!

It has already taught us a few lessons, and the one I have chosen to stress in this talk is the following. We shall do a much better programming job, provided that we approach the task with a full appreciation of its tremendous difficulty, provided that we stick to modest and elegant programming languages, provided that we respect the intrinsic limitations of the human mind and approach the task as Very Humble Programmers.

Categories and Subject Descriptors:

D.2.4 [**Software**]: Program Verification—*correctness proofs*; D.3.0 [**Software**]: General—*standards*; D.3.3 [**Software**]: Language Constructs—*procedures, functions and subroutines*; K.2 [**Computing Milieux**]: History of Computing—*people*; K.7.1 [**Computing Milieux**]: The Computing Profession—*occupations*

General Terms:

Design, Human Factors, Languages, Reliability

Additional Key Words and Phrases:

ALGOL 60, EDSAC, FORTRAN, PL/I

Postscript

EDSGER W. DIJKSTRA
Department of Computer Sciences
The University of Texas at Austin

My Turing Award lecture of 1972 was very much a credo that presented the programming task as an intellectual challenge of the highest caliber. That credo strikes me now (in 1986) as still fully up to date: How not to get lost in the complexities of our own making is still computing's core challenge.

In its proposals of how to meet that challenge, however, the lecture is clearly dated: Had I to give it now, I would devote a major part of it to the role of formal techniques in programming.

The confrontation of my expectations in those days with what has happened since evokes mixed feelings. On the one hand, my wildest expectations have been surpassed: neat, concise arguments leading to sophisticated algorithms that were very hard, if not impossible, to conceive as little as ten years ago are a regular source of intellectual excitement. On the other hand, I am disappointed to see how little of this has penetrated into the average computing science curriculum, in which the effective design of high-quality programs is neglected in favor of fads (say, "incremental self-improvement of the user-friendliness of expert systems interfaces").

There is an upper bound on the speed with which society can absorb progress, and I guess I have still to learn how to be more patient.