1968 ACM Turing Lecture

One Man's View of Computer Science

R. W. HAMMING

Bell Telephone Laboratories, Inc., Murray Hill, New Jersey

ABSTRACT. A number of observations and comments are directed toward suggesting that more than the usual engineering flavor be given to computer science. The engineering aspect is important because most present difficulties in this field do not involve the theoretical question of whether certain things can be done, but rather the practical question of how can they be accomplished well and simply.

The teaching of computer science could be made more effective by various alterations, for example, the inclusion of a laboratory course in programming, the requirement for a strong minor in something other than mathematics, and more practical coding and less abstract theory, as well as more seriousness and less game playing.

KEY WORDS AND PHRASES: computer science, computer engineering, practical programming, mathematical game-playing, computer technician, computer professional, true-to-life programming, computer science curriculum, software, basic research, undirected research, programmers' ethical standards, programmers' social responsibility

CR CATEGORIES: 1.3, 1.50

Let me begin with a few personal words. When one is notified that he has been elected the ACM Turing lecturer for the year, he is at first surprised—especially is the nonacademic person surprised by an ACM award. After a little while the surprise is replaced by a feeling of pleasure. Still later comes a feeling of "Why me?" With all that has been done and is being done in computing, why single out me and my work? Well, I suppose that it has to happen to someone each year, and this time I am the lucky person. Anyway, let me thank you for the honor you have given to me and by inference to the Bell Telephone Laboratories where I work and which has made possible so much of what I have done.

The topic of my Turing lecture, "One Man's View of Computer Science," was picked because "What is computer science?" is argued endlessly among people in the field. Furthermore, as the excellent Curriculum 68 report[1] remarks in its introduction, "The Committee believes strongly that a continuing dialogue on the process and goals of education in computer science will be vital in the years to come." Lastly, it is wrong to think of Turing, for whom these lectures were named, as being exclusively interested in Turing machines; the fact is that he contributed to many aspects of the field and would probably have been very interested in the topic, though perhaps not in what I say.

The question "What is computer science?" actually occurs in many different

[1] A Report of the ACM Curriculum Committee on Computer Science; *Comm. ACM 11*, 3 (Mar. 1968), 151–197.

forms, among which are: What is computer science currently? What can it develop into? What should it develop into? What will it develop into?

A precise answer cannot be given to any of these. Many years ago an eminent mathematician wrote a book *What is Mathematics* and nowhere did he try to define mathematics, rather he simply wrote mathematics. While you will now and then find some aspect of mathematics defined rather sharply, the only generally agreed upon definition of mathematics is "Mathematics is what mathematicians do", which is followed by "Mathematicians are people who do mathematics." What is true about defining mathematics is also true about many other fields: there is often no clear, sharp definition of the field.

In the face of this difficulty many people, including myself at times, feel that we should ignore the discussion and get on with *doing* it. But as George Forsythe points out so well[2] in a recent article, it *does* matter what people in Washington, D. C. think computer science is. According to him, they tend to feel that it is a part of applied mathematics and therefore turn to the mathematicians for advice in the granting of funds. And it is not greatly different elsewhere; in both industry and the universities you can often still see traces of where computing first started, whether in electrical engineering, physics, mathematics, or even business. Evidently the picture which people have of a subject can significantly affect its subsequent development. Therefore, although we cannot hope to settle the question definitively, we need frequently to examine and to air our views on what our subject is and should become.

In many respects, for me it would be more satisfactory to give a talk on some small, technical point in computer science—it would certainly be easier. But that is exactly one of the things that I wish to stress—the danger of getting lost in the details of the field, especially in the coming days when there will be a veritable blizzard of papers appearing each month in the journals. We must give a good deal of attention to a broad training in the field—this in the face of the increasing necessity to specialize more and more highly in order to get a thesis problem, publish many papers, etc. We need to prepare our students for the year 2000 when many of them will be at the peak of their career. It seems to me to be more true in computer science than in many other fields that "specialization leads to triviality."

I am sure you have all heard that our scientific knowledge has been doubling every 15 to 17 years. I strongly suspect that the rate is now much higher in computer science; certainly it was higher during the past 15 years. In all of our plans we must take this growth of information into account and recognize that in a very real sense we face a "semi-infinite" amount of knowledge. In many respects the classical concept of a scholar who knows at least 90 percent of the relevant knowledge in his field is a dying concept. Narrower and narrower specialization is *not* the answer, since in part the difficulty is in the rapid growth of the interrelationships between fields. It is my private opinion that we need to put relatively more stress on quality and less on quantity and that the careful, critical, considered survey articles will often be more significant in advancing the field than new, nonessential material.

We live in a world of shades of grey, but in order to argue, indeed even to think, it is often necessary to dichotomize and say "black" or "white". Of course in doing

[2] FORSYTHE, G. E. What to do till the computer scientist comes. *Am. Math. Monthly 75*, 5 (May 1968), 454–461.

so we do violence to the truth, but there seems to be no other way to proceed. I trust, therefore, that you will take many of my small distinctions in this light—in a sense, I do not believe them myself, but there seems to be no other simple way of discussing the matter.

For example, let me make an arbitrary distinction between science and engineering by saying that science is concerned with *what* is possible while engineering is concerned with *choosing*, from among the many possible ways, *one* that meets a number of often poorly stated economic and practical objectives. We call the field "computer science" but I believe that it would be more accurately labeled "computer engineering" were not this too likely to be misunderstood. So much of what we do is not a question of can it be done as it is a question of finding a practical way. It is not usually a question of can there exist a monitor system, algorithm, scheduler, or compiler, rather it is a question of finding a practical working one with a reasonable expenditure of time and effort. While I would not change the name from "computer science" to "computer engineering," I would like to see far more of a practical, engineering flavor in what we teach than I usually find in course outlines.

There is a second reason for asking that we stress the practical side. As far into the future as I can see, computer science departments are going to need large sums of money. Now society usually, though not always, is more willing to give money when it can see practical returns than it is to invest in what it regards as impractical activities, amusing games, etc. If we are to get the vast sums of money I believe we will need, then we had better give a practical flavor to our field. As many of you are well aware, we have already acquired a bad reputation in many areas. There have been exceptions, of course, but all of you know how poorly we have so far met the needs for software.

At the heart of computer science lies a technological device, the computing machine. Without the machine almost all of what we do would become idle speculation, hardly different from that of the notorious Scholastics of the Middle Ages. The founders of the ACM clearly recognized that most of what we did, or were going to do, rested on this technological device, and they deliberately included the word "machinery" in the title. There are those who would like to eliminate the word, in a sense to symbolically free the field from reality, but so far these efforts have failed. I do not regret the initial choice. I still believe that it is important for us to recognize that the computer, the information processing machine, is the foundation of our field.

How shall we produce this flavor of practicality that I am asking for, as well as the reputation for delivering what society needs at the time it is needed? Perhaps most important is the way we go about our business and our teaching, though the research we do will also be very important. We need to avoid the bragging of uselessness and the game-playing that the pure mathematicians so often engage in. Whether or not the pure mathematician is right in claiming that what is utterly useless today will be useful tomorrow (and I doubt very much that he is, in the current situation), it is simply poor propaganda for raising the large amounts of money we need to support the continuing growth of the field. We need to avoid making computer science look like pure mathematics: our primary standard for acceptance should be experience in the real world, not aesthetics.

Were I setting up a computer science program, I would give relatively more

emphasis to laboratory work than does Curriculum 68, and in particular I would require every computer science major, undergraduate or graduate, to take a laboratory course in which he designs, builds, debugs, and documents a reasonably sized program, perhaps a simulator or a simplified compiler for a particular machine. The results would be judged on style of programming, practical efficiency, freedom from bugs, and documentation. If any of these were too poor, I would not let the candidate pass. In judging his work we need to distinguish clearly between superficial cleverness and genuine understanding. Cleverness was essential in the past; it is no longer sufficient.

I would also require a strong minor in some field *other* than computer science and mathematics. Without real experience in using the computer to get useful results the computer science major is apt to know all about the marvelous tool except how to use it. Such a person is a mere technician, skilled in manipulating the tool but with little sense of how and when to use it for its basic purposes. I believe we should avoid turning out more idiot savants—we have more than enough "computniks" now to last us a long time. What we need are professionals!

The Curriculum 68 recognized this need for "true-to-life" programming by saying, "This might be arranged through summer employment, a cooperative work-study program, part-time employment in computer centers, special projects courses, or some other appropriate means." I am suggesting that the appropriate means is a stiff laboratory course under your own control, and that the above suggestions of the Committee are rarely going to be effective or satisfactory.

Perhaps the most vexing question in planning a computer science curriculum is determining the mathematics courses to require of those who major in the field. Many of us came to computing with a strong background in mathematics and tend automatically to feel that a lot of mathematics should be required of everyone. All too often the teacher tries to make the student into a copy of himself. But it is easy to observe that in the past many highly rated software people were ignorant of most of formal mathematics, though many of them seemed to have a natural talent for mathematics (as it is, rather than as it is so often taught).

In the past I have argued that to require a strong mathematical content for computer science would exclude many of the best people in the field. However, with the coming importance of scheduling and the allocating of the resources of the computer, I have had to reconsider my opinion. While there is some evidence that part of this will be incorporated into the hardware, I find it difficult to believe that there will not be for a long time (meaning at least five years) a lot of scheduling and allocating of resources in software. If this is to be the pattern, then we need to consider training in this field. If we do not give such training, then the computer science major will find that he is a technician who is merely programming what others tell him to do. Furthermore, the kinds of programming that were regarded in the past as being great often depended on cleverness and trickery and required little or no formal mathematics. This phase seems to be passing, and I am forced to believe that in the future a good mathematical background will be needed if our graduates are to do significant work.

History shows that relatively few people can learn much new mathematics in their thirties, let alone later in life; so that if mathematics is going to play a significant role in the future, we need to give the students mathematical training while they are in school. We can, of course, evade the issue for the moment by providing

two parallel paths, one with and one without mathematics, with the warning that the nonmathematical path leads to a dead end so far as further university training is concerned (assuming we believe that mathematics is essential for advanced training in computer science).

Once we grant the need for a lot of mathematics, then we face the even more difficult task of saying specifically which courses. In spite of the numerical analysts' claims for the fundamental importance of their field, a surprising amount of computer science activity requires comparatively little of it. But I believe we can defend the requirement that *every* computer science major take at least one course in the field. Our difficulty lies, perhaps, in the fact that the present arrangement of formal mathematics courses is not suited to our needs as we presently see them. We seem to need some abstract algebra; some queuing theory; a lot of statistics, including the design of experiments; a moderate amount of probability, with perhaps some elements of Markov chains; parts of information and coding theory; and a little on bandwidth and signalling rates, some graph theory, etc., but we also know that the field is rapidly changing and that tomorrow we may need complex variables, topology, and other topics.

As I said, the planning of the mathematics courses is probably the most vexing part of the curriculum. After a lot of thinking on the matter, I currently feel that if our graduates are to make significant contributions and not be reduced to the level of technicians running a tool as they are told by others, then it is better to give them too much mathematics rather than too little. I realize all too well that this will exclude many people who in the past have made contributions, and I am not happy about my conclusion, but there it is. In the future, success in the field of computer science is apt to require a command of mathematics.

One of the complaints regularly made of computer science curriculums is that they seem to almost totally ignore business applications and COBOL. I think that it is not a question of how important the applications are, nor how widely a language like COBOL is used, that should determine whether or not it is taught in the computer science department; rather, I think it depends on whether or not the business administration department can do a far better job than we can, and whether or not what is peculiar to the business applications is fundamental to other aspects of computer science. And what I have indicated about business applications applies, I believe, to most other fields of application that can be taught in other departments. I strongly believe that with the limited resources we have, and will have for a long time to come, we should not attempt to teach applications of computers in the computer science department—rather, those applications should be taught in their natural environments by the appropriate departments.

The problem of the role of analog computation in the computer science curriculum is not quite the same as that of applications to special fields, since there is really no place else for it to go. There is little doubt that analog computers are economically important and will continue to be so for some time. But there is also little doubt that the field, even including hybrid computers, does not have at present the intellectual ferment that digital computation does. Furthermore, the essence of good analog computation lies in the understanding of the physical limitations of the equipment and in the peculiar art of scaling, especially in the time variable, which is quite foreign to the rest of computer science. It tends, therefore, to be ignored rather than to be rejected; it is either not taught or else it is an elective, and

this is probably the best we can expect at present when the center of interest is the general purpose digital computer.

At present there is a flavor of "game-playing" about many courses in computer science. I hear repeatedly from friends who want to hire good software people that they have found the specialist in computer science is someone they do *not* want. Their experience is that graduates of our programs seem to be mainly interested in playing games, making fancy programs that really do not work, writing trick programs, etc. and are unable to discipline their own efforts so that what they say they will do gets done on time and in practical form. If I had heard this complaint merely once from a friend who fancied that he was a hard-boiled engineer, then I would dismiss it; unfortunately I have heard it from a number of capable, intelligent, understanding people. As I earlier said, since we have such a need for financial support for the current and future expansion of our facilities, we had better consider how we can avoid such remarks being made about our graduates in the coming years. Are we going to continue to turn out a product that is not wanted in many places? Or are we going to turn out responsible, effective people who meet the real needs of our society? I hope that the latter will be increasingly true; hence my emphasis on the practical aspects of computer science.

One of the reasons that the computer scientists we turn out are more interested in "cute" programming than in results is that many of our courses are being taught by people who have the instincts of a pure mathematician. Let me make another arbitrary distinction which is only partially true. The pure mathematician starts with the given problem, or else some variant that he has made up from the given problem, and produces what he says is an answer. In applied mathematics it is necessary to add two crucial steps (1) an examination of the relevance of the mathematical model to the actual situation, and (2) the relevance of, or if you wish the interpretation of, the results of the mathematical model back to the original situation. This is where there is the sharp difference: The applied mathematician must be willing to stake part of his reputation on the remark "If you do so and so you will observe such and such very closely and therefore you are justified in going ahead and spending the money, or effort, to do the job as indicated," while the pure mathematician usually shrugs his shoulders and says, "That is none of my responsibility." Someone must take the responsibility for the decision to go ahead on one path or another, and it seems to me that he who does assume this responsibility will get the greater credit, on the average, as it is doled out by society. We need, therefore, in our teaching of computer science, to stress the assuming of responsibility for the *whole* problem and not just the cute mathematical part. This is another reason why I have emphasized the engineering aspects of the various subjects and tried to minimize the purely mathematical aspects.

The difficulty is, of course, that so many of our teachers in computer science are pure mathematicians and that pure mathematics is so much easier to teach than is applied work. There are relatively few teachers available to teach in the style I am asking for. This means we must do the best we can with what we have, but we should be conscious of the direction we want to take and that we want, where possible, to give a practical flavor of responsibility and engineering rather than mere existence of results.

It is unfortunate that in the early stages of computer science it is the talent and ability to handle a sea of minutiae which is important for success. But if the student

is to grow into someone who can handle the larger aspects of computer science, then he must have, and develop, other talents which are not being used or exercised at the early stages. Many of our graduates never make this second step. The situation is much like that in mathematics: in the early years it is the command of the trivia of arithmetic and formal symbol manipulation of algebra which is needed, but in advanced mathematics a far different talent is needed for success. As I said, many of the people in computer science who made their mark in the area where the minutiae are the dominating feature do not develop the larger talents, and they are still around teaching and propagating their brand of detail. What is needed in the higher levels of computer science is not the "black or white" mentality that characterizes so much of mathematics, but rather the judgment and balancing of conflicting aims that characterize engineering.

I have so far skirted the field of software, or, as a friend of mine once said, "ad hoc-ery." There is so much truth in his characterization of software as ad hoc-ery that it is embarrassing to discuss the topic of what to teach in software courses. So much of what we have done has been in an ad hoc fashion, and we have been under so much pressure to get something going as soon as possible that we have precious little which will stand examination by the skeptical eye of a scientist or engineer who asks, "What content is there in software?" How few are the difficult ideas to grasp in the field! How much is mere piling on of detail after detail without any careful analysis! And when 50,000-word compilers are later remade with perhaps 5000 words, how far from reasonable must have been the early ones!

I am no longer a software expert, so it is hard for me to make serious suggestions about what to do in the software field, yet I feel that all too often we have been satisfied with such a low level of quality that we have done ourselves harm in the process. We seem not to be able to use the machine, which we all believe is a very powerful tool for manipulating and transforming information, to do our own tasks in this very field. We have compilers, assemblers, monitors, etc. for others, and yet when I examine what the typical software person does, I am often appalled at how little he uses the machine in his own work. I have had enough minor successes in arguments with software people to believe that I am basically right in my insistence that we should learn to use the machine at almost every stage of what we are doing. Too few software people even try to use the machine on their own work. There are dozens of situations where a little machine computation would greatly aid the programmer. I recall one very simple one where a nonexpert with a very long FORTRAN program from the outside wanted to convert it to our local use, so he wrote a simple FORTRAN program to locate all the input-output statements and all the library references. In my experience, most programmers would have personally scanned long listings of the program to find them and with the usual human fallibility missed a couple the first time. I believe we need to convince the computer expert that the machine is his most powerful tool and that he should learn to use it as much as he can rather than personally scan the long listings of symbols as I see being done everywhere I go around the country. If what I am reporting is at all true, we have failed to teach this in the past. Of course some of the best people do in fact use the computer as I am recommending; my observation is that the run-of-the-mill programmers do not do so.

To parody our current methods of teaching programming, we give beginners a grammar and a dictionary and tell them that they are now great writers. We

seldom, if ever, give them any serious training in *style*. Indeed I have watched for years for the appearance of a *Manual of Style* and/or an *Anthology of Good Programming* and have as yet found none. Like writing, programming is a difficult and complex art. In both writing and programming, compactness is desirable but in both you can easily be too compact. When you consider how we teach good writing —the exercises, the compositions, and the talks that the student gives and is graded on by the teacher during his training in English—it seems we have been very remiss in this matter of teaching style in programming. Unfortunately only few programmers who admit that there is something in what I have called "style" are willing to formulate their feelings and to give specific examples. As a result, few programmers write in flowing poetry; most write in halting prose.

I doubt that style in programming is tied very closely to any particular machine or language, any more than good writing in one natural language is significantly different than it is in another. There are, of course, particular idioms and details in one language that favor one way of expressing the idea rather than another, but the essentials of good writing seem to transcend the differences in the Western European languages with which I am familiar. And I doubt that it is much different for most general purpose digital machines that are available these days.

Since I am apt to be misunderstood when I say we need more of an engineering flavor and less of a science one, I should perhaps point out that I came to computer science with a Ph.D. in pure mathematics. When I ask that the training in software be given a more practical, engineering flavor, I also loudly proclaim that we have too little understanding of what we are doing and that we desperately need to develop relevant theories.

Indeed, one of my major complaints about the computer field is that whereas Newton could say, "If I have seen a little farther than others it is because I have stood on the shoulders of giants," I am forced to say, "Today we stand on each other's feet." Perhaps the central problem we face in all of computer science is how we are to get to the situation where we build on top of the work of others rather than redoing so much of it in a trivially different way. Science is supposed to be cumulative, not almost endless duplication of the same kind of things.

This brings me to another distinction, that between undirected research and basic research. Everyone likes to do undirected research and most people like to believe that undirected research is basic research. I am choosing to define basic research as being work upon which people will in the future base a lot of their work. After all, what else can we reasonably mean by basic research other than work upon which a lot of later work is based? I believe experience shows that relatively few people are *capable* of doing basic research. While one cannot be certain that a particular piece of work will or will not turn out to be basic, one can often give fairly accurate probabilities on the outcome. Upon examining the question of the nature of basic research, I have come to the conclusion that what determines whether or not a piece of work has much chance to become basic is not so much the question asked as it is the way the problem is attacked.

Numerical analysis is the one venerable part of our curriculum that is widely accepted as having some content. Yet all too often there is some justice in the remark that many of the textbooks are written for mathematicians and are in fact much more mathematics than they are practical computing. The reason is, of course, that many of the people in the field are converted, or rather only partially

converted, mathematicians who still have the unconscious standards of mathematics in the back of their minds. I am sure many of you are familiar with my objections[3] along these lines and I need not repeat them here.

It has been remarked to me by several persons, and I have also observed, that many of the courses in the proposed computer science curriculum are padded. Often they appear to cover every detail rather than confining themselves to the main ideas. We do not need to teach every method for finding the real zeros of a function: we need to teach a few typical ones that are both effective and illustrate basic concepts in numerical analysis. And what I have just said about numerical analysis goes even more for software courses. There do not seem to me (and to some others) to be enough fundamental ideas in all that we know of software to justify the large amount of time that is devoted to the topic. We should confine the material we teach to that which is important in ideas and technique—the plodding through a mass of minutiae should be avoided.

Let me now turn to the delicate matter of ethics. It has been observed on a number of occasions that the ethical behavior of the programmers in accounting installations leaves a lot to be desired when compared to that of the trained accounting personnel.[4] We seem not to teach the "sacredness" of information about people and private company material. My limited observation of computer experts is that they have only the slightest regard for these matters. For example, most programmers believe they have the right to take with them any program they wish when they change employers. We should look at, and copy, how ethical standards are incorporated into the traditional accounting courses (and elsewhere), because they turn out a more ethical product than we do. We talk a lot in public of the dangers of large data banks of personnel records, but we do not do our share at the level of indoctrination of our own computer science majors.

Along these lines, let me briefly comment on the matter of professional standards. We have recently had a standard published[5] and it seems to me to be a good one, but again I feel that I am justified in asking how this is being incorporated into the training of our students, how they are to learn to behave that way. Certainly it is not sufficient to read it to the class each morning; both ethical and professional behavior are not effectively taught that way. There is plenty of evidence that other professions do manage to communicate to their students professional standards which, while not always followed by every member, are certainly a lot better instilled than those we are presently providing for our students. Again, we need to examine how they do this kind of training and try to adapt their methods to our needs.

Lastly, let me mention briefly the often discussed topic of social responsibility. We have sessions at meetings on this topic, we discuss it in the halls and over coffee and beer, but again I ask, "How is it being incorporated into our training program?" The fact that we do not have exact rules to follow is not sufficient reason for omitting all training in this important matter.

I believe these three topics—ethics, professional behavior, and social responsibility—must be incorporated into the computer science curriculum. Personally

[3] HAMMING, R. W. Numerical analysis vs. mathematics. *Science 148* (Apr. 1965), 473–475.
[4] CAREY, J. L., AND DOHERTY, W. A. Ethical Standards of the Accounting Profession. Am. Inst. CPAs., 1966.
[5] *Comm. ACM 11*, 3 (Mar. 1968), 198–220.

I do not believe that a separate course on these topics will be effective. From what little I understand of the matter of teaching these kinds of things, they can best be taught by example, by the behavior of the professor. They are taught in the odd moments, by the way the professor phrases his remarks and handles himself. Thus it is the professor who must first be made conscious that a significant part of his teaching role is in communicating these delicate, elusive matters and that he is not justified in saying, "They are none of my business." These are things that must be taught *constantly, all the time, by everyone,* or they will not be taught at all. And if they are not somehow taught to the majority of our students, then the field will justly keep its present reputation (which may well surprise you if you ask your colleagues in other departments for their frank opinions).

In closing, let me revert to a reasonable perspective of the computer science field. The field is very new, it has had to run constantly just to keep up, and there has been little time for many of the things we have long known we must some day do. But at least in the universities we have finally arrived: we have established separate departments with reasonable courses, faculty, and equipment. We are now well started, and it is time to deepen, strengthen, and improve our field so that we can be justly proud of what we teach, how we teach it, and of the students we turn out. We are not engaged in turning out technicians, idiot savants, and computniks; we know that in this modern, complex world we must turn out people who can play responsible major roles in our changing society, or else we must acknowledge that we have failed in our duty as teachers and leaders in this exciting, important field—computer science.