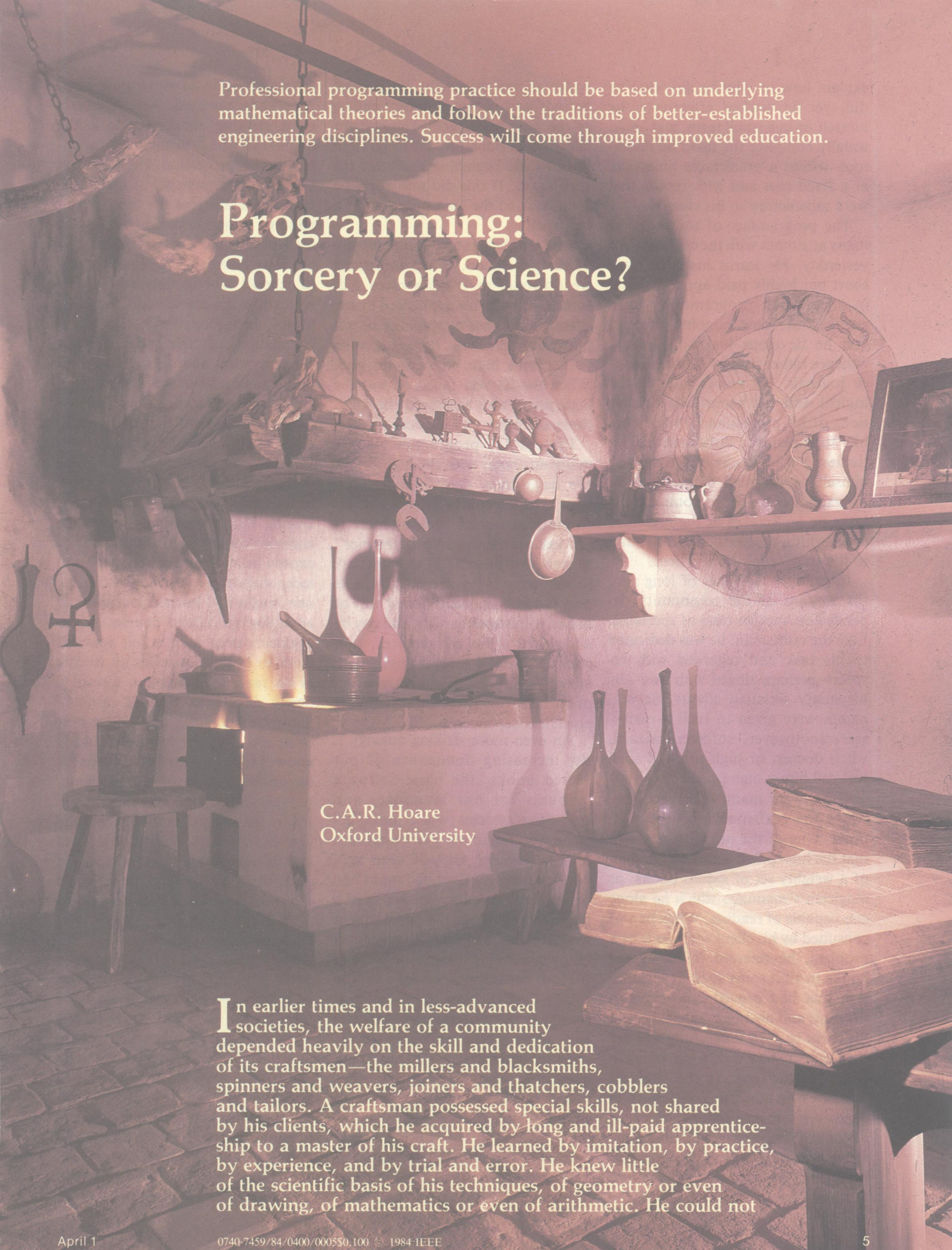


Professional programming practice should be based on underlying mathematical theories and follow the traditions of better-established engineering disciplines. Success will come through improved education.

Programming: Sorcery or Science?

C.A.R. Hoare
Oxford University

In earlier times and in less-advanced societies, the welfare of a community depended heavily on the skill and dedication of its craftsmen—the millers and blacksmiths, spinners and weavers, joiners and thatchers, cobblers and tailors. A craftsman possessed special skills, not shared by his clients, which he acquired by long and ill-paid apprenticeship to a master of his craft. He learned by imitation, by practice, by experience, and by trial and error. He knew little of the scientific basis of his techniques, of geometry or even of drawing, of mathematics or even of arithmetic. He could not explain how or why he did what he did, yet he worked effectively, by himself or in a small team, and could usually complete the tasks he undertook within a predictable timescale, at a fixed cost and with results that were satisfactory to his clients.

The programmer of today shares many attributes with the craftsman of yesterday. He learns his craft by a short but highly paid apprenticeship in an existing programming team engaged in some ongoing project, and he develops his skills by experience rather than by reading books or journals. He knows little of the logical and mathematical foundations of his profession. He does not like to explain or document his activities. Yet he works effectively, by himself or in small teams, and he sometimes manages to complete the tasks he undertakes at the predicted time, within the predicted costs, and to the satisfaction of his client.

In primitive societies of long ago, the welfare of the community depended on another class of specialist. Like the craftsman, he was dedicated to his task and regarded with respect—perhaps tinged with awe—by his many satisfied clients. Several names were given to such a man—seer, soothsayer, sorcerer, wizard, witch doctor, or high priest. I shall just call him a high priest.

There were many differences between the craftsman and the high priest. One of the most striking was that the high priest was the custodian of a weighty set of sacred books, or magician's manuals, which he alone was capable of reading. When he was consulted by his client with some new problem, he referred to his sacred books to find some spell or incantation that had proved efficacious in the past; and having found it, he told his client to copy it carefully and use it in accordance with a set of elaborate instructions. If the slightest mistake was made in copying or in following the instructions, the spell might turn into a curse and bring misfortune to the client. The client had no hope of understanding the nature of the error or why it evoked the wrath of his deity—the high priest himself had no inner understanding of the ways of his god. The best the client could hope for was to go right back to the beginning and start the spell again. If this did not work, he went back to the high priest to get a new spell.

And that brings up another feature of the priesthood. When something went wrong, as it quite often did, somehow it was always caused by the client's ignorance or stupidity or impurity or wickedness. It was never the fault of the high priest or his god. When the harvest failed, it was the high priest who sacrificed the king, never the other way around.

Present-day programmers share many attributes with the high priest. We have many names—coder, systems analyst, computer scientist, informatician, chief programmer. (I will just use the word "programmer" to stand for them all.) Our altars are hidden from the profane, each in its own superbly air-conditioned holy of holies, ministered to night and day by a devoted team of acolytes, and regarded by the general public with mixed feelings of fear and awe appropriate for their condition of powerless dependence.

An even more striking analogy is the increasing dominance of our sacred books, the basic software manuals for our languages and operating systems which have become essential to our every approach to the computer. Only 30 years ago, our computers' valves and tanks and wires filled the walls and shelves of a large room, which the programmer would enter, carrying in his pocket his programming manual—a piece of folded cardboard known as the FACT CARD. Now the situation is reversed: the programmer enters a large room whose walls and shelves are lined with software manuals, but in case he wants to carry out some urgent calculations, he carries in his pocket—a computer.

The rise of engineering

With the advance of technology in recent centuries, a new class of specialist—the professional engineer—has emerged. An engineer's most striking characteristic is the manner in which he qualifies for entry into his profession. He works out the long apprenticeship of the craftsman and undergoes the brief graduation or initiation ceremonies of the high priest, both of which are preceded by many years of formal study in schools and in universities. His education covers a wide range of topics, including the mathematical foundations of differential calculus, the derivation and solution of complex equations, and the physical principles underlying the science of materials, as well as the specific technicalities of a particular branch of his subject and a large catalog of known design methods and specific practical techniques. But this is only a start. During his professional career, the engineer continues his education, to expand his skills and to keep pace with technological progress, by studying new books and learned journals and attending specialized courses. Many engineers even take a full year off work to bring themselves up to date or to reorient themselves toward a newly developed branch of technology. The older craftsmen may complain that the engineer already knows far more than he needs for the day-to-day practice of his profession, but his colleagues and clients will realize that the weight of background learning develops his good judgment and increases his competence and authority at all times. Even if he uses a recondite scrap of knowledge only once in his career, the learning pays for itself many times over.

We would like to claim that computer programming has transcended its origins as a craft, has avoided the temptation to form itself into a priesthood, and can now be regarded as a fully fledged engineering profession. Certainly, we have some right to this claim. Through our professional societies we have formulated a code of professional ethics and a structure and syllabus of professional examinations. We discharge our duty to the community by giving evidence to

Step 7.2. Let g be the <generation> immediately contained in abuf. If abuf contains a <key>, let k be this <key> and let csvb be its immediate component; otherwise let k and csvb be <absent>. Perform construct-record (g,k) to obtain kr.

Step 7.3. If fi contains <keyed> then if k is equal to any <key> in the <record-dataset>, designated by the <dataset-designator> in fi, or if k is unacceptable to the implementation, then perform raise-io-condition(<key-condition>, fv, csvb).

Step 7.4. Perform insert-record(kr, fv) to obtain pos.

Step 7.5. Replace the immediate component of the <current-position> in fi with pos.

Step 7.6. Perform free(g) and delete abuf from fi.

Step 8. Let dd be the <data-description> immediately contained in the <variable> of the <declaration> designated by cdp. Perform evaluate-data-description-for-allocation(dd) to obtain edd.

Step 9. Perform evaluate-size(edd) to obtain an <integer-value>,int. If int is unacceptable to the implementation then perform raise-io-condition(<record-condition>,fv, chs) and optionally perform exit-from-io

Step 10. Perform allocate(edd) to obtain g.

Step 11. Let desc be a <data-description> simply containing <pointer> without other terminal subnodes. Let epsog be an <evaluated-target> containing the <generation> in the <evaluated-pointer-set-option> in els. Let agv be an <aggregate-value> containing <pointer-value>: g. Perform assign(epsog,agv,desc).

Step 12. Let d be the <declaration> designated by the <declaration-designator> in els.

Step 12.1. If the <aggregate-type> of g contains <structure-aggregate-type> then perform initialize-refer-options(g)

Step 12.2. Perform initialize-generation(g,d).

Step 13. Let abuf be an <allocated-buffer>: <generation>,g. If fi contains <keyed> then attach kk to abuf. Attach abuf to the <file- opening> in fi.

Step 14. Perform normal sequence.

8.6.4 THE REWRITE STATEMENT

Purpose: The <rewrite-statement> causes replacement of an existing <record> or <keyed-record> in a <record-dataset>.

8.6.4.1 Execute-rewrite-statement

```
<evaluated-rewrite-statement>::= <file-value>
                                 ((<key>) <evaluated-from-option>)
```

Operation: execute-rewrite-statement(rws)
where rws is a <rewrite-statement>.

Step 1. Let erws be an <evaluated-rewrite-statement> without subnodes.

Step 2. Perform Steps 2.1 through 2.3 in any order.

Step 2.1. Let f be the immediate component of the <file-option> in rws. Perform evaluate-file-option(f) to obtain a <file-value>,fv. Attach fv to erws.

Step 2.2. If rws contains a <from-option>,fr, then perform evaluate-from-option(fr) to obtain an <evaluated-from-option>,efo and attach efo to erws.

Excerpt from *American National Standard Programming Language PL/I* (American National Standards Institute, Inc., 1976, p. 403). Only the highest of high priests can interpret these hieroglyphs.

government commissions on social consequences of computing, on privacy, and on employment. Because of the great demand for our services, our clients and employers offer us professional salaries, and it is hardly likely we will refuse them.

But more than this is needed for true professional status. What is the great body of professional knowledge common to all educated programmers? Where are the reference libraries of standard works on known general methods and specific techniques and algorithms oriented to particular applications and requirements? What are the theoretical mathematical or physical principles which underlie the daily practice of the programmer?

Until recently, these questions had no answers, but now they are beginning to emerge. We can point to both the ACM curriculum for the study of computer science[1] and the IEEE Computer Society curriculum for computer science and engineering[2] as a corpus of common knowledge for the programmer, though the proportion of computer science graduates in the programming profession is still low. Don Knuth's books, *The Art of Computer Programming,*[3] form an excellent encyclopedia of known techniques, but only three volumes have appeared so far. And how many programmers consult even those? Finally, we have only recently come to a realization of the mathematical and logical basis of computer programming. We can now begin, however, to construct program specifications with the same accuracy that an engineer surveys a site for a bridge or road, and on this basis, we can now construct programs proved to meet their specification with as much certainty as the engineer's assurance that his bridge will not fall down. Introduction of these techniques, evidenced in such works as *Structured Programming,*[4] *Systematic Programming,*[5] *Principles of Program Design,*[6] *A Discipline of Programming,*[7] and *The Architecture of Concurrent Programs,*[8] promises to transform the arcane and error-prone craft of computer programming to meet the highest standards of a modern engineering profession.

Let me expand on the nature and consequences of this discovery. It is like the Greek discovery of axiomatic geometry—the basis of land measurement for mapmaking and, later, for plans and elevations used in the design and construction of buildings and bridges. It is like the discovery of the Newtonian laws of motion and the differential calculus—the basis of astronomy as well as more mundane tasks like the navigation of ships and the direction of artillery fire. It is like the discovery of stress analysis—the basis for the reliable and economic construction of steel frame buildings, bridges, and oil platforms.

Large programming projects: A view of the future

In the future, we may hope to see a radical change in the development and life history of large programming projects. The chief programmer, like the architect, will start by discussing requirements with his client. From education and experience, the programmer will be able to guide his client to an understanding of his true needs and avoidance of expensive features of dubious or even negative value. From respect for the professional status of the programmer, the client will accept and welcome this guidance. This kind of mutual understanding and respect is essential to any relationship between a professional and his client or employer.

Specification. The chief programmer at this time will sketch out the overall structure of the specification of a product to meet his client's requirements. These sketches will serve the same role as an architect's preliminary sketches of a building. Gradually, in orderly fashion and in close consultation with the client, details of the design will be slotted into the appropriate place within the structure. This activity will culminate in a complete, unambiguous, and provably consistent specification for the entire end product. It will serve the same role as blueprints in engineering or scaled plans and elevations in architecture.

The sign N means *number* (*positive integer*).

The sign 1 means *unity.*

The sign a + 1 means *the successor of* a or a *plus* 1.

The sign = means *is equal to.*

1. $1 \epsilon N$
2. $a \epsilon N \,.\supset.\, a = a$
3. $a, b \epsilon N \,.\supset:\, a = b \,.=.\, b = a$
4. $a, b, c \epsilon N \,.\supset:.\, a = b.\ b = c \,:\supset.\, a = c$
5. $a = b \,.\, b \epsilon N \,:\supset.\, a \epsilon N$
6. $a \epsilon N \,.\supset.\, a + 1 \epsilon N$
7. $a, b \epsilon N \,.\supset:\, a = b \,.=.\, a + 1 = b + 1$
8. $a \epsilon N \,.\supset.\, a + 1 \neq 1$
9. $k \epsilon K \,:.\, 1 \epsilon k \,:.\, x \epsilon N \,.\, x \epsilon k \,:\supset_x.\, x + 1 \epsilon k \,::\supset.\, N \supset k$

Mathematicians know these laws as the foundation of arithmetic and number theory today. What are the corresponding laws of programming?

Undoubtedly, the client will ask to see and check the full specification before he gives permission to go ahead with implementation. I'm afraid he will get a rude shock. Instead of pretty pictures and drawings, he will see a collection of definitions, mathematical formulas, and logical proofs, which he may be ill equipped to understand. One of the major problems of the programming profession is that our technical and structural decisions are almost invisible; nothing that can be seen in the finished program can be illustrated beforehand by pictures. This sad fact explains simultaneously the persistent longevity as well as the basic futility of program flowcharts.

A proper solution to this communication gap between programmers and clients can be discovered by analogy from other professions. Before a building project goes into implementation, the architect produces from his specification a series of perspective drawings or even models, which can be shown to the client and carefully checked by him. Before a consumer product goes into mass production, an engineer produces a series of working prototypes, which can be subjected to severe and exhaustive tests in a variety of simulated circumstances. In the future, a chief programmer, with the aid of his programming teams, will be able to pursue both of these solutions at the same time.

First, the formal specification will be taken as the basis of a clear, complete, and consistent set of user manuals and operating instructions, explaining exactly how to control the program and how it will behave in all circumstances, including when things go wrong. Of course, these manuals will be illustrated by compelling examples dealing with the main common cases, and the examples will be backed up by well-structured and well-indexed descriptions of the program's full range of capabilities. These descriptions will explain why and when the capabilities are needed, how they can be successfully invoked in conjunction with other features, what can go wrong, and how to recover from failure. These manuals will give the customer a full understanding of what his program will look like and what it will do for him, long before a single word of code is written. Because they will be firmly based on a simple mathematical model, they will be much clearer, much more complete, and much shorter than present-day manuals—just as Newton's Laws of Motion are shorter and more illuminating than the planetary observations of Tycho de Brahe.

At the same time, the chief programmer or his colleagues might construct a prototype of the program as a whole or of its more vital parts. Such a prototype could be cheaply programmed as a simulation, perhaps running on a small model of the database held in the main store of a computer much larger and faster than the one on which the eventual program will run. These simulations would be exact scale models of the final design and could be used by the client to check the details of the design and suggest alterations before the project goes into the more expensive stage of design and implementation.

The construction of models and prototypes will not be cheap but will be amply justified in a large and important project by the chance it will give to modify the design in the light of informed customer experience. Recovery from mistakes in design is much more expensive when they have been cast into the concrete of a million-line program.

Another important task could be completed at this stage. On the basis of the original requirements and formal specifications, it will be possible to devise a series of rigorous and searching acceptance tests, which could be included in the contract between the client and the implementors. Some of these tests could be kept secret from the implementors so that there would be no temptation for them to orient their work toward passing the tests, rather than meeting the specification. This rigorous kind of secret acceptance test will be made possible only by the corresponding mathematical rigor of the original specification. If the product fails the test and the implementors claim that the test is unfair, any competent logician or mathematician will be able to decide who is right.

"Computers are extremely flexible and powerful tools, and many feel that their application is changing the face of the earth. . .(But) their influence as tools might turn out to be but a ripple on the surface of our culture, whereas I expect them to have a much more profound influence in their capacity as intellectual challenge."

—E.W. Dijkstra, 1972

Implementation. The next stage will be to start work on the overall design of a program to meet the agreed and tested specification. The major components of the design will be identified and the interfaces between them defined with mathematical precision. Some of the required components will be selected or perhaps adapted from a library of existing components described in the engineering textbooks. The remaining components will be specified with the same techniques and with the same care used in the earlier design of the complete program. But most important, the chief programmer will convince himself and his colleagues by mathematical proof that if each of the components meets its specification, then when all the components are assembled, the overall product will meet the overall specification agreed to by the client. In the future, this will be taken for granted, just as we now take for granted the fact that components of a bridge ordered to given measurements will fit together when they are assembled on site. So we hope to eliminate the so-called "system integration" phase of many current projects, in which bugs are painfully detected and laboriously removed from the interfaces between the components. This is the most expensive and unpredictable of all the phases of a large project; the fact that it is the final phase only increases the misery.

Why is debugging so expensive, particularly at the stage of system integration and afterwards in program maintenance? The reason is clear: the bugs involved are so subtle that they escaped the designer's attention at a time when the design was still simple and options were still open. They also escaped the programmer's attention when he was devoting his best intellect to each line of code. Now they must be isolated in the context of a million-line program, and they must be eliminated under the additional and even more onerous constraint of changing as few of those million lines as possible! No wonder program maintenance during the whole life of a program is often many times as expensive as the original implementation. Using the new specification and design techniques of mathematics and logic, we hope to eliminate most of that cost by never creating the bugs in the first place.

When the design has progressed sufficiently, it will be possible to build teams and make plans and schedules to estimate code size and performance and, above all, to check preliminary estimates by calculating overall implementation costs and timescales. This corresponds to the activity of quantity surveying in architecture and requires experience and judgement at least as much as mathematical technique. Nevertheless, the estimates will be more accurate than they usually are nowadays because they will be based on complete, consistent, and stable specifications and designs.

At last the project will be ready to go into the construction phase. Now large teams of programmers can be engaged, perhaps from independent contractors or software houses, and all of them can work concurrently on different parts of the design without further consultation. Each programmer will use standard techniques of stepwise development to ensure that his code meets its specification with minimal risk of the intrusion of error. When he has proved that his code is correct, both the code and proof will be signed off by a highly paid checker, and the code will then be typed into a computer.

Delivery. When all the code is complete and compiled from its high-level language and loaded into the computer, it will be subjected to the implementor's tests, which it will usually pass. It will then be delivered to the customer and pass his secret acceptance tests as well. Since all manuals will have been available for training, it will go into immediate service. Nothing can possibly ever go wrong. What, never? Well hardly ever! On the rare occasion of failure, a full and independent enquiry will trace the cause of the fault to the persons responsible. An independent assessment will be made to determine whether the fault is an isolated one or whether it is a symptom of more serious and widespread flaws in the logic of the design or in the technique of implementation. In the latter case, large parts of the documentation and code and proofs will be rechecked by experts before the product is redelivered to the customer and submitted to newly constructed secret acceptance tests. The payment of appropriate penalties to the customer will ensure that this kind of default is not too frequent.

In the years after the first delivery, it is very likely that the customer's requirements will change, and the program must change with them. Because of the clarity of program structure and the completeness of design documentation, it will be quite easy to determine which parts of the design and coding need to be changed in order to meet a new requirement. Because all the assumptions and obligations of each piece of code will have been made explicit, it will be relatively easy to prove that a new piece of code which meets the same obligations can be safely inserted. If the obligations can no longer be met, it will be possible to identify all other pieces of code which rely on these obligations so that this code can be changed too. When a suggested change violates the fundamental structure of a program, the programmer will rack his brains to think of an alternative, and if he can't, he will know in advance that part or all of the program must be rewritten and check that the cost is acceptable. Thus, it will be possible to escape the wild goose chase after consequential effects of each change made to a large program that is common today.

That concludes my description of the life cycle of the large software project of the future. The description hardly makes reference to the most common feature of present programming practice, the program bug. I have left it out because it won't exist. There will be no bugs. There will be no chance for a bug to germinate or to propagate. Every stage of the specification and design and coding

"It is reasonable to hope that the relationship between computation and mathematical logic will be as fruitful in the next century as that between analysis and physics in the last. The development of this relationship demands a concern for both applications and for mathematical elegance."

—John McCarthy, 1967

will have been checked with mathematical rigor.

An essential feature of the work of a professional in any discipline is that he organize his working environment and his working methods to ensure that he does not make mistakes. Most pilots never crash a plane. Most surgeons never kill a patient. Most civil engineers never build a bridge that collapses. Until each programmer displays this kind of professional accuracy and responsibility, all our claims to professional status will be subject to doubt. Every time a member of the public blames "the computer" for an error made by a programmer, it demeans our profession. Every time a supplier of software writes a disclaimer of direct and consequential damages arising from its errors, it demeans our profession. We must always confess that it is the programmer who bears the responsibility for mistakes, not the dumb but accurate machine. We must always point out that unfair disclaimers of responsibility are (or should be) forbidden by law.

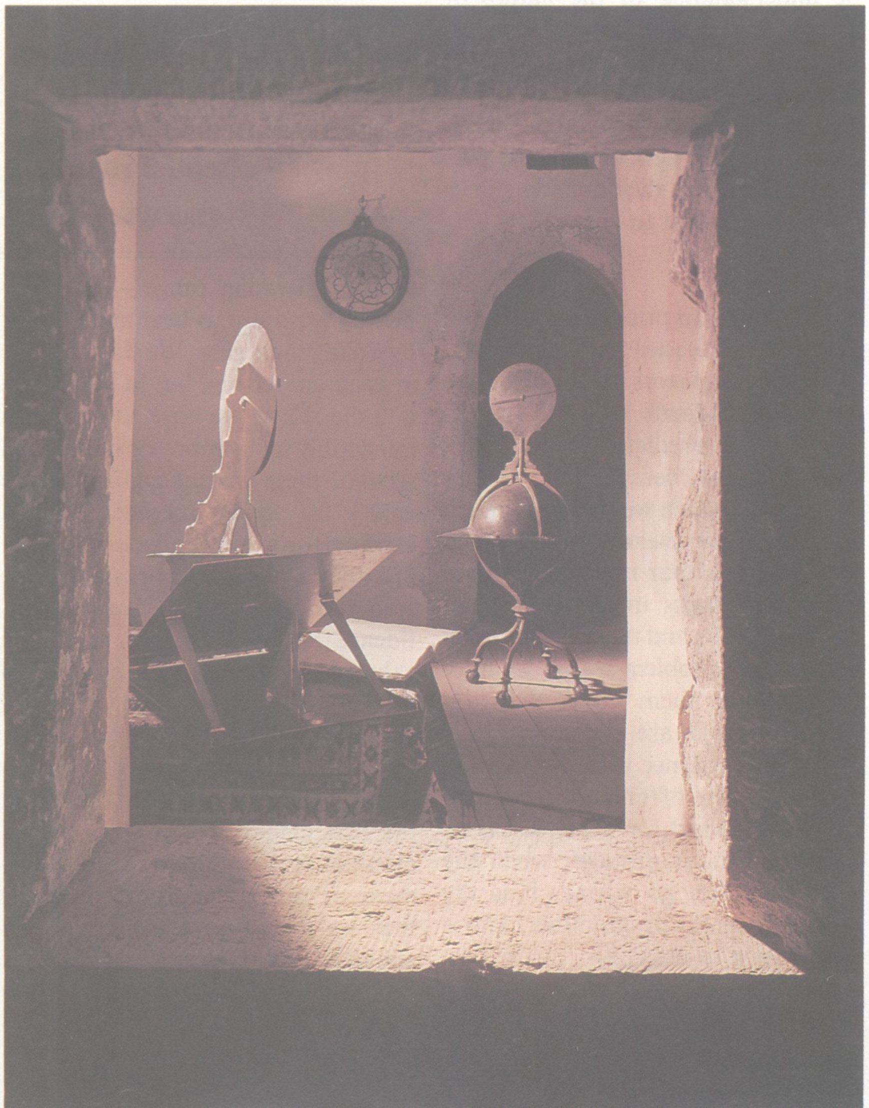
Of course, my remarks apply only to large and important projects. In smaller, less important projects, many of the stages can be merged or omitted, and for the smallest projects (for example, a program written for a single run by its own author), none of what I have said is relevant. One does not use structural engineering analysis to build a sandcastle. But neither does one choose the prize-winning builder of sandcastles as architect for a tower block of offices in a city.

Comparison with other engineering disciplines

My description of the planning of large-scale programming projects follows closely the standard practices in more-traditional branches of engineering. A conventional engineering design passes through the established phases of requirements analysis, specification, design, costing, production engineering, drawing office, prototyping, testing, tool building, quality assurance, etc. It is many years before the design reaches the production floor.

Many present-day data processing departments are organized on the basis of a similar division of labor among systems analysts, programmers, technical authors, coders, testers, and finally maintenance programmers. But all too often this apparently logical division of labor leads to an awkward problem. Gradually, the size of the maintenance programming department increases until it outnumbers all the other groups put together. And it is increasingly difficult to recruit and retain computer programmers for this boring, ill-regarded, and often poorly paid occupation. One likely cause for this problem is that the interfaces between the various groups of programmers have been less precisely defined than in a traditional engineering workshop, and that there is no proper quality control on the project documentation as it passes from one

This selection of early astronomical instruments was used at the Collegium Maius (Old University), Cracow, Poland, around 1500 when Nicolaus Copernicus, the founder of modern astronomy, studied there.
—Erich Lessing/Magnum



group to the next. As a result, each group does its best with what it gets, and it is the poor maintenance programmer at the end of the chain who has to pick up the pieces.

In my view, the standards that must be met by project documentation as it passes between groups are standards of logical accuracy and completeness, which are characteristic of mathematics. A group that takes over such documentation should have the intellectual tools required to check its validity; they also should have the right, or rather the responsibility, to reject a project that fails to meet an adequate standard. Cases of dispute should be resolved by appeal to the line technical manager, who should be experienced and capable of resolving the dispute in a technically sound fashion. It is very unfortunate that many heads of data processing departments are promoted for achievements in accounting, sales, or electronic engineering. They have little understanding of the nature of computer programming and even less of the logical and mathematical techniques required for its control. It is the managers who could benefit most from the new disciplines; perhaps that is why they are sometimes the most resistant to change.

Reliability. In principle, we should find it much easier than other professional engineers to achieve the highest standards of quality, accuracy, and predictability of timescale and cost, because the raw materials with which we work are much simpler, more plentiful, and much more reliable. Our raw materials are the binary digits in the stores and registers, disks and tapes of our computers. Our problem is that we have too many of them rather than too few. These bits are manipulated exactly in accordance with our instructions at a rate of millions of operations per second for many weeks or months without mistake; when the hardware does go wrong, it is the engineer, not the programmer, who is called upon to mend it.

That is why computer programming should be the most reliable of all professional disciplines. We do not have to worry about problems of faulty castings, defective components, careless laborers, storms, earthquakes, or other natural hazards; we are not concerned with friction or wear or metal fatigue. Our only problems are those we make for ourselves and our colleagues by our overambition or carelessness, by our failure to recognize the mathematical and theoretical foundations of programming, and by our failure to base our professional practice upon them.

Yet in some ways the engineers have an advantage over us. Because they are dealing with continuously varying quantities like distance, temperature, and voltage, it is possible for them to increase confidence in the reliability of an engineering product by testing it at the extremes of its intended operating range—for example, by exposure to heat and cold or by voltage margins. We do the same in program testing, but in our case it is futile. First, we have to deal with impossibly many more variables, and second, these variables take discrete values for which interpolation and extrapolation are wholly invalid. The fact that a program works for value zero and value 65 535 gives no confidence that it will work for any of the values in between, unless this fact is proved by logical reasoning based on the very text of the program itself. But if this logical reasoning is correct, then there was no need for the test in the first place. That is why it is an essential prerequisite to the improvement of our professional practices that we learn to reason effectively about our programs, to prove their correctness before we write them, so that we know that they will not only pass all their tests but will go on working correctly forever after.

"In order to use machines either to aid research or to aid teaching, the results, methods, and spirit of formalisation in mathematical logic are to play an essential role."

—Hao Wang, 1967

Structure. Other engineers have a further advantage over programmers. When they split a complex design into a number of component parts to be designed independently of each other, they can take advantage of the spatial separation of the parts to ensure that there can be no unexpected interaction effects. If the parts are wholly unconnected, this is very easy to check by simple visual inspection. Thus, when we turn our car to the left, we may be very confident that this will have no direct effect on the cigarette lighter, the rear-view mirror, or the carburetor. When such interaction effects do occur, they are recognized as the most difficult to trace and eliminate.

But in the programming of conventional computers, there is no similar concept of spatial separation. Any instruction in a binary computer program can modify any location in the store of the computer, including those that contain instructions. And if this happens incorrectly only once in a thousand million instructions executed, the consequences for the whole program will be totally unpredictable and uncontrollable. There is no hope that a prior visual inspection of the binary content of store will enable us to check that such interaction cannot occur or to find the cause of its occurrence afterwards. There is no structure or isolation of components in a binary computer program, other than that which has been carefully designed into it from the start and maintained by the most rigorous discipline throughout implementation.

In spite of this, the programmer is often asked to include some feature in his program as an afterthought, and the only quick way to do this is to insert new instructions which cross all the boundaries between the carefully isolated components and violate all the structural assumptions on which the original design was based. It would be repugnant to an engineer to introduce direct cross-coupling effects between the steering and carburetor of a motor car or the tapedecks and floating-point unit of a computer. A programmer is all too willing to do his best, and his profession gets a bad name when unpredicted side effects occur.

A partial solution to this problem lies in use of a high-level language like Algol 60 with secure rules governing the scope, locality, and types of variables. In such a language the programmer can declare the structure of his program and data, stating which groups of variables are to be accessed or changed by which parts of his program. An automatic compiler can then check that the appropriate disciplines have been observed throughout the whole of a large program and can, therefore, give the programmer the same confidence as the engineer gains from spatial separation of his components. Further confidence can be gained by running the program on a machine like the Burroughs 5500, which makes similar checks while the program is running. In better-established engineering disciplines, the observance of such elementary safety precautions has long been enforced by legislation. It is the law that dictates the measures that prevent unwanted interaction effects between an industrial machine and the body of its operator.

Tools of the trade. This brings me to the final disadvantage suffered by the programmer, the poor quality of the tools of his trade. I refer to his programming languages, operating systems, utility programs, and library subroutines, all of which are supplied in profusion by the manufacturer of his computer. Many of these are so complicated that mastery of them absorbs all his intellectual efforts, leaving him little energy to apply to his client's original problem. Some operating systems are so poorly designed that they require 20 reissues (or "releases"), spread over a decade, before the original design faults are rendered tolerable. And they are so unreliable that each issue has a thousand faults corrected by the next issue, which introduces a thousand new faults of its own. When the agony of reissues finally comes to an end, instead of being left free to rejoice, the poor programmer is cajoled or forced into accepting an early issue of some "new" product. Such complexity, unreliability, and instability of basic tools were doubtless endured by engineers of each newly emergent discipline, but gradually the engineers developed better tool kits for their own use. That task—the design of programming tools which are reliable, stable, convenient, and above all simple to understand, control, and use—still faces the programming profession today.

A crude measure of the simplicity of an engineering tool is the length of the manual required to give a full and complete account of how to use it and avoid misusing it. At present our software manuals are both voluminous and inadequate. I believe that a solution to our problems can be sought in the design of software which can be completely described by shorter manuals. If an electronic

Johannes Kepler, German astronomer and mathematician, died in this room in Regensburg, West Germany, in 1630. On the writing desk is his last letter about his instruments, serving as a kind of will.
—Erich Lessing/Magnum

engineer finds a method of satisfying with 20 components a need which has hitherto required 30, the value of his discovery is immediately recognized and is often highly rewarded by fame or by money. When a software engineer designs a product that can be fully defined in 20 pages whose rival product has been inadequately defined in 100 pages, his achievement is just as great and possibly more beneficial, for he has achieved an economy in our scarcest resource—not silicon, or even gold, but our own precious human intellect.

How do we get there from here?

My description of the professional achievement of programmers of the future may seem to be nothing but an academic dream—a pleasant one for our clients, but perhaps something more like a nightmare for us. How ever are we going to make such a fantastic improvement in our working methods? We are like the barber-surgeons of earlier ages, who prided themselves on the sharpness of their knives and the speed with which they dispatched their duty—either shaving a beard or amputating a limb. Imagine the dismay with which they greeted some ivory-towered academic who told them that the practice of surgery should be based on a long and detailed study of human anatomy, on familiarity with surgical procedures pioneered by great doctors of the past, and that it should be carried out only in a strictly controlled bug-free environment, far removed from the hair and dust of the normal barber's shop. Even if they accepted the validity and necessity for these improvements, how were they ever to achieve them? How could they reeducate all those hairdressers in the essential foundations of surgery? Clearly a two-week course in Structured Surgery is all that we can readily afford. But more is needed, much more.

Professional publications. First we need good books and journals which can be studied by programmers and programming teams to familiarize themselves with the concepts of mathematical proof and show how proof methods can be applied to the everyday practice of program specification, design, and implementation. (Such books are beginning to appear in the publishers' lists.[9-13])

Most of the books and articles on programming methods are of necessity illustrated only by small examples. Indeed, many of the programming methods advocated by the authors have never yet been applied to large programs. This is not a defect of their research; it is a necessity. All advances in engineering are tested first on small-scale models, in wave tanks, or in wind tunnels. Without models, the research would be prohibitively expensive, and progress would be correspondingly slow.

Nevertheless, I believe that the time has come to attempt to scale up the use of formal mathematical methods to industrial application. This can best be achieved by collaborative development projects between a university or polytechnic and an industrial company or software house. Such a project might be an entirely new program, or it might be a restructuring or redesign of some existing software product in current use, perhaps one which has lost its original structure as a result of constant amendment and enhancement. The great advantage of these joint projects is that they bring home to academic researchers some of the exigencies of working on much larger programs, and they give practical training in formal methods to larger numbers of experienced programmers in industry. This is technology transfer in its best sense—a transfer of benefits in both directions.

Education. As I have emphasized already, the major factor in the wider propagation of professional methods is education, an education which conveys a broad and deep understanding of theoretical principles as well as their practical application, an education such as can be offered by our universities and polytechnics. Lecturers and professors regard it as their duty and privilege to keep abreast with the latest developments in their subjects and to adapt, improve, and expand their courses to pass on their understanding to their students. Many entrants to computer science courses have acquired a familiarity with the basic mechanics of programming at their preparatory schools; at the university level they are ready to absorb the underlying mathematical principles, which will help them control the complexity of their designs and the reliability of their implementations.

Over the next decades, while the graduates of computer science courses are entering their profession, we will have an extremely awkward period. Almost none of the senior professionals and managers will have any knowledge or understanding of the new methods, while those whom they recruit will seem to them to be talking academic gibberish. This could be a grave hindrance to the development of our profession. Furthermore, it would be a terrible wasted opportunity. One of the major benefits of the technique of mathematical abstraction is that it enables a chief programmer or manager to exert real technical control over his teams, without delving into the morass of technical detail with which his programmers are often tempted to overwhelm him.

Systems Programmer **$80,000 plus car**

We want to hear from you if you have at least 3 years HAL programming experience with LUR—background in the LRR area would be an advantage. You'll also need in-depth maintenance and support knowledge for large-scale LUR/RD or LUR/RO systems, running with 1DR4 83.7 or 1DR4 MID in a multi-access spool configuration.

Advertisement in *IEEE Software,* 2001.

"It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it do not have any clear understanding of the fundamental principles underlying their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing. One of the central aims of the Programming Research Group as a teaching and research group has been to set up an atmosphere in which this separation cannot happen."

—Christopher Strachey, 1974

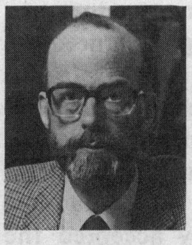
The solution to this problem is for ambitious senior programmers to make the effort now to gain the necessary mastery of the subject, thus ensuring their future effectiveness as chief programmers, technical managers, and technical directors of their companies and institutions.

One way of acquiring a professional reorientation of this kind is to take a specialist postgraduate, post-experience course in a new and important subject. Thus, an electronic engineer might now be going back to university to study VLSI design, or an industrial chemist might be taking a master's course in polymer science or genetic engineering offered by some forward-looking university or polytechnic. I believe that ambitious programmers should not be reluctant to follow the example of the well-established engineering disciplines. That is why at Oxford University we have instituted a new MSc course in computation, devoted primarily to the objective of improving programming methods and ensuring their wider application. A similar course is offered at the Wang Institute in the United States.

In 1828, on the occasion of the grant of a Royal Charter to the Institution of Civil Engineers, Thomas Tredgold defined civil engineering as "the art of directing the great sources of power in Nature for the use and convenience of man." Many branches of engineering have been established since that date. They have all been concerned with capturing, storing, and transforming energy, or with processing, shaping, and assembling materials. Computer programmers work with neither energy nor materials, but with a more intangible concept. We are concerned with capturing, storing, and processing information. When the nature of our activities is more widely understood, both within and outside our profession, then we will be deservedly recognized and respected as a branch of engineering. And I believe that in our branch of engineering, above all others, the academic ideals of rigor and elegance will pay the highest dividends in practical terms of reducing costs, increasing performance, and in directing the great sources of computational power on the surface of a silicon chip to the use and convenience of man. ■

References

1. *ACM Recommended Curricula for Computer Science and Information Processing Programs in Colleges and Universities, 1968-1981,* ACM Press, New York, 1981.
2. *1983 IEEE Computer Society Model Program in Computer Science and Engineering,* IEEE-CS Press, Silver Spring, Md., 1983.
3. D. E. Knuth, *The Art of Computer Programming,* Vol. 1, 2, and 3, Addison-Wesley, Reading, Mass., 1974 (2nd ed.), 1981 (2nd ed.), and 1973.
4. O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming,* Academic Press, New York, 1972.
5. N. Wirth, *Systematic Programming,* Prentice-Hall, Englewood Cliffs, N.J., 1973.
6. M. A. Jackson, *Principles of Program Design,* Academic Press, New York, 1975.
7. E. W. Dijkstra, *A Discipline of Programming,* Prentice-Hall, Englewood Cliffs, N. J., 1976.
8. P. B. Hansen, *The Architecture of Concurrent Programs,* Prentice-Hall, Englewood Cliffs, N. J., 1977.
9. S. Alagic and M. A. Arbib, *The Design of Well-Structured and Correct Programs,* Springer-Verlag, 1978.
10. D. Gries, ed., *Programming Methodology,* Springer-Verlag, New York, 1978.
11. C. B. Jones, *Software Development, A Rigorous Approach,* Prentice-Hall, Englewood Cliffs, N.J., 1980.
12. J. Welsh and R. M. McKeag, *Structured Systems Programming,* Prentice-Hall, Englewood Cliffs, N.J. 1980.
13. D. Gries, *The Science of Computer Programming,* Springer-Verlag, New York, 1981.



Tony Hoare, a professor of computation at the University of Oxford, worked for eight years as a programmer, manager, and research scientist with a small computer manufacturer. He is the recipient of several honors for his contributions to the study of computer programming languages and is generally famed for Hoare's Law: Inside every large program there is a small program trying to get out. He received his MA from Oxford in classical languages, literature, history, and philosophy.

The author's address is Oxford University Computing Laboratory, Programming Research Group, 45 Banbury Rd., Oxford OX2 6PE England.