Copyright Notice

The following manuscript

EWD 709: My hopes of computing science

is copyright © 1979 IEEE. Published in Proc. 4th Int. Conf. on Software Engineering, Sept. 17–19, 1979, Munich, Germany.

This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of The University of Texas's products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by sending a blank email message to pubs-permissions@ieee.org.

By choosing to view this document, you agree to all provisions of the copyright laws protecting it.
Formulas have always frightened me. They frightened me, I remember, when I was sixteen and had bought my books for the next year. I was particularly alarmed by my new book on trigonometry, full of sines, cosines, and Greek letters, and asked my mother—a gifted mathematician—whether trigonometry was difficult. I gratefully acknowledge her wise answer:

"Oh no. Know your formulas, and always remember that you are on the wrong track when you need more than five lines."

In retrospect, I think that no other advice has had such a profound influence on my way of working.

A quarter of a century later, formulas still frightened me. When I saw Hoare's correctness proof of the procedure "FIND" for the first time, I was horrified, and declared that such a ballet of symbols was not my cup of tea.

And even now my first reaction to formulas, written by someone else, is one of repulsion—in particular when an unfamiliar notational convention is used—and when reading an article, my natural reaction is to skip the formulas.

At the same time I have a warm appreciation for well-designed formalisms that enable me to do things that I couldn't possibly do without them. I acquired almost naturally my agility in the first-order predicate calculus like I had learned trigonometry 25 years earlier, and in both cases using the tool effectivly gives me great intellectual satisfaction.

I can explain this love-hate relationship only in one way. Why should I continue to shudder at the sight of formulas, whereas in the meantime I should know better? I think that, by now, I know from sad experience that only too many mathematicians and computing scientists have had the misfortune of missing my mother's wise advice at the impressionable age of sixteen. Too often the five-line limit is ignored and, instead of using the compactness of the formal notation to keep the text concise, authors use it—in a still limited space—for the introduction of much more complexity than I feel comfortable with. Hence my shudder. (I don't know how you feel about the famous Report on the Algorithmic Language ALGOL 60. I admire it very much and think its fame well-deserved. But in retrospect I think ALGOL 60's syntax, though rigorously defined, more because than is desirable, and it is certainly the compactness of BNF that has made the introduction of so much arbitrariness possible.)

Later I learned that for the kind of effectiveness that I loved, mathematicians had a perfectly adequate, technical term: they call it "mathematical elegance" or "elegance" for short. I also discovered that the term is much more "technical" than most mathematicians suspect, much more "technical" in the sense that even among mathematicians of very different brands there exists a much greater consensus about what is a really elegant argument than they themselves seemed to be aware of. Show any mathematician a really elegant argument that is new for him; at the moment it becomes his intellectual property, he starts to laugh.

The discovery of this strong consensus has made a great impression on me. It was very encouraging. It came at a moment that—in private, so to speak—I had already come to the conclusion that in the practice of computing, where we have so much latitude for making a mess of it, mathematical elegance is not a dispensable luxury, but a matter of life and death. But I hesitated to say so very much in public, just for fear of pushing another buzzword; now I dare to do it, assured as I am that mathematical elegance is a clear notion, firmly rooted in our culture. But I am also aware of the fact that my sensitivity for it can be tracked down to how I was educated in my youth. *

Language is another issue. I often feel uneasy about it. At the time I got my mother’s wise advice about trigonometry, I wrote many poems, and often I was dissatisfied: I knew that what I had written was not "it", yet I found myself unable to identify the shortcoming, and had to console myself with H.A.Milne’s "As near as you can get nowadays."
MY HOPES OF COMPUTING SCIENCE (EN7309)

My first task at the Mathematical Centre in Amsterdam was writing the precise functional specification for the computer that was then at that moment under design. I did so to the best of my ability, and thought I had done so rather well --and, from the point of precision, I had. But it was something of a shock for me to discover that, within a few days after its appearance, my beautiful report was generally known as "The Appalling Pons".

At that time I only felt that we had to learn how to write better about our own subject. I think that it was not until 1960, when Peter Neur acquainted me with Wittgenstein's famous quotation:

"We can be said to have, can be said clearly: and on what we cannot talk about, we have to remain silent."

that it slowly dawned upon me that, therefore, we had to learn how to think better about our own subject.

In the meantime I had had another linguistic shock: I became a member of the ACM, shortly before its Communications started to appear. Prior to that I had hardly had any exposure to the foreign literature. What I then read was written in a way so totally different from what I, in relative isolation, had acquired as my own habit, that I was absolutely flabbergasted. The heavily anthropomorphic terminology was totally new for me, and hardly compatible with my cultural roots; so was the jargon betrayed by the term "bug": we had never called it a bug before, we had always called it an error.

Still hesitating whether or not to adopt the jargon, I was confronted with a next term of a glaring inadequacy --was it "program maintenance"? I don't remember-- and I knew that I had to design my own way of writing about our subject in English, as I had done in Dutch. In retrospect this may strike you as a proud decision, but it wasn't: it was a decision taken in desperation, for otherwise I could not think in the way I wished to think.

One more remark about language that seems relevant. With English being computing science's Esperanto, colleagues with English as their native tongue often feel somewhat guilty about what they regard as their undeserved advantage over most foreigners. Their feeling of guilt is misplaced, because the advantage is ours. It is very helpful to have to do your work in what always remains a foreign language, as it forces you to express yourself more consciously. (About the most excellent prose written in our field that I can think of, is to be found in aforementioned ALGOL 60 Report: its editor had the great advantage of being, besides brilliant, a

Dena. I have always felt that much of the stability and well-deserved fame of ALGOL, GO could be traced down directly to the inexorable accuracy of Peter Neur's English.)

The above has been presented, by way of background information, as a help for the interpretation of what follows: a summary of my hopes of computing science. This topic is not as frivolous as it might seem at first sight. Firstly, already in my early youth --from 1940 until 1945-- I have learned to hope very seriously; secondly, my hopes of computing science --which have directed most of my work-- have shown a great stability. They evolved: some hopes have been fulfilled, some hopes, originally far away, became almost challenges as their fulfillment began to appear technologically feasible. Admittedly they are only my hopes, but they are of a sufficiently long standing that I now dare to confess them.

How were we attracted to the field of automatic computing? Why do we remain fascinated? What is really the core of computing science?

Well, everybody got attracted in his or her way. I entered the field by accident. I became attracted by the combination of the urgency of the problems that needed to be solved for the recovery of my country from the damage done to it in World War II, and the discovery that carefully applied brains were an essential ingredient of the solutions. Up till that moment I had never been quite sure whether my love of perfection had been a virtue or a weakness, and I was greatly relieved to find an environment in which it was an indispensable virtue.

I became--and remain-- fascinated by the amazing combination of simplicity and complexity. On the one hand it is a trivial world, being built from a finite number of parts and pieces, so trivial that one feels oneself mathematically entitled, and hence morally obliged, to complete intellectual mastery. On the other hand it has shown itself to be a world of treacherous complexity. The task that should be possible, but wasn't, became the fascinating challenge.

For me, the first challenge for computing science is to discover how to maintain order in a finite, but very large, discrete universe that is intricately intertwined. And a second, but not less important challenge is how to codify what you have achieved in solving the first problem, into a teachable discipline: it does not suffice to have your own intellect (that will join you in your grave), you must
teach others how to hone theirs. The more you concentrate on these two challenges, the clearer you will see that they are only two sides of the same coin: teaching yourself is discovering what is teachable.

As I said, my hopes have evolved, and one way in which they did so was by becoming more precise and more articulate. Ten years ago I expressed my dissatisfaction with the then current state of the art by aiming at programs that would be "intellectually manageable" and "understandable". At the time they represented the best way in which I could express my then still vague hope; I apologize for these terms to the extent that they became buzz-words before they had become sufficiently precise to be technically helpful. What does it help to strive for "intellectual manageability" when you don't know how you would prefer to "manage intellectually"? What guidance do you get from the goal of "understandability" before you have chosen the way of understanding? Very little, of course.

I particularly regret my use of the term "understanding", for in the combination "ease of understanding" it has added to the confusion by not inviting to distinguish carefully between "convenient" and "conventional", and that distinction, I am afraid, is vital: I expect for computing scientists the most convenient way of thinking and understanding to be rather unconventional. (That is not surprising at all, for, after all, also thinking is only a habit, and what right do we have to expect our old habits to be adequate when faced, for the first time in our culture, with a drastically novel universe of discourse?)

My hope became more articulate when programming emerged as an application area par excellence of the techniques of scientific thought, techniques that are well-known because we struggle with the small sizes of our heads as long as we exist. They are roughly of three different forms:
1) abstraction of concerns and effective use of abstraction
2) the design and use of notations, tailored to one's manipulative needs.
3) avoiding case analyses, in particular combinatorially exploding ones.

When faced with an existing design, you can apply them as a checklist; when designing yourself, they provide you with strong heuristic guidance. In my experience they make the goal "intellectually manageable" sufficiently precise to be actually helpful, in a degree that ranges from "very" to "extremely so".

For the techniques of scientific thought I called programming an application area "per excellence", and with that last term I meant two things: indispensable and very effective.

That they are indispensable seems obvious to me. They summarize the only ways in which we have ever been able to disentangle complexity more or less successfully, and we cannot expect our designs to turn out to be any better than the ways in which we have thought about them, for that would be a miracle. The indispensability of the techniques of scientific thought is, admittedly, only my belief, and you are free to dismiss it; you could remark that I am primarily a scientist and that to someone, whose only tool is a hammer, every problem looks like a nail.

I am, however, strengthened in my belief of their indispensability by the outcome of the experiments that we could take, viz. trying how effectively we could learn to apply the techniques of scientific thought. Of the experiments I am aware of, the outcome has been very encouraging. Engaged in these experiments, you start to treasure the just solvable problems, and try to present the most elegant solution you can think of as nicely as possible. I have now joined that game for several years and cannot recommend it warmly enough. It is a highly rewarding and fascinating learning process.

It is very rewarding for its immediate benefits: a significant decrease in the average amount of effort, needed to find a solution. The untrained thinker —unless a genius— spends inordinate amounts of effort in avoidable complications, and only too often, unaware of their avoidability, he falls to disentangle himself again: a vast amount of effort has then been spent on producing an inferior solution.

Our educators have something to answer for. Reading the literature, I must come to the sad conclusion, that untrained thinkers are rather the rule than the exception: people have been taught facts and tricks, but not a methodology for using their brains effectively.

The scope of the educational challenge is enormous. If you accept it—and I think we should—you have my blessing. You'll need it—end much more!—because you will encounter formidable obstacles on your way. You'll have at least two dragons to slay. Confusing "love of perfection" with "claim of perfection", people will accuse you of the latter and then blame you for the first. Furthermore, in spite of all the evidence to the contrary, the teachability of thinking effectively will be flatly denied, and your methodological contributions —needed more than anything else— will be dis-
unleashed as "for geniuses only": remember, while fighting this second dragon, that most frequently the term "genius" is not used as a compliment, but only as an alibi for the mentally lazy.

* * *

For a prosperous future of computing science—like for any science—it is essential that its achievements are published well, so that the next generation can start where the preceding one left. Above, I have expressed some of my dissatisfaction about the quality of today's publications in our field. The problem is a very serious one, and it is more than a purely educational problem. How do we publish a sophisticated piece of software? (We can reproduce the code, but that is only fit for mechanical execution. I meant "to publish" in the scientific sense; our text should fully enlighten the attentive reader.) Admittedly, many papers about algorithms could be written much better already now, but beyond a certain limit, no one knows for certain, how to do it well! And that universal inability makes it a technical problem, urgent and as yet unsolved. One of my fervent hopes is that we shall solve it.

How should a well-written publication about a sophisticated piece of software look like? We don't know yet, but two things seem certain. Firstly, the texts will be "mathematical" in the sense of Morris Kline, when he wrote:

"More than anything else mathematics is a method."

Secondly, the texts will have to be written in a style that is very different from the style of traditional mathematical texts. Depending on your mood you may regard this either as disturbing or exciting, but in any case you should be convinced of the necessity of developing a radically new style of writing mathematical texts, very unlike anything ever written before: this novelty is required by the novelty of the subject matter. Traditionally, mathematical texts are written on a fairly uniform semantic level, and that style cannot suffice for the disentanglement of the many intricate intertwinings we have to deal with.

In the relation between mathematics and computing science, the latter has, up till now, mostly been at the receiving end, and it seems to me that the time has come to start repaying our debts. Besides broadening the scope of applicability of mathematical techniques (as indicated above) we could also change their traditional applications. When, at last, the predicate calculus were to become an indispensable tool in the daily reasoning of all sorts of mathematicians, when the replacement of the asymmetric implication by the symmetric disjunction were to rob the so-called "reductio ad absurdum" from its special status and equivalence would no longer be expressed by the clumsy "if and only if", when mathematics would become enriched by a greater variety of inductive arguments, in all those cases such a development could possibly be traced down to computing science's wholesome influence.

But repaying our debt to mathematics at large is certainly not our only task: also computing proper needs our attention. We know that the problems of programming and system design are such that they cannot be solved without an effective application of the techniques of scientific thought. But how well are we able to apply them?

How well are we, for instance, able to separate the concern for correctness from the concern for efficiency? Both concerns are so "major", that I don't believe that significant progress will be possible unless we manage to separate them completely.

Efficiency has to do with cost aspects of program execution, correctness has to do with a relation between input and output, between initial and final states. Complete separation of these two concerns means that we can deal with the correctness issue without taking into account that our programs could be executed. It means that we can deal with the correctness issue, temporarily ignoring that our program text also admits the interpretation of executable code, i.e. we must be able to discuss correctness independently of any underlying computational model. To learn to dissociate our reasoning from underlying computational models, and to get rid of our operational thinking habits, that is what I regard as computing science's major task. That is what I would like to see achieved more than anything else.

Its difficulty should not be underestimated: it is like asking the average mathematician suddenly to do Euclidean geometry without drawing pictures. As Morris Kline remarks:

"But the pictures are not the subject matter of geometry and we are not permitted to reason from them. It is true that most people including mathematicians, lean upon these pictures as a crutch and find themselves unable to walk when the crutch is removed. For a tour of higher dimensional geometry, however, the crutch is not available."

The analogy is almost perfect: the pictures are to geometry what computational histories (or "traces") are to computing science, and we are not permitted to reason from them."
the operational thinking habits are firmly
rooted in many widespread traditions, ranging
from automated theory, via LISP, to FORTRAN and
BASIC, and many people, including computing
scientists, lean upon traces "as a crutch, and
find themselves unable to walk when the crutch
is removed". In the case of uniprogramming the
trace is a linear sequence of states and events,
about as manageable and "helpful" as a picture
in two- or three-dimensional geometry. But in
the case of multiprogramming traces are unman-
geable and "the crutch is not available".

For Euclidean geometry the analytical
methods of Descartes provided the alternative
to the crutch, and in analytical geometry the
generalization from three to more dimensions
was technically very smooth. In programming,
the postulation methods of Floyd and Hoare
provided the alternative to the crutch; in
uniprogramming they did so very successfully,
but their generalization from uni- to multipro-
gramming is --at the time of writing and to my
knowledge-- less smooth, although after the
successful start of Gries and Owicki I haven't
the slightest doubt that in the long run it will
be done quite successfully. The need to deline-
ate very carefully one's "point actions" is a new
aspect of the game; so is the discovery of
laws that give the implementer a greater free-
dom in embedding in space and time the activi-
ties involved in the implementation of single
point actions. (One of the ways in which we
can appreciate the manifestly greater difficulty
of designing multiprograms is that the imple-
menter is interested in much greater freedoms;
under which circumstances, for instance, is he
allowed to implement in a distributed system
a point action --in the presence of other traf-
fic? -- by an activity in node A, followed by
a "slow" message from A to B, and finally,
upon reception of the message, some activity
in B?)

Dealing successfully with these technical-
ities will, I am very much afraid, be a minor
task, compared to the educational challenge of
generating nonoperational arguments accepted
and getting people thereby out of their operational
thinking habits, for over and over again they
prove to be a mental stumbling block for accept-
ing a nonoperational argument, particularly when,
from an operational point of view, it does not
make sense. It is distressingly hard to make
someone accept a universal invariant while he
all the time remains obsessed by his knowledge
that in his implementation it will never be true,
because his implementation will never show a
moment in which it won't be halfway engaged on
one or more point actions somewhere in the net-
work. The fight against operational thinking
habits is a major educational task (of which
the crusade against anthropomorphic terminology
is only a modest beginning.)

I hope very much that computing science at
large will become more mature, as I am annoyed
by two phenomena that both strike me as symptoms
of immaturity.

The one is the wide-spread sensitivity to
fads and fashions, and the wholesale adoption of
buzzwords and even buzzphrases. Write a paper
promising salvation, make it a "structured"
thing or a "virtual" something, or "abstraction",
"distributed" or "higher-order" or "applicative"
and you can almost be certain of having started
a new cult.

The other one is the sensitivity to the
marketplace, the unchallenged assumption that
industrial products, just because they are
there, become by their mere existence a topic
worthy of scientific attention, no matter how
grease the mistakes they embody. In the sixties
the battle was needed to prevent computing
science from degenerating to "how to live with
the 360" has been won, and "courses" --usually
"in depth"! -- about MVS or what have you are
now confined to the not so respectable subcul-
ture of the commercial training circuits. But
now we hear that the advent of the microproces-
sors is going to revolutionize computing science!
I don't believe that, unless the chasing of day-
flies is confused with doing research. A simi-
lar battle may be needed.

An unmistakable symptom of maturity of
computing science would be a consensus about
"what matters" among its leaders, a consensus
that would enable us to discuss its future
course as if computing science were an end in
itself. Obviously, such a consensus can only
emerge as the byproduct of a coherent body of
knowledge and insights, but the crucial point
is "knowledge of what?" and "insights in what?"
What would be worth knowing? What would be
worth understanding?

I believe that a bold extrapolation from
the past will help us to find the answers.
When programming methodology in the early sev-
tenties adopted formal techniques for verifica-
tion and for the derivation of correct programs,
earlier ways in which programming language fea-
tures had been discussed were suddenly obsolete.
The earlier pragmatic discussions, blurred by
different tastes and habits, had only created a
confusion worthy of Babel, but then the simple
question "does this proposed feature simplify
or complicate a formal treatment?" cut as a
knife through the proposals and did more to
establish consensus than eloquence or bribery
could ever have achieved. My extrapolation
from that experience is that our knowledge
MY HOPES OF COMPUTING SCIENCE (EMT/03)

should concern formal techniques, and our understanding should be of the limits and of the potential of their application.

Let me comment in this connection shortly on two developments currently in bloom, developments that certainly fell under the heading "formal techniques": the knowledge is being developed, but about the understanding of limits and potential I have in both cases my misgivings. I mean abstract data types and program transformation, both under development in an effort to separate correctness concerns from efficiency concerns.

For the blooming of abstract data types I feel some co-responsibility, having coined and launched the term "representation abstraction" back in 1972, and if it turns out to be a mistake, part of the guilt could be mine. One hint that it might be a mistake comes from the fact that it is only a slight exaggeration to state that, after five years of intensive research and development, the stock is still the only abstract data type ever designed. This is in strong contrast to what happened when its inspirator, the closed subroutine, was invented! For this disappointingoutcome, so far, of the research devoted to abstract data types I can, at my current stage of understanding, offer two tentative explanations. The one is simply that an abstract data type is so much more complicated, so much harder to specify, than a subroutine, that it is orders of magnitude harder to invent a useful one. (If that is the case, I have only been impatient.) The other one is that, when all is said and told, the type of interface, as provided by an abstract data type, is inappropriate for its purpose: when we analyze carefully really sophisticated algorithms, we could very well discover that the correctness proof does not admit a paralleling out, such that one or two parcel s can comfortably be identified with an abstract data type. In other words, the hope that abstract data types will help us much could very well be based on an underestimation of the logical complexity of sophisticated algorithms and, consequently, on an oversimplification of the program design process.

Program transformations --each of which embodies a theorem-- have been suggested as a candidate that could contribute to the necessary body of knowledge. The hope is that transformations from a modest library will provide a path from a naive, inefficient, but obviously correct program to a sophisticated efficient solution. I have seen how via cozenous transformations striking gains in efficiency have been obtained by avoiding recomputations of the same intermediate results, even in situations in which this possibility --note that the intermediate results are never part of the original problem statement!-- was, at first sight, surprising. And yet my hope is tempered for the following reason: when, in contrast to the correctness of the naive algorithm one starts with, the correctness of the efficient one critically depends on a (perhaps deep) mathematical theorem, the chain of transformations would constitute a proof of the latter, and, to the best of my knowledge, mechanical proof verification is very cumbersome and is expected to remain so.

I am afraid that great hopes of program transformations can only be based on what seems to me an underestimation of the logical brinkmanship that is required for the justification of really efficient algorithms. It is certainly true, that each program transformation embodies a theorem, but are these the theorems that could contribute significantly to the body of knowledge and understanding that would give us maturity? I doubt, for many of them are too trivial and too much tied to program notation.

And this brings me to my final hope: before I die, I hope to understand by which virtue the one formal notational technique serves its purpose well and due to which shortcomings the other one is just a pain in the neck.

On my many wanderings over the earth's surface I learned that quite a few people tend to help me start an animated conversation by the well-intended question: "And, professor Dijkstra, what are you currently researching?". I have learned to dread that question, because it used to leave me speechless, or at least overwhelmed at best. Came the moment that I decided that I had better design a ready-made answer for it, and for a while I used the answer, both true and short: "Programming.". The usual answer I got was very illuminating "Ah, I see: programming languages.", and when I then said "No, programming.", my good-willling partner suddenly noticed that he was being corrected.

The ACM has a Special Interest Group on Programming Languages, but not one on programming as such: its newest periodical is on Programming Languages and Systems and not on programming as such. The computing community has an almost morbid fixation on program notation, it is subdivided in as many subcultures as we have more or less accepted programming languages, subcultures, none of which clearly distinguishes between genuine problems and problems only generated by the programming language it has adopted (so often as almost as a faith). For this morbid fixation I can only offer one explanation: we fully realize that in our work, more than perhaps anywhere else, appropriate notational conventions are crucial, but also: we suffer more than anyone else from the general misunderstanding of their proper role.
The problem is, of course, quite general: each experienced mathematician knows that
achievements depend critically on the availability of suitable notations. Naively one would
therefore expect that the design of suitable notations would be a central topic of mathematical
methodology. Amazingly enough, the traditional mathematical world hardly addresses the topic.
The clumsy notational conventions adhered to in many mathematical publications leave room for
only one conclusion: mathematicians are not even taught how to select a suitable notation
from the established ones, let alone that they are taught how to design a new one when needed.

This amazing silence of the mathematicians on what is at the heart of their trade suggests
that it is a very tough job to say something sensible about it. (Otherwise they would have
done it.) And I can explain it only by the fact that the traditional mathematician has always
the escape of natural language with which he can glue his formulae together.

Here, the computing scientist is in a unique position, for a program text is, by definition, for one hundred percent a formal text.
As a result we just cannot afford not to understand why a notational convention is appropriate
or not. Such is our predicament.

Nuenen, April 1979.