The fruits of misunderstanding

Some things, it seems, can only happen in the world of computing. The design of a new product is announced and the world is full of high expectations, but, when the product is ready and people start to use it, it fails to meet in a most blatant manner its main objective. It is a failure and one would expect it to be recognized as such. Nothing of the sort: the original main objective is kindly forgotten and the vast majority believes the product to be a great success.

You may have another example in mind, I was referring to COBOL, whose main objective was to make the professional programmer superfluous and which is now the main vehicle for 3 out of 4 professional programmers. In the name of "ease of programming", "readability", "understandability" its verbosity was considered a virtue; in practice that same verbosity turned out to be one of its striking defects.

COBOL was not an isolated case. When in the 60's the promise of salvation by means of "higher-level" programming languages did not materialize, the slow response in batch processing was identified as the culprit that made effecient debugging impossible, and in the name of the fast response needed for debugging, "interactive programming" was introduced on a large scale. Its spread seems to have been totally unaffected by the fact that, in the mean time, controlled

experiments had demonstrated that the beneficial effect of interactive debugging facilities on "programmer productivity" was at best marginal.

In 1968 we had the advertisement with a happily smiling Susie Mayer, who had solved all her programming problems by switching to PL/I. But when, a few years later, Harlan D. Mills and Ted Baker did their "model development" of the system for the New York Times, their team had to be strengthened by the addition of a PL/I expert.

But all this was long ago and won't happen again, because in the mean time we have discovered

(i) that interactive programming failed only because the communication channel between "end-user" and "work station" was of too narrow a bandwidth: high-resolution colour screens and voice communication will overcome that limitation;

(ii) that PL/I failed because it did not cater for the needs of the "embedded programmer" and because it was insufficiently "portable" to be acceptable as a world-wide standard; these problems are now solved by Ada;

(iii) that the now generally admitted difficulty of rigorous programming is not so important after all, since it only applies to "programming-in-the-small"; in the "real-world" the problems of "programming-in-the-large" are much more pressing and for those "software engineering" is developing "programming support environments" with enough "intelligence" and "expert knowledge".

The invention of the slogans that will explain in 1990 why dreams (i), (ii), and (iii) turned into nightmares is left as an exercise for the reader.

The brutal fact is that a large part of the history of computing can be written in terms of multi-million-dollar projects that failed. That may hold for other disciplines as well, but what seems unique to computing is that in most cases the future failure was already obvious at the moment the project was conceived. It is a long history of castles in the air, the one after the other, and the phenomenon is so persistent that it needs an explanation. How do we explain the persistent misjudgement of the technical issues involved?

The high failure rate could be explained by the assumption that the world of computing is largely populated by conscious quacks and deliberate frauds, but that is an explanation I reject because the underlying assumption is, firstly, too bitter to be healthy, and, secondly, wrong: by and large, the hopes are each time genuine. We obviously observe a wide-spread misunderstanding whose persistence is so remarkable that we cannot ignore it.

How do most people understand? What mean most people by "understanding something"? We seem to order our impressions to our satisfaction. Today the trees in our garden are very restless; also the chimney is very noisy. But these observations are neither disquieting nor disturbing. I have my satisfactory way

of coping with them: by booking them, as usual, under the heading "windy" I can almost ignore them. And the next time I am taken on a sailing trip and the ocean waves are very restless, it is fine with me: after all, what is the difference between waves of water and waves of foliage? There is no difference if it suits our purpose not to introduce it. As we grow up we acquire our habits of thought and develop our metaphors – did you notice that I described the trees as "restless"? that was unintentional! – ; they are characteristic for how we order our impressions and for how we use language.

The above tried to capture the most common way in which we seem to cope with novelty: when faced with something new and unfamiliar we try to relate it to what we are familiar with. In the course of the process we invent the analogies that enable us to do so.

It is clear that the above way of trying to understand does not work too well when we are faced with something so radically new, so without precedent, that all analogies we can come up with are too weak and too shallow to be of great help. A radically new technology can create such circumstances and the wide-spread misunderstanding about programming strongly suggests that this has happened with the advent of the automatic computer. (Nuclear weapons could very well provide another example of a discontinuity with which thinking in analogies cannot cope

adequately.)

There is another way of approaching novelty, but it is practised much more rarily. Apparently it does not come "naturally" since its application seems to require a lot of training. In the other way one does not try to relate something new to one's past experience -aware of the fact that that experience, largely collected by accident, could very well be inadequate-. Instead, one approaches the novelty with a mind as blank as possible and tries to come to terms with it on account of its own internal consistency. (It is like learning a foreign language without trying to translate back and forth to one's native tongue. This is clearly one's only hope if the two languages are only adequate for expressing very different thoughts: one has to learn to "think" in the new language.) My guess is that computing is such a novelty that only this second approach is viable, and that, consequently, we must try to free ourselves in this case from the habit of approaching the topic in terms of analogies.

To ease that process of liberation it might be illuminating to identify the most common metaphors and analogies and to see why they are so misleading.

I think anthropomorphism is the worst of all. I have now seen programs "trying to do things", "wanting to do things", "believing things to be true", "knowing things" etc. Don't be so naive as to believe that this use of language is harmless. It invites the programmer to identify himself with the execution of the pro-

gram and almost forces upon him the use of operational semantics.

We should never try to come to grips with a huge set by looking at "enough" of its members, because that is so hopelessly ineffective. Ideally, the individual members disappear from the consideration and one deals with the definition of the set itself. In the case of understanding a program, we should not look at individual computations, ideally – i.e. for the sake of mental efficiency – we should deal with the program in its own right, temporarily ignoring that it also permits the interpretation of executable code. All this is well-known – it is called The Axiomatic Method –. The point to be made here is that the use of anthropomorphic terminology erects a barrier for its application. One of the fruits of anthropomorphism in computing is that the most effective way of developing programs and reasoning about them is rejected on such grounds as "counter-intuitive", or "too abstract" and that is very serious.

I skip the numerous confusions created by calling programming formalisms "languages", except a few examples. We have had the general who wrote "Obviously, NATO is not interested in artificially simplified programming languages such as PASCAL." It also gave birth to the idea that a "living" programming language is better than a "dead" one. Not so long ago a branch of a large industry had received from the central facility of the company a program it had asked for, but the

program had been written in PASCAL. The branch had an external software house translate it into FORTRAN "because PASCAL was not maintained".

And now we have the fad of making all sorts of systems and their components "intelligent" or "smart". It often boils down to designing a woolly man-machine interface that makes the machine as unlike a computer as possible: the computer's greatest strength - the efficient embodiment of a formal system - has to be disguised at great cost. So much for anthropomorphism. (This morning I declined to write a popular article about the question "Can machines think?" I told the editor that I thought the question as ill-posed and uninteresting as the question "Can submarines swim?". But the editor, being a social scientist, was unmoved: he thought the latter a very interesting question too.)

Another analogy that did not work was pushed when the term "software engineering" was invented. The competent programmer's output can be viewed as that of an engineer: a non-trivial reliable mechanism but there the analogy stops.

In contrast to the traditional engineer's mechanism, made of physical components, a program is an abstract mechanism, so to speak made from zeroes and ones alone. And this difference has profound consequences.

To begin with, a program is not subject to wear

and tear and requires no maintenance. Yet the term "program maintenance" established itself, only adding to the confusion.

Secondly, the mechanism being abstract, its production is subsumed in its design. In this respect a program is like a poem: you cannot write a poem without writing it. Yet people talk about programming as if it were a production process and measure "programmer productivity" in terms of "number of lines of code produced". In so doing they book that number on the wrong side of the ledger: we should always refer to "the number of lines of code spent".

Thirdly, in the case of a physical mechanism, higher quality, greater reliability, greater precision, etc. always induce higher cost. In the case of programs the correlation is the other way round since unreliability and high cost stem from the same source, viz. unmastered complexity. Yet, unreliability of commercial software is often defended by the remark that making the software better would have been too expensive.

Finally, the engineer's standard design paradigm does not work. He designs a thing to the best of his abilities, makes a prototype and tries whether it works satisfactorily. If not, he improves the design, and "repetatur". There are circumstances under which such iterative design is defensible; a necessary condition is, however, that the feedback loop can, indeed, be closed. And we know that with

programs this is impossible, except with utterly trivial ones. For many a manager, for whom iteration by means of prototypes is the only design paradigm, this impossibility is so disconcerting that he is unable to see it. But is it a miracle that so-called "software engineering" became a movement devoted to "how to program if you cannot"?

The pre-computer object that offers the closest analogy to a well-designed piece of software is an equally well-designed mathematical theory. But also this analogy has its problems. It has a social problem: it is not very popular. Computing systems are sold to people that are expected to consider anything mathematical as the pinnacle of user-unfriendiness. It has, however, a more technical problem too.

The bulk of traditional mathematics is highly informal: formulae are not manipulated in their own right, they are all the time viewed as denoting something, as standing for something else. The bulk of traditional mathematics is characterized by a constant jumping back and forth between the formulae and their interpretation and the latter has to carry the burden of justifying the manipulations. The manipulations of the formulae are not justified by an appeal to explicitly stated rules but by the appeal to the interpretation in which the manipulations are "obviously" OK. By and large, the mathematicians form a much more informal lot than

they are aware of.

Whether this informality is in general "good" or "bad" is neither here nor there. (There exists a type of mathematician with a puritan streak that feels guilty about it.) The point is that in most of computing this informality seems inappropriate when you wish to approach the topic mathematically. For this inappropriateness there are two reasons.

Firstly, by its very nature, each computing system embodies a formal system of some sort. We can of course (try to) hide this fact by giving it so many complicated -and ill-documented- properties that in its capacity of a formal system it becomes an impossible one to use. Often this has happened by accident, sometimes this is done on purpose, the hope being that it becomes a suitable tool for informal use: such efforts can be characterized as trying to use a computer for the simulation of a non-computer and can be ignored in this context.

Secondly, the intrinsically discrete nature of symbol processing makes programming such a tricky job that -again: when we wish to approach the topic mathematically- the application of formal techniques becomes a necessity. For safety's sake the reasoning has to be carried out mostly by rule-governed manipulation of uninterpreted formulae: the intuitive justification of these manipulations becomes for combinatorial reasons too error-prone.

There is a further aspect in which computing science, when regarded as a branch of mathematics, differs from most of the pre-computer mathematics. Traditional mathematics has subdivided itself into many different branches, each with its own specific body of knowledge. (This to the extent that, by now, the complaint that mathematicians from different branches can no longer communicate with each other is a very common one.) As branch of mathematics, computing science differs from the others in that the size of its body of relevant mathematical knowledge is relatively small. There was a time when I blamed the topic's youth for that circumstance and felt that the building up of such a body of knowledge would be a worthy goal. It never acquired, however, the status of one of my explicitly stated purposes in life, and, as time went by, it just faded away. The relative irrelevance to the core of the topic of much specific knowledge is, in retrospect, quite well understandable: it is a direct consequence of the fact that computers are truly "general-purpose" equipment. Extending the body of knowledge faded away as a purpose, when developing the "quo modo" began to absorb much more of my attention than developing the "quod". (I never felt attracted by the description of education as "knowledge transfer"; now I know why at least in the case of computing science that term is even very inadequate.)

If - as I am beginning to believe - the relatively modest rôle of knowledge is intrinsic to the topic,

this will be reflected in our way of teaching the topic. What nowadays is called "methodology" will play a larger rôle in the teaching of computing science than it has done so far in the traditional teaching of mathematics. Since, all over the world, the practice of mathematics is so tightly interwoven with its teaching, this last aspect in which computing science differs from most of the rest is felt very much. It is a difference that cannot be glossed over. As a consequence of it, many computing scientists are hesitant to stress the mathematical nature of their topic for fear that this will be misunderstood as a plea for "more analysis, more statistics, more algebra and group theory" in the curricula, while what they meant was more rigour, more elegance, more hygiene and more convincing logic.

And here we have the full scope of our difficulties. With yesterday's words we have to plan today for tomorrow, and, the computing challenge being without precedent, the words are no good. If we don't coin new terms, we have to give new meanings to old words. Regrettably, the world of computing seems better at coining new terms for old meanings (or without any meaning at all).

Plataanstraat 5
5671 AL NUENEN
The Netherlands

19 May 1983
prof. dr. Edsger W. Dykstra
Burroughs Research Fellow