Two views of programming.

by Edsger W.Dijkstra.

In the world around us we encounter two radically different views of programming:

View A: Programming is in essence very easy.

View B: Programming is intrinsically very difficult.

One can dismiss this discrepancy by concluding that, apparently, in the two views the same word "programming" is used in two quite different meanings, and then return to the order of the day. Whether view A or view B is the predominant one, however, has a deep influence, not only on the personel policy of computer using organizations and on the curriculum policy of our educational institutes, but even on the direction of development and research in computing science itself. It seems, therefore, worthwhile to explore the nature of the difference between the two meanings and to identify, if possible, the underlying assumptions that would make each of them appropriate. To do so is the purpose of this paper.

In this exploration I have what could be regarded as a handicap: in the controversy I am not neutral at all. I am a strong supporter of view B and regard view A as the underlying cause of many sad mistakes. On the other hand I don't think that having an opinion disqualifies me as an author, certainly not if I warn my readers in advance and do not feign a fake neutrality. As our analysis proceeds we shall discover how these different views of programming (which is a human activity!) are related to different views of Man. This, all by itself, is already a valuable insight, as it explains the nearly religious fervour with which the battle between the defenders of the opposing views --creeds?-- is sometimes fought.

* * *

The early history of automatic computing makes view A only too understandable. Before we had computers, programming was no problem at all. Then came the first machines: compared with the machines we have now they were mere toys, and, compared with what we try to do now, they were used for "micro-applications" only. If at that stage programming was a problem,

it was only a mild one. Add to this the sources of the difficulties that at that time absorbed --or should we say in retrospect: usurped?-- the major part of our attention:

1) arithmetic units were slow with respect to what we wanted to do: that shoe pinched nearly always, and in the name of program efficiency all possible coding tricks were permitted (and very few were not applied)

2) design and construction of arithmetic units were such a novel and, therefore, difficult task that if a next anomaly in the instruction code could save a number of flip-flops, the flip-flops were usually saved --also, of course, because we had so little programming experience that we could not recognize "anomalies in the instruction code" too well-- ; as a result there was, besides pressure to apply coding tricks, also a great opportunity for doing so

3) stores were always too small, a pinching shoe that, together with the general unreliability of the first hardware, prohibited more sophisticated ways of machine usage.

In that time programming presented itself primarily as a battle against the machine's limitations, a battle that was to be won by a cunning, be it not very systematic, exploitation of each machine's specific properties: it was the heyday of the virtuoso coder.

In the next ten to fifteen years processing units became a thousand times faster, stores became a thousand times larger, and high-level programming languages came into general use. And it was during that period, when on the one hand programming was still firmly associated with the pinching shoe, while on the other hand the shoe was felt to pinch less and less, that it was expected that with another five years of technical progress the problems of programming would have disappeared. It was during that period that view A was born. It was at the end of that period that, inspired by view A, COBOL was designed with the avowed intention that it should make programming by professional programmers superfluous by allowing the "user" --was it at that time that the term "user" gained currency?-- to write down what he wanted in "plain English" that everyone could read and understand.

We all know, that that splendid dream did not come true. The next five years gave us, instead of the disappearance of all programming problems, the software crisis, and COBOL, instead of doing away with professional programmers, became the major programming vehicle for ever growing numbers of them; and another ten years later we still have machines with which flaws in the basic software cause on the average one hour down-time for every fifteen hours of production. There are, apparently, still serious programming problems....

The amazing thing is that, in spite of the overwheling evidence to the contrary, view A survived. In explaining that amazing fact some point with a somewhat accusing finger to large organizations, either to computer using ones that, having attracted a large labour force based in view A , thereby have lost the freedom to part from it, or to computer manufacturers and educational institutes that promote a widely-held view A , that they are supposed to suppose essential for their market. Even if that finger is not raised without some justification, I just cannot accept it as a full explanation of view A's survival, and must assume that view A satisfies some deeper, psychological needs.

* * *

How emerged view B ? There were people who felt that the advent of faster and bigger machines would replace the pinching shoe at most by a fitting shoe, and that, therefore, the economics of program execution would remain a serious concern of the programmer, a concern that would even become more important as size of machines and applications would grow, and, with more complex installations, would pose more difficult problems. Also it was observed that switching from machine code to a high-level programming languages did not guarantee all the benefits that were hoped for. In particular, programmers still produced, as willingly as before, large chunks of ununderstandable code, the only difference being that now they did it on a more grandiose scale, and that high-level bugs had replaced low-level ones. They also realized that the advent of high-level programming languages had not reduced the essential need for accuracy: redundancy in high-level programming languages only reduces the ill effects of some inaccuracies. And thus view B was born. (View B is <u>not</u> the reaction to the software crisis

that surfaced in 1968, for it is many years older. View B had, in fact, predicted that crisis, but even that confirmation has not killed view A .)

* * *

After this interlude about the emerging of view B , we return to our question, how and why, face to face with the undeniable software problems, view A , viz, that programming is in essence very easy, survives. The answer is: by faith, not faith in better programmers, but faith in better programming languages or (conversational?) programming systems, and faith in better management techniques.

I happen to be of the considered opinion that programming is one of the more difficult branches of applied mathematics, because it is also one of the more difficult branches of engineering, and vice versa. When I tried to explain to one of my mathematical colleagues why I held that view, he bluntly refused to listen to my arguments and, instead, blamed me and my fellow computing scientists for not yet having designed the programming language that would make programming as easy as it should be! Should I have asked him, why mathematicians have not yet developed a notation that would enable everyone, no matter how ill-equipped otherwise, to do mathematics?

After more probing one discovers that the proponents of view A do not deny the potential complexity of programs, nor the problem of composing them, but believe that life will become easy for the programmer because all the more difficult parts of the task will eventually be taken over by the machine. They point to the advent of high-level programming languages that made programming already so much easier than in the old machine code days, and recklessly extrapolate, that in future programming will become trivial. But is this extrapolation justified? I have programmed extensively, both in machine codes and high-level programming languages, and the latter are undoubtedly more convenient because all sorts of otherwise irrelevant decisions with many clerical consequences --such as details of storage layout-- need not be taken explicitly because one accepts the outcome of the compiler's storage allocation algorithm. The transition to a high-level language freed us from a number of trivia. In doing so, it has made programming an activity with a smaller component of drudgery, and therefore with a larger component

of invention: precisely those parts of the job with which one used to fill the the day when feeling less bright, are the ones that have disappeared! The conclusion that the advent of high-level programming languages has created the need for programmers of higher intellectual caliber has been fully confirmed by my observations in Western Europe (where I could follow the development at close quarters) where in the late sixties many large computer using organizations had problems in finding appropriate employment for the programmers they had attracted in the fifties, because their profession had outgrown their intellectual capacities.

But neither this observation, nor pointing out COBOL's failure to do away with professional programmers, makes any impression upon the faithful. They will explain that the traditional high-level programming languages were failures because they were still "procedural", and that COBOL's failure is evident because, for lack of interaction, it is not really plain English, but that within five or ten years further progress in Artificial Intelligence (AI for the intimi) will enable us to build "context dependent","knowledge-based","automated systems for reasoning and understanding" such that the "user needs only to talk to them".

I may be an incurable sceptic, but I find it very hard to believe such claims to be justified. They are specific instances of the expectation that we shall witness --I quote from a letter I recently received-- ", in general, the assumption by the computer of progressively higher levels of what is now considered human skill, knowledge and intelligence." It is not my intention to repeat parts of the heated discussions we have had about the significance of Artificial Intelligence (see, for instance, [1]), nor is there any need to do so.

Firstly, looking backwards, one conclusion is certain: confusing AI's hopes for the future with tomorrow's reality would be folly, and it would be an act of utter irresponsibility not to prepare ourselves for the case that the dreams of AI will remain dreams as long as we live. Or, to put it in other words, in view of the severity of today's programming problems, general cautiousness forces us not to discard view B .

Secondly, if his creations are going to be relied upon, it will be the programmer's primary task to design his artifacts so understandable, that he can take the responsibility for them, and, regardless of the answer to the question how much of his current activity may ultimately be delegated to machines, we should always remember that neither "understanding" nor "being responsible" can properly be classified as activities: they are more like "states of mind" and are intrinsically incapable of being delegated.

* * *

I think it unwise, particularly for a computing scientist, to underestimate the influence of that school of psychologists that, because they found the human mind to difficult and elusive an object for their study, turned to the study of rats instead, and even restrict that study --as I saw expressed recently-- "to the most mechanical forms of behaviour --often so mechanical that even rats have no chance to show their higher faculties--". By presenting their crude, mechanical models as a valid approximation for the human mind, they have blurred the distinction between man and machine dangerously, and we observe the two complementary phenomena: an anthropomorphic view of machines and a mechanical view of people.

This confusion is by no means confined to the high priests of AI. The preponderance of anthropomorphic terminology in computing science in general --"memory", "interpreter", "programming language", "handshaking", "conversation", to mention but a few-- is a warning not to be ignored. I would not know how to think and talk without metaphors; I also know that each metaphor carries with it the danger of false connotations. In the case of the anthropomorphic terminology in computer science we have since long reached the stage that the dangers of confusion far outweigh the benefits of the analogy.

Also the mechanical view of people seems among computing scientists (and their managers) more widely spread than I can consider healthy. For I suspect that it is this mechanical view that restricts the activity of programmers to the mechanical activity if writing code, and then measures "programmer productivity" by the number of lines of code produced. (When a very well-known and widely respected computing scientist recently used that measure for programmer productivity in a lecture, the suggestion came

from the audience that, instead of talking about "the lines of code produced", we should talk about "the lines of code used", and that, therefore, the speaker was booking them on the wrong side of the ledger. The speaker answered that he stuck to his productivity measure, because he did not know of any alternatives that allowed proper quantification!) This can no longer be classified as a harmless mistake, for the adoption of that nonsensical "productivity measure" for grading programmers is guaranteed to promote the writing of insipid code.

* * *

The influence of psychology has been brought into the picture because it explains the tenacity with which so many people cling to view A .

It is not so much the computer manufacturers, that want to do as if they sell an easy product; it is not so much the managers of software projects, that would like to view the programming activity as a simple and predictable one; it is not so much our educational institutes, that would like to train their students with predictable success.

It is the comfortable illusion of Men as elaborate automata that, like a drug, seems to have freed its victims from the burden of responsibility. Accepting programming as a hard intellectual challenge would place the full weight of that burden back upon their shoulders.

[1] Flowers, B.H., "Artificial Intelligence: a paper symposium" April 1973, Science Research Council, State House, High Holborn, London WC1R 4TA .

Plataanstraat 5
NL-4565 NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow