EWD273.html

The Programming task Considered as an Intellectual Challenge

Edsger W.Dijkstra

In contrast to my original intention to address you without the use of written notes, I have decided to read the text to you verbatim as it now lies before my very eyes. The chosen procedure often leads to a less lively, if not even soporific, performance; I can only express my hope that the duller presentation of my talk is somewhat compensated for by its contents ..... in one way or another. I feel it my duty and pleasure to warn you that this time I shall make full use of the traditional prerogative of the speaker, viz. to say what he wants to say, rather than to tell what his audience might want to hear. I am well aware of the fact that in consequence of this decision I might never be invited again, but if that would be for the loss or benefit of our confession is not for me to judge .....

Let me now mention and describe to you the experience which led to my decision to address you in the manner just announced. It was the second "Conference on Software Engineering", sponsered by the NATO Science Committee and held in Rome during the last week of October 1969. Had I been brought up in the tradition of the British understatement, I would probably have described that conference as "not completely successful"; now, however, the first expressions coming into my mind to describe it, are rather "an utterly miserable affair" or "a wet mess".

In one respect, I am afraid, it was an honest conference: its sorry state may have been a true reflection of the present state of the art of software engineering as generally understood and practised. But before trying to trace possible origins of this sorry state of affairs, let me tell you what happened at the last day of the conference. After that, you can judge for yourself. The incident has been witnessed by some ten or twenty people.

Around five o'clock, just before the closing of the conference, I saw a pile of reproduced notes written by one of the participants,

who handed a copy to me when he saw that I showed some signs of interest. Standing next to him I started to scan his notes, only to see to my horror that also this document, alas, contained nothing more meaningful than the pompous, superficial bla-bla-bla that we had been exposed to all through the week. My disgust got the better of me, I was seduced to rudeness and after one and a half page I put the paper back on the pile and turned myself, without any further comments, away from its author. End of the first act of this drama.

When the curtain rises for the second act it is four hours later, nine o'clock in the evening. Guilt about my rude behaviour is gnawing at my mind and to relieve it I pick the paper up again to reread it; this time I start at the end, planning to read it backwards. Then, at last, my eyes are opened: the paper has carefully worked towards a climax and this last sentence is glorious and superb nonsense, I suddenly discover that I had nearly been fooled by an elaborate leg pull: it was a parody, it was satire, produced by a kindred spirit! Full of excitement I show the document (that very few had seen) to the remaining participants that were still in the hotel; then people are reading it, smiling, chuckling or roaring with laughter, according to their temperament. But some don't laugh at all because they don't believe that it is satirical. In the argument that follows, the question "Is this a joke or is this serious?" remains unanswered and leaves the group divided into two camps. End of second act.

In the third act the author, who had been absent during the second act, returns and tells that this document had not been written as a parody, that he had recorded the conference as he had seen it. End of drama.

My conclusion is that when a number of people who are recognized as knowledgeable in the field cannot settle among themselves whether an apparently professional paper is serious or not, that this is alarming, that the present state of the art is indeed a sorry state of the art.

Another unmistakable symptom of the conference's misery was that some people, in reaction to it, changed their air line reservations and left before the conference had run to completion. I myself would have done so, if such a change of schedule would not have caused inconveniences to some other private arrangements I happened to have made.

I hope that the above will be accepted as a proper documentation for establishing, even beyond the mildest doubts, the existence of the misery. If I left it at that, I would indeed do a poor job, so my next step will be a somewhat closer scrutiny of the symptoms of the illness in order to see whether we can find at least some of its causes.

The most obvious symptom of the misery was that most of the discussion was dragged into the vulgar, but nonetheless rather fruitless controversy between "theory" and "practice". I know, and of course all of us know, that the qualifications "Theoretical" and "Practical" with exclusion of the other are applicable to quite a large percentage of the workers in the field. There are people who refuse to have anything to do with practical applications. There is the story of the man who was preparing his PhD Thesis under Hardy and the subject had to do with Fourier integrals; one day he was so reckless as to mention to Hardy that some people really used Fourier integrals, upon which Hardy answered that under those circumstances the man had to look for another subject. At the other side of the deplorable fence we have those that have given up hope of ever finding anything useful in the work of their more theoretically inclined colleagues; and of course, quite a lot of it is utterly useless. Although the distinction between "theoretical" and "practical" people may be applicable to, say, 95 percent of the workers in the field and the distinction can therefore be regarded as statistically significant, yet I call it fruitless, because it leaves no room for exploring what benefits could be derived from the remaining five percent, i.e. those people and those activities for which neither the qualification "untheoretical" nor the qualification "unpractical" can be justified.

To ignore those five percents is a simplification of though which throws away the child with the bathwater, and that is why I call the distinction vulgar and fruitless.

To attack the fence in another way: I am employed by a Technological University and educational product of my department gets the title of "Mathematical Engineer". I have often observed that many, in particular Americans, start laughing when they hear that title: it strikes them as a contradictio in terminis, for by definition they regard the engineer as useful and the mathematician as useless. I speak from another cultural tradition - otherwise I would not have undertaken my job. This implies that I get extremely suspicious when the engineer justifies adhoccery by an appeal to the presumed law of nature, summarized by "quick and dirty", for from my experience and from my understanding I feel that "quick and elegant" is a much more likely combination. I get suspicious when the mathematician regards the formality of approach as a holy goal all by itself, rather than as a profane means which might be useful (and occasionally even vitally effective). Roughly speaking, a formal treatment is related to my power of understanding as an attorney's act to my sense of justice.

From now onwards I refuse to make that vulgar distinction, I hope that I have made my porision quite clear and that I have given everyone his equal share and, by the way, I am vaguely wondering whether I have turned the 95 percent into my friends or my foes .....

OK, that was my introduction. Let me now start the lecture proper.

The starting point of my considerations is to be found at the "software failure". One or two years ago the wide-spread existence of this regrettable phenomenon has been established beyond doubt; as far as my information tells me, the software failure is still there as vigourous as ever and its effects are sufficiently alarming to justify our concern and attention. What, however, is it?

Depending on the specific instance of failure one chooses, it can be described in many, many ways. One of the most common forms starts

with an exciting software project, but as it proceeds, deadlines are violated and what started as a fascinating thriller slowly turns into a drama, to be played by an ever-increasing number of actors, the majority of which know perhaps their own part but have certainly lost their grasp on the meaning of the performance as a whole. At last the curtain falls, only because it is too late to go on any more, but not because anything has really been completed, for the final piece of software is still full of bugs and will remain so for the rest of its days. There are other forms, but they all have in common, that it turns out to be very, very difficult to get the whole program working with an acceptable degree of reliability.

When we try to explain the present as the natural outcome of the recent past, we get a better understanding of what has happened. In the past ten, fifteen years, the power of commonly available computers has increased by a factor of a thousand. The ambition of society to apply these wonderful pieces of equipment has grown in proportion and the poor programmer, with his duties in this field of tension between equipment and goals, finds his task exploded in size, scope and sophistication. And the poor programmer just has not caught up. Looking backwards, we must conclude that the difficulty of the tasks ahead have been grossly underestimated in the past. Extrapolations concerning the numbers and power of computers to be installed have been made, but society's preparation for this oncoming wave of machinery has been the call for more and more programmers, rather than for more capable ones, who could derive their greater capability from a better understanding of the nature of the programming task.

How little the general programmer's attitude towards his work has changed during this period can be demonstrated in a variety of ways. During that NATO conference in Rome a number of debugging aids were discussed; it was then remarked by one of the participants that all the techniques mentioned were already known and used fifteen years ago, a statement that no one did deny! Secondly, the advent of higher

level languages has been hailed as a tremendous step forward. OK, but is it sufficient? It enables us perhaps to cope with a factor 10 in scope, but not with a factor 1000. In the old days, programs of one or two thousand assembly code instructions were horrors, but in the mean time higher level language programmers produce - admittedly - larger programs of exactly the same degree of unreadability and unreliability, in which the role of the old machine-code tricks has been taken over by cunning higher level language tricks. Truly, I cannot see the difference ..... The conclusion is that, in spite of the factor thousand in scope, present day programming tries to solve its problems essentially with the same old methods. And therefore, if we want to improve matters, we should make it our serious business to minimize the usage of what is now by far our scarcest resource, viz. brainpower. The burning question is "Can we get a better understanding of the nature of the programming task, so that by virtue of this better understanding, programming becomes an order of magnitude easier, so that our ability to compose reliable programs is increased by a similar order of magnitude?"

The fact that program reliability becomes a key issue is not only shown to us by the evidence around us, it is also quite easy to see why. A very large program is, by necessity, composed from a large number, say N, individual components and the fact that N is large implies that the individual program components must be produced with a very high confidence level. If, for each individual component the probability of being right equals p, for the whole program the probability P of being right will satisfy

$$P \leqslant p^N$$

and if we want P to differ appreciably from zero, p must be very close to one, because N is so large.

A common approach to get a program correct is called "debugging" and when the most patent bugs have been found and removed one tries to raise the confidence level further by subjecting the piece of program

to numerous test cases. From the failures around us we can derive ample evidence that this approach is inadequate. Believe it or not, but it has been suggested (at that very NATO conference in Rome) that what we really need now are "automatic test case generators" by which pieces of program to be validated can be exercised still more extensively. But will this really help? I don't think so.

When faced with a mechanism - be it hardware or software - one can ask oneself "How can I convince myself of its being correct?" As long as we regard the mechanism as a black box, the only thing we can do is to subject the mechanism to all possible inputs, all the time checking whether it produces the correct outputs. But for the kind of mechanisms we are considering this is absolutely out of the question, even for the fastest and simplest mechanisms. At my University we have a machine with a fixed point multiplier taking considerably less than 100 microseconds per multiplication. The total time taken by all possible multiplication, however, will exceed 30000 years. And that is only a multiplier! The number of cases one can try in practice is a negligeable fraction of the total number of possible cases and whole classes of in some critical cases can be missed. The first moral of this story is that program testing can be used very effectively to show the presence of bugs, but never to show their absence.

But as long as we regard the mechanism as a black box, testing is the only thing we can do. The conclusion is that we cannot afford to regard the mechanism as a black box, i.e. we have to take its internal structure into account. One studies its internal structure and on account of this analysis one convinces oneself that if such and such cases work "all others must work as well". That is, the internal structure is exploited to reduce the number of still necessary testcases, for all the other ones (the vast majority) one tries to convince oneself by reasoning, the only problem being that the necessary amount of reasoning often becomes excessive.

This function of the mechanism's internal structure opens a new way to attack the reliability problem. Once we have seen that the confidence level can only be reached by virtue of the structure of the program, that the extent to which the program correctness can be established is not purely a function of its external specifications and behaviour, but depends critically upon its internal structure, then we can invert the question and ask ourselves "What forms of program structuring can we find, what elements of programming style and what forms of discipline, all for the benefit of the confidence level of our final product?"

Instead of trying to devise methods to establish the correctness of arbitrary, given programs, I am now looking for the subclass of "intellectually manageable programs", which can be understood and for which we can justify our belief in their proper operation under all circumstances without excessive amounts of reasoning. This is done in order to reduce the number of testcases needed; in the case of software I see no reason at all, why this approach could not be so effective that the number of testcases needed is eventually reduced to zero, i.e. that correctness can be shown a priori. Already now, debugging strikes me as putting the cart before the horse: instead of looking for more elaborate debugging aids, I would rather try to identify and to remove the more productive bug-generators!

In short, I suggest that the programmer should continue to understand what he is doing, that his growing product firmly remains within his intellectual grip. It is my sad experience that this suggestion is repulsive to the average experienced programmer, who clearly derives a major part of his professional excitement from <u>not</u> quite understanding what he is doing. In this streamlined age, one of our most undernourished psychological needs is the craving for Black Magic and apparently the automatic computer can satisfy this need for the professional software engineer, who is secretly enthralled by the gigantic risks he takes in

his daring irresponsibility. For his frustrations I have no remedy .....

Looking for "intellectually manageable" program structures one is immediately faced with the question "But how do we manage complex structures intellectually? What mental aids do we have, what patterns of thought are efficient? What are the intrinsic limitations of the human mind that we had better respect?" Without knowledge and experience, such questions would be very hard to answer, but luckily enough our culture harbours, with a tradition of centuries, an intellectual discipline whose main purpose it is to apply efficient structuring to otherwise intellectually unmanageable complexity. This discipline is called "Mathematics". If we take the existence of the impressive body of Mathematics as the experimental evidence for the opinion that for the human mind the mathematical method is indeed, the most effective way to come to grips with complexity, we have no choice any longer: we should reshape our field of programming in such a way that their methods of understanding become equally applicable, for there are no other means. As an aside, I would like to point out that you cannot put aside my recommendation by saying "Oh gosh, yet another mathematician trying to sell the importance and universal applicability of his subject," because I happen not to be trained as a mathematician; I consider myself to be a programmer .... But I have been asking myself for, say, the last two years, what kind of programs I would care to produce if I wanted to exploit my powers of abstraction for the purpose of understanding as effectively as is done in any other field of mathematics. In particular I have been investigating how one can use one's power of abstraction and one's ability to introduce concepts in order to achieve that the number of cases between which is to be distinguished in one's reasoning, combine additively rather than multiplicatively. I would like to use the next part of my lecture to give you in bird's eye view a survey of my conclusions.

1) When programming, one should constantly bear in mind that although the program text is the last thing that leaves the programmer's hands,

the true subject matter of his trade consists of the pssible computations that may be evoked by his programs, the computations, the "making" of which he delegates to the machine. When we say, sloppily, that a program is OK, we mean that the corresponding computations satisfy the requirements. In other words, we should regard the programmer's activity not as "producing programs", but rather as "designing a large class of computations". The obligation to keep our intellectual grip on what may happen in time, while the static program text is the last thing we can lay our hands upon, the obligation to understand the computations as they evolve in time via the tangible program text, yields an urgent plea to keep the sequencing rules, i.w. the mapping between the progress through the program text and the progress through the computation, as straightforward as possible. As a result I decided to abstain in sequential programming from the goto statement and to perform all sequencing control by conditional, alternative and repetitive clauses and the subroutine mechanisms. In view of our obligation to bridge mentally the conceptual gap between the static program text and the dynamic computations, I came identify the goto statement as one of the combinatorial complexity generators I was looking for.

2) Another conclusion especially related to sequencing is the following. Whenever a construction is composed by means of a sequencing clause, encapsule it and find a description of its net effect in which the fact that it contains a clause is no longer transparent. If to write

"if $x < 0$ then $x := -x$"

this means "replace x by its absolute value"and the latter description is equally applicable to both cases. And if you have not a ready-made function (such as the absolute value) at your disposal in terms of which to describe the net effect of such a compound statement, invent this function and be sure that its properties are nice and also the ones you want. If you cannot find such a function, don't ignore that warning, for then you are on the verge of messing things up. To give you an analogy: when programming in machine code you can appeal to the add-instruction but in doing so it is immaterial for you whether the hardware invoked has a serial or a parallel adder.

3) The above encapsulation of the internal structure of a program component is a specific instance of a more general principle, that we find applied in the structure of any mathematical theory: whenever a piece of mathematical reasoning appeals to a theorem, the only thing that matters on that level is what the theorem asserts and on that level it is equally immaterial how that theorem has been proved (elsewhere). An appeal to a theorem is not an abbreviation for a specific one of its possible proofs, the existence of the theorem as such allows the user to forget about its proof. We can - and should - apply the same principle in program composition, where it is rewarding to separate for each program component clearly "what it does" and "how it works". With the possible exception of the recursive routine, the level in which a component is used on account of what it does is always disjoint from the level which is concerned with how it works. These two sides of the same coin are very well known in the relation between main program and the subroutines it calls; our vision of this relation, however, becomes blurred as soon as we regard the calling sequence as an abbreviation of the body and as a result, try to understand the total activity at the same, homogeneous semantic level: one has then mentally destroyed a useful structure. This wide-spread confusion, I am sorry to say, seems to have been promoted and to be kept alive by Programming Linguistics, as inspired by Automata Theory.

4) Was the previous point that it is unwise to take the internal structure of a program component into account on the level where it is used on account of what it does, this point makes the same statement with respect to compound data structures, which are ultimately represented by aggregates of variables of more primitive types. The levels in which only the collection of the possible values matter is quite distinct from the level which is concerned with the question how these composite values can be represented by aggregates of values of simpler types. One of the most common sources of program errors seems to be that an operation on a variable is inadvertently coded in terms of components of a specific representation. To give again a very simple example: for a binary

machine one may be tempted to replace the question "is this integer even?" by "is its least significant digit zero?". Later, going with the program from one binary machine to another one discovers that this translation is not valid when negative numbers are represented by 1's-complements.

With the subroutine we have the two sides of the operational coin, viz. "what does it for you" versus "how does it"; with abstract data types we have the two sides of the representational coin. viz. "what values can it take" versus "how are these values represented". Most currect programming languages cater via the subroutine mechanism reasonably well for the operational abstraction but their mechanisms for representational abstraction, if any, are less convincing. The possibility to have representational abstraction reflected in the program code seems, however, equally essential.

5) A program should be regarded not as an object all by itself but as a member of a class of related programs, containing both alternative programs for the same job and similar programs for similar jobs. We wish to regard transition from one member of this class to another member of this class as replacing one or more program modules by one or more alternative program modules. Hereby insisting that the correctness proof for the unaffected modules and their interrelationship remains unaffected as well. The analysis of the latter requirement gave me a very much clearer understanding of how different levels of abstraction can be distinguished in a large program and how the distinction between these various levels of abstraction can be used to good advantage, and it enabled me to give a reasonably specific contents to the goal of "program modularity", which is often hardly more than a motherhood statement. Specific consequences of this analysis have been

5a) a proper module is in general more than a subroutine: in its general form it reflects - or "documents", of you wish - all consequences of a locally independent design decision, in general involding a set of joint representational and operational refinements;

5b) the adequacy of context-free methods for representing program structure seems to have been over-estimated.

So far for the survey of the conclusions I reached when I tried to reshape our programming activity into one which is better adapted to our mental capacities and limitations. I could only give you a bird's eye view of them; yet I hope that you have grasped some of their flavour and, possibly even, some concrete guidance. The survey is by no means complete, other insights will follow, perhaps as the fruits of my own activity but hopefully they will be gained by others as well.

Although I have done my best to be as clear as possible, I fear that I must have failed to reach anyone in my audience - if any - who, on account of his current environment, identifies the task of programming with the task of writing programs in FORTRAN - a programming tool which, indeed, was a great step forward when it was conceived some fifteen years ago but which, by now, should be regarded as a lower level language, as a low grade coding device. If he sticks to that conception of his task, he will fail to understand me, as one of my morals is that in the mean time his programming tool and the thinking habits induced by, have grown hopelessly inadequate. This fear of being misunderstood is, alas, supported by many a disappointing experience. As a teacher it is my job to help programmers in clearing up their own thinking. Often this is a highly rewarding activity, equally delightful and instructive for both parties. But when talking to the produce as grown in what is getting known as "the pure FORTRAN environment", I am usually baffled, for unsuspected depths of misunderstanding open themselves before my very eyes. It is well known that we don't gain automatically from every experience, on the contrary, that the wrong experience may easily corrupt the soundness of our judgement. In the case of FORTRAN, it is my impression that it's intellectually degrading influence is not commonly recognized, that too few people realize that the sooner we can forget that it ever existed, the better, as it is now too inadequate, too difficult, and therefore too expensive and too risky to use.

In connection to the preceding paragraph about FORTRAN I would like to give a short explanation of my silence about COBOL. The point is that I have neither first-hand nor second-hand experience with COBOL's influence on its users. But the preceding paragraph was misunderstood by one of my colleagues who came from the mixed FORTRAN - COBOL environment: he was terribly puzzled, saying "Why attack FORTRAN's influence? For COBOL's influence is an order of magnitude worse!" End of explanation.

Meine verehrte Hörer und Hörerinnen - this had to be said in German as my English does not cater for this! - let me come to my final conclusions. Automatic computers are with us for twenty years and in the course of that period of time they have proved to be extremely flexible and powerful tools, the usage of which seems to be changing the face of the earth (and the moon, for that matter!). In spite of their tremendous influence on nearly every activity, whenever they are called to assist, it is my considered opinion that we underestimate the computer's significance for our culture as long as we only view them in their capacity of tools that can be used. In the long run that may turn out to be but a ripple on the surface of our culture. They have taught us much more: they have taught us that programming any non-trivial performance is really very difficult and I expect a much more profound influence grom the advent of the automatic computer in its capacity of a formidable intellectual challenge which is unequalled in the history of mankind. This opinion is meant as a very practical remark, for it means that unless the scope of this challenge is realized, unless we admit that the tasks ahead are so difficult that even the best of tools and methods will be hardly sufficient, the software failure will remain with us. We may continue to think that programming is not essentially difficult, that it can be done by accurate morons, provided you have enough of them, but then we continue to fool ourselves and no one can do so for a long time ubpunished.

Finally: from my words some of you may have concluded that I am just spiteful and bitter. Let me reassure you: I am neither spiteful, nor bitter. Not yet ..... although I must admit that the major part of computing Science (or perhaps more accurately: Programming Folklore) often strikes me as unchallenged prejudices, repeated over and over again as articles of faith. But in my introduction I have announced that I should say what I wanted to say. I know that I have used strong language and harsh words - but the time to be gentle has passed: the situation is much too serious to be covered by politeness.

The floor is now open for discussion and I thank you for your patience.

Eindhoven, dec. 1969.