Correctness concerns and, among other things, why they are resented.

Invited paper, to be presented at the 1975 International Conference on Reliable Software, 21-23 April 1975, Los Angeles, California U.S.A.

Edsger W.Dijkstra
Burroughs
Plataanstraat 5
NUENEN - 4565
The Netherlands

According to Webster's definition of a tutorial: "a paper and esp. a technical paper written to give practical information about a specific subject", this paper is <u>not</u> worthy of the name "tutorial", because I would bever describe what I intend to do as "giving practical information". On the contrary: I intend to give as little "practical information" as I possibly can. I am not going to enumerate facts, results and theories, for those you can find --in abundance, I am tempted to add-- in the published literature. What I do hope to achieve, however, is helping you to understand and evaluate those facts, results and theories when you encounter them in the literature. I intend to do so by providing you with a historical perspective, be it --in more than one sense-- a partial one. It is a lucky circumstance that the amateur historian, like myself, can always come away with very few facts: the fewer the facts, the greater our freedom of interpretation, and it is that freedom that I intend to enjoy.

For a short while we have to go back to the scientific climate at the beginning of this century, for there we seem to find the roots of the philosophica opinions that prevailed a few decades later. And these philosophical opinions, in turn, seem to be the source of today's tacit assumptions. By being tacit, these assumptions tend to escape being challenged and that is bad, because they are third-generation off-spring of scientific hopes that, in the mean time, have been shown to have been unjustified, they express everyman's image of a goal that, in the mean time, has been proved to be unattainable.

The scientific optimism of the late 19th century is responsible for the common opinion that "the greater our knowledge, the more perfect our understanding

This assumption is the ultimate justification for so many of our university curricula, that one can hardly challenge it without running the risk of being accused of preaching the virtues of ignorance. Yet I challenge it: the greater our knowledge, the more perfect our understanding..... what sheer nonsense! Think about the wealth of information that the modern communication media give us about the world and its inhabitants: never before, Mankind has been so confused about itself as today! The overpowering flood of conflicting impressions leaves many of us so bewildered that, in utter despair, they seek salvation in cheap mysticism or a narrowminded ideology: the greater knowledge has not created greater understanding, it has created extremism instead. Another fine example is provided by the last decades of nuclear physics. At a phantastic expense a phantastic number of bubble chamber experiments have been made. And what has happened? By discovering a new "elementary particle" every other week, the nuclear physicists have made themselves the laughingstock of the scientific community.

What, you may ask, has all this to do with Computing Science? Well, everything. In perfect analogy to the belief that our understanding was imperfect, because we did not know enough, it was also felt that many goals were not reached, because what had to be done for that purpose, could not be done fast enough. The usual explanation of the successful advent of the automatic computer refers to Babage's technical failure to make one with mechanical means, and observes that, at last, electronic technology had made its construction feasible. I would like to offer another explanation: at last there was a cultural climate in which the attitude of "the more, the better" was such a predominant one that it was willing to accept the gimmick as the obvious tool for our salvation. And if you do not believe this explanation, try to imagine, how Confucius, Buddha, Jesus, Mohammed or Homer would have reacted when they had been offered a UNIVAC 1..... Who of us doesn't remember the advertisements for the first electronic business-machines, proudly announcing that "our machines will take for you more than a hundred thousand decisions a second!". The advertisement did not warn the reader that they were all trivial, nor that in this connection the use of the term "decision" is misleading, but to that misleading terminology I shall return later. Speed was the key issue and in the thinking of many still is.

The first warnings that with faster and faster machines, the conceptual problem of designing those wonderful computations for which tomorrow's machines would be powerful enough, would at least grow in propertion, were not taken in gratitude. They were not even heard, for they would have spoiled a dream.... Babbage's daydream had first to become a fully transistorized nightmare.... And has the latter traumatic experience cured all of us from confusing daydreams with attainable reality? I am afraid not: the distorting spell of speed still seems to make its victims. We see automatic theorem provers proving toy theorems, we see automatic program verifiers verifying toy programs and one observes the honest expectation that with faster machines with lots of concurrent processing the life-size problems will come within reach as well. But, honest as these expectations may be, are they justified? I sometimes wonder...

Another path via which the last century's unwarranted scientific optimism has influenced our trade is psychology. In an effort to understand Man, psychologists decided to study Rat. They designed a crude model of the Rat's behaviour, a model that showed only a superficial resemblance to the behaviour of the true, average Rat. But in the scientific optimism of those days, that did not disturb anybody: the beginning, full of promise, was there, in principle the modelling worked and from now onwards it was only a question of refining the model. Once we should have a fully acceptable model of Rat, and from there to a model of Man would then be only a next step. That the ensuing image of Man would, in all probability, be somewhat ratomorphic did not seem to bother many psychologists either. But again the question is: how long can one live with a "promising start" without becoming blind to the possibility that the promise will never be fulfilled?

What, one may ask again, has this to do with our trade? Well, as I hope to explain: everything. At the beginning of the computer revolution a book appeared with the ominous title "Giant brains, or machines that think". Once at a time I was tempted to write a compensating companion to it under the title "Giant hearts, or machines that fall in love", but I did not do so, when I realized that the title about the giant brains was more symptom than cause. To the somewhat excited hoped as to what machines can do had been added a somewhat simplified image of Man.

Extreme consequences of that attitude can be found in all efforts aimed at developing "natural language programming systems", a theme that recurs with the same regularity as influenza epidemics. It is observed that without considerable dedication people have a hard time expressing themselves precisely in a formal system as provided by a programming language. Man at large being rather education-resistant in that view, the problem is solved by letting him express himself in a way which precludes precision. By posing smart questions about the Man's intentions --the Man in this connection always being denoted as "the user"-- the system will eventually guess his intentions. Interactive facilities have added a new dimension to this game and ultimately the "user" will produce his design specifications as a kind of Pavlovian slobber.

These are, of course, extreme cases. But even efforts to prove mechanically a posteriori the correctness or --because for the latter there seems to be a bigger market-- the incorrectness of programs, efforts to guess mechanically invariant relations for repetitive constructs --because most programmer's are supposed to be too lazy or too stupid to write them down themselves--, they all have something of that condescending flavour. Just because a machine is very good at a few things we are poor at, it does not follow that ultimately the machine will be very good at something just because we, ourselves, find it difficult. Yet, the euphoric pressure to believe so is still very strong.

Although not directly connected with the theme "software reliability", I cannot resist the temptation to draw from the wider field of Computing Science a few further examples. The desire to understand Man in terms of Machine has --as is only to be expected-- its inverse counterpart, viz. the desire to understand the Machine in terms of Man: in computing science the terminology is shockingly anthropomorphic, What with Babbage was still called "a store" is now "a memory", what used to be called "an instruction code" is now called "a programming language". I picked up the sentence "When this guy wants to talk to that guy..." while the speaker referred to distant components of a computer network. I contend that this preponderance of anthropomorphic terminology is the symptom of a wide-spread confusion, a confusion without which, for instance, so-called "conversational programming" would never have enjoyed the glamour that, at one time, it did enjoy. Finally: the traces of the superstition "the more, the better" can be found in the awe for so-called "powerful" programming languages, with all their bells and whistles, and does the belief "the greater our knowledge, the better our understanding" not find its ultimate confirmation in today's cult of large data bases?

* * *

I would not like to leave you with the impression that Computing Science has developed along 30 years of foolish projects: such a distortion of the truth would be too gross even for the worst amateur historian! With respect to Software Reliability a lot has happened, insights of lasting value seem to have been gained, and all that took place in a relatively short period of time!

The first expression of serious concern about the confidence level of our programs, that I could find in the open literature dates from 1961. In 1965, at the IFIP Congress, it is voiced by more. Stanley Gill, for instance, remarks: "Another practical problem, which is now beginning to loom very large indeed and offers little prospect of a satisfactory solution, is that of checking the correctness of a large program.", certainly an expression of serious concern! John McCarthy opens his introduction with: "The prize to be won if we can develop a reasonable theory of computation is the elimination of debugging. Instead, a programmer will present a computer-checked proof that a program has the desired properties." Here, the reader is left in doubt as to whether McCarty's main concern is really the ultimately attainable confidence level or only the great expense of the debugging process, but serious concern is in any case expressed. In passing we note --we shall return to this later-- that both speakers don't mention yet any alternative for a posteriori verification.

Were these quotations two fairly isolated examples taken from the 1965 Conference Proceedings, in 1968 the climate has changed drastically. In October 1968 the NATO Conference on Software Engineering created a sensation by its open admission of the software crisis. Anyone doubting, that much has changed since then, should read those conference proceedings! One now very well-known professor of Computing Science confesses that "the word "proof" causes him to hace a sort of mental hiccough", another very well-known professor of computing science calls the notion of proof "idyllic" and nowhere one finds a reference to R.W.Floyd's article that by that time is already one year old! Was that NATO Conference perhaps technically not very significant, its political significance can hardly be underestimated: the most significant work could happen since!

* * *

After the above sketches of the intellectual climate, we turn, in a little bit more detail, to some of the more technical aspects: obviously we do so without any claim to completeness.

One of the, in retrospect, most striking things is that for many years the correctness problem was solely viewed as a posteriori verification of given programs. Given a program and given a set of requirements, does the given program meet the given requirements? Phrased as a question that could be answered by "Yes" or "No", it was apparently not without appeal for the mathematicians of that period, mathematicians who, by their training, were on the average perhaps rather analytically oriented. But besides that, I think that a specific tradition pushed them into that analytic direction, and that is the tradition that got its pronounced form with the work of Alan M.Turing. It is the approach, in which a --hopefully well-understood!-- mechanism is started and we are invited to figure out, whether we can prove something about the class of ensuing happenings, corresponding to the class of initial states in which the mechanism may be started. We can ask ourselves whether it will terminate or, if that is too difficult, whether we can say something about the final state provided the activity terminates, and so on. It is the mechanistic, operational point of view which regards the "answer" to be <u>defined</u> as the last one of a long sequence of intermediate machine states of which the initial state is the first one. Turing's work and the branch of mathematics that emerged from it were so impressiv that they caused a strong bias in the earlier work on program correctness, a bias which I do not consider as wholly fortunate. Its main consequences seem to have been the following.

Firstly, nearly all through the sixties, efforts at giving a formal definition of the semantics of programming languages have been in the form of writing an interpreter, i.e. designing an abstract machine, for such a programming language. Developing means for describing the intermediate states of such an abstract machine became soon a major concern.

Secondly, the unsolvability of the halting problem, combined with an early desire to mechanize correctness proving, has caused many to restrict themselves --apparently without much hesitation-- to proving partial correctness only, viz. proving only that an acceptable answer will be produced under the additional assumption that the computational process terminates.

Since the late sixties we distinguish, however, a process for which "shedding the shackles of automata theory" could be an appropriate --be it perhaps too vivid-- description. Two things started to happen in parallel, initially, as far as I can see, rather independently of eachother.

The first development was the result of challenging the choice of a posteriori verification of given programs as the most significant problem, the argument against that choice being that programs are not "given", but must be designed Instead of trying to find out which known proof-patterns are applicable when faced with a given program, the program designer can try to do it the other way round: by first choosing the proof-pattern that he wants to be applicable, he can then write his program in such a way as to satisfy the requirements of that proof. This "inversion" of the problem of program correctness was one of the cornerstones of the field of activity which about five years ago became known as "programming methodology". Correctness proofs began to play in that field of activity two significant roles: firstly, in the constructive approach to the problem of program correctness, proof-patterns provided an important heuristic guidance during the programming process, secondly, the length of the correctness proof required was generally accepted as an objective measure for the "elegance" of programs and for the "adequacy" of proposed language features. This objectivity has probably been more effective in reaching a comfortable consensus of opinion among many than anything else, certainly more effective than eloquence could ever have been. I mention this consensus explicitly, because it has been so important: it was the only way in which we could hope to raise language design from the political and commercial level, aimed at "user satisfaction" to the level of a scientific activity.

In parallel to the exploration of programming methodologies, the methods for proving program correctness and their foundations, slowly divorced themselves from the operational point of view. Naur's article of 1966 and Floyd's article of 1967 are still rather operational, but Hoare's article of 1969 presents a step away from that point of view, as he suggests that "axioms may provide a simple solution to the problem of leaving certain aspects of a language undefined". This remark is deeper than the primarily suggested applications such as leaving wordlength or precise specification of rounding rules unspecified. Hoare's rules for the repetitive construct rely on the fact that the repeatable statement leaves a relevant relation invariant. As a result the same macroscopic proof is applicable to two different programs which only differ in the form of the repeatable statements S1 and S2 , provided that both S1 and S2 leave the relevant relation invariant (and ensure progress). But then the same proof applies to a non-deterministic machine in which a daemon decides quite arbitrarily at each repetition, whether S1 or S2 will be chosen! The fact that the obvious way was shown for dealing with at least some very common and desirable forms of non-determinac although obvious in retrospect, only came to be exploited systematically a few years later. From the operational point of view, such daemon's seem to create problems: they must be assumed to supply some "ghost-input" to the mechanism that, assumed to be deterministic, otherwise would not know which way to go. Reasoning from the other end and starting with the functional specifications, the daemons only enter the picture by the time that we start thinking about implementations: they appear as the freedom of the implementation, viz. wherever the choice does not matter.

* * *

I now skip the enumeration of a long series of important publications, successful development projects and promising research efforts, and would like to round up by drawing your attention to two side-effects of all these efforts, the final impacts of which still lies in the future.

The first one is that most techniques for proving the correctness of a program treat the program text as a static, rather formal, mathematical object that can be dealt with independently of the fact that there may exist machines that could execute such a program. As such, a clear separation of concerns emerges: we might call them the mathematical concerns about correctness and the engineering concerns about efficiency. In contrast to the correctness concerns, the efficiency concerns are only meaningfull in relation to implementations and it is only during the efficiency concerns that we need remember that the program text is intended to evoke computational processes. Both the mathematical and the engineering concerns have, of course, always been with us, but once they used to be dealt with inextricably intertwined. The discovery how to separate them rigorously in our thinking is relatively young, and even when aware of this possibility, we often fall back into our old bad habits. But I guess that this discovery will have profound consequences, which will become fully apparent when new generations of programmers have been educated who separate these concerns more naturally than we do, unhampered as they are by our obsolete experience and out worn-out habits.

The second side-effect concerns the bells and whistles of yesterday's programming languages. When the correctness guys started their efforts, they used at first very simple programming languages, mini-languages, one might say. For the programmers who lived "in the real world of computing", this restriction was often a reason for some scorn: "Toy-problems, solved by toy-programs written in toy-languages: what has that to do with us?".

But things, again, are changing. People decided to try to program without got-statements, and it did not do much harm to the efficiency of their programs, quite often on the contrary even. People decided to do away with "controlled variables" as we know from FORTRAN's DO-loops or ALGOL 60's <u>for</u>-statements, and this did not impair the efficiency either. But after a while, these "restrictions" turned out to bear unexpected fruits: for instance, people who had trained themselves in using a very simple, but very elegant repetitive construct, could discover algorithms that turned out to be much harder to find by a programmer in whose mind repetition was irrevocably connected with a controlled variable. The abolishment of the "controlled variable" turned out to be a gain! And this process seems to be continuing: the correctness guys have still a tendency to use mini-languages, but in the mean time their experience in using them is no longer restricted to toy-problems, on the contrary!

I would like to end this talk by relating a recent personal experience that may be exemplary for the kind of surprises that the use of such mini-languages still has in store for us. I had been exploring programming methodologies and as a carrier for my investigations I had used a mini-language without recursion (although intuitively I was fully familiar with recursive programs for about fifteen years). I did not feel the urgent need to introduce recursion at this stage, because my language did contain extensible one-dimensional arrays and, when tempted to use recursion, I felt that I could always try to program around it.

Tackling a problem from graph theory I developed (as a matter of fact without feeling the need for recursion) without much hesitation in a number of steps a beautiful algorithm, clear, compactly coded and highly efficient in terms of operations needed (viz. linear in the sum of the number of edges and the number of vertices of the given graph). It turned out that, shortly before, an algorithm for the same problem had been published in an American journal, an algorithm that had the same efficiency characteristics, but was very hard to understand and --as a result!-- was generally considered as a great intellectual achievement: no one less than Donald E.Knuth described the published version as "a deep algorithm". For the difference in clarity and ease of discovery I have only one explanation: my program manipulates explicitly --they entered the design quite naturally, one after the other-- four independent stacks, most of them growing and shrinking asynchronously with eachother. The published version, however, was a recursive program, by its very nature favouring a solution with one (anonymous) stack, and squeezing a four-stack algorithm into that straightjacket is in all probability indeed "a great intellectual achievement", which, however, cannot be expected to lead to a very natural program.

The discovery that doubts may have to be casted upon recursion --in Computing Science for more than a decade the hallmark of academic respectability!-- was something of a shock for me. Although at the moment of writing not yet conclusive, evidence is piling up: a number of our established "powerful" programming language features, even beloved ones, could very well turn out to belong rather to "the problem set" than to "the solution set". And even if this were the only lasting contribution of the work of the correctness guys, their efforts seem already well-rewarded!

22nd November 1974