Concern for Correctness as a Guiding Principle for Program Composition.

A State of the Art Report, at least when written by me, is always a mixture of fact and fiction. It is mainly fact but, in a genuine effort to be up-to-date, I cannot refrain from some extrapolation into the future, and a certain amount of wishful thinking ~~from~~ on my side is bound to leave its traces. Let it be so.

Let me start with a well-established fact: by and large the programming community displays a very ambivalent attitude towards the problem of program correctness. A major part of the average programmer's activity is devoted to debugging, and from this observation we may conclude that the correctness of his programs -or should we say: their patent incorrectness?- is for him a matter of considerable concern. I claim that a programmer has only done a decent job when his program is flawless and not when his program is functioning properly only most of the time. But I have had plenty of opportunity to observe that this suggestion is repulsive to many professional programmers: they object to it violently! Apparently, many programmers derive the major part of their intellectual satisfaction and professional excitement from not quite understanding what they are doing. In this streamlined age, one of our most under-nourished psychological needs is the craving for Black Magic, and apparently the automatic computer can satisfy this need for the professional software engineers, who are secretly enthralled by the gigantic risks they take in their daring irresposibility. They revel in the puzzles posed ~~to them~~ by the task of debugging. They defend -by appealing to all sorts of supposed Laws of Nature- the right of existence of their program bugs, because they are so attached to them: without the bugs, they feel, programming would no longer be what it used to be! (In the latter feeling I think -if I may say so- that they are quite correct.)

All this becomes the more surprising when we contrast it with a mathematical technique that is justly famous for its efficiency and its high (and generally accepted!) convincing power, viz. killing a conjecture by means of a single counter-example. Everyone accepts the strength of that killer, but when a program fails, believe it or not, its author will defend himself by explaining that it was only (!) a very, very special case where it went wrong!

In other words, compared with what it should (and what it could!) be, the average programmer's professional standard is shockingly low. Mind you, I am just observing (and making a value judgement), I am not blaming the average programmer,

for he has to work under rather unfortunate circumstances.

Firstly, as a rule he is not free in the choice of his programming tools and the most widely used programming languages are rather inadequate vehicles: when FORTRAN has been called an infantile disorder, PL/1 must be called a fatal disease and for COBOL I have no words..... In the case of programming languages I am very much interested in their influence upon the thinking habits of their users: particularly in the case of what is becoming known as "the pure FORTRAN environment" the strength of its intellectually degrading influence is, as a rule, grossly underestimated. Yet thousands and thousands of programmers still have to live (and die?) in it.

Secondly-and as I would like to avoid a lawsuit for libel, I shall refrain from mentioning any manufacturer's name, but I trust my readers to be able to supply what I have omitted- language implementations and the operating systems supporting them have grown into baroque monstrosities with most curious and -worse!- changing properties. And when programmers have to erect their "computational edifices" on such shaky foundations, it is hardly amazing that they regard full confidence in the correctness of one's program as a ridiculous presumption. In the old days one of the most fundamental properties of automatic computers was that by means of one's program one could keep complete control over what was going to happen. This property was, and still is, essential for the computer's usefulness: many modern computing systems, I am sorry to say, don't have this property anymore, and from adequate tools they have become depressing burdens. The proliferation of error-loaded and therefore unreliable software has done endless harm in the minds of those who tried to use it: one does not gain automatically from experience and the struggle with "the third generation" offers, alas, an example of how the wrong experience may easily corrupt the soundness of one's judgement.

If the sorry state of affairs ~~as~~ described above were a complete picture, it would indeed be most depressing, and we might wonder whether we had not better look for a suitable branch on the nearest tree. Thank goodness, the picture is not complete, there are already ~~at~~ on our side of our intellectual horizon the first solid glimmers of hope; it is to those glimmers of hope that the remaining part of this contribution is devoted. For me, the Conference on Software Engineering, sponsored by the NATO Science Committee and held at Garmisch in October 1968, has been a turning-point: it was the first time I witnessed ~~that~~ a group of experts -all of them so high in their local hierarchies that they could afford to be honest- unanimously admitting that a software crisis did indeed exist. This struck me as very important, because before its existence was admitted it was vain to hope that

something could be done about it.

When we wish to raise the confidence level of our designs the first question to ask is: "On what can we base our confidence?". This question has been explored; well, programmed computers may be amazing pieces of equipment, but as it turned out, many times more amazing are the flimsy grounds on which we have been willing to believe that their output was the output we wanted!

A program can be regarded as an (abstract) mechanism embodying as such the design of all computations that can possibly be evoked by it. How do we convince ourselves that this design is correct, i.e. that all these computations will display the desired properties? A naive answer to this question is "Well, try them all.", but this answer is too naive, because even for a simple program on the fastest machine such an experiment is apt to take millions of years. So, exhaustive testing is absolutely out of the question. As a result, testing by randon sampling is hopelessly inadequate as well, because even the most vigorous exercising possible will only cover a truly negligible fraction of the possible number of cases, and whole classes of in some sense critical cases can -and will!- be missed: only the most obvious blunders will show up. The first moral of this story is that program testing can be used very efficiently 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 unescapable 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 test cases, for all others (the vast, vast majority) one relies on reasoning. In spite of the fact that many, many programmers have dimly suspected that the solution can only be found in this direction, they seldom make more than a half-hearted attempt at it, the snag being that the amount of reasoning needed often becomes excessive.

Yet this function of the mechanism's internal structure opens a new and promising way to attack the reliability problem. Once we have seen that the required 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, 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~~ to raise the confidence level of our final product?".

So, instead of trying to devise methods for establishing the correctness of arbitrary, given programs, we are now looking for the subclass of "intellectually manageable programs", which can be understood and for which we can justify without excessive amounts of reasoning our belief in their proper operation under all circumstances. This is done in order to reduce the number of test cases needed; in the case of programs -i.e. abstract mechanisms- there are now strong indications that this approach can be so effective that the number of test cases needed is eventually reduced to zero, i.e. that the correctness can be shown a priori.

If the amount of reasoning needed is to be kept tolerably low, it is absolutely necessary that the number of cases between which we have to distinguish (in our reasoning!) combine additively, rather than multiplicatively. This implies that in program composition we must exploit our power of abstraction and our ability to introduce useful concepts as effectively as is done in any other branch of mathematics. (And what is more, we have to exploit these powers very consciously, firstly ~~for~~ (because of the) lack of tradition in programming, secondly because programming is a highly creative activity in which we aim at constructing things ~~eventually~~ of an eventual degree of sophistication much higher than the average mathematical theory.)

In answer to the above challenge, two developments are taking place. The first is an investigation as to what structural properties make a program "intellectually manageable"; the second is a search for a methodology for constructing such intellectually manageable programs.

With regard to desirable program structure, one or two very clear conclusions have been reached. 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 possible computations that may be evoked by his program, 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 so much as "producing programs", but rather as "designing a large class of computations". (From this point of view measuring programmer's productivity by the

number of lines of code produced per month is as ridiculous as measuring a composer's productivity by the number of notes scribbled on his score!) 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.e. the mapping between the progress through the program text and the progress through the computations, as straightforward as possible. In sequential programming one can -and should- do this by abstaining from the goto statement and by performing all sequencing control by conditional, alternative, selective 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, the goto statement must be exposed as one of the combinatorial complexity generators we have been looking for.

While the above conclusion deals with avoiding unnecessary complexity, the next ones deal with the exploitation of our powers of abstraction for the purpose of mastering the inherent complexity. They deal with operational abstraction and representational abstraction.

Operational abstraction is well-known; in program texts it is reflected by subroutines or by indentation. It is the kind of encapsulation 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 the theorem has been proved (elsewhere, i.e. on another level). We can and should apply the same principle in program structuring, where it is rewarding to separate for each program component clearly "what it does" and "how it works". With the exception of the recursive routine, the level on which a component is used on account of what it does is always distinct from the level which is concerned with how it works. Our vision becomes unnecessarily blurred when we mix the two levels and try to understand the whole happening on the same, homogeneous semantic level: one has then destroyed a useful structure.

A specific application of the above sketched operational abstraction is the following. Whenever a program component 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 such a clause is no longer apparent. For instance, give to

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

the description "replace x by its absolute value", a description which is equally applicable to both cases. And if you don't have 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 component, invent this function and be sure that its properties are mathematically 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 a hardware analogy: when programming in machine code one can invoke the add-instruction but in doing so, it is on that level of interest immaterial whether the hardware invoked has a serial or a parallel adder.)

Representational abstraction is concerned with compound data structures, which are ultimately represented by aggregates of variables of more primitive types. The level on which only the collection of possible values of a compound data structure matter is quite distinct from the level which is concerned with the question of 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. (E.g. in a binary machine one might be tempted to rephrase the question "is this integer even?" by "is the least-significant digit of this integer equal to zero?" forgetting 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 do for you" versus "How does it do 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 current programming languages cater ~~via the~~ by means of the subroutine mechanism reasonably well for the operational abstraction; their facilities for representational abstraction, if any, are less convincing. The possibility of having ~~have~~ representational abstraction reflected in the program code as well seems, however, to be equally essential.

The final subject to be touched upon is a methodology for constructing such "intellectually manageable" programs. ~~In~~ Within the framework of this contribution, I can only touch upon it, because it is very intimately linked with the much larger and more general field of heuristics: "What are for the human mind the most effective ways ~~for~~ of finding solutions to problems?". To my amazement -apparently I am a very naive person- I find even reasonably creative mathematicians rather unwilling to face this question: the mere suggestion that some sort of helpful answer to that question could be given at all has a tendency to strike them as sacrilegious. (Again I suspect that the mystery in which the act of creation is so tenaciously kept enwrapped,

satisfies one of our deeper, otherwise underhourished psychological needs.) In the restricted field of programming, such a methodology has a much better chance ~~for~~ of survival ~~life~~, firstly because, by sheer necessity, programmers have to be more conscious of methodological aspects, secondly because they are often faced with (huge) problems for which the possible existence of a solution, however, is pretty obvious.

My thesis is, that a helpful programming methodology should be closely tied to correctness concerns. I am perfectly willing to admit that I myself may be the most complete victim of my own propaganda, but that will not prevent me from preaching my gospel, which is as follows. When correctness concerns come as an afterthought and correctness proofs have to be given once the program is already completed, the programmer can indeed expect severe troubles. If, however, he adheres to the discipline to produce the correctness proofs as he programs along, he will produce program and proof ~~in~~ with less effort than ~~just~~ the programming alone would have taken ~~otherwise~~.

The framework of this contribution does not allow the inclusion of a worked-out example. Instead I shall try to sketch how the hand-in-hand construction of a program and its correctness proof ~~takes place~~. guides the programming process.

I found what I regard as the quintessence of this methodology in the summer of 1968, when my attitude towards flow charts changed radically. Up till that moment, I had regarded a flow chart as something incomplete, as a plan, as a (half-pictorial, but that is not essential) representation of my intentions, something that as a description would only make sense if all further details, down to the bottom, had indeed been supplied. But it was then that I saw that such a sketch of a program still to be made, could be regarded as an abstract version of the final program, or ~~stronger~~ even, that it could be regarded as "the program", be it for a -in all probability hypothetical- machine with the proper repertoire of primitive actions operating on variables of the proper types. If such a machine were available, the "rough sketch" would solve the problem; usually it is not available, and actions and data types assumed have to be further detailed in next levels of refinement. It is the ~~purpose~~ function of these next levels to build the machine that has been assumed to be available at the top level, i.e. at the highest level of abstraction. This most abstract version is now no longer ~~something about~~ a description of the final program, it is an essential part of the total program; its correctness is independent of the lower level refinements and can be established beforehand. To give this correctness proof before proceding with the lower level refinements serves many purposes. It establishes the correctness of the top level. Fine, but more important is that we do this by applying

(what is

well-established theorems applicable to well-known sequencing clauses, and as a result it becomes most natural to avoid clever constructions like the plague. But the most important consequence is that the proof for one level ensures that the interface between itself and the lower levels has been given completely, as far as is relevant for that level; and also, by fixing in the interface only what the correctness proof really needs, the interface can be kept free from overspecification.

Finally, a word or two about a wide-spread superstition, viz. that correctness proofs can only be given if you know exactly what your program has to do, that in real life it is often not completely known what the program has to do and that, therefore, in real life correctness proofs are impractical. The fallacy in this argument is to be found in the confusion between "exact" and "complete": although the program requirements may still be "incomplete", a certain number of broad characteristics will be "exactly" known. The abstract program can see to it that these broad specifications are exactly met, while more detailed aspects of the problem specification are catered for in the lower levels. In the step-wise approach it is suggested that even in the case of a well-defined task, certain aspects of the given problem statement are ignored at the beginning. This means that the programmer does not regard the given task as an isolated thing to be done, but is invited to view the task as a member of a whole family; he is invited to make the suitable generalizations of the given problem statement. By successively adding more detail in the lower levels he eventually pins his program down to a solution for the given problem.

July 1970

prof.dr.Edsger W.Dijkstra
Department of Mathematics
Technological University
EINDHOVEN, the Netherlands