Programming methodologies, their objectives and their nature.

Years ago I gave a talk at a large software house, showing what techniques we had at our disposal for proving the correctness of programs. The talk was a disaster. My audience rejected the topic for what they regarded as sound business reasons: from a business point if view, correctness was their last goal: it was much more important to make a customer dependent on a software product so poorly documented and still so full of bugs, that its maintenance contract fell into your hands as well. But not only the managers of the firm rejected my topic, so did the programmers in my audience: they felt that I was tampering with the magic of their craft. While I had urged that a programmer should continue to understand what he is doing, it transpired during the discussion that many a professional programmer derives his intellectual excitement from not quite understanding what he is doing, from the daring risks he takes in his irresponsibility. The wanted the magic of their craft to remain black magic.... I mention this story because their reaction took me by surprise. My first explanation for it was, that in this streamlined society, the craving for magic is one of our most undernourished ones and that programming without truly understanding what one is doing, is one of the ways to satisfy that deep psychological need for magic. Later I understood that there is a second explanation.

All through the centuries, knowledge and skills have been transmitted to the next generation by two techniques.

The one technique we find with the guilds: here the apprentice works for years under the guidance of the master, all knowledge is transferred implicitly, by osmosis, so to speak, until the apprentice has absorbed enough of it to become a master himself. The members of the guild typically keep their common knowledge and skills among themselves as a well-guarded secret.

The other technique we find at the Universities: here teachers try to make the knowledge explicit, to formulate it in words and symbols and, by doing so, try to bring it into the public domain. It is no accident that the rise of the Universities coincided with the emerging art of printing: by being formulated, the knowledge became more stable and lasting and could,

indeed, become public property.

But,whenever the followers of the University try to bring a new field of knowledge into the public domain, try to make an explicit science out of a secret craft, the members of the corresponding guild feel themselves to be threatened. And that feeling of being threatened accounted for the barrier that I hit when I tried to reach the programmers of that software house! But, thank goodness, that audience did not make upt the world. The so-called "software crisis" became so apparent, that all sorts of people involved, both programmers and managers, felt that something should be done about it and started to wonder what.

For a long time many people felt that the software failure was largely due to a failure of management. This is understandable: the larger a project is, the more management is involved, and it were particularly the large software projects turned into disaster, that attracted the most attention. Besides that, many were justly impressed by the American space program that, by being so successful, led many of us to believe that with perfect management one can reach any goal.

The idea, that improved management techniques could solve the problem was further inforced by the discovery that the techniques, used for managing software projects so far,were indeed highly questionable. To take a simple example: in those days it used to be quite usual to measure a programmer's productivity by number of lines of code produced per day. Reasonable, isn't it? It is a programmer's job to produce programs, and the more program he produces, the more productive he is! But on closer inspection, that measure was discovered to be not only meaningless, but even harmful. It is meaningless, because the programmer is not supposed to produce "programs", he is supposed to produce solutions, and the program he writes is only a carrier for that solution, and the more unnecessary lines of code he uses, the poorer his solution. But measuring his productivity by the production speed of lines of code encourages such poor solutions: it encourages the fast production of insipid code. Besides that, that crazy measure of productivity encourages coding as such, and more and more people began to discover that the coding stage, at least most of it, had better be postponed as long as possible: being in a sense the most laborious stage of the programming process, we

had better try to postpone it until we are as sure and certain, as we can, about what the piece of code should accomplish. The modern advice is "Don't rush into coding!" and I expect that in a well-run establishment today a programmer is not allowed to start coding something without explicit permission. I also guess that many manager would be happy with the rule that everyone pays the punched cards he uses out of his own pocket.

But those who had great expectations of the new management techniques, those who embraced Systems Engineering or something of the same nature as the new religion that would bring salvation, were disappointed. The question is simply the following: better management techniques, although indispensable, will, all by themselves, never compensate technological shortcomings. While it is clear that without some sort of quality control it is vain to hope to produce good products, it is also clear that no amount of quality control will ensure the production of good products if you are unable to make them in the first place. And it was at that stage, when the shortcomings of our programming ability were recognized, that people started to talk about Programming Methodology (with two capital letters). Needless to say, a considerable amount of thought had already been given to it, under the surface, so to speak, before that problem area had a name.

Questions that became important were "Why is programming so difficult?" and "When we know why programming is so difficult, can we avoid some of its difficulties?". Again, the spectacular failures of a few large software projects attracted, at first at least, most attention. I remember myself having been responsible for the following argument, showing why the size of a program has consequences for the confidence level of its parts. The argument was as follows: we all know, that we must make a large program by composing it out of a number of "modules", N say, and if p is the probability for a single module to be correct, the probability P for the whole program to be correct, satisfies eomthing like

$$P \leq p^N \quad .$$

We may now laugh a little about that formula and wonder, how much it means as long as we don't know what we call "a module", but the formula showed one thing quite clearly: if N is large (and such programs we were thinking about), then p should be indistinguishable from 1 if P is to differ appreciably from 0. The size of the whole system puts heavier demands upon

upon the confidence level of the individual components. Perhaps was the argument only put forward as a justification for the already existing desire to study software reliability!

There were a number of reasons for turning our attention to the process of programming itself. A really compelling reason was the demonstrated ineffectiveness of the debugging process. People spent more than fifty percent of their time to the debugging of their programs and still they delivered error-loaded products: it was concluded that program testing might demonstrate the presence of bugs very convincingly, but is hopelessly inadequate to demonstrate their absence. And therefore attention turned to the problem how we could prevent most of the bugs from entering the design in the first place. In view of the well-known advice "Prevention is better than cure." not a surprising conclusion; yet it was a conclusion with considerable effects.

There was a second reason for turning our attention to the programming process, equally compelling, although perhaps less obvious. The insufficiency of a posteriori quality control by testing being demonstrated, their might be the possibility of proving the programs to be correct. A correctness proof seems indeed a much more effective alternative for raising the confidence level of our software products. But the first efforts at proving a posteriori that a given program was correct were not encouraging, to put it mildly: if the proofs could be given at all, they were so laborious and sometimes tortuous even, that they failed to inspire much confidence. This was partly due to our limited experience, to the absence of useful theorems, etc.,but it was also a consequence of the programs themselves. Comparing these "given" programs with programs that were designed with the intention of proving their correctness as well, showed that the amount of formal labour and of detailed reasoning needed, could depend critically upon the structure of the program itself.

This was a very valuable discovery, for it drove home the message, that it does not suffice to design a program of which we hope that it meets its requirements, but that we have to design such a program that we can demonstrate that it meets its requirements. The program must be such that whatever we accept as convincing correctness proof must be feasible as well.

This additional requirement of demonstrability of the program's correctness implied very obviously a change in the programming task and therefore a change in the act of programming. At first sight it seemed that this additional requirement would place another burden on the poor programmer's shoulders, but upon closer inspection it turned out to make his task in some sense lighter. Needless to say, this was a very encouraging discovery. As more and more became understood of what it means to prove the correctness of a program, we got gradually a better knowledge of what kind of dirty tricks are most harmful to the feasibility of giving the proof. As such, the additional requirement of the demonstrability of the program's correctness limited the programmer's freedom, but that is only a negative way of describing its influence; the positive way of describing its influence is saying that it gave him elements of a discipline. The smaller his solutions space, the smaller the probability that he looses his way.

It is about here in my story that I must insert a small interlude about the social role of mathematics. In the above I have referred numerous times to "proofs" and one may raise the question, whether formal proofs in a strict mathematical sense are really needed. They are not, of course. The real need is a convincing demonstration of the program's correctness, and as the number of cases that we can try is negligible compared with the number of possible cases, we must to all intents and purposes rely for all of them on reasoning. Whether this reasoning takes the strongly codified form of a mathematical proof is quite another matter. In many cases good English is as adequate a vehicle for expressing the reasoning in a convincing manner than a mathematical symbolism. Most demonstrations of correctness of programs that I see today are largely prose, only at specific points --where the subject matter requires it and prose would give rise to lengthy and clumsy descriptions-- supplied with some formalism. The competent reasoner always chooses the vehicle that is most convenient for the situation.

Why then, one may ask, the stress on formality, a formality that frightens the layman and tires the reader? This question is fully justified, for the formal aspects of correctness proofs for programs are given considerable attention. I would like to give several answers to that question. If you think a formal treatment an obfuscation and failing to convince because formal manipulations are as error-prone as programming is itself,

I must warn you, that verbal arguments are tricky too, often more tricky than we would like to admit, and more often than the superficial reader suspects general cautiousness fully justifies taking recourse to formal treatment. This is a justification. Secondly, believe me or not, there are quite a number of bright young lads, who just like those formal games. That is an explanation. But most important is its consequence, thanks to the fact that it is a generally accepted standard: it has turned out to be instrumental in reaching a consensus on many points.

Since the early sixties I have attended many discussions about the elegance of programs, about the adequacy of proposed language features, discussion which were distressingly inconclusive for lack of a common yardstick that was generally accepted as relevant, and was also effective. Too much we tried to settle in the name of "convenience for the user", which was a nice altruistically sounding name for our ignorance and lack of direction. As soon as the possibility of formal correctness proofs emerged, the picture changed completely. The suggestion that what we intuitively regard as an elegant program is usually also the program that admits the shortest formal correctness proof --there was a reasonable amount of experimental confirmation of that hypothesis-- was immediately accepted by many as a sound and effective working hypothesis. By its objectivity it was very effective and has done more for the reaching of a consensus with regard to the goals of programming methodology than anything else I can think of. Such a consensus is indispensable for any joint effort that should have impact, and, from a historian's point of view, the general acceptance of the working hypothesis is more important than the question how correct it is. The fact that from then onwards a number of mathematicians claim to contribute to computing science by studying the structure of such proofs for their own sake and forgetting that they were intended to convince in the first place, is a price that we should ungrudgingly pay.

So much for the role of mathematical formality: it has a place whenever it assists us in understanding what we are doing or considering. And such greater understanding has been the main target of programming methodologies.

* * *

The considerations that have been most helpful fall into two broad categories. On the one hand we have the general considerations concerning the question how to avoid unmastered complexity, on the other hand we have their applications to the specific problem of programming. And I am very glad that we had both of them: the general considerations without very tangible consequences for the programming activity would have had a hard time in trying to rise above the level of motherhood statements, the specific remarks concerning the programming activity would have had an ad hoc character without the general considerations.

The general considerations try to do justice to the fact that we have small heads and cannot think about many things simultaneously, but, besides that, get tired and unreliable when we have to think about a very great number of things in succession. The way to avoid these situations with which we can hardly cope has been captured more or less by many catch-phrases: the exploitation of one's powers of abstraction, divide and rule, the judicious postponement of commitments, the separation of concerns etc. In any case we must never forget its dual purpose: to parcel out the necessary detailed reasoning into portions of manageable size, but, more important still, to reduce to total amount of detailed reasoning that remains necessary.

Of these, the advice "divide and rule" was, of course, the best known one, and at one stage of the game people have felt that the problem of programming would be solved provided that we could divide the program to be made over "modules" of a manageable size. In retrospect it is not hard to see, that without further guidance, such an arrangement would only postpone the hard problems till the stage of "integration", i.e. when all the modules have to be hooked together. This simple view of "modularization of the program" regarded the program too much as the final product, and took the early decisions as to what the computer was to <u>do</u> too much for granted. From there the attention shifted towards what the computer was supposed to <u>achieve</u> and then it became apparent that what the computer may have to do in order to achieve the desired effect, is greatly dependent on how one structures one's solution. And "separation of concerns" is, since then, a more adequate term, because now one tries to parcel out --or: to use another term: to factorize-- the requirements.

To mention a few of such separations, we want our program to be correct, we want our program to be efficient. Although the choice of the algorithm is usually heavily influenced by efficiency considerations, by the time that we focus our attention upon the question whether under all circumstances the execution of the program will deliver a correct result, all efficiency considerations can be temporarily forgotten, we can even forget the possibility of interpreting our text as executable code and settle the problem of correctness quite independent of possible computational histories. This has given rise to all the theory and practice of proving the correctness of programs, theorems about repetitive constructs expressed in terms of invariant relations --to ensure that no unacceptable result will be delivered-- and variant functions --to ensure effective progress for each repetition--. The emergence of such theory and practice was greatly facilitated by the restriction to more disciplined sequencing, to well-known alternative and repetitive constructs rather than arbitrary jumps. (For instance: the most useful theorems apply to the alternative and repetitive constructs as a whole.)

Secondly, we want our programs to be efficient. It is here that computation times are taken into account. But by that time one does not need to worry about what result will eventually be produced, the only question that then matters is: how long will the computer be engaged on that task? In general this is a very hard question and again, people have discovered that the most efficient way for solving a problem is to run away from it, i.e. to try to avoid such computational processes to be evoked, for which the run-time behaviour becomes time-wise hardly predictable. Here Herbert A.Simon's "The Architecture of Complexity" (reprinted in "The Sciences of the Artificial [1]) has had great influence, as it has pointed out the virtue of what he calls "nearly decomposable systems". It says very roughly the following: if a systems is expected to adjust itself to changes in the environment of two types, A and B say, and the adjustment to a change of type A takes place an order of magnitude faster that the adjustment called for by a change of type B, then, while studying the latter adjustment process, we are allowed to idealize adjustment to a change of type A as instantaneous: the "mechanics" of the two adjustments need hardly to interfere and both can be studied (with respect to speed, stability etc.) in isolation. As soon as one becomes aware of the great benefits that can be derived from the "near decomposability", one immediately tries to design

one's systems with the absence of such undesirable --because uncontrollable-- interferences as one of one's main design criteria. The successful abolishment of such an interference pays twice: we don't need to mess up our thinking, and the machine need not waste its time in elaborate strategies for coping with them.

It is not surprising that these techniques have found at first their most spectacular applications in the design of operating systems. Many of them are now shaped in the form of a family of loosely coupled, harmoniously cooperating sequential processes. The harmonious cooperation is guaranteed by synchronizing them explicitly, without making thereby any assumptions about speed ratios. It is in that realm that mutual integrity of fellow programs in a multiprogramming environment, the absence of deadlock and of individual starvation etc.,can be dealt with rigorously. It is in the way of dealing with microsecond phenomena --such as interrupt handling--, ~~micro~~ milli-second phenomena --such as page fault handling-- and second phenomena -- such as resource allocation-- that we try to achieve a nearly decomposable system.

In a later stage, when these techniques were applied more consciously, similar techniques have been applied to the design of purely sequential programs. One of the ways in which this was possible, was the following. In sequential programs, we have boolean expressions (as part of our alternative and repetitive constructs) influencing during execution the flow of control. The way in which they prevent something either disastrous or only undesirable from happening is very similar to the way in which, in multiprogramming, the so-called synchronizing conditions describe whether a process could continue or should be hold up until a more favourable situation has arisen. The techniques that had been developed for the derivation of such synchronizing conditions could be taken over, practically lock, stock and barrel, for the deriviation of the sequencing expressions as they have to occur in sequential programs.

Also: inspired by the successful decomposition that could be obtained in operating systems, people have tried to separate their concerns equally effectively in the design of purely sequential programs. People trying to do so had the pleasant experience that this could be done to a much higher degree than they were used to.

A simple advice that will often show you the way for achieving such a separation is to become extremely suspicious as soon as one finds oneself faced with a case analysis which has to distinguish between a great number of cases that have been generated by some multiplicative mechanism. I have much experience with a problem in which pebbles which are either red, white or blue, are--under a number of additional constraints to make the problem difficult enough-- to be sorted in the order of the Dutch National Flag. In a not uncommon approach --which from a point of efficiency seems quite reasonable-- people find themselves essentially faced with a case anaysis which has to cover 6 different cases. Six is a very high number and usually they get stuck somewhere in the design process. The answer is: think for a moment with how many cases you would be faced, if we had had five different colours, instead of only three! Then one will discover that the answer is 20 --if "n" is the number of colours, the formula is $n*(n-1)$--; as soon as that observation has been made, one sees that this multiplicative building up could have been avoided by looking at only one pebble at a time and the case analysis collapses to one with as many cases as we have colours, and from then onwards the problem is trivial. And, in retrospect, the discarded efficiency gain turns out to be close to negligible!

* * *

Developing programming methodologies is more than trying to become a more competent programmer: one must not only learn to become that, one must also be able to teach it. From the above it is clear that programming methodologies have close connections with problem solving, with effective ordering of one's thoughts, with at least one important aspect of thinking: how not to get lost! And the teachability of such aspects of thinking is not obvious. It is not so much that one must teach manipulative ability, for that can be done, it is done at all levels: arithmetic at the primary school, algebra at the secondary school, and symbolic logic at the university. Now we must try to teach how to think with the aim of reducing the need for otherwise exploding amounts of manipulation.

To give a course "How To Think Big Thoughts In Ten Easy Lessons" is obviously nonsense. But I firmly believe that something else is quite possible. That is, firstly, to drive home the message, that we should think with the purpose of reducing the amount of detailed reasoning eventually

needed; that is, secondly, to show the most common mechanisms by which the exploding need for manipulation is generated, i.e. don't only warn them for the symptoms, but also for as many causes as you can name explicitly. To a certain extent this can be done in the explicit style of University teaching.

It becomes harder when we know that we should try to separate concerns, but do not know how to untie and to disentangle the amorphous knot of the initial goals. Some striking examples, dealt with by an inspiring teacher can certainly do no harm. The extent to which we have to be content with teaching this in the style of the guilds, by showing only what we do and hoping that the apprentice will discover for himself how to do it, is still an open question for me. My experience over the last years is encouraging.

One thing is certain: learning how to program well requires a great amount of exercise and confrontation. Exercise in order to get the agility, confrontation in the sense that the learning student must struggle in order to discover how hard programming is: and after having constructed a one- or two-page program in four hours of hard work, show him an eight-line solution that a more competent programmer wrote down within fifteen minutes. It is cruel, but without that confrontation the student will remain stuck at his current level of competence, unless he can learn, all by himself, from his own experience. But people who can do that are extremely rare: if one of our students can do that, he will soon be our Master!

[1] Simon, Herbert A. The Sciences of the Artificial, MIT Press, 1969

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow
Plataanstraat 5
NUENEN - 4565
The Netherlands