Programming: from craft to scientific discipline.

Summary:

In response to the software crisis and to the lack of clear guidance in the design of programming languages, "programming methodology" emerged in the second half of the sixties, with the avowed purpose of discovering what would be involved in the design of sizeable high-quality programs. The recognition that high-quality implied correctness and that correctness could be proved in theory and might be provable in practice then became a major driving force. The proper roles of intuition and of formal discipline were reassessed, thereby reshaping the nature of the programmer's task and his way of working. After a general survey of this development and an indication of its significance, some of its consequences will be discussed, because it is certain to have great impact upon our educational practices in computing science and software engineering, on the constitution of our work force, on the division of intellectual labour and on the management of software development projects. (End of summary.)

When, in the late sixties, I coined the term "Structured Programming" I made a few serious mistakes. One mistake has been that I did not make it a registered trademark. Another mistake has been that I introduced the term without giving a definition for it. My only excuse is that I had not foreseen that the term, just the term, mind you, would spread like wildfire, and would become one of the most overworked buzzwords of the computing scene in the seventies (to such an extent, as a matter of fact, that since a number of years I myself have stopped using the term altogether). I would like to use this opportunity to explain why I did not define it and, yet, introduced it. Such an explanation seems a good introduction for a description of what has roughly happened since then: to give you a feeling for the significance of the development in programming during the last eight years is the main purpose of this talk.

* * *

For reasons that I shall mention in a moment, several of us felt that

the activity of computer programming had not only the potential, but also a great probability of changing drastically. Whether the change would be so slow as to be called "evolution" or so abrupt as to be called "revolution" was something we did not venture to predict. But the nature of the change itself was clear enough and we needed a name to label the development that, we felt, was about to occur. Hence I coined the term "Structured programming": I felt that it captured my observations and my hopes very nicely.

In the mid-sixties we observed what was then known as "the software failure" or "the software crisis". Its emergence was no surprise, its occurrence had been predicted. In not much more than ten years the power of commonly available computers had increased by a factor of thousand, and our programming ability had not increased in proportion. Besides that, the logically simple, sequential machines of the fifties had been replaced by much more complicated pieces of equipment, complicated by such features as multi-level stores and asynchronously active peripherals. It was very clear that the programming task was outgrowing our programming capacities and the emergence of the software crisis was, as said, no surprise. The problem, however, was that very little could be done about it, as long as its existence was hardly admitted or even denied: it is vain to urge a world to try to improve its programming habits as long as that world pretends that its programming habits are perfectly adequate. In this respect the conference on "Software Engineering", sponsored by the NATO Science Committee and held in Garmisch, Germany, in October 1968 was the great turning point: here the existence of the software crisis and the urgency of the situation was openly admitted by such an impressive collection of representative authorities, that this admission was sure to have its impact and to help create the climate in which a change in our programming habits could be discussed. This conference was one of the main reasons why we felt that for the expected change the time to happen at last had come.

Another reason was the fate of a number of committees trying to design a new and better programming language, such as the SHARE Committee designing PL/I and IFIP Working Group 2.1 trying to design a successor for ALGOL 60. I --and many share this opinion, although perhaps for different reasons-- consider both efforts, each in their own way, as most unsuccessful. It was already during the design phase of those languages that many of the people

originally involved became very doubtful as to whether things were developing in the right direction. Inexperienced as they were, they first blamed the committee mechanism, and the joke of the season was to define a camel as a horse designed by a committee. But on closer inspection it was discovered that not all the blame could be put on the committee mechanism, for there was a profounder reason: we did not know the nature of the programmer's task well enough.

Each tool shapes its users, and each programming language reflects, in its capacity as a tool, a picture of the programmer and his task. A rather intuitive, not very explicitly described but commonly accepted picture of the programmer and his task had given rise to FORTRAN and ALGOL 60. The failure to achieve striking improvements upon them was a direct consequence of the fact that our view of the programmer and his task had insufficiently evolved. The "typical programmer" still seemed to be the professional physicist or engineer who, for some technical computation, would write as a nonprofessional programmer a three-page program in an afternoon, i.e. very much the same prospective user that had inspired FORTRAN and ALGOL 60 some ten years earlier. Hence, for instance, the paralyzing stress on the requirement that the new language should be "easy to learn"; in practice this meant that the new language should not be too unfamiliar, and too often "convenient" was confused with "conventional". More and more people began to feel that tuning those designs to the supposed needs of the nonprofessional programmer was ..... for lack of any idea how a truly <u>professional</u> programmer would look like! We knew how the nonprofessional programmer could write in an afternoon a three-page program that was supposed to satisfy his needs, but how would the professional programmer design a thirty-page program in such a way that he could really justify his design? What intellectual discipline would be needed? What properties could such a professional programmer demand with justification from his programming language, from the formal tool he had to work with? All largely open questions. In an effort to find their answers a new field of scientific activity emerged in the very late sixties; it even got a name: it was called "Programming Methodology".

* * *

Programming methodology was in its infancy a rather vague and diffuse subject. More stress was certainly given to program correctness than to ex-

ecution efficiency. In some minds this has created the impression that programming methodology did not care about efficiency, but this impression is wrong. In the sorry state of the art that could be observed in those days it was not the inefficiency that seemed most alarming: most alarming seemed that all sizeable programs seemed to be bug-ridden. Efficiency of computer programs had already been such a fashionable topic for so many years, that the balance seemed to need some redressing. When programming methodology first focussed its attention on the problem of program correctness, it did so because many of us felt that that was a relevant concern that for too long a time had not received the attention it apparently deserved.

I called programming mathodology in its infancy vague and diffuse. In spite of the fact that Naur [1], Floyd [2], and Hoare [3] had already published their articles, we used in the beginning hardly the cruel and uncompromising expression "proving the correctness of a program". For at least yet another year their articles stood on the shelf reserved for interesting academic exercises without much practical significance. It was the time when people could make four different programs for the same problem and then could ponder for hours or days on the question which of the four versions they liked best. That was an exploratory activity that encountered little appreciation and even evoked criticism. The practitioners --the programmers of "the real world"-- could not see much significance in all those aesthetic exercises, the theoreticians --the mathematicians and logicians-- saw no depth and, therefore, no significance in them either. The fact that, as an alternative to the quantifiable efficiency, we turned to easthetic criteria has not without justification been qualified as an exaggerated preoccupation with "programming style".

Yet, all those experiments with little programs have not only been valuable, they were at that stage even necessary. Programming at that time was still an intuitive craft, and before the decision to adopt and to further develop a formal discipline can be taken, it should have been established with sufficient evidence that such a formal discipline is needed and that its further development is indeed worth the effort. And in order to prevent the formalization from becoming and end in itself, it should be sufficiently clear what it should achieve.

The exploratory stage, during which the notions of "simplicity" and "elegance" absorbed so much of our attention has had one very important effect. By the time that formal techniques became more general adopted, we had, for "aesthetic reasons" stripped our programming vehicles, removing many of the usual bells and whistles whose presence would only have encumbered the formal treatment. The beauty of some of the programs we had discovered was an incentive to look for correctness proofs of comparable beauty and the circumstance that our most beautiful programs were often by all the usual standards also very efficient gave us the encouraging feeling that the whole exercise made sense.

In short I think that the infancy of programming methodology, with its stress on aesthetic criteria, has not been wasted. As craftsmen that to a certain extent had become artists as well, we had already developed into better intuitive programmers; at the same time the subject matter worth of formal treatment had been filtered out. And, finally: the fact that we talked about "the beauty of a program" in very much the same way as in which mathematicians refer to "the beauty of a proof" provided an emotional link between two at that time rather disjoint cultures, a link that may very well have had a decisive influence.

* * *

In the above I have tried to sketch the emotional and intellectual climate of the infancy of programming methodology. I have done so in the hope that it will assist you in getting some appreciation of the significance of its later achievements, an appreciation I would like to transmit to you without fully going into the technicalities that would take a full semester to cover.

We should remember that, when all this started, programs were almost exclusively considered from the point of view of what would happen when the program would be executed by a computer. It was only via the class of possible computations that could be evoked under control of a program that such a program could be appreciated. Textbooks on programming used to begin with a few chapters devoted to the description of the average computer architecture and the global characteristics of the machine's major components. Efforts at the formal definition of the semantics of programming languages were almost exclusively so-called "operational definitions", i.e. in terms of the properties of the possible computational histories. The quality of a program was very

often equated to run-time efficiency, a notion which is, of course, highly implementation-dependent.

Yet, for progress it was necessary that the close tie to the process of program execution be loosened. One reason for this necessity was that as long as the tie was so close, characteristics of machines as they happened to exist, whether desirable or not, tended to pervade the thinking about programming: the way in which the properties of the IBM/360 had pervaded the design of PL/I was a warning not to be ignored. Another reason --although that one was perhaps discovered only later-- was that program correctness and cost of execution are two so important concerns, that the programmer who has to give full attention to both of them, should be given the mental tools to separate these two concerns.

Two concurrent developments made it possible to loosen the tie between a program and the corresponding class of computational histories. I have already mentioned them both.

The one development consisted of the many, many programming experiments made during the infancy pf programming methodology. As the problem of programming language design was one of the major incentives for these numerous programming experiments, many of these experiments were performed in tentative, unimplemented (and often yet incomplete) programming languages: the purpose of the experiments was very often to explore the consequences of a yet untried language feature. The fact that these experiments were mostly carried out in unimplemented languages loosened the tie between the programs and their executions. The other development was the discovery and application of the papers by Naur [1], Floyd [2], and Hoare [3], which dealt with the possibility of proving program correctness by means of a formal discipline. Such a formal discipline may have been inspired by what happens during program execution via a computer, by the time that the formal discipline is applied it can be used "in its own right" so to speak.

This second development was a necessary complement to the first one: it is all right to push the class of possible computational histories to the background of one's awareness, but this is only possible provided we have an

alternative technique for coming to grips with what a program "means": Naur and Floyd gave a proof technique, Hoare was the one who stressed most clearly that these proof rules could be regarded as axioms. This was an important discovery: from now onwards the proof rules need no longer be regarded as summarizing properties of computers, but they could be regarded as axioms, as postulates, as a functional specification for computing engines that those engines had better satisfy if they were to be useful engines. The discovery was important for its psychological side-effect: while in the past it was regarded as the purpose of our programs to instruct our computers, a shift to the opposite view could now take place, viz. that it is the purpose of our machines to execute our programs. Or, to put it in another way, logic which up to that moment had mostly been a descriptive science, fraught with metaphysics, now also admitted to be regarded as a prescriptive science, almost as a discipline of engineering.

* * *

The transition from the vague and emotional terms as "understandable", "clear", "readable" to the uncompromising and cruel notion of "a formal correctness proof" marks for Programming Methodology the transition from infancy to adolescence. It was a slow and sometimes painful process, like all processes of mental growth. From the people involved it required a greater agility in the propositional calculus and a greater familiarity with various induction patterns than most of them originally possessed. Younger computing scientists are free to laugh in either amazement or contempt, but I am not ashamed of confessing in public that five years ago, I was thoroughly familiar with the logical connectives "and" and "or" but certainly not with the implication, which I, therefore, tended to avoid, programming around it by replacing "a => b" by the more familiar "b or non a".

Besides this inherent cause that made the growth process a slow one, there was an external, and rather accidental one. Floyd's paper [2] was given greater publicity than Naur's earlier one [1]; we must conclude that Naur's paper was published ahead of its time. In contrast to Naur's paper that deals with programming, Floyd's paper has immediately been associated with mechanical verification --or even: discovery-- of formal proofs of the correctness of programs. As a sad result, Programming Methodology has for quite some years been in danger

of being killed in its youth by the superstition that underlies so much of the Artificial Intelligence activity, viz. that everything difficult is so boring that it had better be done mechanically.

I called the growth process, besides slow, also painful --like every adolescence, for that matter-- . For some time during its adolescence, programming methodology indeed had a very difficult time. This was when the first correctness proofs started to circulate: some of them were, indeed, appalling, even distressingly so. I was repelled by them, and at one occasion I declared, full of disgust, with emphasis that such formal techniques "were not my cup of tea". Those present at that occasion take a special delight in reminding me of it. For some time it indeed looked as if formal correctness proofs were totally unfit for human design and for human consumption.

Thank goodness there were also a few beautiful programs with beautiful correctness proofs hanging around and, as a result, Programming Methodology survived its adolescence without committing suicide. The point is that those convincing examples were very inspiring because "length of formal correctness proof" was immediately accepted by all people involved as an objective and relevant yardstick for "quality". Its objectivity caused among those people a greater unity of purpose than eloquence or money could ever have achieved. This unity of purpose was so welcome that programming methodology survived its first disappointing experiences with formal correctness proofs, until it had been discovered that many of those early proofs, indeed, had been unnecessarily ugly and cumbersome.

* * *

The first proofs were very cumbersome because, for lack of any theorems, they were built upon the axioms themselves. In the meantime a few general, but very powerful and useful theorems have been discovered and proved, and we have gained much experience in their effective exploitation.

A second cause for improvement was the discovery that the existence of such theorems and the ease with which they are formulated and used depends on the programming language used. The combinatorial freedom of the flowchart language that was used by Floyd in his fundamental article [2] creates problems

with the satisfactory solution of which people are still struggling today [4]; by adhering to a more strict sequencing discipline, these problems can be made to disappear.

A third cause for improvement was the discovery --in retrospect not very surprising-- that besides a formal theory about one's programming language and its constructs, one also needs a certain amount of formal theory about the subject matter of the computation. For instance: while proving the correctness of a parser it is not enough to have axioms about the relevant programming language constructs such as the operations on strings. Besides those one needs a certain amount of theory about sentences generated according to, say, a BNF-grammar; one may even be expected to need some theorems about the specific grammar of the language in question. In the beginning we often did not clearly separate those two different aspects of our proof obligations, thus confusing the issue.

A fourth improvement was perhaps the most spectacular. For many years the whole correctness issue had been posed in the following form "Given a program and given the specifications of what its execution should achieve, can you prove that the program meets these specifications?" The attention was thus focussed on "a posteriori" verification of <u>given</u> programs. It was then observed, however, that for <u>different</u> programs meeting the <u>same</u> specifications, the corresponding correctness proofs could greatly differ in complexity! And as a result, our picture of the programmer's task changed: it was no longer sufficient to design a correct program, in addition the program should be designed in such a way that its correctness could, indeed, be established. The simplicity of the corresponding correctness proof became thus an important aspect of program quality. But when this message was taken to heart, the programmer's task and his way of working changed radically. For, how does one develop a program that admits a nice correctness proof? Well, by developing the program and its correctness proof hand in hand. In actual fact the correctness proof is often even developed slightly ahead of the actual program text: as soon as the next step in the correctness proof has been chosen, the next refinement of the program is made in such a way that the chosen step in the correctness proof is applicable to it. Instead of seeking for a proof to go with a given program, we now construct a program to go with a chosen correctness proof! The later construction process is so well understood that we are now entitled to talk about a calculus for the formal derivation of programs [5].

Let me repeat. We are trying to find a "matching pair", consisting of a program and a proof and "matching" in the sense that the proof establishes that the programs meets its specifications. Given a program, finding a matching proof may be very hard; given a proof, finding a matching program is almost trivial. This is not the place to ponder about a mathematical or psychological explanation of this phenomenon. I have repeated and described the phenomenon in other words because of its drastic social impact. Here I use the term "drastic social impact" because it is bound to cause a change in a traditional division of labour.

* * *

Ten years ago, before the above had been understood, a tradition of "software production" had already established itself. It was regarded as the programmer's task "to produce programs", and the management of "software production" was organized under that assumption in close analogy to more traditional production processes such as those of cars, TV-sets or washing machines. This view of "software production" has had a few severe consequences.

1) In analogy to the production line worker, for "programmer productivity" the measure "number of lines of code produced per month" became among managers accepted. Doubts about its adequacy and significance have been voiced. It has been remarked that the adoption of this measure of programmer productivity is certain to encourage the production of insipid code. It has also been remarked that "code" is no end in itself, but only a means, and that rather than talking about the lines of code "produced" we should refer to the lines of code "used", and that, therefore, this "productivity measure" books the number of lines on the wrong side of the ledger. But large organizations have a great inertia, and the number of lines of code "produced", no matter how inadequate, is still in use as a grading criterion for programmers.

2) In analogy to the production line, software managers have tried to reduce software production costs by resorting to cheaper labour, which, in each given environment, means less educated people. The results of these tactics are only too well known.

3) In analogy to the production line, completely independent groups for

"quality assurance" have been installed: the people from "quality assurance" had to certify the software "products". Little it was understood, however, how impossible their task was. They could hardly do any better than to "certify" after the successful run of a set of test cases, no matter how inadequate they were for "certification". Also the results of this are only too well known.

So much for the traditional division of labour in the process of software production. This tradition has to be broken because it is based on a false assumption. The tradition will be broken because the modern way of developing programs is so dramatically more effective. In future we shall see in retrospect that today's traditional way of software production was the result when a craft was applied, where a scientific discipline was needed.

* * *

In the last decade programming has made the transition from craft to scientific discipline. It has been a development not unlike the one that took place in medieval painting in Europe. Before the discovery of the relevant rules of projective geometry, painters had only intuitive ideas about perspective. It was the old and experienced craftsman-painter who, on the average, was most successful in rendering the proportions well. But each new painting was in this respect an experiment that involved a certain risk, and, whenever a craftsman-painter had been exceptionally lucky, he had created a work of art that would become famous for its geometrical perfection. But within a few decades the old craftsmen had been superseded by a next generation of painters, mostly pupils of a certain Albrecht Dürer: these youngsters just knew the rules of perspective and produced without risk and with absolute confidence in this respect perfect paintings. Not only that they could do it, they knew that they could do it, they knew how they did it and could teach it to their successors. A next area of human endeavour had shown itself to be amenable to mathematical treatment, and a craft had been replaced by a scientific discipline. But whenever a craft is replaced by a scientific discipline, the old members of the guild feel themselves threatened, and quite understandably so.

I used the term "drastic social impact" because today's "programming guild"

encompasses --depending on how we count-- between 500.000 and 1.000.000 people, for the majority of whom it is totally unrealistic to expect that they can still acquire a scientific attitude. For them the recent developments in programming poses a serious problem, and their existence presents a serious barrier to the more wide-spread adoption of the newer programming techniques. In view of these conflicts it is very hard to predict how, when, and where the old programming tradition will be broken first. I shall just give you a few observations and leave the extrapolation to you.

When in 1968 the software crisis was for the first time openly admitted, a quite well-known professor of computing science waved the idea of correctness proofs away as being "idyllic", to another one the idea of correctness proofs caused "mental hiccups" [6].

As late as 1972, the suggestion that programming was so difficult that it deserved scientifically educated programmers has been waved away: with a reference to the then current intellectual calibre of "the average programmer" the suggestions was waved away as obvious nonsense.

Three years later, however, an International Conference on Software Reliability, where formal techniques played a predominant role, was held at Los Angeles and attended by about a thousand people.

Another year later, in 1976, a draft proposal for an advice to the U.S. government draws attention to the for them alarming circumstance that "by an accident of history, the United States has more inertia (and local vested interests) in current software practives than the rest of the world." It points out the danger for the USA that in the practice of software development they will be overtaken by nations that in this respect are yet more flexible.

So much for my few observation: I gladly leave to you to guess the how, the when, and the where of the breakthrough.

* * *

The conclusion that successful computer programming will eventually require a reasonable amount of scientific education of a rather mathematical

nature is not too welcome among the guildmembers: they tend to deny it and to create a climate in which "bringing the computer back to the ordinary man" is accepted as a laudable goal, and in which the feasibility of doing so is postulated, rather than argued. (This is understandable, because its infeasibility is much easier to argue.) They create a climate in which funds are available for all sorts of artificial intelligence projects in which it is proposed that the machine will take over all the difficult stuff so that the user can remain uneducated. I must warn you not to interpret the fact that such projects are sponsored as an indication that they make sense: the fact of their being sponsored is more indicative for the political climate in which this happens.

I am convinced that all these projects will fail, and that, the more ambitious they are, the more miserably they will do so. Hence I consider these projects as rather foolish, and for programming as a scientific discipline worth teaching, as rather harmless rearguard actions. As a bit pathetic, even.

In the meantime these projects can still do a lot of harm. They can do so by their false promises, pretending that the sophistication of their future systems, combined with decreasing hardware costs, makes it economically attractive to forsake our educational obligations to the next generation. Needless to say, falling into that seductive trap would be the cultural blunder of the decade.

In another respect I sometimes fear that the harm has already been done. There is a wide-spread folklore that in particular correctness proofs for computer programs are intrinsically so long, tedious, boring, uninteresting and prone to error, that the mechanization of their verification is a must. The assumption, however, is wrong: correctness proofs for programs can be --and should be!-- just as beautiful, fascinating and convincing as any other piece of mathematics. But the rumour to the contrary is constantly spread by the advertizing campaigns for the mechanical verification systems. The fact that the most outstanding feature of most artificial intelligence projects seems to be the heavy advertizing campaigns deemed necessary for their support, should instill into our minds a healthy mistrust and suspicion.

[1] Naur, P., "Proof of Algorithms by General Snapshots", BIT 6 (1966) pp. 310 - 316.

[2] Floyd, R.W., "Assigning Meanings to Programs", Proceedings of a Symposium in Applied Mathematics 19 (ed. Schwartz, J.T.), Providence, Rhode Island: American Mathematical Society, 1967, pp. 19 - 31.

[3] Hoare, C.A.R., "An Axiomatic Basis for Computer Programming", Comm.ACM 12, 10 (Oct. 1969), pp. 576 - 583

[4] Manna, Zohar and Waldinger, Richard J., "Is "sometime" sometimes better than "always"? Intermittent assertions in proving program correctness." Stanford Artificial Intelligence Laboratory Memo AIM-281 / Computer Science Department Report No. STAN-CS-76-558, June 1976.

[5] Dijkstra, Edsger W., "Guarded Commands, Nondeterminacy and Formal Derivation of Programs" Comm. ACM 18, 8 (Aug. 1975) 453 - 457.

[6] Naur, P. and Randell, B., "Software Engineering", Report on a Conference Sponsored by the NATO Science Committee, January 1969.

Plataanstraat 5
NL-4565 NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow