"Why is software so expensive?" An explanation to the hardware designer.

Recently I received an invitation from a sizeable (and growing) hardware company. For many years its traditional product line has been high-quality analog equipment; in a more recent past, however, digital components are beginning to play a more important role. The company's corporate management was aware of more or less unavoidably entering the (for the company unfamiliar) field of software, was aware of the existence of its many pitfalls without having a clear understanding of them, and I was invited to explain to the company's corporate management what the design of software is all about, why it is so expensive, etc.

Having many other obligations, I don't know yet whether I shall be able to accept the invitation, but, independent of that, the challenge absolutely delights me. Not only have I programmed for more than 25 years, but right from the beginning up till this very day I have done so in, over periods even close, cooperation with hardware designers, machine developers, prototype testers, etc. I think that I know the average digital hardware designer and his problems well enough to understand why he does not understand why designing software is so difficult. To explain the difficulty of software design to him is hard enough --almost as hard as explaining it to a pure mathematician-- , to explain it to a group of designers with their background and professional pride in high-quality <u>analog</u> equipment adds definitely a distinctive flavour to the challenge! Observing myself thinking about how to meet it and realizing that, even if I accept the invitation, my host will not have the exclusive rights of my explanation, I decided to take pen and paper. Hence this text.

* * *

To the economic question "Why is software so expensive?" the equally economic answer could be "Because it is tried with cheap labour." Why is it tried that way? Because its intrinsic difficulties are widely and grossly understimated. So let us concentrate on "Why is software design so difficult?". One of the morals of my answer will be that with inadequately educated personnel it will be impossible, with adequately educated software designers it might be possible, but will certainly remain difficult. I would like to stress, right at the start, that current problems in software design can only partly be explained by identified lack of competence of the programmers involved. I would like to do so right at the start, because that explanation, although not uncommon, is too facile.

It is understandable: it must be very frustrating for a hardware manager to produce what he rightly considers as a reliable machine with a splendid cost/performance ratio, and to observe thereafter that, by the time the customer receives the total system, that system is bug-ridden and its performance has dropped below the designer's worst dreams. And besides having to swallow that the software guys have ruined his product, he is expected to accept that while he works more and more efficiently every year, the software group is honoured for its incompetence by yearly increasing budgets. Without further explanations from our side we programmers should forgive him his occasional bitterness, for by accusing us of incompetence, he sins in ignorance.... And as long as <u>we</u> haven't been able to explain the nature of our problems clearly, <u>we</u> cannot blame him for that ignorance!

*　*　*

A comparison between the hardware world and the software world seems a good introduction for the hardware designer to the problems of his software colleague.

The hardware designer has to simulate a discrete machine by essentially analog means. As a result the hardware designer has to think about delays, slopes of signals, fan-in and fan-out, skew clocks, heat dissipation, cooling and power supply, and all the other problems of technology and manufacturing. Building essentially from analog components implies that "tolerances" are a very essential aspect of his component specifications; his quality control is essentially of a statistical nature, and, when all is said and done, quality assurance is essentially a probabilistic statement. The fact that with current quality standards the probability of correct operation is very, very high should not seduce us to forget its probabilistic nature: very high probability should not be confused with certainty (in the mathematical sense) and it is therefore entirely appropriate that no piece of equipment is delivered without being exercised by test programs. As technology is more and more pushed to its limits --and it is so all the time-- and tolerances become narrower and narrower, the control of these tolerances becomes a major concern for the hardware builders.

Compared to the hardware designer who constantly struggles with an unruly nature, the software designer lives in heaven, for he builds his artefacts from zeros and ones alone. A zero is a zero and a one is a one: there is no fuzziness about his building blocks and the whole engineering notion of something being "within tolerance" is just not applicable there. In this sense the programmer works indeed in a heavenly environment. The hypthetical one-hundred percent circuit designer who equates the problems of design and building with the problems of keeping the tolerances under control must be blind for the programming problems: once he has simulated the discrete machine correctly, all the really hard problems have been solved, haven't they?

To explain to the hardware world why programming yet presents problems, we must draw attention to a few other differences. In very general terms we can view "design" as bridging a gap, as composing an artefact of given components; as long as "the target artefact" and "the source components" don't change, we can reuse the old design. The fact that we need to design continuously is because they do change. Here, however, hardware and software designers have been faced with very different, almost opposite types of variation, change and diversity.

For the hardware designer the greatest variation has been in "the source components": as long as machines have been designed he has had to catch up with new technologies, he has never had the time to become fully familiar with his source material because before he had reached that stage, new components, new technologies had appeared on the scene. Compared to the drastic variation in his "source components", his "target artefact" has almost remained constant: all the time he has redesigned and redesigned the same few machines.

For the programmer the variation and diversity is just at the other end: the hardware designer's target is the programmer's starting point. The programmer's "source components" have been remarkably stable --in the eyes of some: even depressingly so!-- : FORTRAN and COBOL, still very much en vogue, are more than a quarter of a century old! The programmers finds the diversity at the other side of the gap to be bridged: he is faced with a collection of "target artefacts" of great diversity. Of very great diversity even; of an essentially very great diversity even, because here we find reflected that today's equipment, indeed, deserves the name "general purpose".

During the last decade software designers have carried on an almost religious debate on "bottom-up" versus "top-down" design. It used to be "bottom-up", I think that now the "top-down" religion has the majority as its adherents. If we accept the sound principle that, when faced with a many-sided problem, we should explore the area of our greatest uncertainty first (because the solution of familiar problems can be postponed with less risk), we can interpret the conversion of the programming community from "bottom-up" to "top-down" as a slow recognition of the circumstance that the programmer's greatest diversity is at the *other* side of the gap.

Besides being at the other side of the gap to be bridged, the variation and diversity the programmer is faced with is more open-ended. For the understanding of his source components the hardware designer has as a last resort always physics and electronics to fall back upon: for the understanding of his target problem and the design of algorithms solving it the software designer finds the appropriate theory more often lacking than not. How crippling the absence of an adequate theory can be has, however, only been discovered slowly.

With the first machine applications, which were scientific/technical, there were no such difficulties: the problem to be solved was scientifically perfectly understood and the numerical mathematics was available to provide the algorithms and their justification. The additional coding to be done, such as for the conversions between decimal and binary number system and for program loaders, was so trivial that common sense sufficed.

Since then we have seen, again and again, that for lack of appropriate theory, problems were tackled with common sense, while common sense turned out to be insufficient. The first compilers were made in the fifties without any decent theory for language definition, for parsing, etc., and they were full of bugs. Parsing theory and the like came later. The first operating systems were made without proper understanding of synchronization, of deadlock, danger of starvation, etc., and they too suffered from the defects that in hindsight were predictable. The indispensable theory, again, came later.

That people have to discover by trying that for some problems common sense alone is not a sufficient mental tool, is understandable. The problem is that by the time the necessary theory has been developed, the pre-scientific, intuitive approach has already established itself and, in spite of its patent insufficiency, is harder to eradicate than one would like to think. Here I must place a critical comment on a management practice that is not uncommon among computer manufacturers, viz. to choose as project manager someone with practical experience from an earlier, similar project: if the earlier project had been tackled by pre-scientific techniques, this is then likely to happen to the new project as well, even if the relevant theory is in the meantime available.

A second consequence of this state of affairs is that one of the most vital abilities of a software designer faced with a new task is the ability to judge whether existing theory and common sense will suffice, or whether a new intellectual discipline of some sort needs to be developed first. In the latter case it is absolutely essential not to embark upon coding before that necessary piece of theory is there. Think first! I shall return to this topic later, in view of its management consequences.

* * *

Let me now try to give you, by analogy and example, some feeling for the kind of thinking required.

Since IBM stole the term "structured programming" I don't use it anymore myself, but I lectured on the subject in the late sixties at MIT. A key point of my message was that (large) programs were objects without any precedent in our cultural history, and that the most closely analogous object I could think of was a mathematical theory. And I have illustrated this with the analogy between a lemma and a subroutine: the lemma is proved independently of how it is going to be used and is used independently of how it has been proved; similarly a subroutine is implemented independently of how it is going to be used and is used independently of how it has been implemented. Both were examples of "Divide and Rule": the mathematical argument is parcelled out in theorems and lemmata, the program is similarly divided up in processes, subroutines, clusters etc.

In the meantime I know that the analogy extends to the ways in which mathematical theories and programs are *developed*. By word of mouth I heard recently that Dana S.Scott described the design of a mathematical theory as an *experimental* science, experimental in the sense that adequacy and utility of new notations and concepts were determined experimentally, to wit: by trying to use them. This, now, is very similar to the way a design team tries to cope with the conceptual challenges it faces.

When the design is complete one must be able to talk meaningfully about it, but the final design may very well be something of a structure never talked about before. So the design team *must* invent its own language to talk about it, it *must* discover the illuminating concepts and invent good names for them. But it cannot wait to do so until the design is complete, for it needs the language in the act of designing! It is the old problem of the hen and the egg. I know of only one way of escaping from that infinite regress: invent the language that you seem to need, somewhat loosely wherever you aren't quite sure, and test its adequacy by trying to use it, for from their usage the new words will get their meaning.

Let me give you one example. In the first half of the sixties I designed as part of a multiprogramming system a subsystem whose function it was to abstract from the difference between primary and secondary store: the unit in which information was to be shuffled between storage levels was called "a page". When we studied our first design, it turned out that we could regard that only as a first approximation, because efficiency considerations forced us to give a subset of the pages in primary store a special status. We called them "holy pages", the idea being that, the presence of a holy page in primary store being guaranteed, access to them could be speeded up. Was this a good idea? We had to define "holy pages" in such a way that we could prove that their number would be bounded. Eventually we came up with a very precise definition which pages would be holy that satisfied all our logic and efficiency requirements, but all during these discussions the notion "holy" only slowly developed into something precise and useful. Originally, for instance, I remember that "holiness" was a boolean attribute: a page was holy or not. Eventually pages turned out to have a "holiness counter", and the original boolean attribute became the question whether the holiness counter was positive or not.

If during those discussions a stranger would have entered our room and would have listened to us for fifteen minutes, he would have made the remark "I don't believe that you know what you are talking about." Our answer would have been "Yes, you are right, and that is exactly why we are talking: we are trying to discover about precisely what we should be talking."

I have described this scene at some length because I remember it so well and because I believe it to be quite typical. Eventually you come up with a very formal and well-defined product, but this eventual birth is preceded by a period of gestation during which new ideas are tried and discarded or developed. That is the only way I know of in which the mind can cope with such conceptual problems. From experience I have learned that in that period of gestation, when a new jargon has to be created, an excellent mastery of their native tongue is an absolute requirement for all participants. A programmer that talks sloppily is just a disaster. Excellent mastery of his native tongue is my first selection criterion for a prospective programmer; good taste in mathematics is the second important criterion. (As luck will have it, they often go hand in hand.)

I had a third reason for describing the birth of the notion "holy" at some length. A few years ago I learned that it is not just a romantization, not just a sweet memory from a project we all liked: our experience was at the heart of the matter. I learned so when I wished to give, by way of exercise for myself, the complete formal development of a recursive parser for a simple programming language, defined in terms of some five or six syntactic categories. The only way in which I could get the formal treatment right was by the introduction of new syntactic categories! Those new syntactic categories characterized character sequences which were meaningless in the original programming language to be parsed, but indispensable for the understanding and justification of the parsing algorithm under design. My formal exercise was very illuminating, not because it had resulted in a nice parser, but because in a nice, formal nutshell it illustrated the need for the kind of invention software development requires: the new syntactic categories were exemplary for the concepts that have to be invented all the way long, concepts that are meaningless with respect to the original problem statement, but indispensable for the understanding of the solution.

* * *

I hope that the above gives you some feeling for the programmer's task. When dealing with the problems of software design, I must also devote a word or two to the phenomenon of the bad software manager. It is regrettable, but bad software managers do exist and, although bad, they have enough power to ruin a project. I have lectured all over the world to programmers working in all sorts of organizations, and the overwhelming impression I got from the discussions is that the bad software manager is an almost ubiquitous phenomenon: one of the most common reactions from the audience in the discussion after a lecture is "What a pity that our manager isn't here! We cannot explain it to him, but from you he would perhaps have accepted it. We would love to work in the way you have described, but our manager, who doesn't understand, won't let us." I have encountered this reaction so often that I can only conclude that, on the average, the situation is really bad. (I had my worst experience in a bank, with some government organizations as good seconds.)

In connection with bad managers I have often described my experience as a lecturer at IBM, Hursley, because it was so illuminating. Just before I came, the interior decorator had redone the auditorium, and in doing so he had replaced the old-fashioned blackboard by screen and overhead projector. As a result I had to perform in a dimly lighted room with my sunglasses on in order not to get completely blinded. I could just see the people in the front rows.

That lecture was one of the most terrible experiences in my life. With a few well-chosen examples I illustrated the problem solving techniques I could formulate at that time, showed the designer's freedom on the one hand, and the formal discipline needed to control it on the other. But the visible audience was absolutely unresponsive: I felt as if I were addressing an audience of puppets made from chewing gum. It was for me sheer torture, but I *knew* that it was a good lecture and with a dogged determination I carried my performance through until the bitter end.

When I had finished and the lights were turned up I was surprised by a shattering applause... from the back rows that had been invisible! It then turned out that I had had a very mixed audience, delighted programmers in the back rows and in the front rows their managers who were extremely annoyed at my performance: by openly displaying the amount of "invention" involved, I had presented the programming task as even more "unmanageable" than they already feared. From their point of view I had done a very poor job. It was at that occasion that I formulated for myself the conclusion that poor software managers see programming primarily as a management problem because they don't know how to manage it.

These problems are less prevalent in those organizations --I know a few software houses-- where the management consists of competent, experienced programmers (rather than a banker with colonial experience, but still too young to retire). One of the problems caused by the non-understanding software manager is that he thinks that his subordinates have to produce code: they have to solve problems, and in order to do so, they have to <u>use</u> code. To this very day we have organizations that measure "programmer productivity" by the "number of lines of code produced per month"; this number can, indeed, be counted, but they are booking it on the wrong side of the ledger, for we should talk about "the number of lines of code spent".

The actual coding requires great care and a non-failing talent for accuracy; it is labour-intensive and should therefore be postponed until you are as sure as sure can be that the program you are about to code is, indeed, the program you are aiming for. I know of one --very successful-- software firm in which it is a rule of the house that for a one-year project coding is not allowed to start before the ninth month! In this organization they know that the eventual code is no more than the deposit of your understanding. When I told its director that my main concern in teaching students computing science was to train them to think first and not to rush into coding, he just said "If you succeed in doing so, you are worth your weight in gold." (I am not very heavy).

But apparently, many managers create havoc by discouraging thinking and urging their subordinates to "produce" code. Later they complain that 80 percent of their labour force is tied up with "program maintenance", and blame software technology for that sorry state of affairs, instead of themselves. So much for the poor software manager. (All this is well-known, but occasionally needs to be said again.)

* * *

Another profound difference between the hardware and the software worlds is presented by the different roles of testing.

When, 25 years ago, a logic designer had cooked up a circuit, his next acts were to build and to try it, and if it did not work he would probe a few signals with his scope and adjust a capacitor. And when it worked he would subject the voltages from the power supply to 10 percent variations, adjust, etc. , until he had a circuit that worked correctly over the whole range of conditions he was aiming at. He made a product of which he could "see that it worked over the whole range". Of course he did not try it for "all" points of the range, but that wasn't necessary, for very general continuity considerations made it evident that it was sufficient to test the circuit under a very limited number of conditions, together "covering" the whole range.

This iterative design process of trial and error has been taken so much for granted that it has also been adopted under circumstances in which the continuity assumption that justifies the whole procedure, is not valid. In the case of an artefact with a discrete "performance space" such as a program, the assumption of continuity is <u>not</u> valid, and as a result the iterative design process of trial and error is therefore fundamentally inadequate. The good software designer knows this; he knows that from the observation that in the cases tried his program produced the correct result he is <u>not</u> allowed to extrapolate that his program is OK; therefore he tries to prove mathematically that his program meets the requirements.

The mere suggestion of the existence of an environment in which the traditional design process of trial and error is inadequate and where, therefore, mathematical proof is required, is unpalatable for those for whom mathematical proofs are beyond their mental grasp. As a result, the suggestion has encountered a considerable resistance, even among programmers who should know better. It is not to be wondered at that in the hardware world the recognition of the potential inadequacy of the testing procedure is still very rare.

Some hardware designers are beginning to worry, but usually not because they consider the fundamental inadequacy of the testing approach, but only because the "adjustment" has become so expensive since the advent of LSItechnology. But even without that financial aspect they should already worry, because in the meantime a sizeable fraction of their design activity does take place in a discrete environment.

Recently I heard a story about a machine --not a machine design by Burroughs, I am happy to add-- . It was a microprogrammed multiprocessor installation that had been speeded up by the addition of a slave store, but its designers had done this addition badly: when the two processors operated simultaneously on the two halves of the same word, the machine with the slave store reacted differently from the version without it. After a few months of operation a system breakdown was traced back to this very design error. By testing you just cannot hope to catch such an error that becomes apparent by coincidence. Clearly that machine had been designed by people who hadn't the foggiest notion about programming. A single competent programmer on that design crew would have prevented that blunder: as soon as you complicate the design of a multiprocessor installation by the introduction of a slave store, the obligation to <u>prove</u> --instead of just believing without convincing evidence-- that after the introduction of the slave store the machine still meets its original functional specifications is obvious to a competent programmer. (Such a proof doesn't seem to present any fundamental or practical difficulties either.) To convince hardware designers of the fact that they have moved into an environment in which their conventional experimental techniques for design and quality control are no longer adequate is one of the major educational challenges in the field.

I called it "major" because, as long as it isn't met, hardware designers won't understand what a software designer is responsible for. In the traditional engineering tradition, the completed design is the designer's complete product: you build an artefact and, lo and behold, it works! If you don't believe it, just try it and you will see that "it works". In the case of an artefact with a discrete performance space, the only appropriate reaction to the observation that it has "worked" in the cases tried, is: "So what?". The only convincing evidence that such a device with a discrete performance space meets its requirements includes a mathematical proof. It is a severe mistake to think that the programmer's products are the programs he writes; the programmer has to produce trustworthy solutions, and he has to produce and present them in the form of convincing arguments. Those arguments constitute the hard core of his product and the written program text is only the accompanying material to which his arguments are applicable.

* * *

Many software projects carried out in the past have been overly complex and, consequently, full of bugs and patches. Mainly the following two circumstances have been responsible for this:

1) dramatical increases of processor speeds and memory sizes, which made it seem as if the sky were the limit; only after the creation of a number of disastrously complicated systems it dawned upon us, that our limited thinking ability was the bottleneck;

2) a world that in its desire to apply those wonderful new machines became over-ambitious; many programmers have yielded to the pressure to stretch their available programming technology beyond its limits; this was not a very scientific behaviour, but perhaps stepping beyond the limit was necessary for discovering that limit's position.

In retrospect we can add two other reasons: for lack of experience programmers did not know how harmful complexity is, and secondly they did not know either, how much complexity can usually be avoided if you give your mind to it. Perhaps it would have helped if the analogy between a software design and a mathematical theory had been widely recognized earlier, because everyone knows that even for a single theorem the first proof discovered is seldom the best one: later proofs are often orders of magnitude simpler.

When C.A.R.Hoare writes --as he did early this year-- "...the threshold for my tolerance of complexity is <u>much</u> lower than it used to be" he reflects a dual development: a greater awareness of the dangers of complexity, but also a raised standard of elegance. The awareness of the dangers of complexity made greater simplicity a laudable goal, but at first it was entirely an open question whether that goal could be reached. Some problems may defy elegant solutions, but there seems overwhelming evidence that much of what has been done in programming (and in computing science in general) can be simplified drastically. (Numerous are the stories of the 30-line solutions concocted by a so-called professional programmer --or even a teacher of programming!-- that could be reduced to a program of 4 or 5 lines.)

To educate a generation of programmers with a much lower threshold for their tolerance of complexity and to teach them how to search for the truly simple solution is the second major intellectual challenge in our field. This is technically hard, for you have to instill some of the manipulative ability and a lot of the good taste of the mathematician. It is psychologically hard in an environment that confuses between love of perfection and claim of perfection and, by blaming you for the first, accuses you of the latter.

How do we convince people that in programming simplicity and clarity --in short: what mathematicians call "elegance"-- are not a dispensable luxury, but a crucial matter that decides between success and failure? I expect help from economic considerations. Contrary to the situation with hardware, where an increase in reliability has usually to be paid for by a higher price, in the case of software the unreliability is the greatest cost factor. It may sound paradoxical, but a reliable (and therefore simple) program is much cheaper to develop and use than a (complicated and therefore) unreliable one. This "paradox" should make us very hesitant to attach too much weight to a possible analogy between software design and more traditional engineering disciplines.

Plataanstraat 5
5671 AL Nuenen
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow