An Introduction to Implementation Issues.

The immediate cause of these essays is an elective course on Implementation Issues that I intend to give in the not too distant future to students of the Departments of Mathematics and of Electrical Engineering at the Eindhoven University of Technology. They are written for several reasons: there is the altruistic consideration that it is always nice for students to get some underlying material that they can study at leisure, there is the purely personal consideration that the writing of lecture notes is for me one of the most effective ways of preparing a set of lectures. They are written in English because I hope that eventually they will interest a wider public.

The course is "an introduction" in the true sense of the word. It is a true introduction because no prior knowledge of implementation techniques is assumed, and a general familiarity with the structure of automatic computers and with the main characteristics of programming languages should suffice. It is a true introduction also because it does not aim at "producing" all-round implementors. (Such a goal would require a much more encyclopaedic course.) On the contrary!

These essays are addressed primarily to three types of computer professionals --whether still student or graduated-- .

They are addressed to the hardware designer. Without making him an implementor they should give him a better understanding of the nature of the problems an implementor faces and of the ways in which machine characteristics can alleviate or aggrevate implementation problems.

They are addressed to the general computer user. Without making him an implementor they should give him --in particular if he cannot resist the temptation to suggest programming language improvements-- a better feeling of the nature of the processes evoked --"behind the scenes" so to speak-- by the executions of his programs.

Thirdly they are addressed to the man that is or hopes to become engaged in system design or development. They should give him a framework in which

to appreciate the main aspects of that activity that is more coherent than the mere enumeration of all the techniques that together constitute established practice.

At first sight it may seem presumptuous to hope to be able to write a text addressed at so varied an audience. Yet I don't only think that such a text can be written, I think that it should be written. Over the last two decades implementation --rightly or wrongly-- has become a very elaborate and intricate art, and the basic software in which the naked hardware is wrapped up has become almost impenetrable for the nonexpert, impenetrable to the extent that today it is equally well possible to meet a hardware designer that hasn't the foggiest notion what recursion is all about, as to encounter a computing scientist that has never seen a single bit.

The purpose of this course is the penetration of this "wrapper" of basic software. From the wealth of material collected in a period of twenty years of systems programming, I intend to mention as little accidental detail as possible, for it is exactly the overwhelming amount of accidental detail that has made the wrapper so difficult for the layman to penetrate. I shalln't shun detail wherever the subject matter doesn't allow me to do so, but it is my firm intention to reach my goal by abstracting from whatever can be identified as accidental. (Being abstract is something profoundly different from being vague: by abstraction --from what is uncertain or should be left open-- one creates a new semantic level on which one can again be absolutely precise.)

Of course I cannot avoid mentioning facts entirely; for the sake of self-consistency I have even to mention facts that can be assumed to be very familiar to some of my readers. Those readers I can only ask to bear with me, hoping --as I do-- that my way of presenting the familiar will give them a fresh appreciation of it.

The emergence of "basic software".

In the old days the general purpose automatic computer consisted of a processor that did the work, a store that kept the program and the intermediate results of the computation, an input device for feeding in program texts and data, and finally an output device for returning the results that were of interest in the outside world. Such was the naked machine and it was handed over to its user community almost as such. It was not unusual that the reading in via the input device of a program text to be stored in the computer store for later execution was itself a program-controlled activity of the computer. If this were the case the machine had some special means of loading the "basic input program" --usually no more than several tens of instructions-- under control of which the remaining programs necessary could be read in.

Several tens of instructions of the basic input program was all that was provided. Right from the start, however, the strategy for the intended use of the machine has included the notion of a so-called "library of standard subroutines". People immediately recognized subtasks that would occur in almost any application, conversions from decimal to binary number system during input and from binary to decimal number system during output being the obvious examples. Further examples were: the approximation of a quotient (for a machine without a built-in division), of the square root, of the logarithm, exponential, trigonometric, and other mathematical functions of general utility, programs for performing arithmetical operations on numbers in a floating-point representation (for machines without floating-point hardware) etc. And it was envisaged that from then onwards such a library of standard subroutines could be extended in a way which unavoidable would reflect more and more the specific characteristics of the intended area of application: programs for the numerical integration of differential equations, sorting programs, programs for matrix operations, complete linear programming packages, etc. The latter library extensions, which are so clearly application dependent, fall outside the scope of this lecture course that tries to concentrate upon the general (technical) issues that are pertinent to almost any computer application.

An example of such a general technical issue would be the following. Given that user communities like to tailor the neutral machine to the needs of their specific applications via a library of standard subroutines, observing that such subroutines are not complete programs in themselves but components from which complete programs should be composed, what are the techniques for composing programs from such standard components as a library can provide, and what should be our yardstick for evaluating such techniques?

For some of the "basic software" mentioned above, the need has disappeared; for instance, division and all the arithmetic operations on numbers in a floating-point representation are now quite commonly fairly directly built-in operations of the arithmetic units of the processors. (That they are often built in by means of a technique called "microprogramming" need not concern us here.) Several other circumstances, however, have caused a dramatic increase in the amount of basic software --now easily amounting to several hundred thousands instructions-- modern general purpose computers are equipped with before they are handed over to the user community. I mention the main ones.

1) The introduction of unsynchronized, concurrently active peripheral devices. Under control of special purpose subprocessors --called "channels"-- large amounts of information can be transferred between primary store and peripherals or secondary storage devices; because the exact moment when the channel will signal to the processor the completion of such a transfer is unpredictable, this form of concurrency has introduced nondeterminacy as a new element into the installation as a whole. The need to absorb the major part of this nondeterminacy, so that the total installation again presents itself as a (nearly) deterministic automaton, was one of the major reasons for the introduction of a new component of the basic software, called "the operating system". It took a considerable number of years before the nature of this task was sufficiently well understood. (Even in the early seventies it was unusual to consider concurrency and nondeterminacy as different issues, and it was a minor eye-opener for quite a few that both highly concurrent but structurally deterministic and purely sequential but structurally nondeterministic systems could be conceived and were a worthy subject of investigation.)

When the logical hurdles involved were taken, the extension to installations with two or more identical "central" processors was only a minor step.

2) In the older "batch mode" the machine executed the programs one after the other and always at most one computation was in an intermediate state of progress. In a multiprogrammed system the resources of the installation may be distributed over a larger number of independent computations that have been initiated but not yet completed. Multiprogramming has been introduced with the dual purpose of increasing the resource utilization and reducing the turn-around time of small jobs; as such it has been successful. (So-called "time-sharing systems" in which the central computer is coupled to a possibly large number of keyboard terminals are a later extension of the same idea: the added requirement of quick response defeats the goal of high resource utilization, but before this was discovered a lot of effort had been wasted in trying to combine the incompatible.) The scheduling of the resource sharing between independently conceived programs has caused a considerable complication of the operating systems, the more so because some logical problems --such as the prevention of deadlock and individual starvation-- were insufficiently well understood.

3) The introduction of (high-level) programming languages, such as FORTRAN, ALGOL 60, and COBOL to mention but a few of the first. Historically speaking the programming languages were an outgrowth of the so-called "auto-coders". The auto-coders were a mechanization of part of the clerical work involved in the production of machine code programs for a specific machine: they allowed variables to be denoted by identifiers, they allowed arithmetic expressions to be written down in the normal infix notation, they allowed subscription in one-dimensional arrays, and mechanized the incorporation of standard subroutines from the library. The use of autocoders was a great improvement over the writing of programs in machine code, a process in which all sorts of irrelevant decisions have to be taken --such as which storage locations to allocate to which variables-- , decisions the consequences of which, however, permeate all through the program text. Autocoders were primarily the result of the recognition that that trivial clerical labour had better be done by the machine itself. Their consequence that via the autocoder the machine could be used by people with a less detailed knowledge of the machine

was regarded as a fringe benefit, it was not their primary purpose.

The later "programming languages" were more ambitious; some of the hopes with which they were developed were, however, unrealistic. Their main purpose was to make the available computers "accessible to the nonprogrammer". It was at a time that a programmer's expertise was considered to be his intimate knowledge of the machine he was programming for, so that he could cunningly exploit the specific (and sometimes weird) machine characteristics. And it was hoped that specific machine characteristics could be hidden from the programming user to the extent that his programs written in a high-level programming language would be "machine independent", i.e. could be executed on any machine for which that programming language had been implemented. As in the case of autocoders, such an implementation consisted primarily of a translator or compiler that would accept a program text written in the programming language ("source code") and would produce an equivalent program in machine code ("target code"); the latter program in machine code --sometimes supported by a so-called "run-time system"-- could then be executed by the machine.

I mentioned that some of the hopes were unrealistic. The hope that now non-programmers had easy access to the machine turned out to be vain. It became clear that the programmer's expertise had been misjudged: it is not his intimate knowledge of the machine and his willingness to incorporate tricks, but his ability to conceive large algorithms in such an orderly fashion that the various cost aspects of program execution are predictable and the results produced trustworthy. And being up to this conceptual challenge intrinsically requires besides brains a professional training. Complete machine independence hasn't been achieved either. Some programming languages (early FORTRAN AND PL/I, for instance) were designed with a specific machine in mind, and indeed reflect their features. Other programming languages (ALGOL 60 and COBOL) were not so explicitly designed with a specific machine in mind, but either their definition left details --such as the arithmetic for floating point numbers-- to the implementors, or implementors were by economic considerations forced to deviate from the definition and to implement "a dialect". Furthermore, various programming language constructs influence in different implementations the cost of program execution differently, thereby making the notion of "the most efficient program" implementation dependent.

It seems that the majority of programmers appreciate the programming language they use less on account of its definition and more via the implementation of it they daily work with.

The above enumeration of circumstances that led to the emergence of basic software should not be regarded as complete.

* * *

The above mentions the circumstances that make the emergence of basic software understandable. It does not explain its dazzling complexity, nor its frightening size.

One way of explanation is viewing the final product as the result of the development process and of concentrating upon the latter. The complexity is then understandable as the compound effect of a few major and thousands of little mistakes, as the unavoidable compromise caused by the changing and incompatible pressures from the market place, etc.. Valid as such an analysis may be, with one noticeable exception I shall leave it to the expert in industrial sociology and the sociology of science.

On economy.

At irregular intervals --and sometimes not without reason-- the question is publicly raised why, and if so how much, society should support scientists in their endeavours, and at such moments scientists wonder --also sometimes not without reason-- why their fellow human beings tolerate them at all. Such questions are always asked in terms of the at the moment prevailng prejudices and tacit assumptions. (The answers proposed today are definitely quite different from those that were acceptable during the Enlightment.)

I have come to the conclusion that the average (and perhaps even the not-so-average) scientist feels less secure than he likes to believe, and that he yields more to the pressures of the prevailing prejudices and tacit assumptions of his fairly direct environment than is generally assumed. (I have come to this conclusion as it seems to be the only explanation for the phenomenon that some research topics are heavily pursued in some countries and practically ignored in others, despite the availability of the resources and the awareness of the topic.) The current tendency to justify research in terms of "usefulness" only adds to the scientist's feeling of uncertainty: the problem with "usefulness" as a criterion is that, besides being noble --and therefore hard to challenge-- the notion of "usefulness" is too much dependent on changing fashions to be of great value as a guiding principle for the long-range activity that scientific research always is. So much for the justification of scientific research in general.

In our particular area of automatic computing, efficiency considerations have always played a major role, even to the extent that sometimes efficiency seems to have become the sole concern. The great weight given to efficiency concerns is only too understandable. Firstly, big computers have always been very expensive. Secondly, experience has shown that the inadvertent introduction of gross inefficiencies is only too easy --and when I say "gross" I mean "gross"-- . Thirdly, the growth of the computer industry coincided with the more wide-spread introduction of quantitative techniques of industrial management, techniques which the young computer industry --unhampered by other traditions-- was one of the first to adopt. Fourthly --and this is probably the most profound reason-- avoiding waste is a core problem of computing science,

a subject that would evaporate into nothingness (or symbolic logic) if infinite computing resources were available (in very much the same way as the medical profession would collapse if mankind turned into a race of immortal gods).

I repeat: avoiding waste is a core problem of computing science, but let no reader translate "avoiding waste" inadvertently as "finding the most efficient solution". I most emphatically urge my readers not to do so, firstly because it is so commonly done, and secondly because it is a mistake, for on closer scrutiny the unqualified notion of efficiency is too vague to be helpful and more qualified notions of efficiency are too arbitrary to be of much significane.

Even the simplest batch mode environment, so simple that the cost of executing a program is proportional to its execution time, suffices tp illustrate this. In that environment one might think that of the programs to produce one specific result the one that can be executed in the least time should be regarded as the best program, but carrying this argument through ad absurdum, we would conclude that the best program is the one that doesn't use the machine at all! The point is clear: in the latter case we have moved the interface between "preparation" and "program execution" to the extreme that nothing has been left for the latter stage. In other words: computation times can only decide when all other things are equal, and this is seldom the case. In a program it may be observed that the full precision of a standard function routine that is called frequently is not needed and we may replace it by a special purpose routine that computes that function with less precision but faster. Even if that special purpose routine has been verified as thoroughly as the standard routine, the credibility of the result produced with the new program now depends in addition on the argument that the full precision of the standard routine was, indeed, not needed!

So much for a specific program made to be used once. For programs to be used with many different input data, the notion of "the most efficient program" usually becomes even more blurred. The execution times of two alternative programs may depend differently on the input data and the notion of the (on the average!) most efficient program may then depend (in a usually unknown way) on the (usually unknown) distribution of the input data to be expected in the future.

In the simple batch-processing system just considered, program execution time was in so far still a significant measure that it measured the extent to which the machine was not available for other tasks. In a multiprogrammed system the situation becomes even more confused, because the "cost" aspects of execution now include not only processor requirement but also storage requirement --in the sense that storage space needed for the execution of one program is not available for the concurrent execution of other programs. When now one program is derived from another by trading storage space against computation time, the question which of the two programs is "the more efficient" becomes entirely fictitious, as would be a statement of the nature that this cup of coffee is sweeter than that cup of tea is warm.

One way out of this dilemma --we might call it: the manager's approach-- is to reduce the multidimensional comparison to a one-dimensional one by only considering a linear composition with a chosen weight-factor --price, of course-- for each aspect. To quote from "Cost/Utilization: A Measure of System Performance" by Israel Borovits and Phillip Ein-Dor [Comm.ACM, 20, 3 (Mar.1977), 185 - 191]:

> "Cost is a common dimension which allows us to integrate utilization data for all the components of a system. We can then develop a single measure of cost/utilization for an entire system. While it is impossible to develop meaningful figures of total system performance directly from physical utilization, the common dimension of cost makes a single measure of merit both feasible and meaningful.
>
> The cost/utilization factor measures the extent to which the outlay on the total system is actually utilized. It is computed as $F = \sum_i P_i U_i$ where P_i is the cost of the i-th component as a percentage of total cost and U_i is the percentage utilization of the i-th component."

The above quotation is included on the principle "Audiatur et altera pars". I do not share the authors' high expectation of the meaningfulness of their cost/utilization factor F.

Firstly, I am not always sure how to define "the percentage utilization" of a component. How do I define it for my watch? How do I define it for my telephone if I use that for outgoing calls for 864 seconds a day? Is its percentage utilization then 0.01 or is it 0.99 because for 0.99 percent of the

time I have been reachable for callers from outside? (Or, to take an example from computing: what is the percentage utilization of a bounded buffer?)

Secondly, the authors suggest that high values for F are desirable, apparently unaware of the circumstance that (when utilization can be defined meaningfully) high resource utilization is in general incompatible with other desirable system properties such as fast response.

Thirdly, even in their own article the authors don't show how to use F-values and in their interpretation don't go beyond remarking that $F = 0$ corresponds to complete idleness and $F = 1$ to full utilization.

(To be continued)

Plataanstraat 5
5671 AL NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow