STICHTING
MATHEMATISCH CENTRUM
2e BOERHAAVESTRAAT 49
AMSTERDAM

DR 30

Some meditations on advanced programming.

(Information Processing 1962, N.Holland Publ.Coy., Amsterdam 1963, p 535-538).

E.W.Dijkstra.

1962

SOME MEDITATIONS ON ADVANCED PROGRAMMING

E. W. DIJKSTRA *)
Mathematisch Centrum, Amsterdam, Netherlands

In case you expect me to give a complete, well-balanced and neutral survey of the advanced programming activities of the world, I must warn you that I feel neither inclined nor entitled to do so.

My title already indicates that I am going to meditate on the subject, which is something quite different from giving a survey. Perhaps the title of my paper would have been more outspoken if it had been "My Meditations on Advanced Programming", for I intend to present a picture in the way I wish to see it; and I should like to do so in all honesty without any claim to objectivity. I intend to do so because I have a feeling that I serve you better by giving you an honest personal conviction than by presenting you with the colourless average of conflicting current opinions of other people.

You will observe that I shall fail to give you a generally acceptable definition of the subject "Advanced Programming". I think that, in my own appreciation of the subject, the description "Advancing Programming" would have been a better qualification. I like many activities which are worthy, I think, of the name "Advanced Programming" but I do not like these activities so much for the sake of their output, the programs that have resulted from them, as for what these activities can teach us. If I am willing to study them, to meditate upon them, I am willing to do so in the hope that this study or these meditations will give me a clearer understanding of the programmer's task, of his ends and his means. Therefore I should like to draw your attention in particular to those efforts and considerations which try to improve "the state of the Art" of programming, maybe to such an extent that at some time in the future we may speak of "the state of the Science of Programming".

A look around us will convince us that this improvement is very urgent for, on the whole, the programmers' world is a very dark one with only just the first patches of a brighter sky appearing on the horizon. For the present-day darkness in the programmers' world, the programmers themselves are responsible and nobody else. But before we put too much blame on them let us look for a moment at how their world came into existence.

When the first automatic electronic computers started to work more or less properly, mankind was faced with a new technical wonder, with a most impressive achievement of technical skill, and as a result, everybody was highly impressed, and rightly so. Under these circumstances it was completely natural that the structure of the early machines was mainly decided by the technical possibilities at the time. Under these circumstances it would have been an undreamt-of indecency for programmers to dare to suggest that those clever designers had not at all built the machines that the programmers wanted. Therefore, this thought hardly entered the programmers' minds. On the contrary: faced on the one hand with the new computers, and on the other with heaps of problems waiting for solution, they did their very best to accomplish the task with the equipment that had become available. They have accepted the full challenge. The potentialities of the computers have been exhausted to slightly beyond their utmost limits, the nearly impossible jobs have yet been done, by using the machines in all kinds of curious and tricky ways which were completely unintended and not even foreseen by the designers. In this atmosphere of pioneering, programming has arisen not as a science but as a craft, as an occupation where man, under the pressure of circumstances, was guided more by opportunism than by sound principles. This —I should like to call it "unhygienic"—creativity and shrewdness of the programmers has had a very bad influence on machine designers; for after some time they felt free to include all kinds of curious facilities of doubtful usability, reassuring themselves by their experience that, no matter how crazy a facility they provided, an even crazier programmer would always emerge that would manage to turn it into something profitable—as if this were sufficient justfication for its inclusion.

In the mean time, programming established itself as a discipline in which, on the whole, the standards of quality were extremely crude and primitive. The main—and often the only—possible virtues of a program were its quantitative characteristics, viz. its speed and its storage requirements. Space and time became the exclusive aspects of efficiency. In various places these standards are still in full vigour: not so long ago I heard of two cases, one where a machine was not bought because its multiplication speed was too low—and this may be a valid argument—and another case where a certain machine was selected because its multiplication was so fast. And this last decision was taken without the validity of this criterion being questioned.

Apart from the programs that have been produced, the programmers' contribution to human knowledge has been fairly useless. They have concocted thousands and thousands of ingenious tricks, but they have made this chaotic contribution without a mechanism to appreciate or evaluate these tricks, or to sort them out.

*) Since September 1962: Technological University, Eindhoven, Netherlands.

As many of these tricks could only be played by virtue of some special property of some special machine, their value was rather volatile. But the tricks were defended in the same of the semi-god "Efficiency," and for a long time there was hardly an inkling that there could be anything wrong with tricks. The programmer was judged by his ability to invent and his willingness to apply tricks. This opinion is still a wide-spread phenomenon: in advertisements for programmers, and in psychological tests for the job, it is often required that the man should be "puzzle-minded"; this in strong contrast to the opinion of the slowly growing group of people who think it more valuable that the man should have a clear and systematic mind.

However, as I told you, the sky above the programmers' world is brightening slowly. Before I go on to draw your attention to some discoveries that are responsible for this improvement, I should like to state as my opinion that it is relatively unimportant whether these are really new discoveries or whether they are rediscoveries of things perfectly well known to people like, say, Turing or von Neumann. For in the latter case, the important and new thing is that a greater number of people become aware of such a fact, and that a greater number of people realize that these are not just theoretical considerations but that they may have tangible, practical results. In this light, one might feel inclined to summarize the achievements of advanced programming as some purely educational successes: "At last programmers have started to educate one another to at least some extent." I shall not protest against this summary provided one agrees with my opinion that mutual education is one of the major difficult tasks of mankind.

One important rediscovery is that of the well-known equivalence of designing a machine and making a program. At this moment one might well ask oneself why I call attention to such a well-known fact. I have very good reasons to do so, for it has a great potential influence which is often overlooked: it enables the man who regards himself as a programmer to contribute to the field that is generally regarded as "machine design", and this is a very fortunate circumstance.

Some fifteen or ten years ago, the design and construction of a new, unique computer was a well-established and respectable occupation for university laboratories. Many of these "laboratory machines" were, each in their own private way, revolutionary contraptions. From then onwards this custom died out and design and construction of automatic computers became more and more an exclusively industrial activity. Five years ago most of us felt this as a perfectly natural development: construction of new computers became an extremely costly affair and it was generally felt that the time had come to leave this activity to the specialized industries. Now, five years later, we can only regret this development, for the computers on the market today are, on the whole, very disappointing. Certainly, they are faster, they are much more reliable than the old laboratory machines, but, on the other hand, they are often boring, uninspiring and hopelessly old-fashioned as well. For instance, the commercial requirement that all the programs made for some older machine from the same manufacturer should, without any modification, be acceptable to the new machine, has led to the design of new machines of which the order codes include the order code of the previous one in its entirety. Such a policy, however, is a never failing mechanism for prolonging the lifetime of previous mistakes. Some time ago, we were offered slogans about "the computers of the second generation", but to my taste many of them were as dull as their parents. Apparently a nice computer has at least one property in common with a gentleman, viz., that it takes at least three generations to produce one! Most of the industries, particularly the bigger ones, proved to be very conservative and reactionary. They seem to design for the customer who believes the salesman who tells him that machine so-and-so is just the machine he wants. But the poor customer who happens to know already, all by himself, what he wants is often forced to accept a machine with which he is already disgusted before the thing is installed in his establishment. Under present circumstances it is, commercially speaking, apparently not too attractive to put a nicer computer on the market. This is a sorry state of affairs; many a programmer suffers regularly from the monstrosity of his tool, and we can only hope for a better future with nicer machines. In the meantime he can program; taking some efficiency considerations for granted, he can force his machine to behave as he wishes: when making a programming system he designs a machine as it should have been. Thanks to the logical equivalence between designing a machine and making a program, programmers can contribute to future machine design, by exploring on paper, in software, the possibilities of machines with a more revolutionary structure.

The equivalence of making a program and designing a machine has another, maybe far-reaching consequence of a much more practical nature. It is not unusual to regard a classical computer as a sequential computer coupled to a number of communication mechanisms for input and output. Such a communication mechanism, however, performs in itself a sequential process—usually of a cyclic nature, but that feature is of no importance now. For this reason, we can regard a classical machine, its communication mechanism included, as a group of loosely connected sequential machines, with interlocks, where necessary, to prevent them getting too much out of phase with one another. The next step is to use the central computer, not for only one sequential process, but to equip it with the possibility of dividing its attention between an arbitrary number of such loosely connected sequential processes. One can do so with complete preservation of the symmetry between the sequential processes, to which a distinct piece of hardware corresponds on the one hand, and those which are taken care of by the central computer on the other hand, or even by one of the central processors, as the case may be. The difference between a modest and an ambitious installation may be that a couple of sequential processes, that in the modest installation are performed by the central computer, are performed by private hardware in the ambitious installation. But the above-mentioned equivalence between designing a machine and making a program, between performing a process either by hardware or by software, should be exploited to guarantee that a program acceptable for the one installation is also acceptable for the other. The above considerations are important because a

machine rigorously designed along these lines would greatly facilitate the manufacturer's task of equipping his product with the required software. The moral of this is that, if at the present moment many manufacturers have great difficulties in fulfilling their software obligations, and if one of the main sources of their trouble is that no two installations of the same machine are identical, their trouble could very well be a self-inflicted pain.

In this connection I should like to mention that I am fully aware of the fact that my previous picture of the commercial computer market was somewhat one-sided. Many of you will realize that at least one of the commercial products shows a great number of the "nice properties" just mentioned. In my opinion, this particular computer should be regarded as one of the brighter patches in the sky.

Now I want to turn my attention to one of the most important happenings in the programmers' world since the UNESCO Conference in 1959, viz., the publication of the famous "Report on the Algorithmic Language ALGOL 60", edited by Dr. P. Naur. I shall not discuss here the merits of the language ALGOL 60, nor shall I go into the question whether it has achieved its original aims or not. I intend to restrict myself to a discussion of the consequences of this publication, and of the influence it has had in the world of programming; for this influence has been tremendous. Briefly, I could formulate it as follows: "Through its merits ALGOL 60 has inspired a great number of people to make translators for it, through its defects it has induced a great number of people to think about the aims of a "Programming Language"." ALGOL 60, in all probability and in accordance with the intention of its authors, will be superseded by some better language in due time, but for much, much longer we shall be able to trace its educational effects.

Programming language, translator and computer, these three together form a tool, and in thinking about this tool as a whole, new dimensions have been added to the old concept of "reliability". In connection with the third of the three components, viz. the computer, concern about its reliability is as old as computers themselves; the acceptance test is a well-known phenomenon.

But what is the value of such an acceptance test? It is certainly no guarantee that the machine is correct—that the machine acts according to its specification. It only says that in these specific test programs the machine has worked correctly. If the design is based on some critical assumption, we can only conclude that in these test programs the corresponding critical situations apparently did not arise. If the design still contains errors, we can conclude that in these specific test programs these logical errors apparently did not matter. But as users, we are not interested in the test programs, we are interested in our own programs, and from the successful acceptance test we should like to conclude, that the machine works correctly in our programs also! But we cannot draw this conclusion.

The best thing a successful acceptance test can do is to strengthen our belief in the machine's correctness, and to increase the plausibility that it will perform any program in accordance with the specifications. The basic property of the user's program is that it will certainly require the machine to perform actions it has never done before. Machine designers have seen this difficulty quite clearly. They have realized that the successful acceptance test has only value as far as future programs are concerned, provided the actions performed in the test programs can be regarded as representative of all its possible operations, and they can only be representative by virtue of the clean and systematic structure of the machine itself. The above is common knowledge among machine designers; curiously enough, this is not true for translator makers, to whose activity the same considerations apply.

In order that the tool, consisting of programming language, translator and machine, be a reliable one, it is, of course, mandatory that all its components be reliable. One should expect that the translator maker, who in contrast to the machine designer has to deal with logical errors only, should do his job at least as well as the machine builder. But I am afraid that the converse is true. At the Rome Conference early in 1962, I was surprised to hear that the extensive translators for symbolic languages constructed in the USA continued to show up errors for years. I was shocked, however, when I saw the fatalistic mood in which this sorry state was accepted as the most natural thing in the world. This same attitude is reflected in the terms of reference of an ISO committee which deals with the standardization of programming languages: there one finds the recommendation to construct, for any standard language, a set of standard test examples on which any new translator for such a language could be tried out. But one finds no hint that the correct processing of these standard test examples is obviously only a trivial minimum requirement, no trace of the consideration that our belief in the correctness of a translator can never be founded on successful tests alone, but is ultimately derived from the clean and systematic structure of the translator and from nothing else. In deciding between reliability of the translation process on the one hand, and the production of an efficient object program on the other hand, the choice often has been decided in favour of the latter. But I have the impression that the pendulum is now swinging backwards.

For instance: if one gets a much more powerful machine in one's establishment than the one one had before, one can react to this in two different ways. The classical reaction is that the new machine is so much more expensive, that it is even more mandatory that no expensive computing time on the new machine should be wasted, that the new machine should be used as efficiently as possible, etc., etc. On the other hand one can also reason as follows: as the new machine is much faster, time does not matter so very much any more; as in the new computer the cost per operation is less than in the previous one, it becomes more realistic to investigate whether we can invest some of the machine's speed in other things than sheer production, say in convenience for the user—what we already do when we use a convenient programming language—or in elegance and reliability of the translator, thus increasing the quality of our output.

Also it is more widely recognized now than a couple of years ago that the construction of an optimizing translator is, essentially, a nasty job. Optimizing means improving the object program, i.e. making a more efficient object program than the one produced

by straightworward but reliable and trustworthy translation techniques. Optimization means "taking advantage of a special situation". If one optimizes in one respect, it is not an impossible burden to verify that the shortcut introduced in the object program does not lead to undesired results. If, however, one optimizes in two different respects, the duty of verification becomes much harder, for one has to verify, not only that the two methods are correct in themselves, but one must also check that they do not interfere with one another. If one optimizes in more diffffferent respects, the task of creating confidence in the correctness of the translator explodes exponentially. As a result it is no longer possible to recommend a computer by pointing to, say, the size of the translators available for it. On the contrary, the more extensive and shrewd a translator is, the more doubtful is its quality. Further, for the necessity of such extensive optimization efforts one might, finally, blame the computer in question; if one really needs such an intricate process as an optimizing translator to load one's programs, one feels inclined to defend the opinion that, apparently, the computer is not too well suited for its task. In short, the construction of intricate optimizing translators is an act, the wisdom of which is subject to doubt, and there is certainly a virtue in efforts to remove the need for them, e.g. the design of computers where these optimization tricks do not pay, or at least do not pay so much.

With regard to the structure of a translator, ALGOL 60 has acted as a great promoter of non-optimizing translators. The fact is that the language, as it stands, is certainly not an open invitation for optimization efforts. For those who thought that they knew how to write optimizing translators—be it for less flexible languages—this has been one of the reasons for rejecting ALGOL 60 as a serious tool. In my opinion these people bet on the wrong horse. I do not agree with them although I can sympathize with them; if one has solved a problem one tends to get attached to it, and if one likes one's solution for it, it is of course a little bit hard to switch over to an attitude in which the problem is not considered worth solving any more. The experience with ALGOL 60 translation has taught us still another thing. Some translator makers could not refrain from optimizing but, finding the task as such too difficult to do, they tried to ease matters by introducing additional restrictions into the language. The fact that their translators had only to deal with a restricted language, however, did not speed up translator construction; the task of exploiting the restrictions to full advantage has prevented this.

Thus we have arrived at the third component of our tool, viz. the language, and the language also should be reliable. In other words, it should assist the programmer as much as possible in the most difficult aspect of his task, viz. to convince himself—and those others who are really interested—that the program he has written down defines in fact the process he wants to define. Obviously the language rules should not contain traps of the kind of which there are still some in ALGOL 60, where, for instance, **'real array'** may be abbreviated to **'array'**, but **'own real array'** may not be abbreviated to **'own array'**. The next obvious requirement is that those rules which define a legal text do not leave any doubt as to whether a given text is legal or not, e.g., if there should be a restriction with respect to recursive use of a procedure, it should be clear under what conditions these restrictions apply, and in particular when the term "recursive use" applies. I mention this particular example because here it is by no means obvious. Finally, when faced with an undoubtedly legal text we want to be quite sure what it means. This implies that the semantic definition should be as rigorous as possible. In short, we need a complete and unambiguous pragmatic definition of the language, stating explicitly how to react to any text. So much for the necessity that the tool be reliable.

As my very last remark, I should like to stress that the tool as a whole should have still another quality. This is a much more subtle one and whether we appreciate it or not depends much more on our personal taste and education, and I shall not even try to define it. The tool should be charming, it should be elegant, it should be worthy of our love. This is no joke, I am terribly serious about this. In this respect the programmer does not differ from any other craftsman: unless he loves his tools it is highly improbable that he will ever create something of superior quality. Thus, at the same time these considerations tell us the greatest virtues a program can show: Elegance and Beauty.