A.P.I.C. Studies in Data Processing
No. 8

STRUCTURED PROGRAMMING

O.-J. DAHL
Universitet i Oslo,
Matematisk Institut,
Blindern, Oslo, Norway

E. W. DIJKSTRA
Department of Mathematics,
Technological University,
Eindhoven, The Netherlands

C. A. R. HOARE
Department of Computer Science,
The Queen's University of Belfast,
Belfast, Northern Ireland

1972
ACADEMIC PRESS
LONDON AND NEW YORK

PREFACE

In recent years there has been an increasing interest in the art of computer programming, the conceptual tools available for the design of programs, and the prevention of programming oversights and error. The initial out- standing contribution to our understanding of this subject was made by E. W. Dijkstra, whose Notes on Structured Programming form the first and major section of this book. They clearly expound the reflections of a brilliant programmer on the methods which he has hitherto unconsciously applied; there can be no programmer of the present day who could not increase his skills by a study and conscious application of these principles.

In the second monograph I have tried to describe how similar principles can be applied in the design of data structures. I have suggested that in analysing a problem and groping towards a solution, a programmer should take advantage of abstract concepts such as sets, sequences, and mappings; and judiciously postpone decisions on representation until he is constructing the more detailed code of the program. The monograph also describes a range of useful ideas for data representation, and suggests the criteria relevant for their selection.

The third monograph provides a synthesis of the previous two, and expounds the close theoretical and practical connections between the design of data and the design of programs. It introduces useful additional methods for program and data structuring which may be unfamiliar to many programmers. The examples show that structured programming principles can be equally applied in "bottom-up" as in "top-down" program design. The original inspiration, insight, and all the examples were contributed by O.-J. Dahl; I have only assembled the material, and added some additional explanations where I found it difficult to understand.

June 1972 C. A. R. HOARE

Contents

I Notes on Structured Programming

EDSGER W. DIJKSTRA

1. TO MY READER

These notes have the status of "Letters written to myself": I wrote them down because, without doing so, I found myself repeating the same arguments over and over again. When reading what I had written, I was not always too satisfied.

For one thing, I felt that they suffered from a marked verbosity. Yet I do not try to condense them (now), firstly because that would introduce another delay and I would like to "think on", secondly because earlier experiences have made me afraid of being misunderstood: many a programmer tends to see his (sometimes rather specific) difficulties as the core of the subject and as a result there are widely divergent opinions as to what programming is really about.

I hope that, despite its defects, you will enjoy at least parts of it. If these notes prove to be a source of inspiration or to give you a new appreciation of the programmer's trade, some of my goals will have been reached.

Prior to their publication in book form, the "Notes on Structured Programming" have been distributed privately. The interest then shown in them, for which I would like to express my gratitude here, has been one of the main incentives to supplement them with some additional material and to make them available to a wider public. In particular I would like to thank Bob Floyd, Ralph London and Mike Woodger for their encouraging comments and Peter Naur for the criticism he expressed. Finally I would like to express my gratitude to Mrs. E. L. Dijkstra-Tucker for her kind assistance in my struggles with the English language.

2. ON OUR INABILITY TO DO MUCH

I am faced with a basic problem of presentation. What I am really concerned about is the composition of large programs, the text of which may be, say, of the same size as the whole text of this booklet. Also I have to include ex-

amples to illustrate the various techniques. For practical reasons, the demonstration programs must be small, many times smaller than the "life-size programs" I have in mind. My basic problem is that precisely this difference in scale is one of the major sources of our difficulties in programming!

It would be very nice if I could illustrate the various techniques with small demonstration programs and could conclude with "...and when faced with a program a thousand times as large, you compose it in the same way." This common educational device, however, would be self-defeating as one of my central themes will be that any two things that differ in some respect by a factor of already a hundred or more, are utterly incomparable.

History has shown that this truth is very hard to believe. Apparently we are too much trained to disregard differences in scale, to treat them as "gradual differences that are not essential". We tell ourselves that what we can do once, we can also do twice and by induction we fool ourselves into believing that we can do it as many times as needed, but this is just not true! A factor of a thousand is already far beyond our powers of imagination!

Let me give you two examples to rub this in. A one-year old child will crawl an all fours with a speed of, say, one mile per hour. But a speed of a thousand miles per hour is that of a supersonic jet. Considered as objects with moving ability the child and the jet are incomparable, for whatever one can do the other cannot and vice versa. Also: one can close one's eyes and imagine how it feels to be standing in an open place, a prairie or a sea shore, while far away a big, reinless horse is approaching at a gallop, one can "see" it approaching and passing. To do the same with a phalanx of a thousand of these big beasts is mentally impossible: your heart would miss a number of beats by pure panic, if you could!

To complicate matters still further, problems of size do not only cause me problems of presentation, but they lie at the heart of the subject: widespread underestimation of the specific difficulties of size seems one of the major underlying causes of the current software failure. To all this I can see only one answer, viz. to treat problems of size as explicitly as possible. Hence the title of this section.

To start with, we have the "size" of the computation, i.e. the amount of information and the number of operations involved in it. It is essential that this size is large, for if it were really small, it would be easier not to use the computer at all and to do it by hand. The automatic computer owes it right to exist, its usefulness, precisely to its ability to perform large computations

where we humans cannot. We want the computer to do what we could never do ourselves and the power of present-day machinery is such that even small computations are by their very size already far beyond the powers of our unaided imagination.

Yet we must organize the computations in such a way that our limited powers are sufficient to guarantee that the computation will establish the desired effect. This organizing includes the composition of the program and here we are faced with the next problem of size, viz. the length of the program text, and we should give this problem also explicit recognition. We should remain aware of the fact that the extent to which we can read or write a text is very much dependent on its size. In my country the entries in the telephone directory are grouped by town or village and within each such group the subscribers are listed by name in alphabetical order. I myself live in a small village and given a telephone number I have only to scan a few columns to find out to whom the telephone number belongs, but to do the same in a large city would be a major data processing task!

It is in the same mood that I should like to draw the reader's attention to the fact that "clarity" has pronounced quantitative aspects, a fact many mathematicians, curiously enough, seem to be unaware of. A theorem stating the validity of a conclusion when ten pages full of conditions are satisfied is hardly a convenient tool, as all conditions have to be verified whenever the theorem is appealed to. In Euclidean geometry, Pythagoras' Theorem holds for any three points A, B and C such that through A and C a straight line can be drawn orthogonal to a straight line through B and C. How many mathematicians appreciate that the theorem remains applicable when some or all of the points A, B and C coincide? Yet this seems largely responsible for the convenience with which Pythagoras Theorem can be used.

Summarizing: as a slow-witted human being I have a very small head and I had better learn to live with it and to respect my limitations and give them full credit, rather than to try to ignore them, for the latter vain effort will be punished by failure.

3. On The Reliability Of Mechanisms

Being a programmer by trade, programs are what I am talking about and the true subject of this section really is the reliability of programs. That, nevertheless, I have mentioned "mechanisms" in its title is because I regard

programs as specific instances of mechanisms, and that I wanted to express, at least once, my strong feeling that many of my considerations concerning software are, mutatis mutandis, just as relevant for hardware design.

Present-day computers are amazing pieces of equipment, but most amazing of all are the uncertain grounds on account of which we attach any validity to their output. It starts already with our belief that the hardware functions properly.

Let us restrict, for a moment, our attention to the hardware and let us wonder to what extent one can convince oneself of its being properly constructed. Some years ago a machine was installed on the premises of my University; in its documentation it was stated that it contained, among many other things, circuitry for the fixed-point multiplication of two 27-bit integers. A legitimate question seems to be: "Is this multiplier correct, is it performing according to the specifications?".

The naïve answer to this is: "Well, the number of different multiplications this multiplier is claimed to perform correctly is finite, viz. 2^{54}, so let us try them all." But, reasonable as this answer may seem, it is not, for although a single multiplication took only some tens of microseconds, the total time needed for this finite set of multiplications would add up to more than 10,000 years! We must conclude that exhaustive testing, even of a single component such as a multiplier, is entirely out of the question. (Testing a complete computer on the same basis would imply the established correct processing of all possible programs!)

A first consequence of the 10,000 years is that during its life-time the multiplier will be asked to perform only a negligible fraction of the vast number of all possible multiplications it could do: practically none of them! Funnily enough, we still require that it should do any multiplication correctly when ordered to do so. The reason underlying this fantastic quality requirement is that we do not know in advance, which are the negligibly few multiplications it will be asked to perform. In our reasoning about our programs we talk about "the product" and have abstracted from the specific values of the factors: we do not know them, we do not wish to know them, it is not our business to know them, it is our business not to know them! Our wish to think in terms of the concept "the product", abstracted from the specific instances occurring in a computation is granted, but the price paid for this is precisely the reliability requirement that *any* multiplication of the vast set will be performed correctly. So much for the justification of our desire for a correct multiplier.

But how is the correctness established in a convincing manner? As long as the multiplier is considered as a black box, the only thing we can do is "testing by sampling", i.e. offering to the multiplier a feasible amount of factor pairs and checking the result. But in view of the 10,000 years, it is clear that we can only test a negligible fraction of the possible multiplications. Whole classes of in some sense "critical" multiplications may remain untested and in view of the reliability justly desired, our quality control is still most unsatisfactory. Therefore it is not done that way.

The straightforward conclusion is the following: a convincing demonstration of correctness being impossible as long as the mechanism is regarded as a black box, our only hope lies in not regarding the mechanism as a black box. I shall call this "taking the structure of the mechanism into account".

From now onward the type of mechanisms we are going to deal with are programs. (In many respects, programs are mechanisms much easier to deal with than circuitry, which is really an analogue device and subject to wear and tear.) And also with programs it is fairly hopeless to establish the correctness beyond even the mildest doubt by testing, without taking their structure into account. In other words, we remark that the extent to which the program correctness can be established is not purely a function of the program's external specifications and behavior but depends critically upon its internal structure.

Recalling that our true concern is with really large programs, we observe as an aside that the size itself requires a high confidence level for the individual program components. If the chance of correctness of an individual component equals p, the chance of correctness of a whole program, composed of N such components, is something like

$$P = p^n.$$

As N will be very large, p should be very, very close to 1 if we desire P to differ significantly from zero!

When we now take the position that it is not only the programmer's task to produce a correct program but also to demonstrate its correctness in a convincing manner, then the above remarks have a profound influence on the programmer's activity: the object he has to produce must be usefully structured.

The remaining part of this monograph will mainly be an exploration of what program structure can be used to good advantage. In what follows it

will become apparent that program correctness is not my only concern, program adaptability or manageability will be another. This stress on program manageability is my deliberate choice, a choice that, therefore, I should like to justify.

While in the past the growth in power of the generally available equipment has mitigated the urgency of the efficiency requirements, this very same growth has created its new difficulties. Once one has a powerful machine at one's disposal one tries to use it and the size of the problems one tackles adjusts itself to the scope of the equipment: no one thinks about programming an algorithm that would take twenty years to execute. With processing power increased by a factor of a thousand over the last ten to fifteen years, Man has become considerably more ambitious in selecting problems that now should be "technically feasible". Size, complexity and sophistication of programs one should like to make have exploded and over the past years it has become patently clear that on the whole our programming ability has not kept pace with these exploding demands made on it.

The power of available equipment will continue to grow: we can expect manufacturers to develop still faster machines and even without that development we shall witness that the type of machine that is presently considered as exceptionally fast will become more and more common. The things we should like to do with these machines will grow in proportion and it is on this extrapolation that I have formed my picture of the programmer's task.

My conclusion is that it is becoming most urgent to stop to consider programming primarily as the minimization of a cost/performance ratio. We should recognize that already now programming is much more an intellectual challenge: the art of programming is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible.

My refusal to regard efficiency considerations as the programmer's prime concern is not meant to imply that I disregard them. On the contrary, efficiency considerations are recognized as one of the main incentives to modifying a logically correct program. My point, however, is that we can only afford to optimize (whatever that may be) provided that the program remains sufficiently manageable.

Let me end this section with a final aside on the significance of computers. Computers are extremely flexible and powerful tools and many feel that their application is changing the face of the earth. I would venture the opinion that as long as we regard them primarily as tools, we might grossly underestimate

their significance. Their influence as tools might tum out to be but a ripple on the surface of our culture, whereas I expect them to have a much more profound influence in their capacity of intellectual challenge!

Corollary of the first part of this section:

Program testing can be used to show the presence of bugs, but never to show their absence!

4. On Our Mental Aids

In the previous section we have stated that the programmer's duty is to make his product "usefully structured" and we mentioned the program structure in connection with a convincing demonstration of the correctness of the program.

But how do we convince? And how do we convince ourselves? What are the typical patterns of thought enabling ourselves to understand? It is to a broad survey of such questions that the current section is devoted. It is written with my sincerest apologies to the professional psychologist, because it will be amateurishly superficial. Yet I hope (and trust) that it will be sufficient to give us a yardstick by which to measure the usefulness of a proposed structuring.

Among the mental aids available to understand a program (or a proof of its correctness) there are three that I should like to mention explicitly:

(1) Enumeration

(2) Mathematical induction

(3) Abstraction.

4.1. On enumeration

I regard as an appeal to enumeration the effort to verify a property of the computations that can be evoked by an enumerated set of statements performed in sequence, including conditional clauses distinguishing between two or more cases. Let me give a simple example of what I call "enumerative reasoning".

It is asked to establish that the successive execution of the following two statements

$$\text{“}dd := dd/2;$$
$$\textbf{if } dd <= r \textbf{ do } r := r - dd\text{”}$$

operating on the variables "r" and "dd" leaves the relations

$$0 \leq r < dd \tag{1}$$

invariant. One just "follows" the little piece of program assuming that (1) is satisfied to start with. After the execution of the first statement, which halves the value of dd, but leaves r unchanged, the relations

$$0 \leq r < 2 \times dd \tag{2}$$

will hold. Now we distinguish two mutually exclusive cases.

(1) $dd \leq r$. Together with (2) this leads to the relations

$$dd \leq r < 2 \times dd \tag{3}$$

In this case the statement following **do** will be executed, ordering a decrease of r by dd, so that from (3) it follows that eventually

$$0 \leq r < dd,$$

i.e. (1) will be satisfied.

(2) **non** $dd \leq r$ (i.e. $dd > r$). In this case the statement following **do** will be skipped and therefore also r has its final value. In this case "$dd > r$" together with (2), which is valid after the execution of the first statement leads immediately to

$$0 \leq r < dd$$

so that also in the second case (1) will be satisfied.

Thus we have completed our proof of the invariance of relations (1), we have also completed our example of enumerative reasoning, conditional clauses included.

4.2. ON MATHEMATICAL INDUCTION

I have mentioned mathematical induction explicitly because it is the only pattern of reasoning that I am aware of that eventually enables us to cope with loops (such as can be expressed by repetition clauses) and recursive procedures. I should like to give an example.

Let us consider the sequence of values

$$d_0,\ d_1,\ d_2,\ d_3,\ \ldots \tag{1}$$

given by

$$\text{for } i = 0 \qquad d_i = D \tag{2a}$$

$$\text{for } i > 0 \qquad d_i = f(d_{i-1}) \tag{2b}$$

where D is a given value and f a given (computable) function. It is asked to make the value of the variable "d" equal to the first value d_k in the sequence that satisfies a given (computable) condition "prop". It is given that such a value exists for finite k. A more formal definition of the requirement is to establish the relation

$$d = d_k \tag{3}$$