THE ART OF PROGRAMMING

Professor dr. E. W. Dijkstra

Department of Mathematics,
Technological University,
Post Box 513, Eindhoven,
The Netherlands.

Abstract:

Programming is a very difficult task. In order to improve our understanding we should try to structure programs in some nice manner. This paper indicates an approach to this problem and discusses its implications on writing a program arising from some given problem.

Rapporteurs:

Mr. P. Henderson
Mr. R. Snowdon

THE ART OF PROGRAMMING

Professor dr. E. W. Dijkstra

Department of Mathematics,
Technological University,
Post Box 513, Eindhoven,
The Netherlands.

Abstract:

Programming is a very difficult task. In order to improve our understanding we should try to structure programs in some nice manner. This paper indicates an approach to this problem and discusses the implications on writing a program arising from some given problem.

Preparation:

Dr. E. Dijkstra
Mr. H. Bekker

Introduction

'The efforts of instruction are seldom of much efficacy, except in those happy dispositions where they are almost superfluous'.

Gibbon: 'Decline and Fall of Roman Empire'.

'La plus belle ruse du Diable est de nous persuader qu'il n'existe pas'.

Baudelaire.

Before embarking on this subject it is useful to make a short list of misunderstandings. Such misunderstandings are fostered by advertisements, by programming courses and by instruction manuals.

1. Programming is easy if a programmer knows how to get his underlinings and semi-colons in the right place or how to write fluently in capital letters.
2. All a programmer needs to know is how to use a flow charting stencil properly.
3. All a programmer needs to know is how to write format statements and job control cards.
4. Programming is easy if your machine is big enough.
5. A programmer need only learn not to make mistakes.

The last two misunderstandings are more subtle than the first three. Contrary to these misunderstandings, programming is potentially very, very difficult. Analysing the task of programming leads to the conclusion that programming is a gigantic intellectual challenge without any precedent in the history of mankind. It is one of the most difficult branches of mathematics. During the past 15 years or so, machines have become over a thousand times more powerful, applications have become more ambitious and hance the scope of the programmer's duty has exploded. On the other hand the mathematical basis for programming is very simple compared with some other areas of mathematics. The challenge is to combine these two things, on the one hand to be able to build highly sophisticated programs and yet intellectually to control the whole activity as completely as is dictated by its simple mathematical basis. This is the gigantic intellectual challenge.

This is a lecture in four movements:

1. Why do we need nicely structured programs?
2. What makes programs nicely structured?
3. How do we build nicely structured programs?
4. How do we teach how to build nicely structured programs?

1. On our inability to do much

For practical reasons demonstration programs must be many times smaller than 'life-size programs'. In practice we are faced with large (1,000 page) programs and it would be pleasant if the structuring which we may demonstrate in small programs carried over directly to these larger programs. This is not the case. One of the major sources of difficulty in programming is that a factor of one thousand is so large that if anything is a thousand times as large, as fast, or as difficult as something else, then this difference is so tremendous that the result strikes us as something entirely different. The only way to overcome this is to draw attention to the effects of size explicitly, to point out to the students that 'a thousand times' is a few orders of magnitude beyond their imagination. They do not believe this to begin with and the only recourse is to demonstrate it with metaphors. In a small village, to scan a few columns of a telephone directly in order to find that name corresponding to a particular number is not difficult, but to do the same in a large city would be a major data processing task.

The misunderstanding that programming is easy provided you make no mistakes, suggests that programs are sometimes wrong. Of course, it is not difficult to write a program of the size of a small booklet as long as it does not need to work. The requirement that it should work is what makes programming such a great intellectual challenge.

2. On the reliability of mechanisms

If we have a program constructed from N components each with a probability of correctness p then the probability that the program is correct is P, where

$$P \leq p^N$$

As N will probably be very large, p should be very, very close to 1 if we desire P to differ significantly from zero. The sheer size of programs, therefore, focuses attention on the correctness problem and thus requires a much larger confidence level in the individual components. On what basis then can we increase this confidence level?

As an example of how not to do it let us consider a multiplier for two 27-bit integers. Since machines are so fast let us perform all possible multiplications. This, however, involves $2^{27} \times 2^{27}$ different multiplications, and even assuming our machine is capable of 2^{14} multiplications per second this will take longer than 30,000 years to complete. Although the total number of multiplications which will ever be performed during the lifetime of the computer will be a negligible fraction of those of which it is capable, we still require it to perform these correctly. The moral of this story is that if you consider a mechanism as a black box then the only way of convincing oneself of the correctness of this mechanism is by exhaustive testing, and this is out of the question for practical reasons. It is unsatisfactory from a theoretical point of view also, because we cannot be convinced that the mechanism has been put through all its internal states. We must, therefore, find other ways of convincing ourselves that a program works. As a corollary we see that program testing can be used to show the presence of bugs, but never to show their absence. We conclude that it is necessary to take the structure of the mechanism into account.

Rather than trying to find proof techniques to establish the correctness of arbitrary programs let us try to find those elementary structures for programs such that the intellectual effort needed to prove the correctness does not explode.

3. On our mental aids

What patterns of reasoning we do have at our disposal, how do we apply them and what are the consequences? Among the mental aids at our disposal are:

1. Enumerative reasoning.
2. Mathematical induction.
3. Abstraction.
4. Intuitive stroke of genius.

Only the first three will be considered here.

3.1 Enumeration

Given a specification of the net effect of the execution of statements $S_1, S_2, \ldots, S_N$, to convince oneself of the net effect at the time sequence of execution of these statements

$$S_1 ; S_2 ; \ldots ; S_N$$

may be called an appeal to enumerative reasonong. This requires N steps of reasoning. Another case of enumerative reasoning is the conditional statement:

$$\underline{\text{if}}\ B\ \underline{\text{then}}\ S_1\ \underline{\text{else}}\ S_2$$

which requires two steps of reasoning.

3.2 Mathematical induction

We use this tool to tackle the problem of programs which include loops. An alternative to the direct application of mathematical induction to programs is to appeal to one of the following two theorems, which are stated without proof.

Linear Search Theorem:

Given i) $d_0 = D$

ii) $d_i = F(d_{i-1})$, $0 < i \leq k$

iii) $\underline{\text{non}}\ \text{prop}(d_i)$, $0 \leq i < k$

iv) $\text{prop}(d_k)$

consider the following piece of program:

d:= D;

$\underline{\text{while}}\ \underline{\text{non}}$ prop (d) $\underline{\text{do}}$ d:= F(d)

This program terminates with d equal to d_k.

[A proof is given in Dijkstra (1970). Professor Dijkstra remarked that he had been appalled by the length of this proof.]

Let $P\{S\}Q$

denote the following relationship between a program statement S and assertions P and Q: 'If P is true before initiation of S then Q will be true on its completion'. [This notation is introduced in Hoare (1969)].

Invariance theorem:

If $\vdash P \wedge B\{S\}P$ then $\vdash P\{\underline{\text{while}}\ B\ \underline{\text{do}}\ S\} \neg B \wedge P$ provided the loop terminates.

These two theorems themselves are provided by mathematical induction and once proved we can appeal to these theorems without further direct application of mathematical induction.

3.3 Abstraction

The main purpose of abstraction is to reduce the appeal we have to make to enumerative reasoning. For example, a variable is an abstraction

from its value. In addition we may appeal to either operational or representational abstraction. The number of steps of reasoning in the following example is reduced by operational abstraction. Consider the program:

$$\begin{array}{l} \underline{\text{if}}\ B_1\ \underline{\text{then}}\ S_{11}\ \underline{\text{else}}\ S_{12}\ ; \\ \underline{\text{if}}\ B_2\ \underline{\text{then}}\ S_{21}\ \underline{\text{else}}\ S_{22}\ ; \\ \quad\vdots\qquad\quad\vdots\qquad\quad\vdots \\ \underline{\text{if}}\ B_N\ \underline{\text{then}}\ S_{N1}\ \underline{\text{else}}\ S_{N2}\ ; \end{array}$$

It requires 2N steps of reasoning to reduce this program to the form:

$$S_1\ ;\ \ldots\ ;\ S_N$$

where

$$S_i \equiv \underline{\text{if}}\ B_i\ \underline{\text{then}}\ S_{i1}\ \underline{\text{else}}\ S_{i2}$$

In addition we require a further N steps to understand this. This gives a total of 3N steps which is linear in N (a measure of program length). Alternatively, if we do not introduce the abstract statements S_i then we must consider 2^N possible paths through the program, each path containing N statements. This gives a total of $N \times 2^N$ steps of reasoning which is incomparably large.

A simple example of operational abstraction is given by the replacement of the statement:

$$\underline{\text{if}}\ x < 0\ \underline{\text{then}}\ x := -x$$

by the (abstract) statement

$$x := \text{abs}(x)\ .$$

The object is to find, as quickly as possible, a description applicable to a conditional clause in which the fact that it was controlled by a conditional clause has disappeared. A similar consideration applies to a piece of program controlled by a <u>while</u> clause. All these tools are applied in the following example.

<u>Example 1</u>. For <u>integer</u> A, B, x, y and z, where

$$A \geq 1,\ B \geq 0$$

prove that, after execution of the following program section, we shall have $z = A^B$.

```
x:= A; y:= B; z:= 1;
while y ≠ 0 do
begin if odd(y) then begin y:= y-1; z:= z*x end;
      y:= y/2; x:= x↑2
```

In order to apply the invariance theorem we need to set up an assertion P which remains invariant (and to prove termination). One of the elements of P will be that y remains integer valued and not less than zero:

integer $y \geq 0$

This is certainly true to start with and the statement qualified by the while clause is executed only if $y \neq 0$, and so y will be positive. The abstract effect of:

if odd(y) then begin y:= y-1; z:= z*x end

on y, is to 'make the value of y even without increasing it'. Then since $y > 0$, after this statement is executed we have $y \geq 0$, and y is still integer valued and even. Executing y:= y/2 therefore retains the invariance of

integer $y \geq 0$.

Termination follows since the successive operations, 'make y even without increasing it' and 'halve y' have the net effect of decreasing the value of y, keeping it integer valued and since $y \geq 0$, y must eventually reach zero.

The other element of the invariance relation is:

$A^B = z*x^y$ and integer $x > 0$

Again, this is true to start with. Whenever y is decreased by 1, z is multiplied by x to compensate and whenever y is halved x is squared. Since x is originally integer and greater than zero then it remains greater than zero. P asserts that:

integer $y \geq 0$ and $A^B = z*x^y$ and integer $x > 0$.

The statements

if odd(y) do begin y:= y-1; z:= z*x end

and, y:= y/2; x:= x*x

each leave P invariant, and hence their succession leaves P invariant. The Invariance Theorem allows us to state, therefore, that the whole while clause leaves P invariant. Also, we have that non $y \neq 0$, and combining these results we have:

$$z = A^B$$

If we replaced the condition $y \neq 0$ by $y > 0$ then this does not affect the program, but the conclusion that $y = 0$ does not immediately follow. The invariant assertion P contains the element $y \geq 0$ which combines with non $y > 0$ to give $y = 0$. It is interesting that a seemingly trivial change requires different reasoning.

The total effect of our analysis of example 1 is to replace the whole repetition clause by an abstract statement which no longer reflects the repetition. This statement says 'make $y = 0$ and $z = A^B$ and leave $x > 0$' and the fact that it is a loop is no longer relevant.

Example 2.(sketch only) If $0 < b < 1$ then the program:

```
a:= b; c:= 1-b;
while c > eps do
begin a:= a x (1+0.5 x c);
      c:= c ↑ 2*(.25*c + .75)
end
```

will approximate $\sqrt{b}$ ($a = \sqrt{b}$) if eps is sufficiently small.

One of the simplest ways of showing this is to establish the invariance of the relation:

$$a^2 = b(1-c)$$

This combines with $c \leq eps$ to establish the required approximation. This relation also established termination.

In reasoning about nicely structured programs the next step is to realise that we should regard programs as designs of large classes of computations. A program is never a goal in itself. The real subject matter of the programming activity is the possible computations that may be evoked by one's programs. As soon as we realise that all assertions about programs are always in terms of the computations they can evoke, while the program text is the last tangible thing we can lay our hands on, then we realise that it is worthwhile to shorten the conceptual gap between a static program text and these computations as they evolve. One of the ways to do this is never to use goto statements but to have our sequencing controlled by more orderly things such as conditional and repetition clauses.

Given a certain task to be programmed we can construct alternative programs for it and the question is to what extent can they be mapped upon each other, to what extent are they further refinements of the same concept. This means that a program is not regarded as an isolated object to be composed once, all by itself; it means that we consider our program as a member of a whole class of similar programs and when we are in the process of constructing the program, say half way, the class of all

programs of which our final program will be a member is the class of all forms our program still can take. Consequently, we should pay much more attention to the sequencing of a program than was previously the case. As an example of this consider the following problem.

Example 3. (due to Niklaus Wirth)

Construct a program generating non-empty sequences of 0's, 1's and 2's without non-empty, element-wise, equal, adjoining sub-sequences, generating these sequences in alphabetical order until a sequence of length 100 has been generated. (Such a sequence is known to exist).

To do this we generate all sequences of 0's, 1's and 2's in alphabetical order and test for valid sequences. If it is valid the sequence is printed and extended by adding a 0 at the right; if the sequence is invalid we 'increase the final digit by one' after 'removing terminal sequence of 2's'. This generates solutions in the order:

```
      0
*     00
      01
      010
*     0100
*     0101
      0102
      01020
*     010200
      010201
      0102010
*     01020100
*     01020101
*     01020102
*     0102011
      0102012
```

where * indicates an invalid sub-sequence.

The basic operations on a sequence are INCREASE and EXTEND. We might consider, as a first attempt at a program, the following:

```
INITIALISE SEQUENCE TO SINGLE ZERO;
repeat if GOOD then {PRINT; EXTEND}
            else INCREASE
until length = 101
```

The principal objection is the tortuous reasoning required to establish the correctness of the stopping criterion. A solution to which this objection does not apply is:

```
INITIALISE SEQUENCE EMPTY;
repeat EXTEND;
      while non GOOD do INCREASE;
      PRINT
until length = 100
```

The reason that this is a much more beautiful program is because the sequence of operations:

```
      EXTEND;
      while non GOOD do INCREASE;
```

is simply the operator:

```
      TRANSFORM SEQUENCE INTO NEXT SOLUTION
```

The first program forces us to understand the program in terms of all trial sequences, whereas the second program allows us, at a certain level of detail, to understand it in terms of solutions only. It is a matter of later concern that each solution must be understood in terms of a series of trial sequences. This is precisely what is meant by operational abstraction. If there is an alternative way of implementing the operator TRANSFORM SEQUENCE TO NEXT SOLUTION, this doesn't matter because at this level they are the same thing. This is also an example of a family of related programs.

There seems no way of convincing ourselves that these two programs are output equivalent. We can do it in this specific instance but we have no systematic way of doing it, so it is preferable to regard two such programs as incomparable. There is not a certain degree of abstraction where we can distinguish the same pattern of sequencing except for the fact that they both solve the whole problem. This incomparability in turn makes one more conscious of the sequencing rules one chooses and this is closely related to the fact that one considers a program not so much as a product in itself but as a design of a set of computations.

The two programs are output equivalent. However, the interest does not lie in establishing the equivalence of two given programs. This is the irrelevant aspect. We only have the problem, we must 'make' programs to solve it. If we have two programs then it is more valuable to see them as possible alternative children or grandchildren of the same conception. We should think of programming as the judicious postponement of commitments. The reason for giving these two examples is not because they are equivalent but because the second program is nicer in that we consider it in terms of solutions only.

The program to solve the problem of example 3 is not yet complete to the level of detail prescribed for any programming language. We still have many decisions to make, for instance how to represent sequences and how to do the operations EXTEND and GOOD. The representational abstraction of the variable SEQUENCE allows us to consider the control sequencing first. We would consider such a program nicely structured because we have not had to go down to too detailed a level.

4. On trading storage space for computation speed

A standard technique of constructing different programs out of the same abstract program is a mechanism for trading storage space for computation time. We have two versions A and B of a program:

```
        A                              B
arg:= ...                      arg:= ...   ; fun:= FUN(arg);
... FUN(arg)...                ... fun ...
```

In version A, 'FUN(arg)' is evaluated every time it is required. In version B an extra variable 'fun' is used, which is updated whenever 'arg' is changed, to maintain the relation:

$$fun = FUN(arg) \ .$$

Another reason why this transformation may be attractive is when updating 'Fun' does not require evaluating 'FUN(arg)', but some simpler expression. We should bear this in mind when looking for data representations for 'nicely' structured programs.

5. The problem of the eight queens

We want to teach students how to solve problems, not to give ready made solutions, to teach them not thoughts but how to think. We shall consider the steps of reasoning required to construct a solution to a well-known

problem. The solution should lead the student to the discovery of the method of backtracking.

Example 4. Generate all configurations of eight queens on an 8*8 chess board such that no queen can take any of the others. A queen may take any other on the same row, column or diagonal.

The answer is a set A of configurations of the chess board. A standard technique is to look for a set B of which A is a true subset. We can generate all elements of B in some order and select the elements which are members of A. For this approach to work there are three conditions which must be satisfied.

1. Set B should not be too large.
2. The criterion for exclusion from A should be cheaper the larger B is.
3. It should be possible to generate elements of B and easier than generating elements of A.

The second condition is interesting in that we are not too worried about the cost of not excluding those members which are in A, because in general A is much smaller than B. If generation of the elements of set B presents difficulty then this can in turn be embedded in a still larger set C and the same approach applied again.

Notice also that we need to know a bound for the number of members of B.

The assumption is that we can invent a suitable set B. One way of finding candidates for set B is to make a list of the independent criteria that are satisfied by elements of set A and just remove one of them. In the case of the queen's problem we have:

1. There are 8 queens on the board.
2. No two of any N queens can take each other.

Omitting either of these gives for set B the alternatives:

B1: All configurations of N queens so that no two can take each other.

B2: All configurations of 8 queens.

Under the circumstances this is too crude, since both sets are far too large and this shows that the approach may not always work.

The undergraduate must be able to see that B2 is huge. He can also conclude that B1 is large by considering the case of N = 2.

Since we are still looking for a suitable set B let us list all obvious properties of set A. You do so in the hope that you might find a clue, but you should not spend much time proving a complex property when you are not sure if you can use it. In our case obvious properties include: no two queens may occupy the same row, column, upward diagonal or downward diagonal. Thus, we have at most one queen in each row, column or diagonal and exactly eight queens on the board. Hence, we have exactly one queen in each row and each column; eight of the 15 upward diagonals will contain one queen and similarly for the downward diagonals. Another useful property is obtained by considering set B1. Removing a single queen from any non-empty configuration from set B1 yields a configuration which is also in B1. Conversely, one can conclude that each non-empty configuration of B1 can be obtained by extending some other configuration from B1 by one queen. This extension property does not hold for B2. Such an extension property is useful when generating sets.

Turning now to consider the order in which we shall generate elements of set A this might give us a clue to the mysterious set B. How do we characterise an element of set A? An obvious way to represent a configuration, which follows from the property that each row contains one queen, is as an array x[0:7] such that

x[i] = number of column occupied by queen on i^{th} row.

The only reasonable order is an alphabetic one, which in turn suggests a set B:

B = set of all configurations of N queens occupying the first N rows, such that no two can take each other.

As a consequence, we open the way to algorithms in which rows and columns are treated differently. At first sight this is surprising, because the original problem is completely symmetrical in rows and columns.

We want the elements of set A in alphabetic order and so the best way is to generate the elements of set B in alphabetic order also. First, we have to generate all solutions with x[0] = 0, then all with x[0] = 1 etc.; of the solutions with x[0] = 0 we generate first those with x[1] = 0 (if any), then those with x[1] =1(if any), etc. One way to program this is to

use the abstract operator GENERATE NEXT ELEMENT OF B:

```
INITIALIZE EMPTY BOARD;
repeat GENERATE NEXT ELEMENT OF B;
       if BOARD FULL then PRINT
until  B EXHAUSTED
```

However, GENERATE NEXT ELEMENT OF B is not an attractive operator to construct. This is indicated by the fact that the operator will sometimes increase, sometimes decrease and sometimes leave unchanged the number of queens in a configuration. Additionally, it is not easy to see how to test for B EXHAUSTED.

If we insist on thinking of the elements of set B as a single sequence the only corresponding program structure is a single loop as above. If we are not attracted by this then we may order the elements of B as a sequence of subsequences and one way we can do this is to group the elements of B by position of queen zero:

```
h:= 0;
repeat  SET QUEEN 0 ON COLUMN h;
        GENERATE ALL CONFIGURATIONS WITH QUEEN 0 FIXED;
        REMOVE QUEEN 0;
        h:= h + 1
until h = 8
```

Considering how we might make the operator GENERATE ALL CONFIGURATIONS WITH QUEEN 0 FIXED we can repeat this program structure again for QUEEN 1. This operator is then:

```
h1:= 0;
repeat if COLUMN h1 FREE do
       begin SET QUEEN 1 ON COLUMN h1;
             GENERATE ALL CONFIGURATIONS WITH QUEENS 0 AND 1 FIXED;
             REMOVE QUEEN 1
       end;
       h1:= h1 + 1;
until h1 = 8
```

Again the operator GENERATES ALL CONFIGURATIONS WITH QUEENS 0 AND 1 FIXED can be made by a similar piece of program. In the complete program the outermost loop will differ only trivially from the general structure. The innermost is exceptional in the sense that it is made by just using the operator PRINT. This nest of similar loops suggests a recursive procedure:

```
procedure generate;
begin integer h;    h:= 0;
      repeat if COLUMN h FREE do
             begin SET QUEEN ON COLUMN h;
                   if BOARD FULL then PRINT
                                 else generate;
             end;  REMOVE QUEEN
             h:= h + 1
      until h = 8
end
```

With the aid of this procedure the main program is constructed as:

```
INITIALISE EMPTY BOARD;
generate
```

As far as the queen's problem is concerned what remains is an analysis of how to represent the configurations on the board so that our remaining operators can be easily constructed. So far we have only used the simple structure x[0:7] which is not easy to test. The analysis of the section on trading storage space for computation time suggests that we might look for suitable additional tabulative material to simplify our operators. It turns out that one of the simplest ways is to keep a record of the occupancy of columns and diagonals. We shall not go into this here.

6. Conclusion

One starts with a problem and deals with it minute step after minute step. One makes a list of all considerations and information available. If you lose patience or get tired and are forced to hurry then you will do a lousy job. If you take more time and pay more attention to the effort of describing how you find solutions then this leads to the discovery of new solutions itself. This suggests that this sort of methodology works.

7. Discussion

Professor Wirth pointed out that the invariance theorem had been considered by Hoare as an axiom rather than a theorem and he himself would prefer to call the invariance theorem the definition of a while clause. Then the linear search theorem could be proved using mathematical induction. Professor Knuth interrupted saying he did not see how this could be done, to

which Professor Dijkstra replied that he thought one had to use an explicit count for the number of assignments made. Professor Michaelson asked for a definition of a while clause which Professor Dijkstra gave as

while B do S ≡ if B do {S; while B do S}

Professor Michaelson then said that he did not think that the Invariance Theorem alone could be used to prove the Linear Search Theorem as there was nothing to specify an order of execution.

Some discussion centred around the problem of efficiency in terms of computation time with regard to the sets A and B in the queens problem. Professor Dijkstra ended this by saying that he was prepared to take into account the size of the sets, but not the relative speeds of primitive operations.

Finally, some questions were asked concerning the complexity of the queens problem and what benefits might be expected from the approach that had been given.

Professor Wirth felt that the problem was too large to present at any early stage of programming. It belonged to a course on advanced programming techniques although the ideas on how to represent the information ought to be taught earlier.

Professor Perlis was concerned that students might feel disturbed at having to write so much non-performable code. He also wondered whether one could lay out these algorithms independently of the programming language in which they would ultimately be written. Professor Dijkstra thought that, to a considerable extent, lectures in programming could be given which were language independent.

Professor Michaelson said that he believed that the object of the exercise was to teach students to write programs in a comprehensible way, programs in which they had solved the given problem, in a way that enabled them to modify their programs as their understanding of the problem changed. Students should be convinced that it was worth putting some effort into this. He thought that Professor Dijkstra had given a rather inelegant solution to a rather pointless problem and wondered why this problem had been chosen in preference to a more practical one which might have more attraction for the students.

In reply Professor Dijkstra said that the problem was a difficult one with the charm that it was easy to formulate. Another point was that one had a clear visual terminology in terms of which to describe the states. Professor Perlis put what he thought was a stronger case in that the problem had a great deal of carry over, teaching the students about enumeration, about generating sequences and about making sure that all solutions were found.

8. References

Dijkstra, E. W. (1970) EWD249 'Notes on Structured Programming'. Tech. University Eindhoven Report 70-WSK03 [This report is available on request from the author].

Hoare, C. R. (1969) 'An Axiomatic Approach to Computer Programming'. CACM, Vol.12, No. 10 (October 1969).