Program Development by Stepwise Refinement

Niklaus Wirth
Eidgenössische Technische Hochschule
Zürich, Switzerland

The creative activity of programming—to be distinguished from coding—is usually taught by examples serving to exhibit certain techniques. It is here considered as a sequence of design decisions concerning the decomposition of tasks into subtasks and of data into data structures. The process of successive refinement of specifications is illustrated by a short but nontrivial example, from which a number of conclusions are drawn regarding the art and the instruction of programming.

Key Words and Phrases: education in programming, programming techniques, stepwise program construction

CR Categories: 1.50, 4.0

1. Introduction

Programming is usually taught by examples. Experience shows that the success of a programming course critically depends on the choice of these examples. Unfortunately, they are too often selected with the prime intent to demonstrate what a computer can do. Instead, a main criterion for selection should be their suitability to exhibit certain widely applicable techniques. Furthermore, examples of programs are commonly presented as finished “products” followed by explanations of their purpose and their linguistic details. But active programming consists of the design of new programs, rather than contemplation of old programs. As a consequence of these teaching methods, the student obtains the impression that programming consists mainly of mastering a language (with all the peculiarities and intricacies so abundant in modern PL’s) and relying on one’s intuition to somehow transform ideas into finished programs. Clearly, programming courses should teach methods of design and construction, and the selected examples should be such that a gradual development can be nicely demonstrated.

This paper deals with a single example chosen with these two purposes in mind. Some well-known techniques are briefly demonstrated and motivated (strategy of preselection, stepwise construction of trial solutions, introduction of auxiliary data, recursion), and the program is gradually developed in a sequence of refinement steps.

In each step, one or several instructions of the given program are decomposed into more detailed instructions. This successive decomposition or refinement of specifications terminates when all instructions are expressed in terms of an underlying computer or programming language, and must therefore be guided by the facilities available on that computer or language. The result of the execution of a program is expressed in terms of data, and it may be necessary to introduce further data for communication between the obtained subtasks or instructions. As tasks are refined, so the data may have to be refined, decomposed, or structured, and it is natural to refine program and data specifications in parallel.

Every refinement step implies some design decisions. It is important that these decision be made explicit, and that the programmer be aware of the underlying criteria and of the existence of alternative solutions. The possible solutions to a given problem emerge as the leaves of a tree, each node representing a point of deliberation and decision. Subtrees may be considered as families of solutions with certain common characteristics and structures. The notion of such a tree may be particularly helpful in the situation of changing purpose and environment to which a program may sometime have to be adapted.

A guideline in the process of stepwise refinement should be the principle to decompose decisions as much as possible, to untangle aspects which are only seemingly interdependent, and to defer those decisions which concern details of representation as long as possible. This
will result in programs which are easier to adapt to differ-
ent environments (languages and computers), where differ-
ent representations may be required.

The chosen sample problem is formulated at the be-
ginning of section 3. The reader is strongly urged to try
to find a solution by himself before embarking on the
paper which—of course—presents only one of many
possible solutions.

2. Notation

For the description of programs, a slightly augmen-
ted Algol 60 notation will be used. In order to express
repetition of statements in a more lucid way than by use
of labels and jumps, a statement of the form

\[
\text{repeat (statement sequence)}
\text{until (Boolean expression)}
\]

is introduced, meaning that the statement sequence is to
be repeated until the Boolean expression has obtained
the value true.

3. The 8-Queens Problem and an Approach to Its
Solution

Given are an 8 \times 8 chessboard and 8 queens which are hostile
each other. Find a position for each queen in a configuration such
that no queen may be taken by any other queen (i.e. such that every
row, column, and diagonal contains at most one queen).

This problem is characteristic for the rather frequent
situation where an analytical solution is not known, and
where one has to resort to the method of trial and error. Typically, there exists a set A of candidates for
solutions, among which one is to be selected which
satisfies a certain condition p. Thus a solution is char-
acterized as an x such that (x \in A) \land p(x).

A straightforward program to find a solution is:

\[
\text{repeat Generate the next element of } A \text{ and call it } x
\text{ until (p(x) \lor (no more elements in A))}
\text{ if p(x) then } x = \text{ solution}
\]

The difficulty with this sort of problem usually is the
sheer size of A, which forbids an exhaustive generation of
candidates on the grounds of efficiency consider-
ations. In the present example, A consists of

\[
64! / (56! \times 8!) = 2^{32}
\]

elements (board configurations). Under the assumption that generation and
test of each configuration consumes 100 \mu s, it
would roughly take 7 hours to find a solution. It is
obviously necessary to invent a “shortcut,” a method
which eliminates a large number of “obviously dis-
qualified contenders. This strategy of preselection is char-
acterized as follows: Find a representation of p in the form p = q \land r. Then let B_r =

\[
\{x | (x \in A) \land r(x)\}
\]

Obviously, B_r \subseteq A. Instead of generating elements of A only elements of B are pro-
duced and tested on condition q instead of p. Suitable

candidates for a condition r are those which satisfy the
following requirements:

1. B, is much smaller than A.
2. Elements of B, are easily generated.
3. Condition q is easier to test than condition p.

The corresponding program then is:

\[
\text{repeat Generate the next element of } B \text{ and call it } x
\text{ until (q(x) \lor (no more elements in B))}
\text{ if q(x) then } x = \text{ solution}
\]

A suitable condition r in the 8-queens problem is the
rule that in every column of the board there must be
exactly one queen. Condition q then merely specifies
that there be at most one queen in every row and in
every diagonal, which is evidently somewhat easier
to test than p. The set B_r (configurations with one queen
in every column) contains “only” 8\times 2^{32} elements.
They are generated by restricting the movement of
queens to columns. Thus all of the above conditions are
satisfied.

Assuming again a time of 100 \mu s for the generation
and test of a potential solution, finding a solution
would now consume only 100 seconds. Having a pow-
ful computer at one’s disposal, one might easily be
content with this gain in performance. If one is less
fortunate and is forced to, say, solve the problem by
hand, it would take 280 hours generating and testing
configurations at the rate of one per second. In this
case it might pay to spend some time finding further
shortcuts. Instead of applying the same method as
before, another one is advocated here which is char-
acterized as follows: Find a representation of trial
solutions x of the form [x_1, x_2, \ldots, x_n], such that
every trial solution can be generated in steps which
produce [x_1], [x_1, x_2], \ldots, [x_1, x_2, \ldots, x_n]
respectively. The decomposition must be such that:

1. Every step (generating x_j) must be considerably
   simpler to compute than the entire candidate x.
2. q(x) \supseteq q(x_1 \ldots x_j) for all j \leq n.

Thus a full solution can never be obtained by extend-
ing a partial trial solution which does not satisfy
the predicate q. On the other hand, however, a partial
trial solution satisfying q may not be extensible into a
complete solution. This method of stepwise construc-
tion of trial solutions therefore requires that trial solu-
tions failing at step j may have to be “shortened”
again in order to try different extensions. This technique
is called backtracking and may generally be character-
ized by the program:

\[
\begin{align*}
j & := 1; \\
\text{repeat try step } j; \\
\text{if successful then } & \text{ advance else regress} \\
\text{until } (j < 1) \lor (j > n)
\end{align*}
\]

In the 8-queens example, a solution can be con-
structed by positioning queens in successive columns
starting with column 1 and adding a queen in the next
column in each step. Obviously, a partial configuration
not satisfying the mutual nonaggression condition may

---

1 This problem was investigated by C. F. Gauss in 1850.
never be extended by this method into a full solution. Also, since during the $j$th step only $j$ queens have to be considered and tested for mutual nonaggression, finding a partial solution at step $j$ requires less effort of inspection than finding a complete solution under the condition that all 8 queens are on the board all the time. Both stated criteria are therefore satisfied by the decomposition in which step $j$ consists of finding a safe position for the queen in the $j$th column.

The program subsequently to be developed is based on this method; it generates and tests 876 partial configurations before finding a complete solution. Assuming again that each generation and test (which is now more easily accomplished than before) consumes one second, the solution is found in 15 minutes, and with the computer taking 100 $\mu$s per step, in 0.09 seconds.

4. Development of the Program

We now formulate the stepwise generation of partial solutions to the 8-queens problem by the following first version of a program:

```
begin reconsiderpriorcolumn;
  if $\neg$ regressoustaffirstcol then
    removequeen
end
end
```

The program is expressed in terms of the instructions:

```
considerfirstcolumn
considernextcolumn
considerpriorcolumn
advancepointer
testsquare (sets the variable safe)
setqueen
removequeen
```

and of the predicates:

```
lastsquare
lastcoldone
regressoustaffirstcol
```

In order to refine these instructions and predicates further in the direction of instructions and predicates available in common programming languages, it becomes necessary to express them in terms of data representable in those languages. A decision on how to represent the relevant facts in terms of data can therefore no longer be postponed. First priority in decision making is given to the problem of how to represent the positions of the queens and of the square being currently inspected.

The most straightforward solution (i.e., the one most closely reflecting a wooden chessboard occupied by marble pieces) is to introduce a Boolean square matrix with $B(i,j) = true$ denoting that square $(i,j)$ is occupied. The success of an algorithm, however, depends almost always on a suitable choice of its data representation in the light of the ease in which this representation allows the necessary operations to be expressed. Apart from this, consideration regarding storage requirements may be of prime importance (although hardly in this case). A common difficulty in program design lies in the unfortunate fact that at the stage where decisions about data representations have to be made, it often is still difficult to foresee the details of the necessary instructions operating on the data, and often quite impossible to estimate the advantages of one possible representation over another. In general, it is therefore advisable to delay decisions about data representation as long as possible (but not until it becomes obvious that no realizable solution will suit the chosen algorithm).

In the program presented here, it is fairly evident even at this stage that the following choice is more suitable than a Boolean matrix in terms of simplicity of later instructions as well as of storage economy. $j$ is the index of the currently inspected column; $(x_j,j)$ is the coordinate of the last inspected square; and the position of the queen in column $k < j$ is given by the coordinate pair $(x_k,k)$ of the board. Now the variable declarations for pointer and board are refined into:

```
integer $j$ (0 $\leq$ $j$ $\leq$ 9)
integer array $x[1:8]$ (0 $\leq$ $x_i$ $\leq$ 8)
```

Communications of the ACM April 1971
Volume 14 Number 4
and the further refinements of some of the above instructions and predicates are expressed as:

procedure considerfirstcolumn; begin 
  \( f := 1 \); \( x[1] := 0 \) end
procedure considernextcolumn; begin 
  \( f := f + 1 \); \( x[f] := 0 \) end
procedure reconsiderpriorcolumn; \( f := f - 1 \)
procedure advancepointer; \( x[f] := x[f] + 1 \)
Boolean procedure lastsquare; \( \text{lastsquare} := x[f] = n \)
Boolean procedure lastcolone; \( \text{lastcolone} := f = n \)
Boolean procedure infirstcol; \( \text{infirstcol} := f < 1 \)

At this stage, the program is expressed in terms of the instructions:

\text{testsquare}
\text{setqueen}
\text{removqueen}

As a matter of fact, the instructions \text{setqueen} and \text{removqueen} may be regarded as vacuous, if we decide that the procedure \text{testsquare} is to determine the value of the variable \text{safe} solely on the grounds of the values \( x_1 \cdots x_{n-1} \) which completely represent the positions of the \( n-1 \) queens so far on the board. But unfortunately the instruction \text{testsquare} is the one most frequently executed, and it is therefore the one instruction where considerations of efficiency are not only justified but essential for a good solution of the problem. Evidently a version of \text{testsquare} expressed only in terms of \( x_1 \cdots x_{n-1} \) is inefficient at best. It should be obvious that \text{testsquare} is executed far more often than \text{setqueen} and \text{removqueen}. The latter procedures are executed whenever the column \( f \) is changed (say \( m \) times), the former whenever a move to the next square is undertaken (i.e. \( x_1 \) is changed, say \( n \) times). However, \text{setqueen} and \text{removqueen} are the only procedures which affect the chessboard. Efficiency may therefore be gained by the method of \textit{introducing auxiliary variables} \( V(x_1 \cdots x_f) \) such that:

1. Whether a square is safe can be computed more easily from \( V(x) \) than from \( x \) directly (say in \( u \) units of computation instead of \( ku \) units of computation).
2. The computation of \( V(x) \) from \( x \) (whenever \( x \) changes) is not too complicated (say of \( v \) units of computation).

The introduction of \( V \) is advantageous (apart from considerations of storage economy), if

\[
\frac{n(k-1)u}{m} > mu \quad \text{or} \quad \frac{n(k-1)}{m} > \frac{v}{u},
\]

i.e. if the gain is greater than the loss in computation units.

A most straightforward solution to obtain a simple version of \text{testsquare} is to introduce a Boolean matrix \( B \) such that \( B[i,j] = \text{true} \) signifies that square \((i,j)\) is not taken by another queen. But unfortunately, its recomputation whenever a new queen is removed (\( v \)) is prohibitive (why?) and will more than outweigh the gain.

The realization that the relevant condition for safety of a square is that the square must lie neither in a row nor in a diagonal already occupied by another queen, leads to a much more economic choice of \( V \). We introduce Boolean arrays \( a, b, c \) with the meanings:

\( a_i = \text{true} \) : no queen is positioned in row \( i \)
\( b_i = \text{true} \) : no queen is positioned in the /-diagonal \( k \)
\( c_i = \text{true} \) : no queen is positioned in the \(-/\)-diagonal \( k \)

The choice of the index ranges of these arrays is made in view of the fact that squares with equal sum of their coordinates lie on the same /-/diagonal, and those with equal difference lie on the same \(-/-\)-diagonal. With row and column indices from 1 to \( n \), we obtain:

\text{Boolean array} \( a[1:n], b[2:n], c[1:n] \)

Upon every introduction of auxiliary data, care has to be taken of their correct initialization. Since our algorithm starts with an empty chessboard, this fact must be represented by initially assigning the value \text{true} to all components of the arrays \( a, b, c \). We can now write:

procedure \text{testsquare};
  \text{safe} := \text{false};
  \text{safe} := a[x[1]] \wedge b[x[1]] \wedge c[x[1]]
procedure \text{setqueen};
  a[x[1]] := \text{true}; b[x[1]] := \text{false}; c[x[1]] := \text{false}
procedure \text{removqueen};
  a[x[1]] := \text{false}; b[x[1]] := \text{true}; c[x[1]] := \text{true}

The correctness of the latter procedure is based on the fact that each queen currently on the board had been positioned on a safe square, and that all queens positioned after the one to be removed now had already been removed. Thus the square to be vacated becomes safe again.

A critical examination of the program obtained so far reveals that the variable \( x[f] \) occurs very often, and
In particular at those places of the program which are also executed most often. Moreover, examination of \( x[j] \) occurs much more frequently than reassignment of values to \( j \). As a consequence, the principle of introduction of auxiliary data can again be applied to increase efficiency: a new variable

integer \( l \)

is used to represent the value so far denoted by \( x[j] \). Consequently \( x[j] := i \) must always be executed before \( j \) is increased, and \( i := x[j] \) after \( j \) is decreased. This final step of program development leads to the reformation of some of the above procedures as follows:

\[
\begin{align*}
\text{procedure} & \quad \text{testsquare} ; \\
& \quad \text{safe} := c[l] \land b[i+j] \land c[l-j] \\
& \quad \text{procedure} \quad \text{setqueen} ; \\
& \quad a[i] := b[i+j] := c[l-j] := \text{false} \\
& \quad \text{procedure} \quad \text{removequeen} ; \\
& \quad a[i] := b[i+j] := c[l-j] := \text{true} \\
& \quad \text{procedure} \quad \text{considerfirstcolumn} ; \\
& \quad \text{begin} \quad j := 1 ; \quad i := 0 \text{ end} \\
& \quad \text{procedure} \quad \text{advancepointer} ; \quad i := i+1 \\
& \quad \text{procedure} \quad \text{considernextcolumn} ; \\
& \quad \text{begin} \quad x[j] := i ; \quad j := j+1 ; \quad i := 0 \text{ end} \\
& \quad \text{Boolean procedure} \quad \text{lastsquare} ; \\
& \quad \text{lastsquare} := i = 8
\end{align*}
\]

The final program, using the procedures

\[
\begin{align*}
\text{testsquare} \\
\text{setqueen} \\
\text{regress} \\
\text{removequeen}
\end{align*}
\]

and with the other procedures directly substituted, now has the form

\[
\begin{align*}
& \quad j := 1 ; \quad i := 0 ; \\
& \quad \text{repeat} \\
& \quad \quad \text{repeat} \quad i := i+1 ; \quad \text{testsquare} \\
& \quad \quad \quad \text{until} \quad \text{safe} \lor (i=8) ; \\
& \quad \quad \quad \text{if} \quad \text{safe} \quad \text{then} \\
& \quad \quad \quad \quad \text{begin} \quad \text{setqueen} ; \quad x[j] := i ; \quad j := j+1 ; \quad i := 0 \\
& \quad \quad \quad \quad \text{end} \quad \text{else} \quad \text{regress} \\
& \quad \quad \text{until} \quad (j \geq 8) \lor (j < 1) ; \\
& \quad \quad \text{if} \quad j > 8 \quad \text{then} \quad \text{PRINT}() \quad \text{else} \quad \text{FAILURE}
\end{align*}
\]

It is noteworthy that this program still displays the structure of the version designed in the first step. Naturally other, equally valid solutions can be suggested and be developed by the same method of stepwise program refinement. It is particularly essential to demonstrate this fact to students. One alternative solution was suggested to the author by E. W. Dijkstra. It is based on the view that the problem consists of a stepwise extension of the board by one column containing a safely positioned queen, starting with a null-board and terminating with 8 columns. The process of extending the board is formulated as a procedure, and the natural method to obtain a complete board is by recursion of this procedure. It can easily be composed of the same set of more primitive instructions which were used in the first solution.

\[
\begin{align*}
\text{procedure} \quad \text{Trycolumn}(j) ; \\
& \quad \text{begin} \quad i := 0 ; \\
& \quad \quad \text{repeat} \quad i := i+1 ; \quad \text{testsquare} ; \\
& \quad \quad \quad \text{if} \quad \text{safe} \quad \text{then} \\
& \quad \quad \quad \quad \text{begin} \quad \text{setqueen} ; \quad x[j] := i ; \\
& \quad \quad \quad \quad \quad \text{if} \quad j < 8 \quad \text{then} \quad \text{Trycolumn}(j+1) ; \\
& \quad \quad \quad \quad \quad \text{if} \quad \text{safe} \quad \text{then} \quad \text{removequeen} \\
& \quad \quad \quad \text{end} \\
& \quad \quad \text{until} \quad \text{safe} \lor (i=8) \\
& \quad \text{end}
\end{align*}
\]

The program using this procedure then is

Trycolumn(1) ;

if safe then PRINT(x) else FAILURE

(Note that due to the introduction of the variable \( i \) local to the recursive procedure, every column has its own pointer of inspection \( i \). As a consequence, the procedures

\[
\begin{align*}
\text{testsquare} \\
\text{setqueen} \\
\text{removequeen}
\end{align*}
\]

must be declared locally within Trycolumn too, because they refer to the \( i \) designating the scanned square in the current column.)

5. The Generalized 8-Queens Problem

In the practical world of computing, it is rather uncommon that a program, once it performs correctly and satisfactorily, remains unchanged forever. Usually its users discover sooner or later that their program does not deliver all the desired results, or worse, that the results requested were not the ones really needed. Then either an extension or a change of the program is called for, and it is in this case where the method of stepwise program design and systematic structuring is most valuable and advantageous. If the structure and the program components were well chosen, then often many of the constituent instructions can be adopted unchanged. Thereby the effort of redesign and reverification may be drastically reduced. As a matter of fact, the adaptability of a program to changes in its objectives (often called maintainability) and to changes in its environment

Communications of the ACM April 1971 Volume 14 Number 4
(nowadays called portability) can be measured primarily in terms of the degree to which it is neatly structured.

It is the purpose of the subsequent section to demonstrate this advantage in view of a generalization of the original 8-queens problem and its solution through an extension of the program components introduced before.

The generalized problem is formulated as follows:

Find all possible configurations of 8 hostile queens on an 8 × 8 chessboard, such that no queen may be taken by any other queen.

The new problem essentially consists of two parts:
1. Finding a method to generate further "solutions."
2. Determining whether all solutions were generated or not.

It is evidently necessary to generate and test candidates for solutions in some systematic manner. A common technique is to find an ordering of candidates and a condition to identify the last candidate. If an ordering is found, the solutions can be mapped onto the integers. A condition limiting the numeric values associated with the solutions then yields a criterion for termination of the algorithm, if the chosen method generates solutions strictly in increasing order.

It is easy to find orderings of solutions for the present problem. We choose for convenience the mapping

\[ M(x) = \sum_{j=1}^{8} x_j 10^{j-1} \]

An upper bound for possible solutions is then

\[ M(x_{\text{max}}) = 88888888 \]

and the "convenience" lies in the circumstance that our earlier program generating one solution generates the minimum solution which can be regarded as the starting point from which to proceed to the next solution. This is due to the chosen method of testing squares strictly proceeding in increasing order of \( M(x) \) starting with 00000000. The method for generating further solutions must now be chosen such that starting with the configuration of a given solution, scanning proceeds in the same order of increasing \( M \), until either the next higher solution is found or the limit is reached.

6. The Extended Program

The technique of extending the two given programs finding a solution to the simple 8-queens problem is based on the idea of modification of the global structure only, and of using the same building blocks. The global structure must be changed such that upon finding a solution the algorithm will produce an appropriate indication—e.g. by printing the solution—and then proceed to find the next solution until it is found or the limit is reached. A simple condition for reaching the limit is the event when the first queen is moved beyond row 8, in which case regression out of the first column will take place. These deliberations lead to the following modified version of the nonrecursive program:

```plaintext```
considerfirstcolumn;
repeat trycolumn;
    if safe then
        begin setqueen; considernextcolumn;
            if lastcoldone then
                begin PRINT(x); regress
                end
            else regress
        end
    until regressnotachieved
```plaintext```

Indication of a solution being found by printing it now occurs directly at the level of detection, i.e. before leaving the repetition clause. Then the algorithm proceeds to find a next solution whereby a shortcut is used by directly regressing to the prior column; since a solution places one queen in each row, there is no point in further moving the last queen within the eighth column.

The recursive program is extended with even greater ease following the same considerations:

```plaintext```
procedure Trycolumn(j);
begin integer i;
    (declarations of procedures testsquare, advancequeen, setqueen, removequeen, lastsquare)
    i := 0;
    repeat advancequeen; testsquare;
        if safe then
            begin setqueen; x[i] := i;
                if \( \neg \) lastcoldone then Trycolumn(j+1) else PRINT(x);
                removequeen
            end
    until lastsquare
end
```plaintext```

The main program starting the algorithm then consists (apart from initialization of \( a, b, \) and \( c \)) of the single statement \( \text{Trycolumn}(1) \).

In conclusion, it should be noted that both programs represent the same algorithm. Both determine 92 solutions in the same order by testing squares 15720 times. This yields an average of 171 tests per solution; the maximum is 876 tests for finding a next solution (the first one), and the minimum is 8. (Both programs coded in the language Pascal were executed by a CDC 6400 computer in less than one second.)

7. Conclusions

The lessons which the described example was supposed to illustrate can be summarized by the following points.

1. Program construction consists of a sequence of refinement steps. In each step a given task is broken up into a number of subtasks. Each refinement in the description of a task may be accompanied by a refinement of the description of the data which constitute the means of communication between the subtasks. Refinement of the description of program and data structures should proceed in parallel.

2. The degree of modularity obtained in this way will
determine the ease or difficulty with which a program can be adapted to changes or extensions of the purpose or changes in the environment (language, computer) in which it is executed.

3. During the process of stepwise refinement, a notation which is natural to the problem in hand should be used as long as possible. The direction in which the notation develops during the process of refinement is determined by the language in which the program must ultimately be specified, i.e. with which the notation ultimately becomes identical. This language should therefore allow us to express as naturally and clearly as possible the structures of program and data which emerge during the design process. At the same time, it must give guidance in the refinement process by exhibiting those basic features and structuring principles which are natural to the machine by which programs are supposed to be executed. It is remarkable that it would be difficult to find a language that would meet these important requirements to a lesser degree than the one language still used most widely in teaching programming: Fortran.

4. Each refinement implies a number of design decisions based upon a set of design criteria. Among these criteria are efficiency, storage economy, clarity, and regularity of structure. Students must be taught to be conscious of the involved decisions and to critically examine and to reject solutions, sometimes even if they are correct as far as the result is concerned; they must learn to weigh the various aspects of design alternatives in the light of these criteria. In particular, they must be taught to rework earlier decisions, and to back up, if necessary even to the top. Relatively short sample problems will often suffice to illustrate this important point; it is not necessary to construct an operating system for this purpose.

5. The detailed elaborations on the development of even a short program form a long story, indicating that careful programming is not a trivial subject. If this paper has helped to dispel the widespread belief that programming is easy as long as the programming language is powerful enough and the available computer is fast enough, then it has achieved one of its purposes.

Acknowledgments. The author gratefully acknowledges the helpful and stimulating influence of many discussions with C.A.R. Hoare and E. W. Dijkstra.

References

The following articles are listed for further reference on the subject of programming.


Announcement:

Graphics and Image Processing—A New Department in Communications

With this issue of Communications we are pleased to announce a new department: Graphics and Image Processing. The editor of this department will be William Newman of the University of California at Irvine.

This department will cover five principal areas:

- Hardware and software techniques for producing pictorial output;
- Methodology for the digitization and input of pictorial information;
- Methodology for the digital processing and storage of such information;
- Languages and techniques for interacting with graphical systems;
- Computer applications in which any of the above play a significant role.

Dr. Newman is professor of information and computer science at the University of California at Irvine. Previously, he was at the University of Utah in the Computer Science Department. He holds a Ph.D. in Computer Science from London University, was a Research Fellow at Harvard, and has been a member of ACM since 1967.