# CURRENT IDEAS ON PROGRAMMING METHODOLOGY

David Gries

TR 76-286

July 1976 (revised)

Department of Computer Science
Cornell University & Technical University Munich
Ithaca, New York 14853

A contribution for the book

The Impact of Research on Software Technology

edited by P. Wegner and W. Wulf

## A. Introduction*

### Topic definition

This contribution attempts to review and assess research in the area of programming methodology. Programming methodology covers the management, planning, design, implementation or development, verification, debugging and evaluation of programs. Since most of these topics are treated in other articles of this book, I concentrate here on one aspect of programming: the development of small (one to five pages long) correct programs.

I must justify this restriction. First, experience shows that the production of a small, correct program is itself a difficult task, which relatively few people have mastered. It is hard for me to understand how we expect to effectively develop large programs when an understanding of the development of small programs has to a large extent eluded us. We can even argue that the ability to produce large reliable systems requires the ability to regularly produce small, correct programs, as follows (Dijkstra):

A large program or software system is ultimately composed of n (say) small programs, or program "modules". Suppose each of these n independent modules has probability p of being correct. Then the probability P that the whole system is correct surely

---

* This article contains material rewritten from an early draft by the author of a similar section for the NSF COSERS report. I am grateful to Henry Ledgard for constructive criticisms.

satisfies $P \leq p^n$. Since n is large, in order to have any confidence in the reliability of the system at all, p must be very close to one.

Thus, I feel justified in concentrating on one single aspect of programming: the development of small, correct programs.

This article will not pinpoint numerous deep, detailed results, theorems or mechanical tools which, if only brought to the attention of the programmer, would cause an immediate rise in productivity and reliability. Instead, I hope to give an overall impression of the important ideas that have been emerging over the past ten years. Hopefully, the reader will come to the conclusion that while the ideas behind good programming seem simple, their conscious application is not. It requires education, a change of attitude on the part of the programmer, and much practice. Let us begin by assessing the past in order to provide a perspective on our current thoughts about programming.

## A very short history of programming

During the early years of computing, computers were relatively expensive, limited in power, and often unreliable. The main emphasis was on the computer: keeping it in working order and using it as efficiently as possible. The programmer's task was to code simple (by today's standards), small algorithms in the machine language of a particular computer, using as many clever tricks and techniques as possible in order to overcome restrictions on memory and speed. The fact that programs could modify themselves was thought to be a significant achievement, and sharing of memory for different purposes within a program, sometimes even first as data and then as instructions, was considered a clever way to beat the machine.

A program was a personal thing, rarely to be read by others. Programs were written for one machine, and for one purpose; rarely were they transported to other installations for use elsewhere.

With the emergence of FORTRAN and other high-level languages, programming methodology changed little. True, it became easier to program, but the idea was still to squeeze memory and time out of the machine, using as many clever tricks as possible. The good programmer knew FORTRAN on his machine well enough to "get at" the machine in spite of FORTRAN!

Programming in FORTRAN could be taught in one or two weeks to produce good programmers, depending on their ability to solve puzzles. Little attention was given to readability, adaptability, or even to correctness in the general sense.

This is not to say that the programming process was never discussed. Early programming texts did mention rules like "divide the problem into several, smaller parts", but for the most part these were glossed over as obvious. Learning the language itself was the important point, and programming was still the minor, simple part of the overall use of computers.

As computers became more powerful and flexible, as the cost of hardware decreased, as the problems given to programmers became more complex and large, and as programmers discovered that clever tricks used earlier were not enough, emphasis changed from hardware to software. The appetite of programmers and those who posed problems for complexity outgrew their ability to digest it. More and more time was spent debugging, deadlines were missed more frequently, and cost overruns became the rule rather than the exception.

In 1968 and 1969 two NATO conferences (see Naur [68] and Buxton [69]) were devoted to the problem of producing reliable software at a reasonable cost. While not everybody agreed to the use of the term "software crisis", all participants agreed that we really did not know how to produce software in a reasonable manner. Today we recognize that programming is a difficult task, and much research in programming methodology is being performed. This research has already had an effect on the programming world, enough to warrant more research in both theoretical and practical areas.

## B. Significant past research in programming methodology

The task of organizing ones thoughts in a way that leads, in a reasonable amount of time, to an understandable expression of a computing task, has come to be called *structured programming*. The term, first used by Dijkstra in his monograph *Notes on Structured Programming* [72], has shaken the programming world. Used in its narrowest sense (don't use *gotos*), it has of course been shouted down by most intelligent people. Used in the broader sense given above, it has influenced research and practise of programming methodology.

One might well ask whether programmers naturally practise structured programming. The answer is no, on three counts. First, the average programmer does not complete his task in a reasonable amount of time, as is evidenced by the frequent cost overruns and missing of deadlines. Second, his final program is not understandable; others have trouble reading it and later modifying it. Third, most programs do not satisfy the original specifications and are replete with errors, some of which are not found for years.

Important research in this field has been directed towards answering the following questions:

1. How should (could) the <u>process of developing</u> a program be organized?
2. How should (could) the <u>program</u> be organized?
3. How do we know a program is correct?
4. How should (could) the documentation be written so as to best describe the program?

A discussion of research directed toward answering these questions will make more sense if we first attempt to understand what problems the programmer faces, why programmers have difficulties, and what (mental) tools he has available to overcome them. I attempt this briefly in the next two subsections, interpreting ideas first presented by Dijkstra [72].

## B.1 The programmer's attitude

We must acknowledge that programming is a difficult, intellectual task, because of the size and complexity of the problems we tackle. Size is certainly a factor. Compilers have 5,000 to 50,000 lines of high-level language code, and operating systems sometimes 20 times that amount! Hence, any single person can only hope to remember or even read the details of only a small part of the program or programming system.

However program size is not the only culprit. A program of five or six lines can be difficult to understand if not organized and explained well. Two kinds of "complexity" confront us even in such small programs. First, we have the complexity of the computations effected by execution of the program. This kind of complexity we try to overcome partly by "structuring" the program and its description in some way (to be described later).

The second kind of complexity has to do with the "mathematical system" on which the program's proof of correctness (and other properties) lies. For example, consider a hash-coding scheme, where for a table of size n elements the successive probes for a key K will be at the elements numbered H1(K),H2(K),...,Hn(K). We usually desire these Hi(K) to be all different, so that if necessary <u>all</u> the elements of the table will be tested for the presence of key K. Furthermore, we usually desire other properties of the Hi, such as the absence of primary or secondary "clustering". Such properties can depend on very deep mathematical theorems which the programmer must discover, or at least understand.

The programmer is faced with problems of size and complexity, but there are limits to the amount of material and the degree of complexity he can digest. The programmer must first of all recognize his limitations, rather than ignore them, and seek ways to overcome them. Without this recognition of the difficulty of the task, failure must result.

The wise programmer restricts himself to _intellectually manageable_ programs - - those that can be understood in time proportional to their length. This rule actually helps the programmer. If he finds himeslf incapable of easily understanding something he has written, he immediately redoes it so that it _is_ understandable. He has others read his program _before_ committing it to the computer, so that he can be sure that others understand it. He welcomes their criticism. The ability to understand guides him in his choice of program structures and method of organization.

In other words, the programmer attempts to organize the chaos of details into an understandable program. He attempts to find notation and organization to simplify the complexity. In a sense, we might call structured programming _computational simplicity_, as opposed to computer science's already existing field, computational complexity.

I should also like to discuss the programmer's attitude towards program errors. The historic attitude is that errors are a necessary evil, and that finding and fixing them naturally require a good (30-60%?) percentage of the programmer's time. Hence the emergence of the terms bugs and debugging. Bugs, like mosquitos, are always present and must be swatted when found.

Dijkstra first recognized the futility of such an attitude, saying that program testing can never reveal the absence of errors (which is what we want) but only their presence. Others have noted the relatively high cost of fixing a detected error late in the testing process, as opposed to the cost of spending more time on program design and implementation (before testing) so that the error is detected much earlier or so that it never even enters the program.

Thus, while testing is necessary, the responsible programmer must write his program so that the detection of an error during testing is the exception rather than the rule. He develops and organizes his program so that he knows that it is correct, before testing begins.

What emerges from this discussion is that the programmer's attitude towards programming is extremely important. He must recognize his limitations and discover ways to overcome them. He must realize that his job is to produce a correct, readable program before testing. He understands that only through an intensive, ongoing study of the programming process and of the mental tools available to him can he learn to perform his job well.

## B.2 Our mental aids

In order to know what we can intellectually manage, the programmer must know what mental tools are available to help him. Dijkstra [72] discusses three important ones: enumerative reasoning, mathematical induction and abstraction.

We use enumerative reasoning to understand sequences of statements, conditional statements and the goto. In effect, we try to look at each possible execution path and understand that it works correctly. Enumeration is only an adequate tool when the number of cases to be considered is moderately small.

Mathematical induction is used to understand iteration (loops) and recursive procedures. The typical loop can be executed zero times, once, twice, or any number of times, and we use induction to see that all of these work correctly just as we use induction to prove properties of the integers. The use of induction will be illustrated in section B.3. For now we just mention that programmers must recognize that induction is an important tool and must learn how to handle it formally. Programmers need much more mathematical maturity than is currently recognized.

Abstraction can be thought of as the process of singling out one or more qualities or properties of an object for further use. The purpose is to be able to concentrate only on relevant properties of the situation and to ignore irrelevant ones. Abstraction permeates the whole of programming. The concept of a variable is an abstraction from its current value. When we write a procedure and then write several calls we are using abstraction. That is, when we write it we are concerned with _how_ it works; afterwards, we can forget completely about the _how_ and concern ourselves only with _what_ it does. In effect, we have extended our programming language with another operation. When we implement a new data type, say complex variables or linked lists, we again think of these data types as abstract objects that we can use.

Abstraction is a most powerful tool, used also in mathematics, and the programmer must be aware as to how and why he uses it.

Now, if enumerative reasoning (in small quantities), induction and abstraction are our main mental aids, then we should restrict ourselves to constructs and organizations which allow us to use them efficiently. Sequencing and alternation we understand through enumerative reasoning; iteration through induction; procedure and calls and programmer-defined data types through abstraction. Should we wish to use other program constructs, we must be sure beforehand that we can effectively understand them.

## B.3 On proving the correctness of programs

A proof of a theorem is an argument which convinces the reader that the theorem is correct. The proof may be formal, arising from axioms as a step-by-step application of inference rules as in logic; on the other hand it may be composed entirely of informal reasoning.

Evidence supports the statement that typical, informal reasoning used for programs is insufficient, and because of this debugging takes a major portion of the total project -- from 30% to 60% of the total time.

With simple problems or theorems, informal reasoning often suffices, but as problems become more complex, informal reasoning becomes less and less helpful, and we must rely on more systematic, formal techniques. Programs are by nature complex, detailed objects. Even a five or six line program can be incomprehensible unless explained in a systematic fashion. Hence, systematic, formal techniques must be developed for proving properties of programs. However, they must be practical enough to be used by programmers on "real life" problems, and hopefully should shed light on the programming process as a whole.

Since Floyd's [67] paper on assigning meaning to programs there has been much research on proving properties of programs. Much of the early work was very theoretical and helped little in practical programming; on the other hand, some was directed specifically at understanding the programming process. In 1972, a conference on proving assertions about programs was held in New Mexico. A bibliography on the subject can be found in London [75].

The method which has had the most influence on programming to date is Hoare's <u>axiomatic approach</u> [69], which is based on Floyd's [67] earlier work. Hoare's work has already taught us much about programming; indeed it forms a good part of the foundation of structured programming, and I would like to outline it here.

Hoare [69] formally defines a programming language -- a fragment of ALGOL. The definitions of the language constructs are designed precisely to indicate how to prove properties of programs using the constructs. The meaning of a construct is given in terms of assertions about the input variables and output variables of the construct. As an example, suppose P is an assertion, x:=e an assignment statement, and $P[e \rightarrow x]$ of textually substituting (e) for every occurrence of x in P. Then the definition of the assignment statement is

$$\{P[e \rightarrow x]\} \quad x:=e \quad \{P\}$$

which informally reads: if $P[e \rightarrow x]$ is true before executing the assignment x:=e, then P is true afterwards. As an example we have $\{(a+b)+c>0\}$ d:=a+b $\{d+c>0\}$.

Note that this definition indicates nothing about how to execute the assignment statement. It describes only assertions or relations between variables that hold before and after the execution. Thus our attention is turned away from how to execute things, and towards the more static and easier-to-observe objects, the assertions.

The definition of the while loop is perhaps less trivial and more useful:

(1) Under the assumption $\{P \wedge B\}$ S $\{P\}$
the following holds: $\{P\}$ while B do S $\{P \wedge \neg B\}$

In English, we read (1) as follows. Suppose execution of the loop body S under the precondition B leaves a particular assertion P invariantly true. Then, if the loop <u>while</u> B <u>do</u> S is executed with P true initially, upon termination P will still be true and moreover B will be false.

Remember, (1) is the <u>definition</u> of the loop. It teaches us to understand a particular loop within a program in several steps, as follows.

1. Show that P is true initially, before execution of the loop;
2. Show that the desired result R of execution follows from $P \wedge \neg B$;
3. Show that $\{P \wedge B\}\ S\ \{P\}$;
4. Show that the loop halts (by other means, although this can be included in the loop definition also).

The power of this definition can be seen on the following oft-used but simple example. Suppose we have integers a, b > 0, and suppose we want a program segment to calculate $z = a^b$. A simple program segment for this is

```
z:=1; x:=a; y:=b;
while y ≠ 0 do
   begin s1: while even(y) do
                  begin y:=y/2; x:=x*x  end;
         s2: y:=y-1; z:=z*x
   end
```

This is a <u>short</u> program segment. Yet as given it is difficult to understand; even the comments given by the average programmer would not help, for they would attempt to explain from an operational point of view what is happening. Suppose however that we give the following assertion P and appeal to definition 1 of the while loop:

$$P \equiv (z \cdot x^y = a^b) \wedge y \geqslant 0$$

It is easy to verify (1) that P is true initially; (2) that $P \wedge y=0$ imply the desired result; (3) that execution of the sequence s1; s2 leaves P true; and finally (4) that the loop halts, since each execution of the loop body decreases y by at least one.

Thus the introduction of the single comment $\{P: z \cdot x^y = a^b \wedge y \geqslant 0\}$ is enough to help any educated reader understand the program.

Hoare's work is theoretical; formal proofs of correctness of a program are like proofs in logic, leading from the axioms or definitions of the basic statement types, through a step-by-step application of inference rules, to the program with assertions as a proved theorem. Yet his technique can be applied informally, with the amount of formality and detail needed being directly proportional to the complexity of the program. The practicality of the method cannot be refuted; whether it will be accepted in the near future by programmers, or whether the average programmer has the education and ability to understand and use it, is another question.

Hoare's method teaches us to work with assertions or relations about values of variables instead of the actual values themselves. Thus we begin to think more about the static, mathematical aspects of programs -- the assertions -- instead of the dynamic behavior which is difficult to understand. We learn to think less in terms of test cases; we learn that hand simulation of the program for one particular case does little to help us prove correctness. We learn how to understand loops in terms of the loop invariant. The loop is the most difficult programming construct to use and understand; finally we can control it. Introductory programming texts are beginning to incorporate ideas stemming from work on correctness proofs and structured programming; we mention Conway and Gries [73], McGowan and Kelly [75], and Wirth [73]. Hoare's method has also been an important tool in programming language research in the past 8 years; a discussion of this, however, is beyond the scope of this paper.

Some people have cited the need to produce an invariant relation for a loop as a major disadvantage of Hoare's method. I claim this as a major <u>advantage</u>, for it forces the programmer to make explicit -- both to himself and to the reader -- that which he has been doing implicitly, vaguely, imprecisely and incorrectly all along.

Several examples of proofs of program correctness have appeared in the literature. These show that proofs of correctness can be given for complicated programs of 2-3 pages, and not just small ones as the program given above. The examples show that, if done judiciously, a proof of correctness leads to better understanding in less time. As primary examples we cite Hoare [71], Gries [73], Gries [75], and London [70].

Research is also being performed on the mechanical verification of program correctness, mostly with the aid of interactive systems in which the programmer plays a role. This research is worthwhile and should be continued, but its importance does not lie in the future possibility of having such systems for all programmers to use. It is important because it can help shed more light on the programming process and our understanding of it and thus can help us develop programming methodology and mental tools for the programmer himself to use.

## B.4 Developing programs and their proofs

An important point is that a program and its correctness proof must be developed hand-in-hand, with the proof ideas generally leading the program development. One cannot expect to produce the whole program and then prove it correct. Instead, at each stage of development, the programmer must know that what he has done is correct.

Just how proof ideas could lead program development has not been at all clear. Recently, Dijkstra has published a new book (Dijkstra [76]) which provides some exciting, new insight on this problem. Dijkstra provides a "calculus" -- a set of rules -- for deriving programs. Successful application of these rules leads to a correct program. Of course, as with the integral calculus, we may not be able to apply it successfully. Success depends on the ability of the applier and the problem to which the rules are applied.

Dijkstra's new twist comes from reasoning that the definition of a statement type should reflect how the definition is to be used in deriving programs. Thus, he defines a statement type S by giving the rule for deriving the weakest assertion (the precondition) wp(S,R) for which execution of S will establish the desired postcondition R. For example, the assignment statement defined earlier as $\{P[e \rightarrow x]\}$ x:=e $\{P\}$ is redefined using

$$wp(\text{"x:=e"},P) \underset{df}{=} P[e \rightarrow x]$$

This subtle change is enough to give us deeper insight. No longer are Q, S and R treated equally in $\{Q\}$ S $\{R\}$. Instead, the definitions say that the precondition Q must be derived from S and R.

This is actually the reverse of what most programmers think and it may indeed be difficult for them to break the habit of always thinking in a purely operational manner -- in terms of how the computer executes a program. The typical programmer attempts to work "forward" by developing a statement S which, given precondition Q, will establish the postcondition R. Dijkstra's more mathematical definition advises us to develop the statement S by concentrating on the postcondition R, and by looking on S as a statement which transforms the postcondition R into the precondition Q.

With practice, this new technique turns out to be more convenient and reliable than conventional programming. The book Dijkstra [76] is filled with examples of idealized versions of proof-and-program development in this manner; another example appears in Gries [76]. This book represents one of the most significant advances in programming methodology in the 1970's.

## B.5 Top-down or bottom-up?

Two major methods for developing a program have been recognized: the top-down and bottom-up methods. In the former approach, one starts with a statement S of the problem, and breaks it down into a sequence S1; S2; ...; Sn of statements which, if executed in order, perform the same as statement S. Each of these statements Si is then refined in turn, until each of the statements is in the programming language. This method has also been called step-wise refinement. (One similarly refines data structures in this process.)

In the bottom-up approach, one begins with the programming language and writes low-level subroutines which one hopes will be useful. These are then used to implement higher-level routines, and so on until a subroutine is written which solves the problem.

It should be evident that neither method can be used in its pure form. Any program development has characteristics of both. A programmer who leans toward the top-down method is guided by the programming language into which he is programming, and he will often write low-level routines in an attempt to understand what can be done there. Similarly, a programmer who programs bottom-up is guided by the problem, and hopefully has some idea where he is going. Moreover, there is always an initial design phase (the size of which is proportional to the size of the project - very small for small programs) in which the overall structure and organization is determined.

The top-down approach is exhibited formally in Dijkstra's calculus mentioned briefly in Section B.4. More informally, scientists have been arguing for top-down and against bottom-up programming, for several reasons. First of all, one has a correct program for the problem at each step; the programmer makes his refinements small enough so that its correctness is obvious. Secondly, it is easier to see what the major issues of program design are at each stage, and to choose the most important one to solve next. Thirdly, with the bottom-up approach, one defines interfaces of routines largely in a vacuum; it is difficult to know exactly what will be needed at higher levels. In the top-down approach, one can design interfaces in the context where they will be used, and <u>then</u> write the routines. Fourthly, the best documentation of a program seems to be a top-down description which shows all the abstractions used in a way the reader can understand; top-down development is more attuned to this. Finally, one refinement we always make is to implement some abstract data structure -- a set of values and operations on them. It is important to know just what operations are to be performed and how often, so that the data structure can be implemented as efficiently as possible. The top-down technique allows one to put off implementing data structures until the operations to be performed on them have been written; in the bottom-up technique, these data structures are designed and implemented without this knowledge.

Top-down programming and good examples of developing programs in this way were given by Dijkstra [72], Dijkstra [76], Wirth [71] and Wirth [73]. The technique has been applied, with good results, to small and large programs alike. We mention in particular the Times Information project, described in Baker [72] and McGowan [75].

## B.6 Program organization

The organization of program development and the organization of a program are two entirely different aspects of programming. Many people feel that the term structured programming refers to the latter; my own opinion, after reading Dijkstra [72] where the term first appeared (and there, as far as I can see, only in the title!), is that it refers more to the former.

Of course, the organization of the development has a marked effect on the organization of the program. Too often, the structure and development of a large system is determined by the structure of the group which produces it. Common sense now tells us that the problem and its evolving solution should determine the group structure, although 10 years ago this was not so clear. Evidence, for example from experiments with the chief programmer concept of Baker and Mills (see Baker [72]), supports this common sense.

The final organization of a program is extremely important, for the ability to read and understand a program, to "maintain" it, and to modify it depends to a large extent on its organization.

By and large, a program is modified for years after the initial development is finished, and its "maintainability" is an extremely important factor in its success.

I find it difficult to carefully review research in this area in a few pages. Much of what one can say appears trivial or obvious, after it has been said. And yet the organization of programs is an important topic which has only been studied in the past 10 years. Let me illustrate this by listing five so-called trivial or obvious statements about organization, and with each, try to explain and analyze it somewhat.

1. Don't use gotos - use only sequencing, alternation and iteration. This statement lies at the heart of the structured programming controversy. Behind it lies the real principle to follow: make the structure of the program text reflect the structure of the computations it evokes. In other words, the static program text should be as close as possible to the dynamic aspect of the program -- how it gets executed. We want the conceptual gap between the program as we read it and the program as it gets executed, to be as small as possible.

It is clear that using only one-in-one-out control structures, such as sequencing, alternation and iteration, helps in this respect (see e.g. Ledgard [76]). Moreover, one should attempt to use control structures which have been designed to aid program development, such as Dijkstra's [76] guarded command structures.

Certainly, any programming construct may be used, as long as it can be established that its use adheres to the real principle mentioned above.

2. Make the program structure fit the problem structure. Like most of the phrases listed here, this seems obvious until we begin to think about it, attempt to apply it, or attempt to explain it to others. Perhaps the following example will suffice to illustrate the problems associated with it. Consider the task of reading in a sequence of "words" (on input cards -- a word may be split onto two successive cards) and printing them out, as many to an output line as possible. This is the basic function of any text editor.

I have seen too many students structure their programs with respect to the input cards or to the output line, leading to programs with one of the following forms:

```
while ∃ another card do                    while ∃ more input do
  Process the card                or         begin Build next output line;
                                                   Write out output line
                                             end
```

Although the programmer may refine these into a program which uses only sequencing, alternation and iteration, the resulting program is still far too complex. This problem has to do with words, and thus the program should be organized around words, rather than input or output lines. The following outline can be refined into a program which is much more easily understood:

```
L:=empty line; {L is the current output line being formed}
while ∃ another input word do
    begin if word does not fit on L
             then [write out L; L:=empty line];
          Add word to L
    end;
if L is not empty then Write out L
```

While the idea of making the program structure fit the problem structure may seem trivial, knowing how to do it is not easy, and teaching others how to do it is very difficult. Currently, we teach others mainly by presenting many examples, and hoping the students will catch on. What does become clear is that practicing syntactic restrictions, such as not using gotos, does not automatically mean that a program will be "well-structured". Far more difficult and subtler issues dealing with the meaning of the program are involved.
3. Use hierarchical structure. One should describe a program in terms of hierarchical levels of abstraction. This allows us to read the program at several levels of detail, depending on what we are interested in. We can consider each level to be a "machine language", with the machine being implemented by the level just below it. Such a hierarchical structure helps organize the mass of detail into a coherent whole, which can be read and understood using the mental tool of abstraction.

Typically, one implements each level using procedures, macros, and data type definitions.

Just how levels should be organized is, of course, another matter. We want to organize the program in order to facilitate understanding and later modification. This problem is equivalent to the mathematician's problem of organizing a complicated proof of a theorem, using lemmas which rely on other lemmas, etc.

4. Modularize. We have all heard that programs should consist of small modules, with "narrow" interfaces between them. Yet, few define the term module. A module can be a procedure, a function, a macro, a concrete representation of an abstract data type, a set of various program pieces which together implement some functional specification.

Not all "modularizations" are equally good, of course. The problem is to know how to organize a program into modules in an effective manner, and how to teach others to do this. This calls for more explicit measures of the "goodness" of modularizations. One gets a good feel for effective organization by studying "modern" works on programming such as Dijkstra [72]. Myers [75] attempts to be more explicit about measures of goodness, such as module strength and module coupling, and attempts to illustrate their use on practical examples.

5. Document the program. Programmers do not like to draw flowcharts, and usually draw them after the program is finished. Research, however, indicates that it is the programmer himself who needs this documentation to help him develop the program. Documentation and program should be developed hand-in-hand. An interesting sidelight is that experiments are beginning to confirm our opinion that flowcharts are not of too much use (schneiderman [76])

Documentation should bring out the hierarchical structure of the program; it should point out the abstractions used in it. It is generally felt that a top-down description most easily followed, irrespective of how the program was developed.

With high-level languages, it is possible to use comments and indentation as the full documentation for the reader (outside of documentation which tells how to use the program). This is discussed by Linger [75], while Conway [73] has developed such a style over the years in teaching programming. (See also Gries and Conway [75].) The principal principle is to treat comments as high level statements, indenting them as one would indent any statement. The refinement of a comment, which indicates <u>how</u> the comment statement is to be carried out, appears indented below the comment. For example, consider the program outlined below, using PL/I conventions for comments. The problem S consists of the concatenation of three statements S1, the loop and statement PS4. The high-level statement S1 is itself implemented as S1.1; PS3, while S1.1 is itself implemented PS1; PS2.

```
/*S*/
   /*S1*/
        /S1.1*/
             PS1;
             PS2;
        PS3;
  WHILE B DO
       BEGIN ...
       END;
  PS4
```

Such documentation reveals the hierarchical structure of the whole program, and allows the reader to study the program at any level of details without the need for more, separate documentation.

## C. Discussion

The points mentioned in the previous section may seem too obvious to be presented as research results. That top-down and bottom-up are two major methods of program development seems trivially true. That even small programs can exhibit astounding complexity, and that we do have intellectual limitations is patently clear. That testing can only show the presence of errors and not their absence is obvious. In presenting such an overview we run the risk of turning the reader away from delving further into the subject. The reader must realize that we cannot go much deeper without getting into too many details. In addition, I would like to say the following.

First, before 1968 (or thereabouts) few people realized these so-called obvious facts; only with the emergence of Dijkstra's *Notes on Structured Programming* did computer science as a whole become aware of the problems of programming and their possible solution.

Secondly, the best research is not that which confounds us with its complexity, but that which impresses us with its simplicity and naturalness. The discovery of hitherto unknown simple ideas whose practical application leads to significant advances is what we need, especially in a field like programming. However, we must realize that although the ideas may be simple, we cannot always expect their practical application to be an easy task. In this regard, I like the following saying -- I don't know to whom I should attribute it:

> Never dismiss as obvious any fundamental principle, for it is only through <u>conscious application</u> of such principles that success will be achieved.

Recognizing a principle and consciously applying it are two different things. One of our human shortcomings is that we want simplified, easy solutions to our difficult problems. Because of this, we tend to forget about the fundamental principle we should be following, and concentrate instead on some single, sometimes trivial, idea which implies the principle. For example, many have simplified the principle "make the program text reflect the structure of computations evoked by it" into "don't ever use gotos". Naturally, people presented <u>only</u> with the latter statement balk at it.

Another example from Tony Hoare is the following. When loading programs for execution we want flexibility <u>and</u> efficiency. Since the early 1960's we have attempted to achieve this by having the compiler produce object modules, and having a linking loader link modules together and lead them for execution. Gradually, the principle of efficiency with flexibility has been replaced by the requirement "the compiler must produce an object module, and there must be a linking loader". This latter requirement appears in the specification for every new compiler or system, completely excluding other solutions to the problem of efficiency with flexibility.

As a third example, the seemingly easiest solution to the ever increasing cost and time of testing and debugging is to develop more and better mechanical debugging aids and mechanical verifiers. However, the real solution, which is difficult, is to learn enough about programming so that we can teach the programmer not to put bugs into his program in the first place.

I am supposed to speculate on future research in this article, but I really don't feel capable of doing so. I have difficulty predicting my own particular area of research in two years, much less that of others. I do feel that though we have made fantastic progress in the past ten years, much still remains to be done. We have identified some important principles; we must now learn how to apply them effectively. We have a framework for proving programs correct and a formal calculus for the development of programs; the methods must be developed and refined and extended and made digestible for the programmer. We still do not have practical methods for understanding huge areas of programming, (e.g. pointers), nor do we have practical replacements for them.

Up to this point, there has been some "impact of research on software technology". Most programmers have heard of "structured programming" -- even if they do not understand it completely -- and they try to organize their programs more effectively. They have, to some extent, accepted the more trivial ideas (e.g. don't use gotos).

But practicing programmers do not understand the deeper issues involved in programming, as discussed in this article. Many programmers have not even heard of (and few use) important concepts which they should be using daily in their work; examples are "proof of correctness", precondition", "invariant relation of a loop", and "axiomatic basis for a programming language".

Our main hope of further advancement in the practice of the software development lies not with better management techniques or better automated tools, but with the programmer himself. His attitudes and habits must change. He must have the feeling that he can develop small correct algorithms before testing begins, and that as a professional programmer it is his duty to do so. At the same time, he must realize that he cannot hope to develop a correct algorithm unless he learns to curb complexity -- unless he learns to organize and present a program as simply and clearly as possible. This will not obviate the need for testing, but the detection of errors during testing should tend to become the exception rather than the rule. Furthermore, if he has done his job well, the errors detected will be trivial to fix, arising more from simple transcription errors rather than from gross log ical inconsistencies and bad design.

Such a change of attitude requires education, and this is difficult to implement. For example, I dare say that the majority of the programming <u>teachers</u> do not (yet?) agree with me when I say that the programmer must produce a correct algorithm before he begins testing. More practical experience must be gained with these new ideas, and this experience documented in a convincing manner. The experience must influence the content of textbooks. This is happening to some extent now - see the annotated bibliography - but I would hope that 20 years from now new texts will show the same order of improvement over current texts as current texts show over those produced in and before 1955.

Such a radical change also requires new attitudes on the part of managers. Productivity can no longer be measured only in terms of lines of code produced - irrespective of how good they are. Attention must be given to different project organizations (e.g. the chief programmer team), to new development techniques, and to radical ideas such as having programmers read each other's programs.

Above all, programmers must be given time to study programming, on a regular basis (2-3 hours per week?). Programming methodology has changed radically in the past, and will continue to grow and develop. The only way to lessen the time gap between research and the application of its results is to allow the study of research results on a regular basis.

## D. Annotated Bibliography

This bibliography is supposed to acquaint you with a relatively small set of books and articles from which the important ideas necessary for good programming can be found. These will lead the reader to other sources. Where possible, I have chosen books which are developed in a single consistent, style, as opposed to the set of original articles from which the same information might be gathered.

Conway and Gries [73, 76]. These introductory texts attempt to teach the principles of good programming methodology, rather than just the programming language (PL/I) itself. The success can be partially measured by the fact that the Primer [76] has been quite easily "translated" to use PASCAL instead of PL/I (Conway, Gries and Zimmerman [76]). Conway and Gries [73] includes a section on informal correctness proofs, discussing mainly correctness of loops.

Dahl, O.-J., E.W. Dijkstra, and C.A.R. Hoare [72]. This book contains three important monographs. The first, by Dijkstra, is discussed in this bibliography under Dijkstra [72]. The second, by Hoare, discusses the definition, use and implementation of abstract data structures. The third monograph, by Dahl and Hoare, explores certain ways of program structuring and their relationship to concept modelling, based on SIMULA 67. This book is necessary reading for any one seriously interested in programming.

Denning [74]. This edition of Computing Surveys contains a number of articles with discussions, facts and opinions on structured programming. Knuths [74] detailed article on the use of the goto is included.

Dijkstra, E.W. [72]. Notes on Structured Programming, in Dahl, Dijkstra and Hoare [72]]. This is perhaps the first attempt to understand the programming process. It is extremely well done and illuminating. It discusses most of the principles underlying the development and understanding of programs. It should be studied by all.

Dijkstra, E.W. [76]. A Discipline of Programming, Prentice Hall, Englewood Cliffs, 1976. I quote from the forward by Tony Hoare: "This book expounds, in its author's usual cultured style, his radical new insights into the nature of computer programming. From these insights, he has developed a new range of programming methods and notational tools, which are displayed and tested in a host of elegant and efficient examples. This will surely be recognized as one of the outstanding achievements in the development of the intellectual discipline of computer programming.

Hoare [69], Hoare [70], Hoare [72], Clint and Hoare [72], Hoare and Wirth [73]. Most of the work on defining a language so as to facilitate and guide proofs of correctness is due to Hoare. These articles give the definition of various language constructs and provide examples of proofs of correctness to illustrate their use. The book which best (by far) describes formal aspects of correctness and their use in developing programs is Dijkstra [76].

Infotech State of Art Report on Structured Programming. Infotech International, Maidenhead, Berkshire, England, 1976. This report contains a well-done 130 page analysis of structured programming, together with about 20 invited papers on the subject. It also contains an annotated bibliography with 76 entries.

Jackson, M.A. [75]. This text is about the design of programs for data processing applications; it uses a subset of COBOL. It assumes a knowledge of programming.

Kernighan and Plauger [74]. The book discusses various aspects of style in programming, illustrating their advice by discussing poor programs taken from other programming texts, and their improvements. It follows the format of the noted book on English style, Elements of Style, by Strunk and White.

Ledgard [75]. This book is also based on the format of Strunk and White's Elements of Style; it contains 26 programming "proverbs" with illustrations of their use, a chapter on top-down programming, and a chapter of miscellaneous topics. A version oriented exclusively to FORTRAN also exists.

McGowan and Kelly [75]. This book is included here for its informal, practical treatment of the use of loop "invariants" for understanding iteration, of top-down programming, and of the chief programmer team concept as a method of managing and organizing a programming project. For the latter, see also Baker [72].

Sigplan Notices [75]. This is the proceedings of the International Conference on Reliable Software, held in Los Angles in April 1975. You will find articles on almost all aspects of programming, indicating current and future research in the field.

Weinberg [71]. The Psychology of Computer Programming treats a topic which is mostly ignored by computer scientists. Light, witty, and full of anecdotes, it makes some very good points and should be read and enjoyed by all.

Wirth, N. [73]. An excellent text which introduces concepts of verification and correctness, and which discusses step-wise program development in detail.

## References

Baker, F.T. System quality through structured programming. AFIPS Conference Proceedings 41, Part 1 (1972), 339-343.

___. Chief programmer team management of production programming. IBM Systems J 11 (1972), 56-73.

Buxton, J.N. and B. Randell. Software Engineering Techniques. Report on a conference sponsored by NATO SCIENCE COMMITTEE, Rome, Italy, Oct. 1969.

Clint, M. and C.A.R. Hoare. Program proving: jumps and functions. Acta Informatica 1 (1972), 214-224.

Conway, R. and D. Gries. <u>An Introduction to Programming: a structured approach</u>. Winthrop, Cambridge, Mass. 1973 (2nd edition, 1975).

___ and ___. <u>Primer on Structured Programming using PL/I, PL/C and PL/CT</u>. Winthrop, Cambridge, Mass. 1976.

___, ___ and E.C. Zimmerman. <u>Primer on Structured Programming using PASCAL</u>. Winthrop, Cambridge, Mass. 1976.

Dahl, O.-J., E.W. Dijkstra, and C.A.R. Hoare. <u>Structured Programming</u>. Academic Press, London, 1972.

Datamation 19 (Dec 73)

Denning (ed.) Computing Surveys 6 (Dec 74). Special Issue on Programming.

Dijkstra, E.W. Goto statement considered harmful. CACM 11 (March 68), 147-148, 538, 541.

___. A short introduction to the art of programming. EWD316, Technical University Eindhoven, the Netherlands, (Aug 71).

___. Notes on Structured Programming. In Dahl [72].

___. Turing Award Lecture. In CACM 15 (Oct 72), 859-866.

___. A Discipline of Programming. Prentice Hall, Englewood Cliffs, 1976.

Floyd, R.W. Assigning meanings to programs. In Math. Aspects of Computer Science, XIX American Math. Society (1967), 19-32.

Good, D. Toward a man-machine system for proving program correctness. PhD thesis, Wisconsin, 1970.

Gries, D. Describing an algorithm by Hopcroft. Acta Informatica 1973.

___. Proof of correctness of Dijkstra's on-the-fly garbage collector. In NATO Summer School, Marktoberdorf, Germany, 1975.

___. An illustration of current ideas on the derivation of correctness proofs and correct programs. Second International Conference on Software Engineering (Oct. 1976).

___, and R. Conway. The use of comments in programming. Submitted to CACM.

Heckel, C. (editor). Proceeding of conference on test methods. Prentice Hall, Englewood Cliffs, 1972. Proceedings of a Conference held in North Carolina.

Hoare, C.A.R. An axiomatic approach to computer programming. CACM 12 (Oct 69), 5 76-580, 583.

___. Procedures and parameters: an axiomatic approach. Symposium on Semantics of Algorithmic Languages. Springer Verlag, 1970.

___. A note on the for statement. BIT 12 (1972), 334-341.

___. Proof of a program: FIND. CACM 14 (Jan 71), 39-45.

___, and N. Wirth. An axiomatic definition of the programming language PASCAL. Acta Informatica 2 (1973), 335-355.

Infotech State of the Art Report 11. Software Engineering. Infotech Limited, Berkshire, England, 1972.

Infotech State of the Art Report 7. High-level languages. Infotech Limited, Berkshire, England, 1972.

Infotech State of the Art Report. Structured Programming. Infotech Limited, Berkshire, England, 1976.

Jackson, M.A. Principles of Program Design, Academic Press, New York, 1975.

Knuth, D.E. Structured programming with go to statements. In Denning [74].

Kernighan, B.W. and P.L. Plauger. The Elements of Programming Style. McGraw-Hill, New York, 1974.

Ledgard, H.F. Programming Proverbs. Hayden Book Co., Rochelle Park, N.J., 1973.

Ledgard, H.F. and M. Marcotty. A geneology of control structures. CACM 18 (Nov 75), 639-650.

Linger, R.C. and H.D. Mills. Difinitional text in structured programming. IBM, Federal Systems Division, Gaithersburgh, Maryland.

London, R.L. Computer interval arithmetic: definition and proof of correct implementation. JACM 17 (Oct 1970).

___. A view of program verification. Proceedings, International Conference on Reliable Software, April 1975. (SIGPLAN Notices 10, June 1975, pp. 534-545).

McGowan, C.L. and J.R. Kelly. Top-down Structured Programming Techniques. Petrocelli Charter, New York, 1975.

Mills, H.D. Top-down programming in large systems. In Debugging Techniques in large systems. Randall Rustin (ed). Prentice Hall, Englewood Cliffs, N.J. 1971, 41-55.

___. Mathematical foundations of structured programming. IBM Report FSC 72-6012, May 1972.

Myers, G.J. Reliable Software through Composite Design. Petrocelli Charter, N.Y., 1975.

Naur, P. and B. Randell. Software Engineering, a report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, Oct 1969.

Naur, P. Programming by action clusters. BIT 9 (1969), 250-268.

Owicki, S. Axiomatic approaches to proving properties of parallel programs. PhD thesis, Cornell University, Aug 1975.

Proceeding of conference on Proving Assertions about Programs. New Mexico, Jan 1972.

Schneiderman, B. and D. McKay. Experimental investigations of computer program debugging and modification. TR 48, Computer Science Dept., Indiana University, April 1976.

Sigplan Notices 10 (June 1975). Proceedings, International Conference on Reliable Software.

Weinberg, G.M. The Psychology of Computer Programming. Van Nostrand Reinhold, N.Y., 1971.

Wirth, N. Program development by stepwise refinement. CACM 14 (April 71), 221-227.

___. Systematic Programming: an Introduction. Prentice Hall, New Jersey, 1973.

___. On the composition of well-structured programs. In Denning [76].