Behavior Research Methods & Instrumentation
1979, Vol. 11 (2), 229-233

An introduction to structured programming

KARL P. HUNT
American National Bank, Chattanooga, Tennessee 37350

Structured programming (SP) is a technique devised to improve the reliability and clarity of programs. In SP, control of program flow is restricted to three structures, sequence, IF THEN ELSE, and DO WHILE, or to a structure derivable from a combination of the basic three. Thus, a structured program does not need to use GO TOs or branches (unless it is written in a language that does not have statement forms corresponding to the SP structures, in which case, GO TOs may be used to simulate the structures). The result is a program built of modules that are highly independent of each other. In turn, this allows a programmer to be more confident that the code contains fewer logic errors and will be easier to debug and change in the future. However, SP may be less efficient than an unstructured counterpart. Another disadvantage is the relative difficulty of using SP with a language that doesn't support it, although this situation is changing as languages are updated (e.g., FORTRAN 77).

During the last 15 years, many people working in the area of computing have expressed concern with the problem of software reliability. While the cost of hardware has decreased as power has increased, software costs have increased and become more complex. The result is that, although we can do more than before, it is at the risk of encountering more serious problems than before, and the problems usually prove to be more resistant to solution. In response to this situation, several approaches to programming have been devised to make programs more reliable and, at the same time, boost the amount of relatively bug-free code a programmer can produce. The most popular approach has been structured programming (SP).

Much of the development of SP can be traced to the work of Dijkstra (Dahl, Dijkstra, & Hoare, 1972; Dijkstra, 1965, 1968, 1969). Dijkstra's objective was to define a class of programs for which correctness proofs can be relatively easily provided. The class is that of structured programs.

The European efforts of Dijkstra and others were primarily academic. In the U.S., Mills and his colleagues successfully used SP in applied projects for IBM (Baker, 1972; Baker & Mills, 1973). In fact, Mills (Note 1) is one of the most enthusiastic supporters of SP used in conjunction with other design methods.

The popularity of SP has increased such that a number of texts on it have appeared, some devoted to teaching a specific language via a SP approach (Conway & Gries, 1973; Hughes & Michton, 1977; McGowan & Kelly, 1975). Clearly, SP is believed by many to be an important development in programming.

Requests for reprints should be sent to Karl Hunt, 1413 Cinderella Road, Lookout Mountain, Tennessee 37350.

AN EXAMPLE OF SP

While some may disagree on an exact definition of SP (McCracken, 1973), there is consensus on its two primary features: (1) a structured program is a hierarchical arrangement of highly independent modules, and (2) flow within the program is controlled exclusively by only three forms of control, selection, repetition, and sequencing. The basic control structures proposed to implement these are IF THEN ELSE for selection, DO WHILE for repetition, and simply placing statements one after the other for sequencing. There are other possible structures, but only two are discussed here: DO UNTIL (another way of controlling repetition) and the case structure, which allows for selection from a large number of alternatives. This list does not include the GO TO (or jump or branch, whichever term suits the language you are familiar with).

To help illustrate these characteristics, an example of an on-line program for control of a paired associate recall experiment is given in Figure 1. The number of stimulus-response pairs is variable, and feedback ("wrong" or "good") may or may not be given at recall. Other parameters can be introduced, such as length of presentation, but are treated as constant to keep the example simple.

The example as shown in Figure 1 is written in pseudocode. Pseudocode, which is often suggested as an alternative to flowcharts, is a natural-language version of the code in which the program will actually be written. Just as in a flowchart, its primary purpose is to represent the logic that controls the flow of the program. Thus the control structures are emphasized by being written in all uppercase letters and by the use of indentation. Pseudocode is a good intermediate step to writing a structured program even if a flowchart

```
determine if feedback is to be given;
input number of pairs;
DO UNTIL all pairs are input;
    input stimuli and responses to file;
END DO;
DO WHILE subjects remain to be run;
    present instructions;
    DO WHILE present subject is still below criterion;
        generate random permutation of pairs and of stimuli;
        DO WHILE there are still pairs to be presented on this trial;
            present next pair;
        END DO;
        pause for delay interval;
        reset criterion-not-reached switch;
        DO WHILE pairs remain to be tested;
            present stimulus;
            input response;
            IF response is wrong;
                turn on criterion-not-reached switch;
                IF feedback should be given;
                    print "wrong";
                END IF;
            ELSE;
                IF feedback should be given;
                    print "good";
                END IF;
            END IF;
        END DO;
        pause for intertrial interval;
    END DO;
    dismiss subject;
END DO;
```

Figure 1. Program for an on-line paired associate experiment, written in pseudocode.

is drawn first, since it approximates the actual code itself. It is fairly easy to translate a pseudocoded program into final code, especially when the language used contains statements that correspond to the control structures of SP. Unfortunately, not all languages do this.

Looking at the example, one can see that the first two statements are housekeeping chores in which the experimenter sets up the experiment. These illustrate control by sequence. Here, the sequence is trivial since it could have been reversed with no adverse effect; elsewhere, of course, it will often be more critical, especially in the overall sequencing of the major elements of the program.

The next statement is a DO UNTIL, a variation on DO WHILE. Its effect is to repeat the statement that follows it until the condition is met and then to allow control to proceed to the statement following the END DO. Specifically, the program will ask the experimenter to continue entering the stimuli and responses for the experiment until all n pairs are input. The input statement itself is indented to show that it is under the control of the DO UNTIL.

The rest of the program is a more elaborate demonstration of the use of indentation. The statement "DO WHILE subjects remain to be run" exerts control over all statements that follow it down to the END DO at the bottom that lines up at the margin with the DO WHILE. These statements control the experiment. There are several levels of indentation here, each representing the extent of control of some control structure. Indenting is not a necessary part of SP, but it is almost universally used to make the structure of a program clearer, not just in pseudocode or formal algorithms but in program code also, when possible.

Following the "DO WHILE subjects remain . . ." is a statement that provides for giving the next subject the instructions for the experiment. The DO WHILE below it is responsible for repeating the study and recall phases as long as the subject has not reached criterion, which in the present case is 100% correct on one trial. The extent of this DO WHILE's control is defined by the END DO which is the third statement from the bottom. Again, note that the END DO is indented the same amount as the DO WHILE.

Next, the stimulus-response pairs and a separate set of stimuli are arranged in a new random order for the current trial's study and test phases, respectively. The next two DO WHILEs control the study phase, during which the pairs are shown individually, and the recall phase, in which only the stimuli are shown and the subject's responses are input. Each of these DO WHILEs is followed by pause statements that insert intervals between study and test phases and trials.

Within the range of the DO WHILE for the test phase is the third kind of control structure, IF THEN ELSE. The first of three of these begins after the statement for response input and ends its control at the END IF directly below it, several statements down. At the beginning, it tests the response to determine if it is correct. If this condition is true (i.e., the response is wrong), the next set of statements down to the ELSE will be performed, and the set after the ELSE will be bypassed. On the other hand, if the response is correct, only the statements under the ELSE will be performed. Thus, the IF THEN ELSE structure allows selection of one of two alternatives depending on the truth of some condition. More alternatives can be introduced by nesting additional IFs within the first. This has been done here to allow feedback to be given if the experimenter has requested it. Within each alternative of the first IF is another that causes either "wrong" or "good" to be displayed to the subject if it has been indicated at the beginning of the program that feedback should be given. Neither IF has an accompanying ELSE since there is nothing to be done if no feedback is desired. In this case, the ELSE clause is null, which describes the kind of IF statement one finds in FORTRAN, for example.

The rest of the example consists of END statements for control structures, the intertrial interval pause already mentioned, and a statement for subject dismissal. The next two sections of this paper examine the hierarchical nature of a structured program and the control structures in more detail.

THE HIERARCHY OF MODULES

The hierarchy in a structured program should be apparent from the way that the control structures in the example are completely nested. Another device often used with SP that shows the hierarchy is a top-down chart, pictured in Figure 2 (Hughes & Michton, 1977; Miller & Lindamood, 1973).

The boxes in the chart represent modules in the

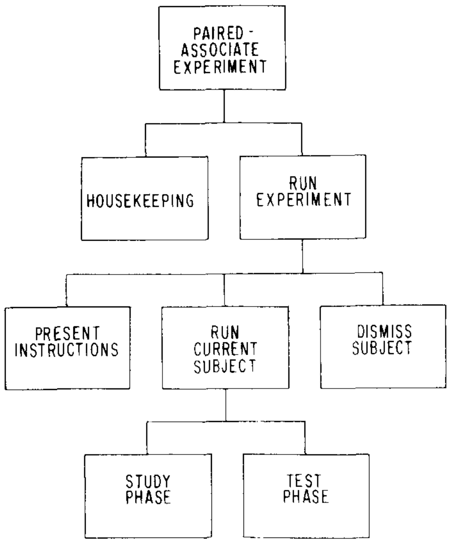

Figure 2. Top-down chart for the paired associate program in Figure 1.

program. Each module controls those immediately below it to which it is connected. Thus, the module "run experiment" has direct control over "instructions," "run current subject," and "dismiss subject." Similarly, "run current subject" controls "present study phase" and "present test phase." To say a module controls another means that it initiates the action of the other, and that the other module returns control to the first when it is finished. This is the only control path for the lower level module; there is no other way to perform its function except through the one higher level module that controls it. The single exception is a subroutine that may be called by more than one module. This has the same effect as several copies of the subroutine being in the program, one copy for each calling module.

The lines of control in a structured program, then, are restricted to the hierarchy. One does not get the common pattern of many programs where any module might be connected to any other if it seems convenient, a pattern that has often been called "a bowl of spaghetti" or "a rat's nest." This is primarily what is meant by saying that the modules are minimally dependent on each other in SP. In a nonstructured program, it is often difficult to tell when a particular section of code will be performed because there are several different paths to it, occurring under a different set of conditions. While there may be a number of conditions that cause a module to be performed in a structured program, the path to that module will be easier to trace because of the hierarchy.

The use of the term module in the literature is often vague and contradictory. In some cases, it means an entire program; in others, it may be restricted to a certain number of lines of code (e.g., a maximum of 50 or so that fit onto a single page of the program listing). Still other meanings are associated with it, such as the assignment given to a programmer working as a part of a system-development team.

One feature is common to all of these usages: A module performs a function. If a person can describe what a part of a program (or a whole program, or even a set of programs) does in one or two sentences, then he or she is probably talking about a module. The function can be very general or very specific; the same is true of a module. The modules in Figures 1 and 2 are fairly specific, more so than they usually are in a top-down chart. Although it is not usually done, there is no conceptual reason for not calling a single statement a module, with the exception of control structures that may be comprised of several statements.

There is a second feature of a module as described here that, while not restricted to SP, is an important part of the SP approach. A module may have only one entry point and one exit. This, of course, fits in with the hierarchy already described. An examination of the hierarchy shows that each module has a single line of control, which represents both the entry and exit points. Figures 1 and 2 indicate, for example, that the module for running a particular subject starts only at the second DO WHILE, which can be arrived at only after the instructions are presented. Similarly, there is just one way of getting out of the module: When the subject reaches criterion, control passes back to the module represented by the first DO WHILE through the single exit at the next-to-last END DO.

THE CONTROL STRUCTURES

The three basic control structures, the fourth introduced in the example program, and a fifth are illustrated in flowchart form in Figure 3. The circles are collector nodes and are used to emphasize the single entry or exit point for a structure. Also emphasized by

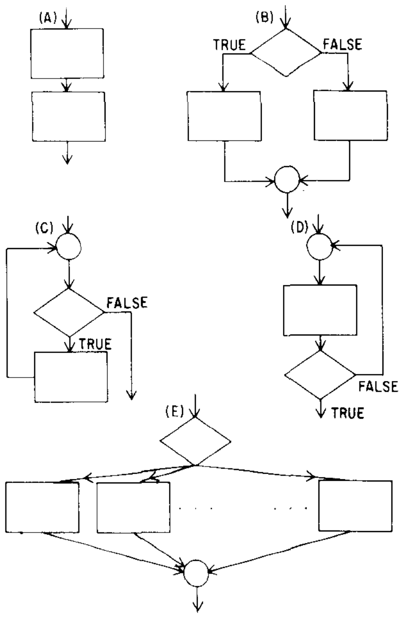

Figure 3. Control structures used in structured programming: (A) sequencing, (B) IF THEN ELSE, (C) DO WHILE, (D) DO UNTIL, (E) CASE.

the flowcharts is the fact that decisions are made and branches are taken in a structured program, even though there may be no statements that explicitly say GO TO or something equivalent. Thus, branches are not eliminated in a structured program but are restricted to those that implement the control structures.

The only structure in Figure 3 that has not been covered is the CASE structure. This is essentially an expansion of the IF THEN ELSE for the situation in which there are more than two possible courses of action; there are several different cases to be taken care of. The CASE structure, as well as the DO WHILE and other proposed additional structures, can be simulated with the proper combination of the three basic structures. The advantage of such additional structures is that one can avoid some fairly clumsy combinations of code. For example, one has to either nest a lot of IF THEN ELSEs or use a lot of switches to handle a large number of cases without using CASE or going through a long, inefficient series of separate IFs. However, it is also possible to build a new structure that is so complex that it defeats the purposes of SP.

But are there programs that cannot be written with these structures? That is, are there programs that require some form of stand-alone GO TO? The answer is no. Bohm and Jacopini (1966) and Mills (Note 2) have proven the theorem that any proper program can be written with just the three basic structures. Since a proper program has one entry and one exit and at least one path between them, and any functioning program can be written as a proper program, the theorem applies to all programs worth considering.

The major objective of the structure theorem, as it is called, is based on the hierarchical structure that results from using SP. Because the modules are highly independent of each other, it is possible to prove the correctness of a program (establish that it performs the appropriate function) by proving the correctness of each separate module. The alternative is to prove the correctness of the program as a whole. Those who remember trying to prove theorems in math or elsewhere can appreciate the enormity of this task, which is tantamount to proving a large set of complexly related theorems.

Proving program correctness is still a tedious and difficult exercise with a structured program. However, there is a parallel between going through a correctness proof and the design and desk-checking of a program. While the programmer may not attempt a rigorous proof, she or he can examine each module separately and be fairly well satisfied that, if each module seems to work properly, the program should also (Mills, Note 2). Parenthetically, it should be emphasized that for this process, as well as for correctness proofs, one should be careful to specify the states of all important variables as the module is entered.

ADVANTAGES AND DISADVANTAGES OF SP

The ability to check each module independently is an obvious advantage of SP. In general, the clarity and systematic nature of the lines of control and the independence of the modules are responsible for the superiority of SP over a more unorganized approach. It is much easier to tell when a module is being performed in a structured program, as there is only one way of getting into it and only one way of getting out, and both the entry and exit connect to the same higher level module. Thus, the logic is more easily followed both within and between modules.

The result is code that is much more likely to be free of logic bugs (i.e., more reliable code). In addition, once the programmer has learned to use SP, programs are written more quickly and with less pain. It is much less difficult, on the average, to rid the program of bugs after it is written. Changes to the program at a later date can be made more easily because fewer modules will be affected; often new modules, if necessary, can be directly inserted into the program with very little modification of existing code. This process is further facilitated by the ease of following the logic. Anyone who has programmed knows that a program that has been ignored for a while can appear to be almost unintelligible when first reexamined.

But SP also has its disadvantages. Two are its relative inefficiency in use of memory and speed of execution. If Module A is to be performed immediately after Module B under certain circumstances, a structured program may route the control path through one or more higher level modules rather than directly from A to B. Also, SP may, upon occasion, force a test to be made more often than might be the case in an unstructured program. The extra instructions, then, take more memory and more time. The latter factor may be a major reason for not employing a strict SP approach in time-critical applications. However, one should probably avoid SP in only those program sections that are time critical. A case could be made that attempts to make the entire program more efficient with direct GO TOs could make the program more inefficient overall, even though there might be improvements in relatively small segments of code.

While the potential inefficiency of SP is a major reason that opposition to it exists, there are other disadvantages that are greater, at least at present. One is the apparent lack of flexibility in SP. While any program can be written using SP, most programmers find this hard to believe and resist having the freedom to branch anywhere in a program taken away from them. A related factor is the emphasis in SP on the human element in programming rather than the logical. Most programmers, probably including psychologists who program, seem to think of themselves as strictly logical thinkers. It does not occur to them that human limitations can keep them from writing the maximally efficient, logically streamlined programs that may be theoretically possible but practically impossible.

Another disadvantage is the lack of statement forms in most popular languages that correspond to the SP structures. To perform an IF THEN ELSE in BASIC, one is forced to use GO TOs. This sort of simulation can produce code that is not much clearer, and probably less clear in some cases, than unstructured code. Nevertheless, the modularity advantage of SP is reason enough to use it in these languages. Also, the situation is improving: The new ANS FORTRAN 77 is an example, although it will likely be some time before this and other language advances will be available in the compilers being used by most people.

USING SP

How does one learn to use SP? First, an individual should read as much about SP and related techniques as possible. An especially good starting point is the book by Yourdon (1975). Second, if it is possible, one should work through a good text on writing structured programs in a specific language, even if the language is not likely to be used in the near future. Changing from the usual nonstructured programming style to SP is almost like learning to program all over again. A text on a language that incorporates the SP approach, such as PASCAL, is a good choice.

The problem of "unstructured" languages can be overcome by simulating the structures, as already noted, or by using a preprocessor or maverick compiler, if either is available, that includes statement forms for the structures. (A preprocessor runs before the compiler/assembler and translates the special statements into simulating code; IBM supplies one to allow SP in FORTRAN.) Since virtually all assembly languages have a macro capability, it should be possible to write macros to handle the structures. This should be done, however, by someone who is experienced in the language and in address determination schemes, since a method will have to be found for determining the exit and, in some cases, entry addresses.

CONCLUSION

While learning to use SP is not as easy as users would like, the benefits are well worth it. One advantage not noted earlier is the relative quickness with which programs can be written. The time gained here and from less debugging should more than compensate for the effort to learn SP. I recommend it for all but the most time-critical operations.

REFERENCE NOTES

1. Mills, H. D. *How to write correct programs and know it* (Report No. FSC 73-5008). Gaithersburg, Md: IBM Federal Systems Division, 1973.
2. Mills, H. D. *Mathematical foundations for structured programming* (Report No. FSC 72-6012). Gaithersburg, Md: IBM Federal Systems Division, 1972.

REFERENCES

Baker, F. T. Chief programmer team management of production programming. *IBM Systems Journal*, 1972, **9**, 366-371.

Baker, F. T., & Mills, H. D. Chief programmer teams. *Datamation*, December 1973, pp. 55-57.

Bohm, C., & Jacopini, G. Flow diagrams, Turning machines, and languages with only two formation rules. *Communications of the ACM*, 1966, **9**, 366-371.

Conway, R., & Gries, D. *An introduction to programming: A structured approach.* Cambridge, Mass: Winthrop, 1973.

Dahl, O. J., Dijkstra, E. W., & Hoare, C. A. R. *Structured programming.* New York: Academic Press, 1972.

Dijkstra, E. W. Programming considered as a human activity. *Proceedings of IFIP Congress 65.* Washington, D. C: Spartan Books, 1965.

Dijkstra, E. W. GOTO statement considered harmful. *Communications of the ACM*, 1968, **11**, 147-148; 538; 541.

Dijkstra, E. W. Structured programming. In P. Naur & B. Randell (Eds.), *Software engineering techniques.* Brussels: NATO Scientific Affairs Division, 1969.

Hughes, J. K., & Michton, J. I. *A structured approach to programming.* Englewood Cliffs, N. J: Prentice-Hall, 1977.

McCracken, D. D. Revolution in programming. *Datamation*, December, 1973. Pp. 50-52.

McGowan, C. L., III, & Kelly, J. R. *Top-down structured programming techniques.* New York: Petrocelli-Charter, 1975.

Miller, E. F., & Lindamood, G. E. Structured programming: Top-down approach. *Datamation*, December 1973, pp. 55-57.

Yourdon, E. *Techniques of program structure and design.* Englewood Cliffs, N.J: Prentice-Hall, 1975.