TUTORIAL SERIES 6

Structured programming is based on the stepwise refinement process—a method of problem decomposition common to all engineering disciplines and the physical, chemical, and biological sciences.

Structured Programming

Randall W. Jensen
Hughes Aircraft Company

During the past decade, the term "structured programming" has appeared innumerable times in the software literature—with nearly as many meanings. It has been defined as everything from "a return to common sense" to "the way our leading programmers program." Examples of typical definitions include:

> Structured programming theory deals with converting arbitrarily large and complex flowcharts into standard forms so that they can be represented by iterating and nesting a small number of basic and standard control logic structures.[1]

> Structured programming is a manner of organizing and coding programs that makes the programs easily understood and modified.[2]

> The fundamental concept [of structured programming] is a proof of correctness.[3]

> Structured programming is no panacea—it really consists of a formal notation for orderly thinking—an attribute not commonly inherent in programmers nor any other type.[4]

Although many of the definitions include a more than casual reference to the absence of GO TO statements in a structured program, other definitions place the GO TO statement in a more realistic perspective. For example, Mills has said that "structured programs should be characterized not simply by the absence of GO TO's, but by the presence of structure."[5] Wirth has provided the most accurate definition:

> "Structured programming" is the formulation of programs as hierarchical, nested structures of statements and objects of computation.[6]

By itself, Wirth's statement does not define the tools or logical structures used to achieve a program structure, nor does it limit the structure formulation techniques available to the program designer.

Structured programming can be understood as the application of a basic problem decomposition method to establish a manageable hierarchical problem structure. This method of problem decomposition is common to all engineering disciplines as well as to the physical, chemical, and biological sciences. The highest conceptual level represents the general description of the problem, with each lower level providing greater magnification or additional problem detail. The process of refinement is carried out in a series of steps, with each step bringing greater problem detail into focus or refining the knowledge available in the preceding step. Hence, the expression "stepwise refinement" suggested by Wirth[7] is often used to describe the abstracting or decomposition process that is fundamental to the definition of structured programming.

Objectives

The search for new programming methodologies can be attributed to a software crisis characterized by the increasing cost of producing software systems and the relatively poor quality/cost ratio of the completed systems. The increasing life-cycle cost of the software product is experienced in terms of a low productivity rate for the average programmer—sometimes as low as one or two lines of finished code per person-day—and a high proportion of total data-processing system costs attributable to the software subsystem.

Adapted by permission of Prentice-Hall, Inc., Englewood Cliffs, New Jersey, from "Structured Programming," Randall W. Jensen, in *Software Engineering,* Randall W. Jensen and Charles C. Tonies, © 1979.

0018-9162/81/0300-0031$00.75 © 1981 IEEE

Most of us have had to correct errors or modify programs that exhibited behavior better resembling Brownian motion than any kind of orderly flow. Attempts to read and understand such a program are frustrated by the continued skipping between program segments separated sometimes by several pages. Furthermore, incorporating a change is always accompanied with a deep fear that the modification will create a disaster in a remote part of the program.

Structured programming—improperly applied—is no better than traditional methods of program design.

We know that the primary productivity limitations are not errors in coding but errors in communication and thinking. We also know that the poor quality of the completed software product is simply another aspect of that problem. The large number of errors that continue to exist in a system after delivery is largely related to the system's complexity, which is sometimes artificial due to poor logical construction in the original design and the difficulty of isolating software faults by testing in an environment with a finite schedule and funds.

From our knowledge of the causes of the software crisis, we can conclude that the primary goals of structured programming must be

- to minimize the number of errors that occur during the development process;
- to minimize the effort required to correct errors in sections of code found to be deficient, upgrading sections when more reliable, functional, or efficient techniques are discovered; and
- to minimize the life-cycle costs of the software.

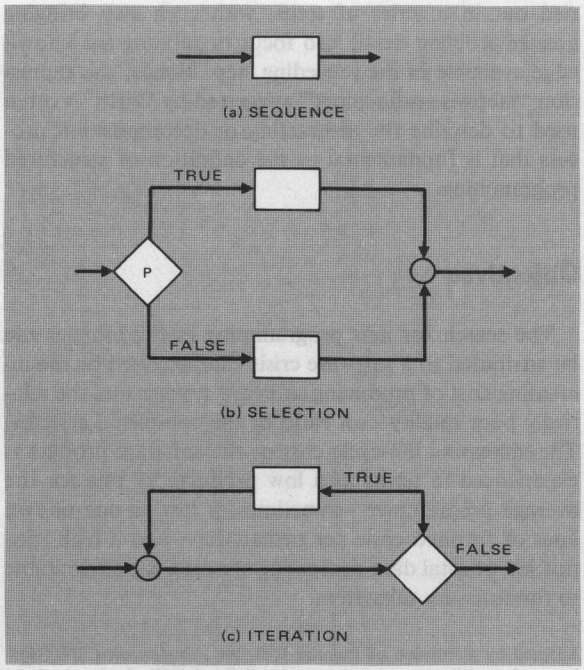


Figure 1. Three basic control structures.

In other words, we must reduce software complexity if we are going to attack the problems characterizing the crisis. Some of the benefits accompanying the reduction of program complexity are fewer testing problems, increased programmer productivity, and improved program clarity, maintainability, and modifiability.

Structured programming, however, is not a remedy for all of software's difficulties. Any assessment of its role in software development must also consider the particular problem areas where it can offer little or no improvement. There are two major areas where structured programming provides no solution. First, many errors in software arise from failures in the software specification. Low programmer productivity is largely due to time wasted in solving the wrong problem. In other words, structured programming cannot resolve communication failures caused by deficient specifications or a poor development organization. Second, the discipline and rigor of structured programming are beyond the abilities of a certain class of programmers. Structured programming—improperly applied—is no better than traditional methods of program design.

History and background

It is difficult to ascertain the earliest applications of the approach to programming that we now refer to as "structured programming." Over the years, attempting to produce cost-effective programs, many unknowingly approached but failed to achieve this style of programming.

Professor E.W. Dijkstra of the University of Eindhoven, Netherlands, was one of the first to formally recognize the value of structured programming. In 1965, he published a paper advocating the construction of programs in a structured manner.[8] His comments, and similar comments published about the same time, had little impact on the programming community for two reasons. First, most of the industry was struggling with early versions of Fortran and Cobol. Second, no positive approach had been proposed for solving the problems alluded to by Dijkstra and others.

The first major step toward structured programming was made in a paper published in 1966 by Böhm and Jacopini.[9] They demonstrated that three basic control structures were sufficient to express any flowchartable program logic. These basic constructs, shown in Figure 1, include a sequence mechanism, a selection mechanism, and an iteration mechanism.

As we will see later, these three basic structures do indeed form a sufficient, but not necessarily optimum, set of blocks for expressing any program flowchart. Thus, it is theoretically possible to write progams in languages like PL/I and Algol without using an explicit GO TO statement. When writing programs in Fortran, the use of the GO TO statement can be restricted theoretically to those instances necessary to emulate the three basic structures.

In 1968, Dijkstra forcefully repeated his ideas about program structure and the GO TO statement in a letter

published in the *Communications of the ACM* that began

> For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of GO TO statements in the programs they produce. More recently I discovered why the use of the GO TO statement has such disastrous effects, and I became convinced that the GO TO statement should be abolished from all "higher level" programming languages, (i.e., everything except, perhaps, plain machine code).[10]

Because of the sentiment that had built up against the GO TO statement in some circles, the reaction to Dijkstra's letter was immediate and strong. Hordes of programmers rushed to support Dijkstra's proposal with an enthusiasm akin to religious zeal. Structured and GO TO-less programming became synonymous in their eyes, and the objectives of structured progamming—readability, reliability, and programmer efficiency—were ignored in their effort to eliminate the GO TO. As an example of the present impact of this letter, I quote from a published interview with a prominent nameless computer science professor:

> In the earlier days we used to take off a full letter grade for each GO TO in the program. Now I simply state that an unstructured program is unacceptable.

The 25th ACM Annual Conference included a session entitled "The GO TO Controversy." Although W.A. Wulf advocated the elimination of the GO TO statement in "A Case Against the GOTO,"[11] he primarily attacked the misuse of the GO TO which resulted in programs with obscure logical structures. Similarly, M.E. Hopkins pointed out in "A Case For the GOTO"[12] that, although the GO TO statement was misused in many programs, it was still valuable in creating readable and maintainable programs.

Niklaus Wirth's 1971 paper[7] established the basic structured programming procedure. He suggested that program construction consists of a sequence of refinement steps. Each step of the construction consists of breaking a given task into a number of subtasks. This refinement in the program description should be accompanied by a parallel refinement in the data description, which constitutes the communication means between subtasks.

Later in 1971, F.T. Baker[13] demonstrated that there was considerable value in applying structured programming techniques in a production programming environment. The qualified success of the classic *New York Times* on-line information retrieval system project—in spite of the short delivery schedule and the magnitude of the effort—proved that structured programming was more than an academic exercise.

The next major driving force toward acceptance of structured programming techniques was the largest consumer of software products—the US government. Its increased awareness of the value of this new methodology in combating skyrocketing software costs led to a series of directives that "encouraged" the use of structured programming in all software procurements.

Since the end of 1973, the new technology has evolved at an ever-increasing rate. As a continuing stream of positive results is obtained, software developers and universities are striving to gain practical experience with structured program development. We can expect that future software development projects will continue to demonstrate greater programmer efficiency and program reliability by using these concepts. We can also expect to see considerably wider acceptance of structured programming in the software community as the methodology evolves further.

Theory and techniques

The heart of structured progamming is the "stepwise refinement" process. In addition, other subordinate concepts are important to the proper utilization of the program development methodology. We find ourselves, however, in a quandary at this point due to the difficulty of presenting these ideas in their order of importance. We cannot illustrate the refinement process without introducing the subordinate logical control structures, nor can we describe the necessary logical control structures without establishing some fundamental terminology.

The obvious solution is to present these topics in the reverse order—i.e., foundations, control structures, and, finally, the structuring process. By introducing the individual control structures before we introduce the process, we artificially and unwillingly emphasize the impor-

tance of the structures over the thinking or design approach. This artificial emphasis on individual structures is amplified by the general outward appearance of a structured program.

Two prominent characteristics attract our attention when we inspect a well-structured program—the presence of rather unique control structures and the nested indentation of program sequences. These two characteristics, frequently misconstrued as structured programming, are merely symptoms of the methodology; they are not structured programming. With this forceful note of caution, we will introduce the theory and techniques of structured programming as we proposed.

Since the intent of this section is to present the basic concepts of the methodology—and not the restrictions of any implementing language—we will avoid language details by using a pseudolanguage throughout the section. Our pseudolanguage is a combination of a hierarchical English dialect and the most convenient features of the existing high-order programming languages.

Foundations. A *flowchart* is a directed graph that describes the flow of execution control of the program.

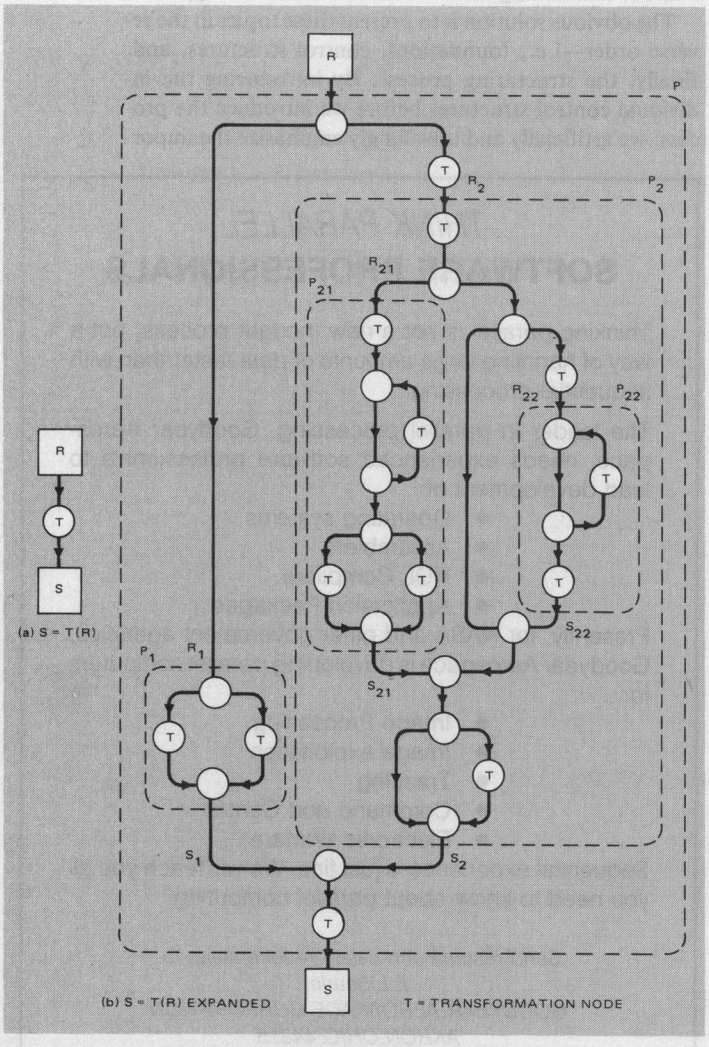


Figure 2. The flowchart, a directed program graph.

The graph is constructed of directed line segments whose orientation—or direction of flow—is indicated by arrows, as shown in Figure 2. The line segments originate and terminate in nodes, as shown in the same figure. A *node* may represent a data transformation, a decision point, or simply a collection point for the program control paths.

Using this information about the program flowchart, we can define a *well-formed program* as one in which there is precisely one input line, and in which there exists for every node a path from the program input line through that node to a program output line. This implies that there are no infinite loops and no unreachable code. In addition, a *proper program** contains precisely one output line. The directed graph in Figure 2 represents a proper program since there is exactly one input line R and one output line S, and a path exists through each node from R to S.

If a program contains a data transformation node T where $S = T(R)$, it may be possible to define another program that performs the function specified by node T. A program defined in this manner we call an *expansion* of the original program. A node that cannot be expanded in this manner (e.g., $A = B$) is called *nonexpansible*.

Considering the proper program shown in Figure 2(a), we find that we can amplify the original transformation to the flowchart shown in Figure 2(b) by successive expansions where the proper programs P_{21} and P_{22} are expansions of transformation nodes contained in expansion P_2 and, similarly, where P_1 and P_2 are expansions of nodes contained in proper program P. We might also say that programs P_1 and P_2 are contained in refinements of program P and that programs P_{21} and P_{22} are contained in refinements of program P_2. Each of the refinements is totally contained in its *abstraction*. Thus, P is an abstraction of P_1 and P_2, etc. *Total containment* implies that the refinements occurring within the individual nodes at any level are independent of each other—an extremely important idea in the development of the structured programming methodology.

We define a *proper structured program* as any program derived by a stepwise refinement process that is defined by using the three basic control structures shown in Figure 1. A *well-formed structured program* is derived by stepwise refinement that allows the exit control structure in addition to the three basic control structures.

The classic definitions were introduced in 1966 by Böhm and Jacopini.[9] They demonstrated that flowchart programs of any size and complexity can be represented by proper programs built from the small set of simple and regular programs of Figure 1. Based on the result of their work, a *structured program* was defined as any program whose flow of execution control can be described by using only the three basic control structures—sequence, selection, and iteration. Using this definition, it was possible to prove the Structure Theorem:

> Any proper program can be transformed into an equivalent "structured program" using the functions and deci-

*The word "proper" is used in this definition to be consistent with classic structured program definitions. By this definition, well-formed programs with multiple exit paths are improper—but only in terms of the definition.

sions of the original proper program and assignments and tests on one additional variable.[5]

The "one additional variable" mentioned in the theorem is normally a flag variable required to provide a means of handling abnormal exit conditions from the three basic control structures. By introducing a fourth basic control structure class that we refer to as the exit class, we essentially eliminate the need for the "one additional variable." Peterson, Kasami, and Tokura[14] demonstrated the value of the exit class of statements on the formation of well-formed structured programs.

An input program full of "garbage" will result in properly structured garbage after the transformations.

The Structure Theorem, as presented by Böhm and Jacopini, states that a structured program can be obtained by performing a series of elementary manipulations or transformations on the logic control structure of a proper, but unstructured, program. The classic structured program definition and the Structure Theorem might lead one to think that the way to write structured programs is to write unstructured code and then convert it to structured form. This is not the case. If a program has a logic structure that resembles a bowl of spaghetti, there is no transformation possible—short of redesign—that will significantly improve the program's clarity or maintainability. If the original structure is spaghetti-like, it is reasonable to assume that the quality of the transformed program will be a little better. That is, an input program full of "garbage" will result in properly structured garbage after the transformations.

Our structured programming definition emphasizes that the program must be obtained through the process of stepwise refinement. Indeed, a proper structured program achieved by the refinement process contains only the three basic control structures necessary to satisfy the weaker definition of a structured program associated with the Structure Theorem and an occasional exit structure. Development of the program logic structure by a stepwise refinement process not only arrives at a valid structure directly, but it also obviates the need for the Structure Theorem, since there is never a requirement to transform a proper program into an equivalent structured program.

Basic control structures. The Böhm and Jacopini paper established that three basic constructs or logic control structures were sufficient for expressing any problem logic. Nevertheless, their paper did not claim that the three control structures were the only structures that could be used, nor did it claim that these constructs were the best set of structures for any application. In this section, we describe nine control structures that comprise a useful set of tools for the construction of maintainable software. The set is divided into four basic structure classes:

(1) Sequence: concatenation

(2) Selection: if-else
if-orif-else (often described as if-elseif-else)
case
posit

(3) Iteration: while
until

(4) Exit: escape
cycle

Fundamental elements. The nine control structures can each be reduced to collections of three atomic components, which we refer to as nodes: a process node, a predicate node, and a collector node. The symbols for these components are shown in Figure 3.

The *process node* is associated with a data transformation in the program flowchart. The process can be as simple as an elementary assignment statement or as complex as an entire program segment. We define a *proper* process node as one that contains only a single entry point and a single unit.

By the same reasoning, a *well-formed* process node contains a single entry point and one or more exits due to the presence of an abnormal exit from the process. Note that a proper node is also well-formed. The well-formed process node shown in Figure 4 includes a variation of the predicate node to indicate the logical condition causing the abnormal exit from the process.

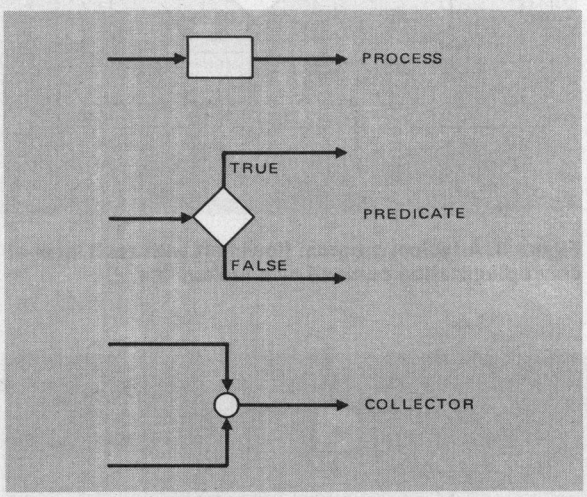


Figure 3. The three fundamental structure nodes.

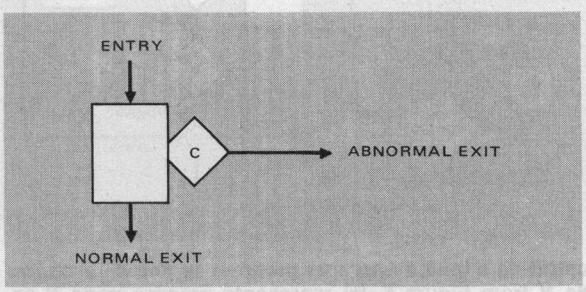


Figure 4. A well-formed process node.

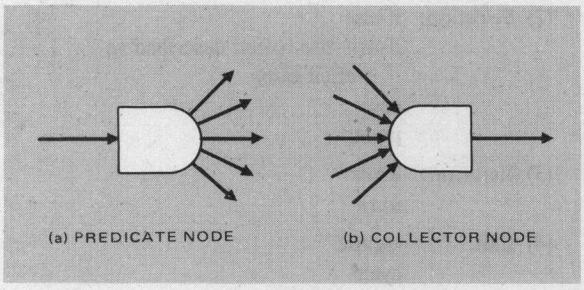


Figure 5. Special selection forms of predicate and collector nodes.

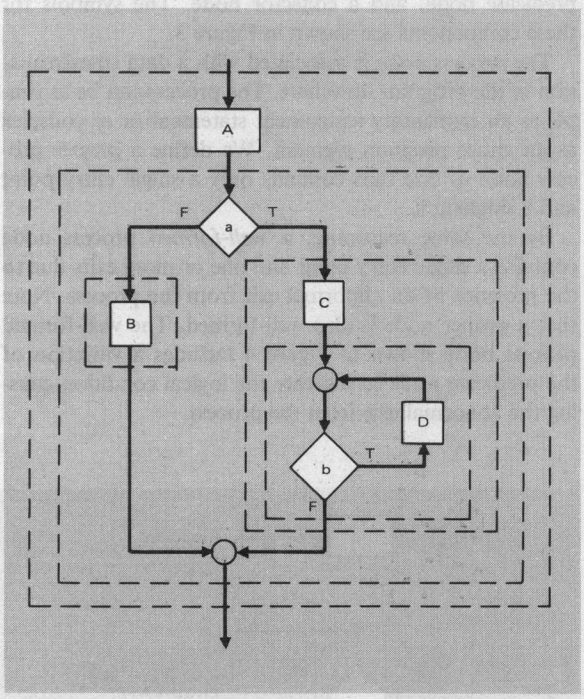


Figure 6. A typical program flowchart with each level of conceptualization outlined by a broken line.

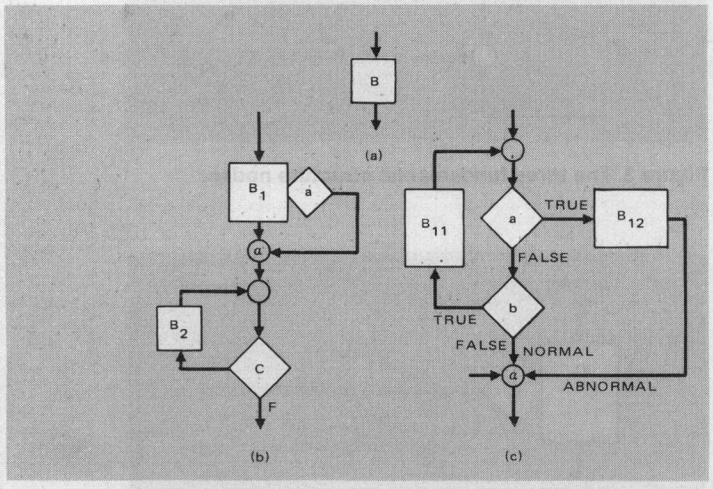


Figure 7. Illustrating a buried improper program segment, (a) shows process node *B* from Figure 6, (b) represents process node *B* expanded, and (c) illustrates the amplification of process B_1.

The process node, since it is characterized as a function with an input and an output, exhibits some of the properties of the classic engineering "black box." The black box characteristic is extremely important because it allows us to verify the correct performance of the process nodes at each level of abstraction until the entire program can be verified as a process node at the highest abstract level. In this process, the immediate lower-level process node is treated as a verified entity or component to reduce the verification complexity to a manageable magnitude. Well-formed, but improper, nodes increase the verification difficulty by introducing alternate paths in the process. However, if the exit structure is used judiciously, the increase is not of great significance.

The *predicate node*, or decision node, is associated with a predicate function. The predicate function is usually a binary-valued logical test that is evaluated each time the program execution enters the predicate node through its input path. Program data is not transformed by the predicate node; that is, no operations on data can be performed with this type of node. In the simplest case, the predicate might be "X is equal to Y" where a true or false evaluation is adequate. In a more complex case, we might ask, "What is the marital status of the subject?" to which the answer would be any one of this group: "single," "married," "divorced," or "widowed." For this case, we advise by necessity a special form of a predicate node that is controlled by a mutually exclusive selection of one of several output lines, rather than by a binary-valued logical test. The special selection form of the predicate node is shown in Figure 5(a). Each output line should be labeled with the condition causing that line to be selected.

The *collector node* combines a set of two or more program control flow lines (input) into a single output line. The number of control lines entering a collector node is two, except for the special cases where the collector node is associated with the special selection form of the predicate node, as shown in Figure 5(b), or where one or more of the input lines are related to abnormal process node exits.

The three atomic components can be assembled in any configuration to form the program control structure, such as the typical flowchart shown in Figure 6. The program representation in this figure appears to be proper—that is, the program at each level of conceptualization, as outlined by a broken line, represents a proper process node with a single entry point and a single exit.

One of the process nodes A, B, C, or D, however, may contain processes with multiple exit paths. For example, when process node B shown in Figure 7(a) is expanded, we discover that B contains a well-formed process node B, as shown in Figure 7(b). The amplification of process B_1 in Figure 7(c) shows that process node B_{12} represents processing required to correctly terminate process B_1 under a discovered exit condition. Thus, process B_{12} appears internal to process B_1, rather than externally as a separate process between predicate node A and collector node α in Figure 7(b).

Sequence. The sequence structure is simply the concatenation of two or more process nodes, as shown in

Figure 8(a). If the process nodes are proper—each with a single entry and a single exit—the linear sequence of nodes can be compressed or abstracted into a single process node. When one or more of the sequence of processes are improper, the sequence compresses into a process node of the type shown in Figure 8(b). In the refinement process, a single process node (A) can be replaced by a sequence of two or more process nodes ($A_1, A_2 \ldots A_n$) which is an expansion of that node. The expansion process can be repeated as additional process detail is obtained.

Selection. The basic selection structure is the if-else binary decision mechanism shown in Figure 9. This control structure provides a choice between two alternatives which can be written as

```
if (p)
        Process A
else
        Process B
end if
```

where process A is executed if predicate p is true and process B is executed otherwise.

Each of the processes may contain complex logical structures. For example, consider the algorithm for selecting the largest of three values A, B, and C. The flowchart for the algorithm we refer to as MAX3 is shown in Figure 10.

The MAX3 algorithm is written as

```
if (a ≥ b)
    if (a ≥c)
        big = a
    else
        big = c
    end if
else
    if (b ≥c)
        big = b
    else
        big = c
    end if
end if
```

to correspond to the structure specified in the flowchart.

Here we recognize the value of indentation in clearly presenting the program structure. Indenting the source listing to typographically indicate the depth of nesting of iteration and selection structures effectively emphasizes structure, improving the clarity of the program listing.

Three special forms of the selection structure occur frequently in program designs: if-orif-else, case, and posit. Requirements for the first two are often related to a "transaction center" derived during decomposition of the program into functions, but the requirement can occur due to a multitude of other reasons. A transaction center transfers control to one—and only one—of a set of mutually exclusive processes in response to information input to the center.

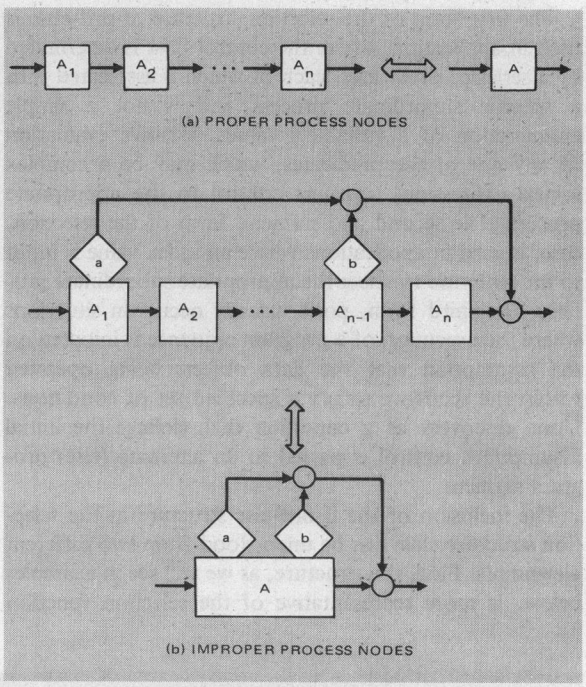


Figure 8. Concatenation of process nodes.

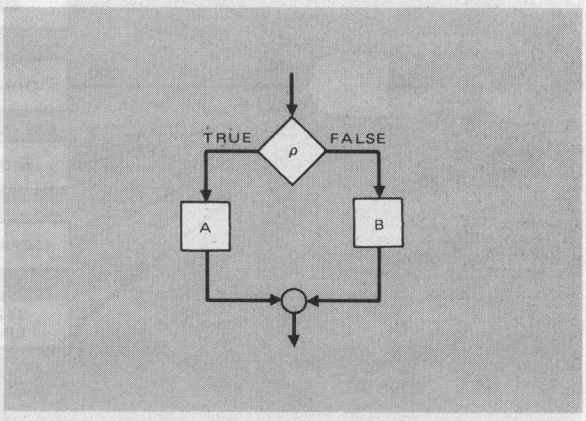


Figure 9. The if-else control structure.

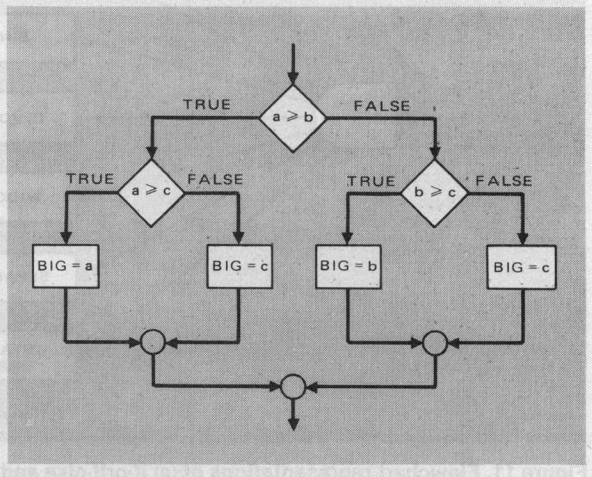


Figure 10. The flowchart for the MAX3 algorithm.

The first form of the selection structure, if-orif-else, is used in applications where the control flow is determined by a series of predicates, each of which is associated with a specific subordinate process and is not a simple enumeration of a variable's values. Positive evaluation of any one of the predicates, which may be a complex logical expression, transfers control to the appropriate process. The second and numeric form of the selection, case, is used in applications where an index value is input to the structure to select the appropriate subordinate process. The third form, posit, usually occurs in situations where the execution of a program sequence is initiated on the assumption that the data objects being operated within the sequence satisfy a specified set of conditions. Upon discovery of a condition that violates the initial assumption, control is passed to an alternate (else) program segment.

The inclusion of the if-orif-else structure in the selection structure class can be understood from two different viewpoints. First, the structure, as we will see in examples below, is more representative of the selection function being performed and possesses greater clarity than an equivalent implementation using nested if-else structures. Let us consider an application in which we select one of several processes according to an alphanumeric marital-status tag read from input data. The necessary structure for this selection is

```
if (status = 'married')
   Process married
orif (status = 'single')
   Process single
orif (status = 'divorced')
   Process divorced
orif (status = 'widowed')
   Process widowed
orif (status = 'separated')
   Process separated
else
   Process status error
end if
```

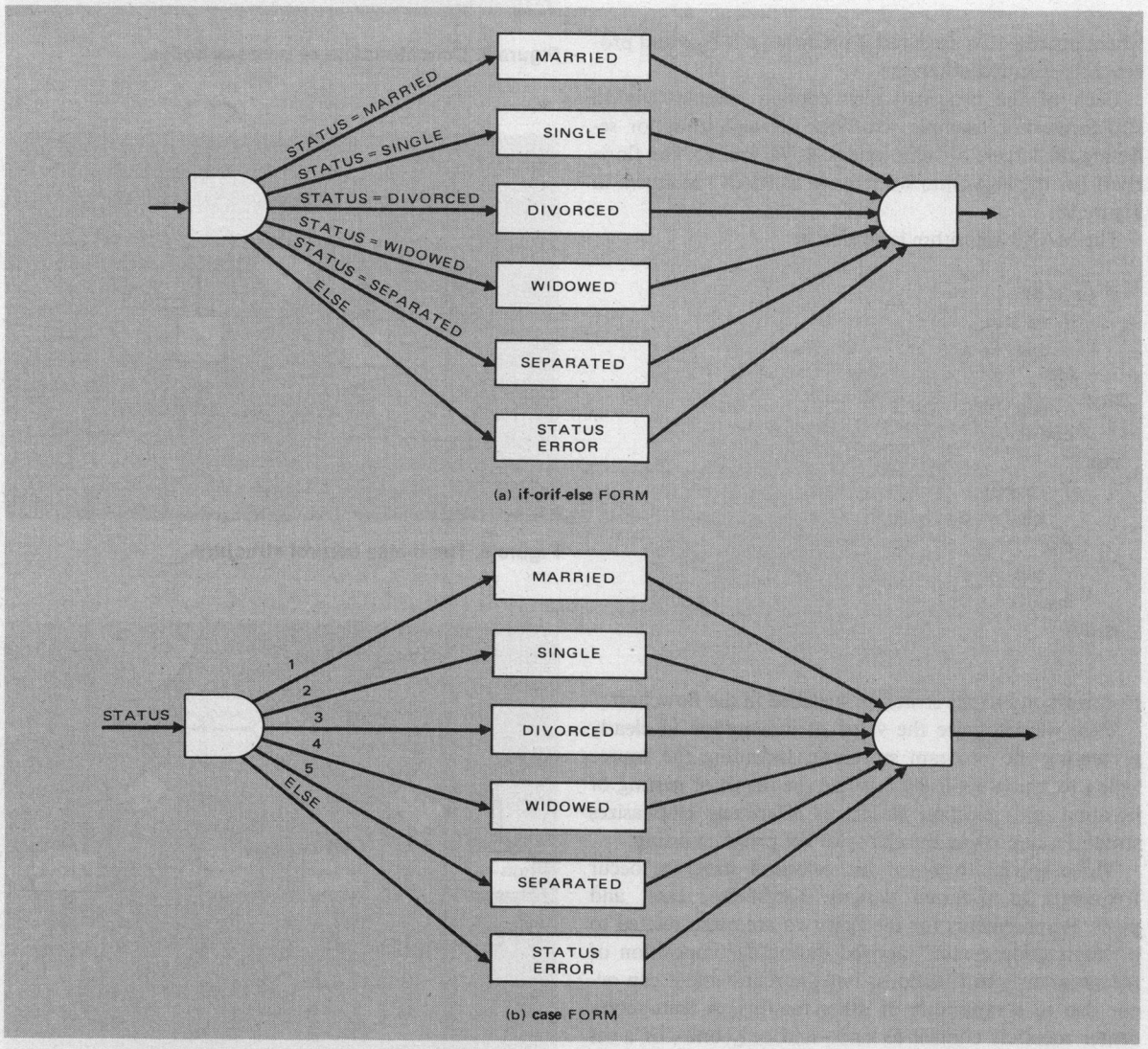


Figure 11. Flowchart representations of (a) if-orif-else and (b) case logical control structures for marital-status example.

The second justification for the if-orif-else structure is based on the human ability to interpret the meaning of code nested several levels deep, as in the if-else version of the marital status selection scheme. Psychological studies have shown that few people can grasp the overall meaning of a nested logic structure that is more than three levels deep. Beyond that depth, the apparent structural complexity increases the difficulty both in creating and maintaining the program.

In addition to the comprehension problem, we have to contend with the physical limitations of our programming tools. The 80-column input standard for source code allows only 66 columns for input after we eliminate eight columns for sequence information and another six columns for labeling and continuation information. If we assume an indentation standard of three columns and at some point in our program find a nesting depth of nine levels, we have only 39 columns available for program input. If we manage to create a selection structure involving 25 processes, the nested if-else implementation of the structure will not allow us to input the source program in indented form.

The flowchart representation of the marital status structure shown in Figure 11(a) uses the single transaction variable status to select the appropriate subordinate process. One could argue that this transaction could have been implemented nearly as easily with a case structure. That is true for this simple example. However, consider the selection problem if status equals "married" when married or separated for less than seven months and equals "separated" only when separated for seven or more months. The more complex selection criteria emphasize the need for the if-orif-else structure in addition to the case structure.

The integer form of the special selection structure, case, selects one of a set of processes for execution based on the value of an integer index. Let us consider again, for a moment, the problem of selecting a process to be executed based on the marital status of the subject. The case structure requires that the marital status tag be supplied in numeric form where the code supplied by the input data will represent the process. The case form of the selection structure is then

```
case of (status)
case (1)
     Process married
case (2)
     Process single
case (3)
     Process divorced
case (4)
     Process widowed
case (5)
     Process separated
else
     Process status error
end case
```

The flowchart representation of the marital status structure using the case construct is shown in Figure 11(b).

The third form of the selection structure, posit,[15] is required because of limitations imposed by the simple if-else structure. We have demonstrated the need for the if-orif-else and case structures on the basis that the structures are more representative of the specific selection process. The nesting in the "if-else" representation of the if-orif-else and case structures always occurs within the "else" clause of the construct. We must also allow for situations in which the nesting occurs within the "if" clause, when we have a condition somewhat analogous to a logical inverse of the selection process. Instead of entering the structure and executing only one selected process, we execute all processes or exit prematurely if a given logical condition within the structure is satisfied. The posit structure is written

```
posit
  Process A
else
  Process B
end posit
```

which is interpreted as "Perform A as long as the conditions assumed (posited) upon entry to the structure are satisfied. If the conditions are violated, perform B."

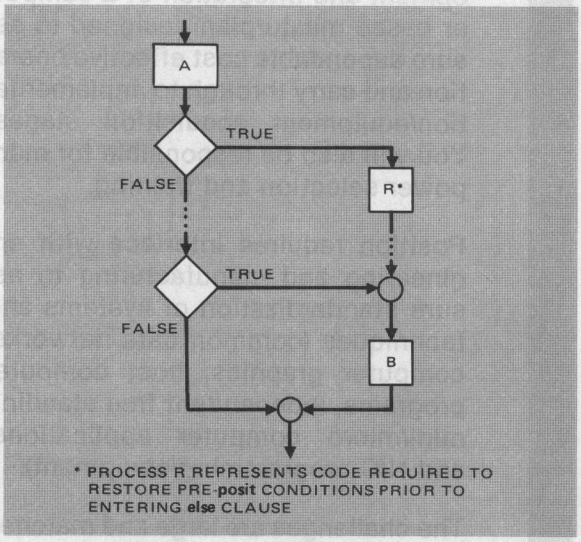


Figure 12. The posit-else logical control structure.

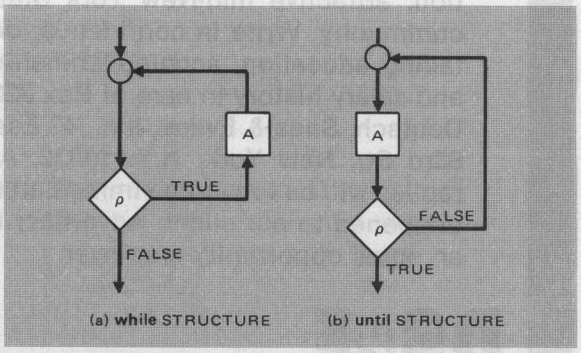


Figure 13. Two forms of the basic iteration control structure.

For example, consider a problem in which we are to execute a sequence of logical processes, $P_0(f_0)$, $P_1(f_1)$, $P_2(f_2)$, $P_3(f_3)\ldots$, where $f_n = P_{n-1}(f_{n-1})$. If the result of any one of the processes is false, we must execute an error process P_x. If process P_i fails, a specified set of conditions must be restored before executing P_x. The solution can be represented by the posit structure as

```
posit
      Process P0
      quit posit if (f̄1)
      Process P1
      if (f̄2)
         Process Restore
         quit posit
      end if
      Process P2
      quit posit if (f̄3)
      Process P3
else
      Process Px
end posit
```

We introduced the concept of an exit (quit) mechanism to complete this special form of the selection structure. In this instance, the quit provides the limited range branching capability necessary to implement the structure. In addition to improving the format of the program, the posit-else structure allows us to include only one copy of process P_x in the source instead of n copies as required by the equivalent nested-if version.

The posit-else structure, schematically represented in Figure 12, is an important tool in the implementation of the "backtracking"[16] concept in program design. The sequence of processes P_i that we used as an example in introducing the posit structure can be construed as an example of the backtracking concept. We initially assumed that sequence P was valid upon entry to the posit. At certain points in the sequence, we tested the validity of our assumption and, as long as the assumption was valid, we continued the execution of the sequence. If the assumption proved incorrect, we backtracked to the alternate block (else), which contained the sequence to be executed if sequence P was invalid. In many situations, it is necessary to "restore" conditions or variables to a state that existed prior to entry into the posit.

Iteration. The collection of structured programming tools must contain at least one mechanism to provide a basic looping capability or a controlled repetition of portions of program code. This basic iteration structure can be one of two types, referred to as the *while* and *until* forms. Both forms are shown in Figure 13.

The most widely used iteration form is the while structure. It specifies that process A is to be executed repetitively as long as the controlling predicate p is true. The structure can be represented in source code in several ways. The most obvious way, of course, is

```
while (p)
      Process A
end while
```

and is read "while p is true, execute repetitively process A."

The until form of the basic iteration structure shown in Figure 13(b) is written as

```
until (p)
    Process A
end until
```

The representation of the until control structure is fundamentally different in two ways from the while form of the iteration structure. First, the until form tests the predicate *after* each execution of the process rather than before, as in the while structure. Thus, an until loop will always be performed at least once, regardless of the value of predicate p. Second, the until iteration terminates when the predicate value is true, whereas the while loop terminates when the value is false.

The program defined by the codes sequence

```
until (p)
    Process A
end until
```

using the until structure, is equivalent to the program

```
Process A
while (p̄)
    Process A
end while
```

utilizng the while form of the iteration structure.

Exit. The fourth class of essential control structures has been cloaked in controversy ever since Dijkstra's famous GO TO letter.[10] There are too many GO TO statements in many programs—programs which appear more complex than they really are—but this does not justify the elimination of the GO TO from our programming languages. The absence or rare occurrence of GO TO statements is no more than a symptom of structured programming.

Before we ban the GO TO statement, we should consider three important points concerning the unconditional branch and its relationship to structured programming. First, the effect of the GO TO statement on program complexity is depicted in Figure 14. The figure is a gross simplification—the scales on the axes are subjective and the surface should be somewhat lumpy—but the principle is still valid. The minimum program complexity for a simple problem occurs with zero GO TO statements. As the complexity of the problem to be solved increases, attempts to eliminate all GO TO statements artificially increase the program's complexity by adding unnecessary program code and additional logical tests in the program control structure predicates. Sometimes it is necessary to use an unsightly GO TO statement to minimize the complexity and enhance the maintainability of the program.

The investigation by Peterson, Kasami, and Tokura[14] demonstrated that a program could increase in length and execution time if its control logic was limited to the three basic constructs—sequence, if-else, and while. They also demonstrated that if we allow a program to increase in size while retaining the execution speed of its unstructured equivalent program, the sequence, selection, and iteration structures must be supplemented by a multiple-level exit structure.

The GO TO statement can also be used as a primitive to construct missing or more advanced and elegant control structures in a programming language. For example, the case control structure is not available in the major high-order programming languages—Fortran, Cobol, and some implementations of PL/I. As the software development methodologies evolve, creative programmers will develop new, powerful control structures, such as the case and posit structures. These new structures will be difficult to implement without the GO TO primitive.

There are two important well-formed program characteristics that we must emphasize:

- by using the refinement process to develop the program control structure, the unconditional branch statement rarely occurs (some of the program conditions requiring the branch statement are enumerated by Knuth[17]), and
- a program block is limited to precisely one input line.

Thus, a sequence of code can be entered only through the path connecting it to a preceding program block, as shown in Figure 15(a). Unconditional branches are then limited to paths leading to the end of the program block. Exit paths to any other part of the control structure will create additional input lines to a program block. The abnormal exit in Figure 15(b) is illegally reentering the previous program block in violation of the well-formed program requirement.

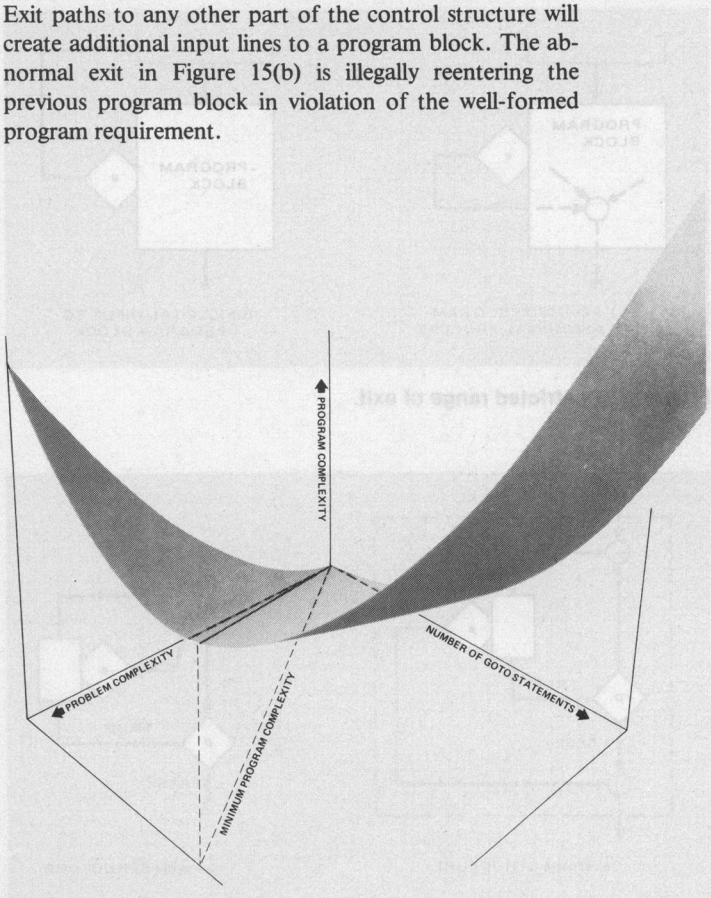


Figure 14. Effect of the GO TO statement on program complexity.

We will define two structures in the exit class—escape and cycle. The *escape* structure is an unconditional branch to the end of the associated structure. If the exit is from an iteration structure, the branch is to the outside of the iteration loop, as shown in Figure 16(a). Conversely, the *cycle* structure is an unconditional branch to the predicate controlling the next iteration or to the inside of the iteration loop, as shown in Figure 16(b).

Program development tools

Of the numerous development techniques or tools available, three can be used effectively to describe a program's logic during the design or refinement process at the module level. These techniques are flowcharts, pseudocode, and processing logic trees. In the discussion that follows, we will emphasize the processing logic tree as the most effective technique and say little about flowcharts, the weakest of the three techniques.

Pseudocode. A notation that bridges the gap between the programmer's native language and the computer's language, pseudocode is frequently referred to as "structured English" because of its apparent combination of basic elements from both structured programming and the English language. It is a language that allows the programmer to think about the problem to be solved, and it expresses the program's logic in a somewhat formalized way, without concern for the syntax of any specific programming language. The notation allows the programmer to deal with the problems at various levels of abstraction without being concerned about program details—much like the flowchart, which can also depict the functional properties of a program without complicating the program representation with unnecessary details. Pseudocode resembles a programming language in that explicit operations can be specified, such as

```
Set initial count = 1
Print range rate error
```

It differs from a programming language in two ways. First, operations can be specified at any level of complexity. For example,

```
Temporary = (a(i, n + 1) − sum)/a(i,i)
Compute target rate of closure
```

Second, there are no formal syntactical rules to limit freedom of use. The only conventions governing the use of pseudocode are related to the use of the control structures and the indentation that improves clarity.

Pseudocode is also a language that can be used during the refinement process to describe the program's operation at each level of refinement. This provides a convenient way for a programmer to document the stages of program development for his own use, allows other programmers to review the function of the program before it is translated into a programming language, and facilitates an assessment of the development status at any stage of the refinement process. Since pseudocode allows the designer to work with a language at a level higher than a programming language, it is easier to implement design changes in the program. The changes in the program logic are more apparent at the pseudocode level than when the logic is complicated by the details of the language at machine level. The pseudocode notation also provides a detailed description of the completed source program that can be, or should be, maintained as part of the program documentation as a substitute for flowcharts.

Pseudocode has no formal guidelines to direct the user in his application of the tool. The tool is natural and should be used in any way that makes sense to the programmer. To make the use of pseudocode more effective, a few general guidelines are available.

Pseudocode is an extension of the thinking process.

The statements used should be related to the stepwise refinement process and result in a structured program. To guide the thinking process and assist in creating an acceptable structure, the use of control structures not allowed on a given project should be discouraged.

Indent code to highlight logic structure.

Use self-defining data names.

Keep the program logic simple.

Code first for clarity, then modify for efficiency. As Knuth aptly stated,

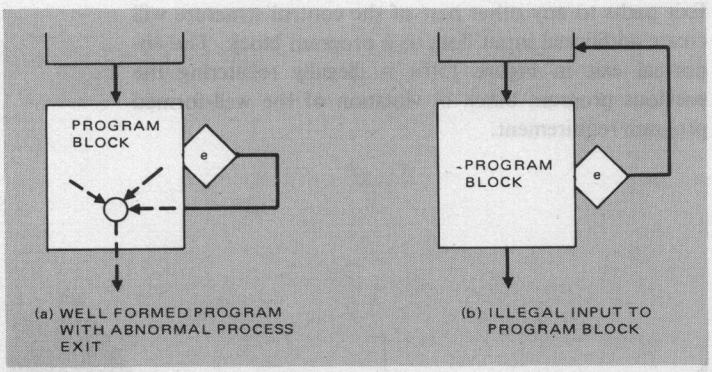


Figure 15. Restricted range of exit.

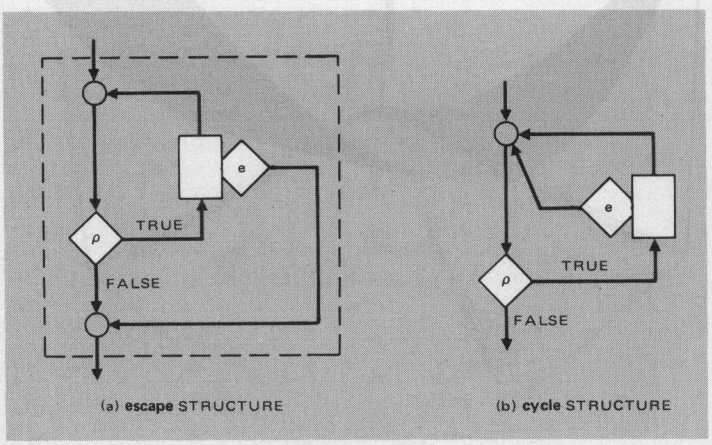


Figure 16. Two structures of exit.

> Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.
>
> Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he will be wise to look carefully at the critical code; but only *after* that code has been identified.[17]

Use control structures in the form allowed by the project.

If a specified structure is of the form if-else-endif, use the structure in that form. For example,

```
if (malfunction switch = off)
  Calculate mean smog density for 24-hour period
else
  Set mean smog density = 1000 ppm
endif
```

Each statement in a sequence should be written on a separate line, since a statement may be expanded in a later step of the refinement process. If multiple operations are allowed on a single line, chaos and confusion will result.

Processing logic trees. The third notation for describing a module's logic structure is called the processing logic tree, often abbreviated as PLT. This method provides a clear graphic representation of the program structure and a notation amenable to the stepwise refinement process. A predecessor of the notation was introduced by Jackson[16] as a means for describing the data structure and the structure of the associated program, which processes the data. A modification of Jackson's original notation provides us with a unique and powerful stepwise refinement design tool.

The processing logic tree is a design notation, much like pseudocode, that allows the programmer to think in terms of the problem to be solved in a quasi-formal way, without having to deal with the syntax of a programming language. It provides a graphic representation of the various stages of the solution, as does a functional flowchart. However, the PLT (pronounced "plit") allows for expansion in the level of detail without destroying any of the abstract information describing the program and without requiring partial or total reconstruction of the program representation. Thus, it can present a history of the development though the levels of conceptualization and provide a useful analysis aid in evaluating design trade-offs at any level. The PLT also forces a structured design process by prohibiting all but a hierarchical approach to the problem solution.

Each node of a PLT represents the execution of a specified task (or process). The process can be as trivial as a simple assignment statement (Pi = 3.14159) or as complex a function as "Formulate 30-day weather forecast." The degree of complexity is a function of the abstraction level, or tree level, at which the task is defined. The most elemental means of graphically representing a process is by means of a "box" containing a textual description of the process.

The PLT notation uses several process box configurations to represent specific process types. The following discussion describes the limited set of control structure representations provided within the notation and the rules associated with the usage of those structures.

Sequence structure. The most common control logic structure is the simple sequence structure shown in Figure 17. This structure represents the sequence of operations $B, C, \ldots, n$. Node A (or process A) represents collectively the sequence of operations $\{B, C, \ldots, n\}$. We can state this as a simple and fundamental PLT rule:

> Rule 1: The processing performed by a node in a processing logic tree (PLT) is completely defined by its subtrees (or subplts).

A corollary of Rule 1 states:

> Corollary 1: The processing of a node consists of processing all subtrees (subplts) of that node.

Thus, the sequence of operations $\{B, C, \ldots, n\}$ cannot imply operations performed by node A that are not present in the sequence.

Another rule that should be apparent at this point is:

> Rule 2: Offspring of any given parent are processed sequentially from left to right. The processing of each sibling must be complete before the processing of the next sibling can begin.

This rule is inherent in the definition of an ordered tree. As we will observe later, processing any parent node includes processing all of its offspring, all of the offspring's offspring, etc.

As the PLT is expanded through successive levels, the terminal nodes—or "leaves"—of the tree will contain elementary processes which cannot be expanded meaningfully into greater detail. Obviously, the expansion along these branches is complete.

In a practical sense, a program segment or routine should never be allowed to exceed a reasonable, manageable size. Some rigid guidelines have been proposed which limit the length of a program segment to only 30 to

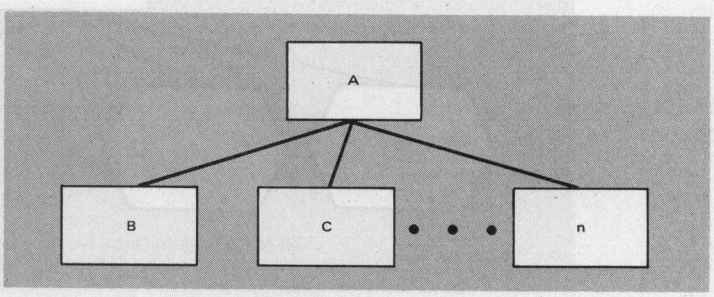


Figure 17. The basic PLT sequence structure.

100 lines of code. We will extricate ourselves from this controversy by acknowledging that routines should be limited to a manageable length and should contain no more than one processing function. It may become apparent that a particular node or process should become a subroutine as the process is being defined, or the need for a subroutine may not become apparent until the PLT has already become large. In the first instance, the node can become a terminal node representing a subroutine call by drawing a horizontal line through the lower portion of the process box and naming the subroutine. The terminal node serves as the subroutine reference (or call) in the referencing PLT. An individual subroutine can be referenced from several terminal nodes. The processing logic tree for the subroutine is independent of the referencing PLT. To summarize the use of subroutines, we introduce a third rule:

Rule 3: Subroutines are always defined by root nodes, a main program being a special subroutine form.

We can simplify an unwieldy program or PLT by isolating portions of the program in the form of subrou-

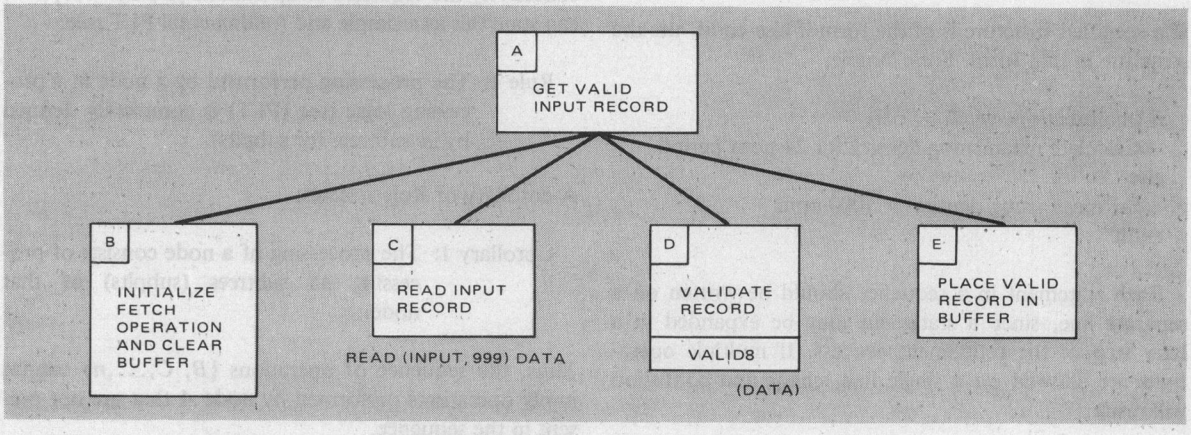


Figure 18. Processing logic tree of a sequence showing subroutine call definition.

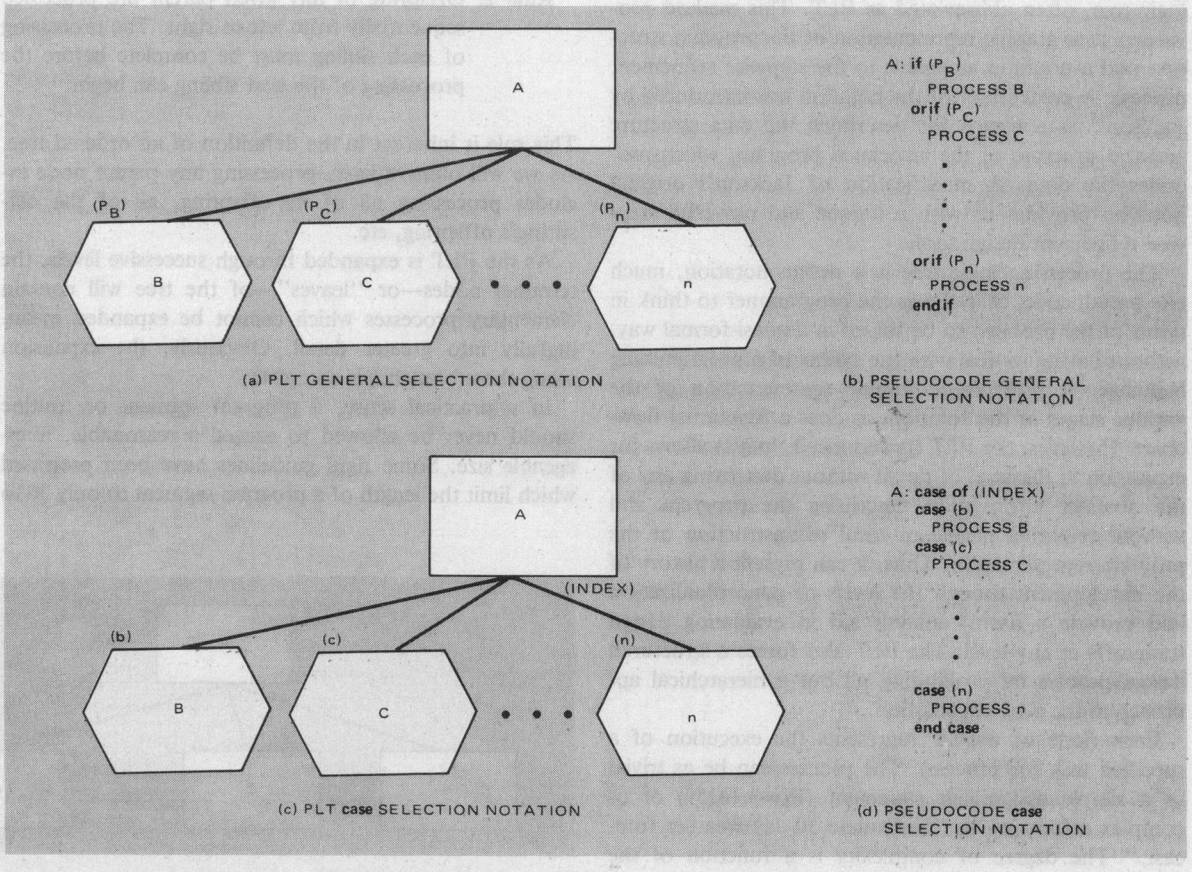


Figure 19. The basic PLT selection structure forms.

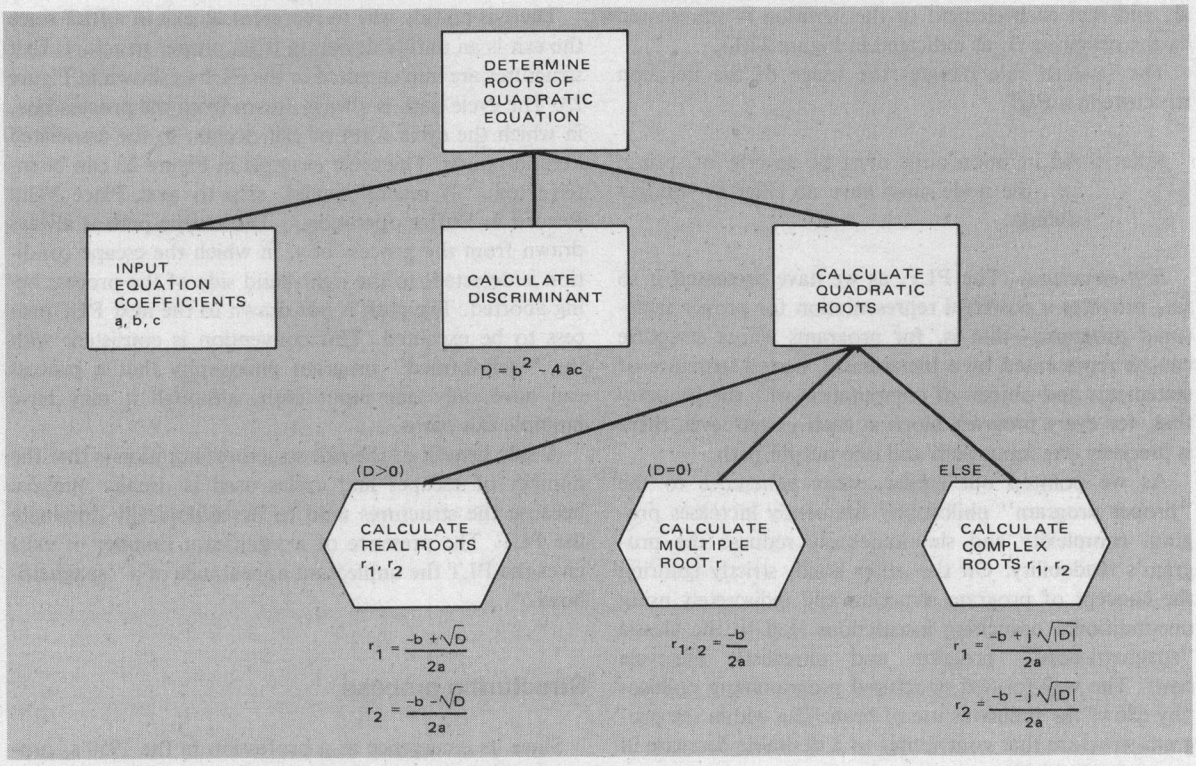


Figure 20. PLT structure to obtain roots of the quadratic equation.

tines performing specific functions. The nature of the PLT dictates that the branch connecting an offspring to a parent isolates a functional entity. The sequence structure in Figure 18 represents separate, isolatable processes. According to Rule 4, we can define a functional subroutine by drawing an arc through any one of the structure's branches, as shown in Figure 18.

Rule 4: Subroutines may be formed by drawing an arc through a single branch and replacing the subplt by a terminal node representing the subroutine call. The root of the subplt defines the subroutines.

Selection structure. The selection structure provides the capability to conditionally perform tasks based upon predetermined criteria. The general form of the selection structure in PLT notation and equivalent pseudocode description is shown in Figure 19. Each offspring in the structure is represented by an irregular six-sided polygon, as indicated by the elements labeled $B, C, \ldots, n$.

The processes that comprise function A in the selection structure are mutually exclusive—that is, only one of the processes subordinate to function A will be executed. The logic condition (or predicate) causing a specific function to be selected (selection criteria) in the general selection form is defined by a fifth rule:

Rule 5: The selection criteria for the offspring of any given parent in a selection structure is examined sequentially from left to right. Processing of the parent node is equivalent to processing either the first offspring, satisfying the selection criteria, or no offspring if none satisfy the selection criteria.

For example, we can expand the functional process, Determine Roots of Quadratic Equation, into the PLT shown in Figure 20.

PLT Rule 6 contains the evolution of a software design and the PLT to one level of refinement at a time. The rule can be stated:

Rule 6: All first-generation offspring of a parent must be of the same generic type.

In other words, it is not only poor practice, but it is also illegal to assign offspring of different types (sequence, selection, etc.) to a given parent.

The posit is a special form of the selection structure, as shown in Figure 21. This form always consists of two offspring labeled "posit" and "else" to completely define the structure. Process B, controlled by the posit selection criteria, is always executed upon entry into process A; the else-controlled process C is executed upon failure of the conditions assumed on entry into process A. An abnormal exit from process B to process C via the quit path is taken upon the conditional failure. This form is not an exception to Rule 5 since the posit condition is defined to be initially true.

Iteration structure. The iteration structure provides a compact representation of an iterated process, as illustrated in Figure 22. The oval-shaped box in Figure 22(a) represents the iterated process. It is important to note that the iteration as a group is represented by the parent

A, and that each element of the iteration is represented by the offspring B, as indicated in Figure 22(b).

The seventh rule dictates the usage of the iteration structure in a PLT:

> Rule 7: An iteration node must be an only offspring; i.e., the node must have no elder or younger siblings.

Exit structures. The PLT, as we have presented it so far, provides a powerful representation for proper structured programs—that is, for programs whose structure can be represented by a hierarchical, nested structure of statements and objects of computation with the property that, for every program block at each nested level, there is precisely one input path and one output path.

As we pointed out earlier, strict adherence to the "proper program" philosophy frequently increases program complexity and simultaneously reduces the program's readability. On the other hand, strictly ignoring the concept of program structure and indiscretely using unconditional branching instructions lead to the classic "spaghetti-bowl" program and unrealistic life-cycle costs. The well-formed structured programming philosophy allows the judicious use of branching within the program structure that contributes to a desirable decrease in program complexity.

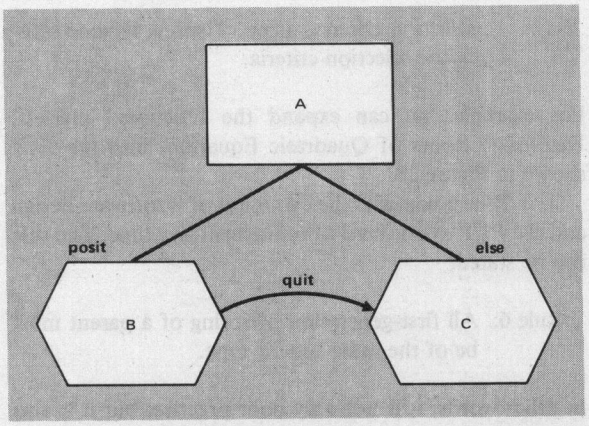


Figure 21. The general posit form of the selection structure.

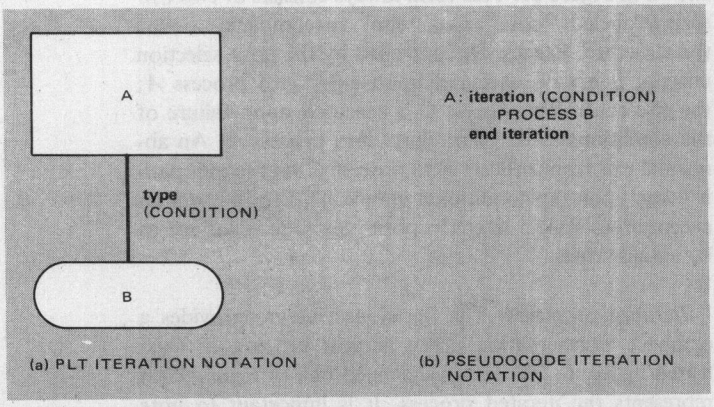


Figure 22. The basic PLT iteration structure.

There is no tidy way to represent an exit in a PLT since the exit is an untidy deviation from proper structure. Exit structures are implemented in the PLT as shown in Figure 23. The cycle path is always drawn from the process box, in which the cycle form of exit occurs, to the associated iteration node. The cycle example in Figure 23 can be interpreted: "If record invalid, skip to next Place Valid Record in Buffer operation." The escape path is always drawn from the process box, in which the escape condition is detected, to the right-hand side of the process being aborted. The path is *not* drawn to the next PLT process to be executed. This convention is consistent with the "well-formed" program philosophy that a process can have only one input path, although it may have multiple exit paths.

A side benefit of the exit structures notation is that the number of escapes and cycles used is usually minimal because the structures tend to psychologically dominate the PLT. The presence of a significant number of exits gives the PLT the unpleasant appearance of a "spaghetti-bowl."

Structuring process

Since its emergence as a profession in the 1950's, programming has been considered an art, mainly because of a lack of definition or understanding of the design methods used by programmers. Although many excellent programs have been completed during the intervening years, the methods by which they were developed remain a mystery to the observer. The program developer is frequently at a loss for words when asked to explain his programming methods. Programming is still, to a great extent, an art applied by craftsmen of the trade.

Until recently, very little had been written on the process of designing and implementing program code. Furthermore, training in the "science" of programming had been almost nonexistent. Consider for a moment the training you received in elementary programming. We can assume with relatively high confidence that the training consisted of a few brief lectures on the syntax of a specific programming language, supported by either a manufacturer's reference manual or an elementary text covering the language's syntax, with a few coding examples and homework problems. Upon successful (?) completion of the homework assignments and one or more tests on the language's syntax, a diploma was awarded, certifying mastery of the "art of programming." (Just like the scarecrow in the Land of Oz.)

The work of Dijkstra, Wirth, Weinberg, and several others in recent years has contributed much to the realization that the most effective programming methodology is based on the human method of mentally analyzing and solving problems.

The mental tool fundamental to effective program development is common to all forms of engineering. This tool for coping with complex problems is *abstraction*. A complex problem cannot be attacked initially in terms of machine instructions, but it can be approached in terms and entities natural to the problem itself. In this process, the abstract program formulated in a suitable language,

such as pseudocode, performs operations on abstract data. The operations (e.g., compute mean temperature value) are then considered as program components that are subjected to decomposition to the next *lower* level of abstraction. This process is continued to a level that finally can be understood by the computer.

The absence or rare occurrence of GO TO statements is no more than a symptom of structured programming.

Stepwise refinement, the process we have briefly outlined above, is a procedure that involves decomposing the function of a module into an expanded set of subfunctions. Each of these subfunctions is independently decomposed further into an expanded, but equivalent, set of subfunctions that are ultimately expanded into operations that can be translated directly into a programming language. Although this refinement technique can be applied to the total decomposition of a program into its elementary program statements, it is most effective as a tool for decomposing each module of a program into the internal logic necessary to perform the module function.

Note that this method of stepwise refinement permits only the rare occurrence of unconditional branches because the subfunctions have no requirement to communicate with each other. If the requirement does occur during the refinement process, the decomposition is usually in error. The absence of unconditional branches is not the goal but, instead, a result of the process.

When using stepwise refinement as a programming methodology, each iteration provides a more detailed description of the program control logic. The initial iterations provide very general processes, and the designations of these processes in the development are in a language that is very close to the specification. As the refinement process approaches the machine level, the language appears more like a programming language.

Structured programming can be summarized as a set of four important guidelines that must be kept in mind while developing a program structure by stepwise refinement.

Postpone details. Program details at the abstract design levels are trivia that cloud the thinking process. It is important to consider the major functions of the module early in the refinement without getting confused by details that are not necessary until later refinement steps. Details should come into focus only when they have a direct impact on the refinement process.

Make decisions at each level of abstraction carefully. At each refinement stage, some decisions are obvious and easy to make, while others are subtle. We must try to analyze each decision as it is made to thoroughly understand the implications associated with that decision. We must consider all of the alternatives, but we must not get

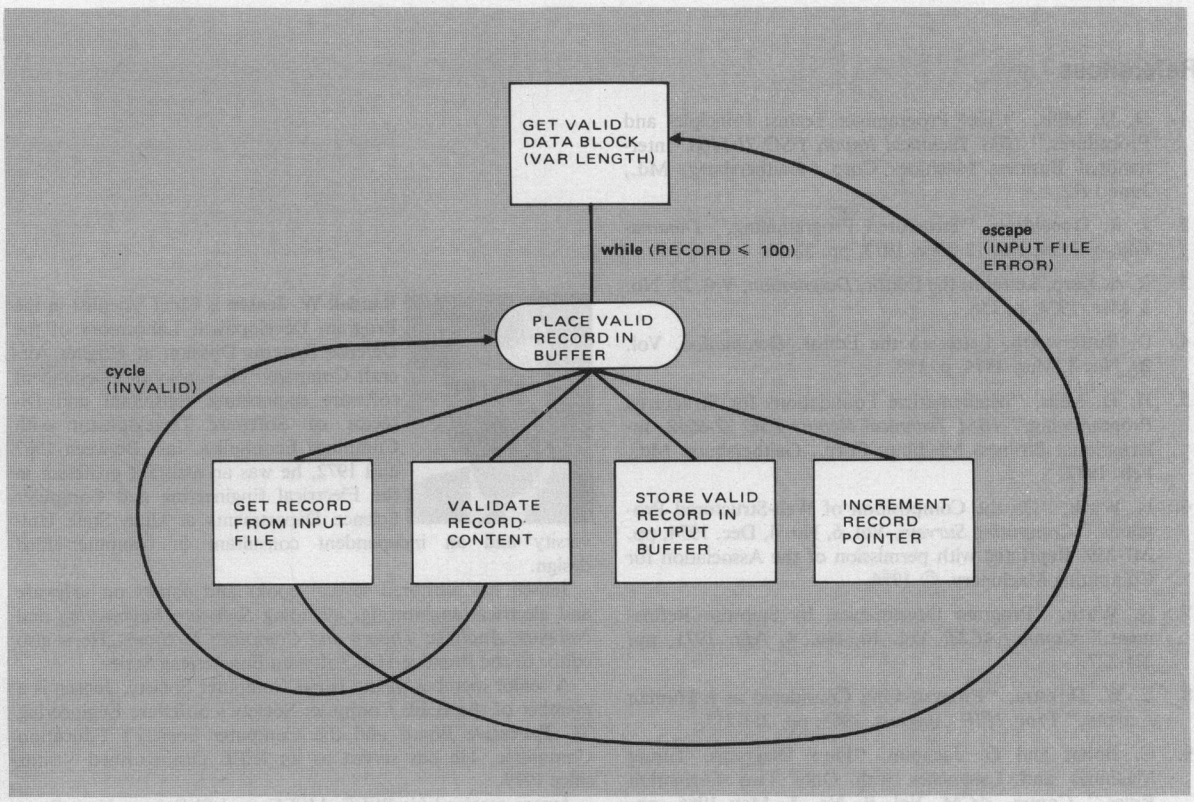


Figure 23. Implementation of PLT exit structures.

trapped by details at far-removed lower levels. This trap is basically the one related in the old saying, "can't see the forest for the trees."

Be flexible. No decision is final. Problems will arise requiring earlier decisions to be revoked or modified. We should not be intimidated by the existing structure. Each revision of the program logic, due to a correction of a discovered error, should improve the overall structure.

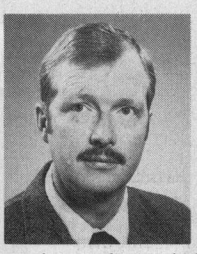
Consider the data. As the refinement process is carried to lower levels of abstraction, data items necessary for the function of the module will be uncovered. As these data items appear (they will also be abstract at first), we need to keep track of them and refine their definitions as greater detail is available. The data items and the logic structure are both functions of decisions made during the decomposition of the module functions. The data required for any program configuration is an important item to consider in the decision process. ■

Acknowledgment

The author acknowledges the support of Prentice-Hall, Inc., for their permission to use numerous excerpts from *Software Engineering* by R.W. Jensen and C.C. Tonies (1979) in this tutorial. The author also expresses his gratitude to R. Gipson and J. Poulson for their meticulous review and constructive criticism of this work.

References

1. H. D. Mills, "Chief Programmer Teams: Principles and Procedures," *IBM Technical Report FSC 71-5108,* International Business Machines Corp., Gaithersburg, Md., June 1972.
2. J. R. Donaldson, "Structured Programming," *Datamation,* Vol. 19, No. 12, Dec. 1973, pp. 52-54.
3. R. A. Karp, Letter to the Editor, *Datamation,* Vol. 20, No. 3, Mar. 1974, p. 158.
4. D. Butterworth, Letter to the Editor, *Datamation,* Vol. 20, No. 3, Mar. 1974, p. 158.
5. H. D. Mills, "Mathematical Foundations for Structured Programming," *IBM Technical Report FSC 72-6012,* International Business Machines Corp., Gaithersburg, Md., Feb. 1972.
6. N. Wirth, "On the Composition of Well-Structured Programs," *Computing Surveys,* Vol. 6, No. 4, Dec. 1974, pp. 247-259. Reprinted with permission of the Association for Computing Machinery, © 1974.
7. N. Wirth, "Program Development by Stepwise Refinement," *Comm. ACM,* Vol. 14, No. 4, Apr. 1971, pp. 221-227.
8. E. W. Dijkstra, "Programming Considered as a Human Activity," *Proc. IFIP Congress,* 1965, pp. 213-217.
9. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Comm. ACM,* Vol. 9, No. 5, May 1966, pp. 366-371.
10. E. W. Dijkstra, "Go To Statement Considered Harmful," *Comm. ACM,* Vol. 11, No. 3, Mar. 1968, pp. 147-148. Reprinted with permission of the Association for Computing Machinery.
11. W. A. Wulf, "A Case Against the GOTO," *Proc. ACM Ann. Conf.,* Aug. 1972, pp. 791-797.
12. M. E. Hopkins, "A Case For the GOTO," *Proc. ACM Ann. Conf.,* Aug. 1972, pp. 787-790.
13. F. T. Baker, "Chief Programmer Team Management of Production Programming," *IBM Systems J.,* Vol. 11, No. 1, Jan. 1972, pp. 56-73.
14. W. W. Peterson, T. Kasami, and N. Tokura, "On the Capabilities of While, Repeat, and Exit Statements," *Comm. ACM,* Vol. 16, No. 8, Aug. 1973, pp. 503-512.
15. *Posit* means "to lay down or assume as a fact; affirm; postulate." *Standard Dictionary of the English Language,* Funk and Wagnalls, New York, 1960.
16. M. A. Jackson, *Principles of Program Design,* Academic Press, London, 1975.
17. D. E. Knuth, "Structured Programming with GO TO Statements," *Computing Surveys,* Vol. 6, No. 4, Dec. 1974, PP. 261-301.



Randall W. Jensen is Chief Scientist in the Program Development Laboratory of the Defense Systems Division at Hughes Aircraft Company. He is also an independent software engineering consultant and Director of Software Development with Computer Economics, Inc. Between 1967 and 1972, he was an assistant professor in the Electrical Engineering and Computer Science Departments at Utah State University and an independent consultant in computer-aided design.

Jensen has authored several books and papers on software and electrical engineering, including *Software Engineering* and *Network Analysis: Theory and Computer Methods*. He is also editor of the Prentice-Hall Software Engineering Series.

A senior member of the IEEE Computer Society, Jensen is a member of the IEEE Computer Society's Software Engineering TC Executive Board and the Computer Society's Education Committee. He has served as an IEEE Distinguished Visitor since 1979.

Jensen received his BSEE, MSEE, and PhD from Utah State University in 1959, 1961, and 1970, respectively.