Association for Information Systems
AIS Electronic Library (AISeL)

ICIS 1985 Proceedings

International Conference on Information Systems (ICIS)

1985

Techniques for Understanding Unstructured Code

Mel A. Colter
University of Colorado at Colorado Springs

Follow this and additional works at: http://aisel.aisnet.org/icis1985

Recommended Citation

Colter, Mel A., "Techniques for Understanding Unstructured Code" (1985). *ICIS 1985 Proceedings*. 6.
http://aisel.aisnet.org/icis1985/6

This material is brought to you by the International Conference on Information Systems (ICIS) at AIS Electronic Library (AISeL). It has been accepted for inclusion in ICIS 1985 Proceedings by an authorized administrator of AIS Electronic Library (AISeL). For more information, please contact elibrary@aisnet.org.

Techniques for Understanding Unstructured Code

Mel A. Colter
Associate Professor of Management Science and Information Systems
College of Business Administration
University of Colorado at Colorado Springs
P.O. Box 7150
Colorado Springs, Colorado 80933-7150

ABSTRACT

Within the maintenance activity, a great deal of time is spent in the process of understanding unstructured code prior to changing or fixing the program. This involves the comprehension of complex control structures. While automated processes are available to structure entire programs, there is a need for less formal structuring processes to be used by practicing professionals on small programs or local sections of code. This paper presents methods for restructuring complex sequence, selection, and iteration structures into structured logic. The procedures are easily taught and they result in solutions of reduced complexity as compared to the original code. Whether the maintenance programmer uses these procedures simply for understanding, or for actually re-writing the program, they will simplify efforts on unstructured code.

Introduction

The maintenance of existing software comprises a major portion of the productive effort of the software industry. Though estimates vary and questions concerning the exact demarcation between development and maintenance persist, most authorities agree that 40-75% of all DP budgets are expended on maintenance (Boehm, 1981; Couger and Coulter, 1985; Elshoff, 1976; Lientz and Swanson, 1978). From another perspective, Boehm (1981) has surveyed estimates which indicate that up to 75-80% of all life cycle costs are expended on maintenance. The increasing average life of software (Boehm, 1981), along with the growing amount of software entering the maintenance process, indicate that maintenance costs will continue to rise, both in terms of absolute budgets and in terms of total life cycle costs.

A growing body of literature reflects this importance by treating the maintenance effort from multiple perspectives. Some authors have contributed papers which aid understanding of the maintenance process itself (Colter and Couger, 1984; Harrison, et. al., 1982; Vessy and Weber, 1983). Others have concentrated on factors affecting maintenance loads (Berns, 1984; Colter and Couger, 1984; Elshoff, 1976; Gremillion, 1984). Another subset of the literature discusses the management and productivity issues related to the maintenance task (Colter and Couger, 1984; Couger and Colter, 1985; Guimaraes, 1983; Lientz and Swanson, 1981).

Despite the growing literature on maintenance, however, very little published support for practical tools and techniques for performing maintenance exists. While we have greatly improved our understanding of the maintenance process, we have done little to aid the maintenance programmer directly. Early work on translating programs with GOTO's into programs with DOWHILES was published in 1971 (Ashcroft and Manna, 1971). Except for a few other translation and style articles, little else of direct applicability to maintenance programmers appeared until 1982. Then, Elshoff and Marcotty suggested a method for improving the readability of existing code through a series of transformations of the code (Elshoff and Marcotty, 1982).

It is the thesis of this paper that tools, techniques, and methodologies are badly needed to aid the maintenance programmer. In the maintenance mode, most of the target code is of substantially lower quality than we would like. At the same time, a growing percentage of our new employees are coming into maintenance with a good background in structured programming but absolutely no preparation for understanding, modifying, and retesting unstructured programs. This paper extends the work of Elshoff and Marcotty by providing some simple

techniques to aid maintenance programmers in the understanding of poorly structured code.

The Unstructured Code Problem

THE QUALITY OF MAINTAINED CODE

A large percentage of the code in maintenance fails to meet today's generally accepted program quality standards. The reasons for this are many, but the following points are most explanatory. First, much of the industry's existing code is old, having been written prior to conversion to the structured techniques. Second, many organizations have yet to implement improved program design and construction standards. Finally, for those shops where clean code is delivered into maintenance, that clean code often degenerates rapidly due to unconstrained maintenance efforts.

The sad truth is that much of the code in maintenance today is of poor design and construction. This problem was noted in a survey of programmers by Lientz and Swanson (1980) and in another survey of programmers and managers by Couger and Colter (1984, 1985). In those studies, programmers reported that poor program design and poor program code accounted for the majority of their problems in the maintenance environment.

The concept of code quality may be discussed at a number of different levels. For the purposes of this paper, a well-structured program is considered to be one which is comprised of a set of hierarchically related modules where the individual modules are of low complexity and easy to understand. In addition, at the code level, the control structures are expected to be predictable and recognizable, reflecting the practices of structured logic.

Unfortunately, much of the code in maintenance consists of large programs (hundreds and even thousands of lines of code per module), which reflect anything but structured logic. This code exhibits complex control structures which must be understood before any maintenance efforts can be successful.

CONTROL FLOW COMPLEXITY

A great deal of discussion on complexity as it relates to maintenance has appeared in the literature. Harrison, et. al. (1982) suggest that control flow metrics do a good job of differentiating between two programs which are otherwise similar. In addition, it appears that the complexity of the control flow of a program is directly related to the amount of time spent in understanding existing code prior to making a change. Therefore, this paper concentrates on a set of techniques for understanding the control flow of unstructured code.

There are two types of control structures contained within any program. They are:

1) Problem-related control structures

2) Implementation-related control structures.

Problem-related control structures are those necessary to solve the program problem effectively. Implementation-related control structures, on the other hand, exist in code only because of the nature of the program solution chosen by the programmer or maintainer and these structures may have little or nothing to do with the original program problem. The issue here is that the maintenance programmer, upon examining the program, has no simple way to determine which control structures are integral to the functionality of the module and which are there simply as a result of poor design or coding practices.

The study and understanding of these combined sets of control structures comprise a significant portion of the amount of time necessary to perform a specific maintenance task. In an informal study of over 200 maintenance programmers undertaken by this author, respondents reported that over 50% of their time in a maintenance effort is taken up by the efforts necessary to understand code prior to making a change. When questioned about this understanding effort, the vast majority of respondents indicated that the clarification of control structures accounted for a large portion of the understanding effort.

Because of the importance of the control structure to maintenance efforts, a simple control related complexity measure will be used throughout the remaineder of this paper to provide comparisons between similar pieces of code. That measure will be the number of branching statements plus 1, which is an approximation to McCabe's cyclomatic complexity number and the lower bound on the complexity calculated by Myers (See Harrison, et. al. (1982)). While a number of other code measures and metrics exist, this simple metric is useful for the comparison of alternative solutions.

THE NEED FOR TOOLS

As indicated above, a great deal of software maintenance is performed on large, complex programs which exhibit unpredictable control flows which require up to 50% of the maintenance effort to understand. Worse yet, much of the effort spent on understanding the existing code is of only short term value since maintenance program-

mers' notes and other on-the-spot documentation are usually thrown away after the change is successfully made. As a result, the same understanding effort may occur on the same piece of code multiple times over the life of the program. This is an unnecessarily redundant expenditure of scarce resources in the maintenance environment. If the understanding component can be reduced, and if the results of that understanding can be saved effectively, then it should be possible to dramatically reduce the cost of maintaining many systems.

Clearly, the maintenance programmer needs tools to aid in the understanding of existing code. In this paper, a set of procedures are offered to meet this need. As noted before, these procedures are extensions of the technique offered by Elshoff and Marcotty (1982). While their approach results in the restructuring of code, it suffers from three major weaknesses. First, it is highly formal and implies that the programmer will actually rewrite the code as a part of the restructuring process. Unfortunately, the rewriting of code is often frowned upon in shops which subscribe to the old adage, "If it ain't broke, don't fix it!" As a result, maintenance programmers who could otherwise benefit from the Elshoff and Marcotty approach fail to reap those benefits because of their perception that they must use all of the procedure and not just part of it. Second, the procedures described by Elshoff and Marcotty require more detailed instructions to be useable to most practicing professionals. Finally, the procedure appears highly formalized. As a tool, it is therefore hard to expect maintenance programmers to use it frequently. Weiser (1982) comments that programmers approach complex programs by using tools in a hierarchical manner. That is, they first attempt to use simple techniques to solve their problems, then move to more complex approaches only when necessary. They continue to apply stronger tools in a stepwise fashion until they succeed in using a tool strong enough to meet the complexity of the problem. The techniques presented in this paper may be used in a highly informal manner, yet they are sufficiently robust to aid in the understanding of extremely complex code.

The goal of the paper is therefore to describe code analysis and understanding tools which:

1) are easy to use

2) significantly decrease the understanding component of a maintenance effort

3) support documentation to aid future maintenance efforts

4) support actual code rewriting when desired.

The processes discussed here use pseudocode as an alternative representation of logic. If unstructured logic can be represented in pseudocode with only sequence, selection, and iteration, then complexity is reduced and understandability is increased.

AUTOMATED VERSUS NON-AUTOMATED RESTRUCTURING

In the past several years, a growing number of automated code restructuring systems have become available. One can now submit COBOL programs to one of several companies and receive a restructured version which meets the rules of structured programming. While some programs are candidates for this type of automated restructuring, the process is not without problems. First, the number of source lines usually increases significantly as a result of the process. Second, the size of the load module increases, as does the average run time for most such programs. Third, the control structures inserted by the automatic restructuring routines seldom have anything to do with the original problem, resulting in a preponderance of implementation related control which obscures the problem related control. Therefore, the understandability of the resulting code remains lower than one would like. Finally, it is often helpful if small programs or program segments can be restructured for understanding purposes without submitting a large program or system for automatic restructuring. It is clear that a large amount of code in existing production libraries will remain in its present state for some time and that human intelligence will be the vehicle for understanding of code prior to maintenance. As Elshoff and Marcotty said in 1982,

> "The understanding developed by the programmer is generally well beyond the capability of artificial intelligence, and the undesirable side-effects often introduced by automatic restructuring techniques can be avoided."

The following section describes tools and techniques which utilize the knowledge of the programmer to achieve a true understanding of code.

Restructuring Techniques

THE RESTRUCTURING PROCESS

When working with poorly structured, complex code, it is generally impossible to attach all of the weakness of the program simultaneously. As a result, programmers usually seek to identify subsets of the program which support meaningful efforts. Weiser (1982) refers to these

program subsets as "sliced" which represent relevant portions of a program for the purposes of specific analysis.

The techniques presented here explicitly assume the use of slices to segment code into understandable and modifiable segments. As Weiser points out, there are many different types of slices, and more than one will be used here. However, the most common slice will be the *code block*. A code block is defined as a set of contiguous statements which have a single entry and a single exit. The code block may be a few lines of code, or it may be an entire program. The importance of the code block in the analysis of a program for understandability is twofold. First, statements in code blocks may be clumped together to simplify the program portion in which the block resides. Second, in order to reorder or otherwise modify the logic in a section of code, that section of code must represent a code block. That is, the single entry, single exit criterion is critical to the re-representation of logic.

In this paper, it is suggested that the restructuring and understanding process begin with the most straightforward targets of opportunity and progress towards the more difficult portions of code. In general, the easiest way to simplify a program is to deal with code blocks which are simply out of place. When code blocks are moved to their appropriate location in the program, control structures are reduced and the sequential nature of the logic is clarified.

After the sequential nature of the logic is cleaned up, then the selection constructs are usually the next easiest portions of the code to understand. In languages which do not support the IF-ELSE-ENDIF structure, the selection construct accounts for a great deal of implementation related control. As a result, the re-representation of unstructured selection contructs greatly simplifies the logic of the program. Finally, after the sequence and selection contructs are understood, the maintenance programmer can concentrate on the iteration constructs. Unstructured loops are among the most difficult to understand and it is best to simplify the program to the greatest extent possible before tackling them. The following discussions present detailed examples of the understanding and re-representation of sequence, selection, and iteration.

CODE BLOCKS—THE SEQUENCE PROBLEM

The simplest code block to recognize and deal with results when a block of code simply resides in one portion of the program while its execution belongs in another location. For example, Figure 1 shows a situation in which code block C has been added to the end of a subroutine rather than inserted into the logic where it belongs. This type of situation may reflect a last minute addition during design, or it may be the result of an addition of code during maintenance. In any case, it decreases the readability of the program and increases the complexity through the addition of two unnecessary control statements. These control statements are classic examples of implementation related control. They have absolutely nothing to do with the original problem and greatly decrease understandability.

In this situation, code block C cannot be reached through sequential execution and it is clear that it can simply be moved to the appropriate location in the program. This is illustrated in Figure 1, resulting in a reduction in the control flow complexity of two. Notice that, assuming no other reference to Label-1 and Label-2, they may be removed, further simplifying the program. Furthermore, note that the new configuration of code blocks A, C, and B may support their merging into a single conceptual block, since no control structures exist to separate them.

A more common type of sequential code block problem is illustrated in Figure 2. There, the code block is reused rather than duplicated in the code. For the purposes of understanding the section of code in which this structure resides, it is worth copying the code block to achieve a reduction in the control complexity of the code of interest. As shown in Figure 2, the copying of the code block C allows us to remove two inplementation related control structures, delete the use of the control variable, FLAG, and remove references to Label-1 and Label-2.

When trying to simplify code to aid understanding, this treatment of code blocks is the best place to begin restructuring the code. First, the structures are relatively easy to identify in the code. Second, each time a code block is moved or copied to its proper sequential location, the control complexity is reduced by two and the understandability is greatly increased. Even though this process may result in an increase in the actual amount of code in the program, that increase is easily offset by the positive results of the process. Once all of the opportunities for the clarification of the sequential structure of the program have been exhausted, then the more complex structures may be examined.

THE SELECTION STRUCTURE

Of the three major logical structures (sequence, selection, and iteration), the selection construct becomes the most awkward when it is not implemented cleanly. When the IF-ELSE-ENDIF structure is not available in a lan-

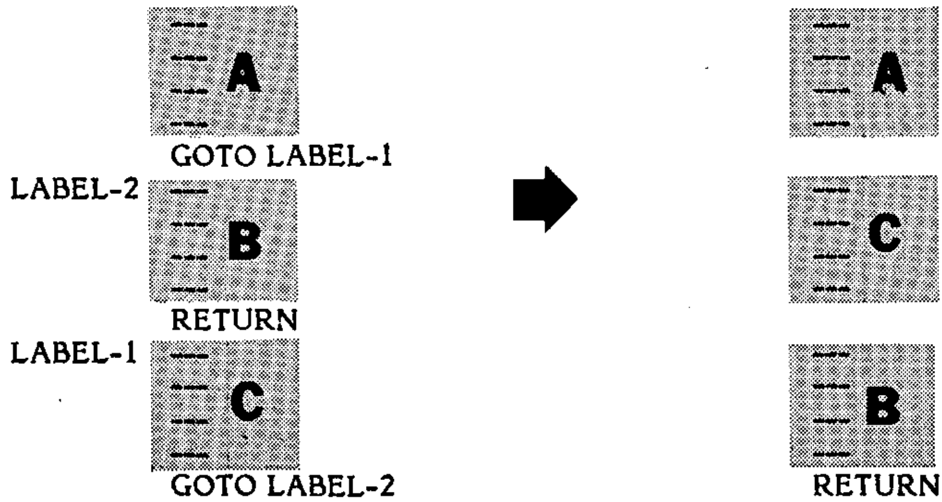

Figure 1

Code Block Out Of Place

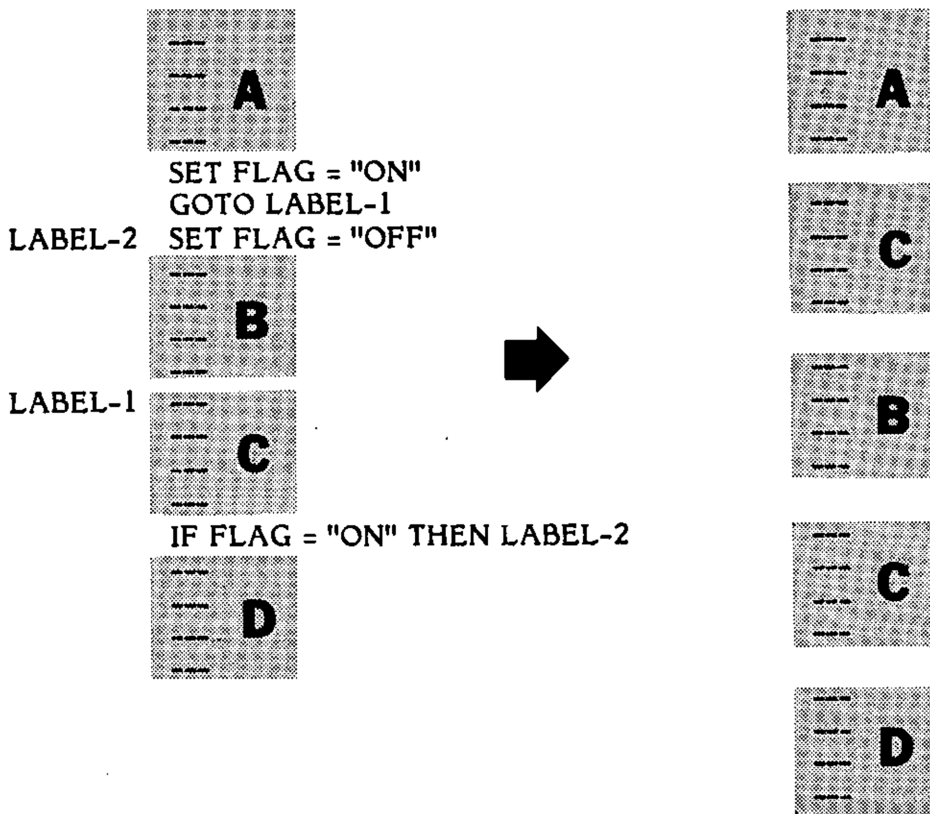

Figure 2

Code Block Duplicated

guage, or when it is available but not used, the program will exhibit complex combinations of conditional and unconditional branches. As a result, the exact nature of the original problem becomes obscure and maintenance efforts are extremely difficult.

The restructuring of complex selection contructs requires a careful set of steps as indicated below.

1) Isolate the selection structure as a code block with single entry and exit.

2) Expand the structure by formalizing the IF-ELSE-ENDIF structures.

3) Collapse the structure into itself by moving internal code blocks.

4) Remove redundant control statements.

This process is illustrated in Figure 3, and the following discussions clarify the series of transformations suggested for the example. This code is a simplification of code from an actual program, and it is common within the programs of many shops. First, note that the structures of interest for this example are simplified by summarizing all but the important control structures. For example, the line ∘ PROCESS-A ∘ represents an internal code block withing the structure. That code block may be a single line of code or it may contain significant complexity of its own. Second, the structure must be recognized as a selection structure and isolated as a code block with single entry and exit.

Considering the problem of recognizing this type of structure, note that the initial version of the code in Figure 3 represents the original unstructured code. In that version, note that the control flows are all downward and intersecting with a subset of the control branches terminating at a common exit. This set of characteristics is commonly seen in structured implementation of the selection construct.

The second version of the program in Figure 3 results from the expansion of the structure through the formalization of the IF-ELSE-ENDIF structures. In this process, a simple translation into formalized pseudocode has occurred. The IF-ELSE-ENDIF structures are clarified through some expansion of the original code. In addition, note that an implied GOTO has been added at the end of the code at LABEL-5. In the original code, there is a sequential execution of LABEL-4 after LABEL-5. However, the restructured version will probably result in the movement of LABEL-5 as a single entry, single exit code block and the transfer code of control to LABEL-4 must be maintained. As a result, the implied (GOTO LABEL-4) at the end of the structure is always added to the code at this point.

The next step in the restructuring of the code involves the collapsing of the structure into the selection constructs. While this process may be performed quickly by a professional maintenance programmer who is familiar with the process, it is broken into two steps here for the purposes of illustration. The key to the collapsing process is to realize that the second version of the code contains two code blocks which present opportunities for relocation. First, note that the code at LABEL-3 and LABEL-5 ends with control transfers to the end of the code segment. Second, note that both of these code blocks are single entry, single exit, and that they are accessed only through the execution of additional GOTO's in LABEL-1 and LABEL-2. As a result, the code block at LABEL-3 can replace the GOTO LABEL-3 within LABEL-1 and the code block at LABEL-5 can replace the GOTO LABEL-5 within LABEL-2. The third version of the program segment reflects this set of code block movements.

Now, recognize that all of the code under LABEL-1 represents a code block with a single entry and exits to a common location. Furthermore, this block is directly accessed through the execution of the GOTO LABEL-1 at the top of the code. As a result, all of the code under LABEL-1 can be moved to replace the GOTO LABEL-1 statement. The same argument allows us to move all of the code under LABEL-2 to replace the GOTO LABEL-2 statement. Note here that the explicit declaration of the implied (GOTO LABEL-4) which was added early in the process is now critical. Without that implied GOTO, the code block movement would be highly constrained.

The fourth version of the program segment illustrates the complete collapsing of the structure into the set of IF-ELSE-ENDIF structures. Nore that there are four occurrences of the statement, GOTO LABEL-4, in this version. However, the natural operation of the selection construct makes these statements totally redundant. Whenever the execution reaches one of these statements, the operation of the selection constructs would result in a clean jump to the end of the construct anyway. The removal of these unnecessary control statements is illustrated in Figure 4, along with the original code for comparison. It is clear that the restructured logic is much easier to read and that the programmer who understands the restructured version will be able to work with the original version if necessary. Note also that the complexity of this code has been reduced to a value of 4 from an original value of 8.

UNSTRUCTURED LOOPS

The last major structure causing problems in unstructured code is the iteration structure. Here, because of the weaknesses of specific languages or due to improper use of stronger languages, programmers create multiple exit loops and intersecting loops which make maintenance

```
          IF CONDITION-1 THEN GOTO LABEL-1
          GOTO LABEL-2
LABEL-1   IF CONDITION-2 THEN GOTO LABEL-3
          · PROCESS-A ·
          GOTO LABEL-4
LABEL-3   · PROCESS-B ·
          GOTO LABEL-4
LABEL-2   IF CONDITION-3 THEN GOTO LABEL-5
          · PROCESS-C·
          GOTO LABEL-4
LABEL-5   · PROCESS-D ·
LABEL-4   · CONTINUE ·
```

```
          IF CONDITION-1
               GOTO LABEL-1
          ELSE
               GOTO LABEL-2
          ENDIF

LABEL-1
          IF CONDITION-2
               GOTO LABEL-3
          ELSE
               · PROCESS-A ·
               GOTO LABEL-4
          ENDIF

LABEL-3
          · PROCESS-B ·
          GOTO LABEL-4

LABEL-2
          IF CONDITION-3
               GOTO LABEL-5
          ELSE
               · PROCESS-C ·
               GOTO LABEL-4
          ENDIF

LABEL-5
          · PROCESS-D ·
          (GOTO LABEL-4)

LABEL-4·
          · CONTINUE ·
```

```
          IF CONDITION-1
               GOTO LABEL-1
          ELSE
               GOTO LABEL-2
          ENDIF

LABEL-1
          IF CONDITION-2
               · PROCESS-B ·
               GOTO LABEL-4
          ELSE
               · PROCESS-A ·
               GOTO LABEL-4
          ENDIF

LABEL-2
          IF CONDITION-3
               · PROCESS-D ·
               (GOTO LABEL-4)
          ELSE
               · PROCESS-C ·
               GOTO LABEL-4
          ENDIF

LABEL-4
          · CONTINUE ·
```

```
          IF CONDITION-1
               IF CONDITION-2
                    · PROCESS-B ·
                    GOTO LABEL-4
               ELSE
                    · PROCESS-A ·
                    GOTO LABEL-4
               ENDIF
          ELSE
               IF CONDITION-3
                    · PROCESS-D ·
                    (GOTO LABEL-4)
               ELSE
                    · PROCESS-C ·
                    GOTO LABEL-4
               ENDIF
          ENDIF

LABEL-4
          · CONTINUE ·
```

Figure 3

Understanding Unstructured Selection Constructs

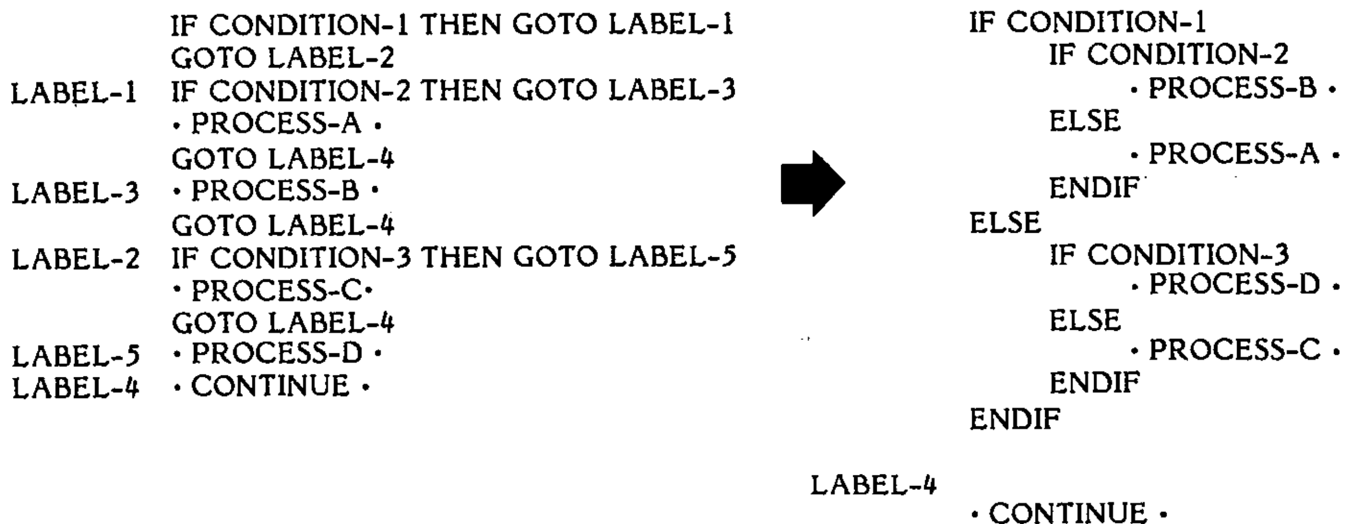

Figure 4

The Restructured Selection Construct

efforts extremely difficult. This section first treats the multiple exit loop problem. Then, the intersecting loop problem is discussed at length.

Multiple Exit Loops

Loops with multiple exits are quite common in older, unstructured code. Additionally, newer code often exhibits this characteristic due to the need for multiple paths out of iterative structures. For example, in on-line systems, loops may terminate normally, because of a bad data value, or because of the use of an interrupt key by the operator. While these problems may be handled with purely structured logic, the resulting solutions often contain multiple levels of nested IF structures and programmers commonly refuse to implement such structures.

Figure 5 shows a multiple exit loop and an alternative solution to the code segment. First, notice that the original code has two branches to labels which are external to this code segment. These branches violate the single entry, single exit criterion. Worse, they may transfer control to portions of the program which are far away from the segment of interest. The maintenance programmer who must trace an error through this loop will have to locate the external labels. In some cases, it may be difficult or impossible to determine exactly which exit from the loop was accomplished for a given situation.

In the alternative solution, the total amount of code has been increased in order to clarify the loop structure. It is assumed in this case that the variable, I, was originally used simply to create a looping structure which would be exited through one of the internal exit structures. However, that variable has been included in the alternative solution to provide an error procedure in case the loop is not exited in a normal fashion. Otherwise, the loop will be terminated when the variable, EXIT-CONDITION, is set to anything other than "NULL". The form of this solution requires that GOTO statements be embedded in the code, but they branch downward and only to the end of the logical structure. This use of GOTO statements, while not approved by purists, is still an improvement over the original code.

The real strength of the new solution is that it explicitly indicates the methods by which the loop exit can be accomplished at the end of the loop structure. An examination of the current value of EXIT-CONDITION will clearly indicate the nature of the last loop exit. Furthermore, the structure easily accommodates the later insertion of additional exit criteria during maintenance of the program.

The Intersecting Loop Problem

Of all unstructured program problems, the intersecting loop situation is among the most difficult to understand, debug, or modify. This section presents a stepwise transformation process which converts intersecting loops into a set of logical structures using only DOWHILE and DOUNTIL structures. The discussion uses the example

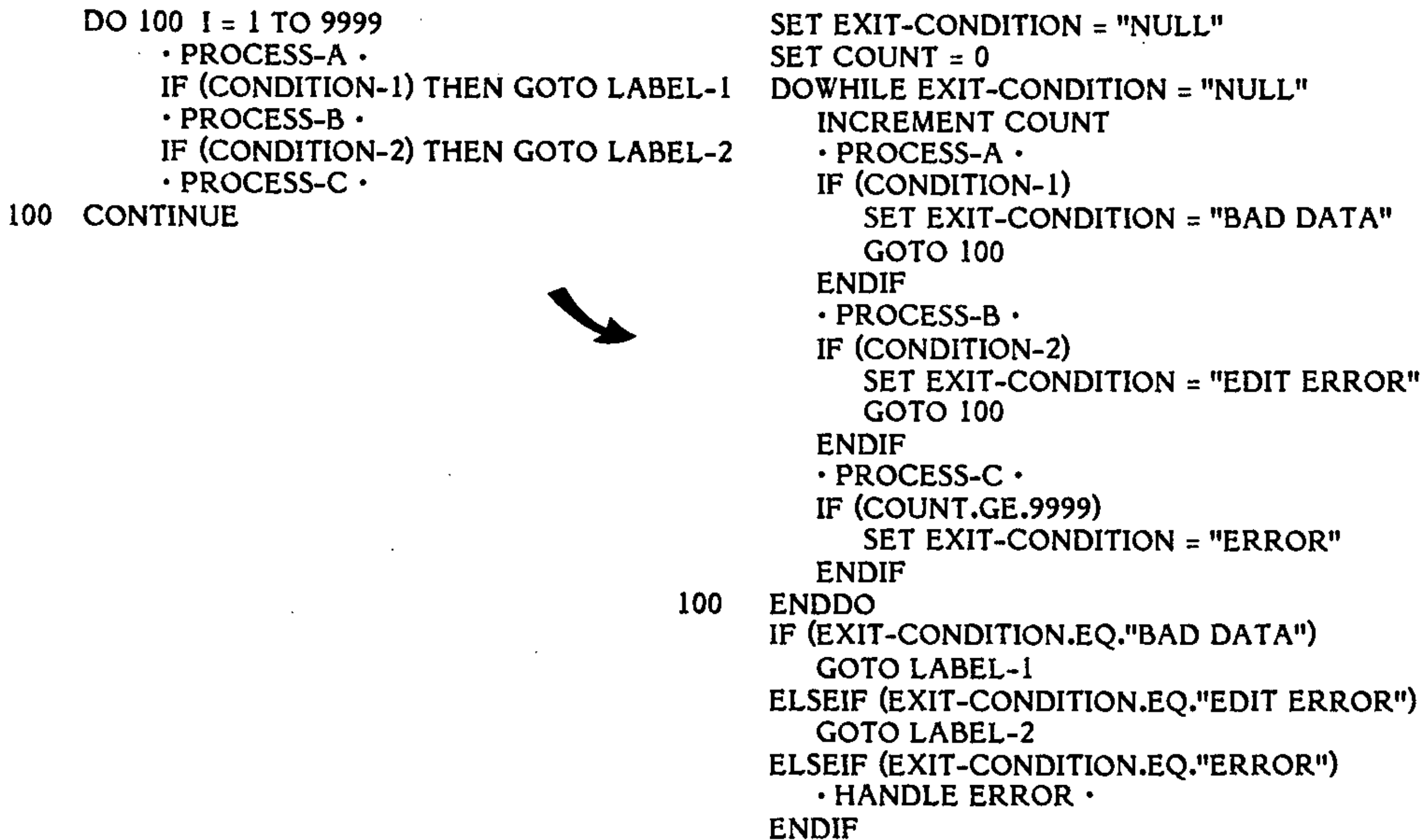

Figure 5

Multiple Loop Exits

in Figure 6 and consists of the following steps.

1) Isolate the looping structure as a code block with single entry and exit.

2) Simplify the structure by identifying internal code blocks.

3) Represent the simplified structure as a flowchart.

4) Convert the flowchart to pseudocode using only structured logic.

5) Simplify the pseudocode.

In Figure 6, it is assumed that the code represented in the example is a single entry and exit code block and that no other references to the statement labels 100 and 200 exist. Furthermore, assume that the lines of code between the labels and the control statements are irrelevant to this analysis. That is, those lines of code are either pure sequence, or they contain control structures of their own, but they may be represented as code blocks for the purposes of understanding the looping constructs. It is critical to this analysis that only the looping structures remain in the target code. This is why the simplification of the sequence and selection structures is performed first. If all other opportunities for simplification have been taken, then only looping structures remain for consideration. The second portion of Figure 6 shows the introduction of code blocks A through E to achieve the simplification necessary to consider the loops.

Once the code is simplified and the loops are clearly identified, a simplified flowchart of the program may be drawn. This step is important in the transformation of the intersecting loops into structured logic. Remember that intersecting loops are not possible when only sequence, selection, and iteration are used. Therefore, the original program cannot be converted directly into structured pseudocode. In this case, the more general logical representation available through flowcharts must be used as an

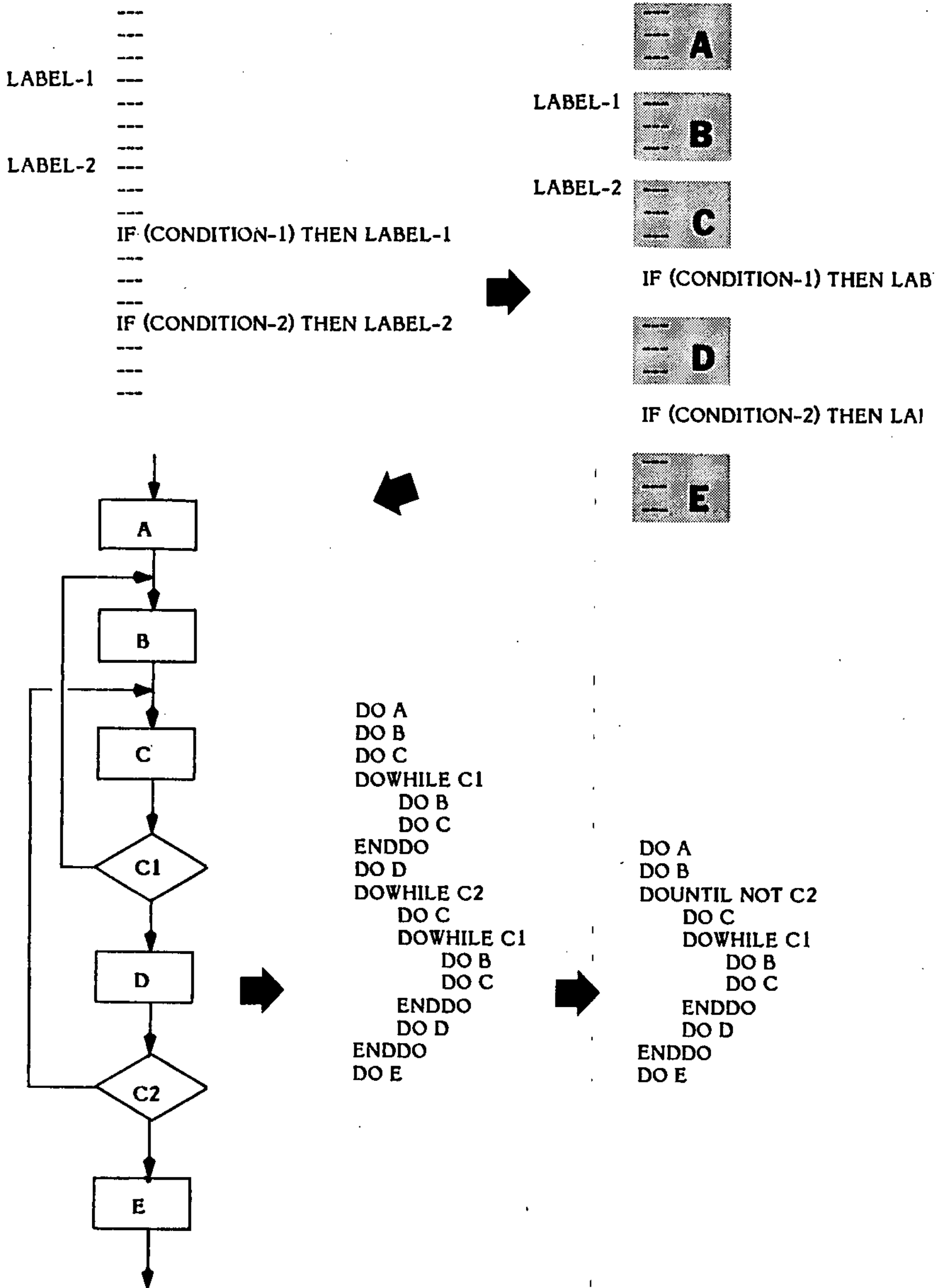

Figure 6

Intersecting Loops

intermediate transformation. Only then can the structured logical constructs be identified.

The fourth section of Figure 6 shows the pseudocode equivalent of the flowchart. The process by which this pseudocode is obtained is simplified if a few straightforward rules are followed. First, simply represent a single flowchart symbol, one at a time, resulting in a single code block operation for each line of pseudocode. Second, never anticipate loops. Always wait to implement a loop until the condition branch symbol is encountered. In this case, the first conditional branch checks the value of Condition-1. At this point, since the program checks the condition prior to the execution of the loop, the pseudocode representation requires a DOWHILE structure. This should always be the case when translating from flowcharts to pseudocode. Always wait until the conditional control transfer is encountered and then implement a DOWHILE. This is not to say that no DOUNTILs will be encountered in this process. They will be discovered in later steps as the pseudocode is simplified.

After the pseudocode representation is obtained, examine the structure for opportunities for simplification. In general, this simplification will occur when common code blocks are identified and recombined. In this case, the two shaded portions of the pseudocode solution are duplicate blocks. In fact, the common block is performed once and then performed again in a DOWHILE structure. This can be re-represented as a DOUNTIL. The final portion of Figure 6 shows that simplification. At this point, there is no further obvious simplification possible of the logic.

Note that the original problem in this case contained two loops which intersected. However, the structured solution contains two nested loops. Though no formal proof is known to this author, it has been my experience and the experience of others using this simplification process that a set of n intersecting loops always converts into a set n nested loops in the structured logic.

A GENERAL EXAMPLE

The preceding sections have discussed a set of procedures by which problems of sequence, selection, and iteration can be restructured through re-representations in structured logic. This section provides a detailed discussion of a code section in which a number of such problems exist. Within the code section presented in Figure 7, there is a code block out of place due to the re-use of code within the program. Additionally, there is a selection construct which has been implemented with conditional and unconditional branching statements. Finally, when all of the other issues are clarified, a set of intersecting loops are found to exist. The following paragraphs detail the use of the rules discussed in the earlier sections to obtain an alternative representation of this logic. The goal of the new representation is to obtain a version of the logic which can be understood by the maintenance programmer.

First, in Figure 7, assume that the relevant code has been examined and that there are no external references to any of the statement labels indicated in the code. Furthermore, assume that the segment is single entry and single exit. Also, in Figure 7, note that the program segment has been broken into a set of code blocks to simplify discussion of the problem.

The next step involves the identification and handling of any code blocks which are out of sequence. Examination of the code reveals that code block E is used in two ways. First, it is executed sequentially immediately after block D. However, it is also executed through the use of a switch and some control code after the processing of block B. Here, the copying of code block E between blocks B and C allows us to delete both references to SW1, and remove the two control statements, GOTO 300 and IF SW1 = "ON" THEN 400. Finally, statement labels 300 and 400 are no longer needed.

Figure 8 shows the results of moving the code block and the removal of the implementation related control. The control related complexity of the original program was 10. Now, that complexity has been reduced to 8.

Figure 8 also shows that, without the control statements and unnecessary labels, the blocks within the program may be re-identified. Blocks B, E, and C may now be combined for the remainder of the analysis, resulting in the new block designations shown in the figure.

Since there are no more opportunities to move code blocks to simplify the sequence structure of the program, we now seek to identify selection constructs within the code. It is clear that the code in blocks D, E, F, G, and H is related through a set of control structures which are downward branching and intersecting. Furthermore, there is a common exit indicated at line 600. As noted earlier, this is an example of the kind of solutions which result when selection constructs are built with conditional and unconditional branches.

Figure 9 shows the program segment after the selection structure has been restructured. As a result of this restructuring, the complexity has dropped to 6. Therefore, the understanding process has begun with a program with a complexity of 10 and reduced it to a complexity of 6 in only two steps. The restructuring process used here was exactly the same as that discussed in the section on selection construct. In the first version in Figure 9, the code blocks are labeled as they were in Figure 8. However, the entire section from block D through block H may now be treated as a single code block because there are no further opportunities to simplify any code within that section.

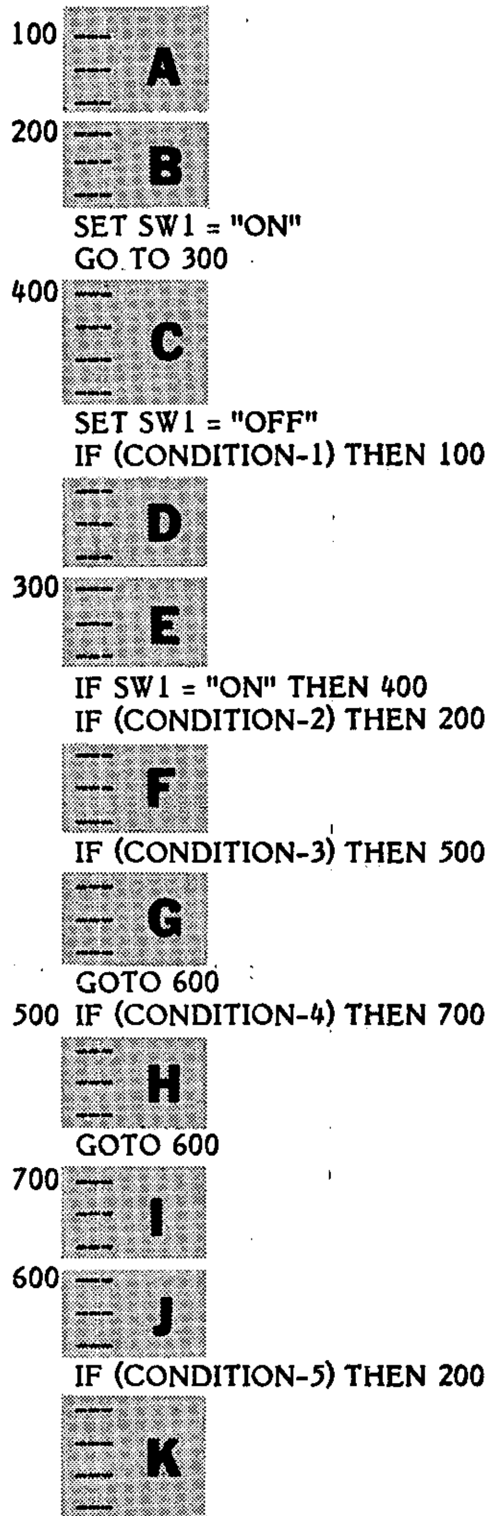

Figure 7

Spaghetti Code Program With Code Blocks Identified

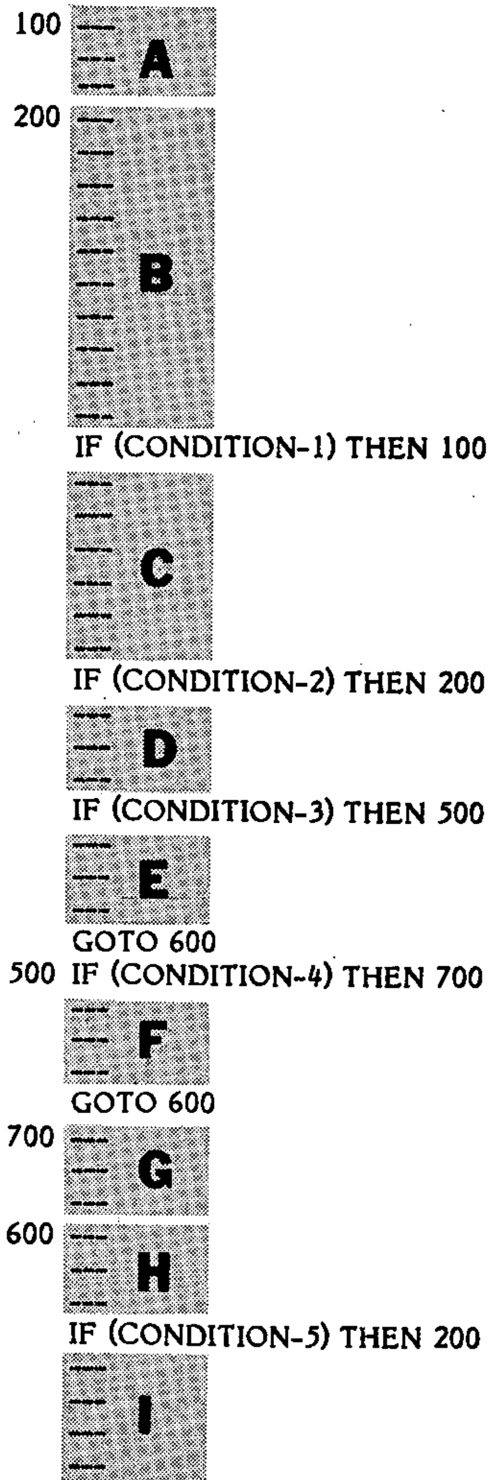

Figure 8

Code Block E Duplicated in Proper Sequential Position

Therefore, the second version of the problem in Figure 9 has compressed all of that code into a single code block and relabled the blocks. This leaves the problem ready for treatment of the looping constructs. Clearly, there are three loops in this code, all of them intersecting.

Since the first two steps of the loop restructuring process have already taken place (isolating the code and labeling the code blocks), we are now ready to represent the program as a simplified flowchart which shows only the looping structures. That flowchart is illustrated in Figure 10. Also in Figure 10, the parallel representation of the problem in pseudocode is shown. Obviously, the pseudocode solution is much longer and appears to be more complex than the flowchart. This is because of the limited representational ability of pseudocode. Because only sequence, selection, and iteration may be used, the complexity of the flowchart solution must be handled through an expanded use of a limited set of structures. However, opportunities for simplification exist in this pseudocode solution.

In Figure 10, two large code blocks exist. These blocks are exact duplicates of each other. Furthermore, the first block is performed and then the second block is immediately performed inside of a DOWHILE loop. The conversion of this structure into a DOUNTIL reduces the amount of pseudocode by approximately one-half. This reduction is shown in Figure 11. There, another set of common code blocks exist. Again, the blocks are duplicates with one performed just prior to the performance of the other inside of a DOWHILE. The second portion of Figure 11 shows the further reduction of the code which is possible as a result of this second set of duplicate blocks.

This completes the simplification of the pseudocode. Furthermore, no further restructuring of the program is necessary. The original code has been simplified, restructured, and clarified through the processes discussed in earlier sections. To clearly show the differences in the two versions, Figure 12 contains the original program, along with the final version. Note that the complexity of the original version is 10, while the complexity of the simplified version is reduced to 6.

USING THE RESTRUCTURED SOLUTION

The previous sections have detailed methods for understanding complex, unstructured code by restructuring it into structured pseudocode. The primary direction of the presentation has been to provide an aid to understanding code in the maintenance environment.

Once the code is re-represented, one must decide what to do with the simplified solution. In the most informal case, the programmer simply uses these techniques with a primary goal of understanding the code to support maintenance changes. In the most formal case, the maintenance programmer actually uses the simplified solution to rewrite the section of code of interest. Between these two extremes, there are other alternatives. First, the simplified representation may be added to the program documentation package to support future maintenance efforts. Second, the pseudocode may be added in comments just prior to the affected code segment. This saves the results of the understanding effort in the most usable location and makes them easily available to future maintenance personnel.

The maintenance programmer who uses these techniques and then saves the results, either by rewriting code or by formalizing the simplified solution into the documentation, benefits in two ways. First, the understanding of the existing program will take significantly less time with these methods. Second, if the results are saved, the understanding component will be reduced for all future maintenance efforts on that code section.

Summary

There is no argument as to the scope and importance of maintenance expenditures. Also, there is little doubt that much of the maintenance effort is spent on the understanding of code prior to debugging and modifying programs. Clearly, the understanding component of maintenance is a major target of opportunity for those seeking to reduce or control maintenance expenditures.

This paper strongly suggests that the understanding effort can be significantly reduced through the formalization of techniques which may be used in that effort. The approaches discussed here have been used successfully by a number of organizations in the public and private sectors with great success. The key to this approach lies in its ability to reduce the complexity of a code section through the creation of predictable code structures. Furthermore, the method can be applied to localized code sections when automated restructuring is unavailable or not desired.

In summary, complex code may be understood best by concentrating first on the sequential aspects of the program. Next, the selection constructs may be examined, particularly if the selection structures are implemented with conditional and unconditional branches instead of the IF-ELSE-ENDIF structures. Finally, the looping constructs may be simplified. With this set of procedures, each step yields reductions in control flow complexity and makes it possible for the next set of logical structures to be isolated and simplified.

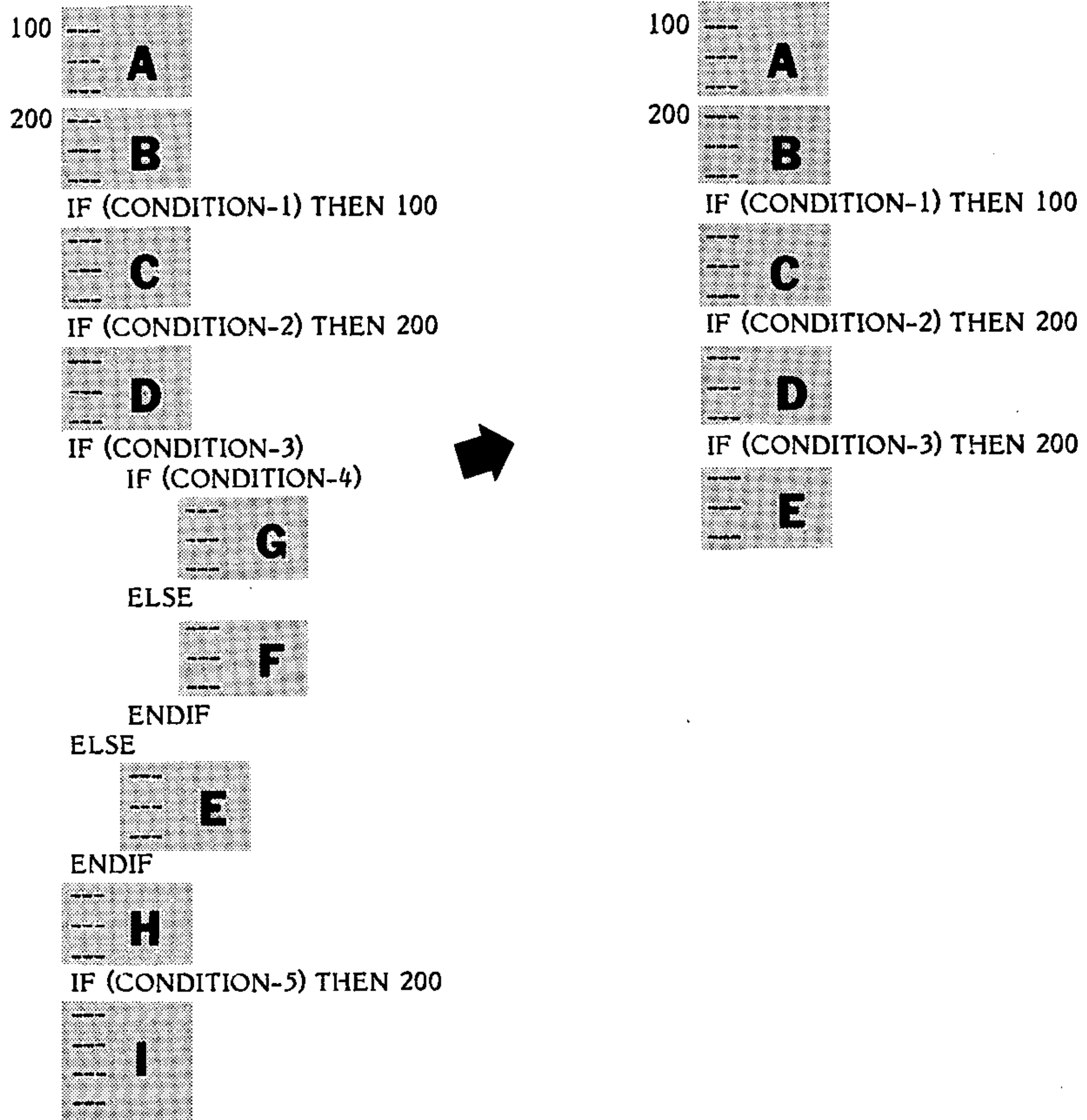

Figure 9

Selection Construct Structured and Blocked

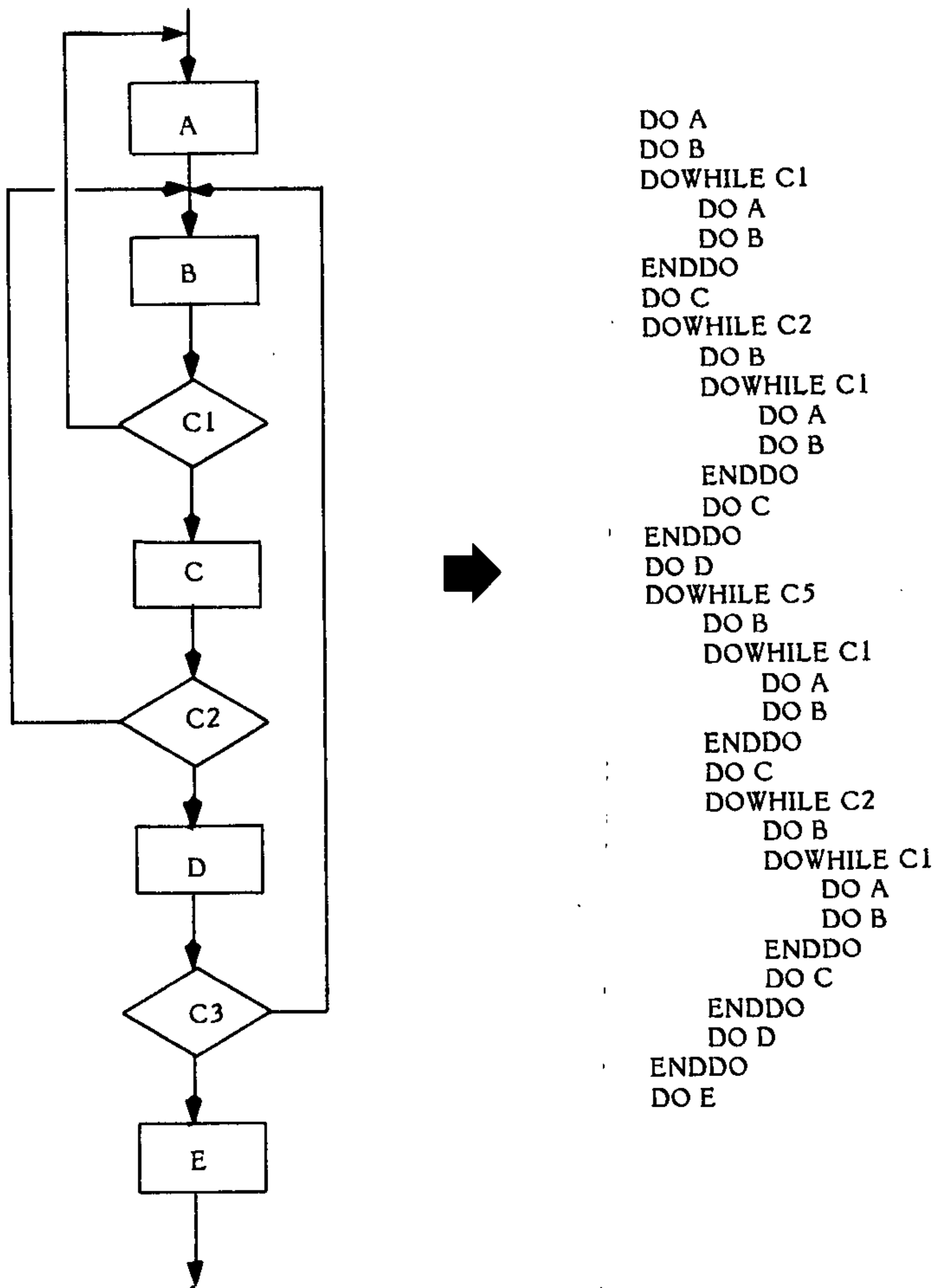

Figure 10

Flowchart Representation and Pseudocode Translation of Loops

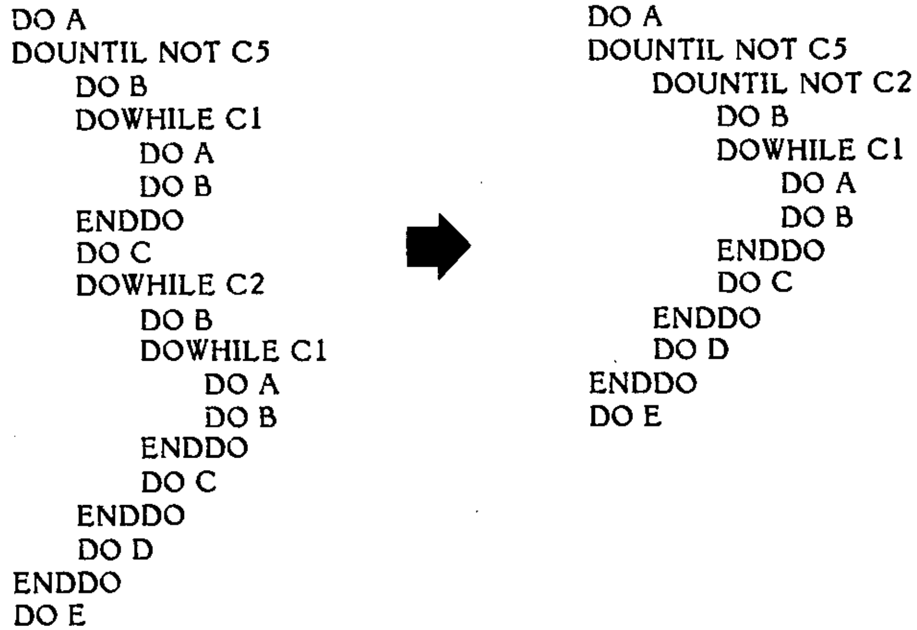

Figure 11

Final Contraction of the Structured Logic From the Spaghetti Code

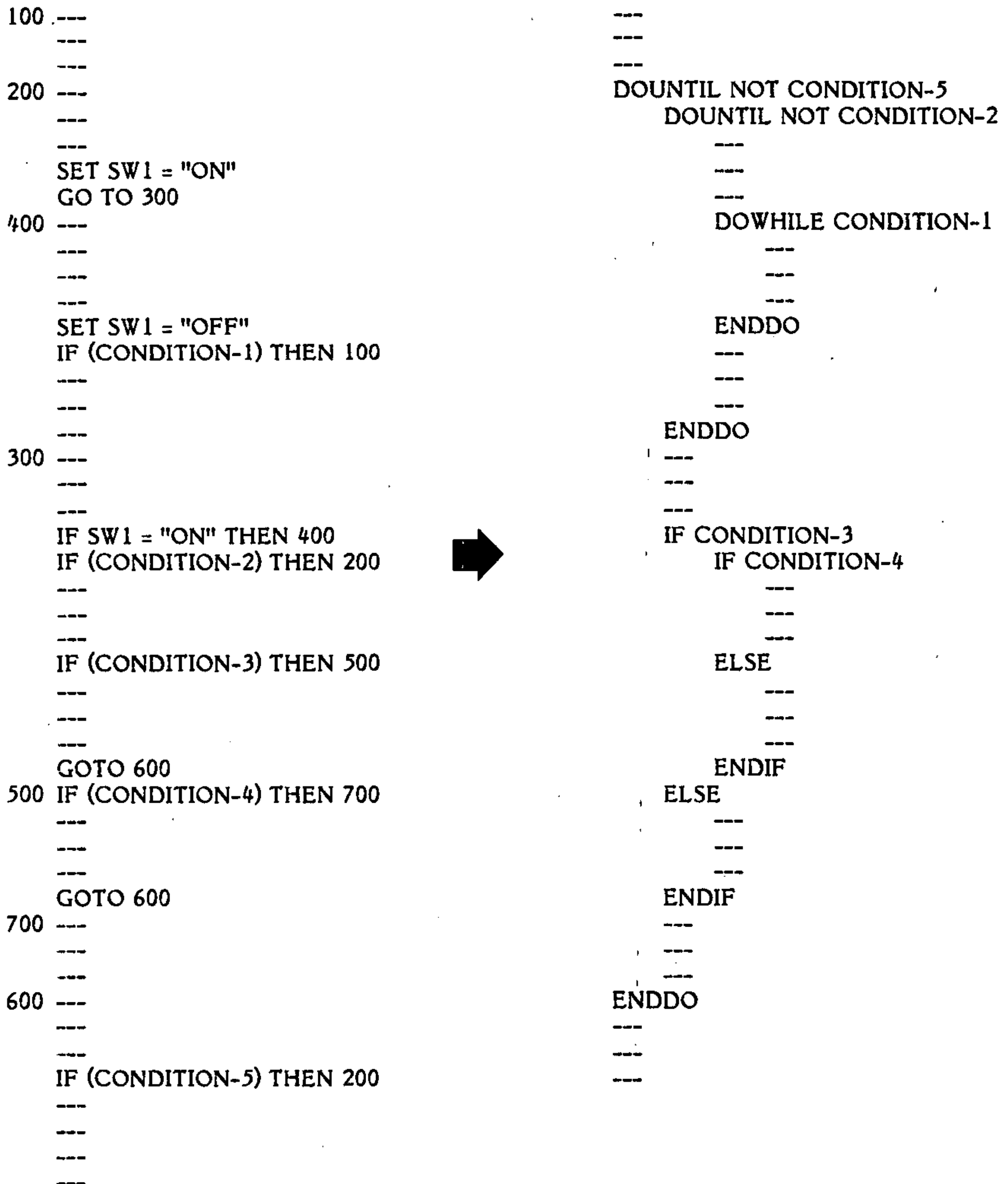

Figure 12

Original Spaghetti Code versus Restructured Version

REFERENCES

Ashcroft, E., and Manna, Z., "The Translation of 'GOTO' Programs to 'WHILE' Programs", *Proceedings of the 1971 IFIP Congress*, Ljubljana, Yugoslavia, August 1971, pp 250–255.

Bartol, K.M., "Turnover among DP personnel: a causal analysis", *Communications of the ACM*, Volume 26, Number 10 (October 1983), pp 807-811.

Berns, G.M., "Assessing Software Maintainability", *Communications of the ACM*, Volume 27, Number 1 (January 1984), pp 14-23.

Boehm, B.W. *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

Colter, M.A., and Couger, J.D., "Management and Employee Perceptions of the Maintenance Activity", *Proceedings of the Software Maintenance Workshop*, IEEE Computer Society Press, Silver Springs, Maryland, 1984, p 86.

Couger, J.D., and Colter, M.A., "The Effects of Maintenance Assignments on Goal Congruence for Programmers and Analysts", *Proceedings of the Fifth International Conference on Information Systems*, Tucson, Arizona, (November 1984), pp 83–100.

Couger, J.D., and Colter, M.A., *Maintenance Programming: Improved Productivity Through Motivation*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1985.

Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., and Love, T., "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics", *IEEE Transactions on Software Engineering*, SE-5,2 (March 1979), pp 96–104.

Elshoff, J.L., "An Analysis of Some Commercial PL/I Programs", *IEEE Transactions on Software Engineering*, SE-2, 2 (June 1976), pp 113–120.

Elshoff, J.L., "The Influence of Structured Programming on PL/I Program Profiles", *IEEE Transactions on Software Engineering*, SE-3, 5 (September 1977), pp 364–368.

Elshoff, J.L., and Marcotty, M., "On the Use of the Cyclomatic Number to Measure Program Complexity", *SIGPLAN Notices*, Volume 13, Number 12 (December 1978), pp 29–40.

Elshoff, J.L., and Marcotty, M., "Improving Computer Program Readability to Aid Modification", *Communications of the ACM*, Volume 25, Number 8 (August 1982), pp 512-521.

Gremillion, L.L., "Determinants of Program Repair Maintenance Requirements", *Communications of the ACM*, Volume 27, Number 8 (August 1984), pp 826–832.

Guimaraes, T., "Managing Application Program Maintenance Expenditures", *Communications of the ACM*, Volume 26, Number 10 (October 1983), pp 739–746.

Harrison, W., Magel, K., Kluczny, R., and DeKock, A., "Applying Software Complexity Metrics to Program Maintenance", *IEEE Computer*, Volume 15, Number 9 (September 1982), pp 65–79.

Lientz, B.P., Swanson, E.B., and Tompkins, G.E., "Characteristics of Application Software Maintenance", *Communications of the ACM*, Volume 21, Number 6 (July 1978) pp 466–471.

Lientz, B.P., and Swanson, E.B., *Software Maintenance Management*, Addison-Wesley, Reading, Mass., 1980.

Lientz, B.P., and Swanson, E.B., "Problems in Application Software Maintenance", *Communications of the ACM*, Volume 24, Number 11 (November 1981), pp 763–769.

Parikh, G., and Zvegintzov, N., *Tutorial on Software Maintenance*, IEEE Computer Society Press, Silver Spring, Maryland, 1983.

Vessy, I. and Weber, R., "Some Factors Affecting Program Repair Maintenance: An Empirical Study", *Communications of the ACM*, Volume 26, Number 2 (February 1983), pp 128–134.

Weiser, M., "Programmers Use Slices When Debugging", *Communications of the ACM*, Volume 25, Number 7 (July 1982), pp 446–452.