Improving Computer Program Readability to Aid Modification

James L. Elshoff and Michael Marcotty
General Motors Research Laboratories

SUMMARY: Frequently, when circumstances require that a computer program be modified, the program is found to be extremely difficult to read and understand. In this case a new step to make the program more readable should be added at the beginning of the software modification cycle. A small investment will make (1) the specifications for the modifications easier to write, (2) the estimate of the cost of the modifications more accurate, (3) the design for the modifications simpler, and (4) the implementation of the modifications less error-prone.

1. The Modification Cycle

The modification of computer programs is a costly and constant job. An informal survey conducted at General Motors and reported on by Elshoff [6] concluded that about 75 percent of all programmer/analysts' time in a commercial data processing installation is spent on program modification. This conclusion agrees with independent assessments made by Liu [16], Boehm [2], and Lientz and Swanson [15]. Moreover, the reasons for modifying programs will not disappear. As pointed out by Lehman [14], all programs are models of some part of the real world and, as the world changes, programs must be modified to keep pace with these changes or they become progressively less relevant, less useful, and less cost-effective. As new software is developed, the inventory of programs to be maintained grows, and thus this high level of modification work is not expected to decrease.

The modification cycle is composed of a sequence of steps such as:

(1) The user requests that a program be changed.
(2) The specifications for the change are written and the cost of the change estimated.
(3) It is decided that the changes are worth being made.
(4) The program is changed to meet the new specifications.

Unfortunately, the modification environment is not as simple as this list. The frequency of change, the extent of a change, the acceptable cost for a change, and other change attributes vary with the individual program. The one common denominator of the modification process is that it starts with an existing program and its documentation. In most cases this means a listing of the program's source text. The readability of that source text can have a great impact on the decisions made during the modification cycle.

2. Unreadable Programs

In a study of commercial programming practices by Elshoff [6], it was found that most programs were poorly written. They were very large, extremely difficult to read, and more complex than necessary. Furthermore, the study determined that programming language usage was poor and inconsistent. The results of the survey by Lientz and Swanson [15] show that the quality of programming is a generally perceived problem.

During the last five years and continuing today, there has been a major effort in data processing installations

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.7 [**Software Engineering**]: Distribution and Maintenance–*documentation, enhancement, and restructuring.*
General Terms: Documentation, Human Factors, Languages
Additional Key Words and Phrases: software modification cycle.
Authors' present address: J.L. Elshoff and M. Marcotty, Computer Science Department, General Motors Research Laboratories, Warren, MI 48090-9055.

to improve programming practices. Programmer training and installation procedures are being upgraded through the use of better practices as described by Kernighan and Plauger [13], and Elshoff [4, 5]. The improvements achieved with better practices have been shown by Elshoff [7] to be significant and are supported by the experimental evidence of Sheppard et al. [21].

Nevertheless, most data processing installations still have large inventories of programs that are nearly impossible to read. Programs from this inventory must regularly be modified or replaced. Before this can be done, it is first necessary to understand exactly what that program currently does. In fact, the very decision whether to modify or completely replace a program may hinge on how well the program is understood. The need for readability is apparent and imperative in a communication medium like the source text of a computer program. The life of the program depends on it.

The thesis of this paper is that modifying a program simply to improve its readability is generally a worthwhile endeavor. With proper timing, the improvements in readability can be achieved at little or no cost. Furthermore, once the program is readable, the advantages of improved readability will accrue with each subsequent modification. Here, we present a method for improving the readability of a program through a set of specific transformations that can be applied directly to the program text. The effects of applying the transformations to a sample program are shown and discussed.

3. Readability

The readability of a computer program depends on many factors. The reader's familiarity with the program, knowledge of the application area, and own programming style are important factors that are mostly independent of the program to be modified. In this paper, we concentrate on those attributes of the program's text that impact its readability. Thus, we will take the pragmatic, realistic point of view of a programmer who is knowledgeable in the application area but who is seeing a particular program for the first time.

A readable program always seems to exhibit a common set of properties, as listed, for example, by Kernighan and Plauger [13], Yourdon [24], and Myers [19]. The program is well commented. The logical structure of the program is constructed of single-entry single-exit flow of control units. Variable names are mnemonic and references to them localized. The program's physical layout makes the salient features of the algorithm that is implemented stand out. It is true that a program may have all these properties and still be unreadable; however, the readability of a program is certain to suffer when it lacks one or more of the properties.

4. Program Transformations

There are many known source program transformations described by Kernighan and Plauger [13], and Standish et al. [22]. Algorithms have even been developed to perform the complete restructuring of programs; these are described by Mills [18] and Ashcroft and Manna [1], and have also been implemented in computer programs. However, as Dijkstra pointed out in 1968 [3],

> The exercise to translate an arbitrary flow diagram more or less mechanically to a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

This has been borne out in actual examples—for instance, Elshoff and Marcotty [8].

Our own experience with the manual restructuring of PL/I programs indicates that the use of the set of transformations listed in the next section is a key to making programs more readable. We have found that the actual text manipulation gives the programmer an increased understanding of the program and insights for further modifications. 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.

All the transformations described in the next section aim to simplify the program by modifying the executable statements and rearranging the sequence in which they are executed. As a result of these changes, the program may need to be reformatted and additional comments added. These operations are really program transformations that enhance readability without altering the program's execution and are discussed in this section. Reformatting and commenting should be done for each pass over the source text. As understanding increases, the programmer will be able to add more meaningful comments.

4.1 Add Comments

Programmers consistently state that few programs have documentation outside of the source text. Moreover, when there is external documentation, it is most frequently no longer in step with the program text. Since the source text represents reality, the final authority on what is executed, it should be self-documenting, which means it must be readable.

Comments should be used to make the source text of a program understandable. Block comments should be placed at the beginning of a program to describe the program's purpose, external interface, and how it works. The program should be divided into major sections, paragraphs, separated by blank lines or page boundaries. Block comments should also be used to describe the functions performed by the paragraphs.

Comments can be the most important contribution that a programmer makes. The programmer modifying a program must be able to read and understand it even though it is difficult. This difficulty can be reduced for all future modifications by adding appropriate comments as discoveries about the program are made. Surprisingly, adding comments is often one of the last tasks that can be done; the programmer just cannot understand the program text well enough to add comments early on.

4.2 Reformat

Maintaining a consistent format adds greatly to the readability of a program. Just as paragraphing and sectioning help written English, so can indentation, key word positioning, and logical grouping aid a programming langauge. Using an automatic formatter such as the one on the IBM PL/I Checkout Compiler [12] can standardize style for an installation. However, even when the reformatting must be done by hand, it should be done consistently. Consistency of style is more important than the details of the style itself. The few extra minutes the programmer spends keeping a program consistently formatted will pay dividends the next time the program is read.

5. Readability Transformations

In this section, we describe a set of simple changes that can be made to a program to improve its readability. A programmer using a good editor can quickly apply these transformations. Where sample program text is provided as an illustration, the PL/I programming language is used. However, most of the transformations described have direct analogies in other programming languages. Some of the programming examples are accompanied by simple flowgraphs with the convention that at branch points, the true branch is *always* to the left.

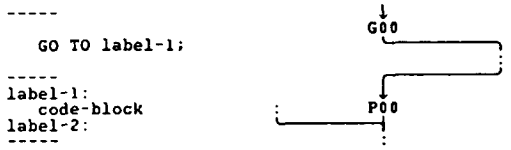
The transformations are presented in approximately the order they will be applied, although the specific ordering will vary from program to program. Moving labeled blocks and adding ELSE clauses are easy transformations to apply and should be done early on. Frequently, the application of one transformation will change the pattern of the program text so that additional transformations may be applied. The recommended approach is to read the source code, apply a set of straightforward and obvious transformations, add comments, and readjust the indentation.

Since the programmer may make a mistake while applying a transformation, a policy of checking the program after each pass is recommended. The first simple check is to compile the program. The compiler will check the syntactical correctness of the program and produce a symbol table that can be easily compared with the symbol table produced for the preceding pass. A second check is to execute the program against a set of test data. The idea behind this testing is not to check all possible paths but to simply check the repeatability of results. An execution test can prevent an error in an early pass from being compounded in succeeding passes.

The modified program should then be reread to find the next set of transformations to apply. The process is thus an iterative one with the program's readability and the programmer's understanding increasing simultaneously. Depending on the size of the program and its unreadability, the number of passes will vary, but sooner or later the mainline of the program will begin to become obvious and the program will be understood by the programmer.

5.1 Move Single Entry Labeled Blocks

A structure frequently found in an unreadable program is the single-entry labeled block, called code-block. This consists of a sequence of statements that may only be entered at the first statement and, when executed, will be executed to the last statement without any other possible exit.



A quick check of a program's symbol table can usually be used to find labels that are only referenced once. After the programmer verifies that the code-block cannot be reached by normal sequential execution, the code-block is moved to its proper location.

There are many minor variations of this change. Often, the code-block must be embedded in statement blocking symbols such as the DO-END statements in PL/I. Frequently, the code-block ends with a GO TO statement and the additional GO TO label-2 statement is unnecessary. In any case, this modification removes a label and relocates a code-block physically closer to the decisions governing its invocation.

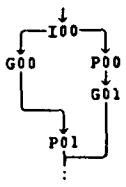
5.2 Duplicate Labeled Blocks

This transformation is directly analogous to the previous one except that the label on the block, i.e., label-1, is referenced more than once. When this is the case and the code-block is small (say, less than 10 statements), the code-block is simply duplicated at each of the locations where a GO TO label-1 statement occurs. If the code-block is large or invoked many times, consideration might be given to making it into a procedure, as described in a succeeding section. However, at this stage in the transformation process, we are expanding text in order to gain understanding. The fact that a sequence of code is repeated several times does not necessarily mean that it would be wise to make it into a procedure; the function that it performs must instead be considered. This usually cannot be done until understanding is reached.

5.3 Add ELSE Clauses

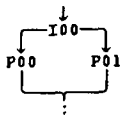
The addition of an ELSE clause to every IF statement clarifies a program immensely. In the simplest case, the programmer walks through the program finding each IF statement that has no ELSE clause and adds one with a null statement. The null ELSE clause is a construct that many programmers view as a waste of time. It takes a second to write, has no effect on a program's compilation or execution, and can save a reader hours of effort by making a program more explicit and thus easier to read. The presence of the ELSE clause on all IF statements resolves any ambiguity that might be present in the reader's mind because of the optional nature of the ELSE clause. The structure

```
-----
   IF test
   THEN GO TO label-1;
   code-block-a
   GO TO label-2;

label-1:
   code-block-b

label-2
-----
```



is not uncommon. Its readability can be improved by making the relationship of the code-blocks to the IF test explicit in terms of THEN and ELSE clauses.

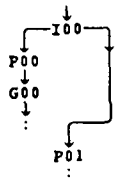
```
-----
   IF test
   THEN
     DO;
       code-block-b
     END;
   ELSE
     DO;
       code-block-a
     END;
-----
```



5.4 Renest IF Statements

After null ELSE clauses, as suggested in the previous section, have been inserted, it will become obvious in many instances that the ELSE clause is not really null. The pattern

```
-----
IF test
THEN
  DO;
    code-block-a
    GO TO label-1;
  END;
ELSE
  ;
code-block-b
-----
```

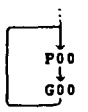


is found in the program such that code-block-b is really the ELSE clause but is not packaged that way. Eliminating the null statement and putting code-block-b in a DO-END group make the program text more obvious. This change has the additional benefit of increasing the probability that the GO TO label-1 statement can be easily removed.

5.5 Make Loops Obvious

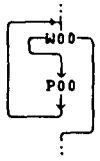
Using a GO TO statement to implement a loop greatly obscures a program. The program segment

```
-----
label-1:
   code-block

   GO TO label-1;
-----
```



in which the code-block may be from one to several hundred statements long is not unusual. The problem is that the programmer reads the source text from top to bottom and does not realize the code-block is a loop body until the GO TO statement is read. Simply replacing the label and the GO TO with a DO WHILE as in

```
-----
DECLARE NEVER_ENDING   BIT(1) STATIC INITIAL('1'B);
-----
DO WHILE (NEVER_ENDING);
   code-block
END;
-----
```



establishes the fact that the program contains a loop structure at this point. This modification also alerts the reader to the existence of a loop whose termination condition is not yet understood, as will be described in the next section.

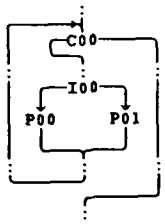
Experience has shown that making more than one of these modifications during a pass can sometimes result in intersecting loops. When this occurs, either the modification of one of the GO TO loops will have to be delayed until a subsequent pass or some sub-code-blocks will have to be interchanged.

5.6 Make Loop Termination Explicit

As discussed by Gries [10], one of the hardest programming constructs to understand is the loop. This difficulty is increased considerably when the conditions for terminating the loop are not explicit. This can arise when the loop itself is hidden, as discussed in the previous section. Another common fault is to use an iterative loop when it is not an intrinsic part of the process being performed. The third method is to use a LEAVE or GO TO statement to branch out of the loop, as will be discussed in the next section. The basic problem is that the reader cannot determine from the statement at the head of the loop the exact conditions that will cause termination of the loop and thus cannot determine the real reason for the loop.

Using an iterative loop when it does not apply, as in

```
-----
DO I = 1 TO 9999;
  -----
  IF test
  THEN I = 10000;
  ELSE
    ;
  -----
END;
-----
```



is one example of a misleading loop termination. The programmer probably used the wrong form of the DO statement. If the index I is not used anywhere else in the body of the loop, a simple DO WHILE should have been used, as in the example

```
-----
DECLARE CONTINUE_LOOP   BIT(1),
        YES             BIT(1) STATIC INITIAL('1'B),
        NO              BIT(1) STATIC INITIAL('0'B);
-----
CONTINUE_LOOP = YES;
DO WHILE (CONTINUE_LOOP);
   -----
  IF test
  THEN CONTINUE_LOOP = NO;
  ELSE
     ;
  -----
END;
-----
```

to clarify the loop termination condition. The selection of the name for the loop control variable, CONTINUE_LOOP above, can greatly improve the structure's readability. A name that makes the DO WHILE read in a straightforward manner, such as

```
DO WHILE (NOT_END_OF_FILE_A);
DO WHILE (OUTSIDE_ERROR_BOUNDS);
DO WHILE (MORE_CHARACTERS_IN_STRING);
```

should be used. When the programmer really understands the loop, the termination condition is obvious and the selection of a variable name follows naturally.

If the index I is referenced within the loop, the programmer can choose one of two ways to make the loop termination explicit. The variable I can be explicitly controlled by initializing it before entering the loop and incrementing it within the loop, or a more complicated form of the DO statement

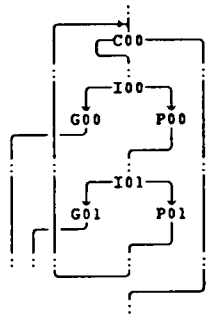
```
DO I = 1 TO MAX WHILE (NOT_FOUND);
```

can be used. The latter approach should only be used when the loop may be terminated by either the indexing condition or the WHILE condition. This is a form of multiple loop termination that is examined more closely in the following section.

5.7 Remove Multiple Exits from Loops

It is not unusual to find a loop with more than one exit. In addition to the normal loop termination, the loop may be exited with a LEAVE statement, a GO TO statement, or an exception condition that is trapped by an ON unit. In order to change the multiple exit loop

```
-----
DO I = 1 TO 9999;
   -----
   IF test-1
   THEN GO TO label-1;
   ELSE
      ;
   -----
   IF test-2
   THEN GO TO label-2;
   ELSE
      ;
   -----
END;
-----
```

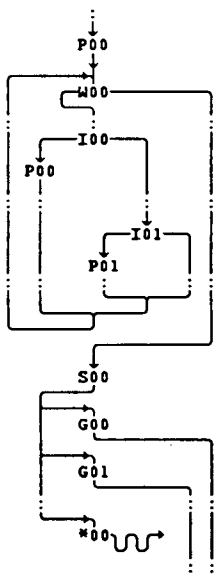


to a single exit loop, the WHILE clause must usually be made into a compound conditional like

```
DO WHILE (NOT_END_OF_FILE_A & NO_ERROR_ENCOUNTERED);
```

using techniques like those discussed in the two previous sections. In some tougher cases, the introduction of a variable may be required. A SELECT statement or a nest of IF statements can then be used to maintain the proper logical flow. For example, the variable STATE could be used to modify the code above to

```
-----
DECLARE STATE   CHARACTER(8);
-----
STATE = 'NORMAL';

DO WHILE (STATE = 'NORMAL');

   -----
   IF test-1
   THEN STATE = 'label-1';
   ELSE
     DO;
        -----
        IF test-2
        THEN STATE = 'label-2';
        ELSE
          DO;
             -----
          END;
     END;

END;

SELECT (STATE);

   WHEN ('label-1')
      GO TO label-1;

   WHEN ('label-2')
      GO TO label-2;

   -----
   OTHERWISE /* should not occur */
      SIGNAL ERROR;
END;
-----
```



A proper choice of names for the values assigned to STATE can further increase the readability of the program. Limited experience with this tougher case has shown that moving from loops with many exits to single-exit loops greatly clarifies the program text even though a multiple-exit SELECT structure is introduced. In fact, in all observed cases, the multiple-exit SELECT structure was quite easily transformed to a single-exit structure in subsequent passes over the program.

5.8 Remove Label Variables Used for Blocking

Label variables are used occasionally to simulate internal, nonparameterized procedures. For example, the code sequence

```
-----
   label-variable = label-1;
   GO TO label-2;
label-1:
-----
label-2:
   code-block
   GO TO label-variable;
-----
```

sends control to the code-block at label-2 and then returns control to the next sequential statement following label-1. Either the code-block should be made into a PROCEDURE that is called, or the code block should be distributed throughout the program. In either case, the labels, the GO TOs, and the label-variable with its multiway branch are removed and the program becomes more readable in a top-to-bottom fashion.

5.9 Remove Label Variables Used as Memory

Another common use for label variables is to remember a particular decision or path in a program by assigning a label to a label variable. In the code sequence

```
-----
   label-variable = label-1;
-----
   label-variable = label-2;
-----
   label-variable = label-3;
-----
   GO TO label-variable;
-----
```

for example, the label-variable is used to remember which of three different paths was last executed in order to determine which path of the multiway branch is taken. The modification suggested in this case is the same as that recommended for the more difficult multiple-exit loops. Use a state variable as the memory device instead of a label variable. The resulting source text

```
-----
DECLARE STATE   CHARACTER(8) STATIC INITIAL('UNKNOWN');
-----
   STATE = 'label-1';
-----
   STATE = 'label-2';
-----
   STATE = 'label-3';
-----
SELECT (STATE);
WHEN ('label-1') GO TO label-1;
WHEN ('label-2') GO TO label-2;
WHEN ('label-3') GO TO label-3;
OTHERWISE SIGNAL ERROR; /* should not occur */
END;
-----
```

may even appear slightly more complex initially. However, as with multiple-exit loops, experience has shown that removing label-variables is necessary to clarify the program text so that the multiple-exit SELECT structure can, in turn, be changed to a single-exit structure by applying other simple transformations within each WHEN clause. Although we appear to be swapping one kind of memory for another, this form makes the program easier to read and has the added advantage that its value can be printed for debugging purposes.

5.10 Use Status Variables to Track Execution

A frequently used programming form that contributes to unreadability is the use of long branches to a label that does standard error processing. Whether long branches are for error handling or other purposes, the introduction of a status variable is recommended to eliminate the branches and the resulting multiple-exit, multiple-entry code. As with the examples for multiple-exit loops and label variables, a character string variable is declared. Set the variable to 'NORMAL' and in the event an error is uncovered, set the variable to a value indicating the nature of the error. The variable can then be tested at the beginning of each major functional block within the program to determine whether that function should be performed or bypassed. The program text has a form like

```
-----
IF STATUS = 'NORMAL'
THEN DO;
        major-function-1
     END;
ELSE ;
IF STATUS = 'NORMAL'
THEN DO;
        major-function-2
     END;
ELSE ;
-----
```

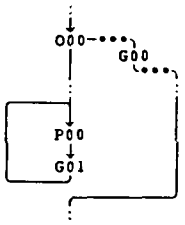
and execution proceeds through the functions as long as everything is normal.

The judicious selection of the character string values assigned to a status variable can also make the program clearer by making it more self-documenting. A simple method is to maintain a block comment with the declaration for the status variable that indicates all of the values the status variable may have and what each value means.

5.11 Use Switches in ON-Units

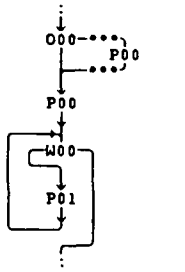
The use of switches in ON-units can eliminate an excess of branching. In particular, switches should be used to control the program flow for conditions such as the end of file. A typical code sequence

```
-----
ON ENDFILE(file-a)
   GO TO label-1;
-----
label-2:
   READ FILE(file-a) ... ;
   GO TO label-2;
label-1;
-----
```



can be transformed to a sequence of single-entry, single-exit control structures like

```
-----
DECLARE MORE_RECORDS    BIT(1),
        YES             BIT(1) STATIC INITIAL('1'B),
        NO              BIT(1) STATIC INITIAL('0'B);
-----
ON ENDFILE(file-a)
   MORE_RECORDS = NO;

MORE_RECORDS = YES;
DO WHILE (MORE_RECORDS);
   READ FILE(file-a) ... ;
   ----
END;
----
```



with the addition of a switch.

5.12 Localize References

The transformation implied here is to move statements around so that the references to a single variable or name are close together. The use of the file constant, FILE_A, in the source text

```
-----
ON ENDFILE(FILE_A)
   MORE_RECORDS = NO;
-----
many-statements
-----
OPEN FILE_A;
-----
many-statements
-----
MORE_RECORDS = YES;
DO WHILE (MORE_RECORDS);
   READ FILE(FILE_A) ... ;
END;
-----
many-statements
-----
CLOSE FILE_A;
-----
```

is not uncommon. However, there is no rule that ON-units and OPEN statements must come first and CLOSE statements last in a program. Since the association between an input statement and its corresponding ON ENDFILE statement is implicit, putting the two close together, as was done in the previous section, makes this association more obvious. Localizing the uses of the name FILE_A,

```
-----
OPEN FILE_A;
ON ENDFILE(FILE_A)
   MORE_RECORDS = NO;
MORE_RECORDS = YES;
DO WHILE (MORE_RECORDS);
   READ FILE(FILE_A) ... ;
END;
CLOSE FILE_A;
-----
```

means the reader does not have to keep details of that file in mind while reading other parts of the program. Moreover, if the reader is particularly interested in FILE_A, its uses are not spread all over the program. A pleasant side-effect of localizing references is that the execution efficiency of a program may be improved due to reduced paging.

5.13 Extract Common Code Sequences

The final area to be discussed is the extraction of common code sequences into procedures. Common code sequences may be labeled blocks that are either too large or too frequently referenced to be distributed throughout the program, as discussed earlier. They may be labeled blocks that are terminated by GO TO label variables, as discussed earlier. They may just be duplicate blocks of code that the reader discovers in the code. Finally, a common code sequence may simply be a single-entry, single-exit, functional block of code, in which case the extraction of the code block will make the main program easier to comprehend merely by making it smaller.

Just because a large block of code happens to appear many times is not sufficient grounds for making it into a procedure. In order to be of help in the readability and subsequent modifiability of the program, procedures should be constructed so that they each perform a specific logically self-contained task. The fact that an identical sequence of instructions happens to occur repeatedly does not mean that those instructions perform a cohesive task. Guidelines for recognizing and organizing code into functional procedures are described by Myers [19] and Stevens [23].

Once a sequence of instructions has been identified as suitable for transformation into a procedure, a simple method can be followed:

(1) Remove the common code sequence from the main program and wrap a set of PROCEDURE-END statements around it.
(2) Replace each reference to the common code sequence by a CALL statement in the main program.
(3) Recompile the main program and compile the common code sequence.
(4) Determine parameters by finding symbols common to both programs.
(5) Determine local variables for the common code sequence by finding symbols no longer referenced in the main program.
(6) Add a declaration for the new procedure in the main procedure. Update all CALL statements for the new procedure to use a proper argument list.
(7) Add a block comment to the new procedure describing its purpose and use.
(8) Move declarations for local variables into the new procedure.

6. Experience with the Transformations

Rather strange sections of source text may arise while transforming a program for readability. Some examples encountered in the past are (1) program text that cannot be reached via any execution path, (2) branches into the middle of loops, and even (3) an IF statement with identical code in its THEN and ELSE clauses. The wise programmer will go back and check the original program text when an odd section of code arises to make sure that a transformation has not been improperly applied, but the programmer will usually discover that the oddity really exists.

Often, the programmer will find other transformations to apply to make a program more concise and more readable. For example, statements common to both the THEN and ELSE clauses of an IF statement or common to all clauses of a SELECT statement can frequently be extracted to either immediately precede or follow the IF or SELECT statement. The programmer should make modifications whenever the readability of the program can be enhanced.

7. An Example Program

In this section the application of readability transformations to a production program is discussed. The particular program used here was selected because it was the smallest nontrivial program in a set of commercial data processing programs. The program turned out to be particularly unreadable. Multiple applications of every transformation mentioned above were used to make the program more readable. The program was modified in 13 separate passes. It began as a single PL/I procedure, P1, and finished as a program, P2, comprising six procedures, M1 through M6, with M1 identifying the residual mainline procedure. Table I shows some of the two program's basic statistical properties.

Although the lines of source text have increased by more than half, the program is actually smaller in many respects. Duplicate code in the form of header blocks of

Table I. Comparison of Basic Statistical Properties.

Property	P1	P2	M1	M2	M3	M4	M5	M6
Lines of source text	597	916	431	160	117	94	49	65
Identifiers	218	274	132	22	42	33	19	26
Non-DECLARE statements	270	336	176	83	29	23	8	17
CALL statements	3	48	25	12	5	2	1	3
Assignment statements	125	92	48	18	9	5	4	8
IF statements	24	39	21	11	3	3	0	0
DO statements	4	47	23	16	4	3	0	1
GO TO statements	80	0	0	0	0	0	0	0

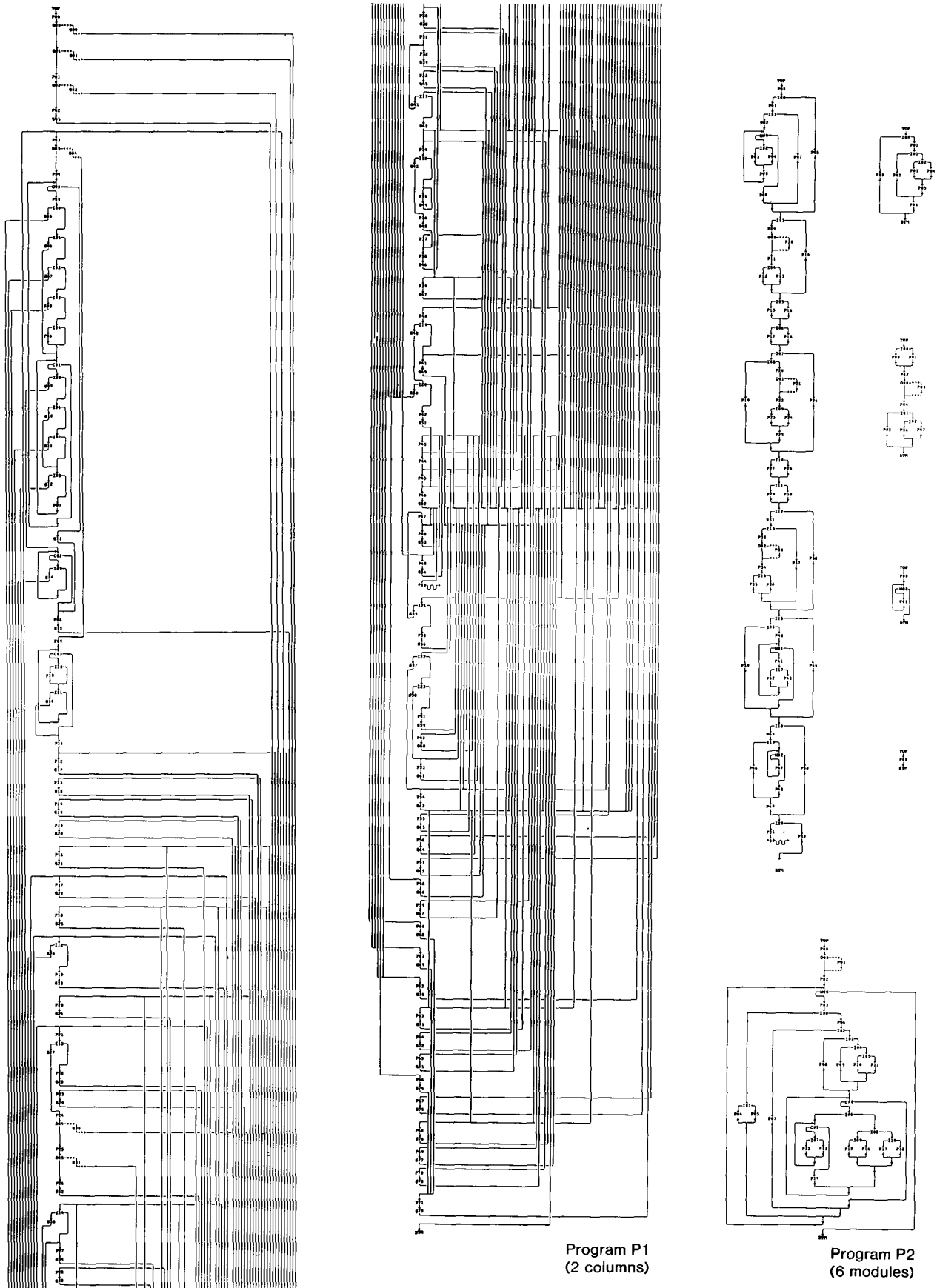


Fig. 1. Logical Control Flows of Two Versions of a Program.

comments and declarations accounts for most of the increase. Duplicate declarative information also accounts for the increase in identifiers. The extraction of subprocedures increases the number of CALL statements, while reducing the numbers of the type of statements extracted, such as assignment statements. All 80 GO TO statements were eliminated from the original program. The introduction of status variables and their testing account for the increase in IF statements. Introducing ELSE clauses and logically grouping blocks of code account for most of the increase in DO statements and a large increase in the total number of lines of text and total statements; 40 grouping DO-END blocks now exist in a program that started with none.

The load module increased in size from 8,800 bytes for P1 to 13,400 bytes for P2. This increase is due mostly to the additional prologue and epilogue code generated for the subprocedures. Execution measurements were not done for this program, but experience with other programs has shown that an improvement of 5 to 10 percent is not unusual.

The decrease in complexity of the logical flow was monitored as the program was modified. The cyclomatic complexity measure, introduced by McCabe [17] and discussed by Elshoff and Marcotty [8] and Myers [20], associated with the number of testable paths in the program was used. The results shown in Table II indicate that the complexity of the program was reduced by more than 40 with respect to its flow of control. The logical flow of the program was also mapped using the same conventions as were used in the flowgraphs illustrating the program samples earlier in this paper. Figure 1 is a photo-reduced picture of the logical flow of control for the program's two versions. The reader can readily observe the difference.

An experimental measure of program clarity described by Gordon [9] was also applied to the program after each pass. This clarity measure theoretically determines the effort required to understand the program. With a factor of 43,200 (12 units/second, as suggested by Halstead [11]) used to convert the effort units to hours, the results are listed in Table III. Although this measurement indicated that the program grew slightly more complicated after a few initial passes, the end-result of applying the readability transformations represents a large reduction in the estimated effort required to understand the program.

Although the clarity measure has not been validated and must be treated as an average for any programmer, the relative difference seems to understate the case for the readable version of the program. In our opinion, the original program could not be fully understood in 24 hours. On the other hand, the program module M1, because of its use of status variables and the similarity of several sections of source text, should not require 8 hours to understand. The 13 passes to improve the program's readability required 16 hours to complete. A single pass took from 15 minutes to two hours. Thus, if we view the clarity measure as an absolute value, the total time to understand this program was increased by about four hours, a small amount of time that should easily be recouped when the program is modified. Real net benefits should then accrue on all subsequent modifications since the program will be more readable from the start.

8. Recommendation—A New Modification Cycle

A new step should be added to the modification cycle: modifying the program to make it readable. If the new program is judged to be already readable, this new step may be skipped. However, when the program is judged to be difficult to read, readability transformations should be applied to make it more readable.

The time to make a program readable is at the beginning of the modification cycle. The small investment will start paying dividend by making (1) the specifications for the modifications easier to write, (2) the estimate of the cost of the modifications more accurate, (3) the design for the modifications simpler, and (4) the implementation of the modifications less error-prone. Once the program is made readable, these benefits should apply to all future modifications as well. In fact, doing a better job on one modification cycle may eliminate the need for some future cycles.

Table II. Comparison of the Flow of Control.

Program	Cyclomatic complexity
P1 (before)	91
P2 (after)	52
M1	25
M2	16
M3	4
M4	4
M5	1
M6	2

Table III. Clarity—Time to Understand the Program.

Program	Time (hours)
P1 (before)	23.6
P2 (after)	11.4
M1	7.5
M2	2.8
M3	0.3
M4	0.4
M5	0.1
M6	0.3

The effective application of good programming practices to new program development and the application of readability transformations during the modification cycle should eventually result in an inventory of readable programs. However, until all the programs in an installation's inventory are readable, the modification cycle introduced in the first section of this report should be changed to the five-step cycle listed below, where step 2 has been inserted.

(1) The user requests that a program be changed.
(2) The source text of the program is made readable.
(3) The specifications for the change are written and the cost of the change estimated.
(4) It is decided that the changes are worth being made.
(5) The program is changed to meet the new specifications.

References

1. Ashcroft, E., and Manna, Z. The translation of 'GOTO' programs to 'WHILE' programs. Proc. 1971 IFIP Congress, Ljubljana, Yugoslavia, Aug. 1971, pp. 250–255. Demonstrates that every flowchart program can be written without GO TO statements by using WHILE statements.

2. Boehm, B. Software engineering. *IEEE Trans. Comptrs. C-25*, 12 (Dec. 1976), 1226–1241. Provides a definition of the term "software engineering" and a survey of the state of the art of software production in 1976. Contains an extensive set of references.

3. Dijkstra, E.W. GO TO statement considered harmful. *Comm. ACM 11*, 3 (March), 147–148. This famous letter contends that the quality of programmers is a decreasing function of the density of GO TOs in the programs they produce, and advocates the abolition of the GO TO from high-level languages because it is too primitive a construct.

4. Elshoff, J.L. A case study of experiences with top down design and structured programming. GMR-1742, Comptr. Sci. Dept., General Motors Res. Labs., Warren, Mich., Oct. 1974. Describes the author's personal experiences in consciously using top down development techniques and structured programming techniques.

5. Elshoff, J.L. Defensive programming. GMR-1799, Comptr. Sci. Dept., General Motors Res. Labs., Warren, Mich., Feb. 1975. Presents a description of the techniques of defensive programming and some of the trade-offs that should be considered by programmers using them.

6. Elshoff, J.L. An analysis of some commerical PL/I programs. *IEEE Trans. Software Eng. SE-2*, 2 (June 1976), 113–120. Presents the results of studying 120 commercial PL/I programs with respect to their size, readability, complexity, programming discipline, and use of programming language.

7. Elshoff, J.L. The influence of structured programming on PL/I program profiles. *IEEE Trans. Software Eng. SE-3*, 5 (Sept. 1977), 364–368. Studies two sets of commercial PL/I programs representing programming practice before and after the introduction of structured programming techniques.

8. Elshoff, J.L., and Marcotty, M. On the use of the cyclomatic number to measure program complexity. *SIGPLAN Notices 13*, 12 (Dec. 1978), 29–40. Further discussion of the cyclomatic complexity measure of McCabe [17] and its extension by Myers [20].

9. Gordon, R.D. Measuring improvements in program clarity. *IEEE Trans. Software Eng. SE-5*, 2 (March 1979), 79–90. A functional relation between the clarity of a program and the number and frequency of operators and operands in the program is presented. This measure of program clarity gives an estimate of the amount of mental effort required to understand the program.

10. Gries, D. *The Science of Programming.* Springer-Verlag, New York, 1981. Describes with many examples and exercises the basic principles behind the construction of programs that can be demonstrated to be correct through reasoning.

11. Halstead, M.H. *Elements of Software Science.* Elsevier North-Holland, New York, 1977. Halstead investigates the natural laws that govern the construction of programs and presents some measures of the effort required to write and understand programs.

12. IBM. *OS PL/I Checkout,Compiler: Programmer's Guide.* Pub. SC33-0007, IBM Corp., White Plains, New York, Oct. 1976, 4th edition.

13. Kernighan, B.W., and Plauger, P.J. *The Elements of Programming Style.* McGraw-Hill, New York, 1974. A study of programming style that discusses the shortcomings of examples drawn from programming textbooks. General rules of style are then used to rewrite the examples for readability.

14. Lehman, M.M. Laws and conservation in large-program evolution. Proc. 2nd Software Life Cycle Management Workshop, Atlanta, Georgia, 1978 (IEEE Pub. 78CH1390-4C, pp. 140–145). Lehman describes natural phenomena observed about the way in which the maintenance and evolution of large programs are planned, managed, and implemented.

15. Lientz, B.P., and Swanson, E. B. *Software Maintenance Management.* Addison-Wesley, Reading, Mass., 1980. The results of the authors' survey of almost 500 companies to compare software maintenance and costs.

16. Liu, C.C. A look at software maintenance. *Datamation 22*, 11 (Nov. 1976), 51–55. Investigates the problems of software maintenance and describes some improvements, in particular, in the areas of documentation and testing.

17. McCabe, T.J. A complexity measure. *IEEE Trans. Software Eng. SE-2*, 4 (Dec. 1976), 308–320. McCabe describes a graph-theoretic program complexity measure that depends only on the decision structure of the program. The use of this measure to manage and control program complexity is described.

18. Mills, H.D. Mathematical foundations for structured programming. Doc. FSC72-6012, IBM Federal Syst. Div., Gaithersburg, Md., Feb. 1972. The programming process is formulated as a step-by-step expansion of mathematical functions. A structure theorem guaranteeing that any program that can be represented as a flowgraph can be transformed into one containing only three types of structures—sequence, conditional, and iterative—is proved.

19. Myers, G.J. *Software Reliability.* John Wiley, New York, 1976. Defines software reliability, analyzes the major causes of unreliability, discusses the design and testing of reliable software, and touches on other factors in the production of reliable software such as project organization.

20. Myers, G.J. An extension to the cyclomatic measure of program complexity. *SIGPLAN Notices 12*, 10 (Oct. 1977), 61–64. Discusses anomalies found when calculating the complexity of a program under the assumption that it depends only on the program's decision structure and describes a simple extension to McCabe's complexity measure [17] to eliminate the anomalies.

21. Sheppard, S.B., et al. Modern coding practices and programmer performance. *IEEE Computer 12*, (Dec. 1979), 41–49. Describes the results of a series of experiments on the effects of modern coding practices on programming comprehension, program modification, and debugging performance.

22. Standish, T.A., et al. *The Irvine Program Transformation Catalogue.* Dept. Inform. and Comptr. Sci., Univ. of Calif. at Irvine, Irvine, Calif., 1976. A source book of ideas for improving programs through source-to-source transformations.

23. Stevens, W.P. *Using Structured Design.* Wiley-Interscience, New York, 1981. Illustrates the techniques of structured design with numerous examples that demonstrate guidelines for splitting a program into separate modules.

24. Yourdon, E. *Techniques of Program Structure and Design.* Prentice-Hall, Englewood-Cliffs, N.J., 1975. Discusses program design philosophies and methods; presents practical strategies for developing modular programs that are clear and readable.