1-1-1977

An analysis of structured programming.

Claire Meier Hamme

AN ANALYSIS OF STRUCTURED PROGRAMMING

by

Claire Meier Hamme

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1977

CERTIFICATE OF APPROVAL

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

Sept. 14, 1977
(date)

Professor in Charge

Chairman of Department

TABLE OF CONTENTS

ABSTRACT

The method of structured programming analyzed is a combination of three concepts: structured programming as conceived by Dijkstra, top-down programming, and stepwise refinement. There are three advantages to this method: one always has a working program, the program is relatively easy for another programmer to understand, and it is easy to modify. The method is described as a "structuring algorithm" consisting of the following directives: (1) exact problem definition of next level, (2) design of present level, (3) program to insure correctness, (4) refining of next level. Using this algorithm, a program to facilitate matrix arithmetic, is developed. This development is done in levels. It is illustrated that the "structuring algorithm" is intended to be a guideline only. In actuality it is necessary to look ahead or back up a level or two. The resultant program, however, appears as a clear logical progression. The value of well placed honest comments are examined, as well as the confusion that is caused

by improper commenting. The last chapter discusses the problems and criticisms formulated by P. Henderson and R. Snowdon in their paper entitled "An Experiment in Structured Programming". These problems include: the desire to reuse an already written procedure, a possible oversimplification with the structured programming approach, a derived false sense of security, and data structure decisions that are not consciously made by the programmer.

1. INTRODUCTION

The term "structured programming" has many interpretations as evidenced by the frequency that one is asked, "What is structured programming?" The technique that I wish to analyze, as proposed by Ledgard [2], [3], is a combination of three concepts.

(1) "Structured programming" as originally conceived by Dijkstra is a method of program development, such that at each step the program can be logically divided into distinct sub-structures, and such that correctness is apparent from the internal structure of the program.

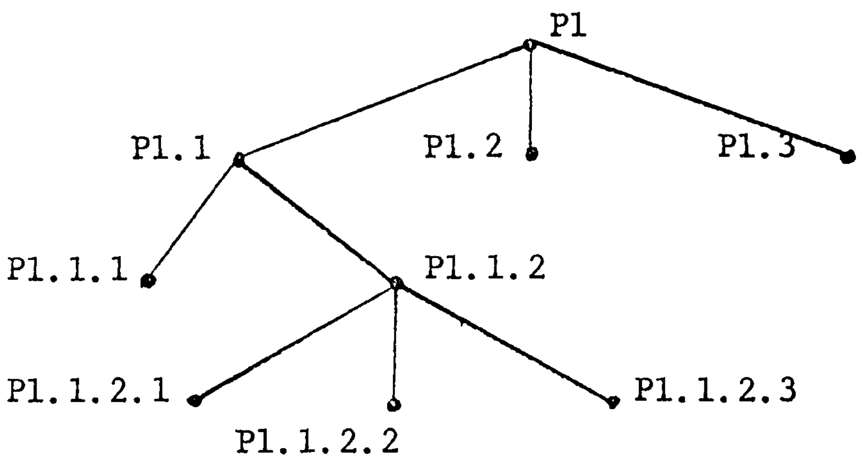
(2) "Top-down programming" is a type of structured programming where the development is done in the source language at hand. The main procedure is the first piece of code written, and then the sub-procedures are written, which in turn have been further split into sub-procedures, and so on, until the entire program has been written.

(3) "Stepwise refinement" is similar to top-down programming but one is not constrained to

work in a particular language. The development is done in phrases that will later, after the program is fully developed, be translated into a source language.

This combined technique then is a programming method whereby the program is developed and written in levels, the topmost level being the main program. At each level, the program is divided into sub-structures first by the method of stepwise refinement as the next level is conceived and then backing up a level to code, run and debug the previous level. This is done before committing oneself to further refinement. Thus the two concepts of stepwise refinement and top-down programming are alternately used in a back and forth manner.

The successive elaboration can be easily observed in a tree diagram. In the refining stage, each node represents a concept and the nodes emanating from it are elaborations or refinements of that concept. In the coding stage, each node represents a procedure or function and the nodes emanating from it are the procedures and functions that are called by the parent node.



The numbers indicate the route taken to reach each node. Thus the calling procedure is always conceived before its refinements and the calling procedure is always coded before its refinements. The result is, therefore, a strictly "top-down structure".

There are three distinct advantages to be gained from programming in this structured, top-down, stepwise refined method.

(1) One always has a complete working program.

- a) By running and debugging the previous level before further refinements are made, one is assured that the program, thus far, does what it was intended to do.
- b) By adding onto an already working program, any mistakes must necessarily be in the new portion or in the interaction of the

old and new. Therefore, if the programmer does not know what the mistake is at least he knows where it is.

(2) The program is relatively easy for another programmer to understand. The reader is given a logical step by step progression to follow from the most general concept to the most particular. Furthermore, he is never asked to read a module before he knows how and why it was called. This is an important consideration in programming. One does not stay at the same job in perpetuum. It is probable that someone else will one day be reading and updating your programs. In the spirit of professionalism, a programmer has a moral responsibility to write readily understandable programs.

(3) The program is easy to modify. A strictly top down design, as illustrated by the tree structure, requires that there be only one calling procedure for each particular sub-procedure. Therefore, if it is necessary to modify a specific module, one need not worry

about the effect that this change may cause in the rest of the program. The only possible problems must occur in procedures that it calls, or in the one module where it interacts with the entire program, its calling procedure.

More will be said about these advantages of structured programming later.

2. STRUCTURING ALGORITHM.

It is difficult to indicate the generality of this method by starting with the topmost level, because there have been no previous refinements. I must, therefore, ask the reader to assume that we have somehow managed to get to level 2. A demonstration of how this is done will be shown later in the illustrative program. Level 1 is the main program which has been coded, tested, and debugged. It calls on several, thus far, dummy procedures which will embody level 2. One of these procedures has been further refined in a tentative manner which will later become level 3.

(1) Exact problem definition of next level.

Since we are working on level 2, this means to exactly define level 3. Heretofore, level 3 has been sketchy. We must now decide exactly what will be accomplished in each level 3 module of this branch of the program, before we can write the code for level 2. This defining is done with careful procedure names and comments as unambiguous as possible. We might not get back to write some of these modules for some time and we must leave no doubt as to the originally intended purpose and strength of the module.

(2) <u>Design of present level.</u>

Level 2 had been exactly defined when level 1 was being worked on. In the above directive much of the work of the module we are working on has been passed on to the level 3 procedures that it will call. It is now time to plan the interaction of these level 3 procedures with the remaining function of this module. In other words, "How will it work?"

(3) <u>Program to insure correctness.</u>

The level 2 module that was just designed is now coded. In order to test whether it works, by that I mean whether it accomplishes its purpose, we can do one or both of two things:

a) Supplant it immediately for the dummy procedure, that was written to hold its place when working on level 1.

b) Write a dummy main program to test it. In order to determine whether the module behaves properly, we must be able to look at some output that is accomplished because of the module. If the module in question has some natural write statements in it that, given the proper data, will attest to the accuracy of the code, then it can immediately replace the dummy procedure of the same name and the program tested in its piecemeal entirety. If, however, the module, does not have this self testing facility, a dummy program must be written that calls this procedure and then writes out some information to prove that it worked. Then the module is supplanted for the dummy module to insure that the interaction of the procedures is correct. The main point in this, is that we want to replace the dummy procedure with the new procedure without changing it. We do not want to put in dummy write statements, then take them out, and assume that all is well. An error made in this manner is most difficult to find.

(4) Refining of next level.

This is the stage where most of the thinking is done. Alternate methods of refining level 3 must be considered. In order to choose wisely, it is necessary to look ahead a level or two to see where a particular refinement might lead. Changes are often necessary, so this refining is done in an informal phrase-like manner. When we are completely satisfied with the workability of a refinement for level 3, we are finished with level 2. Now the process begins again. By applying this algorithm, one leg of the program is seen through to completion while the other completely defined refinements await their turn.

The foregoing method of structured top-down programming, enables the global data structures to be built up piece by piece along with the program. It is, therefore, only necessary to conceive of the part of the data sturucture that is needed for the particular module under construction. Since the global data structures are defined in the main program which is the first module written, we must allow this module to be updated as the data structures are expanded. In keeping with this concession in the, otherwise, strictly top-down design is the treatment of the initialization procedure which is called by the main program. As the data structures grow, so does the procedure that initializes them, and so PROCEDURE INITIALIZE is also allowed continual updating.

While I am on the subject of exceptions, possibly this is the time to mention that I am making one other exception to the top-down design, for the procedure that handles fatal errors. This procedure is called by almost every other module in the program, with always the same results - an error message is printed, and an exit from the program is executed. I am, therefore, allowing PROCEDURE ERROR to be global to the program with new error messages added on as they are needed.

3. ILLUSTRATIVE EXAMPLE.

Perhaps the best way to describe how to structure a program is with an illustration. I have chosen a program that enables the user to do matrix arithmetic. In order to add enough complexity, to give an overall feel for the method, the user will be allowed to return space for matrices no longer needed and reuse it. The program will do some garbage collection to combine unused and returned space. The ensuing section shows how the structuring algorithm is applied to each successive level of the program. The programming language used is Pascal 1.

3.1 LEVEL ZERO

3.1.1 Exact problem definition of level 1.

Level 1 is the main program. It will be a user operated program to facilitate matrix arithmetic with the added feature of dynamic memory.

3.1.2 Design of level 0.

Level 0 is a comment describing the program.

3.1.3 Program to insure correctness.

Usually the program to insure correctness will call dummy procedures that will later become the next level. In this case, the next level is the main program; so we have a dummy program.

```
BEGIN
(*Program to facilitate matrix arithmetic*)
END.
```

3.1.4 Refining of level 1.

Level 1, the main program, will call six level 2 procedures.

(1) error routine

(2) dimensioning of matrices

(3) reading in of matrix entries

(4) matrix arithmetic section

(5) display of matrix entries

(6) return of space

The refinement tree is therefore:

$$\begin{tikzcd}
 & & \text{MAIN} \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] \arrow[drrr] & & & \\
\text{ERROR} & \text{DIMENSION} & \text{INPUT} & \text{COMPUTE} & \text{SHOW} & \text{RETURN}
\end{tikzcd}$$

3.2 LEVEL ONE

3.2.1 Exact problem definition of level 2.

Before coding level 1, we must know exactly what the level 2 procedures will do.

(1) PROCEDURE ERROR. The purpose of this module is to print an appropriate error message and to abort the program. This is a very unusual procedure. It will be expanded as new possible errors are determined and it will be considered as global; all other procedures will be allowed to call it. A later discussion will consider when such an allowance is feasible.,

(2) PROCEDURE DIMENSION. Routine to read in the name, number of rows, and number of columns of each matrix. It will also allocate space and maintain the file system.

(3) PROCEDURE IINPUT (spelled with two I's because "INPUT" is a reserved word). It will read in matrix entries, convert them to real numbers and store them.

(4) PROCEDURE COMPUTE. Routine to read instructions, do directed arithmetic, and store answers.

(5) PROCEDURE SHOW. Module to read name of matrix, and display the entries of the matrix of that name.

(6) PROCEDURE RETURN. Procedure to read the name of a matrix, return the space used by the matrix of that name, and update the file system.

3.2.2 Design of level 1.

Level 1 is the main program. It will consist of a large loop that will expect to read a D, I, C, S, or R and will transfer control to PROCEDURE DIMENSION, IINPUT, COMPUTE, SHOW, or RETURN respectively. Each section will have its own read statements and will return to the main program when it reads a semi-colon. Then the main program will again look for a D, I, C, S, or R. If there have been no errors, this will continue until an END OF FILE is encountered. There will need to be a procedure to initialize global variables. Althrough this procedure will be called by the main program, it is not considered part of level 2. It is written at the same time as level 1 and is a part of it. The calling tree is, therefore, slightly different than the refining tree.

$$\begin{tikzcd}
\text{INITIALIZE} & & \text{MAIN} \arrow[ll] \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] \arrow[drr] \arrow[drrr] & & & \\
\text{ERROR} & \text{DIMENSION} & \text{IINPUT} & \text{COMPUTE} & \text{SHOW} & \text{RETURN}
\end{tikzcd}$$

3.2.3 <u>Program to insure correctness</u>.

The dummy procedures will later become level 2. Each module begins with a comment describing its purpose. Pascal 1 is a one pass compiler, therefore the calling modules are written after the procedures that they call.

```
(*Program to facilitate matrix arithmetic*)

LABEL
     13;

VAR
     CH:CHAR;

PROCEDURE INITIALIZE;
(*Procedure to initialize global variables*)
BEGIN
END;

PROCEDURE ERROR (N:INTEGER);
(*Prints error message, aborts program*)
BEGIN
  WRITE (#   DUMMY PROCEDURE ERROR #,EOL);
END;

PROCEDURE DIMENSION;
(*Reads name, number of rows, number of columns
 of matrix, allocates space, maintains file
 system*)
BEGIN
  WRITE (#   DUMMY PROCEDURE DIMENSION #,EOL);
END;
```

```
PROCEDURE IINPUT;
(*Reads in matrix entries, converts them to
 real numbers and stores them*)
BEGIN
  WRITE(#   DUMMY PROCEDURE IINPUT#,EOL);
END;

PROCEDURE COMPUTE;
(*Arithmetic section*)
BEGIN
  WRITE(#   DUMMY PROCEDURE COMPUTE#,EOL);
END;

PROCEDURE SHOW;
(*Displays matrices*)
BEGIN
  WRITE(#   DUMMY PROCEDURE SHOW#,EOL);
END;

PROCEDURE RETURN;
(*Returns space, updates file system*)
BEGIN
  WRITE(#   DUMMY PROCEDURE RETURN#,EOL);
END;

BEGIN (*Main*)
  INITIALIZE;
  READ(CH);
  WHILE CH = # # DO
        READ(CH);
  WHILE NOT EOF(INPUT) DO
        BEGIN
              IF CH = #D# THEN DIMENSION ELSE
              IF CH = #I# THEN IINPUT ELSE
              IF CH = #C# THEN COMPUTE ELSE
              IF CH = #S# THEN SHOW ELSE
              IF CH = #R# THEN RETURN ELSE
              IF CH = EOL THEN (*NOTHING*) ELSE
              ERROR(1);
              READ(CH);
              WHILE CH = # # DO
                    READ(CH);
        END;
   13:
  END.
```

The above program is then run with good and bad data to determine if control is transferred correctly, if it skips over blanks, and continues to the next line. PROCEDURE INITIALIZE has nothing in it because there are no data structures as yet. It is not, however, a dummy procedure to be replaced when level 2 is written. It is part of the main level and will be augmented as new data structures are conceived.

Before going on to refine level 2, consider PROCEDURE ERROR. This is an end node needing no further refinement. Before I forget the cause of Error 1 in the main program, I want to expand PROCEDURE ERROR to give an appropriate error message. Since this procedure will be called from many places in the program, I also want to note, with a comment, from where this particular call was made.

```
PROCEDURE ERROR (N:INTEGER);
(*Prints error message, aborts program*)
BEGIN
  CASE N OF
       1:(*Called from Main*)
         WRITE(# #,CH,# WRITTEN WHERE
            KEY LETTER EXPECTED#,EOL);
  END;
  WRITE(#   PROGRAM ABORTED#,EOL);
  GOTO EXIT 13;
END;
```

This PROCEDURE ERROR now supplants the dummy procedure, and the program is again tested with good and bad data. This time the bad data should cause the program to abort. I hope you noticed the GOTO statement in PROCEDURE ERROR. This is the only one in the program. It transfers control to the only label in the program, the label 13 at the end of the main program. It is not good practice and certainly against the precepts of "structured programming" to transfer control from one level to another, other than through formal calls. It is, however, unavoidable in this case, because the main program is a continuous loop. The only way to terminate the program without an encountered END OF FILE is to jump out of that loop.

3.2.4 Refining of level 2

I originally said that level 2 had six components. PROCEDURE ERROR, which is part of level 2 has already been considered. I am, therefore, left with five components of level 2 to consider. I must now narrow my attention to one component and follow its development until fully expanded. It is important to the best evolution of the data structures that I develop the components in the correct order. The main data structure will be a file system that will record the location of each matrix. Since the purpose of PROCEDURE DIMENSION is to allocate space and maintain the file system, PROCEDURE DIMENSION is the logical choice as first procedure to develop.

The refining in this case is not as clear cut as it was in the previous level. Before, we had several almost unrelated actions to perform, whereas now the functions of PROCEDURE DIMENSION progress linearly. I will, therefore, use this refining stage to reduce some of the responsibilities of PROCEDURE DIMENSION, so that it will be easier to code. This module will call only two level 3 modules, one of which is PROCEDURE ERROR and the other a continuation of PROCEDURE DIMENSION.

The refining tree is, therefore:

$$\begin{tikzcd} & \text{DIMENSION} \arrow[dl] \arrow[dr] & \\ \text{FINDSPACE} & & \text{ERROR} \end{tikzcd}$$

3.3 LEVEL TWO

3.3.1 Exact problem definitions of level 3

(1) PROCEDURE FINDSPACE. Module to allocate space and maintain file system.

(2) PROCEDURE ERROR. Same as before

3.3.2 Design of level 2

Control will remain in PROCEDURE DIMENSION until a semi-colon is read. The procedure will want to read the name, number of rows, and number of columns of each matrix that will be either input or evaluated. All information will be read in as type CHARACTER. A matrix name must be a single letter of the alphabet. Since, the number of rows and number of columns are read in as type CHARACTER, there will need to be a routine to find the integer value of a string of digits. I would also like a routine that would skip over delimiter marks such as commas. I must, therefore, back up a level and add two more refinements to PROCEDURE DIMENSION.

New tree:

$$\begin{tikzcd}
 & & \text{DIMENSION} \arrow[dll] \arrow[dl] \arrow[d] \arrow[dr] & \\
\text{FINDSPACE} & \text{ERROR} & \text{GETNONDELIMITER} & \text{IINTEGER}
\end{tikzcd}$$

Definition of new refinements:

(3) PROCEDURE GETNONDELIMITER. Routine to skip over blanks, commas, slashes, parentheses, and end of line character.

(4) FUNCTION IINTEGER (spelled with two I's because "INTEGER" is a reserved word). Function to return the evaluated number represented by the string of digits formed by CH followed by subsequent characters until a non digit is read. The global variable CH will be set equal to that non digit.

Continuation of design of level 2 (PROCEDURE DIMENSION):

We observe the need to recognize letters, digits, and delimiters and define these sets of characters as global variables. The contents of the sets are initialized with an addition to PROCEDURE INITIALIZE. We also need a data structure to hold the name, number of rows, and number of columns of a matrix before it is allocated space.

3.3.3 Program to insure correctness

At this level, we are not writing a program, we are writing a procedure. In order to test it, we must either supplant it for the dummy procedure in our already working program, or write a dummy program to test it in first. PROCEDURE DIMENSION does not have any natural write statements that will allow us to test the accuracy of the procedure, but it does call the dummy PROCEDURE FINDSPACE. Since FINDSPACE will later be changed, we can put write statements in it to test PROCEDURE DIMENSION.

My purpose in this directive is to code and test PROCEDURE DIMENSION. Although the new refinements, PROCEDURE GETNONDELIMITER and FUNCTION IINTEGER, belong to the next level, I'm going to code them along with PROCEDURE DIMENSION. There are two reasons for this: (1) They are very small modules, needing no further refinements (2) I cannot properly test PROCEDURE DIMENSION without having available for use PROCEDURE GETNONDELIMITER and FUNCTION IINTEGER. The following then are segments of program that will either be added onto the already existing program or will replace the dummy PROCEDURE DIMENSION.

```
ADDITIONS TO DATA STRUCTURES
TYPE
     HS = RECORD
            TITLE: CHAR;
            RWS: INTEGER;
            COLMS: INTEGER;
            END;
VAR
     HOLD: HS;
     LETTERS, DELIMITERS, DIGITS: SET OF CHAR;

ADDITIONS TO PROCEDURE INITIALIZE
     LETTERS := [ ];
     FOR CH := #A# TO #Z# DO
            LETTERS := LETTERS JOIN[CH];
     DIGITS := [ ];
     FOR CH := #0# TO #9# DO
            DIGITS := DIGITS JOIN [CH];
     DELIMITERS := [# #,#,#,#/#,#(#,#)#,EOL].

ADDITIONS TO PROCEDURE ERROR
     2:
     3:
     4:
     5:
```

I don't know what the error messages will be yet, but I want to add the possiblity of more errors.

```
PROCEDURE DIMENSION;
(*Reads name, number of rows, number of columns of matrix,
 allocates space, maintains file system*)

          PROCEDURE FINDSPACE;
          (*Allocates space, maintains file system*)
          BEGIN
            WRITE(#  DUMMY PROCEDURE FINDSPACE#,EOL);
            WRITE(# #,HOLD.TITLE,# #,HOLD.RWS,# #,
                                         HOLD.COLMS,EOL);
          END;
```

```
            FUNCTION IINTEGER(CH: CHAR): INTEGER;
            (*Gives integer  value to string of digits,
             sets CH to next character after string*)
            VAR
              INT: INTEGER;
            BEGIN
                 INT := 0;
                 WHILE CH IN DIGITS DO
                      BEGIN
                         INT := INT *10+ORD(CH)-ORD(#0#);
                         READ(CH);
                      END;
                 IINTEGER := INT;
            END;

            PROCEDURE GETNONDELIMITER;
            (*Skips over delimiters*)
            BEGIN
                 WHILE CH IN DELIMITERS DO
                         READ(CH);
            END;

   BEGIN(*Dimension*)
        READ(CH);
        GETNONDELIMITER;
        WHILE CH NE #;# DO
              BEGIN
                   IF NOT (CH IN LETTERS)
                          THEN ERROR(2)
                          ELSE HOLD.TITLE := CH;
                   READ(CH);
                   GETNONDELIMITER;
                   IF NOT (CH IN DIGITS)
                          THEN ERROR(3)
                          ELSE HOLD.RWS := IINTEGER(CH);
                   (*CH IS NOW NONDIGIT FOLLOWING THE FIRST
                    INTEGER*)
                   GETNONDELIMITER;
                   IF NOT (CH IN DIGITS)
                          THEN ERROR(3)
                          ELSE HOLD.COLMS := IINTEGER(CH);
                   (*CH IS NOW NONDIGIT FOLLOWING THE SECOND
                    INTEGER*)
                   FINDSPACE(HOLD);
                   GETNONDELIMITER;
              END;
   END;
```

The following is some sample data with which to test the program.

```
D   A(2,3)   D  43 10   B  3,8;   C   S
R   D   L(11,12);    I
```

When we are satisfied that it works correctly, we want to add the appropriate error messages to PROCEDURE ERROR, and retest the program adding some bad data.

```
    2: (*Called from Dimension*)
       WRITE(# #,CH,# WRITTEN WHERE
                 LETTER EXPECTED#,EOL);
    3:(*Called from Dimension*)
      WRITE(# #,CH,# WRITTEN WHERE
                 DIGIT EXPECTED#,EOL);
```

3.3.4 Refining of level 3

There is only one remaining module to consider at this level and that is PROCEDURE FINDSPACE. There are clearly three possibilities that could occur when PROCEDURE FINDSPACE looks for space to store the matrix. There will either not be enough space, in which case it will call PROCEDURE ERROR, be enough space but not together, in which case it will call PROCEDURE CARBAGECOLLECTION or find a large enough segment of space. The refining tree is therefore:

$$\begin{tikzcd}
 & \text{FINDSPACE} \arrow[dl] \arrow[dr] & \\
\text{ERROR} & & \text{GARBAGECOLLECTION}
\end{tikzcd}$$

Level 3 would then be developed in much the same way as level 2. PROCEDURE CARBAGECOLLECTION will probably not require further refinement, so that when level 4 is developed, the first branch of the tree will be completed.

I hope that the reader did not find this section too tedious. I thought it necessary to show that programming rules are meant as guidelines only. Even a program especially picked for illustrative purposes does not perfectly conform to the rules.

4. COMMENTS.

In a non structured language, such as Fortran, comments can be a very effective means of transmitting the structure of a program. Visually they divide the program into modules, and conceptually they give internal documentation as to the purpose and method of the module. Consider the following example.

```
        MAX = 1
        DO 110 I = 2,N
 110    IF(PRES(I) .GT. PRES(MAX))MAX = I
        MORE = LAST+1
        FLOW(MORE) = FLOW(MAX)+100.00
        DO 112 I = 1,N
 112    IF(FLOW(I) .LT. FLOW(MORE) .AND.
       +FLOW(I) .GT. FLOW(MAX))MORE = I
        IF(MORE .EQ. LAST+1) GO TO 300
        LESS = LAST+1
        FLOW(LESS) = 0.0
        DO 114 I = 1,N
 114    IF(FLOW(I) .GT. FLOW(LESS) .AND.
  .    +FLOW(I) .LT. FLOW(MAX))LESS = I
  .     IF(LESS .EQ. LAST+1) GO TO 300
  .     .
  .     .
  .     .
 300    CONTINUE
```

Now consider the same example with comments:

```
*GET  3 POINTS FOR SPLINE FIT
*   * FIND MASS FLOW OF MAXIMUM PRESSURE
        MAX = 1
        DO 110 I = 2,N
 110    IF(PRES(I) .GT. PRES(MAX))MAX = I
*   * FIND NEXT LARGER MASS FLOW
        MORE = LAST+1
        FLOW(MORE) = FLOW(MAX)+100.00
        DO 112 I = 1,N
 112    IF(FLOW(I) .LT. FLOW(MORE) .AND.
       +FLOW(I) .GT. FLOW(MAX))MORE = I
        IF(MORE .EQ. LAST+1)GO TO 300
*   * FIND NEXT SMALLER MASS FLOW
        LESS = LAST+1
        FLOW(LESS) = 0.0
        DO 114 I = 1,N
 114    IF(FLOW(I) .GT. FLOW(LESS) .AND.
       +FLOW(I) .LT. FLOW(MAX))LESS = I
        IF(LESS .EQ. LAST+1) GO TO 300
  .     .
  .     .
  .     .
  .     .
  .     .
 300    CONTINUE
```

The second example is easier to read, understand, and modify. Using comments also leads to better programming because the programmer is encouraged to think in modules, and in order to keep his comments accurate cannot branch all over the program.

Unfortunately improperly used comments can be just as effective a way of hiding and confusing the structure of a program. Comments that are too detailed break up the code so much that one is unable to see the logical module. Consider the last example again:

```
* GET  3 POINTS FOR SPLINE FIT
* *    FIND MASS FLOW OF MAXIMUM PRESSURE
* * *   SET MAX EQUAL TO FIRST IN ARRAY
        MAX = 1
* * *   TEST EACH NEXT ENTRY IN ARRAY AGAINST MAXIMUM
* * *   IF LARGER THAN MAXIMUM,EXCHANGE
        DO 110 I = 2,N
110     IF(PRES(I) .GT. PRES(MAX))MAX = I
* *    FIND NEXT LARGER MASS FLOW
* * *   INITIAL MORE BEYOND LAST ENTRY IN ARRAY
        MORE = LAST+1
* * *   INITIAL MASS FLOW FOR MORE LARGER THAN
* * *   WILL BE FOUND
        FLOW(MORE) = FLOW(MAX)+100.00
         etc.
```

The excessive comments shown here, hinder the eye from viewing a complete module at one time. They also took too much work to write.

A more serious problem than over commenting is wrong commenting. A comment that groups code improperly or is not accurate as to the function or operation of the code can baffle the reader who tends to believe the comment rather than the code. Take for example this piece of a binary search from an interpreter.

```
C         SCOPE
    47       IF(L(IAL-2)-IPLUS)79,72,74
C         BINARY PLUS OR MINUS
    72       ILAG = 3
             GO TO 75
    74       IF (L(IAL-2)-ITIMES)72,81,77
C         RIGHT PARENTHESIS
    76       ILAG = 2
             GO TO 75
    77       IF L(IAL-2)-IRTPAR)81,76,80
C         UNARY MINUS
    78       ILAG = 5
             GO TO 75
C         POWER
    79       ILAG = 6
             GO TO 75
    80       IF(L(IAL-2)-IUMINS)84,78,82
```

It appears, in this example, as though the test at label 74, is executed if and only if there has been a previous branch to the "Binary plus or minus" section. Actually the opposite is true. After branching to "binary plus or minus" the program branches to label 75. The test at label 74 has, therefore, nothing to do with the "binary plus or minus section". The proper grouping should have been as follows:

```
C          TEST
      47      IF(L(IAL-2)-IPLUS)79,72,74
      74      IF(L(IAL-2)-ITIMES)72,81,77
      77      IF(L(IAL-2)-IRTPAR)81,76,80
      80      IF(L(IAL-2)-IUMINS)84,78,82
C          BINARY PLUS OR MINUS
      72      ILAG = 3
              GO TO 75
C          RIGHT PARENTHESIS
      76      ILAG = 2
              GO TO 75
C          UNARY MINUS
      78      ILAG =5
              GO TO 75
C          POWER
      79      ILAG = 6
              GO TO 75
```

A comment that does nothing to enhance the understanding of the reader is simply a waste of file space and a waste of the programmers time in writing it. Example.

```
*THIS IS A LITTLE FUDGE FACTOR TO MAKE THE PLOTS LOOK
 NEATER
 528       TMAX(I) = 2.0
           DX = DX .AND. (.NOT. 777B)
           IF (DX .LE. 2.0) GO TO 529
           TMAX(I) = 4.0
           IF (DX .LE. 4.0) GO TO 529
           TMAX(I) = 5.0
           IF(DX .LE. 5.0) GO TO 529
           TMAX(I) = 8.0
           IF(DX .LE. 8.0) GO TO 529
           TMAX(I) = 10.0
```

This piece of code truncates the last three digits of the X increment in a ploter routine in order to divide the axes into neater intervals. This was not obvious from the coding, nor does the comment help to understand it. Witty remarks, such as, "LITTLE FUDGE FACTOR" if anything give an unprofessional appearance to the program.

All things considered, a programmer is tempted to not use comments. While writing the program, the code appears perfectly clear, and it is a bother to stop and put ones thoughts into words. However, a few days to a week later, the code that appeared so clear becomes indiscernable. One needs only to read someone elses program to appreciate the value of a well placed honest comment.

5. PROBLEMS AND CRITICISMS.

These problems and criticisms were formulated by P. Henderson and R. Snowdon in "An Experiment in Structured Programming". [1]

5.1 The top-down tree structure requires that each procedure that is called must be on a lower level than the calling procedure. Suppose, in the illustrative program, another branch of the program, such as the portion that reads in matrix entries and evaluates the input into real numbers, wants to call a function exactly like FUNCTION IINTEGER(CH). Do we rewrite the same function to keep the tree structure intact, or do we allow FUNCTION IINTEGER(CH) to be global as we did PROCEDURE ERROR?

The introduction considered three advantages to be gained from a structured, top-down program. The question we must decide is, "Can we retain these three advantages if we allow FUNCTION IINTEGER(CH) to be global?"

(1) The first advantage was that having a correct working program at all times insures correctness of logic. This was demonstrated in the illustrative program. Including FUNCTION

IINTEGER(CH) as a global function along with PROCEDURE ERROR would in no way harm the ability to insure correctness at each level.

(2) The second gain was the ease of another programmer in understanding your program. Suppose we allowed FUNCTION IINTEGER(CH) to be global, let us look at the resultant program from the eyes of another programmer.

After reading the main program, he reads the global procedures. The purpose of PROCEDURE ERROR is apparent and then he reads FUNCTION IINTEGER(CH). The name and comment do little to explain the purpose of the routine since at this level he doesn't know that the program reads all input as type CHARACTER. The clarity of logic so carefully built will have been clouded. I would, therefore rewrite FUNCTION IINTEGER(CH) in the new section where it is called and retain the clarity of the program. This would not have been a problem with a two pass compiler since the order that the procedures are in the program would not matter. But with a one pass compiler the modules must have been defined before they can be called. Therefore in order for a module to be global it must be in the beginning of the program.

What if the purpose of the routine that one is considering to make global was apparent to the reader? Suppose the procedure that we want to use again is PROCEDURE GETNONDELIMITER. Would we have to relinquish any advantages if we made this procedure global? The guarantee of correctness for each level again remains intact. What about the clarity of understanding? Since the variable DELIMITERS is declared with the global variables and the set is defined in PROCEDURE INITIALIZE which is called by the main program, the purpose of PROCEDURE GETNONDELIMITER would be understandable to anyone reading the program.

(3) The third gain was ease of modification. This advantage is always lost, when a procedure is made global. The question to consider then is, how likely is it that this procedure will need modification? In this case, the only change I can foresee is in the set of delimiters, which is already global. I would, therefore, have no objections to making PROCEDURE GETNONDELIMITER global.

After establishing that none of the advantages of structured programming would be lost by making a module global, it becomes a matter of personal preference. I would, personally, make any routine that I was reasonably sure another programmer would understand, and that would not require updating or modification global. A procedure as short as PROCEDURE GETNONDELIMITER could just as easily be rewritten for each new calling procedure, since it requires so little space. However, many procedures with the same name could cause confusion. An error in one might be in all the similar ones and would need to be checked. Therefore, again in the interest of clarity, I would make PROCEDURE GETNON-DELIMITER global.

Suppose the procedure that you are considering to make global, is not used <u>exactly</u> the same way each time it would be called; but it is possible to pass a parameter to indicate how it is to be used. Again, it is first necessary to establish that clarity and ease of modification are preserved. The next consideration is time and space. Passing a parameter costs time and rewriting a module costs central memory. Each decision must be made separately regarding how much space is available and the importance of speed.

5.2 There is an oversimplification with the structured programming approach. In other words, "It sounds so easy, but -". This is a justified criticism. It is seldom easy to refine the level you are working on. Initially, it is done in a tentative vague manner. Then one must look ahead a level or two to see how a particular refinement might work out. It is often necessary to change refinements many times before one sees a workable solution. This is not an easy thing to do. The structuring algorithm is not a "formula" to crank out a program. It is a guide to help keep ones logic clean. Although, the refining stage is difficult, one is not writing any code at this stage. Only after you can see a workable solution, do you commit yourself to a refinement, and only when that refinement is well defined do you write the code for the previous level.

The completed part of the program appears in clear logical levels. This makes it easy for others to understand, and gives you a true picture of where you are. The actual design of the program has not been helped a great deal. This is similar to the presentation of the proof of a mathematics theorem in school, which is a step by step method leading the student from what he already knows to some new knowledge. This is usually very different from the way the mathematician first perceived the theorem. A logical presentation, however, instills confidence in the validity of the program or theorem.

5.3 Approach leads to a false sense of security. By defining modules in a descriptive manner, rather than with formal code, a problem of interpretation arises. One must be able to attach meaning to the description, and the elaboration of the concept must be correct in that it fulfills precisely the meaning as specified by the defining conditions.

The basis of this criticism comes from a program written, by P. Henderson and R. Snowdon [1], in a top-down, structured manner as part of a student demonstration. Everyone agreed that the program was conceptually correct, but when it was coded, it didn't work. It turned out that when a particular procedure had been conceived it had a slightly stronger purpose than when it was coded.

Ledgard, in his answer to this criticism [2], very well shows that this particular problem would have been avoided if there had been a correct running program at all times. The entire program had been structured and refinements locked into without any code having been written. Once a refinement is chosen, after many possible choices, the previous level should be coded and run. This keeps the possibility of a problem within two levels. He goes on to solve the same programming problem in a structured manner, but insuring correctness with a running program, and shows that the problem then does not arise.

Much as I admire the programming concepts of Ledgard, I don't believe he completely disproved this criticism. Even with a working program, if a module does not do exactly what it was intended to do the resultant program will not work. It is the code that counts not the name that you give it. Another danger is that a reader of the program will misunderstand the workings of a procedure because the name or comment was misleading. There is a tendency to believe the comment or name rather than to read the code. I think this is a very real danger in modular programming. Hopefully, awareness of the problem, will result in programmers giving greater care to names and comments, and a reader using names and comments as a help to understanding the code not a replacement.

5.4 Data structure decisions sneak in that are not consciously made by the programmer. Programming in a structured, top-down manner delays the conception of data structures until they are needed in the program. Only the part actually needed for the particular module under construction is conceived. The data structure, therefore, grows in complexity along with the program. It is for this reason truly a structure. The problem is that it is not a "top-down" structure. By only looking at one module at a time we might begin building a data structure that will prove to be inefficient in other parts of the program. Since, data structures are usually global to the program this results in having to "fix" any procedure that used the data structure previously. This is alot of work, and one usually tries instead to "patch" the existing structure. The result is often a cumbersome inefficient data structure, which is difficult for the programmer to update and the reader to understand. The solution to this problem appears to lie in the order that the modules are designed. We want to structure the data structures but to do it in a top-down manner rather than an inside out forest approach. This involves writing the section that requires a heavy use of a data structure <u>before</u> we write a section that uses it trivially. Again, this is not easy to do and involves much preplanning. Writing programs in a structured, top-down, stepwise refined manner does not lead to easier programming, it leads to better programs.

REFERENCES

1. P. Henderson and R. Snowdon, An Experiment in Structured Programming, BIT 12 (1972), 38-53.

2. Henry F. Ledgard, The Case For Structured Programming, BIT 13 (1973), 45-57.

3. Henry F. Ledgard, Programming Proverbs, Hayden Book Company, Inc., Rochelle Park, New Jersey, 1975.

VITA

Claire M. Hamme, daughter of Mr. and Mrs. Christian W. Meier, was born in New York City, New York on March 14, 1942. She attended Gettysburg College in Gettysburg Pennsylvania where she received the Baum Mathematics Prize for outstanding potential in Mathematics. She graduated in 1963 receiving the degree of Bachelor of Arts. Mrs. Hamme is married to Robert W. Hamme of Easton. They have three children. In September 1974, Mrs. Hamme began working toward the degree of Master of Science in Computer Science at Lehigh University in Bethlehem, Pennsylvania. During that time, she held a teaching assistantship in the Department of Mathematics at Lehigh University. She is presently employed with Ingersoll-Rand Company of Phillipsburg, New Jersey.