Technical Note
Human Aspects of Computing

Henry Ledgard
Editor

Control Flow and Data Structure Documentation: Two Experiments

Ben Shneiderman
University of Maryland

Two experiments were carried out to assess the utility of external documentation aids such as macro flowcharts, pseudocode, data structure diagrams, and data structure descriptions. A 223 line Pascal program which manipulates four arrays was used. The program interactively handles commands that allow the user to manage five lists of items. A comprehension test was given to participants along with varying kinds of external documentation. The results indicate that for this program the data structure information was more helpful than the control flow information, independently of whether textual or graphic formats were used.

CR Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques—*flow charts.*

General Terms: Documentation; Experimentation; Human Factors

Additional Key Words and Phrases: pseudocode, data structure diagrams

1. Introduction

Proposals for program documentation techniques that allegedly aid comprehension frequently appear in journal articles and textbooks. This healthy outpouring of new ideas stimulates discussion but often leads to controversy. Usually these proposals are based on an individual's experience with a limited number of projects, languages, and problem domains, so disagreements are not surprising. In recent years there has been increased interest in controlled experimental evaluations to ascertain under which conditions a particular documentation technique is most effective [5].

In five previous experiments Shneiderman et al. [7] could not demonstrate that detailed standard flowcharts were of assistance to undergraduate Fortran programmers in comprehending, debugging, or modifying when a copy of the program was available. The detailed flowcharts had approximately one box per Fortran statement, making them larger than the programs, but information from declarations, FORMAT statements, and comments were not included. Two unpublished student projects run under my direction during the past three years have supported these results with up to 200 line Fortran programs.

Knowledgeable programmers apparently prefer to work with the code itself rather than the lengthier detailed flowcharts. This is not surprising since a detailed flowchart is merely a syntactic recoding of the Fortran program and provides little additional aid. This coincides with the syntactic/semantic model of programmer behavior [6] which suggests that a useful aid must facilitate encoding of the program syntax into higher level semantic units. Competent programmers deal more with problem domain related units than with program domain related syntactic tokens. High level comments using problem domain terminology have been shown to be more effective in aiding comprehension than numerous low level comments using program domain terminology.

These results and the syntactic/semantic model suggest that helpful documentation would provide a high level framework which reveals information that is difficult to obtain from the code itself. With a high level framework, a programmer can anchor the knowledge acquired from reading each line or small unit of code.

There are two kinds of knowledge which are difficult to obtain from the code and can serve as a high level framework for anchoring detailed knowledge. The first is control flow information which might be represented by pseudocode or by various forms of a macro or system flowchart. Numerous proposals and examples appear in the literature. Some proposals even blend the graphic representation of a macro flowchart with the textual form of pseudocode by suggesting indentation strategies or limited use of boxes and arrows. Control flow information in a macro flowchart or pseudocode should probably be about one-tenth of the code length for it to be effective. A one-page macro flowchart that shows the calling relationships among 20 one-page modules is probably useful because it shows information which is difficult to obtain from reading the code.

The second form of program documentation which contains high level semantic concepts is data structure information which might be represented by a diagram or by a textual description. Linked list structures or arrays are often shown pictorially while some texts recommend a textual description of data structure contents be included in a comment block. Database management system programmers have traditionally used Bachman diagrams or variants to describe complex record relationships. Fitter and Green [3] provide an excellent survey of graphic notations in programming.

Ramsey and Atwood [4] studied the use of a form of pseudocode called Program Design Language (PDL) and flowcharts during the design and implementation phases of a two-pass assembler for a minicomputer. Those using

Author's present address: B. Shneiderman, Department of Computer Science, University of Maryland, College Park, MD 20742.

Fig. 1. Listing of Program UPKEEPER (used for Experiments 1 and 2).

```
        (* THE PURPOSE OF THE PROGRAM UPKEEPER
           IS TO CREATE LISTS OF ITEMS  *)
                                        (*          *)
PROGRAM UPKEEPER (INPUT, OUTPUT);
    CONST
        MAXLIST  = 5;
        MAXENTRIES  = 101;
        MAXLENGTH  = 25;
        MAXITEMS  = 20;
    TYPE
        PACKCHARS = PACKED ARRAY    [1..80] OF CHAR;
    VAR
        CMD       : PACKCHARS;
        N         : 0..MAXLIST;
        STATUS    : ARRAY   [1..MAXLIST] OF CHAR;
        TOTAL     : ARRAY   [1..MAXLIST] OF INTEGER;
        TITLES    : ARRAY   [1..MAXLIST]  OF PACKCHARS;
        ENTRIES   : ARRAY   [1..MAXENTRIES] OF PACKCHARS;
        X         : INTEGER;
(*************************************************)
(* ASSUME PROCEDURE GETWORD IS IN THE FOLLOWING PROGRAM.  *)
(* THEREFORE, THE STATEMENT GETWORD(CMD) RETRIEVES    *)
(* THE NEXT WORD INPUT.                               *)
(*************************************************)
                                        (*          *)
FUNCTION FINDINDX(POS: INTEGER; COUNTER:  INTEGER): INTEGER;
    BEGIN
        FINDINDX := (20*(POS - 1) + COUNTER)
    END;
                                        (*          *)
FUNCTION LOCATE(LISTNAME: PACKCHARS ):  INTEGER;
    VAR
        X         : INTEGER;
        SEARCHING  : BOOLEAN;
    BEGIN
        LOCATE := 0;
        X := 1;
        SEARCHING := TRUE;
        WHILE SEARCHING AND (X <= MAXLIST) DO BEGIN
            IF STATUS  [X] = 'F'
                THEN
                    IF TITLES  [X] = LISTNAME
                        THEN BEGIN
                            LOCATE := X;
                            SEARCHING := FALSE
                        END;
            X := SUCC(X)
        END;
    END;
                                        (*          *)
PROCEDURE CREATPROC;
    VAR
        LISTNAME  : PACKCHARS;
        DONE      : BOOLEAN;
        X         : INTEGER;
    BEGIN
        DONE := FALSE;
        GETWORD(LISTNAME);
        IF N >= MAXLIST
            THEN BEGIN
                WRITELN ('MAXIMUM ALLOWED LISTS');
            END;
        X := 1;
        WHILE (X <= MAXLIST) AND (NOT DONE) DO BEGIN
            IF STATUS  [X] = 'F' THEN
                IF TITLES  [X] = LISTNAME
                    THEN BEGIN
                        WRITELN('ALREADY HAS BEEN CREATED');
                        DONE := TRUE
                    END;
            X := SUCC(X)
        END;
        X := 1;
        WHILE (X <= MAXLIST) AND (NOT DONE) DO BEGIN
            IF STATUS  [X] = 'E'
                THEN BEGIN
                    STATUS [X] := 'F';
                    TITLES [X] := LISTNAME;
                    N := N + 1;
                    DONE := TRUE
                END;
            X := SUCC(X)
        END;
    END;
                                        (*          *)
PROCEDURE DELETEPROC;
    VAR
        LISTNAME : PACKCHARS;
        POS      : 0..MAXLIST;
    BEGIN
        GETWORD(LISTNAME);
        POS := LOCATE(LISTNAME);
        IF POS = 0
            THEN
                WRITELN('DOES NOT EXIST')
            ELSE BEGIN
                STATUS[POS] := 'E';
                TOTAL[POS] := 0;
                N := N - 1;
            END;
    END;
                                        (*          *)
PROCEDURE ADDENT(ENTITEM: PACKCHARS; POS: INTEGER);
    VAR
        INSERTPOS, NEWPOSITION: 1..MAXENTRIES;
    BEGIN
        INSERTPOS := TOTAL[POS] + 1;
        NEWPOSITION := FINDINDX(POS, INSERTPOS);
        ENTRIES[NEWPOSITION] := ENTITEM;
        TOTAL[POS] := INSERTPOS
    END;
                                        (*          *)
PROCEDURE ADDPROC;
```

the PDL in the design phase apparently included greater detail and were judged to have higher quality designs. For the implementation phase the PDL and flowchart were found to be equally comprehensible although subjective ratings indicated a mild preference for the PDL. The authors indicate that data structure issues were not addressed, but that this would be a worthwhile topic for further research.

Sheppard, Kruesi, and Curtis [9] compared comprehension with nine forms of program description. Natural language, a program design language, and flowchart symbols were prepared in three spatial arrangements: sequential (vertical flow), branching (flowchart style), and hierarchical (treelike). Subjects did not have access to the program text. Different results were obtained for different types of questions, but no style appeared to dominate.

Sheppard and Kruesi [8] studied program coding from the nine documentation forms and found that the program design language and the flowchart symbol notations were more helpful than the natural language descriptions. The spatial arrangement did not significantly affect the outcome but the branching style appeared to be superior. Brooke and Duncan [1, 2] found that flowcharts were more useful than a program listing in tracing execution sequences in a debugging task.

Although texts and industrial reports emphasize control flow documentation strategies such as macro flowcharts and pseudocode, it seems clear that in some instances the data structure information may be more useful. Some programs do have complex control flow with relatively simple data structures—traditional numerical analysis programs might be an example. But other programs have complex data structures with relatively simple control flow—traditional commercial applications with multiple record structures might be an example.

A pressing and practical problem is to find out what documentation aids assist programmers in comprehending, debugging, and modifying programs. For a knowledgeable programmer with a program listing, supplementary documentation must perform a different function than for a programmer or nonprogrammer with no listing. While detailed flowcharts may be preferable to prose in some problem solving situations, when the program listing is available macro flowcharts, pseudo-

```
VAR
    POS    : 0..MAXLIST;
    LISTNAME  : PACKCHARS;
    ENTITEM   : PACKCHARS;
    COUNTER, INDEX  : INTEGER;
    INVALID   : BOOLEAN;
BEGIN
    INVALID := FALSE;
    GETWORD(LISTNAME);
    GETWORD(ENTITEM);
    POS := LOCATE(LISTNAME);
    IF POS = 0
        THEN WRITELN('DOES NOT EXIST')
        ELSE BEGIN
            IF TOTAL[POS] = MAXITEMS
                THEN BEGIN
                    WRITELN('MAXIMUM ALLOWED FILLED');
                    INVALID := TRUE
                END
                ELSE BEGIN
                    COUNTER := 0;
                    WHILE COUNTER <= TOTAL[POS] DO BEGIN
                        COUNTER := SUCC(COUNTER);
                (* GET VALUE FOR FINDINDEX FUNCTION *)
                        INDEX := FINDINDX(POS, COUNTER);
                (* COMPARE ENTRY ITEM TO CURRENT    *)
                (* ITEMS IN LIST                    *)
                    IF ENTRIES[INDEX] = ENTITEM
                        THEN
                            INVALID := TRUE
                    END;
                END;
        END;
    IF NOT INVALID THEN    (* VALID *)
        ADDENT(ENTITEM, POS)
END;
                                    (*          *)
PROCEDURE DISPLAYPROC;
    VAR
        LISTNAME : PACKCHARS;
        POS      : 0..MAXLIST;
        INDX, X  : INTEGER;
    BEGIN
        GETWORD(LISTNAME);
        POS := LOCATE(LISTNAME);
        IF POS = 0
            THEN
                WRITELN('DOES NOT EXIST')
            ELSE BEGIN
                WRITELN(LISTNAME, 'LIST');
                FOR X := 1 TO TOTAL[POS] DO BEGIN
                    INDX := FINDINDX(POS, X);
                    WRITELN('     ',ENTRIES[INDX])
                END;
            END;
    END;
                                    (*        *)
```

```
PROCEDURE PRINTPROC;
    VAR
        CMD : PACKCHARS;
        X   : INTEGER;
    BEGIN
        GETWORD(CMD);
        IF CMD <> 'LISTS'
            THEN WRITELN('INCORRECT COMMAND')
            ELSE BEGIN
                FOR X := 1 TO MAXLIST DO
                    WRITELN('     ',TITLES[X])
            END
    END;
                                    (*            *)
FUNCTION CHECKCMD(VAR CMD: PACKCHARS): INTEGER;
    BEGIN
        IF CMD = 'CREATE' THEN CHECKCMD := 1
            ELSE IF CMD = 'DELETE' THEN CHECKCMD := 2
            ELSE IF CMD = 'ADD' THEN CHECKCMD := 3
            ELSE IF CMD = 'DISPLAY' THEN CHECKCMD := 4
            ELSE IF CMD = 'PRINT' THEN CHECKCMD := 5
            ELSE CHECKCMD := 0
    END;
                                    (*          *)
                                    (*          *)
PROCEDURE GETLEGALCMD(VAR CMD: PACKCHARS);
    VAR
        CODE : 0..6;
    BEGIN
        CODE := CHECKCMD(CMD);
        CASE CODE OF
            0 : WRITELN(CMD,'IS NOT A LEGAL COMMAND');
            1 : CREATPROC;
            2 : DELETEPROC;
            3 : ADDPROC;
            4 : DISPLAYPROC;
            5 : PRINTPROC
        END;
    END;
                                    (*          *)
                (*   MAIN PROCEDURE   *)
    BEGIN
        N := 0;
        FOR X := 1 TO MAXLIST DO BEGIN
            STATUS[X] := 'E';
            TOTAL[X] := 0
        END;
        GETWORD(CMD);
        WHILE NOT EOF(INPUT) DO BEGIN
            GETLEGALCMD(CMD);
            GETWORD(CMD)
        END;
    END.
```

code, or data structure diagrams may become more useful. We conducted two experiments to explore the utility of several documentation materials as an aid to programmer comprehension of an available listing.

2. Experiment 1: Pseudocode vs Data Structure Diagram

This experiment, conducted by Betty Mastorakis and Karen Schlossberg, tested the utility of pseudocode and data structure diagrams as aids to program comprehension. For this 1 × 3 factorial design, subjects were presented with a coded program under one of three experimental conditions:

(1) Program with input specifications only.
(2) Program, input specifications plus pseudocode.
(3) Program, input specifications plus graphic representation of data structures used.

2.1. Materials

Experimental materials included a program, one of the three program supplements, and a comprehension test. The 223 line program written in Pascal, was an interactive system to maintain lists of items (Figure 1). Seven procedures and three functions were included in the program. The program is a typical student program similar in quality to commercial software. The application domain was chosen to be familiar to our undergraduate student participants. The first of three program supplements (one for each condition) was a sheet containing the format of the five possible input commands (Figure 2). The second supplement was the same input specifications plus one and a half pages of pseudocode (Figure 3). The third supplement was the same input specifications plus a pictorial layout of the four arrays used in the program with arrows indicating the relationship among the arrays (Figure 4).

The comprehension test (Figure 5) provided specific input commands to be used as a continuous stream when answering the questions (instructions explaining this were included in the test.) The beginning questions required tracing the control flow in the program so as to introduce participants to the program's function and to demonstrate the input command usage. Later questions were designed to test overall comprehension of the program logic.

2.2. Administration

A pilot study with 30 students was conducted during a 50 minute session of an intermediate undergraduate

Fig. 2. Input specifications only for Program UPKEEPER (used for Experiments 1 and 2).

Program: UPKEEPER

Data for the program consist only of input commands. The following is the format for the commands which are used in the program:

```
CREATE  ⟨listname⟩  (*Creates a new list*)
DELETE  ⟨listname⟩  (*Deletes a current list*)
ADD  ⟨listname⟩  ⟨entry⟩  (*Adds a new entry to the given list*)
DISPLAY  ⟨listname⟩  (*Output commands*)
PRINT LISTS
```

Fig. 3. Input specifications plus pseudocode for Program UPKEEPER (used for Experiments 1 and 2).

Program: UPKEEPER
(algorithm)

Data for the program consists of only input commands. The following is the format for the commands which are used in the program:

```
CREATE  ⟨listname⟩  (*Creates a new list*)
DELETE  ⟨listname⟩  (*Deletes a current list*)
ADD  ⟨listname⟩  ⟨entry⟩  (*Adds a new entry to given list*)
DISPLAY  ⟨listname⟩  (*Output commands*)
PRINT LISTS
**********
```

Main Procedure
```
Initialize Status array and entry Total array
While there are more commands
  Read command
  Determine command type
  Perform procedure for appropriate command type
End.
```

Create Procedure
```
Read list name
If maximum number of lists already exists
  Print list name given and an appropriate message
Else
  Add list name to Title array of list names
  Change status for list added
End.
```

Delete Procedure
```
Read list name
Search for given list name in Title array of list names
If not found
  Print list name and appropriate message
Else
  Change corresponding element in Total array to 0
  Change status for given list
End.
```

Add Procedure
```
Read list name and entry item given
Search Status array for indication of a current list name
If list name given is not found
  Print given list name and appropriate message
Else
  If list contains maximum number of entries
    Print appropriate error message
  Else
  Search current list of entries to find given entry item
  If found
    Print given entry item and appropriate error message
  Else
    Find position where entry item it to be added
    Insert entry item at this position
    Increment number of current entry items by 1
  End
End.
```

Display Procedure
```
Read list name given
Search Status array for indication of a current list name
If given list name is not found
  Print given list name and appropriate error message
Else
  Print list name and entries
End.
```

Print Procedure
```
Read command flag
If not equal to LISTS
  Print error message
Else
  Search Status array for indication of current list name
  Print current list names
End.
```

programming course using Pascal. The students had no training in data structure diagrams although pseudocode was occasionally used in the course. Test materials were revised, the task was simplified, and the time extended.

The experiment was conducted in November 1979 during a 75 minute class period with 57 different students from the same intermediate programming course. Experimental consent sheets were provided for all students to sign. All students present agreed to participate. Students were told to use the entire class period if necessary, to please remain seated if finished early, to ask any questions they had during the test, and to feel free to write on the blank sheet provided or on the other materials if they desired. Also, students were told that they were each given a program supplement (below the cover sheet in the packet) which should be used as an aid when answering questions. However, because only a few students were actually using the supplement while taking the test, students were reminded that the supplement would be useful, and they then began using the aid.

Several questions were asked during the administration of the test. Because of typographical errors on the test, students were twice interrupted to make note of the necessary changes on the test. One student questioned the use of input commands as continuous input, but seemed to understand this idea after a brief explanation.

Most students needed the entire class time to complete the test. After the test, a number of students asked whether the program actually ran and also asked to keep a copy of the program.

2.3. Results

Initially there were 19 students for each of the three conditions. It was decided that if any of the students completed less than the first seven questions, their tests would be excluded from the analysis. Two such cases occurred in the data structures group and one in the

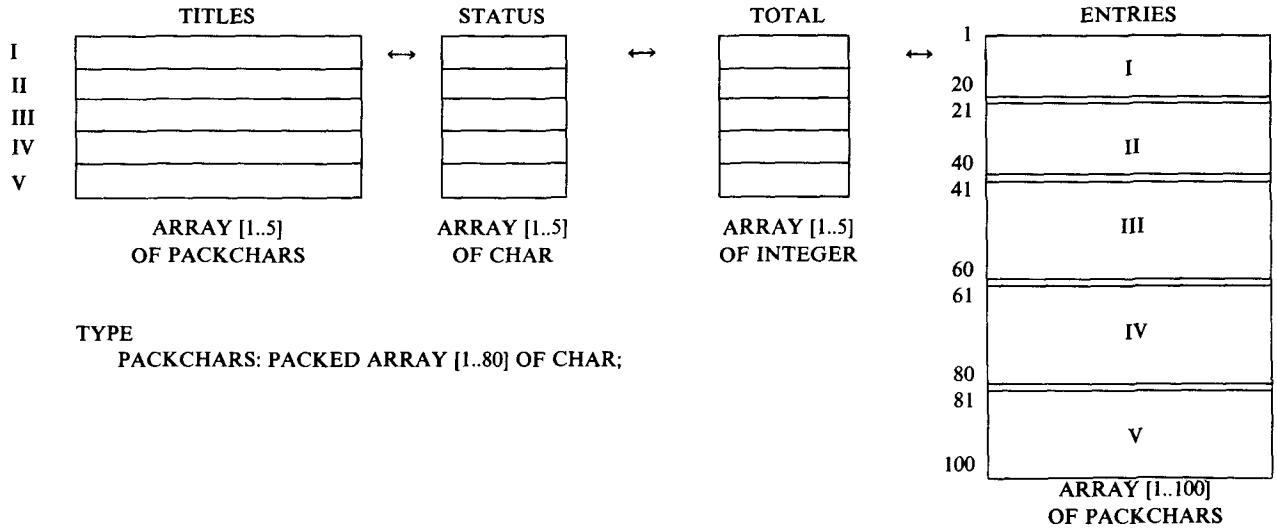
Fig. 4. Input specifications plus data structure diagram for Program UPKEEPER (used for Experiments 1 and 2).

Program: UPKEEPER

Data for the program consist only of input commands. The following is the format for the commands which are used in the program*:

```
CREATE  ⟨listname⟩  (*Creates a new list*)
DELETE  ⟨listname⟩  (*Deletes a current list*)
ADD  ⟨listname⟩  ⟨entry⟩  (*Adds a new entry to the given list*)
DISPLAY ⟨listname⟩  (*Output commands*)
PRINT LISTS
```

The following are the data structures for UPKEEPER. Consider them as a pictoral representation of the arrays used in the program:




```
TYPE
    PACKCHARS: PACKED ARRAY [1..80] OF CHAR;
```

specifications-only group. Such occurrences could have resulted from a lack of motivation to complete the test or possibly from a serious lack of understanding of the program and test. Those who answered only the first few questions may not have been willing to invest the time and effort to complete the 15-question test and, consequently, their incomplete results may have demonstrated low motivation rather than poor comprehension; these cases were discarded. To maintain equal numbers of students for the three groups, two tests from the pseudocode and one from the specifications-only condition were randomly chosen to be excluded.

Each question was graded as right or wrong; one point was given for each correct answer and the score was unaffected by an incorrect answer. Most questions were multiple choice but for those requiring a short answer, acceptable correct responses were established before grading to maintain objectivity. Any uncertainties were discussed by the experimenters to adhere to these standards.

Table I presents the mean number of correct scores and the standard deviation per group for each of 15 test questions. Two statistical analyses were performed. The first was a one-way analysis of variance comparing results for the three experimental treatments. This analysis revealed a statistically significant difference among the groups at the 0.01 level.

Two-tailed t-tests were performed in pairs. The data structures/pseudocode analysis and the data structures/specifications-only analysis were both significant ($p < 0.01$) but the specifications-only/pseudocode analysis did not yield a significant difference. These results reveal that for this program external documentation did improve comprehension. The data structure diagram was more helpful than the pseudocode.

Comprehension test questions were designed to assess understanding of low level details (questions 1 to 7) such as variable assignments and execution sequencing and higher level concepts (questions 8 to 15) such as array use and procedure purpose. For questions focusing on array usage such as number 11, we might not be surprised to find that the data structure diagram group did better. However, it is striking that the data structure diagram group had higher mean scores on *every* question, even those focusing on procedural details.

These results strongly suggest that for this program and these subjects, at least, the data structure diagram facilitated overall program comprehension. This is in spite of the greater detail and volume of information in the pseudocode.

In focusing on only two common documentation aids, pseudocode and data structure diagrams, we confounded the issue of textual vs graphic presentations of information. To deal with this as an independent variable we conducted a second experiment.

3. Experiment 2: Control Flow vs Data Structure in Textual vs Graphical Formats

This experiment, conducted by Toni Deliso and Gary Stambaugh, tested program comprehension aids by com-

Fig. 5. Multiple choice test (used for Experiments 1 and 2).

The following questions pertain to the program UPKEEPER. Consider the commands given between questions as a continuous stream of input (i.e., as if they appeared together as data for the program.)

For each of the multiple choice questions which follow, select the one best alternative by circling the letter which precedes it.

Assume the input command CREATE CARS is the first for this program.

1. In the main procedure, beginning with GETWORD (CMD), what sequence of execution takes place before CARS is read?
   (a) PROCEDURE GETLEGALCMD, FUNCTION CHECKCMD, PROCEDURE CREATPROC.
   (b) PROCEDURE GETLEGALCMD, FUNCTION CHECKCMD, PROCEDURE GETLEGALCMD, PROCEDURE CREATPROC.
   (c) PROCEDURE GETLEGALCMD, FUNCTION CHECKCMD, MAIN PROCEDURE, PROCEDURE CREATPROC.
   (d) PROCEDURE GETLEGALCMD, PROCEDURE CREATPROC.
2. In PROCEDURE CREATPROC, what will the variable X signify after execution of this procedure?
   (a) Current number in MAXLIST.
   (b) N is changed to assume the value of 1.
   (c) First empty position where LISTNAME will be added.
   (d) X will always assume the value 1 after execution.

   The next two commands in the data are: CREATE ROLLS
   ADD ROLLS SCROLL
3. Within PROCEDURE ADDPROC, the FUNCTION LOCATE returns:
   (a) First occurrence of "F" in STATUS.
   (b) Position where LISTNAME will be added.
   (c) Position of LISTNAME.
4. With the above two commands, this function LOCATE in PROCEDURE ADDPROC returns the value.
5. In this same procedure, ADDPROC, the function FINDINDX returns the value
6. With the above two commands, in PROCEDURE ADDPROC, control will be transferred to PROCEDURE ADDENT.
   (a) TRUE
   (b) FALSE
7. Assuming control is now in PROCEDURE ADDENT, what is the first value assigned to INSERTPOS?
8. Why is the number 20 used in the formula in FUNCTION FINDINDX? (briefly state)
9. Why was TOTAL initialized to 0 at the start of the program in the main procedure? (briefly state)

   Next command in the input data is: ADD ROLLS GOAL
10. When execution of ADDENT is completed for the above command, the value of TOTAL [POST] will be
11. What does TOTAL array indicate? (briefly state)

    Next command in input data is: DISPLAY ROLLS
12. What output will this command generate?

    Next command in input data is: DELETE CARS
13. After this command:
    (a) LISTNAME still exists in TITLES array.
    (b) LISTNAME can no longer be accessed.
    (c) STATUS [POS] in PROCEDURE DELETEPROC will be the location of the next list added.
    (d) All of the above.
    (e) None of the above.
14. Briefly state the purpose of STATUS in the program (without using code to answer the question):
15. PROCEDURE PRINTPROC is to print a list of current list titles. As it is written, the procedure may also print titles which are not current. What modification (addition or deletion) must be made in this procedure so that only active titles will be printed?

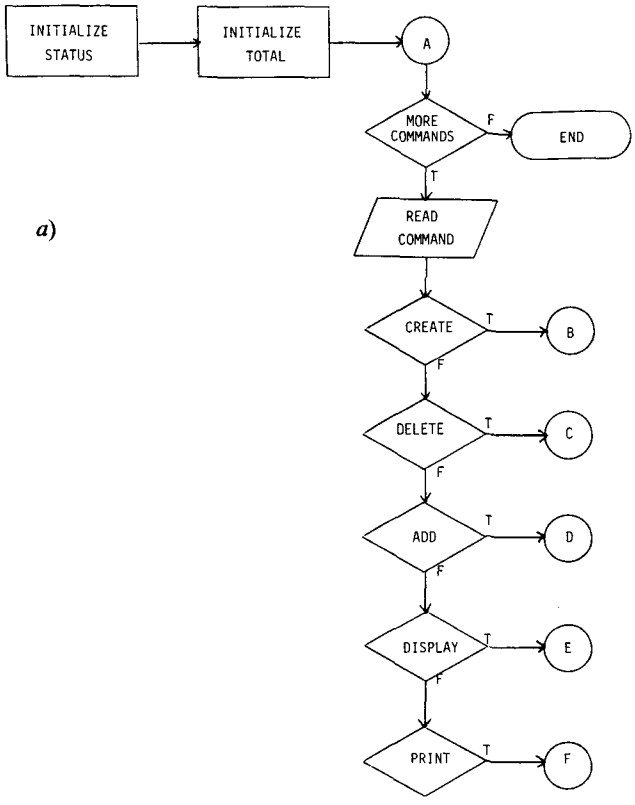
Table I. Numbers of Correct Scores on Comprehension Test for Experiment 1 (15 questions, 51 subjects, 17 per cell).

	MEAN (STANDARD DEVIATION)
DATA STRUCTURE DIAGRAM + SPECIFICATIONS	8.47 (2.48)
PSEUDOCODE + SPECIFICATIONS	6.06 (1.68)
SPECIFICATIONS ONLY	5.06 (2.59)

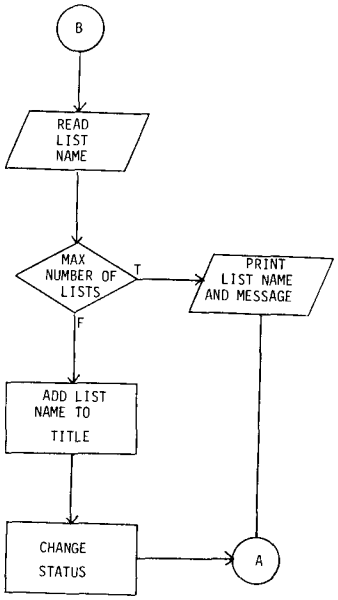
paring the effect of presentation of the control flow in pseudocode and flowchart form and the presentation of the data structures in textual and graphic formats. Our hypothesis was that the relatively high level data structure information would be more useful to programmers than the control flow information. We also believed that the type of representation, written or graphic, would have no effect on program comprehension.

Fig. 6. Flowchart for Program UPKEEPER (used for Experiment 2 only).

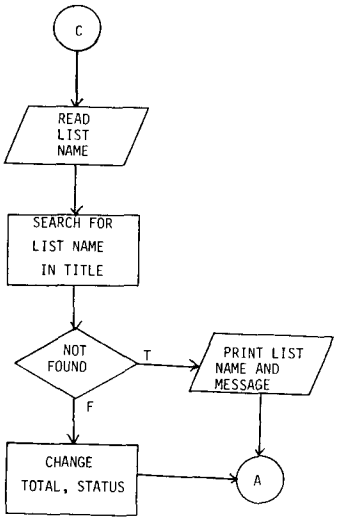
a)



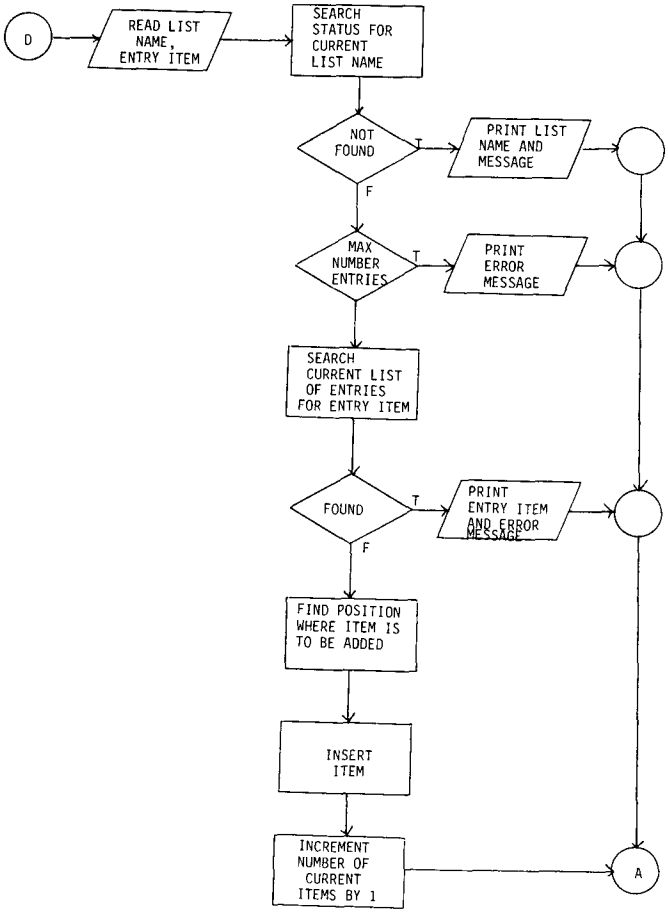
b)


B
READ LIST NAME
MAX NUMBER OF LISTS
T
F
PRINT LIST NAME AND MESSAGE
ADD LIST NAME TO TITLE
CHANGE STATUS
A

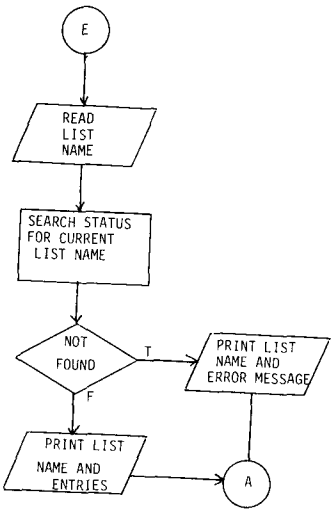
c)


C
READ LIST NAME
SEARCH FOR LIST NAME IN TITLE
NOT FOUND
T
F
PRINT LIST NAME AND MESSAGE
CHANGE TOTAL, STATUS
A

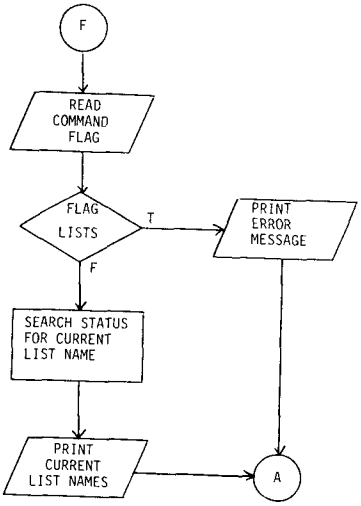
d)


D
READ LIST NAME, ENTRY ITEM
SEARCH STATUS FOR CURRENT LIST NAME
NOT FOUND
T
F
PRINT LIST NAME AND MESSAGE
MAX NUMBER ENTRIES
T
PRINT ERROR MESSAGE
SEARCH CURRENT LIST OF ENTRIES FOR ENTRY ITEM
FOUND
T
F
PRINT ENTRY ITEM AND ERROR MESSAGE
FIND POSITION WHERE ITEM IS TO BE ADDED
INSERT ITEM
INCREMENT NUMBER OF CURRENT ITEMS BY 1
A

e)


E
READ LIST NAME
SEARCH STATUS FOR CURRENT LIST NAME
NOT FOUND
T
F
PRINT LIST NAME AND ERROR MESSAGE
PRINT LIST NAME AND ENTRIES
A

f)


F
READ COMMAND FLAG
FLAG LISTS
T
F
PRINT ERROR MESSAGE
SEARCH STATUS FOR CURRENT LIST NAME
PRINT CURRENT LIST NAMES
A

3.1. Materials

For this 2 × 2 factorial design, participants were presented a packet consisting of:

(1) An introduction and explanation of the experiment.
(2) One of four program design aids (i.e. flowcharts, pseudocode, data structure diagram, or description of the data structures.)
(3) The program listing.
(4) A comprehension test.

All participants were given the same program as in Experiment 1 but the comprehension test was shortened by omitting question 15. Each program supplement contained the format of the five possible input commands. The first of the four program supplements was a set of six flowcharts for the program (Figure 6). The second supplement was the one and a half page pseudocode (Figure 3). The flowcharts and pseudocode were presented at the same level of detail with the same terminology. The third supplement was the diagram of the four arrays used in the program and arrows indicating the relationship among the arrays (Figure 4). The fourth supplement was a written description of the data structures. This gave the title of each array and a brief description of what each array represented (Figure 7).

3.2. Administration

The experiment was conducted in November 1980 during three 50 minute discussion sections of the same intermediate programming course in Pascal. Subjects were introduced to the experiment at the beginning of the class; they had no prior knowledge that they would be participating in an experiment. All 32 students signed the experimental consent sheets which were provided. Participants were told to use the entire class period if necessary, to remain seated if finished early, to ask any questions they had during the test, and to feel free to write on any of the materials if they desired. Also, participants were told that they were each given a program supplement (below the cover sheet in the packet) which should be used as an aid when answering questions.

Most students needed the entire class time to complete the test. After the test, a number of participants asked to keep a copy of the program. In short, the administration was similar to Experiment 1. No student participated in both studies.

3.3. Results

One point was given for each correct answer and the score was unaffected by an incorrect answer. The test consisted of 14 multiple choice and short answer questions. Most of the questions were multiple choice, but for those requiring a short answer, acceptable correct responses were established before grading.

All 32 subjects were used in the experiment. These students were divided in four groups corresponding to the four program aids. No data were thrown out and low scores were kept. Table II presents the mean number of correct answers and standard deviations by group for the total of 14 questions.

Fig. 7. Data structure description for Program UPKEEPER (used for Experiment 2 only).

```
The following is a description of the data structures for upkeeper.
This program uses 4 arrays.
The array (entries) consists of 100 entries, each entry may be from 1
to 80 characters long. The array (entries) is divided into 5 separate
lists. Each list contains at most 20 entries.
These 5 lists are represented by a separate array (titles). The title of
each list can be 1 to 80 characters long.
The status, (e, f), of each list is represented by the array (status).
The total number of entries in each list is represented by the array
(total).
```

The two-way analysis of variance showed a significant ($p < 0.02$) main effect for information content (data structure vs control flow information) but no effect for the form (textual vs graphic). The interaction effect was also not significant. These results support the hypothesis that for this program, at least, the data structure information was more helpful than the control information and that the form of presentation does not matter. These results are impressive since the control flow information was far *more* detailed than the data structure information.

4. Conclusions

In this study we sought to test empirically several forms of external program documentation to assess their impact on comprehension of a program listing. Earlier results with detailed standard flowcharts showed them to be ineffective and a possible distraction from the program text. Higher level semantic information was conjectured and demonstrated to facilitate comprehension. A brief data structure description or half-page diagram were shown to be more helpful to comprehension of a 223 line Pascal program than a one and a half page pseudocode or six-page macro flowchart. The effectiveness of external documentation was amply demonstrated in Experiment 1 where the specifications-only group had the poorest performance.

Table II. Number of Correct Scores on Comprehension Test for Experiment 2 (14 questions, 32 subjects, 8 per cell).

MEAN (STANDARD DEVIATION)		FORMAT TEXTUAL	FORMAT GRAPHIC
CONTENT	DATA STRUCTURE	7.75 (4.49)	8.37 (4.40)
	CONTROL FLOW	4.50 (3.50)	3.87 (4.05)

For this program, which used four arrays to manage data for an interactive list keeping system, the data structure information was more difficult to extract from the code than the control flow information. Lengthy detailed flowcharts or voluminous internal comments do not appear to aid competent programmers when the code is available. In fact, excess documentation does interfere with comprehension.

Henry Ledgard, in a personal communication, argues that this program could be redesigned in a more lucid way, in particular

(1) making better use of Pascal's data definition facilities;
(2) choosing better mnemonic variable names;
(3) giving more careful attention to global variables.

Ledgard argues that such an improved program would be comprehensible without the use of data structure diagrams. Just as the evolution towards the use of higher level control structures has reduced the utility of detailed flowcharts, the use of higher level data structures would reduce the utility of data structure diagrams.

Replications of this study with other programmers, programs, and documentation forms would be useful to obtain a more precise understanding of under which conditions data structure information is more helpful than control flow information. It might be interesting to study control flow information which is less detailed than the materials used in these experiments.

The syntactic/semantic model and the results of this experiment suggest that carefully designed high level semantic information is useful as external documentation. The compactness brought about by high level abstraction appears to aid comprehension too. Maybe a crude rule of thumb would be to encourage external documentation to be one-tenth the size of the code while using problem domain terminology as much as possible. Of course, exceptionally good external documentation will only partially compensate for poorly written code. Functionally oriented modular design, carefully composed algorithms, meaningful variable names, and well-chosen brief comment blocks in the code are necessary components of quality programs.

Competent programmers can judge which aspects of their program require external documentation. Control flow or data structures are two familiar aspects but others such as timing information in real-time environments or interprocedure coordination in multiprocessor architectures are worthy of consideration. External documentation should provide a guiding abstraction which is above the level of the code. For large programs two or even three levels of abstraction or detail may be necessary.

Acknowledgments. I would like to thank J. Gannon, M. Weiser, and L. Chmura for their comments on earlier versions of this paper. H. Ledgard was extremely effective as the section editor by providing numerous reviews and useful guidance in responding to the referees' divergent comments. I thank M. Johnson for her careful typing and preparation of figures as she cheerfully endured the multiple revisions.

Received 1/81; revised 5/81; accepted 7/81

References

1. Brooke, J.B. and Duncan, K.D. An experimental study of flowcharts as an aid to identification of procedural faults. *Ergonomics, 23,* 4 (1980) 387–399.
2. Brooke, J.B. and Duncan, K.D. Experimental studies of flowchart use at different stages of program debugging. *Ergonomics, 23,* 11, (1980), 1057–1091.
3. Fitter, M. and Green, T.R.G. When do diagrams make good computer languages? *Int. J. of Man-Machine Studies, 11,* (1979), 235–261
4. Ramsey, H.R., Atwood, M.E., and Van Doren, J.R. A comparative study of flowcharts and program design languages for the detailed procedural specification of computer programs. (ARI Tech. Rept. TR-78-A22) Colorado: Science Applications, Inc., U.S. Army Research Institute for the Behavioral and Social Sciences, 1978.
5. Shneiderman, B. *Software Psychology: Human Factors in Computer and Information Systems,* Winthrop, Cambridge, MA, 1980.
6. Shneiderman, B. and Mayer, R. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *Int. J. of Computer and Information Sciences, 7,* (1979) 219–239.
7. Shneiderman, B., Mayer, R., McKay, D., and Heller, P. Experimental investigations of the utility of detailed flowcharts in programming. *Comm. ACM, 20,* (1977) 373–381.
8. Sheppard, S.B. and Kruesi, E. The effects of the symbology and spatial arrangement of software specifications in a coding task. *Proc. Trends and Applications 1981: Advances in Software Technology.* Held at NBS, Gaithersburg, MD, available from IEEE, (1981) 7–13.
9. Sheppard, S.B., Kruesi, E. and Curtis, B. The effect of symbology and spatial arrangement on the comprehension of software specifications. *Proc. 5th Int. Conf. on Software Engineering.* San Diego, CA, available from IEEE, 1981, 207–214.