Program Readability: Procedures Versus Comments

TED TENNY

Abstract—**A 3 × 2 factorial experiment was performed to compare the effects of procedure format (none, internal, or external) with those of comments (absent or present) on the readability of a PL/I program. The readability of six editions of the program, each having a different combination of these factors, was inferred from the accuracy with which students could answer questions about the program after reading it. Both extremes in readability occurred in the program editions having no procedures: without comments the procedureless program was the least readable and with comments it was the most readable.**

Index Terms—**Comments, experimental testing of readability, experiment design, procedure format, procedures: internal versus external, program readability.**

Introduction

CODING is a small part of software engineering but an important part where program maintenance is concerned. The more readable the coding is, the more quickly and accurately a programmer can obtain critical information about a program by reading the program text. Thus readability is defined within the context of maintenance: a program is readable if information needed to maintain it is easily found by reading the code. Elshoff and Marcotty [2] proposed adding another step to the program modification cycle, in which the program is modified to make it more readable. Their proposal was motivated by the high cost of program maintenance, which depends on the ability of maintenance programmers to read and understand the code.

Results of the author's previous experiment [9] suggest that comments and inline code (as opposed to internal procedures) improve the readability of the Banker's Algorithm. This experiment featured the Banker's Algorithm coded in Pascal with the code for subtasks either inline or as internal procedures, and with or without comments. The procedureless edition with comments was the most readable, while the edition with internal procedures and no comments was the least readable. This unexpected loss of readability with internal procedures prompted the investigation of both internal and external procedures in the present experiment.

Background

Advocates of separate compilation claim that external procedures improve readability because the text of each procedure appears as a unit, uninterrupted by internal procedures. Yet languages such as Pascal, which have internal procedures, are cited for their high readability. If a computer program has a modular organization with procedures, can comments improve its readability or are they superfluous? Can comments rescue a program which is not modular, and make it as readable as a modular program? Is a program more readable with internal procedures or with external procedures? This experiment was designed to test the hypothesis that two factors—procedure format (none, internal, or external), and comments—have no effect on program readability.

There is a qualitative difference between reading a program with internal procedures, reading a program with external procedures, and reading a program with all of the code for logical functions inline. If each procedure is invoked from only one place then a program with procedures is necessarily longer than the same program with all of the code inline, because the declarations of procedures and their formal parameters are absent in the procedureless program. Although it is longer the program with procedures ought to be a more readable, because the procedure boundaries isolate the code for each logical function. But the effect of procedures may be important only when programs exceed a certain minimal size, as suggested by the Banker's Algorithm experiment and this experiment.

Yourdon and Constantine [13] described lexical inclusion of one module inside another a mild form of content coupling. Such content coupling does not occur with external procedures. On this basis one would expect a program with external procedures to be more readable than the corresponding program with internal procedures.

While comments have been a factor in some program readability studies, only a few experiments have suggested which kind of comments are most effective at promoting program readability. The effect of procedure format has only been debated, without experimentally testing the readability of internal versus external procedures. Dijkstra [1] reasoned that the ease of understanding a program depends critically upon the simplicity of its sequencing control, and proposed a strict sequencing discipline to keep programs from becoming too complex to be understood. Kernighan and Plauger [4] presented rules of programming style as a guideline for producing understandable code. The rules are illustrated by programming examples, many of which feature comments or subroutines. Of Ledgard's 26 Programming Proverbs [5], 5 are clearly directed toward making programs more readable.

Weissman [10], [11] compared the effects of program indentation and comments in PL/I experimentally and found a significant interaction between them: in the pres-

Manuscript received January 23, 1986; revised June 17, 1986.

The author is with the Department of Computer Science, Texas Christian University, Fort Worth, TX 76129.

IEEE Log Number 8822450.

ence of comments, two-column indentation made the programs less readable. These programs contained GOTO's. Norcio's experiments [6] used the Cloze technique, requiring subjects to reconstruct source statements that were replaced by blank lines in a Fortran program, to test the effects of documentation and identation. The interaction between documentation and indentation was significant in both experiments, with the best comprehension resulting from indented programs having one line of documentation interspersed with the code. Shneiderman [8] performed two experiments to assess the utility of external documentation, such as macro flowcharts, pseudocode, and illustrations of data structures. The latter were found to be more helpful to students in understanding a program, although this information might have been equally helpful if it had appeared as comments within the code. The present experiment tests the effect of providing this information as comments rather than as external documentation.

Experiment Design

The more readable a program is, the more quickly and accurately a programmer can obtain information about it by reading the program text. Thus readability is inferred from the speed and accuracy with which experimental subjects can answer questions about a program after reading it. Readability is expressed as the average number of right answers to a series of questions about the program in a given length of time. This score certainly depends on the questions as well as on the program, so the same questions are asked about different editions of the same program to obtain comparative scores.

The program chosen for this experiment is an athletic record-keeping program which reads a series of team performance records, sorts them on the basis of the team's total points, and prints the current standings. It is coded in PL/I, because PL/I provides both internal procedures and external procedures. The standings program is sufficiently complex that it cannot be understood in just a few moments, but short enough that most students can answer the questions in an hour.

Factors

The experiment has a 3 × 2 factorial design [7] as shown in Table I. Procedure format is the first factor: editions 2, 3, 4, 5 of the program have procedures to perform its major subtasks, while editions 0 and 1 have no procedures: all of their code for subtasks has been merged into the main program. The procedures in editions 4 and 5 are external, each compiled independently by the PL/I compiler, while the procedures in editions 2 and 3 are internal, i.e., each procedure is lexically enclosed within the main program or within another procedure.

Comments are the second factor. The comments in editions 1, 3, 5 were designed to briefly describe the purpose of each procedure (editions 3, 5) or each block of code (edition 1, which has all of the code inline), and to graphically illustrate the principal data structures. The same comments are used in each of these editions. The data structure comments were inspired by Shneiderman's illustrations of data structures [8] which improved program comprehension when presented as external documents. The other descriptive comments are in a style suggested by Norcio's experimental results [6] and by Kernighan and Plauger [4]. The comments do not explain PL/I, nor do they directly address any of the questions which accompany the program. Instead they seek to clarify the algorithms and data structures. Editions 0, 2, and 4 have no comments at all.

TABLE I
Six Editions of the Standings Program

Editions	Attribute
0, 1	The standings program is expressed as a single procedure: all of the code for subtasks has been merged into the text of the main program STAND.
2, 3	The standings program is expressed as the main program STAND, containing internal procedures FETCHD, PRINT1, SORT, LISTOUT. FETCHD contains the internal procedure DECODE.
4, 5	The standings program is expressed as the main program STAND, accompanied by the external procedures FETCHD, DECODE, PRINT1, SORT, and LISTOUT, each of which is compiled independently.
0, 2, 4	There are no comments.
1, 3, 5	Comments have been added to describe the algorithms and explain the purpose of each section of code, and to graphically illustrate the principal data structures.

or

Edition	Attributes
0	inline code, no comments
1	inline code, comments
2	internal procedures, no comments
3	internal procedures, comments
4	external procedures, no comments
5	external procedures, comments

Each of these editions was tested on the computer. They compile and execute with no error messages, and produce the expected results. Each edition has the same variable names and the same three-column indentation. The text of editions 0 and 5 is shown in the Appendix, along with the questions.

The questions which accompany these program editions are the same, except that references to line numbers were adjusted to match the line numbers of each edition. All of the questions are in short-answer format. They are concerned with the control and data structures of the program and certain details of its execution with given inputs. The questions are language-independent: they would be essentially the same if the standings program were coded in any other high-level language.

Subjects

The subjects for this experiment were students enrolled in software engineering at the University of Oklahoma in the Spring 1985, and in the Fall 1985. These students were mostly seniors. All of them had at least six programming courses involving Pascal, Fortran, Cobol, and assembly languge, and some of them, with programming experi-

ence in business and industry, had become very skillful programmers. Thus the subjects were a fairly homogenious group in terms of their academic preparation but they varied somewhat, as expected, in terms of programming experience and skill.

Most of the students had no experience in PL/I, because it was not used in their earlier computer science courses. The standings program, however, is coded in easy PL/I. SUBSTR, the only built-in function appearing in the program, is used in a straightforward way. The more esoteric features of PL/I do not appear, so that the PL/I edition of the standings program looks like a Pascal or Modula 2 edition.

None of the students had any prior knowledge of the standings program.

Since programmer's skill was not intended to be a factor in the experiment, the cells were constructed to avoid any concentration of the best or worst students in a single cell. First the students were ranked by the sum of their overall grade point average (GPA) plus their grade in data structures and their previous experience (if any) in PL/I. Then each group of 6 students was distributed among the 6 cells, making each cell a cross section of the class in terms of the students' abilities. Each cell contained one of the top 6 students, one of the next 6 students, . . . , and one of the bottom 6 students. Also the cells were approximately balanced between spring students and fall students. This method of cell construction (rather than random selection) was deemed necessary to confound the effect of differences in the students' abilities. If random selection put the best or worst students in the same cell then the results would be statistically significant but would measure the wrong thing—differences in students' abilities instead of differences in the readability of various program editions.

While these experimental subjects are students rather than professional programmers, they are college seniors majoring in Computer Science, so most of them are professionals-to-be. This level of programmer's skill is appropriate for a readability experiment because program maintenance tasks in business and industry are often assigned to new employees who are recent college graduates.

Administration

The experiment was conducted during four 50-minute class periods, two in January 1985, and two in September 1985. There were no repeat students in the Fall semester. The program was handed out on standard computer forms, with line numbers but no other compiler-added information. Questions appeared on separate mimeographed pages. To perform the experiment each student was asked to read one edition of the program and answer the questions provided with it. Students were told that this was an experiment, not an exam, and assured that it would have no effect on their grades. They were asked to read the program carefully, answer the questions, and hand their answers and program listings back at the end of the class period. The experimenter explained that the program compiles and executes with no error messages. Students were permitted to mark on the program printouts and separate the pages. They were told to hand in their answers anonymously (although it was possible to identify individual students later to compute the correlation between scores and students' abilities). Students were advised to work independently. They were reminded that the programs are not identical, so their answers may be different from those of other students. The experimenter's actual words were recorded on tape during the experiment.

All answer sheets and program printouts were collected at the end of the 50-minute class period. Each student's paper was scored by adding the number of right answers to questions 1–12. The scores were tabulated and an analysis of variance (ANOVA) performed using F-tests [12] to determine the statistical significance of the differences between mean scores.

Results

Out of 189 students enrolled in software engineering, 157 took part in the experiment. Unfortunately 9 of these 157 did not notice that there were questions on both sides of the page, so they never attempted to answer questions 8 through 12. Their scores were discarded, leaving 148 experimental observations.

Both extremes in readability occurred in the program editions which have no procedures: the procedureless program with comments (edition 1) is the most readable, while the procedureless program without comments (edition 0) is the least readable. Table II shows the results for each cell, with the simple effects of procedures and comments illustrated in Fig. 1. The main effect of comments is significant at the 0.05 level [$F(1, 142) = 4.34$, $p < 0.05$] as is the simple effect of comments in the procedureless program [$F(1, 142) = 4.52$, $p < 0.05$].[1] The effect of procedure format is not statistically significant, nor is the interaction between procedures and comments. In each case, however, the edition with comments received a higher mean score than the corresponding edition without comments. This is the expected pattern, since the comments provide information that is not immediately apparent from the program text.

The results were further checked by computing the correlations between students' experimental scores and 1) GPA, 2) previous knowledge of PL/I (obtained from a class survey), and 3) grade in Data Structures. Of these the student GPA's were most highly correlated with experimental scores ($r = 0.39$; 95 percent confidence interval $= [0.25, 0.52]$; $p < 0.00001$)[2] while PL/I knowledge and grades in Data Structures were less correlated (r

[1] The F ratio is a measure of (variation between cells)/(variation within cells). If the F ratio of the experimental data, $F(m, n)$, with m degrees of freedom in the numerator and n degrees of freedom in the denominator, is greater than the F distribution of m, n, α, then the probability p of exceeding this F distribution by chance is less than α.

[2] r is the product-moment coefficient of correlation, and p is the probability of obtaining a correlation of this magnitude by chance.

TABLE II
CELL EDITIONS, SIZES (N) MEANS (m), STANDARD DEVIATIONS (s)

	edition 0	edition 1
no	$N = 23$	$N = 24$
procedures	$m = 4.52$	$m = 5.96$
	$s = 1.81$	$s = 2.78$
	edition 2	edition 3
internal	$N = 25$	$N = 26$
procedures	$m = 4.76$	$m = 5.12$
	$s = 2.20$	$s = 2.22$
	edition 4	edition 5
external	$N = 27$	$N = 23$
procedures	$m = 4.96$	$m = 5.61$
	$s = 2.71$	$s = 1.95$
	no comments	comments

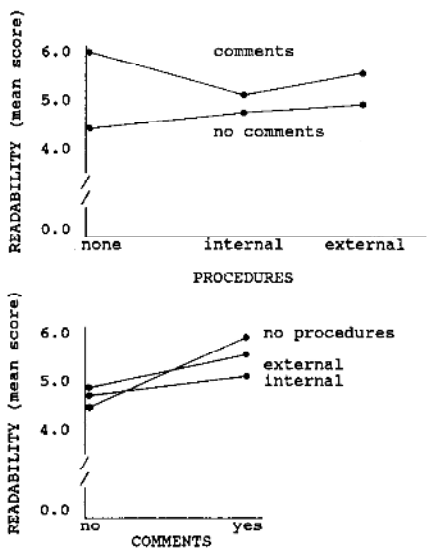

Fig. 1. Simple effects of procedures and comments.

$= 0.14$ and $r = 0.20$, respectively). Most of the students had no prior knowledge of PL/I.

Next, the experiment was repeated with raw scores replaced by scores adjusted for differences in GPA:

$$\text{adjusted score} = \text{raw score} - \text{GPA} * 2.5.$$

This adjustment preserved all of the inequalities between cell means (Table II) but it improved most of the F ratios, so that the simple effect of comments in the absence of procedures was significant at the 0.025 level. These results indicate that the method of cell construction was successful in confounding the effect of differences in students' abilities, even though it systematically added a certain amount of scatter to the data in each cell. This "systematic scatter" made the experimental results with raw data less statistically significant than they otherwise might have been.

Comparison to the Banker's Algorithm Results

These results are qualitatively different from the results of the Banker's Algorithm experiment [9] in which the

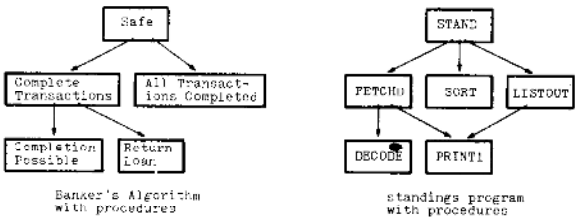

Fig. 2. Referential structure of the Banker's Algorithm and the standings program.

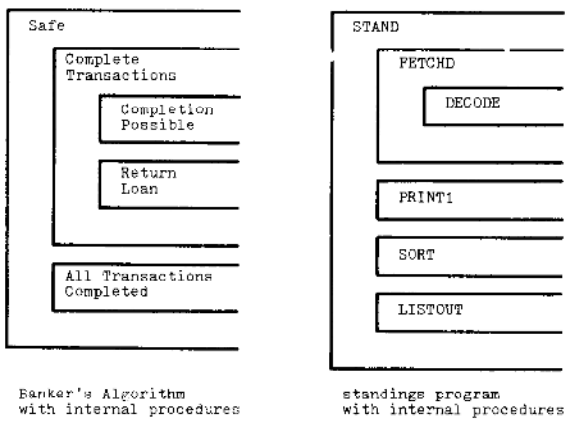

Fig. 3. Lexical structure of the Banker's Algorithm and the standings program, with internal procedures.

procedureless program got higher scores than the program with internal procedures, with or without comments. Both the control structures and data structures of the Banker's Algorithm are more complex than those of the standings program, although the Banker's Algorithm is shorter overall. While the referential and lexical structures of the Banker's Algorithm are no more complex than those of the standings program (Figs. 2, 3), the Banker's Algorithm is purely computational, with no I/O statements and fewer procedure calls. Comparison of the results suggests that these comments are less effective at improving the readability of very short and simple modules. The comments are more helpful when the modules are longer (as in the procedureless editions of both programs) and have more computational complexity (as in the Banker's Algorithm).

Language may have affected the difference between the results of these experiments. The Banker's Algorithm is coded in Pascal using WITH statements, whereas the standings program is coded in PL/I without them. WITH statements avoid the repetition of record names, but the name of the record is lexically separated from the name of its component. Further investigation of such language design decisions (as in Gannon and Horning's experiment [3]) is needed to determine the many effects of language on program readability. The students in both experiments had at least two semesters of Pascal programming experience, but most of them had never used PL/I. It is cer-

tainly easier to understand a program in a familiar language, so one might have expected comments to have been more helpful in the PL/I program (an unfamiliar language) than in the Pascal program. They were not. Clearly the effect of language familiarity (as opposed to the inherent readability of the language itself) bears further investigation.

Conclusions

For this particular program, procedures have little effect on readability. There are compelling reasons to believe that a large program is more readable with the modules expressed as separate procedures [1], [3], but the standings program is apparently not large enough to show this effect.

The author's comments improve the readability of the standings program, supporting Shneiderman's results with graphic data structure descriptions [8] and Norcio's results with one-line comments [6]. But the effect of comments is significant only in the absence of procedures, i.e., when the code for the subtasks has been merged into the main program. This may be a result of module size, inasmuch as the "module" formed by merging the subtasks into the main program in editions 0 and 1 is larger than any of the modules expressed as procedures in editions 2, 3, 4, 5. While this would explain edition 0 getting the lowest readability score, it does not explain why the same comments failed to make a similar improvement in the program editions which have procedures. The modular organization of these editions may have led readers to view the comments in isolation, as part of a procedure detached from the rest of the program instead of an integral part of the whole program. Thus partitioning in editions 3 and 5 and rearrangement in edition 3 may have made the same comments less effective. This explanation suggests an interaction between comments and procedures, although none was observed statistically in the experiment. In any event it would seem that comments have rescued a small program which is not modular (edition 1) and made it as readable as the modular editions.

While it would be unwise to extrapolate these results to all programs, they do indicate that procedures can have little effect on the readability of programs below a certain size. The effect of procedures on readability should be tested on a variety of programs, and much experimentation with larger programs is needed to determine the size at which procedures become an important factor in readability.

Appendix

Edition 0: inline code, no comments

```
 1.  STAND: PROCEDURE OPTIONS(MAIN);
 2.  DECLARE  1  TEAMRECORD(14),
 3.               2 NAME CHARACTER(16),
 4.               2 SCORE(4) FIXED BINARY(31),
 5.               2 TOTAL FIXED BINARY(31);
 6.  DECLARE (STANDING(14),PLACE(14)) FIXED BINARY(31),
 7.     (LASTTEAM,NEXT,K,L,TOPTEAM,TOPSCORE,POS,IVALUE) FIXED BINARY(31);
 8.  DECLARE INPUTBUF CHARACTER(80),
 9.     CVALUE CHARACTER(1);
10.  DECLARE MOREDATA BIT(1),
11.     TRUE BIT(1) INITIAL('1'B),
12.     FALSE BIT(1) INITIAL('0'B);
13.  ON ENDFILE(SYSIN) MOREDATA = FALSE;
14.
15.  MOREDATA = TRUE;
16.     NEXT = 0;
17.     GET EDIT(INPUTBUF) (A(80));
18.     DO WHILE (MOREDATA);
19.        NEXT = NEXT + 1;
20.        TEAMRECORD(NEXT).NAME = SUBSTR(INPUTBUF,1,16);
21.        POS = 17;
22.        TEAMRECORD(NEXT).TOTAL = 0;
23.        DO K = 1 TO 4;
24.           CVALUE = SUBSTR(INPUTBUF,POS,1);
25.           DO WHILE (CVALUE = ' ');
26.              POS = POS + 1;
27.              CVALUE = SUBSTR(INPUTBUF,POS,1);
28.              END;
29.           IVALUE = 0;
30.           DO WHILE ((CVALUE >= '0') & (CVALUE <= '9'));
31.              IVALUE = 10*IVALUE + (CVALUE - '0');
32.              POS = POS + 1;
33.              CVALUE = SUBSTR(INPUTBUF,POS,1);
34.              END;
35.           TEAMRECORD(NEXT).SCORE(K) = IVALUE;
36.           TEAMRECORD(NEXT).TOTAL = TEAMRECORD(NEXT).TOTAL + IVALUE;
37.           END;
38.        PUT SKIP LIST('');
39.        PUT EDIT(NEXT,'.   ',TEAMRECORD(NEXT).NAME) (F(4),A(4),A(16));
40.        DO L = 1 TO 4;
41.           PUT EDIT(TEAMRECORD(NEXT).SCORE(L)) (F(4));
42.           END;
43.        PUT EDIT('   TOTAL',TEAMRECORD(NEXT).TOTAL) (A(8),F(5));
44.        GET EDIT(INPUTBUF) (A(80));
45.        END;
46.     LASTTEAM = NEXT;
47.     DO L = 1 TO 14;
48.        PLACE(L) = LASTTEAM;
```

```
49.         END;
50.      DO NEXT = 1 TO LASTTEAM;
51.         TOPSCORE = 0;   TOPTEAM = 0;
52.         DO K = 1 TO LASTTEAM;
53.            IF (PLACE(K) = LASTTEAM)  THEN
54.               IF (TEAMRECORD(K).TOTAL > TOPSCORE)  THEN
55.                  DO;
56.                  TOPSCORE = TEAMRECORD(K).TOTAL;
57.                  TOPTEAM  = K;
58.                  END;
59.            END;
60.         STANDING(NEXT) = TOPTEAM;
61.         PLACE(TOPTEAM) = NEXT;
62.         END;
63.      PUT SKIP LIST('');
64.      DO NEXT = 1 TO LASTTEAM;
65.         K = STANDING(NEXT);
66.         PUT SKIP LIST('');
67.         PUT EDIT(NEXT,'.   ',TEAMRECORD(K).NAME) (F(4),A(4),A(16));
68.         DO L = 1 TO 4;
69.            PUT EDIT(TEAMRECORD(K).SCORE(L)) (F(4));
70.            END;
71.         PUT EDIT('   TOTAL',TEAMRECORD(K).TOTAL) (A(8),F(5));
72.         END;
73.   END STAND;
```

1. _____ How many DO WHILE statements are there?

2. _____ If each character requires one byte of memory and each FIXED BINARY(31) value requires 4 bytes, how many bytes of memory are required for the data structure array TEAMRECORD?

3. _____ What is the final value of POS (line 32) when processing the following line of input data?

   KNICKERBOCKERS 16 4 9 11

4. _____ How many times will IVALUE be multiplied by 10 (line 31) when processing the following line of input data?

   EL CAPITAN   23 10 01   0

5. _____ What TOTAL will be computed (line 36) when processing the following line of input data?

   BLUE DEMONS   1   01   +1   1

6. _____ What would happen if the name of a team (line 20) were longer than 16 characters?
   (A) all of the name would be printed, with the proper SCOREs
   (B) the first 16 characters would be printed and the first SCORE would be 0
   (C) the first 16 characters would be printed and all of the SCOREs would be 0
   (D) the first 16 characters would be printed and the first SCORE would be undefined (i.e. garbage)
   (E) the first 16 characters would be printed and all of the SCOREs would all be undefined (i.e. garbage)

7. _____ The sorting algorithm (lines 46-62) can best be described as
   (A) bubble sort (B) selection sort (C) heap sort
   (D) string sort (E) partition exchange sort

8. _____ What is the maximum number of comparisons that can be made in sorting (line 54) when LASTTEAM is equal to 10?

9. _____ What happens in the sorting algorithm if two or more teams have the same TOTAL? (i.e. TEAMRECORD(K).TOTAL is the same for two or more values of K).
   (A) the teams with identical TOTALs stay in their original order
   (B) the teams with identical TOTALs are put in reverse order
   (C) the teams with identical TOTALs are put in a different order, not necessarily (A) or (B)
   (D) PLACE(K) will be the same for all teams having identical TOTALs (line 61)
   (E) the sorting algorithm will get caught in an infinite loop

10. _____ What happens in the sorting algorithm if TOTAL is equal to 0 for one of the teams?
   (A) nothing unusual happens, the team with 0 will be last place
   (B) the sorting algorithm will get caught in an infinite loop
   (C) TOPSCORE will be reset to 0 (line 56)
   (D) STANDING(NEXT) will be set equal to 0 (line 60)
   (E) the team with 0 will never be chosen as TOPTEAM (line 57)

11. _____ Which best explains the distinction between PLACE (line 61) and STANDING (line 60)?
   (A) STANDING(NEXT) = 1 when NEXT represents the team with the highest TOTAL
   (B) the highest total is TEAMRECORD(PLACE(1)).TOTAL
   (C) PLACE and STANDING are equivalent: when the sorting is done both arrays contain the same sequence of values
   (D) after the sorting is complete, STANDING(PLACE(K)) = K for all 1 <= K <= LASTTEAM
   (E) PLACE(TOPTEAM) = 1 when TOPTEAM represets the team with the highest TOTAL

12. _____ In what order are the TEAMRECORDs printed at the end of the program?
(A) TEAMRECORD(1) ... TEAMRECORD(LASTTEAM)
(B) TEAMRECORD(STANDING(1)) ... TEAMRECORD(STANDING(LASTTEAM))
(C) TEAMRECORD(PLACE(1)) ... TEAMRECORD(PLACE(LASTTEAM))
(D) in order of increasing TOTALs
(E) none of these

Edition 5: external procedures, comments

```
/*       THIS PROGRAM READS A FILE CONTAINING ONE RECORD FOR  */
/*  EACH TEAM,  WITH  THE  TEAM'S  NAME  IN  THE  FIRST  16  */
/*  COLUMNS,  FOLLOWED BY THE TEAM'S SCORES IN  1  OR  MORE  */
/*  EVENTS.   EACH SCORE MUST BE REPRESENTED AS AN UNSIGNED  */
/*  INTEGER,  AND THE  SCORES  MUST  BE  SEPARATED  BY  ONE  */
/*  OR MORE BLANKS.      THE  TEAM  RECORDS  ARE  ECHOED  AS  */
/*  THEY ARE READ IN,  AND THEN THEY ARE SORTED AND PRINTED  */
/*  IN ORDER OF DECREASING TOTALS.                           */

STAND: PROCEDURE OPTIONS(MAIN);
DECLARE  1  TEAMRECORD(14),
            2 NAME CHARACTER(16),
            2 SCORE(4) FIXED BINARY(31),
            2 TOTAL FIXED BINARY(31);

/*    TEAMRECORD:                                            */
/*    +--------------+--------------------+-------+         */
/*    |     NAME     |       SCORE        | TOTAL |         */
/*    +--------------+--------------------+-------+         */
/*      CHARACTER(16)    INTEGER ARRAY     INTEGER          */

DECLARE STANDING(14) FIXED BINARY(31);
DECLARE LASTTEAM FIXED BINARY(31);
DECLARE MOREDATA BIT(1),
        TRUE BIT(1) INITIAL('1'B),
        FALSE BIT(1) INITIAL('0'B);
ON ENDFILE(SYSIN) MOREDATA = FALSE;
DECLARE (FETCHD,SORT,LISTOUT) ENTRY;

MOREDATA = TRUE;
CALL FETCHD(TEAMRECORD,LASTTEAM,MOREDATA);
CALL SORT(TEAMRECORD,STANDING,LASTTEAM);
CALL LISTOUT(TEAMRECORD,STANDING,LASTTEAM);
END STAND;

* PROCESS;
 /*  READ THE NAME AND SCORES FOR EACH TEAM:  */
 FETCHD: PROCEDURE(TEAMRECORD,LASTTEAM,MOREDATA);
 DECLARE  1  TEAMRECORD(*),
             2 NAME CHARACTER(16),
             2 SCORE(4) FIXED BINARY(31),
             2 TOTAL FIXED BINARY(31);
 DECLARE LASTTEAM FIXED BINARY(31);
 DECLARE MOREDATA BIT(1);

    DECLARE NEXT FIXED BINARY(31);
    DECLARE INPUTBUF CHARACTER(80);
    DECLARE (DECODE,PRINT1) ENTRY;

    NEXT = 0;
    GET EDIT(INPUTBUF) (A(80));
    DO WHILE (MOREDATA);
       NEXT = NEXT + 1;
       CALL DECODE(INPUTBUF,TEAMRECORD,NEXT);
       CALL PRINT1(TEAMRECORD,NEXT,NEXT);
       GET EDIT(INPUTBUF) (A(80));
       END;    /* WHILE */
    LASTTEAM = NEXT;
    END FETCHD;

* PROCESS;
 /*  DECODE THE NAME, SCORES, TOTAL FOR ONE TEAM:  */
 DECODE: PROCEDURE(INPUTBUF,TEAMRECORD,WHICH);
 DECLARE INPUTBUF CHARACTER(80);
 DECLARE  1  TEAMRECORD(*),
             2 NAME CHARACTER(16),
             2 SCORE(4) FIXED BINARY(31),
             2 TOTAL FIXED BINARY(31);
 DECLARE WHICH FIXED BINARY(31);

    DECLARE (POS,IVALUE,K) FIXED BINARY(31);
    DECLARE CVALUE CHARACTER(1);

 /* COPY THE NAME INTO TEAMRECORD: */
    TEAMRECORD(WHICH).NAME = SUBSTR(INPUTBUF,1,16);
 /* DECODE EACH SCORE FOR THE TEAM AND COMPUTE THE TOTAL: */
    POS = 17;
    TEAMRECORD(WHICH).TOTAL = 0;
    DO K = 1 TO 4;
    /* SKIP OVER BLANKS: */
       CVALUE = SUBSTR(INPUTBUF,POS,1);
       DO WHILE (CVALUE = ' ');
          POS = POS + 1;
          CVALUE = SUBSTR(INPUTBUF,POS,1);
          END;   /* WHILE */
```

```
86.      /* CONVERT THE DECIMAL DIGITS TO AN INTEGER VALUE: */
87.         IVALUE = 0;
88.         DO WHILE ((CVALUE >= '0') & (CVALUE <= '9'));
89.            IVALUE = 10*IVALUE + (CVALUE - '0');
90.            POS = POS + 1;
91.            CVALUE = SUBSTR(INPUTBUF,POS,1);
92.            END;   /* WHILE */
93.         TEAMRECORD(WHICH).SCORE(K) = IVALUE;
94.         TEAMRECORD(WHICH).TOTAL = TEAMRECORD(WHICH).TOTAL + IVALUE;
95.         END;    /* DO K */
96.      END DECODE;
97.
98.  * PROCESS;
99.   /*  PRINT THE SCORES AND TOTAL FOR ONE TEAM:  */
100.  PRINT1: PROCEDURE(TEAMRECORD,WHICHTEAM,PLACE);
101.  DECLARE  1  TEAMRECORD(*),
102.              2 NAME CHARACTER(16),
103.              2 SCORE(4) FIXED BINARY(31),
104.              2 TOTAL FIXED BINARY(31);
105.  DECLARE WHICHTEAM FIXED BINARY(31);
106.  DECLARE PLACE FIXED BINARY(31);
107.
108.     DECLARE L FIXED BINARY(31);
109.
110.     PUT SKIP LIST('');
111.     PUT EDIT(PLACE,'.   ',TEAMRECORD(WHICHTEAM).NAME) (F(4),A(4),A(16));
112.     DO L = 1 TO 4;
113.        PUT EDIT(TEAMRECORD(WHICHTEAM).SCORE(L)) (F(4));
114.        END;   /* DO L */
115.     PUT EDIT('   TOTAL',TEAMRECORD(WHICHTEAM).TOTAL) (A(8),F(5));
116.     END PRINT1;
117.
118.  * PROCESS;
119.   /*  SORT THE TEAM TOTALS:  */
120.  SORT: PROCEDURE(TEAMRECORD,STANDING,LASTTEAM);
121.  DECLARE  1  TEAMRECORD(*),
122.              2 NAME CHARACTER(16),
123.              2 SCORE(4) FIXED BINARY(31),
124.              2 TOTAL FIXED BINARY(31);
125.  DECLARE STANDING(*) FIXED BINARY(31);
126.  DECLARE LASTTEAM FIXED BINARY(31);
127.
128.     DECLARE (NEXT,K,L,TOPTEAM,TOPSCORE) FIXED BINARY(31);
129.     DECLARE PLACE(14) FIXED BINARY(31);
130.
131.  /* INITIALIZE THE "PLACE" OF EACH TEAM TO "LAST": */
132.     DO L = 1 TO 14;
133.        PLACE(L) = LASTTEAM;
134.        END;   /* DO L */
135.     DO NEXT = 1 TO LASTTEAM;
136.        TOPSCORE = 0;   TOPTEAM = 0;
137.     /* FIND OUT WHICH REMAINING TEAM HAS THE HIGHEST TOTAL SCORE: */
138.        DO K = 1 TO LASTTEAM;
139.           IF (PLACE(K) = LASTTEAM)  THEN
140.              IF (TEAMRECORD(K).TOTAL > TOPSCORE)  THEN
141.                 DO;
142.                 TOPSCORE = TEAMRECORD(K).TOTAL;
143.                 TOPTEAM  = K;
144.                 END;     /*  IF  */
145.           END;      /* DO K */
146.     /* HIGHEST TOTAL SCORE => NEXT PLACE IN THE STANDINGS. */
147.        STANDING(NEXT) = TOPTEAM;
148.        PLACE(TOPTEAM) = NEXT;
149.        END;    /* DO NEXT */
150.     END SORT;
151.
152.  * PROCESS;
153.  /*  PRINT THE TEAM STANDINGS, SCORES, AND TOTALS:  */
154.  LISTOUT: PROCEDURE(TEAMRECORD,STANDING,LASTTEAM);
155.  DECLARE  1  TEAMRECORD(*),
156.              2 NAME CHARACTER(16),
157.              2 SCORE(4) FIXED BINARY(31),
158.              2 TOTAL FIXED BINARY(31);
159.  DECLARE STANDING(*) FIXED BINARY(31);
160.  DECLARE LASTTEAM FIXED BINARY(31);
161.
162.     DECLARE K FIXED BINARY(31);
163.     DECLARE PRINT1 ENTRY;
164.
165.     PUT SKIP LIST('');
166.     DO K = 1 TO LASTTEAM;
167.        CALL PRINT1(TEAMRECORD,STANDING(K),K);
168.        END;   /* DO K */
169.     END LISTOUT;
```

[Questions are the same as edition 0, except for the line numbers.]

REFERENCES

[1] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. London: Academic, 1972.

[2] J. L. Elshoff, and M. Marcotty, "Improving computer program readability to aid modification," *Commun. ACM*, vol. 26, no. 8, pp. 512–521, Aug. 1982.

[3] J. D. Gannon and J. J. Horning, "Language design for programming reliability," *IEEE Trans. Software Eng.*, vol. SE-1, no. 2, pp. 179–191, June 1975.

[4] B. W. Kernighan and P. J. Pauger, *The Elements of Programming Style*. New York: McGraw-Hill, 1974.

[5] H. F. Ledgard, *Programming Proverbs*. Rochelle Park, NJ: Hayden, 1975.

[6] A. F. Norcio, "Indentation documentation and programmer compre-

hension," in *Proc. Human Factors in Computer Systems*, ACM, Washington, DC, 1981, pp. 118–120.
[7] B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*. Boston, MA: Little, Brown, 1980.
[8] ——, "Control flow and data structure documentation: Two experiments," *Commun. ACM*, vol. 25, no. 1, pp. 55–63, Jan. 1982.
[9] T. C. Tenny, "Procedures and comments vs. the Banker's Algorithm," *SIGCSE Bull.*, vol. 17, no. 3, pp. 44–53, Sept. 1985.
[10] L. M. Weissman, "A methodology for studying the psychological complexity of computer programs," Ph.D. dissertation, Univ. Toronto, Tech. Rep. CSRG-37, Aug. 1974.
[11] ——, "Psychological complexity of computer programs: An experimental methodology," *SIGPLAN Notices*, vol. 15, no. 6, pp. 25–36, June 1974.
[12] B. J. Winer, *Statistical Principles in Experimental Design*, 2nd. ed. New York: McGraw-Hill, 1971.
[13] E. Yourdon and L. L. Constantine, *Structured Design*. Englewood Cliffs, NJ: Prentice-Hall, 1979.

Ted Tenny received the Ph.D. degree in mathematics and computer science from Clarkson College of Technology (now Clarkson University), Potsdam, NY, in 1982.

Since then he has taught at the University of Oklahoma and is now an Assistant Professor of Computer Science at Texas Christian University, Fort Worth. His current research interests include program comprehension, team programming, and software methodology.