TECHNICAL NOTE

Human Aspects of Computing

Henry Ledgard Editor

PROGRAM INDENTATION AND COMPREHENSIBILITY

Richard J. Miara, Joyce A. Musselman, Juan A. Navarro, and Ben Shneiderman
University of Maryland

ABSTRACT: ***The consensus in the programming community is that indentation aids program comprehension, although many studies do not back this up. We tested program comprehension on a Pascal program. Two styles of indentation were used—blocked and nonblocked—in addition to four possible levels of indentation (0, 2, 4, 6 spaces). Both experienced and novice subjects were used. Although the blocking style made no difference, the level of identation had a significant effect on program comprehension. (2–4 spaces had the highest mean score for program comprehension.) We recommend that a moderate level of indentation be used to increase program comprehension and user satisfaction.***

Ben Shneiderman is currently associate professor and head of the Human–Computer Interaction Laboratory. Richard Miara, Joyce Musselman, and Juan Navarro were students of Shneiderman at the time of writing.

Authors' Present Addresses: Richard J. Miara, 8807 Enfield Ct., #8, Laurel, MD 20708; Joyce A. Musselman, Software Architecture and Engineering, 1401 Wilson Boulevard, Suite 1220, Arlington, VA 22209; Juan Navarro, Engineering Research Associates, 1517 West Branch Drive, McLean, VA 22012; Ben Shneiderman, Department of Computer Science, University of Maryland, College Park, MD 20742

0001-0782/83/1100-0861 75¢

1. INTRODUCTION

Recent studies have concentrated on program indentation along with other variables such as commenting, blank-line insertion, and control flow as a factor in program comprehension. The general consensus is that programs formatted with indentation are "better," that is, are easier to follow and modify, although there is experimental evidence showing that indentation does not always aid program comprehension. Our experiment focused solely on the effect of indentation on program comprehension and user satisfaction. Two styles of indentation were tested—blocked and nonblocked, and four possible levels of indentation (0, 2, 4, 6 spaces).

Our intention was to gather experimental evidence to support the notion that intermediate indentation, which we define to be 2–4 spaces, would provide an optimal level of program comprehension and user satisfaction. We believe that minimal and excessive indentation inhibits program comprehension and leads to greater programmer dissatisfaction.

Many studies to test the effects of indentation on program comprehension and user satisfaction have been done. Weissman [21, 22] conducted several studies in this area using PL/I and 2-space indentation. Weissman tested the interaction between indentation and commenting. He found that the main effect of indentation alone was not significant in any of his measures, but that a significant interaction occurred with commenting. When comments were absent in the programs, indentation helped only slightly; when comments were present, indentation hurt drastically. Weissman was surprised by these negative results and tried to explain them by the fact that the programs had to be split across page boundaries and that the programs were not split at a reasonable point. Also, the programs contained GOTO statements that did not lend themselves to indentation [21]. While Weissman thought this might partially explain the negative effects of indentation, he did not think it explained the interaction with commenting.

In another experiment, Weissman tested the interaction between indentation and control flow. This experiment supports the hypothesis that indentation aids in program comprehension and user satisfaction, since in most cases it improved performance (sometimes significantly) and did not significantly hurt performance.

In an experiment conducted by Shneiderman and McKay [20], subjects were given two programs—one indented and one not. The subjects were asked to locate and repair a bug in each program. The data from this experiment suggest that as program complexity increases, program comprehension is aided by indentation, although the authors indicate significant differences were not found.

Clifton [1] states that one of the most important attributes of a program is readability. A program that is easy to read and understand is easier to test, maintain, and modify. Clifton also states that even though structured programming should aid

readability, a structured program can be difficult to understand if the control structures are heavily nested or many lines separate parts of the control structures.

Indentation is the most popular technique for making control structures easier to read [1]. However, the usefulness of indentation diminishes when parts of control structures are widely separated or heavily nested. Clifton states that this will make it difficult for a reader to skip around a group of statements or find the path back from the end to the beginning of a loop.

Leinbaugh [12] argues that indentation alone is sufficient to determine the block structure of a program. He states that the only indentation rule that is needed is that "... all statements directly belonging to a control statement are right indented an equal amount from the beginning of that control statement." Leinbaugh claims that if this indentation rule is followed, the use of compound statements or closing keywords is not necessary to express a program's block structure.

Some new techniques have been used with indentation to aid in program readability. One of these techniques, proposed by Clifton, is connector lines. These lines connect parts of control structures on entire programs. The connector lines and indentation could be automatically generated on listings to show the logical structure of programs. The following is an example of the use of connector lines.

```
1—IF JUST THEN DO;
1     PRINTLINE = "";
1     ⎩
1    —DO I =1 TO SEP #;
1 2     SEPSPACES = SEPSPACES "";
1 ** *** END;
1       ⎪
⎩       ⎪
1—      END;
```

Clifton suggests that the connector-lines technique could be useful for teaching programming since the lines clearly show students the relationships among different parts of control structures.

Another technique, Contour [4], graphically illustrates a program's structure by bounding the scope of loops and conditionals with solid lines. Gimpel states that Contour has the advantage that it makes fewer demands on the reader's linguistic expertise and may be used for presenting algorithms in a language-independent manner.

Conrow and Smith [2] suggest the use of NEATER2 to aid program readability. As the name implies, this techique neatens up the program source listing. This is useful in detecting logic errors because it produces unexpected indentation patterns and reveals some syntax errors by peculiarities in indentation patterns or specifically flagging them.

Krall and Harris' [11] experiment yielded some significant results. This experiment used the two metrics of a comprehension quiz and reconstruction to measure the effects that indentation, no indentation, blank lines, no blank lines would have on novice undergraduate programmers. Highly significant results were found below the .01 level for the comprehension quiz. The results clearly show that indentation alone, over any other combination of the 2 × 2 design, gave the highest level of comprehension of a Cobol program containing nested **IF** statements. This can be explained two ways. First, with indentation, the corresponding **ELSE** block of an **IF** Statement could easily be found by vertically scanning directly beneath the **IF**. Second, indentation aids in highlighting the end of a Cobol statement, which is identified by a period. The location of a period in a Cobol program greatly affects the program's execution. Lack of indentation creates more confusion for the programmer trying to locate the period. However, if indentation is used, the period and end-of-statement are easily found. Furthermore, these results significantly demonstrate that indentation used with blank lines hinders comprehension of the above-mentioned program. However, using a second metric, memorization/reconstruction, no statistically significant results were produced within their 2 × 2 design.

Krall [10] extended and expanded the above experiment. The designer used experienced professional Cobol programmers to see if previous results could be applied to experts. Once again, for the quiz sheet metric, a 1 percent statistical-significance level resulted between the indented and nonindented programs with greater comprehension resulting from indentation. However, this time, no significant interaction was produced between any form of indentation and the use or nonuse of blank lines. Overall, both experiments support the idea that indentation aids program comprehension and not reconstruction.

An experiment by Norcio and Kerst [15] used program reconstruction as a metric to test the effects on indentation. The experimental materials consisted of five unstructured Fortran programs containing various combinations of documentation and indentation. The results implied that indentation and documentation do not enhance the reconstruction tasks. The authors offered the explanation that "indentation might interfere with the visual image of the program." Regardless of the reason, indentation did not aid in program recall.

Norcio [14] conducted two experiments using the Cloze technique that tested the effects of documentation and indentation on program comprehension. This technique substituted blank lines for various source statements and required the subject to supply the missing statements. Norcio used the same experimental procedures and indented versus nonindented Fortran programs with various levels of documentation in both experiments. The only difference was the location of the missing statements within each logic segment of code. The first experiment had source statements removed from the beginning of each logic segment. In the second experiment, the missing statements were within the logic segments. A multivariate analysis of variance (MANOVA) for documentation and indentation showed a significant interaction effect ($p < .04$) in both experiments. The use of indentation and one line of interspersed documentation resulted in the highest degree of program comprehension. Norcio also noted that the use of indentation significantly aided program comprehension in both experiments.

Love [13] conducted a within-subject experiment to test the effects produced by the independent variables of indentation of source code and complexity of control flow on program comprehension. Less complex control-flow structures differ from complex structures in that they only allow the control structures of sequence, selection, and repetition. As stated by Love, "These structures have only one entrance and only one exit and do not allow unconditional GOTO statements." The dependent variable was the percentage of lines correctly recalled from the Fortran programs. The results of this experiment show that programs with less complex control flow are easier to understand than those with complex forms. However, the presence or absence of indentation of source code produced no significant differences.

It is generally agreed that programs that are indented, spaced, and commented are easier to read and edit; the problem seems to be on what approach to use in accomplishing this formatting goal. A novel technique for improving the readability of Pascal programs is the implementation of the prettyprinting programs that provide a listing of the source code using semicolons to connect the **begin–end** blocks [17]. The idea is to connect a **begin** with its corresponding **end** by drawing a line of semicolons from one to the other. Hueras and Ledgard [8] wrote a prettyprinting program that rearranges the spacing and indentation of certain constructs to make the logical structure of the program more visually apparent. Unlike other prettyprinting programs, the Hueras and Ledgard program does no syntax checking and will even work on program fragments. The authors suggest that the prettyprinter should be used as an aid in editing and should not impose rigid syntax checks when the program is in the development stages. Other research has concentrated on defining a set of rules for programmers to follow when writing Pascal programs. Crider [3] suggests a style that emphasizes the structured aspect of the language by having an introductory phrase such as a **while** statement followed by all the dependent clauses indented directlv below. As an example, he shows the following:

```
while r >=dd do begin
    r :=r -dd;
    dd :=dd + dd end;
```

The **while** statement is the introductory phrase and the remaining assignment statements are the dependent clauses. The **while** is the keyword in the phrase and the **do** and **begin** are control information that indicate how the dependent clauses are carried out. Crider claims that this format clearly emphasizes the structuring of statements.

Noting that programs have to be written in a form that is readable and easily modifiable, Peterson [16], suggests that **begin–end** blocks be clearly discernable by indenting the statements that are enclosed within these delimiters. The effect of this indentation is to provide a programmer with the ability to quickly identify **begin–end** blocks and all the statements enclosed within these delimiters. Two examples are given to show the advantage of separating the statements within the **begin–end** blocks:

```
BEGIN
      J := J + 1;
      OUTCARD[J] :=C
END;

BEGIN
          INFO := I;
          LEFT := L;
          RIGHT := R

END;
```

The first example shows the minimum indentation of five spaces that account for the number of characters in the **begin.** The second example shows Peterson's preferred method of shifting the following statements sufficiently to the right so that they stand out and are easily identified as contained within the encompassing **begin–end** block.

Richardson et al. [18], say that "... the 'classical' structured programming 'rules' include a set of language-dependent conventions to dictate how to indent program statements which are designed to make constructions more visible to the reader." They state that even though manual indentation by the programmer may initially make the program harder to write, the indentation helps to simplify the reading and understanding of the program.

In our experiment, we place the major emphasis on levels of indentation and which level, if any, yields significantly better results. The experiment measured whether novice or experienced undergraduates would be most affected by the varying degrees of indentation.

2. EXPERIMENTAL PROCEDURES

Hypothesis: Expert and novice Pascal programmers will show no increase in program comprehension when excessive indentation is used instead of no indentation. Furthermore, we suggest that there exists a moderate level of indentation where both experts and novices will show an increase in performance. Two methods of block indentation will also be tested.

Independent Variables:

1. Levels of Indentation
   a. No Indentation
   b. Indentation—Using Two Spaces
   c. Indentation—Using Four Spaces
   d. Indentation—Using Six Spaces
2. Level of Programmer Experience
   a. Novice: Less than three years of programming experience in school and/or less than two years professionally.
   b. Expert: Three or more years of programming experience in school and/or two years or more professionally.
3. Method of Block Indentation
   a. Nonblocked: Indentation after the beginning of a block.
   b. Blocked: Body of block flush with beginning of a block.

Dependent Variables:

1. Comprehension Quiz Scores
2. Subjective Rating of the Program Difficulty

2.1. Subjects

Our novice subjects had less than three years of programming experience in school and/or less than two years of professional programming experience. They were selected from an intermediate-level programming class in Pascal at the University of Maryland. The experiment was administered in the tenth week of the semester. By this time, the students had already written several Pascal programs beyond the complexity of the program they were given for the experiment. In general, these subjects tended to be freshmen or sophomores.

Our expert subjects had three or more years of programming in school and/or more than two years of professional programming experience. These subjects were selected by administering our experiment to a senior-level computer-science class. Generally, these students were graduating computer-science majors. The majority of these students qualified as experts.

2.2. Materials

A Pascal program (Appendix A) from Grogono's book, *Programming in Pascal* [5] was modified to produce seven different versions. Our program was a text-concordance program

that calculated the number of occurrences of a word for a given string of input. The output was each unique word with its frequency of occurrence. Individual letters were accepted as words and numbers and blanks functioned as delimiters. Each version had 102 statements and contained no blank lines or comments. Each version of the two-page program was divided at the same location. A wide range of syntactical structures were used , (that is, sets, records, packed-arrays, **while**-loops, and **if–then–elses**. All of the syntactical structures in the program had already been taught to the students. The semantics of this concordance program could be considered challenging for both novices and experts. Finally, the programs distributed to the students were produced on computer paper from a line printer and were easy to read.

The difference among versions of the concordance program was indentation. In order to test a wide range of indentation levels, four degrees of indentation were separately tested. The four levels were 0, 2, 4, and 6 spaces. For the nonindented version, each statement began in column one. For the indented versions, indentation was used to highlight each semantically related block of codes.

Another factor tested for in each level of indentation was blocked and nonblocked structuring. Blocked structures are defined as **begin–end** blocks of code with inner statements starting in the same column as the **begin** and **end**. Nonblocked structures are defined as **begin–end** blocks of code with inner statements starting at least one level (2, 4, 6 spaces) of indentation to the right of the **begin** and **end**. For each

```
PROGRAM TEST;
  CONST
    TABLESIZE = 1000;
    MAXWORDLEN = 20;
  TYPE
    CHARINDEX = 1 .. MAXWORDLEN;
    COUNTTYPE = 1 .. MAXINT;
    TABLEINDEX = 1 .. TABLESIZE;
    WORDTYPE = PACKED ARRAY [CHARINDEX] OF CHAR;
    ENTRITYPE =
      RECORD
        WORD : WORDTYPE;
        COUNT : COUNTTYPE
      END;
    TABLETYPE = ARRAY [TABLEINDEX] OF ENTRITYPE;
  VAR
    TABLE : TABLETYPE;
    ENTRI, NEXTENTRI : TABLEINDEX;
    TABLEFULL : BOOLEAN;
    LETTERS : SET OF CHAR;
PROCEDURE READWORD (VAR PACKEDWORD : WORDTYPE);
  CONST
    BLANK = ' ';
  VAR
    BUFFER : ARRAY [CHARINDEX] OF CHAR;
    CHARCOUNT : 0 .. MAXWORDLEN;
    CH : CHAR;
  BEGIN
    IF NOT EOF
      THEN
        REPEAT
          READ(CH)
        UNTIL EOF OR (CH IN LETTERS);
    IF NOT EOF
      THEN
        BEGIN
          CHARCOUNT := 0;
          WHILE CH IN LETTERS DO
            BEGIN
              IF CHARCOUNT < MAXWORDLEN
                THEN
                  BEGIN
                    CHARCOUNT := CHARCOUNT + 1;
                    BUFFER[CHARCOUNT] := CH
                  END;
              IF EOF
                THEN
                  CH := BLANK
              ELSE READ(CH)
            END;
          FOR CHARCOUNT := CHARCOUNT + 1 TO MAXWORDLEN DO
            BUFFER[CHARCOUNT] := BLANK;
          PACK(BUFFER,1,PACKEDWORD)
        END
  END;
PROCEDURE PRINTWORD (PACKEDWORD : WORDTYPE);
  CONST
    BLANK = ' ';
  VAR
    BUFFER : ARRAY [CHARINDEX] OF CHAR;
    CHARPOS : 1 .. MAXWORDLEN;
  BEGIN
    UNPACK(PACKEDWORD,BUFFER,1);
    FOR CHARPOS := 1 TO MAXWORDLEN DO
      WRITE(BUFFER[CHARPOS])
  END;
BEGIN
  LETTERS := ['A' .. 'Z'];
  TABLEFULL := FALSE;
  NEXTENTRI := 1;
  WHILE NOT (EOF OR TABLEFULL) DO
    BEGIN
      READWORD (TABLE[NEXTENTRI] . WORD);
      IF NOT EOF
        THEN
          BEGIN
            ENTRI := 1;
            WHILE TABLE[ENTRI].WORD <> TABLE[NEXTENTRI].WORD
              ENTRI := ENTRI + 1;
            IF ENTRI < NEXTENTRI
              THEN
                TABLE[ENTRI].COUNT := TABLE[ENTRI].COUNT + 1
            ELSE IF NEXTENTRI < TABLESIZE
              THEN
                BEGIN
                  NEXTENTRI := NEXTENTRI + 1;
                  TABLE[ENTRI].COUNT := 1
                END
            ELSE TABLEFULL := TRUE
          END
    END;
  IF TABLEFULL
    THEN
      WRITELN('THE TABLE IS NOT LARGE ENOUGH')
  ELSE
    FOR ENTRI := 1 TO NEXTENTRI - 1 DO
      WITH TABLE[ENTRI] DO
        BEGIN
          PRINTWORD(WORD);
          WRITELN(COUNT)
        END
END.
```

Appendix A. Program Listing with 2-Space Indentation in Nonblocked Form.
(From P. Grogono's *Programming in Pascal*.)

program containing an unique level of indentation, there was a blocked and nonblocked version. This resulted in seven unique programs being produced. There were not eight different programs because the nonindented blocked program is exactly the same as the nonindented, nonblocked program. However, in order not to complicate matters, all eight cells of the design were used in the experiment.

The dependent variables consisted of a comprehension quiz and a subjective rating of how difficult the program was to comprehend. The quiz sheet consisted of 13 questions. The first nine questions were multiple choice (e.g., "The maximum number of input records is?" or "The output is? followed by choices") or true/false (e.g., "All variables in this program are global.") Question 10 was a short essay question that asked the subjects to describe what the program did. Partial credit was assigned in the following manner: one-third counted occurrence of words, one-third prints each unique word, and one-third credit for answering that the program prints the number of occurrence next to each word on the output. Question 11 was a subjective rating from 1 to 7 of the difficulty encountered in comprehending the program, with 1 being very easy, 4 moderate, and 7 very hard. Questions 12 and 13 ask how many years of programming experience each subject had in school and professionally. All questions were printed on 8½ × 11 sheets that were clear and easy to read. The 20 min time limit for the test was printed in the instructions at the top of the first page.

2.3. Administration

The administration of the experiment to the novice group went well. We began by introducing ourselves and explaining that we were conducting an experiment. No details of the experiment were explained. The novices were told that they would be given a consent form, program, and a quiz. We asked them to sign the consent form but not to attach it to their quiz to insure their anonymity. A few students asked if this would affect their course grade. We explained that it would not have any bearing on their grade. We told them to answer the questions to the best of their ability and that they could write any comments on the programs or quiz sheets. Finally, we told them they would have a maximum time limit of 20 minutes. Then, we asked them to begin.

At the beginning of the experiment, a few students said the program would not execute because the INPUT and OUTPUT parameters were missing. To avoid any possible confusion, we explained to the entire class that these are default values and that the programs are correct and will execute. A comfortable room temperature was maintained throughout the experiment. There was proper lighting, and outside noise was minimized by keeping the classroom doors closed. None of the students for the next class was allowed to enter until the experiment was over. Five minutes before the end of the test, the subjects were advised of the remaining time.

The administration of our experiment to the expert group also proceeded well. We used the same introductory format for the expert group. However, before the experts began, we explained that the INPUT and OUTPUT parameters were not needed and that the programs were correct and would execute. Once again, temperature, lighting, and lack of noise produced a proper environment for testing. The subjects were advised when five minutes were left. Once again, no students for the next class were allowed to enter and disrupt the class until the experiment was complete.

For both subject groups, the experiment was conducted during the last portion of the class. The desk space for each student was extremely limited. There was not sufficient room for a student to spread out the program and quiz sheets. However, both groups had exactly the same type of desks. Consequently, this bias was controlled for both groups. In both groups, we also noticed that approximately one-third of the subjects completed the experiment before the end of the designated time period.

3. RESULTS

A pilot study was conducted using approximately one subject per cell. Although no statistical analysis was done on the results, several changes were made on the quiz after considering the feedback from the subjects. The changes consisted of rewording several questions for clarity, discarding questions that were trivial or irrelevant, and adding questions that were more challenging.

For the experiment, several of the subjects tested in the expert group were categorized as novices by predefined criteria and, as a result, there were approximately 30 percent fewer expert subjects than novice. A total of 86 subjects were used in the final analysis of the data. Seven quizzes were excluded from the analysis for the following reasons: two subjects had seen the program before and were familiar with its function, three subjects did not know Pascal, and two subjects were observed not participating in the task. See Table I for the breakdown of subjects per cell.

The novices were more critical about what they termed "poor" programming practices than the experts. Many of the novices wrote comments on their program listings and quizzes complaining about the lack of indentation, commenting, and spacing in the programs. The experts made very few comments about the program's structure.

A significant number of the subjects who received the nonindented version of the program spent much of the 20-minute time limit blocking the program off into control blocks (50 percent of the novices and 62 percent of the experts). An interesting observation was made with the novice subjects: In the group that received the nonblocked, 6-space indentation version, none of the returned program listings were marked off into control blocks. Of those who received the blocked, 6-space indentation version, 50 percent of the returned listings were marked off into control blocks.

Overall, the experts did better on the quizzes than the novices. The mean score was 6.7 for experts and 4.9 for novices, out of a possible 10 points. Both the novices and experts had the highest mean scores for the program with the 2-space indentation; 7.5 for experts and 6.0 for novices. Both

TABLE I. Subjects Per Cell

	Novice (54) Indention Level (Spaces)			
	0	2	4	6
Nonblocked	9	7	5	5
Blocked	8	7	7	6
	Expert (32) Indention Level (Spaces)			
	0	2	4	6
Nonblocked	3	5	4	6
Blocked	3	3	2	6

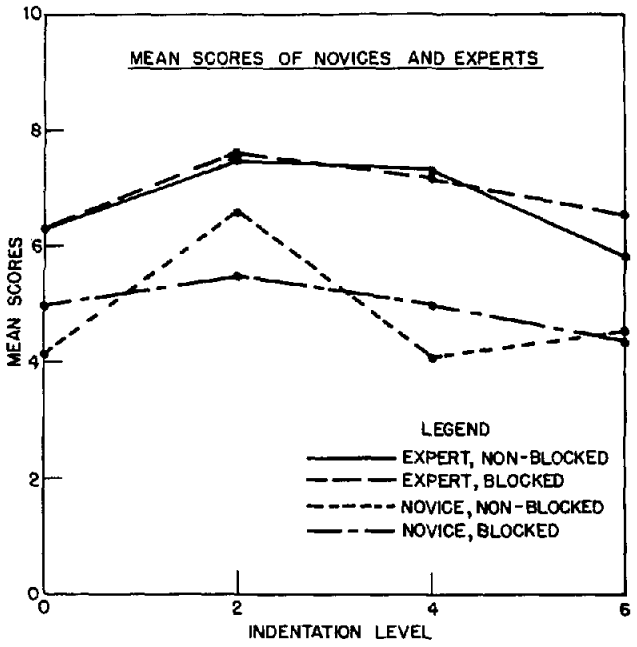

FIGURE 1. Mean Scores of Novices and Experts.

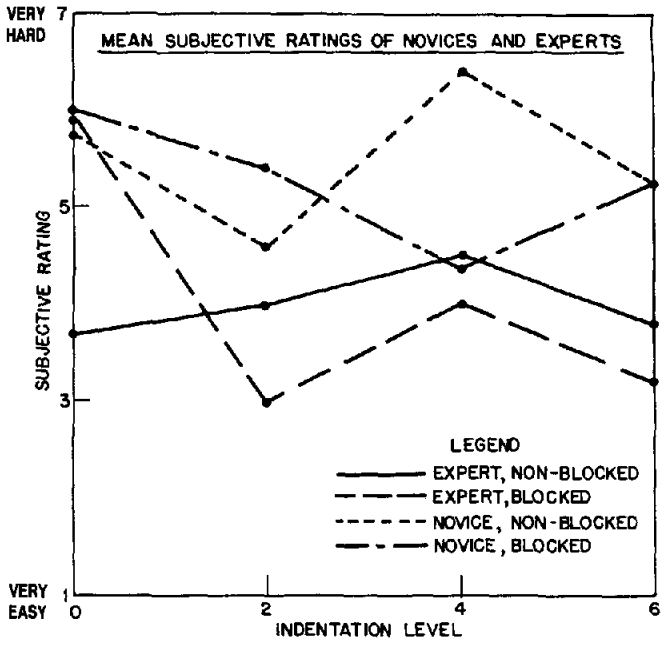

FIGURE 2. Mean Program Ratings of Novices and Experts.

also had significantly lower mean scores on the nonindented program, with 6.3 for experts and 4.5 for novices. The mean scores for nonblocked and blocked styles of indentation were very close. For novices, the nonblocked mean score was 4.8 and the blocked mean score was 5.0. For experts, the nonblocked and blocked mean scores were 6.7 and 6.7, respectively. Figure 1 represents the mean scores of novices and experts.

Generally, novices rated all versions of the program to be more difficult to comprehend than the experts (5.4 and 4.0, respectively, with 7 being the most difficult). The nonindented program was rated more difficult to comprehend than indented versions of the program by both novices and experts, except for the experts who received the nonblocked, nonindented program. Between the nonblocked and blocked styles of indentation, there was no significant difference in the rating of program comprehensibility. Figure 2 represents the mean program ratings of novices and experts.

The combined results ran about the same as for the groups separately. Those subjects who received the nonindented program had a lower mean score than other subjects. Those subjects who received the program with the 2-space indentation had a higher mean score than the other subjects. The mean scores for the nonblocked versus blocked styles of indentation were very close; 5.6 for the nonblocked and 5.6 for the blocked. The program rating of the combined subjects ran about the same as for the separate groups.

The analysis of variance (ANOVA) of the quiz scores showed that experience level had an effect on program comprehension at the $p < 0.001$ significance level. The ANOVA also showed that the level of indentation had a significant effect on the mean scores at the $p = 0.013$ level. No significant effect was found with the nonblocked versus blocked styles of indentation and in any of the 2-way or 3-way interactions. Approximately 36 percent of the variance of the quiz scores were explained.

The ANOVA of the program difficulty ratings showed several significant effects. Again, both experience levels and indentation levels show an effect at significant levels of $p < 0.001$ and $p = 0.072$, respectively. Another result was a significant interaction between the indentation level and nonblocked and blocked styles of indentation at the $p = 0.093$ level. There was a modest 3-way interaction between experience, indentation levels, and blocking styles at the $p = 0.099$ level. Approximately 40 percent of the variance in the subjective ratings was explained.

4. DISCUSSION

The results indicate that the level of indentation has a statistically significant effect on program comprehension and that deeper indentation could become more of a hindrance than an aid. The level of indentation that seems to produce optimal results in comprehension is between 2 and 4 spaces; as the number of spaces increase, the comprehension level decreases. The decreasing level of comprehension might be attributed to the fact that as the nesting level in a deeply indented program (i.e., 6 spaces or more) increases, the program is shifted so far to the right of the page that scanning becomes difficult. In the nonblocked, 6-space version, it became necessary to continue statements on the next line when the nesting level brought the text to the 80-column limit of the compiler. With 2–4-space indentation levels, however, the program is more compact and the control blocks do not become obscured by increased nesting levels.

Novices showed great displeasure with the nonindented version of the program and had significantly lower scores on that version. Their best overall performance was with the version that they rated the least difficult (2 spaces). Novices seemed more concerned with the program style than whether it would run. We feel this bias is a result of the requirements placed upon the novices in their programming class; they were required to write programs that were indented, spaced, and commented. Also, most Pascal textbooks, including the one being used by the subjects' professor, show programs that

are indented. Novices consider indentation to be a "good" programming practice and the lack of indentation produced negative feelings towards the program comprehension task as noted by the comments on the quiz. These negative feelings towards the nonindented versions explain the quiz results.

Experts, on the other hand, did not express any negative opinion towards the nonindented version of the program. We feel that experienced programmers will generally approach a comprehension task without much consideration of the style in which the program was written. Very few comments were received by the experts when the experiment was implemented.

Fifty percent of the novices and 62 percent of the experts with nonindented programs marked their listings to connect the control blocks. This result indicates that some form of indentation is needed to clearly distinguish control segments in the program. However, when the program is deeply indented, control blocks might not be clearly identifiable; some subjects marked their 6-spaced version to reflect the control-block structure.

The blocked and nonblocked styles of the program yielded no significant differences between the experts or the novices. We are not sure why this result occurred because we expected a significant difference in comprehension with the type of blocking used for control structures. It may be possible that comprehension scores for a longer and more complex program would show a greater difference with the type of blocking used for the control structures.

Overall, experts did better on the comprehension task and rated the program less difficult than the novices. These results were reassuring because we expected the experts to do better and to rate this type of task less difficult than novices.

Finally, the combined results of the expert and novice subjects showed the highest mean scores in the 2-space indent programs. It is interesting to note, however, that the 6-space indent programs were rated as least difficult to use. We feel that this result occurs because programmers find a deeply indented program visually pleasing since it seems to spread out neatly the constructs of the language. However, when a comprehension task is assigned, this exaggerated spacing causes problems when control blocks become harder to locate with deep indentation, thus resulting in lower scores. The fact that some subjects marked their 6-space version with block-connecting lines provides evidence that control blocks do become harder to distinguish with deep indentation.

5. CONCLUSION

This experiment tested the effects of indentation on program comprehension. The levels of indentation we tested (0–6 spaces) gave strong results favoring 2 or 4 spaces. We believe future experiments should employ the metric of program comprehension, and recommend that nine indentation levels (0 to 8 spaces) be studied. It would be interesting to see how significantly comprehension would be affected beyond the 6-space indentation level.

In summary, we conclude that some indentation does aid program comprehension. From our results, we suggest that the optimal level of indentation is 2–4 spaces. No indentation produced significantly lower mean scores and the subjects found working with this program difficult. We conclude that in a large program, no indentation would be a real hindrance and very difficult to use. The same is true for overly indented programs. With large programs, overindentation may make it difficult for the user to easily scan the program for a particular structure block because the program statements are spread across the page instead of being in a compact format. Although no significant differences were found between the blocked and nonblocked program styles, we suggest that other blocking styles may aid program comprehension and increase user satisfaction. In any case, the blocking style should be consistent throughout the program so that users can easily find the statement or statement segment they are trying to locate. In closing, we agree with Kernighan and Plauger [9] who stated that, "Indentation must be done carefully, however, lest you confuse rather than enlighten."

REFERENCES

1. Clifton, M. H. A technique for making structured programs more readable. *ACM SIGPLAN Notices 13*, 4 (April 78), 58–63.
2. Conrow, K. and Smith, R. G. NEATER2: A PL/I source statement reformatter. *Comm. ACM 13*, 11 (Nov. 70), 669–675.
3. Crider, J. E. Structured formatting of Pascal programs. *ACM SIGPLAN Notices 13*, 11 (Nov. 78), 15–22.
4. Gimpel, J. F. Contour, a method of preparing structured flow charts. *ACM SIGPLAN Notices 15*, 10 (Oct. 1980), 35–41.
5. Grogono, P. *Programming in PASCAL*. Addision-Wesley Publishing Company, Inc., Reading, MA, 1978, 186–188.
6. Grouse, P. Flowblocks—A technique for structured programming. *ACM SIGPLAN Notices 13*, 2 (Feb. 78), 46–56.
7. Gustafson, G. G. Some practical experiences formatting Pascal programs. *ACM SIGPLAN Notices 14*, 9 (Sept. 79), 42–49.
8. Hueras, J. and Ledgard, H. An automatic formatting program for PASCAL. *ACM SIGPLAN Notices 12*, 7 (July 77), 82–84.
9. Kernighan, B. W. and Plaugher, P. J. *The Elements of Programming Style*. McGraw-Hill Book Company, 1978.
10. Krall, A. An investigation of program style on the readability/understandability of a complex COBOL conditional structure. Unpublished research project rpt., Univ. of Maryland, Dec. 11, 1980.
11. Krall, A. and Harris, W. An investigation of program style on the readability/understandability of a simple COBOL program: The effects of indentation and vertical spacing. Unpublished research project rpt., Univ. of Maryland, Dec. 11, 1980.
12. Leinbaugh, D. W. Indenting for the Computer. *ACM SIGPLAN Notices 15*, 5 (May 1980), 41–48.
13. Love, T. An experimental investigation of the effect of program structure on program understanding. *Proc. ACM Conference on Language Design for Reliable Software*. March 1977, 105–113.
14. Novcio, A. F. Indentation, documentation and programmer comprehension, Proceedings of Human Factors in Computer Systems. ACM Washington, DC, Chapter, 1981. 118–120.
15. Norcio, A. F. and Kerst, S. M. *Human Memory Organization for Computer Programs*. Catholic University of America, unpublished manuscript (1978).
16. Peterson, J. L. On formatting of Pascal programs. *ACM SIGPLAN Notices 12*, 12 (Dec. 77), 83–86.
17. Ramsdell, J. Prettyprinting structured programs with connector lines. *ACM SIGPLAN Notices 14*, 9 (Sept. 79), 74–75.
18. Richardson, G. L., Butler, C. W., and Tomlinson, J. D. *A Primer on Structured Program Design*. Petrocelli Books, Inc., New York, 1980.
19. Shneiderman, B. *Software Psychology: Human Factors in Computer and Information Systems*. Little, Brown and Co., Boston, 1980.
20. Shneiderman, B. and McKay, D. Experimental investigations of computer program debugging and modification. *Proc. 6th International Congress of the International Ergonomics Association*. July 1976, College Park, MD.
21. Weissman, L. M. A methodology for studying the psychological complexity of computer programs. Technical Report CSRG-37, University of Toronto, Ph.D. Dissertation, August, 1974.
22. Weissman, L. M. Psychological complexity of computer programs: An experimental methodology. *ACM SIGPLAN Notices 15*, 6 (June 1974), 25–36.

CR Categories and Subject Descriptors: D.2.3 [**Software**]: Software Engineering—*coding*; D.M. [**SOFTWARE**]: Miscellaneous—*software psychology*

General Terms: human factors, experimentation, languages

Additional Key Words and Phrases: indentation, program format, program readability

Received 11/82; revised 12/82; accepted 4/83