Effects of Programming Experience in Debugging Semantic Errors

Paul W. Oman, Curtis, R. Cook, and Murthi Nanja
Computer Science Department, Oregon State University, Oregon

This paper presents the results of a controlled experiment comparing debugging abilities of novice, intermediate, and skilled student programmers. Debugging-performance differences were studied using two single-page Pascal programs: a binary search program and a median calculation program. Two types of semantic errors, array bounds and undefined variable, and two types of error messages, with and without line number, were varied within the two programs. Subjects were asked to find and correct a single error in each program. Results demonstrate skill-level differences, show the importance of error and message interaction, and support previous research claiming that programmers can almost always correct an error once it is located.

1. INTRODUCTION

Debugging, the location and correction of the errors in a computer program, is one of the most common computer-programming tasks. It is not unusual for debugging to consume nearly half of the software-development resources.

It is generally recognized that debugging is a skill acquired through experience. There appear to be two general debugging strategies. In the comprehension approach, the programmer attempts to find and correct bugs by first understanding what the program actually does as compared to what it is supposed to do. In the isolation approach, the programmer attempts to identify candidate bug locations by searching for clues in the output, recalling similar bugs, testing internal program states, or using knowledge of the application domain. Some programmers may use a combination of strategies; for example, resorting to the comprehension approach when the isolation approach fails to locate the bug. Hence, debugging requires the effective application of problem-solving skills, programming-language knowledge, problem-domain knowledge, and debugging techniques learned from previous experience.

Level of debugging skill is one of the major differences between novice and expert programmers. Experts make fewer errors and locate and correct bugs faster than novices [8]. Novices frequently add additional bugs during debugging, whereas experts seldom if ever introduce new bugs [4]. Several studies have also shown considerable differences in debugging speed even among experts [2, 3]. Although some debugging techniques and tricks are included in programming courses, there are no courses devoted entirely to program-debugging strategies. Thus, debugging skill is primarily learned through general programming experience.

There have been many debugging studies involving novice, intermediate, and skilled student programmers and expert professional programmers. Gould and Drongowski [3] asked professional programmers to find one of three types of errors (array, iteration, and assignment statement) in single-page Fortran programs. Several different types of debugging aids were studied including program listing, indicator of type of bug, line number where the error occurred, input data plus corresponding incorrect output data, and input data plus incorrect output data and desired (correct) output data. Their results showed the assignment-statement bug took four times longer to find and was found less frequently than the other two types of bugs. They also demonstrated that the line number of the statement where the error occurred was by far the best debugging aid. The line-number debugging aid was added to the study because there were little or no differences between the debugging speeds of the other four groups.

Gould [2] asked ten professional programmers, with at least four years experience, to identify the one line containing an error in the same 12 programs used in his previous study. Subjects were given a program listing and output and were told they could use an interactive debugging package if desired. The results were similar

Address correspondence to Curtis R. Cook, Computer Science Department, Oregon State University, Corvallis, OR 97331.

The Journal of Systems and Software 9, 197-207 (1989)

0164-1212/89/$3.50

to the previous study. Surprisingly, subjects used the debugging package on only 15% of the programs.

In verbal protocol analysis of 16 professional programmers, Vessey [7] found that chunking ability was a better measure of debugging expertise than years of programming experience. "Chunking" refers to a person's capacity to organize data elements into meaningful units that assist in problem solving. Vessey classified the 16 programmers, into groups of 8 novices and 8 experts, on the basis of their chunking ability. Each programmer was asked to debug a Cobol program containing a single error. She found that experts took less time to find and correct the bug, stated fewer hypotheses about the bug, and made fewer mistakes. Experts attempted to gain a high-level understanding of the program, with the goal of placing the error in context. Novices appeared more anxious to solve the problem, used a depth-first search strategy, and frequently changed hypotheses about the origin of the bug. Vessey concluded that although both groups used essentially the same basic debugging methods, the experts were more effective in their application of specific techniques.

This paper presents the results of a controlled experiment that compared the debugging abilities of novice, intermediate, and skilled student programmers. We were interested in comparing the debugging performance of beginning and experienced programmers given limited information about the bug. In particular, we concentrated on how programming experience and limited error messages affect debugging ability. The subjects were presented with two Pascal programs and were asked to find and correct the single error in each program. The only clue about the error was the error message; there was no actual output to compare with expected output and on-line debugging aids were not permitted.

We were also interested in testing two of Gould and Drongowski's [3] findings. They found that the line number of the statement where execution terminated was the most helpful debugging aid. To determine if this was true for both beginning and experienced programmers, we tested error messages with and without line numbers. They also found that, in almost every instance, professional programmers were able to correct an error once it was located. We were interested in seeing if this held true for various levels of student programmers.

This study is different from other studies in three respects. First, two types of semantic errors, array bounds and undefined variable, were studied. The programs compiled correctly, but, when executed, they terminated abnormally with an error message and no other output. Second, we considered two types of error messages, one with and one without the line number of the statement where the program terminated. Third, the debugging aid was restricted to just a simple error message with or without a line number. Subjects had no input data, no correct or expected outputs, and were not permitted to use other debugging aids.

2. EXPERIMENT

2.1 Purpose

In addition to testing the two findings of Gould and Drongowski as described above, this experiment investigated three general hypothesis:

1. Programmers' proficiency at using debugging aids increases with general programming experience; in this case, the debugging aids are the error message and line number of the statement at which the program terminated.
2. With increased experience, programmers become less dependent upon debugging aids; specifically, the line number of the statement at which the program terminated.
3. Programmers' ability to locate and correct errors increases with general programming experience; they become faster and make fewer mistakes.

2.2 Subjects

The subjects were novice, intermediate, and skilled student programmers. The levels of experience correspond to the number of computer science courses taken by the subjects. The novices were students in CS 212, the second term of a sophomore-level Pascal programming sequence. Intermediates were students in CS 319, the third term of a junior-level data-structures sequence. The skilled group were students in CS 416, the third term of a senior-level operating-systems sequence. CS 212 is a prequisite for CS 319, and CS 319 is a prerequisite for CS 416. There were 66 subjects in the novice category, 70 intermediates, and 57 in the skilled group.

2.3 Materials

A binary-search program and a median-calculation program were the two Pascal programs used in the experiment. The programs are given in Appendix A. All subjects were familiar with the binary-search algorithm and the bubble-sort algorithm used int he median program.

Each program contained a single semantic error. Semantic errors are defined here relative to the debugging process. Hence, we define semantic errors as violations of the semantic meaning (or processing

capabilities) of the programming language as detected by the computer system. (Syntactic errors are violations of the syntactic specification of the programming language as identified by a compiler.) Two types of semantic errors were studied: (1) an index exceeding array bounds and (2) use of an undefined variable (i.e., variable referenced before it was assigned a value). Although some Pascal compilers detect an undefined variable during compilation, it is generally considered a semantic error. When the program executes the statement where the error manifests itself, the computer system outputs an error message and terminates abnormally. These two errors were selected because they are errors commonly made by students.

The materials distributed to subjects consisted of a page of instructions and the two program listings. Appendix B contains an example test packet. The bottom of each listing had an error message with or without the line number of the statement where program execution was terminated. The terse error messages for the two types of errors were (1) index out of range [at line number xx], and (2) Undefined value [at line number xx].

Thus, the independent variables were programmer expertise (novice, intermediate, skilled), program type (binary search or median), error type (array bounds error or undefined variable), and message type (with and without line number).

2.4 Procedure

The experiment was conducted during the first 20 min of class. Subjects were randomly assigned to treatment groups for error type and message type and asked to debug the binary-search program first and then the median program. Subjects were told that each program contained a single error that could be repaired by changing only one statement. They were asked to circle the statement on the program listing that caused the error and then write the correct version of the statement. Subjects were informed that it was a timed exercise with a maximum of 10 min for each program. The experimenters wrote the elapsed time in minutes on the blackboard during the experiment. Subjects were asked to record the time when they finished the task. At the end of 10 min, they were instructed to turn the page and repeat the process using the median program without further work on the binary-search program.

3. RESULTS

A debugging score was computed for each program for each subject on a three-point basis: one point for circling the statement in error, one point for a semantically correct version of the incorrect statement, and one point if the repair was also syntactically correct. Dependent measures for each subject consisted of a debugging score (range: 0–3) and a debugging time (range: 0–9) for both of the programs.

Table 1. Debugging-Score Frequency

		Debugging score[a]			
		0	1	2	3
Binary search					
Novice	array bounds	10	1	9	10
	undefined variable	14	0	0	22
Intermediate	array bounds	1	1	18	15
	undefined variable	8	0	0	27
Skilled	array bounds	4	0	0	24
	undefined	2	0	0	27
Median calculation					
Novice	array bounds	22	0	1	10
	undefined variable	21	0	0	12
Intermediate	array bounds	17	3	1	15
	undefined variable	16	0	0	18
Skilled	array bounds	10	2	2	14
	undefined variable	10	0	0	19

[a] 0—bug not located.
1—bug located not corrected.
2—semantic correction only.
3—semantically and syntactically correct.

A frequency distribution of debugging score by skill level, program, and error type is shown in Table 1. The rarity of scores equal to 1 is striking. This score would only be obtained if a subject found the error but could not correct it. This data supports Gould and Drongowski's conjecture that if a programmer can find an error, then he or she can almost always correct the error. In our study of 386 debugging trials, only seven times did subjects find the error and not provide a semantically correct repair. Frequently, however, the correction was not syntactically proper and would have cause another error to occur (e.g., using / instead of DIV). Interestingly, there were no scores of 1 or 2 for the undefined variable error in either program. This reflects an all-or-none debugging condition; if the programmer can locate the error, then he or she can properly correct it.

Average debugging scores for all three levels of expertise and all program conditions are shown in Table 2. In every experimental condition, debugging score improved with increase in subjects' experience. In all but one condition, the median program with an unde-

Table 2. Average Debugging Scores

		Novice	Intermediate	Skilled
Binary search				
Array bounds	no line number	1.5	2.2	2.5
	line number	1.8	2.4	2.6
Undefined variable	no line number	1.1	1.5	2.5
	line number	2.4	3.0	3.0
Median calculation				
Array bounds	no line number	0.37	1.1	1.2
	line number	1.5	1.6	2.2
Undefined variable	no line number	1.1	1.4	2.2
	line number	1.0	1.7	1.7

fined variable and a message without a line number, subjects debugging scores for the line-number condition exceeded those in the no-line-number condition.

Averages for debugging times are shown in Table 3. Average debugging time decreased with increase in expertise in all but two cases. Intermediate subjects were slightly faster than the skilled subjects on the binary program with an array-bounds error and a line-number message. Novices were slightly faster than intermediates on the median program with an undefined variable and a message without a line number. Debugging times for the line-number conditions were less than or equal to those for the no-line-number condition in all but three instances: novice and skilled programmers working on the

Table 3. Average Debugging Times

		Novice	Intermediate	Skilled
Binary search				
Array bounds	no line number	4.1	3.5	3.5
	line number	4.1	3.1	4.0
Undefined variable	no line number	6.5	5.7	4.0
	line number	3.7	3.0	2.8
Median Calculation				
Array bounds	no line number	5.9	4.9	4.7
	line number	5.1	4.5	3.4
Undefined variable	no line number	5.5	5.8	3.6
	line number	5.9	5.1	4.5

Table 4. Significant Effects

	F	Degree of freedom	Probability
(a) Analysis of Debugging Score			
Binary search			
Expertise main effect	11.9	2,181	$p < 0.0001$
Message main effect	15.53	1,181	$p < 0.0001$
Error X message interaction	7.87	1,181	$p < 0.0056$
Median calculation			
Expertise main effect	5.04	2,181	$p < 0.0074$
Message main effect	4.10	1,181	$p < 0.0472$
Error X message interaction	5.51	1,181	$p < 0.0200$
(b) Analysis of Debugging Times			
Binary search			
Expertise main effect	4.98	2,181	$p < 0.0079$
Error main effect	4.38	1,181	$p < 0.0378$
Message main effect	15.41	1,181	$p < 0.0001$
Error X message interaction	16.00	1,181	$p < 0.0001$
Median Calculation			
Expertise main effect	6.15	2,181	$p < 0.0026$

undefined variable error in the median program, and skilled programmers working on the array-bounds error in the binary program.

The data were analyzed using the BMDP4V [1] multivariate analysis-of-variance program. For the binary-search program, significant main effects of expertise and message type, and a significant error by message interaction, were found for both the debugging score and time. The main effect of error type was also significant for debugging time but not debugging score. F statistics and corresponding probabilities are shown in Table 4. For the median-calculation program, effects for debugging score followed the same pattern as in the binary-search program, but for debugging times, only a main effect of expertise was found to be significant.

Debugging score interactions between error type and message type are shown in Figure 1 for each level of expertise. Two-way interactions between error type and message type were significant for debugging score on both the binary-search and median-calculation programs. Corresponding debugging-time interactions are shown in Figure 2. Significant error by message debugging-time interaction was found in only the binary program. Although the three-way interaction between expertise, error type, and message type was not significant, the data are presented with separation by skill level to show possible trends in this regard.

The effects of message type across skill levels are best illustrated by differences in debugging success as defined to be the percent of subjects capable of semantically correcting the error in the given amount of time. These percentages are shown in Table 5 for each program and both programs combined. Overall, percent

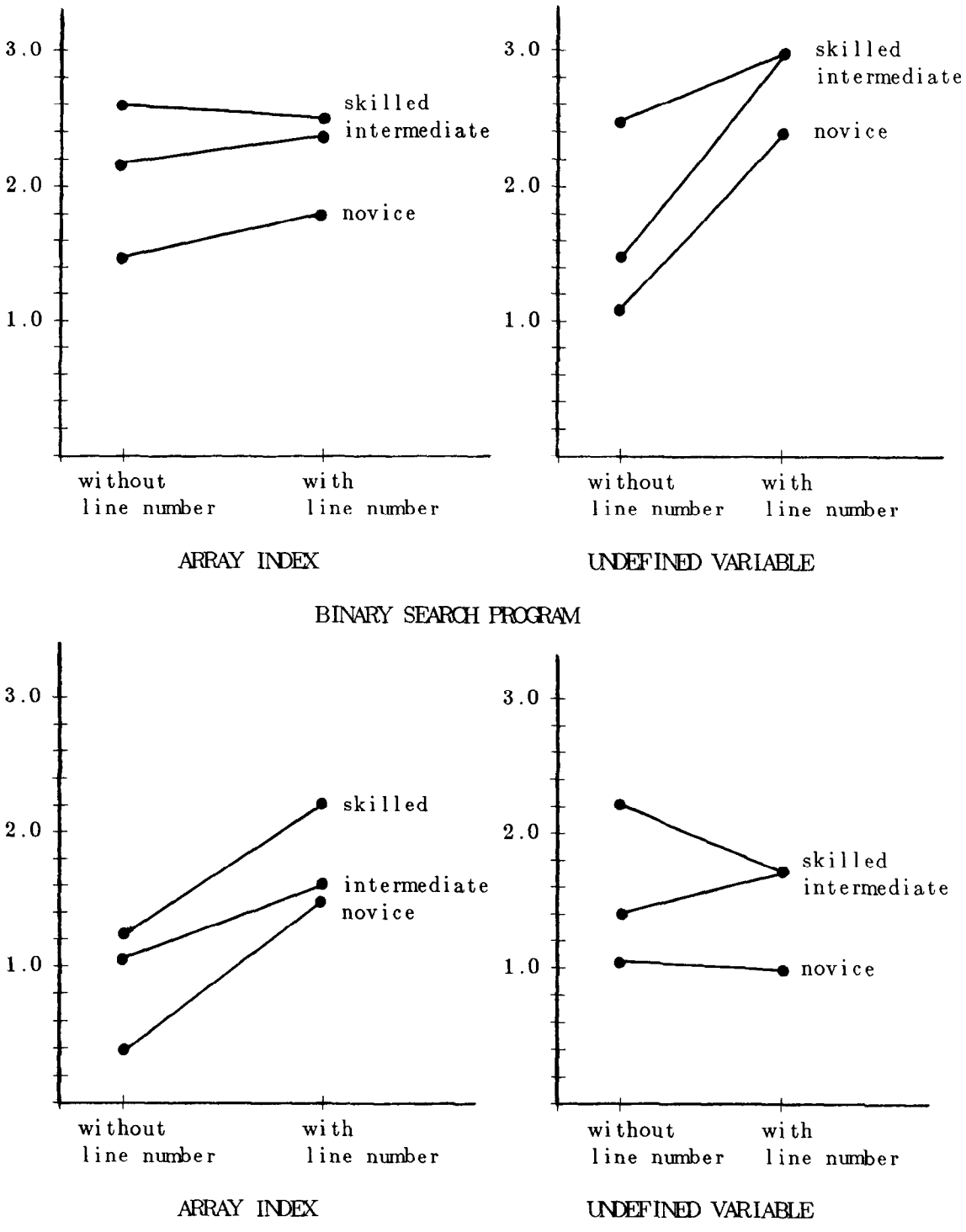

Figure 1. Debugging score (expertise X error X message).

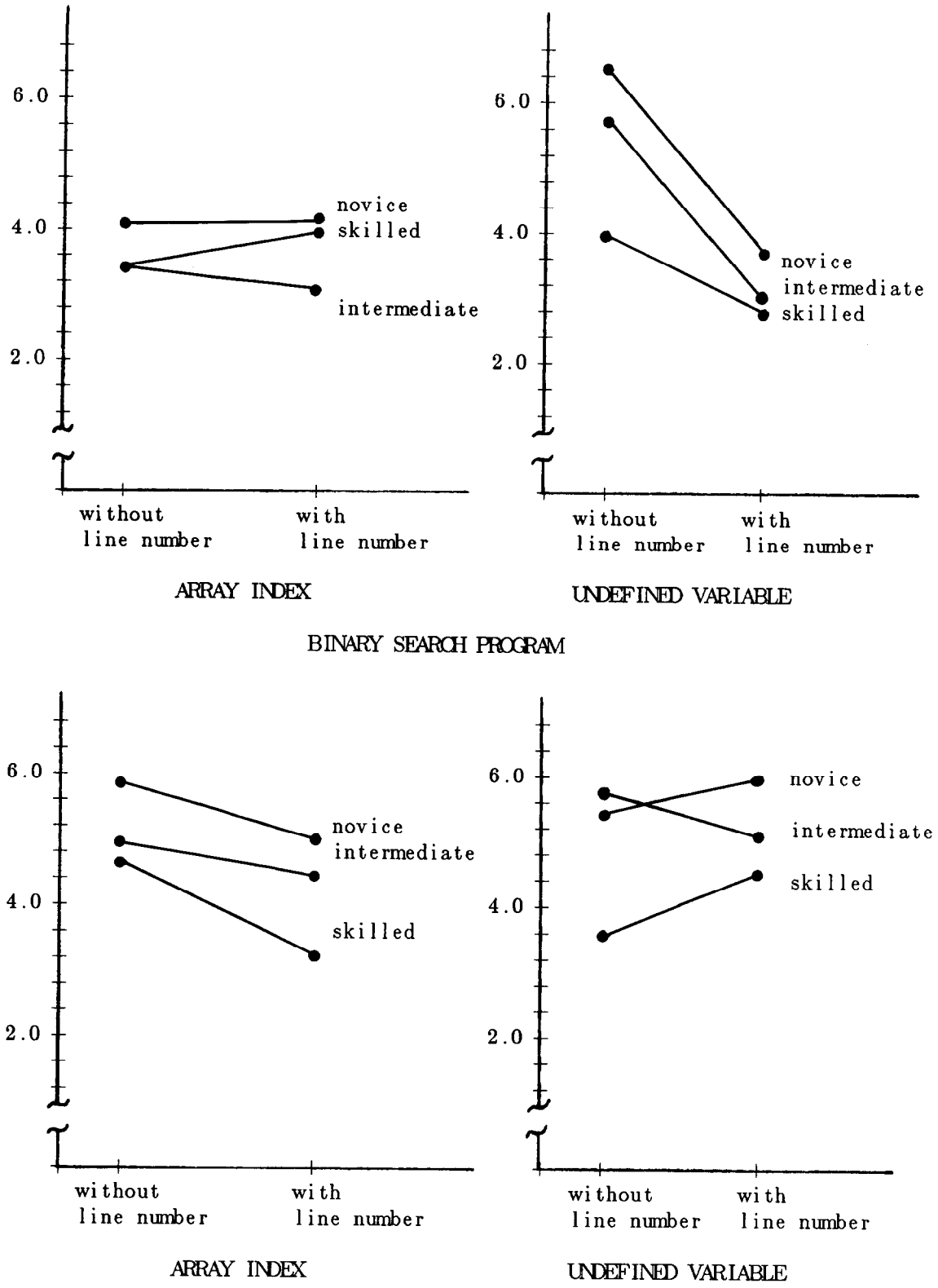

Figure 2. Debugging time (expertise X error X message).

Table 5. Percent Success

		Novice	Intermediate	Skilled
Binary search				
Array bounds	no line number	62.5	88.8	85.7
	line number	66.6	100	86.6
Undefined variable	no line number	38.8	52.9	85.7
	line number	82.3	100	100
Median Calculation				
Array bounds	no line number	12.5	38.8	42.8
	line number	52.9	50.0	71.4
Undefined variable	no line number	37.5	47.0	73.3
	line number	35.2	58.8	57.1
Both programs				
	no line number	37.8	57.1	71.9
	line number	59.0	77.1	78.9

success increases with both expertise and when error messages include line numbers. However, the positive effect of having line numbers with error messages appears to diminish with increased skill level.

4. DISCUSSION

Results show support for all three hypotheses. Experienced programmers are more successful in locating and correcting errors in program code. They locate errors faster and make the proper correction more often than less experienced programmers. Furthermore, they are less dependent upon debugging clues (like line numbers) and can perform reasonably well without such aids. They appear to make better use of available debugging aids, which may be due to practice, a wider range of experience to draw from, or increased domain knowledge. Intermediate programmers appear stable and predictable. Their performance invariably improves when error messages contain line numbers, but in general, is not as good as that of more experienced programmers. Novice programmers' debugging success deteriorates in the absence of clues indicating the nature and location of the errors. Their performance is very poor in the absence of debugging aids. More experienced programmers are less impacted by the loss of these aids.

Vessey's suggestion, that chunking ability is a better measure of expertise than years of experience, may partially explain why performance differences between the intermediate and skilled groups were not as great as expected. Our subject grouping was based entirely on computer science coursework and not strictly on programming ability or on the ability to organize "memory chunks." Another explanation is that intermediates had more recent Pascal programming experience. They were in the third (Spring) term of a data-structures sequence that required much Pascal programming. On the other hand, subjects in the skilled group completed the data-structures sequence the previous year and were in the third term of an operating-systems sequence that required much C programming.

Skill-level differences in debugging performance apparently decrease with loss of application domain knowledge. The binary-search program was chosen as an example of a well-studied program familiar to all subjects. The median program was chosen as an unusual computation that subjects would not have studied as a whole even though all were familiar with the parts of the program. Although expertise differences hold across both programs, in general, the differences are less pronounced in the median-calculation program. This can be seen by comparing the group averages for both programs as shown in Table 6. Debugging-score group differences for the median program are considerably less than those for the binary-search program. However, group differences in debugging time for the median program are comparable to those for the binary-search program. This suggests that domain knowledge increases the probability of finding an error but does not necessarily decrease the time required to find and correct the error.

Shifts in debugging strategy by skilled programmers could, in part, account for some of the peculiarities in our data. Sheppard et al. [6] found that their subjects used two general debugging strategies: (1) understand the entire program before searching for the section with the bug, and (2) use clues in the output to go directly to the section of the program with the bug. The use of a particular strategy was dependent upon the type and amount of available debugging aids. It is interesting that our skilled programmers, looking for an undefined

Table 6. Program X Skill-Level Averages

	Novice	Intermediate	Skilled
Debugging Score			
Binary search	1.7	2.3	2.6
Median calculation	1.3	1.4	1.8
Debugging time			
Binary search	4.7	3.8	3.6
Median calculation	5.6	5.1	4.0

variable in the median program, with no line number for a clue, actually scored higher, more often, and did so faster, than comparable programmers given the same error message with a line number. This suggests either the occurrence of two different strategies (invoked by the different message types) or the possibility of a negative effect from the line-number condition. Because this trend is not observed in the intermediate programmers, and marginal in the novice programmers, we speculate that skilled programmers change strategies according to the type of debugging aids available. This conjecture is supported by Sheppard's findings, but is inconsistent with Vessey's conclusion that experts first try to gain an overall understanding of the program before trying to isolate a bug. This discrepancy may be due to differences between professional and student programmers, or caused by the different methodologies used to gathering data. Further study is need to resolve this issue.

An inadvertent error in one set of test materials provided additional support for the theory that the debugging strategy of skilled programmers depends on the information available. Besides the seeded error, the function in the binary-search program given to the skilled programmers did not assign a value to the function name in one case. Even though the instructions indicated that the program contained a single error, six subjects noted and corrected both errors. Interestingly, all six had received the program version containing the error message without a line number. None of the subjects, given the error message with line number, found the accidental error. This strongly suggests that the six subjects used a comprehension debugging strategy and observed both errors while attempting to understand the entire program. In our analysis, we excluded the data for these subjects.

Specifically, we believe that skilled programmers first use one of the two debugging strategies, the choice depending on the problem environment, and then, if no progress is made, promptly shift to the other approach. Occasional shifts to and from the complementary strategy may occur later, as the programmer gains knowledge of the specific application.

We suspect that the use of multiple debugging strategies has not been learned by less experienced programmers. Intermediate programmers had improved scores and substantial time decreases whenever line numbers accompanied the error messages. Their predictable performance indicates a stable debugging approach across all treatment conditions. Novice programmers appear lost without debugging clues. Because they are relatively new to the programming language syntax and control structures, we speculate that their debugging efforts are near-random repetitions of past experiences rather than organized strategies as outlined above.

5. CONCLUSIONS

Although this work demonstrates the existence of debugging behavioral differences due to differences in programming expertise, it does not explain how or when debugging strategies are employed; nor does it explain the interaction between type of error and the presence or absence of line numbers on error messages. The strong interaction between error and message supports the notion of different classes of errors based on difficulty of location and correction. That is, there appears to be some errors where debugging clues are less helpful than others, and some errors where debugging clues are crucial for certain programmers. This has ramifications in both computer science education and programming-language design.

Additional studies should investigate when programmers use the program comprehension strategy versus when they use the isolation strategy. How common is the latter strategy? Is the former strategy used when the latter fails? Is the choice of strategy dependent on personality type as suggested by Littman et al. [5]? Future work should address the following issues:

1. Can these findings be replicated with logical errors?
2. Does minimizing problem domain knowledge remove all skill-level differences?
3. How do multiple errors affect shifts in debugging strategy?
4. Why are intermediate programmers so consistent in their approach to debugging? When do they start using strategy shifts?
5. What effect does proximity between error location and the statement(s) causing program termination have on debugging behavior?

ACKNOWLEDGMENT

We wish to acknowledge Chris Weiss and Saeed Sadation for their assistance in formulating and testing materials used in this study.

REFERENCES

1. W. J. Dixon, *BMDP Statistical Software 1981*, University of California Press, Berkeley, California, 1981,
2. J. D. Gould, Some Psychological Evidence on how People Debug Computer Programs, *Int. J. Man-Machine Studies* 7, 151–182 (1975).
3. J. D. Gould and P. Drongowski, An Exploratory Study of Computer Program Debugging, *Human Factors* 16, 258–277 (1974).
4. L. Gugerty and G. M. Olson, Comprehension Differences

in Debugging by Skilled and Novice Programmers, in *Empirical Studies of Programmers*, (E. Soloway, and S. Iyengar, eds.), Ablex, Inc., Norwood, New Jersey, 1986, pp. 13–27.
5. D. Littman, J. Pinto, S. Letovsky & E. Soloway, Mental Models and Software Maintenance, in *Empirical Studies of Programmers*, (E. Soloway, and S. Iyengar, eds.), Ablex, Inc., Norwood, New Jersey, 1986, pp. 80–98.
6. S. Sheppard, B. Curtis, P. Milliman, and T. Love, Modern Coding Practices and Programmer Performance, *Computer* 12(12), 41–49 (1979).
7. I. Vessey, Expertise in Debugging Computer Programs: A Process Analysis, *Int. J. Man-Machine Studies* 23, 459–494 (1985).
8. E. Youngs, Human Errors in Programming, *Int. J. Man-Machine Studies* 6, 361–376 (1974).

APPENDIX A

```
1   { Binary search algorithm }
2   program binary(input, output);
3   const
4     size = 21;
5   type
6     arraytype = array [1..size] of integer;
7   var
8     t : arraytype;
9     i, j : integer;
10
11  function BinarySearch(a : arraytype; key : integer) : integer;
12  var
13    low, high, middle : integer;
14  begin
15    low := 1;
16    high := size;
17    while low <> high do
18      begin
19        middle := (low + high);
20        if key <= a[middle]
21          then
22            high := middle
23          else
24            low := middle + 1;
25      end;
26    if key = a[low]
27      then
28        BinarySearch := low
29      else
30        BinarySearch := 0;
31  end; { BinarySearch function }
32
33  begin { Main program }
34    for i := 1 to 21 do
35      t[i] := 2 * i;
36    j := 12;
37    writeln('key = ', j, ' value = ', BinarySearch(t, j));
38    j := 100;
39    writeln('key = ', j, ' value = ', BinarySearch(t, j));
40  end.
```

```
********** ERROR **********
error message : index out of range at line number 20
**********
```

A. Binary program with index out of range error.

```
1   { Binary search algorithm }
2   program binary(input, output);
3   const
4     size = 21;
5   type
6     arraytype = array [1..size] of integer;
7   var
8     t : arraytype;
9     i, j : integer;
10
11  function BinarySearch(a : arraytype; key : integer) : integer;
12  var
13    low, high, middle, size : integer;
14  begin
15    low := 1;
16    high := size;
17    while low <> high do
18      begin
19        middle := (low + high) div 2;
20        if key <= a[middle]
21          then
22            high := middle
23          else
24            low := middle + 1;
25      end;
26    if key = a[low]
27      then
28        BinarySearch := low
29      else
30        BinarySearch := 0;
31  end; { BinarySearch function }
32
33  begin { Main program }
34    for i := 1 to 21 do
35      t[i] := 2 * i;
36    j := 12;
37    writeln('key = ', j, ' value = ', BinarySearch(t, j));
38    j := 100;
39    writeln('key = ', j, ' value = ', BinarySearch(t, j));
40  end.
```

```
********** ERROR **********
error message : undefined value at line number 16
**********
```

B. Binary program with undefined variable error.

```
1   { purpose: to find median of a given set of integers }
2   program Median(input, output);
3   type
4     arraytype = array [1..100] of integer;
5   var
6     a : arraytype;
7     i, j, size, temp : integer;
8     medianvalue : real;
9
10  procedure Readdata(var a : arraytype; var size : integer);
11  var
12    i : integer;
13  begin
14    i := 1;
15    while (not eof) do
16      begin
17        readln(a[i]);
18        i := i + 1;
19      end;
20    size := i - 1;
21  end; { Readdata function }
22
23  begin {Main program }
24    Readdata(a, size);
25    { sort array }
26    for i := 1 to size - 1 do
27      for j := 1 to size do
28        if a[j] > a[j+1] then
29          begin
30            temp := a[j] ;
31            a[j] := a[j+1];
32            a[j+1] := temp;
33          end;
34    if (size mod 2 = 0)
35    then
36      medianvalue := (a[size div 2] + a[size div 2 +1]) / 2.0
37    else
38      medianvalue := a[size div 2];
39    writeln('median = ', medianvalue:5:2);
40  end.
```

```
********** ERROR **********
error message : index out of range at line number 28
**********
```

C. Median program with index out of range error.

```
1   { purpose: to find median of a given set of integers }
2   program Median(input, output);
3   type
4     arraytype = array [1..100] of integer;
5   var
6     a : arraytype;
7     i, j, size, temp : integer;
8     medianvalue : real;
9
10  procedure Readdata(a : arraytype; var size : integer);
11  var
12    i : integer;
13  begin
14    i := 1;
15    while (not eof) do
16      begin
17        readln(a[i]);
18        i := i + 1;
19      end;
20    size := i - 1;
21  end; { Readdata function }
22
23  begin {Main program }
24    Readdata(a, size);
25    { sort array }
26    for i := 1 to size - 1 do
27      for j := 1 to size - 1 do
28        if a[j] > a[j+1] then
29          begin
30            temp := a[j] ;
31            a[j] := a[j+1];
32            a[j+1] := temp;
33          end;
34    if (size mod 2 = 0)
35    then
36      medianvalue := (a[size div 2] + a[size div 2 +1]) / 2.0
37    else
38      medianvalue := a[size div 2];
39    writeln('median = ', medianvalue:5:2);
40  end.
```

```
********** ERROR **********
error message : undefined value at line number 28
**********
```

D. Median program with undefined variable error.

APPENDIX B. EXAMPLE TEST PACKET

Written Instructions to Subjects

This test is designed to measure your debugging proficiency. It will not effect your grade in this, or any other, course. It will not effect your standing in computer science in any way.

→Do not turn the page until told to do so←

The test is made up of two parts.

Part I: This part contains a listing of a *standard* Pascal program that contains *one* error. The program compiles but will not execute properly. At the bottom of the listing is an execution error *message. The error can be corrected by changing only one statement.* Find and circle the one statement that is causing the error. Then correct the error by changing only that statement.

It is a timed test, you will have 10 min to complete Part I.

STOP when you have completed Part I.

Wait for your instructor to tell you when to start Part II.

Part II: This part is the same as Part I except that a different Pascal program is listed. The program contains *one* error; it compiles but will not execute properly. At the bottom of the listing is *an execution error message.*

PART I

```
1   { Binary search algorithm }
2   program binary(input, output);
3   const
4     size = 21;
5   type
6     arraytype = array [1..size] of integer;
7   var
8     t : arraytype;
9     i, j : integer;
10
11  function BinarySearch(a : arraytype; key : integer) : integer;
12  var
13    low, high, middle : integer;
14  begin
15    low := 1;
16    high := size;
17    while low <> high do
18      begin
19        middle := (low + high);
20        if key <= a[middle]
21          then
22            high := middle
23          else
24            low := middle + 1;
25      end;
26    if key = a[low]
27      then
28        BinarySearch := low
29      else
30        BinarySearch := 0;
31  end;  { BinarySearch function }
32
33  begin { Main program }
34    for i := 1 to 21 do
35      t[i] := 2 * i;
36    j := 12;
37    writeln('key = ', j, ' value = ', BinarySearch(t, j));
38    j := 100;
39    writeln('key = ', j, ' value = ', BinarySearch(t, j));
40  end.
```

```
********** ERROR **********
error message : index out of range at line number 20
**********
```

STOP: Do not go on to Part II until you are told to.

The error can be corrected by changing only one statement. Find and circle the one statement that is causing the error. Then correct the error by changing only that statement.

It is a timed test, you will have 10 min to complete Part II.

DO NOT go back to Part I.

PART II

```
1   { purpose: to find median of a given set of integers }
2   program Median(input, output);
3   type
4     arraytype = array [1..100] of integer;
5   var
6     a : arraytype;
7     i, j, size, temp : integer;
8     medianvalue : real;
9
10  procedure Readdata(a : arraytype; var size : integer);
11  var
12    i : integer;
13  begin
14    i := 1;
15    while (not eof) do
16      begin
17        readln(a[i]);
18        i := i + 1;
19      end;
20    size := i - 1;
21  end; { Readdata function }
22
23  begin {Main program }
24    Readdata(a, size);
25    { sort array }
26    for i := 1 to size - 1 do
27      for j := 1 to size - 1 do
28        if a[j] > a[j+1] then
29          begin
30            temp := a[j] ;
31            a[j] := a[j+1];
32            a[j+1] := temp;
33          end;
34    if (size mod 2 = 0)
35    then
36      medianvalue := (a[size div 2] + a[size div 2 +1]) / 2.0
37    else
38      medianvalue := a[size];
39    writeln('median = ', medianvalue:5:2);
40  end.
```

```
********** ERROR **********
error message : undefined value
**********
```

STOP: Do not go back to Part I. Wait for instructions.

STOP when you have completed Part II and wait for further instructions.

Please enter your social security number

→_____

Please circle one → Male / Female

When your instructor says "Go" Turn the page and start Part I.