Experimental Investigations of Computer Program Debugging and Modification

Ben Shneiderman

Don McKay

Computer Science Department

Indiana University

Bloomington, Indiana 47401

TECHNICAL REPORT NO. 48

EXPERIMENTAL INVESTIGATIONS OF
COMPUTER PROGRAM DEBUGGING AND MODIFICATION

BEN SHNEIDERMAN
DON MCKAY

APRIL, 1976

Prepared for 6th Congress of the International Ergonomics Association hosted by the Human Factors Society, July 12-16, 1976, College Park, Maryland.

EXPERIMENTAL INVESTIGATIONS OF COMPUTER PROGRAM DEBUGGING AND MODIFICATION

Ben Shneiderman and Don McKay

Indiana University, Bloomington, Indiana 47401

ABSTRACT

Although greater emphasis is placed on the task of computer program composition, debugging and modification often consume more time and expense in production environments. Debugging is the task of locating syntactic and semantic errors in programs and correcting these errors. Modification is the change of a working program to perform alternate tasks.

The factors and techniques which facilitate debugging and modification are poorly understood, but are subject to experimental investigation. Controlled experiments can be performed by presenting two groups of subjects with two forms of a program or different programming aids and requiring the same task. For example, in one study we presented an 81 line FORTRAN program containing three bugs to distinct groups of subjects. One of the groups received a detailed flowchart, but our results indicated that this aid did not facilitate the debugging procedure. Similar negative results were obtained for a modification task.

In other experiments, comments and meaningful variable names were useful in debugging and modularity facilitated modification. Other potentially influential factors, which are subject to experimental study, include indentation rules, type of control structures, data structure complexity and program design.

These and other human factor experiments in programming have led to a cognitive model of programmer behavior which distinguishes between the hierarchically structured, meaningfully acquired semantic knowledge and the rotely memorized syntactic knowledge. Errors can be classed into syntactic mistakes which are relatively easy to locate and correct and two forms of semantic mistakes. Semantic errors occur while constructing an internal semantic structure to a representation in the syntax of a programming language. Modification is interpreted as the acquisition of an internal semantic structure by studying a program, followed by modification of this structure and revision of the code.

INTRODUCTION

Debugging of computer programs is a complex human information processing task which consumes 25-50% (Boehm, 1973) of the initial program development process. Modification or maintenance of computer programs which are already in use require the total programmer resources of some organizations (Boehm, 1975). An improvement which simplifies or facilitates these tasks can have substantial impact on the programming process by reducing costs, speeding up the implementation process, making adaptation easier and improving the reliability of computer-based systems.

This paper reviews previous human factors research in this area, presents results from recent experiments and interprets debugging and modification in terms of a recently proposed cognitive model of programmer behavior (Shneiderman and Mayer, 1975).

Previous Research

Most authors distinguish two types of debugging errors: syntactic errors which the compiler recognizes and semantic errors (or logic errors) which the compiler cannot recognize. This is a crude dichotomy since different compilers for the same language have varying error checking facilities. Still, this dichotomy is useful and is operationally effective since a subject's perception of a syntactic or semantic error is dependent on the compiler that he/she works with.

Studies of syntactic errors have been made in conjunction with an assigned program composition task to a selected group of subjects (Gannon and Horning, 1975; Youngs, 1974; Miller, 1973; Sime, Green and Guest, 1973) or by merely capturing the output of compilers in a large computing facility (Boies and Gould, 1974; Litecky and Davis, 1975). These studies of bugs made during program composition tasks gives realistic insights to programmer behavior and suggest improvements to compilers and/or languages to minimize the impact of syntactic errors.

Another approach to debugging

experiments has been to present subjects with program listings containing one or more semantic errors and require the subjects to locate the errors. This artificial, but more controlled, environment is defended by researchers on experimental design grounds, on the existence of debugging consultants in many computing centers, and on the reports from many installations that debugging is often done by a person other than the original creator of the program (Gould and Drongowski, 1974; Gould, 1974).

Classifying semantic errors is more of a challenge, since the cause of an error is difficult to determine. Gould (1974) and Gould and Drongowski (1974) both focus on assignment bugs (errors in assignment statements), array bugs (errors which cause array bounds to be exceeded) and iteration bugs (errors which cause an incorrect number of executions of a loop) but all of their bugs result from a minor error in the programming language representation of an algorithm. We will classify these errors as well as the syntactic errors as "composition errors" since they result from the improper representation of an algorithm. A more complex error, called a "formulation error", results from an improper understanding of the solution to a problem. These errors may be more global than one line errors and more difficult to detect.

Although maintenance and program modification, that is, changes made to a functioning program, consume a large part of the resources of a programming installation, relatively little research has been done in this area. Lucas and Kaplan (1974) attempted a modification experiment in conjunction with work on structured programming. Their results are an initial confirmation of the ease of modification of programs written in a structured manner, but much work remains to be done to confirm these results.

Debugging and modification are similar in that they each require a programmer to take a program listing and a program description and make changes to the listing. In the case of debugging the change is designed to correct a fault, while in modification the change is designed to alter the present function of the given program. Research in this area could simplify debugging and modification by providing suggestions for improved stylistic standards (for example, commenting techniques, indentation rules, standards for variable name selection), program design guidelines (modularization rules, parameter passing techniques, flowchart rules, top-down vs. bottom-up standards) or new language features (improved control structures, new operators, simplified programming language syntax, machine checkable assertions). Other benefits of such research would be to directly simplify debugging and modification by improving on-line or batch debugging facilities such as traces, snapshots, dumps and monitors. Longer range goals would be to understand the intellectual skills necessary for these two tasks, which might eventually lead to improved programming techniques. Wirth's stepwise refinement technique (1971), Mills' top-down design and Dijkstra's approach to structured programming (1972) are all geared to simplifying debugging and modification as well as program composition. Although these ideas were outgrowths of programming experience, they all have familiar analogues in the problem solving and cognitive psychology literature (Shneiderman and Mayer, 1975). It may be possible to take other lessons directly from cognitive psychology as well. Geller (1975) proposes debugging other languages in APL since he feels that APL is a convenient high level language in which to represent programs. Others have advocated LISP as the ideal language for discussing algorithms to be implemented in other languages. These approaches reflect the popular technique of using "pseudo-code" (an informal high level problem oriented notation familiar to the programmer) in early stages of program development to minimize notational errors.

Although the potential benefits are great, experimentation in this area is a challenge. The complex human information processing tasks involved are difficult to isolate and study. New techniques will have to be developed and validated and the results will have to be carefully replicated and verified.

The three basic experimental paradigms might be called case study, protocol analysis and controlled experiment. The case study approach is to take data about programmers pursuing their normal activities with as little interference as possible. An enormous amount of data can be captured, but it may be difficult to pinpoint critical issues and new techniques cannot be tested. The protocol analysis technique is to have programmers introspect and attempt to reveal to the experimenter what mental processes he/she uses during debugging or modification. This method reveals more complex behaviors, but it is a purely subjective experiment and the introspection process can interfere with normal performance. The controlled experiment approach requires careful planning and must focus on a small number of specific issues if reliable results are to be obtained. Controlled experiments are more objective and it is possible to test techniques which programmers might not normally use. Combinations of these approaches are also feasible.

We have favored the controlled experiment technique in our previous work (Shneiderman, 1975; Shneiderman, Mayer, McKay and Heller, 1975; Shneiderman and Mayer, 1975) and report on further research in this paper. We feel that a series of

replicable specific objective results will eventually give us a more complete and precise picture of programmer behavior; much like a mosaic which is made up of small but numerous discrete tiles.

The obvious experimental technique for testing debugging is to present subjects with a program description and listing which contains one or more bugs and require the subject to find the bugs. Issues that can be tested in this manner include: the utility of detailed and/or macro flowcharts; the effect of meaningful variable names, comments, indentations; the impact of various modularization schemes or data sharing techniques; the usefulness of different output listings, dumps, traces or snapshots; or the use of various control structures. Our experiments are comparative; groups of subjects receive different program aids and we evaluate the bug-finding performance of one group against the other. Full credit is given if the bugs are located and corrected. Partial credit can be given if the bug is only located but not corrected properly. In one of our experiments we allowed subjects to make several guesses at the bug's location (if they were not sure of the bug's location); the credit given was inversely proportional to the number of guesses. Grading the correction can be difficult, if the subject does not precisely make the correction expected. Partial credit was assigned by experienced graders, but the technique could be made more reliable if multiple graders were used.

A more objective, but questionable, testing technique is to give multiple choice questions about the location and/or correction of a bug. This approach is easy to grade but may eliminate central issues in the debugging experiments.

Timing data may also be useful, but since accuracy in bug finding is the critical issue, the subjects might be given immediate responses as to the correctness of their guesses and the time to accurately find the bug would be recorded. Finally, a subjective estimate of the difficulty of locating the bug could be recorded.

Experimental testing of modification is as difficult since the new or deleted segments of code must be graded on a qualitative basis. Multiple experienced graders should be used and criteria should be established to ensure consistent grading.

Besides the testing technique, a number of important experimental factors such as program size, program application area, program complexity, subject experience and subject motivation must be considered. Programmers working on long programs may apply different strategies than those working on short programs. Experience with an application area should help in locating certain kinds of bugs. A final factor of importance is that unmotivated programmers may be able to find certain bugs but would not invest the intellectual effort to locate more subtle and complex bugs.

Experiments

We investigated four factors in earlier studies (Yasukawa, 1974; McKay, 1974; Kinicki and Ramsey, 1974; Shneiderman, Mayer, McKay and Heller, 1975): commenting, choice of variable name, modularity and use of flowcharts. In these experiments debugging and modification are considered two different task domains. Debugging is the task of correcting a program which does not run appropriately. Modification is the changing of a program to meet new requirements. Program comprehension is an important factor within each experiment since either modification or debugging is attempted by an individual who is not the original author of the program.

Commenting, Debugging and Modification

In an experiment on commenting techniques (Yasukawa, 1974), subjects from an introductory programming course were given a 30 line FORTRAN program and asked multiple choice questions to determine program comprehension. The program contained either high level organizational comments or no comments. Subjects who received the program with organizational comments performed well on questions which required a general understanding of the program. Subjects who had only the program performed as well as the comment group on questions which required program execution. In a further experiment, novice programmers were asked to memorize a 26 line FORTRAN program with either high level organizational comments or low level detailed comments which merely explained a line of code. The subjects in the organizational group recalled significantly more of the program than the detail group. Subjects were also given three modification tasks for three different programs. Each program contained either organizational comments or detailed comments. On two out of three modifications, the high level comment group performed better than the low level comment group. The results of the two experiments indicate high level organizational comments facilitate program comprehension. The high level comments aid the "chunking" of the program into logical modules while low level comments do not impose a logical structure to the program. As a documentation tool, high level comments appear useful.

In a second experiment by Yasukawa (1974), subjects who were novice programmers received a 25 line FORTRAN program either with high level comments or without comments. The program contained one error. Their task was to locate the bug and to correct the program. Subjects in both groups performed

equally well. Many subjects said they did not use the comments (if present) but used some sort of hand-simulation to find the error. The results of the experiment and the remarks of the subjects indicate that comments are not useful for a debugging task. The fact that many of the subjects did not use the comments suggests that they have already learned that comments may "hide" the error by providing the programmer with misleading information as to what the code really does.

In summary, high level organizational comments facilitate program comprehension as measured by a multiple choice quiz and a recall task; but comments do not appear to aid a debugging task. Since high level comments seem to convey information which is useful in understanding a working program, one would expect a modification task to be facilitated by high level comments. In a debugging task the problem is to find the error in the code itself. Comments seem to be ignored for a debugging task.

Modularity and Modification

Modularity, the blocking of a program into logical "chunks", was the subject of another experiment (Kinicki and Ramsey, 1974). Subjects were novice-intermediate programmers in a second semester introductory assembly language course and intermediate programmers in an upper division computer science course. Both groups of subjects had just recently learned the assembly language used for the experiment. Subjects were provided a modular, non-modular, or random modular version of the same assembly language program. After a period of study, the subjects were given a quiz to determine program comprehension, included in the quiz were items which required the subjects to modify the program. The results showed the modular group performed the best for both types of quiz questions, followed by the non-modular and random modular groups. While this experiment suggests programs be decomposed into conceptual modules, it does not supply any guidelines as to how a program should be modularized. Certainly the degree of modularization would vary with the sophistication of the programmer. The danger here is what one programmer views as a module may not be comprehensible to another programmer. Further experimentation with module decomposition is required.

Flowcharts, Debugging and Modification

The use of flowcharts, which supplement a program, as a documentation tool has been an issue for a number of years. We recently concluded a number of experiments on the utility of flowcharts (Shneiderman, Mayer, McKay and Heller, 1975), two of which are relevant to this paper.

The first experiment dealt with debugging and comprehension. Subjects were students from a second semester programming course and from two different flowcharting backgrounds. Half the subjects had been required to turn in flowcharts with their programming assignments while the other half had not been required to do so. Each subject was given an 81 line FORTRAN program which contained three errors. In addition to the program, the subjects were given either a one page macro (high level), a four page micro (detailed), or no flowchart. Initially, the subjects attempted to find the errors. After 45 minutes, the subjects were told what the errors were and how to correct them. The subjects were then given a comprehension quiz. While none of the main effects were significant (micro vs macro, macro vs none, micro vs none), for the debugging score, comprehension quiz or total score, there was a mild interaction effect. For those subjects who had used flowcharts as part of their programming routine, the group which was supplied with the micro flowchart performed slightly better. For the group which had not used flowcharts, the subjects who had only the program performed slightly better. The results suggest that flowcharting as a debugging tool is marginally useful to those who have been trained with flowcharts.

A second flowchart experiment with a modification task was also conducted. The same experimental design was used as in the previous flowchart experiment. Subjects were grouped according to experience and then randomly assigned to one of the experimental groups, either a no flowchart group, a macro flowchart group, or a micro flowchart group. Each subject was provided with a listing of a 70 line FORTRAN program, output and a list of modifications. The most critical type of error, incorrect placement of the modification within the existing program, was as frequent in each of the six experimental groups. The trend concerning flowcharting experience observed in the first experiment was not observed in the modification experiment. While the results for the program-only group versus the micro flowchart group were not expected to differ, the program-only group versus the macro flowchart group results were surprising. A macro flowchart is not equivalent to the program it describes since a one to one correspondence does not exist between the program statements and the flowchart. A macro flowchart, as a high level description of a program, "chunks" the program into modules which should facilitate placement of a modification. Further experiments with macro flowcharts are necessary.

From the results of these two experiments and other experiments with flowcharts, we concluded flowcharts do not seem to be useful in a debugging or modification task. Since the experiments were conducted utilizing small programs, further research with larger complex programs is needed.

Variable names and debugging

Another factor influencing program comprehension is the choice of variable names used within a program. Newsted (1973) gave subjects an "easy" or "hard" program and measured comprehension with a multiple choice quiz. The program contained either mnemonic or nonmnemonic variable names. Newsted reported significant results for the difficulty of program, choice of variable name and their interaction. The nonmnemonic groups performed as well on both the "easy" and "hard" programs. The mnemonic groups performed better on the "hard" program than on the "easy" program. The nonmnemonic groups performed better than the mnemonic groups for both the "easy" and "hard" programs. The results of Newsted's experiment may have been biased since the programs contained a short paragraph defining the variables, thus possibly negating the initial advantage of the meaningful variable names.

We conducted an experiment similar to Newsted (McKay, 1974), but the programs did not contain any comments. Four different programs were used and ranked by experienced programmers as to difficulty. The programs were then given to novice programmers who were given a comprehension quiz after a period of study. The mnemonic groups performed better than the nonmnemonic groups for all four programs. For the most difficult program, the difference between the mnemonic and nonmnemonic groups was nearly significant. These results indicated mnemonic variable names facilitate comprehension as program complexity increases. For short simple programs, the choice of variable name was not critical.

In a recent experiment, intermediate level student programmers were subjects in an investigation of variable names and debugging. The experiment was administered as part of a course quiz.

Two algorithms, sequential search and recursive binary search, were coded in PASCAL using either mnemonic or nonmnemonic variable names. Subjects were randomly assigned to one of two groups. One group received a mnemonic sequential search program followed by a nonmnemonic recursive binary search program. A second group was provided with a nonmnemonic version of the sequential search followed by a mnemonic recursive binary search program. Subjects were told each program contained one bug. Their task was to find the bug and correct it.

A method was devised to give partial credit. Full credit was given for locating and correcting the bug. Partial credit was awarded on the following basis: four-fifths credit for finding the bug but not correcting it, three-fifths credit for making two guesses as to which program statement was incorrect and one of them contained the bug, two-fifths credit for three guesses and locating the bug. No credit was given if the bug was not found.

The raw means appear in Table I. None of the main effects nor interactions were significant by an analysis of variance.

		Program	
		Sequential Search	Recursive Binary Search
Type of Variable	Mnemonic	12.9	12.9
	Nonmnemonic	12.6	10.0

Table I

Means for Variable Name Experiment

Since the subjects were familiar with the programs, both programs were previously discussed in class, the results are not too surprising. The familiarity of the algorithms resulted in the program being fairly "easy" to understand. From the results of the previous experiment, the performance of the mnemonic group versus the nonmnemonic group should not differ. The observed performance for the nonmnemonic recursive binary search appears to be different than the other groups. Since recursion was introduced shortly before the quiz, the recursive program may have been more difficult for the subjects. The more complex the program, the more mnemonic variable names aid comprehension.

While we have not clearly shown mnemonic variable names to aid in a debugging task, the results of this experiment display a trend in that direction. A further experiment is needed to discern the importance of presentation order of the mnemonic or nonmnemonic program and to validate the use of mnemonic variable names in debugging.

Indentation and Debugging

When writing programs in a block oriented programming language such as ALGOL, PL/I or PASCAL, programmers usually adopt some indenting scheme to present the code in a more readable form. Some programmers even use an indenting scheme in languages such as FORTRAN or COBOL.

Another recent experiment similar to the previous variable name and debugging experiment was conducted as part of the same quiz. Subjects were assigned to one of two groups. One group received an indented form of an iterative binary search program followed by a nonindented merge

2. Boehm, B.W., The High Cost of Software. Practical Strategies for developing Large Software Systems, Horowitz, E. (ed.), Addison-Wesley, 1975.

3. Boies, S.J., and Gould, J.D., A Behavioral Analysis of Programming: On the Frequency of Syntactical Errors. IBM Thomas J. Watson Research Center, Yorktown Heights, New York, RC 3907, 1972, 1-18.

4. Brooks, F.P., The Mythical Man-month, Essays on Software Engineering. Addison-Wesley, Reading, Ma., 1975.

5. Gannon, J.D., and Horning, J.J. The Impact of Language Design on the Production of Reliable Software. IEEE-TSE, Vol. 1, No. 2, 1975.

6. Geller, D.P., Debugging other Languages in APL. Software Practice and Experience, Vol. 5, No. 2, 1975.

7. Gould, J.D, Some Psychological Evidence on How People Debug Computer Programs. International Journal of Man-Machine Studies, Vol. 7, No. 2, 1975.

8. Gould, J.D., and Drongowski, P., An Exploratory Study of Computer Program Debugging. Human Factors, Vol. 16, No. 3, 1974.

9. Kinicki, R., and Ramsey, M., An Experiment in Programming Methodology. Unpublished paper.

10. Litecky, C.R., and Davis, G.B., A Study of Errors, Error-proneness, and Error Diagnosis in COBOL. CACM, Vol. 19, No. 1, 1976.

11. Lucas, H.C., and Kaplan, R.B., A Structured Programming Experiment. Research Paper Series, Research Paper No. 196, Graduate School of Business, Stanford University, February, 1974.

12. McKay, D., Effects of Variable Names on Program Understandability. Unpublished paper.

13. Miller, L., Programming by Non-Programmers. IBM Research Report RC 4280, 1973.

14. Mills, H.D., How to Write Correct Programs and Know It. , Gaithersburg, Maryland, 1972.

15. Newsted, P.R., FORTRAN program comprehension as a function of documentation. School of Business Administration Report, University of Wisconsin, Milwaukee, Wisconsin.

16. Shneiderman, B., Experimental Testing in Programming Languages, Stylistic Considerations and Design Techniques, Proc. National Computer Conference, AFIPS Press, Montuale, NJ, 1975, 653-565.

17. Shneiderman, B., and Mayer, R., Towards a Cognitive Model of Programmer Behavior. Indiana University Computer Science Department, Technical Report No. 37, Bloomington, Indiana, 1975.

18. Shneiderman, B., Mayer, R., McKay, D., and Heller, P., Experimental Investigations of the Utility of Flowcharts in Programming. Indiana University Computer Science Department, Technical Report No. 36, Bloomington, Indiana, 1975.

19. Sime, M.E., Green, T.R.G., and Guest, D.J., Psychological Evaluation of Two Conditional Constructions Used in Computer Languages. International Journal of Man-Machine Studies, Vol. 5, No. 1, 1973.

20. Wirth, N., Program Development by Stepwise Refinement. CACM, Vol. 14, No. 4, 1971.

21. Yasukawa, K., The Effect of Comments on Program Understandability and Error Correction. Unpublished paper.

22. Youngs, E.A., Human Factors in Programming, International Journal of Man-Machine Studies, Vol. 6, No. 3, 1974.

program. The other group was provided with a nonindented binary search program followed by an indented form of the merge program. Both programs were written in PASCAL and contained one bug. Again, the task was to locate and repair the bug. The same grading scheme was used as in the previous experiment.

Table II displays the raw means. While none of the main effects were significant, the interaction of mnemonic versus non-mnemonic groups was significant ($F=11.9$, $df=1/24$, $p<.01$). The means for the binary search are comparable to the means obtained in the previous experiment (Table I). Again, since the binary search program was discussed in class, performance on that part of the quiz should have been good.

		Program	
		Binary Search	Merge
Form of Program	Indented	13.6	9.2
	Nonindented	13.5	7.6

Table II

Means for Indentation Experiment

The merge program was not presented in class. Performance for both the indented and nonindented versions of the programs indicate the merge algorithm was more difficult than the binary search program. The performance for both versions of the merge program suggests that as program complexity increases, program comprehension is aided by an indentation scheme. Again, we have not satisfactorily investigated the effect of presentation order but speculate that as program complexity increases, program comprehension is aided by an indentation scheme.

Conclusions

Our experience in conducting psychological experiments on programmers has led us to insights into the programming process and to a better understanding of the experimental methodology that should be applied. We feel that we have found support for the cognitive model of programmer behavior, proposed by Shneiderman and Mayer (1975), which suggests that semantic knowledge about programming is heirarchically structured. The effectiveness of high level organizational comments and functionally organized modules support this model.

Comments and mnemonic variable names are of increasing value as the complexity of a program increases. Comments which reiterate the function of a single line or flowcharts which are straightforward representations of the program do not help and may even hinder debugging and modification. Low level comments and detailed flowcharts may interfere with the ability of the programmer to concentrate on reading the actual source code.

The critical importance of previous experience and the high variability of performance among subjects were constantly brought forward in our experimental results. Subjects could deal with familiar algorithms far more easily than with novel algorithms. This suggests that programmers in large programming shops should be allowed to work on the same type of problems rather than be arbitrarily shifted to new projects.

Our results reinforce previous findings of the wide range of programming ability. Managers should recognize that programming is a high level skill and that proficient programmers should be appreciated and properly rewarded. This finding supports Frederick Brooks, Jr., the author of The Mythical Man-Month, who recognized that the "man-month" concept is not applicable to programmers, since it is not possible to ensure that a month's work of two different programmers would yield the same results. Our experiments confirm the wide variance in performance and ability, which must be taken in account in professional environments and in the design of experiments.

Other methodological insights include the recognition of the range of skills that are required in programming tasks. Even the basic task of program comprehension has multiple dimensions: low-level execution details, comprehension of input/output relationships, high level conceptual understanding, etc. Experimental designs must take a cross-section of these comprehension levels if the results are to be reliable, replicable and generalizable. The complex tasks of program debugging and modification are difficult to deal with experimentally and care must be taken to avoid inadvertent biasing of the experiment. The choice of the program, the bug and the modification are extremely delicate matters. A final methodological point: although initially skeptical of the usefulness of subjective measures, we now support the collection of this information.

Much work remains to be done and the benefits of research in this area are attractive. We have the opportunity to significantly improve productivity and quality in the vital field of software development, and to investigate complex human problem solving behavior.

REFERENCES

1. Boehm, B.W., Software and its impact: A quantitative assessment. Datamation, May, 1973, 48-59.