Experimental testing in programming languages, stylistic considerations and design techniques

by BEN SHNEIDERMAN
Indiana University
Bloomington, Indiana

BACKGROUND

In the early stages of the development of high-level languages, radically differing alternatives were often promulgated. Each language had a dedicated corps of adherents who advocated the primacy of their facility. Turbulent debates among the protagonists were a common affair at conferences and in the trade journals. Now as the field matures, the vehement discussions have subsided and there is a widespread recognition of the usefulness of a variety of languages. Even the proponents of a single universal language have softened their tone and have accepted the multiple language condition.

New proposals for algorithmic languages offer only slight variations, and much effort has been devoted to standardization. Simultaneously, there has been a proliferation of modest language extensions producing a conglomeration of dialects of the accepted standard. Widely varying languages are still developing, but primarily for specific problem domains, such as data description, data manipulation, or artificial intelligence research.

As the issues become more subtle, it is no longer acceptable for developers and implementors to make highly subjective and personalized statements concerning the worthiness of a particular language feature or stylistic technique. The rampant proliferation of new dialects or entirely new languages is counterproductive since it limits the sharing of programs. A concerted effort must be made to ensure that new features, dialects, languages, and techniques are truly improvements. Control cards for operating systems and utility programs could also be improved by proper experiments with users.

As computer utilization becomes more widespread, large numbers of programming amateurs in diverse disciplines will demand facilities which are simple to use. Professional parametric users, such as bank tellers and reservations clerks, with a minimum of training will also have to be accommodated. Since the background and orientation of these users is profoundly different from that of the programming language designer, experimental techniques must be devised to guide the designer to the optimum language specification.

Although Dijkstra explicitly stated that computer programming was primarily a *human* activity as early as 1965,[1] it was not until the publication, in 1971, of Gerald Weinberg's text *The Psychology of Computer Programming*[2] that this notion was widely recognized. This stimulating and insightful work set the stage for research into the human factors in programming. Weinberg's text concentrates on defining the programming task in the context of the professional environment and promotes the notion of "egoless programming teams." This team organization concept may be contrasted with the "chief programmer team" strategy advocated by IBM.[3] Experimental comparison of interactions in these personal organization strategies would be an intriguing task for social psychologists.

Other sections of Weinberg's book concentrate on individual personality factors, training, and motivational factors. Although the conference reports of the ACM Special Interest Group on Computer Personnel Research describe initial steps, much more research needs to be done on the psychological make-up of programmers. Fortunately, psychologists have begun to study programming behavior as an aspect of problem solving.[4]

Training and teaching of programming has long been of interest to academically oriented researchers, as witnessed by the papers presented at the annual conferences of the ACM Special Interest Group on Computer Science Education. Programming has only recently become a subject for related disciplines such as educational psychology.[5]

Although experimentation in all of the above mentioned areas would undoubtedly be welcome, the focus of this paper is on experiments in programming language features, stylistic considerations and design techniques.

Research in programming language design

The volume of written material on programming language design is enormous. Thorough comparative surveys can be found in the work of Sammet,[6] Higman[7] or Elson[8]. Detailed remarks by the designers occur in the classic

reports on languages such as FORTRAN, COBOL, ALGOL 60, ALGOL 68, or PASCAL. Standardization reports on FORTRAN and COBOL also provide certain insights. Recently the overall topic of programming language design has been discussed by Hoare,[9] Wirth[10] and Carlson.[11] All of these works are based on the observations of individuals or small groups. The controversy over structured programming is closely related to program design issues. Although much has been written on this topic, the experimental evidence for eliminating the GOTO statement in favor of the three Bohm and Jacopini[12] structures has never been collected or reported. While individual experiences are useful,[13,14] they do not provide meaningful results to make judgments for the entire community of programmers.

Two groups of researchers have recognized the usefulness of studying non-programmers in the hopes of developing languages more closely conforming to "natural" thought processes. Sime, Green, and Guest[15] describe a fascinating experiment on non-programmers to determine which of two conditional statements these subjects found easier to use. Motivated by psycho-linguistic considerations, they attempted to compare the ease of use of the IF-THEN-ELSE construction and the IF(CONDITION)GOTO statement. Their result, based on a relatively small sample size in a carefully controlled, but artificial programming environment, was that the IF-THEN-ELSE construction was easier to use and resulted in fewer bugs, particularly with more complex problems. The paper does not mention structured programming, but the results are an initial confirmation of the concept.

Miller[16] also tested non-programmers using a highly simplified, specially constructed programming language. His subjects constructed programs for a series of simple card-sorting problems (the cards had a single name printed on them) containing conjunctive ("and") and inclusive disjunctive ("or") conditions expressed in the negative or affirmative. An example of the inclusive disjunctive form where one clause was in the affirmative and one was in the negative, was the problem of writing a program to "put a card in box three if either the name's second letter is *not* 'L' or if its last letter is 'N' . . . count the number of cards in box three using counter 1. Put the remaining cards in box 2." The results indicate that it was more difficult for the subjects to deal with the inclusive disjunctive than the conjunctive and that a negative clause made the problem still more challenging. The difficulty was measured by the time used and by the number of errors, both of which were significantly higher for "or" problems. Although this experiment does not directly shed light on language design issues, the experimental methodology is useful, and it suggests that further work with non-programmers would be very useful.

A third group[17] is comparing two proposed data retrieval sublanguages by testing programmers and non-programmers. This is the first time that a thorough and carefully controlled experiment has been performed prior to implementation.

Research in stylistic considerations

A number of texts have focussed on issues of programming style.[18,19] Although these texts offer valid practical suggestions for reducing execution time or storage utilization, they can only proffer subjective suggestions for stylistic decisions. Those stylistic issues include documentation standards, keypunching rules to increase clarity, guidelines for the selection of variable names, suggestions about programming techniques to increase the readability of programs and principles of program design.

Newsted[20] has elevated the discussion by conducting an experiment to determine the influence of comment cards in FORTRAN programs. His results indicate that on short FORTRAN programs (less than 30 lines) comments and mnemonic names may actually interfere with attempts to understand programs.

Weissman[21-23] has carried out a number of interesting experiments, concentrating on stylistic issues such as commenting, meaningful variable name selection, indentation, choice of flow of control techniques and subroutine use. Unfortunately, none of these experiments resulted in clear-cut recommendations for programmers. Weissman focuses heavily on issues of experimental design and techniques for measuring comprehension.

Research in design techniques

Some of the most provocative current debates are on the topic of program design methodologies such as modularity, step-wise refinement, and top-down design, each of which is often coupled to the structured programming concept. Although Baker[24] has reported on remarkable successes in a single project and Weissman conducted a single inconclusive experiment, no steps have been taken to confirm or refute these proposed design techniques. Personal testimonials do not suffice; replicable experimental results from a wide range of subjects are necessary.

Even the fundamental technique of flowcharting can be controversial. While some programmers reject the usefulness of flowcharting, others find it essential in planning and documenting large programs. A pilot experiment has demonstrated that for short programs, preliminary flowcharting does not simplify the programming task. A new flowcharting technique for structured programming[25] is gaining acceptance, but it has not yet been experimentally validated.

EXPERIMENTAL PARADIGMS

The fundamental difficulty in this research area is that it is so broad and so ill defined. Basic research into the way people think about programs would serve as a stepping stone to more precise experiments to resolve

particular issues. But first, the underlying experimental methodologies must be developed and verified.

Problem domains

There are at least five highly interwoven tasks which unify the questions of programming language design, stylistic considerations and design techniques. In each case the goal is to facilitate the interrelated tasks of:

- learning
- program understanding
- program writing
- debugging
- modification

Preferred improvements would impact positively for each of these tasks, but it is conceivable that an improvement in one area would hinder another. For example, a complex indentation or keypunching rule might make the program easier to read and understand, but more difficult to write. While modularity may simplify debugging and modification, errors might be committed in passing parameters when the program is composed. Finally, a powerful but complex and difficult to learn concept may ease the burden of program writing.

Complications result from the fact that a technique which is beneficial in long programs may be a hindrance in short programs. Different stylistic and design rules seem appropriate for programs of differing lengths.

Another variable that must be explored is programmer ability. Useful principles for professional programmers may be too complex for novices. Preliminary results from several studies indicate that techniques appropriate for novice programmers working on short simple programs are substantially different from the techniques applied by professionals on large difficult projects.

In summary, proposed improvements must be evaluated with respect to the tasks of learning, program understanding, writing, debugging, and modification, while simultaneously considering the spectrum of programmer abilities and program complexity.

Experimental techniques

Developing suitable experimental techniques is a nontrivial task. Care must be taken to minimize the number of variables, and proper experimental controls must be established. A sufficiently large group of subjects must be secured and pre-tests or other measures obtained to ensure the homogeneity of the subjects. Replicable, objective tests must be constructed and validated. Finally, accepted statistical techniques should be applied to the data to produce results which would be acknowledged by other researchers.

Testing program understandability is the most straightforward of the tasks. After studying a program for a prescribed length of time, subjects are asked to describe in words the function of the program. Grading the responses can be difficult, but if more than one person scores each response, it should be possible to obtain reliable results. The subjects may be asked to introspect and respond as to the difficulty of the problem, say, on a scale ranging from one to ten. This simple experimental procedure may be used to compare two proposed language features or stylistic rules. Alternatively, the subjects may be told to study until they feel their comprehension is adequate, making time and correctness the measured variables.

The numerous other methods of measuring understanding include asking subjects to determine the output for a given set of inputs. Weissman utilized detailed hand simulations of execution and paragraph fill-in techniques as measures of comprehension. Simpler traces such as listing the line number executed for a given input may be useful. Two other measures are the correctness and time needed to make a specific modification or to locate a bug (which has been included in the program). Although subjective measures of difficulty have dubious value, they may be used to enhance each of the above techniques.

Finally, subjects may be asked to memorize the program. Since memorization is best accomplished by "chunking" of the program, to gain meaningful (as opposed to rote) learning, more detailed recollection suggests more thorough comprehension. This technique is described in the companion paper, entitled "Two Exploratory Experiments in Program Comprehension."[26]

To simplify grading, Newsted used multiple choice questions rather than fill-ins in testing the subject's ability to comprehend, trace execution, determine outputs, etc. Multiple choice questions ensure objective and clear-cut grading standards.

A number of recent studies have concentrated on errors in programming and in debugging techniques.[27-30] By including monitoring code in compilers researchers have been able to capture the diagnostic reports and determine which errors are most frequently made. Controlled experiments can be conducted by providing programmer subjects with listings containing one or more errors and studying how they attempt to locate the errors.

CONCLUSION

Experimental techniques are being developed to resolve the human factors issues in program development. These experimental techniques can be applied to objectively validate proposals for programming language features, stylistic guidelines, and design paradigms. Much work remains to be done to extend the scope of these experiments so that concrete recommendations can be made to professional programmers for improving the quality of their work and their productivity. Additional experimental results should enable instructors to provide better training programs.

REFERENCES

1. Dijkstra, E. W., "Programming Considered as a Human Activity," *Proc. IFIP Congress 1,* 1965, pp. 213-217.
2. Weinberg, G. M., *The Psychology of Computer Programming,* Van Nostrand Reinhold, New York, New York, 1971.
3. Baker, F. T., "Chief Programmer Teams," *IBM Systems Journal 11,* 1, 1972.
4. Mayer, R. E., *Instructional Variables in Meaningful Learning of Computer Programming.* Indiana Mathematical Psychology Program, Report No. 75-1, 1975.
5. Kreitzberg, C. and L. Swanson, "A Cognitive Model for Structuring an Introductory Programming Curriculum," *AFIPS Proc. National Computer Conference,* 1974.
6. Sammet, J. E., *Programming Languages: History and Fundamentals,* Prentice-Hall, Inc., Englewood Cliffs, N.J., 1969.
7. Higman, B., *A Comparative Study of Programming Languages,* American Elsevier Publishing Company, Inc., New York, 1967.
8. Elson, M., *Concepts of Programming Languages,* Science Research Associates, Inc., Chicago, Illinois, 1973.
9. Hoare, C. A. R., *Hints on Programming Language Design,* invited address at SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Boston, Mass., October 1-3, 1973.
10. Wirth, N., *On Certain Basic Concepts of Programming Languages,* Technical Report No. CS 65, Computer Science Department, Stanford University, Stanford, California, May 1, 1967.
11. Carlson, C. R., *Programming Language Design,* Computer Sciences Department, The Technological Institute, Northwestern University, Evanston, Illinois, 1973.
12. Bohm, C. and G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Comm. ACM 9,* May, 1966, pp. 366-371.
13. Henderson, P. and R. Snowdon, "An Experiment in Structured Programming, *BIT 12,* 1972, pp. 38-53.
14. Standish, T. A., *Observations and Hypotheses About Program Synthesis Mechanisms.* Automatic Programming Memo 9, Report No. 2780, Computer Science Division, Bolt Beranek and Newman, Cambridge, Mass., December 19, 1973.
15. Sime, M., T. Green, and D. Guest, "Psychological Evaluation of Two Conditional Constructions Used in Computer Languages," *International Journal of Man-Machine Studies,* Vol. 5, No. 1, 1973.
16. Miller, L., *Programming by Non-Programmers,* IBM Research Report RC 4280, 1973.
17. Reisner, P., R. F. Boyce, and D. D. Chamberlin, "Human Factors Evaluation of Two Data Base Query Languages: SQUARE and SEQUEL, *Proc. National Computer Conference,* AFIPS Press, Montvale, New Jersey, 1975.
18. Kreitzberg, C. B. and B. Shneiderman, *The Elements of FORTRAN Style: Techniques for Effective Programming,* Harcourt Brace Jovanovich, Inc., New York, New York, 1972.
19. Van Tassel, D. *Program Style, Design, Efficiency, Debugging, Testing,* Prentice-Hall, Englewood Cliffs, New Jersey, June, 1974.
20. Newsted, P. R., *FORTRAN Program Comprehension as a Function of Documentation,* School of Business Administration Report, The University of Wisconsin, Milwaukee, Wisconsin.
21. Weissman, L., *Psychological Complexity of Computer Programs: An Initial Experiment,* Technical Report CSRG-26, Computer Systems Research Group, University of Toronto, Toronto, Canada, 1973.
22. ——, "Psychological Complexity of Computer Programs: An Experimental Methodology," *SIGPLAN Notices 9,* 6, June, 1974.
23. ——, A *Methodology for Studying the Psychological Complexity of Computer Programs,* Ph.D. thesis, University of Toronto, 1974 (available as Technical Report, Computer Science Research Group CSRG-37).
24. Baker, F. T., "System Quality Through Structured Programming," *Proc. FJCC,* 1972, pp. 339-343.
25. Nassi, I. and B. Shneiderman, "Flowchart Techniques for Structured Programming, *SIGPLAN Notices 8,* 8, August, 1973, pp. 12-26.
26. Shneiderman, B. and Mao-Hsian Ho, *Two Exploratory Experiments in Program Comprehension,* Technical Report No. 27, Computer Science Department, Indiana University, 1974.
27. Gould, J. D. *Some Psychological Evidence on How People Debug Computer Programs,* IBM Research Report RC 4542, 1973.
28. Boies, S. J. and J. D. Gould, "Syntactic Errors in Computer Programming, *Human Factors 16,* 3, 1974, pp. 253-257.
29. Young, E. A., "Human Errors in Programming," *International Journal of Man-Machine Studies 6,* 1974, pp.361-376.
30. Gould, J. D. and P. Drongowski, "An Exploratory Study of Computer Program Debugging," *Human Factors 16,* 3, 1974, pp. 258-277.