On the use of psychology in evaluating software documentation

DEBORAH A. BOEHM-DAVIS

General Electric Company, Arlington, Virginia 22202

Recent research evaluating the usefulness of program documentation has borrowed techniques used in psychology to define the variables of interest and to measure those variables. The results of the research suggest that (1) using a branching or sequential program design language (PDL) produces performance that is superior to that found with other formats (such as flowcharts), and (2) using a detailed design written in a PDL leads to superior coding performance over other formats due, in part, to the small amount of translation that is required in mapping from the design to the code. These conclusions are based primarily on time and accuracy data. This paper discusses the research conducted in this area and the conclusions drawn. The influence of the methodology used to conduct these experiments on the evaluation of programmer performance is also discussed.

During the 25 years from the development of the first computer until about 1970, research in the computer field focused largely on the development of better hardware and on programming methodologies to use hardware more efficiently. Little thought was given to the investigation of human skills and capacities for using computers; however, the emphasis is now shifting to studies of the user. This trend in research is reflected in the proportion of money spent on hardware vs. software over the last 25 years. In 1955, 85% of the total money spent in the computer industry was associated with hardware costs. By 1980, that figure had been reduced to only 15% (Boehm, 1981), with the other 85% representing the cost of software. While part of this increase in the amount of money spent on software is due to an increased demand for software, a large part of the increase is due to the continued use of antiquated methods and tools for developing software. This problem was foreseen by Sackman, Erikson, and Grant as long ago as 1968 (p. 3): "Amid all [the] portents of the dominating role that computer programming will play in the emerging computer scene, one would expect that computer programming would be the object of intensive applied scientific study. This is not the case. There is, in fact, an applied scientific *lag* in the study of computer programmers and computer programming—a widening and critical lag that threatens the industry and

the profession with the great waste that inevitably accompanies the absence of systematic and established methods and findings and their substitution by anecdotal opinion, vested interests, and provincialism." While the situation is somewhat better today, the amount of research is still not keeping pace with the growth of the computer industry.

In the 1960s, the involvement of human-factors specialists in computer systems was largely limited to the traditional operations research role of designing the keyboards for computer terminals. Some comparisons of programming languages and batch vs. timesharing systems were reported by Sackman (1970), but serious empirical research in other areas did not appear until after 1970. Weinberg's (1971) *The Psychology of Computer Programming* emphasized the importance of controlled experimentation in evaluating performance in computer-related tasks. Martin's (1972) *Design of Man-Computer Dialogues* was the first book to elevate the human-computer interface to the level of an independent area of study (Shneiderman, 1980).

At present, scientists from a wide range of disciplines are interested in the behavioral aspects of users' interactions with computers. Mathematicians, computer scientists, information systems specialists, graphics designers, linguists, and educational, cognitive, and experimental psychologists are all considering methods to improve computer systems from a human-factors point of view. Research in this area, now being referred to as software psychology, is emerging as a separate discipline that combines the expertise of scientists with a variety of backgrounds.

Specifically, software psychology is the study of the performance of people using computer systems. Research focusing on users' cognitive skills, decision styles, and modes of processing information is used to develop improved management strategies, programming aids, and

The author would like to extend many thanks to Dave Morris and Pete McEvoy, who designed and wrote the program for the original experimental system, to Bryan Wolfe, who modified the system for some of the subsequent studies, and to Betsy Kruesi and Sylvia Sheppard for helpful comments on earlier drafts of this report. This research program was supported by the Office of Naval Research, Engineering Psychology Group (Contract N00014-79-C-0595). The views expressed in this paper are not necessarily those of the Office of Naval Research or the Department of Defense.

coding techniques. Research in this area also involves the evaluation of various software development technologies and tools to determine which tool, or combination of tools, will provide the highest level of productivity on a given project. Some of the constructs that researchers are trying to measure in this area include, for example, programmer performance, program comprehensibility, and program maintainability.

This paper describes an application of psychological research methods to a research program that examined one of these constructs, the usefulness of different forms of program documentation in representing program information. The paper describes the research, the operational definitions used, and the influence of these research techniques and definitions on the conclusions drawn regarding software documentation.

THE RESEARCH

Documentation is an especially important part of the software life cycle, since major tasks in the cycle such as design, coding, testing, and maintenance are frequently performed by different individuals. Lientz and Swanson (Note 1), for example, found that, typically, only about half of a software system's maintenance personnel had been involved in its development. Communication among these personnel is no better than the documentation they develop. Poor documentation techniques can dramatically increase labor costs throughout the labor-intensive software life cycle by making both development and maintenance tasks more difficult. Design information, in particular, must be communicated effectively so that the integrity of a system will not be compromised when modifications are implemented.

Early work on program documentation focused on comparisons between different forms of documenting programs. Shneiderman, Mayer, McKay, and Heller (1977) compared the performance of students using a program listing alone or a program listing plus a flowchart in a series of experiments using composition, comprehension, debugging, and modification tasks. In 1978, Ramsey, Atwood, and Van Doren (Note 2) compared the effectiveness of flowcharts to that of a program design language (PDL). While these studies produced useful and interesting information, they were able to provide only specific answers for a limited set of documentation formats, not general answers about different forms of documentation.

The research program undertaken at General Electric (Boehm-Davis, Sheppard, & Bailey, 1982; Sheppard, Kruesi, & Bailey, 1982; Sheppard, Kruesi, & Curtis, 1981) tried to answer this more general question by using an experimental technique often used in psychology to define a general set of stimulus materials. The use of this technique, a factorial design, led to the useful distinction between two dimensions, the symbology with which information is represented and the spatial arrangement of that information. These two dimensions

were chosen because they are the primary dimensions for categorizing the way in which available documentation aids configure the information they present to programmers (Jones, 1979).

Using this design, Sheppard et al. (1981) were able to construct and examine nine different documentation formats from the factorial combination of three types of symbology with three types of spatial arrangement. The three types of symbology in which information was presented consisted of normal English, PDL, and ideograms. PDL was originally designed as a method to produce a program design in a structured manner (Caine & Gordon, 1975). It uses the vocabulary of one language (i.e., English) and the syntax of another (i.e., a structured programming language). Ideograms are symbolic representations of information originally created for use in flowcharts. The spatial arrangements of the information used in these experiments were sequential, branching, and hierarchical. This factorial design allowed Sheppard et al. to examine the separate contributions of the type of symbology and the spatial arrangement of the information as well as their interaction. It was hoped that general principles of information presentation would emerge from this approach.

The primary purpose of the research program was to investigate the effectiveness with which design information was transmitted by these different forms of software documentation. This was done by giving the participants a computer program and examining their performance in carrying out one of a number of software tasks. In the first experiment, participants were asked to answer a number of questions about a program. In the second, they were asked to code a portion of an incomplete program. In the third, they were asked to debug a program. And in the fourth, they were asked to modify a working program. In each of these experiments, the participants were provided a listing of the program, a data dictionary that defined the variables used in the program, and one of the documentation formats. The question being addressed was the usefulness of the various forms of documentation in completing the assigned task.

While the experiments in this series produced slightly different results depending on the type of experimental task, there was a sufficient degree of consistency to characterize the overall effects of symbology and spatial arrangement. Across the three experiments, the more succinct symbologies, the PDL and ideograms, led to better performance than the less succinct symbologies. Overall, the PDL symbology was associated with the best performance. In terms of spatial arrangement, the branching spatial arrangement was associated with shorter performance times and fewer errors than were the other arrangements. This was especially clear in the coding and modification tasks, in which successful performance depended on understanding the flow of control in the program. However, the sequential PDL was also an excellent version, and the hierarchical

ideograms were surprisingly usable in view of the participants' previous lack of experience with hierarchical versions of documentation (Sheppard et al., 1982).

This raised a question as to why these forms of documentation were superior to the other formats tested. One possible explanation of this superiority was that the superior formats were the most code-like of the documentation formats tested. As a result, there was less translation required in mapping between the documentation and the code, especially since the participants were coding in FORTRAN and using FORTRAN-like documentation.

This explanation is most reasonable for the coding task. In this experiment, participants were provided a program from which several lines had been deleted. The task was to complete the code so that the program would execute correctly. As such, the documentation format that was closest in form to the code should provide the most useful information, even though all the formats provided the same amount of information.

If these results were largely a function of the amount of translation required of the participant, they suggest that no single form of documentation would be optimal for all coding languages. Rather, the optimal documentation format would be one that is tailored toward the particular coding language.

A fifth experiment examined this hypothesis by creating and evaluating several different PDLs, each tailored to a different coding language. Sequential PDLs were chosen as the materials for this study because they led to high levels of performance in the first experiments and because our professional programmers were already familiar with them. In this final experiment, the results showed that performance for those programmers coding in FORTRAN was best when they used a FORTRAN-like PDL, whereas performance for those programmers coding in MACRO-11 (the assembly language for the PDP-11) was best when they used a MACRO-11-like PDL. This suggests that coding from a documentation format will take less time when the documentation is tailored to that coding language. Further, it suggests that the results of the earlier studies were probably due, at least in part, to the reduction in the amount of translation required by the documentation formats that led to superior performance. While it is apparent why they might be true for the coding task, it is not clear why these formats prove as useful in the comprehension, debugging, and modification tasks. Further research investigating the kind of information carried by the documentation formats is being conducted to shed more light on this issue.

EVALUATING SOFTWARE DOCUMENTATION

The research program described here represents a first step in impacting the industry's use and acceptance of experimental results in choosing among various software documentation formats. Early work investigating software documentation focused on direct compar-

isons of specific documentation formats. Sheppard et al.'s (1981) analysis of formats as falling along the levels of two dimensions, symbology and spatial arrangement, was a novel and important contribution to the field. Without this distinction, it may not have been realized that the way in which information is represented and the arrangement of that information contribute independently to the ability of a documentation format to accurately and easily transmit design information.

A major obstacle faced by Sheppard et al. (1981) was the choice and measurement of the important variables. Given the lack of systematic research in the area, it was not clear which measures would provide the clearest account of programmer performance. As a result, Sheppard et al. chose to collect a large amount of information so that a number of measures could be derived from the data set and the utility of each type of measure could be assessed. The interactive data collection system that resulted from this approach collected two primary types of data; these were time and protocol data.

In these experiments, Sheppard et al. (1981) felt that several of the time measures that the experimental system recorded would provide interesting measures of performance. In addition to a total time measure, the time between the start of a new program and the start of the first editor command was used as a measure of study time, and the times to perform various editing tasks were used as a measure of working or response time. In the course of running the experiment, however, it became apparent that these measures were difficult to interpret at face value. While some participants did study the program at some length before entering their first editor command, others performed an immediate listing and studied the program on-line. These actions invalidated the first time entry as a study time measure. Likewise, while some participants completely formulated their editor entries on paper before starting the entry, others began a command and formulated the input to that command on-line. Since the more detailed time data could not be interpreted without knowledge of the participants' strategies, the most usable performance measure to emerge from the time data collected was the total time required to study and edit each program.

The experimental system also created a data file for each system user. All of the entries made while the user was in the editor were stored in this file along with the time taken for each entry and a notation of whether or not the entry was a valid editor entry. It was originally felt that this information would provide some insight into the processes being performed. In practice, it became clear that which commands were used and the times to form these commands were subject to great variation, both between and within users, and that these data did not provide much useful information.

A listing of each version of the program after the system attempted a compilation was also recorded to provide a record of any compilation errors. If the

program compiled successfully, the program was run and its output was compared with the correct output. The result of this comparison (which was displayed to the user) was also recorded in the protocol. These data provided a complete account of the experimental sessions and allowed the experimenter to determine the number and kinds of errors made.

In the debugging and modification experiments, Sheppard et al. (1981) found that a large portion of the errors made while the user was in the editor were due to typing errors and incorrect use of the editor commands, and that few other errors were made; these data were not analyzed further. However, a sufficient number of errors remained in the program at the time of compilation in the two coding experiments to provide some insight into the difficulties experienced by the programmers. These measures were especially useful when they varied as a function of experimental condition, since they could be used to explain observed time differences. The kinds of errors were also important, since they provided insight concerning the particular nature of the difficulties experienced by the programmers. This analysis was particularly useful in the first coding study, in which it was found that the formats that were associated with fast performance also led to a reduction in errors, particularly in coding the control flow of the program. Unfortunately, in the second coding experiment, the number of errors was quite low and the errors were not as helpful in identifying specific areas of difficulty across programmers. As a result, the number of errors made was the most useful measure derived from the protocol data across all of the experiments.

In the final analysis, then, the two most useful measures derived from the data collection were the speed and accuracy data. These measures actually formed a very useful operational definition of programmer performance, since the generalization from these studies will be to the speed and accuracy of programmers performing similar tasks in the real world.

Another obstacle in doing this research was the problem of the generalizability of the results. This problem was only partially solved. The use of a highly controlled laboratory situation with narrowly defined tasks means that generalizations must be made very cautiously. The problem is often overcome in psychological work by running a series, or several series, of experiments, with each experiment attacking the problem from a different perspective. The current research program was able to investigate a number of different software tasks and, hence, followed this strategy in the choice of tasks. However, some interesting variables, such as the number of programmers working on a project, have not been investigated fully yet.

In general, then, the research program described here has provided a careful and systematic investigation of the ability of different forms of software documentation to transmit design information, and several general conclusions can be drawn from the research.

The major conclusion is that both symbology and spatial arrangement are independent and important determinants of programming performance. Specifically, the data suggest that the symbology should be succinct; in particular, the PDL symbology has been found to be very effective. While the issue of spatial arrangement is less clear, it appears that overall, the branching and sequential forms of arrangement provide for the best performance. The data from the last experiment suggest that these forms of documentation are useful, at least for coding tasks, because they reduce the amount of translation required in going from the documentation to the code.

These conclusions seem to suggest that programmers should document their code with branching or sequential PDLs. Further, they suggest that programmers who want to code from a PDL should use a language-tailored PDL rather than a language-independent PDL. However, a few cautions are in order. The research was conducted with individual programmers working on small programs, and we do not yet have enough evidence to suggest that the same effects will hold for teams of programmers working on large-scale projects. In addition, the trade-offs between decreased maintenance time and accuracy vs. design time and accuracy need to be considered before PDLs are recommended without reservations. Our experience in designing solutions using language-tailored PDLs suggests that designing the solution to a problem requires more time when the PDL is language tailored than when it is language independent. This limitation needs to be considered when choosing a design and documentation format. In addition, certain problems are better suited to different types of PDLs. For example, mathematically oriented problems, such as calculating a correlation coefficient, were easy to code in the APL-like PDL, whereas other types of problems were difficult to code in this PDL due to the mathematical structure of APL. This suggests that attention must be paid to the problem structure as well as to the target coding language.

Further research needs to be done. However, more research is not the entire answer. While research programs of the sort described here can provide a solid foundation for applied problems, more naturalistic techniques, which allow for a consideration of external validity and industry concerns, are needed. As psychologists and methodologists, it appears to be incumbent upon us to take some time away from our research on small domains to develop new methodologies and techniques of measurement. Only when these techniques and measures have been developed will we be able to answer all of the questions that industry is asking of us.

REFERENCE NOTES

1. Lientz, B. P., & Swanson, E. G. *The use of productivity aids in system development and maintenance* (Tech. Rep. 79-1). Los Angeles: UCLA, Graduate School of Management, 1979.
2. Ramsey, H. R., Atwood, M. E., & Van Doren, J. R. A

comparative study of flowcharts and program design languages for the detailed procedural specification of computer programs. (Tech. Rep. SAI-78-078-DEN). Denver: Science Applications, Inc., 1978.

REFERENCES

- BOEHM, B. W. *Software engineering economics*. Englewood Cliffs, N.J.: Prentice-Hall, 1981.
- BOEHM-DAVIS, D. A., SHEPPARD, S. B., & BAILEY, J. W. An empirical evaluation of language-tailored PDLs. In *Proceedings of the Human Factors Society 26th Annual Meeting*. Seattle, Wash: Human Factors Society, 1982.
- CAINE, S. H., & GORDON, E. K. PDL—A tool for software design. In *Proceedings of the 1975 National Computer Conference*. Montvale, N.J.: American Federation of Information Processing Societies, 1975.
- JONES, C. A survey of programming design and specification techniques. In *Proceedings of the IEEE Conference on Specifications of Reliable Software*. New York: IEEE, 1979.
- MARTIN, J. *Design of man-computer dialogues*. Englewood Cliffs, N.J.: Prentice-Hall, 1972.
- SACKMAN, H. *Man-computer problem solving*. Princeton, N.J.: Auerbach, 1970.
- SACKMAN, H., ERIKSON, W. J., & GRANT, E. E. Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM*, 1968, **11**, 3-11.
- SHEPPARD, S. B., KRUESI, E., & BAILEY, J. W. An empirical evaluation of software documentation. In *Proceedings of Human Factors in Computer Systems*. Gaithersburg, Md: ACM/NBS, 1982.
- SHEPPARD, S. B., KRUESI, E., & CURTIS, B. The effects of symbology and spatial arrangement on the comprehension of software specifications. In *Proceedings of the Fifth International Conference on Software Engineering*. San Diego, Calif: IEEE, 1981.
- SHNEIDERMAN, B. *Software psychology*. Cambridge, Mass: Winthrop, 1980.
- SHNEIDERMAN, B., MAYER, B. R., MCKAY, D., & HELLER, P. Experimental investigations on the utility of detailed flowcharts in programming. *Communications of the ACM*, 1977, **20**, 373-381.
- WEINBERG, G. M. *The psychology of computer programming*. New York: Van Nostrand Reinhold, 1971.