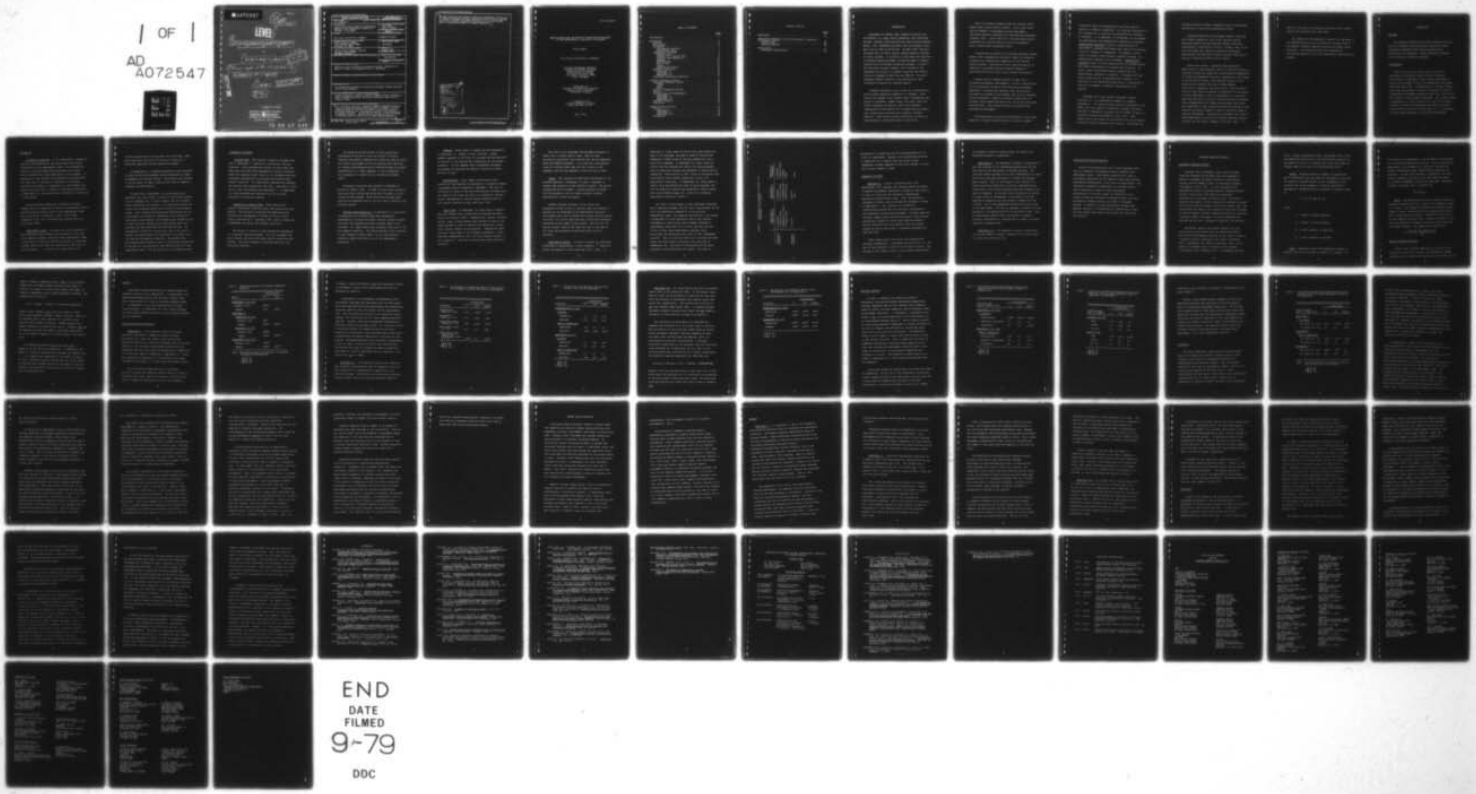
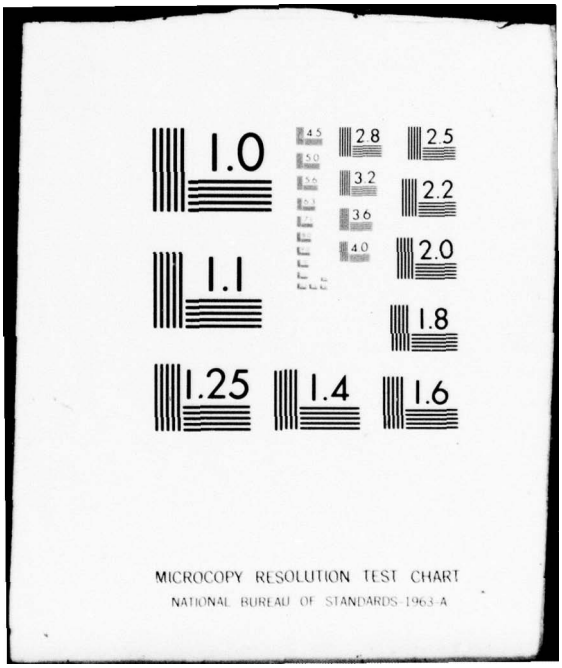
AD-A072 547 GENERAL ELECTRIC CO ARLINGTON VA F/G 9/2
IDENTIFICATION AND VALIDATION OF QUANTITATIVE MEASURES OF THE P--ETC(U)
APR 79 B CURTIS, S B SHEPPARD N00014-77-C-0158
UNCLASSIFIED TR-79-388100-7 NL

1 OF 1
AD A072547



END
DATE FILMED
9-79
DDC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(18) (14) TR-79-388100-7

LEVEL II

(6) IDENTIFICATION AND VALIDATION OF QUANTITATIVE MEASURES OF THE PSYCHOLOGICAL COMPLEXITY OF SOFTWARE

by

(10) Bill Curtis and Sylvia B. Sheppard

(9) Final rept. 1 Mar 77–28 Feb 79,

(15) N00014-77-C-0158

(11) 27 April 1979

(12) 71 p.

409446

SOFTWARE MANAGEMENT RESEARCH
GENERAL ELECTRIC
INFORMATION SYSTEMS PROGRAMS
ARLINGTON, VIRGINIA

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR-79-388100-7	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Identification and Validation of Quantitative Measures of the Psychological Complexity of Software: Final Report		5. TYPE OF REPORT & PERIOD COVERED Final Report 3/1/77 - 2/28/79 6. PERFORMING ORG. REPORT NUMBER 388100-7
7. AUTHOR(s) Bill Curtis and Sylvia B. Sheppard		8. CONTRACT OR GRANT NUMBER(s) N00014-77-C-0158
9. PERFORMING ORGANIZATION NAME AND ADDRESS General Electric Company 1755 Jefferson Davis Highway Arlington, VA 22202		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 197-037
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, Code 455 800 North Quincy Street Arlington, VA 22217		12. REPORT DATE 27 April 1979 13. NUMBER OF PAGES
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited. Reproduction in whole or in part for any purpose of the U.S. Government.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

This research was supported by Engineering Psychology Programs (Code 455), Office of Naval Research

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software engineering, Complexity metrics, Software science, Halstead's E, Modern programming practices, Structured programming, Human factors in computer science.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This final report describes a program of research conducted to identify factors affecting the psychological complexity of software. The work conducted pursuant to this task consisted of three within subjects, multifactor experiments. The methodology employed in these experiments is described in detail. Separate sections discuss the results obtained for software complexity metrics and for modern coding practices.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

The major results of this research program are experimental verifications of increased programmer efficiency when working with structured code and the ability to predict programmer performance with Halstead's effort metric (a quantitative measure of software complexity).

TR-79-388100-7

IDENTIFICATION AND VALIDATION OF QUANTITATIVE MEASURES OF THE PSYCHOLOGICAL COMPLEXITY OF SOFTWARE

Final Report

Bill Curtis and Sylvia B. Sheppard

Software Management Research
Information Systems Programs
General Electric Company
1755 Jefferson Davis Hwy
Arlington, Virginia 22202
(703) 979-6000

submitted to:
Engineering Psychology Programs
Office of Naval Research
Arlington, Virginia

in completion of:
Contract #N00014-77-C-0158
Work Unit #NR 197-037

April 1979

TABLE OF CONTENTS

Content (Cont'd)

INTRODUCTION

Department of Defense (DOD) software production and maintenance is a large, poorly understood, and inefficient process. Recently Frost and Sullivan (The Military Software Market, 1977) estimated the yearly cost for software within DOD to be as large as $9 billion. De Roze (1977) has also estimated that 115 major defense systems depend on software for their success. In an effort to find near-term solutions to software related problems, the DOD has begun to support research into the software production process. A formal 5 year R&D plan (Carlson & DeRoze, 1977) related to the management and control of computer resources was recently written in response to DOD Directive 5000.29. This plan requested research leading to the identification and validation of metrics for software quality.

Interest continues to grow in the use of quantitative metrics which assess the complexity of software. Such metrics are assumed to be valuable aids in determining the quality of software. Boehm, Brown, and Lipow (1976) and McCall, Richards, and Walters (1977) have proposed combinations of such metrics which assess numerous factors that collectively constitute this nebulous "software quality". Such factors include reliability, portability, maintainability, and myriad other xxx-abilities.

There are numerous potential uses for measures which assess these various quality factors. First, they can be used as feedback to programmers during development, indicating potential problems with code they have developed (Elshoff, 1978). Use of metrics in this way would require guidelines for altering code so as to bring different metric values within acceptable limits.

A second use for metrics is in guiding software testing. McCabe (1976) proposed the cyclomatic number as a means of assessing the computational complexity of the software testing problem. Other metrics which index the quality or complexity of software may help identify modules or subroutines which are likely to be the most error-prone.

Another use for software metrics is their use in estimating maintenance requirements. If one or more metrics can be empirically related to the difficulty programmers experience in working with software, then more accurate estimates can be made of the manpower that will be necessary during maintenance. Empirical validity studies will be necessary before employing metrics for any of the three uses described here. Such research should be conducted with professional programmers.

The experimental investigations described in this report comprised a research program seeking to provide valuable

information about the psychological and human resource aspects of computer programming. The challenge undertaken in this research was to quantify the psychological complexity of software. It is important to distinguish clearly between the psychological and computational complexity of software. Computational complexity refers to characteristics of algorithms or programs which make their proof of correctness difficult, lengthy, or impossible. For example, as the number of distinct paths through a program increases, the computational complexity also increases. Psychological complexity refers to those characteristics of software which make human understanding of software more difficult. No direct linear relationship between computational and psychological complexity is expected. A program with many control paths may not be psychologically complex. Any regularity to the branching process within a program may be used by a programmer to simplify understanding of the program.

Halstead (1977) has recently developed a theory concerned with the psychological aspects of computer programming. His theory provides objective estimates of the effort and time required to generate a program, the effort required to understand a program, and the number of bugs in a particular program (Fitzsimmons & Love, 1978). Some predictions of the theory are counterintuitive and contradict results of previous psychological research. The theory has

attracted attention because independent tests of hypotheses derived from it have proven amazingly accurate.

Although predictions of programmer behavior have been particularly impressive, much of the research testing Halstead's theory has been performed without sufficient experimental or statistical controls. Further, much of the data were based upon imprecise estimating techniques. Nevertheless, the available evidence has been sufficient to justify a rigorous evaluation of the theory.

Rather than conduct a research program designed specifically to test Halstead's theory of software science, a research strategy was chosen which would generate suggestions for improving programmer efficiency regardless of the success of any particular theory. This research focused on four phases of the software life-cycle: understanding, modification, debugging, and construction. Since different cognitive processes are assumed to predominate in each phase, no single experiment or set of experiments on a particular phase were believed to provide a sufficient basis for making broad recommendations for improving programmer efficiency. Each experiment in this research program was designed to test important variables assumed to affect a particular phase of software development. Professional programmers were used in these experiments to provide the greatest possible external validity for the results (Campbell & Stanley, 1966). In

addition, the theory of software science and other related metrics were evaluated with these data.

The results of this program of research are described in two separate sections, each considering a primary focus of the program: software complexity metrics and modern coding practices. A preliminary section will describe the methodology employed in the three experiments comprising this program.

METHODOLOGY

Overview

The research conducted to evaluate software complexity metrics and modern coding practices consisted of three experiments designed around program comprehension, modification and debugging. These experiments employed within subjects, multifactor designs.

Participants

Each of the first two experiments involved 36 programmers from several General Electric locations. Participants in Experiment I had a working knowledge of Fortran and averaged 6.8 years of professional programming experience (SD = 5.8). In Experiment II, the participants had a working knowledge of Fortran, averaged 5.9 years of professional programming experience (SD = 4.0), and had not participated in the previous experiment. The 54 participants in Experiment III (30 civilians and 24 from the military) averaged 6.6 years of programming experience in Fortran, ranging from 1/2 year to 25 years (SD = 6.1). The majority of participants possessed an engineering background.

Procedures

Introductory exercise. In all experiments, a packet of materials was prepared for each participant with written instructions on the experimental tasks. As a preliminary exercise, all participants were presented with a short Fortran program and a brief description of its purpose. In Experiment I, they studied this program for 10 minutes and were then given 15 minutes to reconstruct a functional equivalent from memory. In Experiment II, participants were allowed unlimited time to complete a specified modification. In Experiment III, participants were instructed to find a bug in the program.

The introductory program was intended to provide a commmon basis for comparing the skills of participants and to diminish learning effects prior to the experimental tasks. This latter point is important, since a pilot study (Sheppard & Love, 1977) indicated that learning may occur during such tasks.

Experimental tasks. Following the initial exercise in each experiment, participants were presented in turn with three separate programs comprising their experimental tasks. In Experiment I, they were allowed 25 minutes to study each program, during which they were permitted to make notes or draw flowcharts. At the end of the study period, the

original program and all scrap paper were collected. Each participant was then given 20 minutes to reconstruct a functional equivalent of the program from memory.

In Experiment II, a separate modification was indicated for each of the three programs and was described on a sheet accompanying the program listing. Participants were allowed to work at their own pace, taking as much time as needed to implement the modification.

On each task in Experiment III, participants were presented with input files, a listing of the Fortran program with the embedded bug, a correct output, and the erroneous output produced by the program. All differences between the correct and erroneous outputs were circled on the erroneous output. Also included were explanatory descriptions of any subroutines or functions not presented in the listing but referenced by the program. Participants were allowed to work at their own pace, signalling the experimenter when they believed they had identified and corrected the bug. The experimenter verified all corrections, and in the case of a mistake, the participants was instructed to try again until the task was successfully completed. The maximum time participants were allowed to work on a particular program was 45 minutes for the preliminary task and 60 minutes for each experimental task. Time was measured to the nearest minute.

Independent Variables

Program class. Three general classes of programs were used in Experiment I: engineering, statistical, and non-numerical. Three programs were chosen for each class from among many solicited from programmers at several locations. These nine programs varied from 36 to 57 statements and were considered representative of programs participants might actually encounter. All experimental programs were compiled and executed using appropriate test data. Experiment II used three of the nine programs from Experiment I. In Experiment III, three programs were selected which had not been employed in the previous studies.

Complexity of control flow. Three control flow versions, performing identical tasks, were defined for each program. The naturally structured and graph-structured versions were implemented in Fortran IV, while a third version used Fortran 77 (Brainerd, 1978), which includes the IF-THEN-ELSE, DO- WHILE, and DO-UNTIL constructs.

The Fortran 77 version of each program was implemented in a precisely structured manner. All flow proceeded from top to bottom, and only three basic control constructs were allowed: the linear sequence, structured selection, and structured iteration.

The graph-structured version of each program was implemented in Fortran IV from the Fortran 77 version, replacing the special constructs but producing code for which the control flow graphs of the two versions were identical. All nested relationships could be reduced through structured decomposition to a linear sequence of unit complexity. A full discussion of reducibility is presented by McCabe (1976).

Structured constructs were awkward to implement in Fortran IV (Tenny, 1974). In order to test a more naturally structured flow, limited deviations were allowed in a third version of each program. These deviations included such practices as branching into or out of a loop or decision, and multiple returns.

Variable name mnemonicity. In Experiment I, three levels of mnemonicity for variable names were manipulated independently of program structure. Several non-participants were shown the programs and asked to assign names to the variables. The names chosen most frequently were used in the most mnemonic condition. The medium mnemonic level consisted of less frequently chosen names. In the least mnemonic condition, names consisted of one or two alphanumeric characters.

Comments. Three levels of commenting were manipulated in Experiment II: global, in-line, and none. Global comments appeared at the front of a program and provided both an overview of its function and a definition of the primary variables. In-line comments were interspersed throughout the program and described the specific functions of small sections of code.

Modifications. Three types of modifications were selected for each program in Experiment II as typical changes a programmer might be expected to implement. The level of difficulty for seven of the nine modifications increased with the number of new lines that had to be inserted to achieve a correct implementation, and the hardest modifications for each program required the most additional lines.

Type of Bug. In Experiment III three types of semantic bugs were chosen from a classification developed by Hecht, Sturm, and Trattner (1978): computational, logical, and data errors. Bugs in each category were defined for each of the three programs in order to maximize the similarity of bugs from a single category across programs. Computational bugs involved a sign change in an arithmetic expression. Logic bugs were implemented by using the wrong logical operator in an IF condition. Data bugs involved wrong index values for variables.

Each bug in this experiment was purposely designed to affect only a limited area of code. That is, each calculation containing a bug occurred near the corresponding WRITE and FORMAT statements. In no case did a bug produce errors in routines other than the one in which it was embedded, and each bug appeared in only one line of code.

Length. The inclusion of additional subroutines to the programs employed in Experiment III made it possible to present each program in three different lengths. The shorter programs had 25-75 statements, medium programs contained 100-150 statements, and the longer programs contained approximately 175-225 statements.

Program listings included a two or three line explanation of any routine or function that was called by a program but not presented in the experimental materials. Participants were told to assume that missing routines worked correctly. All of the input and output files were presented regardless of the length of the program. That is, for the shorter version, some of the input was read in and some of the output was produced by subroutines which were not presented.

Experimental design. In order to control for individual differences in performance, a within subjects, 3^4 factorial design was employed in each experiment (Kirk, 1968). In

Experiment I, three types of control flow were defined for each of nine programs, and each of these 27 versions was presented in three levels of variable mnemonicity, for a total of 81 programs. In Experiment II, three levels of control flow were defined for each of the three programs. Each of these nine versions was presented in three levels of commenting. Modifications at three levels of difficulty were developed for each program, generating a total of 81 experimental conditions. In Experiment III, three types of control flow were defined for each of three programs, and each of these nine versions was presented in three lengths with three different bugs, for a total of 81 different experimental conditions (Table 1).

The first 27 participants in each experiment exhausted the 81 separate programs, and the final 9 participants in the first two experiments repeated 27 of the previous experimental tasks. Two complete replications of the design in Experiment III were afforded by the 54 participants. Programmers at each location were randomly assigned to experimental conditions, but in such a way that over the course of their three experimental programs, every participant had experienced each level of each independent variable. That is, they had worked with a program from each class, with each type of structure, with each type of commenting, etc. The order of presentation of the three programs to each participant was random. The order of

Table 1. Independent and dependent variables for Experiments 1, 2, and 3

	Experiment 1 Understanding	Experiment 2 Modification	Experiment 3 Debugging
Independent variables	3 Programs	3 Programs	3 Programs
	3 Control Flows	3 Control Flows	3 Control Flows
	3 Mnemonicity Levels	3 Documentations	3 Lengths
	3 Program Types	3 Modifications	3 Bugs
Dependent variables	% Correct	% Correct	
		Time	Time

presentation of conditions was not counterbalanced in the first two experiments. However, this problem was corrected in Experiment III to assure that each level of each independent variable appeared as the first, second, or third task an equal number of times.

Dependent Variables

Experiment I. Current literature (Love, 1977; Shneiderman, 1977) suggests that the most sensitive measure of whether programmers understand a program is their ability to learn its structure and reproduce a functionally equivalent program without notes. Thus, the percent of statements correctly recalled became the dependent variable in Experiment I. The criterion for scoring the reconstructed programs was the functional correctness of each separately reconstructed statement. Variable names and statement numbers which differed from those in the original program were counted as correct when used consistently. Control structures could be different from the original program so long as the groups of statements performed the same functions.

Three judges scored each of the 108 reconstructed programs independently. Interjudge correlations of .96, .96, and .94 were obtained across the three sets of scores. The average of the judges' scores on each program (mean percent

of statements correctly reconstructed) was used as the performance measure in Experiment I.

Experiment II. The dependent variables in Experiment II were the accuracy of the implemented modification and the time taken by the participant to perform the task. The individual steps necessary for correct implementation of each modification had been delineated in advance and assigned equal weights. That is, prototypes of each version of a program with each modification correctly implemented were established as the criteria against which participants' work would be compared. An accuracy score reflecting the percent of steps correctly implemented in each modification was computed by comparing each participant's changes with the criteria. All of the implemented modifications were scored by the same grader. The time to implement a modification was measured to the nearest minute by an electronic timer. Thus the performance measures were the percent of changes correctly implemented to a program and the number of minutes required to complete them.

Experiment III. The dependent variable in Experiment III was the number of minutes necessary for the participant to locate and correct the bug.

Individual Differences Measures

Scores on the preliminary task in each experiment were used as a measure of programming ability related to the experimental tasks. Participants reported their type of programming experience and the number of years they had been programming professionally in the first two experiments. The information requested in Experiment III included specific type of experience, number of years programming professionally in Fortran, number of statements in the longest Fortran and non-Fortran programs written, the first programming language learned, and number of languages learned. In addition, various programming concepts that appeared relevant to the experimental programs were listed, and participants were asked to mark those with which they were familiar.

SOFTWARE COMPLEXITY METRICS

Halstead's Software Science

Halstead (1977) developed a theory which provides objective estimates of the effort and time required to generate a program, the effort required to understand a program, and the number of bugs in a particular program (Fitzsimmons & Love, 1978). In 1972, Halstead first published his theory of software physics (renamed software science) stating that algorithms have measurable characteristics analogous to physical laws. According to Halstead, the amount of effort required to generate a program can be calculated from simple counts of the actual code. The calculations are based on four quantities from which Halstead derives the number of mental comparisons required to generate a program; namely, the number of distinct operators and operands and the total frequency of operators and operands.

Preliminary tests of the theory reported very high correlations (some greater than .90) between Halstead's metrics and such dependent measures as the number of bugs in a program (Cornell & Halstead, 1976; Funami & Halstead, 1975), programming time (Gordon & Halstead, 1976), and the quality of programs (Bulut & Halstead, 1974; Elshoff, 1976; Gordon, 1977, Halstead, 1973). Fitzsimmons and Love

(1978), Funami and Halstead (1975), and Akiyama (1971) found that Halstead's effort metric was a much better predictor of the number of errors in a program than either the number of program steps or the sum of the decisions and calls.

Volume. Halstead presents a measure of program size which is different from the number of statements in the code. His measure of program volume is also independent of the character set of the language in which the algorithm is implemented. Halstead defines his measure of program volume as:

$$V = (N_1 + N_2) \log_2 (\eta_1 + \eta_2)$$

where,

η_1 = number of unique operators,

η_2 = number of unique operands,

N_1 = total frequency of operators,

N_2 = total frequency of operands.

Level. Halstead's theory also generates a measure of program level which indicates the power of a language. As

the program level approaches 1, the statement of the problem or its solution becomes more succinct. As the program level approaches Ø, the statement of a problem or its solution becomes increasingly bulky, requiring many operators and operands. A higher level language is assumed to have more powerful operators available; thus fewer operators need to be used to implement a particular algorithm. Halstead's estimate of program level is computed as:

$$\hat{L} = 2\eta_2/\eta_1 N_2.$$

Effort. Halstead theorized that the effort required to generate a program would be a ratio of the program's volume to its level. He proposed this measure as representing the number of mental discriminations a programmer would need to make in developing the program. Halstead's effort metric (E) was computed precisely from a program (Ottenstein, 1976) which was modified to accept as input the source code from the program studied. The computational formula was:

$$E = \frac{\eta_1 N_2 (N_1 + N_2) \log_2 (\eta_1 + \eta_2)}{2\eta_2}$$

McCabe's Complexity Metric

McCabe (1976) defined complexity in relation to the decision structure of a program. He attempted to assess complexity as it affects the testability and reliability of a

module. McCabe's complexity metric, v(G), is the classical graph-theory cyclomatic number indicating the number of regions in a graph, or in the current usage, the number of linearly independent control paths comprising a program. The computational formula is:

$$v(G) = \#\ \text{edges} - \#\ \text{nodes} + 2(\text{connected components}).$$

Simply stated, McCabe's v(G) counts the number of basic control path segments through a computer program. These are the segments which, when combined, will generate every possible path through the program. McCabe presents two simpler methods of calculating the metric. McCabe's v(G) can be computed as either the number of predicate nodes plus 1, or as the number of regions in a planar graph of the control flow.

The simplest possible program would have v(G) = 1. Sequences do not add to the complexity. IF-THEN-ELSE, DO-WHILE, or DO-UNTIL constructs each increase the complexity by 1. It is assumed that regardless of the number of times a DO loop is executed there are really only two control paths: the straight path through the DO and the return to the top. Clearly, a DO executed 25 times is not 25 times more complex than a DO executed once.

Results

Significant intercorrelations were observed among the complexity metrics in all experiments. In Experiment I the Halstead and McCabe metrics were strongly related, while their correlations with lines of code were moderate (Table 2). In Experiment II all three measures were strongly intercorrelated. In Experiment III, with longer programs, the relationship between Halstead's E and the other measures was only moderate.

Correlations with Performance

Experiment I. Since different levels of variable mnemonicity and type of commenting neither affected performance nor caused any change in the value of the complexity metrics for a particular program, some of the data reported in the sections on performance predictions were aggregated over the three levels of mnemonicity in Experiment I and the three types of commenting in Experiment II. Thus, when analyses are reported for 27 data points, each datum represents the average of at least three performance scores.

The correlations between percent of statements correctly recalled and complexity metrics were all negative, indicating that fewer lines were correctly recalled as the level of complexity represented by these three measures

Table 2. Intercorrelations for Software Complexity Measures

Metric	Correlations E	Correlations v(G)
Experiment 1 (n = 27)		
McCabe's v(G)	.84***	
Length	.47**	.64***
Experiment 2		
Unmodified (n = 9)		
McCabe's v(G)	.85**	
Length	.97***	.90***
Modified (n = 27)		
McCabe's v(G)	.88***	
Length	.92***	.89***
Experiment 3 (n = 27)		
McCabe's v(G)	.76***	
Length	.56***	.90***

Note: Halstead E values are reported in thousands of mental discriminations.

*$p \leq .05$
**$p \leq .01$
***$p \leq .001$

increased. Length and McCabe's v(G) were moderately related to performance, while little relationship was found for Halstead's E (Table 3).

Investigation of a scatterplot of Halstead's E with performance indicated that there were three extreme scores which were obtained from three participants who consistently outscored others on both the pretest and the experimental task. With the three data points of the exceptional group removed, the correlations for all three complexity metrics improved. Further, there were considerable differences in difficulty among the programs studied. As a heuristic device to determine whether the complexity metrics were more predictive of performance within programs than across them, a transformation was applied separately to the data for each program. Although Halstead's E was unrelated to performance in the raw data, a strong correlation was observed after corrections were made for differences among programs and participants. Such an improvement was not observed in the results for v(G) or length.

Experiment II. The complexity metrics were generally more strongly correlated with time to completion than with the accuracy of the implementation, especially on the modified programs. Both metrics and length were moderately related to both criteria on modified programs (Table 4).

Table 3. Correlations of Complexity Metrics With Percent of Statements Correctly Recalled in Experiment 1

Criterion	Correlations		
	E	v(G)	Length
Unaggregated data (n = 108)	-.19*	-.34***	-.47***
Aggregated data (n = 27)	-.13	-.35*	-.53**
Exceptional group removed (n = 24)	-.36*	-.55**	-.61***
Transformed scores (n = 27)	-.10	-.24	-.38*
Exceptional group removed and transformed scores (n = 24)	-.73***	-.21	-.65***

*$p \leq .05$
**$p \leq .01$
***$p \leq .001$

Table 4. Correlations of Complexity Metrics With Accuracy and Time in Experiment 2

Criterion	Correlations		
	E	v(G)	Length
Unaggregated (n = 108)			
Accuracy			
Unmodified	-.12	-.21*	-.17*
Modified	-.17*	-.21*	-.20*
Time to completion			
Unmodified	.16*	.15	.13
Modified	.28**	.24**	.30***
Aggregated (n = 27)			
Accuracy			
Unmodified	-.21	-.36*	-.28*
Modified	-.29	-.36*	-.34*
Time to completion			
Unmodified	.25	.23	.20
Modified	.44**	.38*	.46**

*$p \leq .05$
**$p \leq .01$
***$p \leq .001$

Experiment III. All three metrics predicted performance equally well at the subroutine level. At the program level, however, E was the best predictor, accounting for more than twice the variance in performance than program length (56% versus 27%, respectively). The variance accounted for by v(G) fell between these values (42%). A stepwise multiple regression analysis indicated that length and v(G) added no increments to the prediction afforded by E (Table 5).

The scatterplot of performance with Halstead's E suggested the existence of a curvilinear trend in the data. The significance of this trend was tested using the second degree polynomial regression approach suggested by Cohen and Cohen (1975) and Kerlinger and Pedhazur (1973) for investigating curvilinear relationships. A multiple correlation coefficient of .84 indicated that the curvilinear trend accounted for an additional 15% ($p \leq .001$) of the variance beyond that accounted for by a linear relationship. The prediction equation generated from these data was:

$$\text{minutes to find bug} = 9.837 + .00239E - .00000000079E^2$$

However, with very few data points in the right tail of this distribution for Halstead's E, it is difficult to extrapolate to the exact shape of the curvilinear trend. No curvilinear trend was detected with either the lines of code or McCabe's v(G).

Table 5. Correlations of Complexity Metrics With Performance Time in Experiment 3

	Correlations		
Criterion	E	v(G)	Length
Unaggregated (n = 162)			
Subroutine	.25***	.24***	.25***
Program	.28***	.25***	.20**
Aggregated (n = 27)			
Subroutine	.66***	.63***	.67***
Program	.75***	.65***	.52**

**$p \leq .01$
***$p \leq .001$

Moderator Effects

In order to determine the effects of possible moderators, correlations between performance measures and software complexity metrics were computed under different types of control flow and commenting, and at different levels of programmer experience. In Experiment I, Halstead's E and McCabe's v(G) correlated significantly with performance only on unstructured programs (Table 6). While a similar pattern of correlations emerged in Experiment II between Halstead's E and time to complete the modification, differences among these correlations were not significant. The moderating effects did not appear to result from restrictions of range on the variable involved. That is, means and variances for complexity metrics were identical across types of control flow, and although mean performance scores differed across types of control flow, no significant differences were observed in variance. This moderating effect could not be tested in Experiment III, since unstructured programs were not employed.

Significant moderating effects were also found for types of commenting. All but one of the significant correlations between complexity metrics and modification accuracy and time occurred when no comments were included in the code. Differences in correlations between in-line and no comment

Table 6. Correlations Between Performance Measures and Complexity Metrics Under Different Types of Control Flow

Criterion and type of control flow	Correlations		
	E	v(G)	Length
Experiment 1 (n = 36)			
% recalled			
Unstructured	-.45***	-.55***	-.68***
Naturally structured	.07	-.08	-.20
Structured	-.01	-.11	-.51***
Experiment 2 (n = 36)			
Time to completion			
Unstructured	.38*	.24	.37*
Naturally structured	.28*	.20	.34*
Structured	.08	.21	.12

*$p \leq .05$
**$p \leq .01$
***$p \leq .001$

Table 7. Correlations Between Performance Measures and Complexity Metrics Under Different Types of Commenting in Experiment 2

Criterion and type of comments	Correlations		
	E	v(G)	Length
Accuracy (n = 36)			
None	-.34*	-.35*	-.37*
Global	-.18	-.31*	-.23
In-line	.03	.04	.03
Time (n = 36)			
None	.47**	.44**	.55***
Global	.21	.18	.18
In-line	.16	.11	.16

*$p \leq .05$
**$p \leq .01$
***$p \leq .001$

conditions either achieved or bordered on significance in all cases (Table 7).

Finally, relationships between complexity metrics and performance measures were moderated by the participants' years of professional programming experience. The dividing point between three or fewer years and more than three years experience was arbitrary and represented a compromise between minimizing the years of experience in the less experienced category and having a sufficient number of participants for a correlational analysis. The complexity metrics were more strongly related to performance among less experienced programmers in all three experiments (Table 8).

Discussion

The three experiments comprising this study produced empirical evidence that software complexity metrics were related to the difficulty programmers experienced in understanding and modifying software. The correlations observed in the first two experiments, however, were not as high as those reported by Halstead (1977) in other verifications of this theory. While many of the correlations reported here will not seem large to readers of the engineering or physical science literature, their magnitudes

Table 8. Correlations Between Performance Measures and Complexity Metrics Among Programmers Differing in Experience

Criterion and level of experience	Correlations		
	E	v(G)	Length
Experiment 1			
% recalled			
≤ 3 years (n = 42)	-.35*	-.47***	-.55***
> 3 years (n = 60)	-.03	-.18	-.31**
Experiment 2			
Time to completion			
≤ 3 years (n = 32)	.55***	.52***	.56***
> 3 years (n = 75)	.20*	.15	.22*
Experiment 3			
≤ 3 years (n = 75)	.38***	.29***	.18
> 3 years (n = 87)	.20*	.21*	.22*

Note: Two participants in Experiment 1 did not report their years of experience

*$p \leq .05$
**$p \leq .01$
***$p \leq .001$

broader range of program sizes. Second, individual differences among programmers exerted significant effects on the results obtained. In order to control for these differences, the number of participants in the experiments would need to be increased. When the data from Experiment I were transformed in an attempt to control for differences among programs and programmers, a correlation of -.73 ($p \leq .001$) was obtained between the performance criterion and Halstead's E. However, the question is not whether theories can be validated with mystical transformations of data, but whether the results of these heuristic transformations can be replicated in an experiment designed to overcome the limitations of previous research.

In Experiment III some of the limitations of the previous experiments were corrected and the results not only replicated those of previous experiments, but also demonstrated that far stronger results could be obtained when their limitations were overcome. The curvilinear relationship observed in this experiment between Halstead's E and performance could not have been observed if only short programs had been used in the experimental tasks. The Halstead's E and length values at the subroutine level suggest that both are capturing program volume in modular sized programs (approximately 50 lines of code or less). With larger programs the information measured appears to differ; that is, Halstead measures something in addition to,

are typical of significant results reported in human factors research.

In the first two experiments lines of code proved to be a better predictor of programmer performance than either of the complexity metrics studied. These results were disappointing, since the complexity metrics were believed to capture constructs much more closely associated with factors which affect the psychological complexity of software than lines of code. That is, counts of operators, operands, and elementary segments of the control flow should be more closely related to the difficulty programmers experience in working with software.

Stronger relationships in the first two experiments may have been obscured by variation in performance scores related to differences among participants and programs which were enhanced by the economical multifactor designs employed in these experiments. There were several limitations in the experimental procedures employed in obtaining the data which may have diluted the results we observed. First, all of the programs studied were short (35-55 lines of code). The limited range of metric values calculated on programs of this length may not have been sufficient for an adequate test of the predictive worth of the metrics. Studies reporting higher correlations for Halstead's E usually involved a

but inclusive of, that which is measured by length.

Many small-sized programs can be grasped by the typical programmer as a cognitive gestalt. The psychological complexity of such programs is adequately represented by the volume of the prcgram as indexed by the number of lines. When the code grows beyond a subroutine or module, its complexity to the programmers is better assessed by measuring constructs other than the number of lines of code. This may result partly because programmmers may represent the program to themselves in ways which are more accurately captured by counts of operators, operands, and control paths. Thus, as the size of a program increases, Halstead's E seems to become a better measure of its psychological complexity.

One possible explanation for the superior predictive ability of Halstead's E is that the relation between program size and performance is curvilinear and the logarithmic transformation within the Halstead measure captures this relationship while lines of code does not. However, there was no support in these data for a curvilinear relationship between lines of code and performance. On the other hand, a curvilinear relationship did exist between Halstead's E and performance. This curve suggests that as Halstead's E grows larger, a program becomes more psychologically complex, but the increments in difficulty grow smaller and smaller. In the debugging task there seemed to be an amount of time that

was typically required to locate a bug within a subroutine once the correct subroutine had been identified (approximately 16 minutes). Added to this base was the time required to identify the proper subroutine. The curvilinearity of the relationship between time to find the bug and Halstead's E appeared to result from the time required to isolate the problem subroutine.

A distinguishing characteristic of psychological complexity is the interaction between program characteristics and individual differences, such as programming experience. Chrysler (1978) demonstrated the value of experiential variables in predicting the time to complete a programming task. In these experiments the complexity metrics were more highly related to the performance of less experienced programmers. Thus, the complexity metrics may not represent the most important constructs for predicting the performance of experienced programmers. These programmers probably conceptualize programs at a level other than that of operators, operands, and basic control paths. Programmers may have fit the program into a schema they recognized from previous experience, just as chess players learn to visualize the gameboard differently with experience. Results did not indicate, however, that more experienced programmers performed the tasks more efficiently. Since the ratio comparing good to poor programmer performance is often as great as 28 to 1 (Sackman, Erickson, & Grant, 1968), the

selection, training, and placement of programmers can have a significant impact on project costs and product quality.

Software complexity metrics appear to be capable of satisfying the uses described in the introduction. That is, they can be used in providing feedback to programmers about the complexity of the code they have developed and to managers about the resources that will be necessary to test or maintain particular sections of code. Further evaluative research needs to assess the validity of these uses in on-going software projects.

Halstead's Software Science has provided some important initial directions for investigating psychological complexity. Subsequent work by McCabe (1976) and others has also proven relevant to this purpose. Yet, assessing the psychological complexity of software requires more than a simple count of operators, operands, and basic control paths. If the ability of complexity metrics to predict programmer performance is to be improved, the metrics used to predict programmer performance must be related by psychological principles to the memory, information processing, and problem solving capacities of programmers. In identifying a set of psychological principles relevant to programming tasks, it will be important to determine methods for quantifying factors in the code which represent information concerning the program. This approach might not only generate improved

metrics for assessing psychological complexity, but might also identify programming practices which could lead to simplified, more easily maintained software.

MODERN CODING PRACTICES

Structured coding techniques, mnemonic variable names, and commenting are among the modern coding practices which supposedly increase a programmer's efficiency in working with code. Dijkstra (1972) contended that program construction should proceed in a top-down, structured fashion. By limiting the control structures allowed, he assumed that structured programs would be easier to understand, debug, and modify than unstructured ones because the simplified control flow would make the functions performed by the program easier to trace. Weissman (1974) demonstrated that well-structured programs were easier to understand, and Lucas and Kaplan (1974) found that structured programs took less time to modify. Love (1977) observed that simplified control flow made programs more understandable to graduate students, but not to beginning student programmers.

Mnemonic variable names provide a form of documentation commonly believed to simplify the cognitive task of understanding or modifying a program. In experiments using short Fortran programs, Shneiderman (1976) found that meaningful variable names produced superior comprehension and debugging performance when compared to one or two letter variable names. Newsted (1979), however, cautioned that a poor selection of mnemonic names may interfere with

comprehension: "One programmer's mnemonic is another's gobbledygook" (p.21).

The inclusion of comments is another type of documentation purported to simplify modification tasks, although there is some contention over how they should be implemented. Global comments preceding a program indicate what objectives are accomplished, while in-line comments delineate how and where the objectives are fulfilled. Some authors encouraged the use of in-line comments to simplify the process of making changes to programs (Wilkes, Wheeler, & Gill, 1951; Poole, 1973). Others (Musa, 1976; Shneiderman, 1977) found that global comments improved student programmer's ability to comprehend and modify programs, but that in-line comments seemed distracting. In a Fortran modification task with student programmers, Yasukawa (1974) found that a group given global comments performed better than a group given in-line comments. However, Newsted (1974) observed that comments might actually interfere with attempts to understand short Fortran programs of less than 30 lines. Still other computer scientists recommend both global and in-line comments, suggesting that there is never too much documentation.

Results

Experiment I. In Experiment I, 50% of the statements were correctly recalled across all programs and experimental conditions. Performance differed significantly among the program classes. These differences accounted for 8% of the variance in performance in addition to that accounted for by individual differences among participants. Engineering programs were the most difficult (41% of the statements correctly recalled), followed by statistical (52%), and non-numeric (57%) programs. When the specific program was taken into account, another one-fifth of the variance in performance was explained. However, this result was not strictly a function of differences among programs, because variance related to specific programs was confounded with variance related to participants. That is, each participant saw only three of the nine programs.

The complexity of the control flow significantly affected performance. As expected, the least structured level was the most difficult to reconstruct. Contrary to the tenets of structured programming, however, the most structured level did not produce the best performance. A greater percent of statements were recalled from naturally structured (56%) than from structured programs (52%), although this difference was not significant. A post hoc analysis showed the means for naturally structured and

unstructured programs (56% versus 42%) to be significantly different.

Assigning different levels of mnemonicity to variable names had no significant effect on performance. Also, performance was not affected by the order in which programs were presented to participants, suggesting that any learning process which might have affected the results occurred during the pretest rather than during the three experiment tasks.

Experiment II. Across all experimental conditions, an average of 62% of the requirements for each modification was accurately implemented (SD = 31%). The average time to complete the modifications was 17.9 minutes (SD = 11.4), ranging from 2 to 59 minutes with a positive skew. Score and time were uncorrelated.

Only a small percentage of the variance in accuracy scores were related to the factors studied here. However, there were substantial differences in the degree to which performance on each of the three programs could be predicted. Performance on two of the programs was significantly predicted. Half of the variance was accounted for in the separate results for each program, and 35% was accounted for in the combined results of both programs. However, the results for the third program were not significant.

Order of presentation significantly affected accuracy scores. Participants made more complete modifications in less time with each succeeding experimental task. However, the two programs on which performance proved most predictable were more frequently presented second or third. Thus, random assignment of presentation orders failed to counterbalance the number of times each condition appeared in each position order.

The difficulty of the modification affected accuracy scores on only the two most predictable programs. Performance on those programs was poorer on modifications which required more lines of code to be inserted. The complexity of the control flow also affected accuracy scores on the two programs for which accuracy was most predictable. Modifications to the structured programs were more accurate than those made to unstructured programs. Accuracy scores did not differ among programs, nor among the type of documentation included in the program.

Over one quarter of the variance in the time required to complete the modifications across all three programs could be accounted for by variables studied here. Time to complete the modifications was more easily predicted than accuracy scores across all three programs and on the program where accuracy was poorly predicted. Results for time

generally were similar to those observed for accuracy. The specific program and type of documentation were unrelated to the criterion. Significant effects were observed for difficulty of the modification and order of presentation, although again, the interpretation of the effect for this latter variable is confounded. Control flow complexity was not significantly related to time, although it was modestly related to accuracy.

Further inspection verified that the number of additional statements required in the code to complete a modification accurately was related to the time required to insert them. Fitting a curvilinear function to these data using least squares procedures resulted in a curvilinear correlation of .80 and a standard error of estimate of 2.5 minutes. No such relationship was found for accuracy.

Experiment III. The average time to locate bugs across all experimental conditions was 20.1 minutes (SD = 16.2). All but six of the 162 experimental tasks comprising this experiment were completed successfully during the allotted 60 minutes. These six conditions were not associated with any particular factor. Despite the use of a preliminary task to familiarize the participants with the experiment, a significant order effect occurred ($p < .04$), indicating that learning took place during the first of the three experimental tasks.

Differences in solution time for the three programs were significant. Finding the bug in the accounting program required an average of 15.1 minutes, 20.0 minutes in the program that sorted questionnaire data, and 25.0 minutes in the grade-scoring program. Increasing the length of the programs had a modest effect on the time to locate and correct the error. The average time for the short programs was 16 minutes, while the medium and long programs required a mean of 21 and 23 minutes, respectively.

Averages for the three error categories were not significantly different from one another. However, a very large interaction occurred between type of bug and program. This interaction accounted for the largest percent of variance (26%) of any of the experimental relationships studied. No significant differences in performance resulted from the three types of control flow.

Discussion

Control flow complexity was significantly related to programmer performance in two of the experiments. In Experiment II structured code tended to produce more accurate modifications than unstructured code. In Experiment I, however, naturally structured code was more easily comprehended than unstructured code. It was not clear from

the results of these two experiments whether rigidly structured code or code structured with a more natural control flow for Fortran can be maintained more efficiently. However, both structured control flows proved superior to unstructured code in at least one of the experiments.

In order to determine the most effective method for structuring a program, Experiment III compared naturally and graph-structured Fortran IV to Fortran 77. No significant differences were evident among these three types of top-down control flow. This finding agreed with previous results where differences were found between top-down and convoluted control flow, but not between types of top-down control flow. The minor deviations from strictly structured code allowed in the naturally structured version of Experiment III did not adversely affect performance. Summarizing the combined results of the three experiments, it would appear that the overall top-down quality of the control flow is important to performance, but careful attention to strict structuring does not appear to improve programmer performance significantly. Further, since no difference was found between the graph-structured and Fortran 77 program versions, it would appear that the new constructs provide little additional aid in a debugging task beyond that provided by a forward flow.

The mnemonic value of variable names did not affect

performance. However, many participants seemed to prefer mnemonic names, since they used their own, more meaningful names when rewriting the least mnemonic versions of the programs. For the medium and most mnemonic versions, they tended to use the original names supplied. Thus, the contribution of mnemonic variable names is supported by anecdotal rather than statistical evidence.

It was expected that the inclusion of either global or in-line comments would significantly improve performance on a modification task. No such improvement was observed. Nevertheless, this counterintuitive result concurs with the non-significant effect for mnemonic variable names in Experiment I. Lack of effects for documentation aids in both experiments may have occurred for several reasons. First, in Experiment I where levels of variable mnemonicity were manipulated, global comments were provided with all programs. In Experiment II where types of comments were manipulated, mnemonic variable names were provided in all programs. Thus, the existence of one type of documentation may have reduced the additional information available from the documentation aid being experimentally manipulated, reducing its impact on performance.

A second possibility is that documentation aids do not contribute significantly to performance for programs of the modular size (approximately 50 lines) employed here. In

large systems with many modules and thousands of lines of code, documentation may have more impact on performance because of the increased amount of information to be processed. Thus, program size may moderate the relationship between documentation and performance.

Performance effects due to differences among programs in Experiment I are not easily explained from these results. The significant effect due to class of program may have been a function of some familiarity factor specific to the samples of programs and programmers studied.

In Experiment III a large percent of the variance in performance was accounted for by a program-by-error interaction. It appears that some quality of the algorithm in which the bug is embedded influences a programmer's ability to locate it. This result has implications for the usefulness of various schemes for categorizing software bugs. The implied value of these taxonomies is to identify properties of bugs which suggest how they are created or how difficult they are to detect. Simple taxonomies based on syntactic relationships will probably not prove sufficient for this purpose. The results of this experiment suggest that the detectability of a bug depends on the context of the algorithm surrounding it. This contextual effect may determine the optimal strategy for finding the bug, and it is this search strategy that needs to be understood if debugging

performance is to be improved.

In the last section of the post-session questionnaire, the participants were asked to describe their searching strategies for locating the bugs. Typically, one of two approaches was described. In the first strategy the programmer tried to understand the whole program from beginning to end before searching for the section with the bug. In the second strategy the programmer used appropriate clues in the output to go directly to the section containing the bug. The latter appeared to be a much quicker strategy for debugging, but there were insufficient data for a meaningful statistical analysis. In order to improve the debugging performance of programmers it will be important not only to identify effective search strategies, but also to identify conditions under which they will be differentially effective.

Different search strategies emphasize the importance of individual differences among programmers. As measured by a pre-test, these differences accounted for significant variance in performance in the first two, but not in the third, experiments. The effect of individual differences might have been even greater if the sample had been expanded beyond the programmers studied. It was also possible to predict the performance of an individual programmer from job history data. Several important factors seemed to be the

number of languages a programmer had used and familiarity with certain programming concepts. Thus, the important factor in work history seemed to be the breadth rather than the length of experience. These predictions from job history were more valid for programmers who had three or fewer years of experience in Fortran. Future work is needed to refine experiential questionnaires for use in personnel functions such as selection, assessment for training needs, and placement.

In summary, several factors influencing the understandability and modifiability of computer programs were identified. Yet, results for the modern programming practices studied here were probably conservative due to the small size of the programs studied. The cognitive load placed on programmers attempting to understand or modify approximately 50-line programs did not require the degree of cognitive assistance provided cumulatively by structured coding, mnemonic variable names, and comments. While the information provided by these practices were not necessarily redundant, the task could be mastered with less information than was presented. However, in a large system composed of many modules, the cognitive burden of implementing modifications may be so great that each of these programming practices may contribute significantly to efficiency. Thus, future research needs to assess the independent benefits of these practices in substantially larger programs.

REFERENCES

Akiyama, F. An example of software debugging. In Proceedings of the 1971 Congress of the International Federation of Information Processing Societies. Amsterdam: North-Holland, 1971.

Boehm, B.W., Brown, J.R., & Lipow, M. Quantitative evaluation of software quality. In Proceedings of the First International Conference of Software Engineering. New York: IEEE, 1976.

Brainerd, W. Fortran 77. Communications of the ACM, 1978, 21, 806-820.

Bulut, N. & Halstead, M.H. Impurities found in algorithm implementation (Tech. Rep. CSD-TR-111). West Lafayette, IN: Purdue University, Computer Science Department, 1974.

Campbell, D., & Stanley, J.C. Experimental and quasi-experimental designs for research. Chicago: Rand-McNally, 1966.

Carlson, W.E., & DeRoze, B. Defense system software research and development plan. Unpublished manuscript, Arlington, VA: Defense Advanced Research Project Agency, September 1977.

Chrysler, E. Some basic determinants of computer programming productivity. Communications of the ACM, 1978, 21, 472-483.

Cohen, J., & Cohen, P. Applied multiple regression/correlation analysis for the behavioral sciences. New York: Wiley, 1975.

Cornell, L., & Halstead, M.H. Predicting the number of bugs expected in a program module (Tech. Rep. CSD-TR-205). West Lafayette, IN: Computer Science Department, Purdue University, 1976.

DeRoze, B. Software research and development technology in the Department of Defense. Paper presented at the AIIE Conference on Software, Washington, D.C.: December 1977.

Dijkstra, E.W. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, & C.A.R. Hoare (Eds.), Structured programming. New York: Academic Press, 1972.

Elshoff, J.L. Measuring commercial PL/1 programs using Halstead's criteria, SIGPLAN Notices, 1976, 11, 30-36.

Elshoff, J.L. A review of software measurement studies at General Motors Research Laboratories. In Proceedings of the Second Annual Software Life Cycle Management Workshop. New York: IEEE, 1978.

Fitzsimmons, A.B., & Love, L.T. A review and evaluation of software science. ACM Computing Surveys, 1978, 10, 3-18.

Funami, Y., & Halstead, M.H. A software physics analysis of Akiyama's debugging data (Tech. Rep. CSD-TR-144). West Lafayette, IN: Purdue University, Computer Science Department, May 1975.

Gordon, R.D. A measure of mental effort related to program clarity. Unpublished doctoral dissertation, Purdue University, 1977.

Gordon, R.D., & Halstead, M.H. An experiment comparing Fortran programming time with the software physics hypothesis. AFIPS Conference Proceedings, 1976, 45, 935-937.

Grant, E.E., & Sackman, H. An exploratory investigation of programmer performance under on-line and off-line conditions. IEEE Transactions on Human Factors in Electronics, 1967, 8, 33-48.

Halstead, M.H. An experimental determination of the "purity" of a trivial algorithm (Tech. Rep. CSD-TR-73). West Lafayette, IN: Purdue University, Computer Science Department, 1973.

Halstead, M.H. Elements of software science. New York: Elsevier, 1977.

Hecht, H., Sturm, W.A., & Trattner, S. Reliability measurement during software development. Unpublished manuscript. Redondo Beach, CA: Aerospace, 1978.

Kerlinger, F.N., & Pedhazur, E.J. Multiple regression in behavioral research. New York: Holt, Rinehart, & Winston, 1973.

Kirk, R.E. Experimental design procedures for the behavioral sciences. San Francisco: Freeman, 1968.

Love, L.T. Relating individual differences in computer programming performance to human information processing abilities. Dissertation Abstracts International, 1977, 38, 1443b.

Lucas, H.C. Jr., & Kaplan, R.B. A structured programming experiment. The Computer Journal, 1974, 19, 136-138.

McCabe, T.J. A complexity measure. IEEE Transactions on Software Engineering, 1976, 2, 308-320.

McCall, J.A., Richards P.K., & Walters, G.F. Factors in software quality (Tech. Rep. 77CIS02). Sunnyvale, CA: Information Systems Programs, General Electric, 1977.

Musa, J.D. An exploratory experiment with "foreign" debugging of programs. In Proceedings of the Symposium on Computer Software Engineering. New York: Polytechnic Institute of New York, 1976.

Newsted, P.R. Fortran program comprehension as a function of documentation. Program Information Abstracts: Second Annual Computer Science Conference. Detroit, 1974, 25.

Newsted, P.R. Flowchart-free approach to documentation. Journal of Systems Management, 1979, 30 (4), 18-21.

Ottenstein, K.J. A program to count operators and operands for ANSI-Fortran modules (Tech. Rep. CSD-TR-196). West Lafayette, IN: Computer Science Department, Purdue University, 1976.

Poole, P.C. Debugging and testing. In F.L. Bauer (Ed.) Advanced course in software engineering. New York: Springer-Verlag, 1973.

Sackman, H., Erickson, W.J., & Grant, E.E. Exploratory experimental studies comparing on-line and off-line programming performance. Communications of the ACM, 1968, 11, 3-11.

Sheppard, S.B., & Love, L.T. Testing influences on human understanding of software. Proceedings of the 21st Meeting of the Human Factors Society. Santa Monica, CA: Human Factors Society, 1977, 167-171.

Shneiderman, B. Exploratory experiments in programmer behavior. International Journal of Computer and Information Sciences, 1976, 5, 123-143.

Shneiderman, B. Measuring computer program quality and comprehension. International Journal of Man-Machine Studies, 1977, 9, 465-478.

Tenny, T. Structured programming in Fortran. Datamation, 1974, 20, 110-115.

The military software market (Rep. 427). New York: Frost & Sullivan, 1977.

Weissman, L.M. A methodology for studying the psychological complexity of computer programs (Tech. Rep. TR-CSRG-37). Toronto, Canada; University of Toronto, Computer Systems Research Group, 1974.

Wilkes, M.V., Wheeler, D.J., & Gill, S. The preparation of programs for an electronic digital computer. Reading, MA: Addison-Wesley, 1951.

Yasukawa, K. The effect of comments on program understandability and error correction. Unpublished paper, 1974.

QUANTITATIVE MEASURES OF THE PSYCHOLOGICAL COMPLEXITY OF SOFTWARE PROJECT

Project Team

Dr. Bill Curtis
Dr. Tom Love
Sylvia B. Sheppard
Phil Milliman
M.A. Borst
Ann Fitzsimmons
Judith McWilliams
Beverly Day

Technical Reports

TR-77-388100-1 (AD A041916)	A Preliminary Experiment to Test Influences on Human Understanding of Software	Sheppard & Love
TR-78-388100-2 (AD A051495)	Predicting Software Comprehensibility	Sheppard, Borst, & Love
TR-78-388100-3 (AD A056079)	Predicting Programmer's Ability to Modify Software	Sheppard, Borst, Curtis, & Love
TR-78-388100-4	Programmers Versus Non-Programmers in a Fortran Comprehension Test	Borst
TR-79-388100-5	Factors Affecting Programmer Performance in a Debugging Task	Sheppard, Milliman, & Curtis
TR-79-388100-6	Experimental Evaluation of On-line Program Construction	Milliman, Curtis, & Sheppard
TR-79-388100-7	Identification and Validation of Quantitative Measures of the Psychological Complexity of Software: Final Report	Curtis & Sheppard

Publications

Curtis, B., Sheppard, S.B., Borst, M.A., Milliman, P., & Love, T. Some distinctions between the psychological and computational complexity of software. In V. Basili (Ed.), Proceedings of the Second Software Life Cycle Management Workshop. New York: Institute of Electrical and Electronics Engineers, 1978.

Curtis, B., Sheppard, S.B., & Milliman, P. Third time charm: Stronger prediction of programmer performance with software complexity metrics. In Proceedings of the Fourth International Conference on Software Engineering. New York: Institute of Electrical and Electronics Engineers, 1979.

Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., & Love, T. Predicting performance on software maintenance tasks with the Halstead and McCabe metrics. IEEE Transactions on Software Engineering, 1979, 5, 95-104.

Fitzsimmons, A.B., & Love, L.T. A review and evaluation of software science. ACM Computing Surveys, 1978, 10, 3-18.

Love, T., & Curtis, B. Applying multifactor experimental design to the study of programming. In Proceedings of Computer Science and Statistics: 12th Annual Symposium on the Interface. Waterloo, Ontario: University of Waterloo, 1979.

Sheppard, S.B. Predicting programmers' performance from models of software complexity. In Proceedings from the Third Software Engineering Workshop. Greenbelt, MD: National Aeronautics and Space Administration, 1978.

Sheppard, S.B., Borst, M.A., Curtis, B., & Love, T. Predicting programmers' ability to understand and modify software. In W. Camm & R.E. Granda (Eds.), Symposium Proceedings: Human Factors and Computer Science. Santa Monica, CA: Human Factors Society, 1978.

Sheppard, S.B., Curtis, B., Milliman, P., Borst, M.A., & Love, T. First year results from a research program on human factors in software engineering. In Proceedings of the 1979 National Computer Conference. Montvale, NJ: American Federation of Information Processing Societies, 1979.

Sheppard, S.B., Curtis, B., Milliman, P., & Love, T. Human factors experiments on modern coding practices. Computer, in press.

Sheppard, S.B., & Love, L.T. Testing influences on human understanding of software. In Proceedings of the 21st Meeting of the Human Factors Society. Santa Monica, CA: Human Factors Society, 1977, 167-171.

Conference Presentations

10/77	Love	21st Meeting of the Human Factors Society, Human Factors Society, San Francisco
6/78	Love	Human Factors and Computer Science, Human Factors Society, Washington, D.C.
8/78	Curtis	Second Software Life Cycle Management Workship, AIRMICS, Atlanta
9/78	Sheppard	Third Summer Software NASA Engineering Workshop, Greenbelt, MD
9/78	Curtis	Workshop on Performance Modeling and Error Analysis, Syracuse U. IEEE, Blue Mtn. Lake, NY
12/78	Sheppard	ACM '78, ACM, Washington, D.C.
1/79	Curtis	Air Force Systems Command, Steering Committee on Man-Machine Interaction, ONR, Arlington, VA
5/79	Love	Computer Science and Statistics: 12th Annual Symposim on the Interface, University of Waterloo, Waterloo, Ontario
6/79	Curtis	1979 National Computer Conference, AFIPS, New York
8/79	Curtis	Second Minnowbrook Conference on Software Performance Evaluation, Syracuse U. IEEE, Blue Mtn. Lake, NY
9/79	Curtis	American Psychological Association, APA, New York
9/79	Curtis	Fourth International Conference on Software Engineering, IEEE & ACM, Munich, W. Germany

OFFICE OF NAVAL RESEARCH

Code 455

TECHNICAL REPORTS DISTRIBUTION LIST

OSD

CDR Paul R. Chatelier
Military Assistant for Training and
Personnel Technology
Office of the Deputy Under Secretary
of Defense
OUSDRE (E&LS)
Pentagon, Room 3D129
Washington, D.C. 20301

Department of the Navy

Director
Engineering Psychology Programs
Code 455
Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217 (5 cys)

Director
Information Systems Program
Code 437
Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217

Director
Physiology Program
Code 441
Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217

Special Assistant for Marine
Corps Matters
Code 100M
Office of Naval Research
800 North Quincy Street
Arlington, Virginia 22217

Commanding Officer
ONR Branch Office
ATTN: Dr. J. Lester
Building 114, Section D
666 Summer Street
Boston, MA 02210

Commanding Officer
ONR Branch Office
ATTN: Dr. C. Davis
536 South Clark Street
Chicago, IL 60605

Commanding Officer
ONR Branch Officer
ATTN: Dr. E. Gloye
1030 East Green Street
Pasadena, CA 91106

Office of Naval Research
Scientific Liaison Group
American Embassy, Room A-407
APO San Francisco 96503

Director
Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375 (6 cys)

Department of the Navy (Continued)

Dr. Bruce Wald
Communications Sciences Division
Code 7500
Naval Research Laboratory
Washington, D.C. 20375

Dr. Robert G. Smith
Office of the Chief of Naval
Operations, OP987H
Personnel Logistics Plans
Washington, D.C. 20350

Naval Training Equipment Center
ATTN: Technical Library
Orlando, FL 32813

Human Factors Department
Code N215
Naval Training Equipment Center
Orlando, FL 32813

Dr. Alfred F. Smode
Training Analysis and Evaluation Group
Naval Training Equipment Center
Code N-00T
Orlando, FL 32813

Dr. Gary Poock
Operations Research Department
Naval Postgraduate School
Monterey, CA 93940

Dean of Research Administration
Naval Postgraduate School
Monterey, CA 93940

Capt. Horace M. Leavitt
Naval Electronics System Command
Room 554 JP1
Washington, D.C. 20360

Mr. Warren Lewis
Human Engineering Branch
Code 8231
Naval Ocean Systems Center
San Diego, CA 92152

Dr. A.L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380

Stephen Fikas
Naval Ocean Systems Center
San Diego, CA 92152

Mr. Arnold Rubinstein
Naval Material Command
NAVMAT 98T24
Washington, D.C. 20360

Commander
Naval Air Systems Command
Human Factors Programs
NAVAIR 340F
Washington, D.C. 20361

Commander
Naval Air Systems Command
Crew Station Design,
NAVAIR 5313
Washington, D.C. 20361

Mr. Phillip Andrews
Naval Sea Systems Command
NAVSEA 0341
Washington, D.C. 20362

Dr. James Curtin
Naval Sea Systems Command
Personnel & Training Analysis
Office
NAVSEA 074C1
Washington, D.C. 20362

Commander
Naval Electronics Systems Command
Human Factors Engineering Branch
Code 4701
Washington, D.C. 20360

J.B. Blankenheim
Naval Electronics Systems Command
Code 47013
Washington, D.C. 20360

Bureau of Naval Personnel
Special Assistant for Research
Liaison
PERS-OR
Washington, D.C. 20370

Department of the Navy (Continued)

CDR R. Gibson
Bureau of Medicine & Surgery
Aerospace Psychology Branch
Code 513
Washington, D.C. 20372

LCDR Robert Biersner
Naval Medical R&D Command
Code 44
Naval Medical Center
Bethesda, MD 20014

Dr. Arthur Bachrach
Behavioral Sciences Department
Naval Medical Research Institute
Bethesda, MD 20014

Dr. George Moeller
Human Factors Engineering Branch
Submarine Medical Research Lab
Naval Submarine Base
Groton, CT 06340

K. Heninger
Naval Research Lab
Code 7503
Washington, D.C. 20375

Chief
Aerospace Psychology Division
Naval Aerospace Medical Institute
Pensacola, FL 32512

Dr. Fred Muckler
Navy Personnel Research and
Development Center
Manned Systems Design, Code 311
San Diego, CA 92152

CDR P.M. Curran
Human Factors Engineering Division
Naval Air Development Center
Warminster, PA 18974

Dr. H.G. Steubing
Naval Air Development Center
Code 5030
Warminster, PA 18974

LCDR William Moroney
Naval Postgraduate School
Code 55 MP
Monterey, CA 93940

Mr. J. Williams
Department of Environmental
Sciences
U.S. Naval Academy
Annopolis, MD 21402

Dean of the Academic Departments
U.S. Naval Academy
Annapolis, MD 21402

Human Factors Section
Systems Engineering Test
Directorate
U.S. Naval Air Test Center
Patuxent River, MD 20670

Human Factors Engineering Branch
Naval Ship Research and Development
Center, Annapolis Division
Annapolis, MD 21402

Mr. Robert Kahane
Naval Electronics System Command
Code 330
NC Bldg #1
Washington, D.C. 20360

LCDR T. Berghage
Naval Medical Research Institute
Behavioral Sciences Department
Bethesda, MD 20014

Department of the Army

Mr. J. Barber
HQS, Department of the Army
DAPE-PBR
Washington, D.C. 20546

Dr. Joseph Zeidner
Technical Director
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Director, Organizations and
Systems Research Laboratory
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Edgar M. Johnson
Organizations and Systems Research
Laboratory
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Technical Director
U.S. Army Human Engineering Labs
Aberdeen Proving Ground, MD 21005

AFI Field Unit-USAREUR
ATTN: Library
C/O ODCSPER
HQ USAREUR & 7th Army
APO New York 09403

Department of the Air Force

U.S. Air Force Office of Scientific
Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, D.C. 20332

Dr. Donal A. Topmiller
Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HES
Wright-Patterson AFB, OH 45433

Air University Library
Maxwell Air Force Base, AL 36112

Dr. Gordon Eckstrand
AFHRL/ASM
Wright-Patteron AFB, OH 45433

Richard Nelson
Rome Air Development Center
Griffiss AFB
Rome NY 13440

Other Government Agencies

Defense Documentation Center
Cameron Station, Bldg, 5
Alexandria, VA 22314 (12 cys)

Dr. Stephen J. Andriole
Director, Cybernetics Technology Office
Defense Advanced Research Projects Agency
1400 Wilson Blvd
Arlington, VA 22209

Dr. Judith Daly
Cybernetics Technology Office
Defense Advanced Research Projects
Agency
1400 Wilson Blvd
Arlington, VA 22209

Other Government Agencies (Continued)

Dr. Stanley Deutsch
Office of Life Sciences
National Aeronautics and Space
Administration
600 Independence Avenue
Washington, D.C. 20546

Director, IPT
DARPA
1400 Wilson Blvd
Arlington, VA 22209

Other Organizations

Dr. Meredith P. Crawford
Department of Engineering Administration
George Washington University
Suite 805
2101 L Street, N.W.
Washington, D.C. 20037

Dr. Robert G. Pachella
University of Michigan
Department of Psychology
Human Performance Center
330 Packard Road
Ann Arbor, MI 48104

Dr. Gershon Weltman
Perceptronics, Inc.
6271 Variel Avenue
Woodland Hills, CA 91364

Dr. Arthur I. Siegel
Applied Psychological Services, Inc.
404 East Lancaster Street
Wayne, PA 19087

Human Resources Research Office
300 N. Washington Street
Alexandria, VA 22314

Dr. J. A. Swets
Bolt, Beranek & Newman, Inc
50 Moulton Street
Cambridge, MA 02138

Dr. Jesse Orlansky
Institute for Defense Analyses
400 Army-Navy Drive
Arlington, VA 22202

Foreign Addressees

North East London Polytechnic
The Charles Myers Library
Livingstone Road
Stratford
London E15 2LF
UNITED KINGDOM

Director, Human Factors Wing
Defense & Civil Institute of
Environmental Medicine
Post Office Bos 2000
Downsville, Toronto, Ontario
CANADA

Professor Dr. Carl Graft Hoyos
Institute for Psychology
Technical University
8000 Munich
Arcisstr 21
FEDERAL REPUBLIC OF GERMANY

Dr. A.D. Baddeley
Director, Applied Psychology Unit
Medical Research Council
15 Chaucer Road
Cambridge, CB2 2EF
UNITED KINGDOM

Foreign Addressees (Continued)

Dr. J. Baal Schem
Acting Director
Tel - Aviv University
Interdisciplinary Center for Technological
Analysis and Forecasting
Ramat-Aviv, Tel-Aviv
Israel